Distance and Proximity Queries for Distributed Moving Points

Tobias Castanet

▶ **To cite this version:**

Tobias Castanet. Distance and Proximity Queries for Distributed Moving Points. Data Structures and Algorithms [cs.DS]. Université de Bordeaux, 2022. English. NNT : 2022BORD0241 . tel-03875856

HAL Id: tel-03875856

https://theses.hal.science/tel-03875856v1

Submitted on 28 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ECOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

SPÉCIALITÉ : INFORMATIQUE

Par **Tobias CASTANET**

Distance and Proximity Queries for Distributed Moving Points

Sous la direction de : **Nicolas HANUSSE et Olivier BEAUMONT**
Co-encadrant : **Corentin TRAVERS**

Membres du jury :

M. Olivier BEAUMONT	Directeur de recherches Inria	Inria Bordeaux	Examinateur
Mme. Lélia BLIN	Maîtresse de conférences	Université d'Évry	Rapporteure
M. Nicolas HANUSSE	Directeur de recherches CNRS	Université de Bordeaux	Examinateur
M. Nicolas NISSE	Chargé de recherche Inria	Inria Sophia-Antipolis	Rapporteur
M. Éric SANLAVILLE	Professeur	Université du Havre	Président
M. Corentin TRAVERS	Maître de conférences	Université de Bordeaux	Invité

Abstract

With the development of information technology, computing devices become more available and more connected, and are increasingly used in contexts where mobility plays an important role. Video games enable huge amounts of players to interact in some virtual world, and networks of connected vehicles are envisioned to improve road safety. In such distributed contexts, one cannot assume that each entity can share its position with all the participants. This thesis presents different methods to allow moving entities, that we call nodes, and that move in some metric space, to answer to queries related to their distances, with guarantees on the accuracy of the approximations.

First, we propose a synchronous distributed algorithm, that allows two nodes to estimate the distance between them, with a guarantee on the relative error. It is proven that when applied to nodes that follow random movements, the algorithm is optimal in terms of number of exchanged messages.

Then, queries returning, for a given node, the set of nodes that are at a distance smaller than a given distance r are studied. We describe a synchronous distributed algorithm for positions on a line, that ensures each node knows at all time all the nodes that are at a distance r, where r is a fixed value given as input to the algorithm; the answer to the query thus takes $O(1)$ communication rounds. The algorithm needs $O(1)$ communication rounds per node movement, and the local memory cost is of the same order as the worst case largest size of nodes returned by a query.

After that, two algorithms are given for positions in any metric space of constant doubling dimension, where r may vary and is now a parameter of the query. First, a centralized algorithm is given, with a computational cost of $O(\log \Phi)$ operations per movement of a node (where Φ is the ratio between maximal and minimal distance between two nodes), with $O(n)$ memory usage (where n is the number of nodes), and with an answer to the query in $O(\log r + k)$ computations (where k is the size of the set returned by the query). Then it is shown how to adapt that algorithm to the distributed setting, resulting in an algorithm that needs $O(1)$ communication rounds per movement of the nodes, that uses $O(n)$ memory for a node at worst, but $O(n)$ memory in total, and that answers to the query in $O(\log r)$ communication rounds.

Keywords Distributed computing, proximity queries, kinetic data structures, mobile entities, movement models, complexity analysis.

Résumé

Avec le développement des technologies de l'information, les ordinateurs deviennent plus accessibles et mieux connectés, et sont de plus en plus utilisés dans des contextes où la mobilité joue un rôle important. Des jeux vidéo permettent à un grand nombre de joueurs d'interagir dans un même monde virtuel, et des réseaux de véhicules connectés sont envisagés pour améliorer la sécurité routière. Dans de tels contextes distribués, on ne peut partir du principe que les entités peuvent partager leur position avec tous les participants. Cette thèse présente différentes méthodes pour permettre à des entités, que nous appelons nœuds, et qui se déplacent dans certains espaces métriques, de répondre à des requêtes liées à leurs distances, avec des garanties sur la qualité des approximations.

Premièrement, nous proposons un algorithme distribué synchrone, permettant à deux nœuds d'estimer la distance qui les sépare, avec une garantie sur l'erreur relative. Il est démontré qu'appliqué à des nœuds qui suivent des déplacements aléatoires, l'algorithme est optimal en nombre de messages échangés.

Ensuite, des requêtes sont étudiées permettant de retourner, pour un nœud donné, l'ensemble des nœuds situés à une distance inférieure ou égale à une certaine valeur r. Nous décrivons un algorithme distribué synchrone pour des positions sur une ligne, pour une valeur de r fixée et donnée en entrée de l'algorithme. Pour assurer ce résultat, nous proposons de faire en sorte que tout nœud connaisse à tout moment les nœuds qui font partie de la réponse de la requête, ce qui fait qu'un nœud peut y répondre avec $O(1)$ rondes de communication. L'algorithme utilise également $O(1)$ rondes de communication par mouvement des nœuds, et le coût en mémoire locale est du même ordre que la plus grande taille de l'ensemble retourné par une requête.

Ensuite, deux algorithmes sont donnés pour des positions dans n'importe quel espace métrique à dimension doublante constante, et où la valeur r est maintenant un paramètre de la requête qui peut varier à chaque appel. D'abord, un algorithme centralisé est donné, dont le coût en temps est de $O(\log \Phi)$ opérations par mouvement d'un nœud (où Φ est le ratio entre distance maximale et minimale entre deux nœuds), dont le coût en mémoire est de $O(n)$ (où n est le nombre de nœuds), et avec lequel la réponse à la requête s'obtient en $O(\log r + k)$ opérations (où k est la taille de l'ensemble retourné par la requête). Ensuite, il est montré comment adapter cet algorithme dans le contexte distribué, résultant en un algorithme qui nécessite un nombre constant de rondes communications par mouvement des nœuds, avec un coût en mémoire de $O(n)$ pour un nœud, et de $O(n)$ au total, et qui permet de répondre à la requête en $O(\log r)$ rondes de communication.

Mots clés Algorithmique distribuée, requêtes de proximité, structures de données cinétiques, entités mobiles, modèles de mouvement, analyse de complexité.

Résumé substantiel (Introduction in French)

Avec le développement des technologies modernes de communication, les ordinateurs deviennent de plus en plus accessibles et de bon marché, et de plus en plus connectés. De nouvelles formes de divertissement et de nouveaux outils pour permettre à divers objets de la vie courante de se connecter ont émergé.

Avec l'aide de cette évolution des technologies, les jeux vidéos, qui à l'origine ne se jouaient qu'en local sur une seule machine, ont intégré des possibilités de jouer à plusieurs via des réseaux. Les jeux multijoueurs représentent maintenant une part importante de l'industrie du jeu vidéo, avec par exemple plus de 250 millions de joueurs inscrits sur le jeu *Fortnite*. Même des jeux solos comme *Assassin's Creed* ou *Bloodborne* intègrent maintenant des fonctionnalités qui permettent aux joueurs de communiquer ou voir les personnages d'autres joueurs du même jeu.

Dans les jeux en ligne, les joueurs contrôlent des personnages qui se déplacent et interagissent avec l'environnement virtuel. Les ordinateurs des joueurs doivent être en mesure d'estimer précisément où les autres personnages se situent, afin d'afficher à l'écran un état correct du monde virtuel. De plus, la distance et la proximité des éléments du jeu jouent un rôle important, puisque les joueurs sont en général intéressés surtout par l'état de ce qui se trouve à proximité de leur personnage.

Un autre domaine d'application où le mouvement et la proximité jouent un rôle important sont les réseaux ad-hoc de véhicules (VANET). Il s'agit ici d'étudier les véhicules routiers connectés, qui sont capables de communiquer les uns avec les autres pour obtenir les positions d'autres véhicules afin d'améliorer la sécurité et le confort des trajets. Les véhicules d'un VANET sont souvent capables d'utiliser des capteurs afin d'estimer la distance qui les sépare d'autres véhicules à proximité équipés de capteurs similaires. Il est cependant intéressant de proposer en complément des solutions logicielles capables de répondre à des requêtes liées à la proximité. En effet, ces puces ont souvent des portées limitées. De plus, les véhicules d'un VANET sont en général peu contraints en termes de dépenses énergétiques, contrairement à d'autre appareils connectés comme par exemple les drones, ce qui leur permet de se connecter à des éléments d'infrastructure fixes comme le réseau cellulaire pour les aider à transmettre leurs messages.

Ceci nous mène à étudier des applications où le mouvement joue un rôle central.

Dans des réseaux distribués comme mentionnés ci-dessus, même avec la meilleure infrastructure, les puissances de calcul et les capacités de communication sont limitées. Il est impossible de partir du principe que chaque nœud du réseau est capable de partager sa position avec tous les autres nœuds, et l'ajout d'un serveur central pour récupérer et diffuser l'ensemble de ces positions est au mieux coûteux, au pire impossible. Il y a donc un besoin fort d'algorithmes capables de maintenir, sur un ensemble de points, des propriétés en lien avec les distances relatives entre ces points, tout en minimisant les ressources nécessaires en termes de mémoire, de

temps de calcul, et de messages échangés dans le réseau.

Dans cette thèse, nous étudions des méthodes pour être capable de répondre à des requêtes en lien avec les distances entre des points mobiles.

Modèle de base

Soit $\mathcal{V}$ un ensemble de n nœuds, chacun associé à une position dans un espace euclidien. Nous étudions ici deux cadres distincts.

Dans le modèle *centralisé*, les nœuds sont des entités locales à une seule machine. Un algorithme centralisé a pour but de maintenir une structure de données, tout en limitant le temps de calcul nécessaire pour sa maintenance, et en réduisant son impact en mémoire.

Dans le modèle *distribué*, les nœuds sont des ordinateurs distribués sur un réseau. Il s'agit alors pour les algorithmes distribués de maintenir, localement à chaque nœud, une partie seulement de la structure de données ; la *structure de données distribuée* est alors l'union de toutes les connaissances des nœuds. Comme les nœuds ne peuvent envoyer des messages qu'à des nœuds pour lesquels ils retiennent les informations nécessaires pour l'envoi des messages (typiquement, l'adresse), les algorithmes distribués doivent aussi maintenir le graphe de communication, qui définit à qui chaque nœud est capable d'envoyer des messages. On dit qu'un nœud qui peut envoyer des messages à un autre nœud est *connecté* à ce dernier. Les performances des algorithmes distribués sont mesurées principalement en termes de quantité de mémoire locale utilisée, et de coût en communications. Dans le modèle distribué, nous nous focalisons principalement sur des algorithmes *synchrones* : nous supposons que le temps est divisé en *rondes de communications* (éventuellement virtuelles), de telle sorte que lorsqu'un nœud envoie un message pendant une ronde de synchronisation, il est garanti que le destinataire reçoit le message en question avant le début de la prochaine ronde de synchronisation. L'impact du coût de communication d'un algorithme synchrone peut être mesuré par le nombre de rondes de communications nécessaires ou par le nombre total de messages envoyés par les nœuds.

Les positions des nœuds peuvent correspondre soit à leurs positions physiques, réelles (par exemple pour des véhicules d'un VANET qui se déplacent et communiquent), soit aux positions d'entités virtuelles (par exemple les personnages d'un jeu vidéo en ligne). Dans ce deuxième cas, nous supposons sans perte de généralité que chaque nœud est associé à une seule de ces entités.

Dans beaucoup d'applications concrètes, chaque nœud de $\mathcal{V}$ est intéressé par certaines valeurs donnant des informations sur l'état des autres nœuds (par exemple l'énergie restante dans la batterie d'un drone ou le nombre de points de vie restant d'un personnage de jeu vidéo), mais cet intérêt est souvent lié à la distance qui les sépare. Soit cet intérêt décroît de façon continue, c'est-à-dire que la précision requise sur les valeurs maintenues dépend de manière inversement proportionnelle à la distance qui sépare les nœuds, soit cet intérêt dépend d'un paramètre fixé r, de telle sorte que les nœuds ont besoin de connaître les informations concernant les autres nœuds dans un rayon de r d'eux, mais qu'aucune information n'est requise sur les nœuds qui se situent plus loin. Une autre approche similaire à cette seconde vision constitue à considérer que ce paramètre r n'est pas connu à l'avance, et que les nœuds peuvent être temporairement intéressés par l'état des nœuds à une distance r d'eux, où r serait une valeur définie en fonction du contexte, et qui n'est pas prévisible. La valeur de r peut représenter une distance de visibilité, ou tout autre distance au-delà de laquelle on peut juger qu'il est superflu de maintenir des informations.

Nous modélisons l'intérêt des nœuds sous la forme unifiée de *requêtes*. Chaque nœud peut, à tout moment, demander une réponse à une requête en lien avec la distance ou la proximité

des nœuds. Le coût associé à la récupération de la réponse à la requête doit être minimisé, bien que plusieurs de nos algorithmes maintiennent ces réponses en tout temps, de telle sorte que la réponse est parfois immédiate : dans le Chapitre 2, avec notre algorithme, les nœuds maintiennent une estimation de la distance qui les sépare d'autres nœuds, renvoyer cette distance ne demande donc pas de calculs supplémentaires. Il en va de même dans le Chapitre 3, où les nœuds maintiennent dans leurs structures de données locales les réponses aux requêtes. Dans le Chapitre 4 en revanche, une requête nécessite de faire des calculs pour récupérer la réponse à renvoyer.

Un premier intérêt est celui de trouver des algorithmes qui permettent d'estimer efficacement les distances qui séparent les nœuds. Outre l'intérêt intrinsèque que certaines applications pourraient avoir pour ces estimations de distance, celles-ci pourraient également être utiles pour d'autres méthodes de gestion d'intérêt comme ci-dessus.

La principale difficulté liée à ces problèmes de distance et de proximité est la mobilité des nœuds. La plupart des résultats de la littérature qui traitent de problématiques de proximité, considèrent des structures *statiques*, c'est à dire que l'ensemble de nœuds $\mathcal{V}$ et la position de chaque nœud est fixe. Certains résultats s'intéressent au cas plus difficile de structures *dynamiques* : bien que les positions des nœuds restent statiques, ces structures doivent permettre d'ajouter et d'enlever des nœuds. Nous traitons ici un cas encore plus difficile, celui des *structures de données cinétiques* (KDS), dans lesquelles l'ensemble $\mathcal{V}$ des nœuds reste fixe, mais avec des positions qui changent avec le temps. Le cas cinétique peut être résolu avec des structures dynamiques, dans lesquelles les nœuds sont enlevés puis rajoutés à chaque déplacement, mais cette solution mène en général à des mauvaises performances.

Pour les KDS, il existe plusieurs modèles de mouvements. Nous considérons pour l'ensemble de la thèse le modèle Black-Box : nous supposons que des instants à intervalles réguliers peuvent être identifiés, que nous appelons *pas de temps*, de telle sorte que les nœuds ne peuvent que se déplacer à chaque pas de temps, et qu'ils restent immobiles entre deux pas de temps. Une autre distinction peut être faite concernant le nombre de nœuds qui peuvent bouger à chaque pas de temps : dans le modèle de *mobilité réduite*, un seul nœud peut se déplacer, tandis que dans le modèle de *pleine mobilité*, il n'y a pas de restrictions quand au nombre de nœuds qui peuvent changer de position à chaque pas de temps.

Dans le modèle distribué, et en particulier quand le nombre de nœuds est très grand, il faut éviter une sur-utilisation des ressources à disposition, et en particulier, l'utilisation de bande passante doit être limitée. Nous identifions deux approches complémentaires pour cela. Rappelons qu'un nœud qui peut envoyer des messages à un autre nœud est dit connecté à ce dernier.

- Premièrement, on peut essayer de faire en sorte qu'un nœud qui est connecté à un autre nœud envoie le moins de messages possibles.

- On peut également essayer de faire en sorte que chaque nœud soit connecté à aussi peu d'autres nœuds que possible. En effet, deux nœuds connectés risquent échanger des messages pour maintenir les estimations de leur état. Réduire le nombre de connexions permet donc de réduire le nombre de messages échangés, en plus de réduire le coût en mémoire locale.

Dans cette thèse, nous présentons trois algorithmes, ciblant deux objectifs différents.

Estimation de distance

Un premier algorithme est proposé qui utilise la première approche : dans le modèle distribué, on considère deux nœuds connectés et l'algorithme cherche à minimiser les nombre d'échanges

de messages. Nous cherchons à faire en sorte qu'un nœud maintienne une estimation d_{est} de la distance qui le sépare de l'autre nœud. L'aspect continuellement décroissant de l'intérêt mutuel que se portent les nœuds est modélisé par une erreur *relative*. En notant d_{act} la distance réelle qui sépare deux nœuds, et ε l'erreur relative maximale autorisée, nous voulons garantir qu'à tout instant :

$$(1-\varepsilon)d_{est} < d_{act} < (1+\varepsilon)d_{est}.$$

Une méthode couramment utilisée dans les jeux vidéos en ligne pour garantir une erreur *absolue* maximale sur les estimations des *positions*, est appelée *Dead-reckoning*. Soit p_u la position réelle d'un nœud u, et $\widetilde{p_u}$ cette position telle qu'estimée par un autre nœud v. On suppose qu'il existe un algorithme d'estimation déterministe, qui permet à v d'obtenir $\widetilde{p_u}$ à partir du dernier message reçu de u (typiquement, il s'agit d'extrapoler en utilisant le vitesse de u). u peut alors appliquer le même algorithme, et connaître également la valeur de $\widetilde{p_u}$. Il calcule ensuite $d(p_u, \widetilde{p_u})$, la distance entre p_u et $\widetilde{p_u}$, et envoie un nouveau message à v dès que l'erreur absolue de l'estimation de v dépasse un maximum fixé, et seulement à ce moment là. Le Dead-reckoning permet de réduire le nombre de messages, mais aussi de minimiser les effets négatifs des latences.

Nous nous inspirons du Dead-reckoning pour proposer un algorithme distribué synchrone, appelé *Local Change*, qui lui permet d'estimer les *distances* entre les nœuds. Dans Local Change, u envoie un message à v dès que la distance entre p_u et $\widetilde{p_u}$ devient trop grande, plus précisément dès que $d(p_u, \widetilde{p_u}) \geq d_{est} \times \varepsilon/2$; v répond alors avec une message réciproque. Nous avons montré que cet algorithme satisfait la garantie sur l'erreur relative, et que le nombre de messages échangés est le même, à une constante multiplicative près, que pour un algorithme idéal (c'est-à-dire qui échange des messages uniquement quand la condition sur l'erreur relative n'est plus satisfaite), sur plusieurs modèles de déplacements aléatoires.

Requêtes de proximité

Deux autres algorithmes sont ensuite proposés, ciblant la seconde approche : des structures de données sont étudiées en centralisé et en distribué, telles que chaque nœud n'a besoin de maintenir des informations que sur un nombre réduit d'autres nœuds. Les nœuds sont supposés ne pas se déplacer de plus de d_{mv} unités de distance par pas de temps.

La requête à laquelle les nœuds doivent répondre est la suivante :

Definition ($\textsc{CloseNodes}_u(r)$)**.** *Étant donné un nœud $u \in \mathcal{V}$ et une distance $r \in \mathbb{R}$, retourner tous les nœuds $v \in \mathcal{V}$ tels que $d(u, v) \leq r$.*

Un premier algorithme est proposé pour des réseaux distribués synchrones, et avec des positions sur une ligne (en 1D), pour des valeurs de r fixées à l'avance, c'est-à-dire données en entrée de l'algorithme. Le réponse à la requête CloseNodes est maintenue en tout instant par chaque nœud, de telle sorte que la réponse s'obtient en temps constant. Les déplacements des nœuds sont dans le modèle de Black-Box en pleine mobilité, et l'algorithme a besoin d'un nombre constant de rondes de communication par pas de temps. Le coût en mémoire locale est de $O(b_{max}(r))$ par nœud, où $b_{max}(r)$ représente le nombre maximal de nœuds qui peuvent se trouver à distance r d'un nœud, ce qui est asymptotiquement optimal.

Enfin, des algorithmes sont présentés se basant sur la notion de *navigating net*. Un navigating net est une structure hiérarchique définie de telle sorte que tous les nœuds se trouvent dans le niveau le plus bas de la hiérarchie, et telle qu'à chaque niveau, les nœuds sont filtrés en fonction de la distance qui les sépare, jusqu'à ce qu'il n'en reste qu'un seul au niveau $O(\log \Phi)$, où $\Phi = \frac{d_{max}}{d_{min}}$, avec d_{max} et d_{min} respectivement la distance maximale et la distance minimale

autorisée entre deux nœuds. Une généralisation de la notion de navigating net est donnée, avec les conditions sur les paramètres de la structure nécessaires pour obtenir certaines des propriétés utilisées dans d'autres travaux, avant de définir une version plus stricte que nous appelons *constrained navigating nets*, qui s'inspire d'un autre travail afin d'obtenir des propriétés intéressantes.

Un algorithme centralisé, $\mathcal{A}_{cnnptr}$, basé sur les constrained navigating nets est donné pour la requête, pour des positions dans n'importe quel espace métrique à dimension doublante constante. Cette fois-ci, la valeur de r n'est pas fixée, et est donnée comme entrée de la requête. Chaque nœud n'a besoin de maintenir d'informations que sur $O(\log \Phi)$ autres nœuds, et le coût en mémoire total est en $O(n)$, tout comme pour le DefSpanner, structure proposée dans d'autres travaux. Dans le modèle de mobilité réduite en Black-Box, le coût en temps de l'algorithme est de $O(\log \Phi)$ opérations par pas de temps, tout comme les DefSpanners. Cet algorithme présente cependant des propriétés intéressantes, qui nous permettent d'en proposer une version distribuée, notée $\mathcal{A}_{cnndist}$, qui permet de maintenir des constrained navigating nets dans des réseaux distribués synchrones, dans le modèle de mobilité réduite en Black-Box, en utilisant un nombre constant de rondes de communication par pas de temps. Le coût en mémoire pour un nœud peut aller jusqu'à $O(n)$, mais le coût total sur l'ensemble des nœuds reste $O(n)$. Cet algorithme est le premier à maintenir des navigating nets dans des réseaux synchrones en Black-Box.

Contents

Abstract **2**

Résumé substanciel (Introduction in French) **5**

1 Introduction **15**
1.1 Context . . . 15
1.1.1 Base Model and Studied Problems . . . 16
1.2 Distributed Systems . . . 18
1.3 Distributed Virtual Environments . . . 21
1.3.1 Challenges . . . 21
1.3.2 Dead-Reckoning . . . 25
1.3.3 Fully distributed DVEs . . . 26
1.4 Peer-to-peer Overlays . . . 26
1.4.1 Unstructured Overlays . . . 27
1.4.2 Structured Overlays . . . 27
1.4.3 DVE Overlays . . . 29
1.5 Data Structures for Proximity and Distance Queries . . . 31
1.5.1 Proximity Queries . . . 31
1.5.2 Geometrical Data Structures . . . 32
1.5.3 Doubling Dimension . . . 38
1.6 Kinetic setting and Related Definitions . . . 38
1.6.1 Mobility Models . . . 38
1.6.2 Parameters . . . 39
1.7 Kinetic Data Structures . . . 40
1.7.1 Flight Plan Model . . . 40
1.7.2 Black-Box Model . . . 44
1.7.3 Tree Structures . . . 46
1.7.4 Spanners . . . 46
1.7.5 Navigating Nets . . . 48
1.7.6 Conclusion on Centralized KDSs . . . 48
1.7.7 The Case of Distributed KDSs . . . 49
1.8 Contribution . . . 51
1.8.1 Distance Estimation . . . 51
1.8.2 Kinetic Data Structures for Proximity Queries . . . 52

2 Distance Estimation **55**
2.1 Introduction . . . 55
2.1.1 Related Work . . . 55
2.1.2 Limitations and Hypotheses . . . 56

2.1.3 Contribution 56
2.2 Algorithm and Movement Models 58
2.2.1 Model 58
2.2.2 Algorithm 59
2.3 Random Walk Movement 59
2.3.1 1D Case, Only One of the Nodes Moves 61
2.3.2 1D Case, Both Nodes Move 66
2.3.3 2D and 3D Case 67
2.4 Continuous Movement, Discrete Time 70
2.4.1 1D Case 70
2.4.2 2D Case 71
2.4.3 3D Case 75
2.5 Experiments 76
2.5.1 Synthetic Traces 76
2.5.2 Actual Traces 78
2.6 Conclusion and Future Work 81

3 Distributed Kinetic Data Structure for Proximity Queries in One Dimension: Unit Disc Graphs 85
3.1 Introduction 85
3.1.1 Model 86
3.1.2 Objectives 87
3.2 The Connection Graph 88
3.3 Data Structure 90
3.3.1 Local Variables 90
3.3.2 Certificates 91
3.4 Updating the Structure 93
3.5 Validity 96
3.6 Performance Analysis 101
3.7 Conclusion and Future Work 104
3.7.1 Possible Extensions 104

4 Kinetic Data Structures for Proximity Queries in Higher Dimension: Navigating Nets 109
4.1 Introduction 109
4.1.1 Model 109
4.1.2 Contribution 111
4.1.3 Other Related Structures 112
4.2 Navigating Nets 114
4.2.1 A Tree of Nodes 114
4.2.2 Enriching the Structure by Adding Neighbors 117
4.2.3 Ancestors 118
4.3 Centralized Navigating Nets 121
4.3.1 Related Work 121
4.3.2 Constrained Navigating Nets 124
4.3.3 Maintain Constrained Navigating Nets When Nodes Move 130
4.3.4 Validity of the Algorithm in the Low Mobility Setting 139
4.3.5 Performance Analysis in the Low Mobility Setting 150
4.3.6 Improving $\mathcal{A}_{cnn}$ for a better update time 157

4.4 Distributed Navigating Nets 162
4.4.1 Related Work 162
4.4.2 Adapting D-Spanners to the Black-Box Model 163
4.4.3 Adapting $\mathcal{A}_{cnn}$ to the Distributed Setting: $\mathcal{A}_{cnndist}$ 166
4.5 Conclusion and Possible Extensions 171
4.5.1 Adapting $\mathcal{A}_{cnn}$ to the High Mobility Setting 172

5 Conclusion **179**

A Glossary **197**

B Notations **199**

C Figures for Continuous Movement **201**

D Full description of $\mathcal{A}_{cnnptr}$ **205**

Chapter 1

Introduction

1.1 Context

With the modern development of information technology, computing devices become more powerful, more available, and more connected. New forms of entertainment and new ways to interconnect people or objects have arisen.

With the help of this technological evolution, video games, which were at first played only on single local devices, have been increasingly opened up to online gameplay. Niche at first, online games now represent a huge part of the entertainment industry. Since the mid-two-thousands and the launch of *World of Warcraft*, online games with huge amounts of simultaneous players have gained increasing popularity. Nowadays, with over 250 million registered players, the game *Fortnite* has become such a phenomenon that its social impact on the younger generation is the subject of research [34]. Even solo games like *Assassin's Creed* or *Bloodborne* now feature online elements to enhance the game experience.

In online games, players control avatars that move and interact with the virtual environment. The computers of the players must be able to estimate where other players are located, in order to correctly display the virtual world; distance and proximity play an important role, as players are generally more interested in the state of other players when they are nearby.

Another application domain where movement and proximity play an important role is Vehicular Ad-hoc Networks (VANETs). VANETs study connected vehicles, that may communicate with each other in order to know the locations of other vehicles, so as to improve safety and comfort. While VANET nodes can typically use hardware devices to compute distances to other nodes or to look for close-by vehicles [9, 113], having software solutions for these problems can be useful, since VANET vehicles have less power consumption requirements than other moving devices like drones [121, 133], and can thus connect to infrastructure elements and cellular networks to help them route their messages [148].

This leads us to study applications where movement plays an important, or even central role.

In such distributed contexts, even with the best Internet infrastructure, computing power and communication capabilities will always remain limited. One cannot assume that each node can share its position with all the participants, and having a central server that gathers information about all participants is expensive if not outright impossible. There is therefore a strong need for algorithms that enable to maintain a number of valid properties on the knowledge of (estimated) distances, while leading to a minimum number of message exchanges and computations, and where all nodes play a symmetric role. In this thesis, we investigate methods to be able to respond to positional queries on sets of moving points.

1.1.1 Base Model and Studied Problems

The goal of this thesis is to *maintain data structures on moving entities*, so as to be able to efficiently answer to queries about their positions. We will call these entities *nodes*.

Let $\mathcal{V}$ be a set of n nodes, each associated with a position in an Euclidean space. Depending on the studied problem, the nodes may be local entities on a single computer, or distributed computing units in a network (see Section 1.2). The positions may correspond to the physical positions of the nodes (like drones or cars that move and communicate) or to a position in a virtual world (like players in online games).

A lot of previous results handle *static* entities [69, 93], that is, both the set of nodes $\mathcal{V}$ and the positions of these nodes are given once and do not change. Some results deal with the more difficult *dynamic* setting [115], where the positions of the nodes also do not change, but nodes may be added to or removed from $\mathcal{V}$. In the even more challenging *kinetic* setting, studied in this thesis, $\mathcal{V}$ remains the same, but the positions of the nodes may change, according to some specified constraints called movement model (each algorithm is tailored for a different movement model). This thesis targets the kinetic setting.

We focus on two additional settings.

- In the *centralized setting*, an algorithm maintains a data structure (that is maintained and entirely known by the same computer that runs the algorithm), and aims at bounding the update time needed to handle the movement of the nodes, as well as the amount of used memory. While not as dense as results in the static and dynamic setting, quite a few articles have been published on centralized kinetic data structures [14, 61, 75]. These will be discussed in Section 1.7.

- In the *distributed setting*, the network is assumed to be *synchronous* (see Section1.2). The nodes are computing units, and the data structure is distributed among all the nodes; we call *local data structure* the part that is known by one node. The main difficulty of the distributed setting is that *each node knows only its own true position*, and communications are needed to know the position of another node. Thus, *we cannot assume that a node knows, in real time, the exact positions of all the nodes in the network*. At each movement of one or several nodes (depending on the movement model), a distributed algorithm should make sure that the local data structures and potentially the connection graph itself (see Definition 1.3) are updated, by having nodes exchange messages with other nodes they are aware of. The aim is to have as small a number of communication rounds as possible, while minimizing the size of the local data structures. While a lot of results have been published on centralized kinetic data structures, only very few are available on distributed kinetic data structures (see Section 1.7.7).

In this work, we follow the widely used *random-access machine* (RAM) model, describing how a computer operates its local computations, so as to compare the performance of different algorithms. In the RAM model, each memory entry is stored in a register, and each register has an address encoded in $\log N$ bits where N is the number of registers. We suppose that accesses to memory take exactly one unit of time computation; simple operations (additions, subtractions, multiplications, divisions, and comparisons) take also one time unit, while loops are comprised of several simple operations, and thus take longer time [5, 136]. We also suppose that the positions have a limited accuracy a, so that each position is in $\{0; a; 2a; \cdots; Ua\}^d$, with U an integer representing the range of possible positions for one coordinate, and with d the number of dimensions. It is thus possible to store one coordinate using $\log U$ bits. In the distributed setting, we make an additional supposition: nodes have identifiers that can be stored with $\log n$

bits. Finally, we assume that registers have a size in bits higher than $\max(\log N, \log U, \log n)$, so that one register is enough to store either a coordinate, an address, or a node identifier.

The number of registers needed to store a data structure and/or to run an algorithm gives the *memory cost* of that data structure and/or algorithm. Memory and time complexity usually depends on n and on characteristics of the positions (such as the minimal or maximal distance between the nodes), but also on d (as a position takes d registers to be stored). In all our results, however, we consider that d is constant, so that it is usually omitted[1].

We assume that each node is interested in some values about the state of the other nodes, but that this interest is related in some way to their distance. Either this interest is *continually decreasing*, like in Chapter 2 where precision is inversely proportional to the distance between nodes, or this interest is related to a *fixed parameter r*, so that a node is interested only in nodes that are at a distance smaller than r from it, like in Chapter 3. Another approach is to consider that this *parameter r is not known in advance*, and that nodes may be temporary interested in all nodes at some distance r from them (for any value of r), like in Chapter 4.

Before trying to determine which nodes are close and which nodes are far away, it is needed in the distributed setting to find ways for nodes to compute the distance between them, since, as mentioned earlier, nodes know only their own true position. To the best of our knowledge, no algorithm has been proposed previously for this specific problem.

Distance estimation In Chapter 2, an algorithm, based on a widely used technique called dead-reckoning (see Section 1.3.2), is proposed for two distributed nodes to estimate the distance that separates them, while minimizing the number of exchanged messages. The continually decreasing aspect of their mutual interest is represented by a *relative error*, that is relative to the distance between the nodes. We denote by $d(u, v)$ the distance between two nodes u and v. This distance estimation with relative error is modeled by a query (see also Section 1.5.1):

Definition 1.1 (ε-DISTANCE$_u(v)$ query)**.** *A node u that calls this query returns d_{est} such that $(1-\varepsilon)d_{est} < d(u, v) < (1+\varepsilon)d_{est}$.*

A node u that knows its own true position and runs our algorithm, is able to answer to the ε-DISTANCE$_u(v)$ query for any node v that runs the same algorithm, and with a fixed value of ε (ε is an entry of the algorithm). The algorithm is shown, for several random movement models, to be optimal in terms of exchanged messages, up to a constant factor.

Kinetic Data Structures for Proximity Queries Chapter 3 and Chapter 4 target the following query:

Definition 1.2 (CLOSENODES$_u(r)$)**.** *Given a node $u \in \mathcal{V}$ and a distance $r \in \mathbb{R}$, return all nodes $v \in \mathcal{V}$ such that $d(u, v) \leq r$.*

In Chapter 3, a distributed algorithm is proposed for one-dimensional positions, with fixed r. The algorithm ensures that each node is connected to all nodes that are at distance r from it and only few other nodes, so that each node $u \in \mathcal{V}$ has at all time access to the output of the CLOSENODES$_u(r)$ query. In Chapter 4, both a centralized structure and a distributed structure are maintained, that allow each node u to answer to CLOSENODES$_u(r)$, where r may be different for each query; the positions are from any metric space that has a constant doubling dimension.

[1]As we give the memory cost in terms of registers and not bits, as d is supposed to be constant, and as registers have a size in bits higher than $\max(\log N, \log U, \log n)$, the bit cost of our algorithms may be higher by a factor $d \max(\log N, \log U, \log n)$ than the memory costs we give for our algorithms.

The main difficulty related to these problems is the mobility of the nodes. Most of the results about proximity deal with *static* structures: the set of nodes and their positions remain the same. Some results propose *dynamic* structures: while the nodes are still static, it is allowed to add and remove nodes to the structure. However it is much more difficult to efficiently handle mobility, and to design a *kinetic* structure, that copes with movement. These structures are usually called Kinetic Data Structures (KDS).

In the next sections, we will present different concepts and fields of research that will gradually bring us towards our contribution, while giving context for the studied problems, and presenting related work. First, we will talk about the general concept of Distributed Systems (Section 1.2), and the special case of Distributed Virtual Environments (Section 1.3). This will lead us to some problems related to Peer to Peer networks, in particular Distributed Hash Tables (Section 1.4). Before moving on, we will present proximity queries related to our problem (Section 1.5). In our context, Distributed Hash Tables have some drawbacks, which will lead us to another concept, Kinetic Data Structures (Section 1.7).

1.2 Distributed Systems

Our main focus in this work is the design of algorithms for distributed systems. Distributed systems are widely studied both in theory [35] and in practice [141], and represent their own field of study in Computer Science. Most of what will be said in this section can thus be found in introductory books [100, 116, 128, 142].

A distributed system is a system where several distinct computing units cooperate towards a shared goal. We will call those computers the *nodes* of the distributed system. A node can be several things, for example the desktop computer or mobile phone of an internet user, or a computing unit embedded in a plane or a car.

Nodes have their own processing power, memory, and communication capacity. We will suppose, as it is often done in distributed systems research, that the processing power of the nodes is unlimited; in other words, local computations are considered instantaneous with respect to the communications. However, the views the nodes have of the system may be different and incomplete, and to be able to cooperate, they will need to communicate. These communications can be abstracted into messages: when a node wants to communicate with another node, it will send a message. These messages may not match exactly the reality, as they may take different forms. On the Internet, for example, they may be divided in several smaller sized packets, and some applications may have access only to so-called beeps [58] to communicate. Constraints on message sizes may thus be different depending on the context.

However, even when knowing another node, it may not be straightforward to send a message to it, when there is no direct link. Usually, nodes can send messages only to a limited subset of other nodes, creating the need of transferring information from node to node, by appropriately routing the messages. For instance in Wireless Sensor Networks (WSN), the range of communication of devices may be very short, so that each node is limited as to which nodes it can relay messages, and the battery power of the nodes is usually so small that optimality of routing mechanisms is important [134]. Thus, routing in WSN is still a field of active research [11].

However, some infrastructures like the Internet have become sufficiently powerful so that routing can be considered a separate problem. Some models (like ours later in this document) suppose that all nodes are connected, and that it is sufficient for a node to know an address, or identifier (shortened to ID) of another node in order to be able to send messages to it, implicitly supposing that the routing mechanisms are already implemented in the underlying network. We

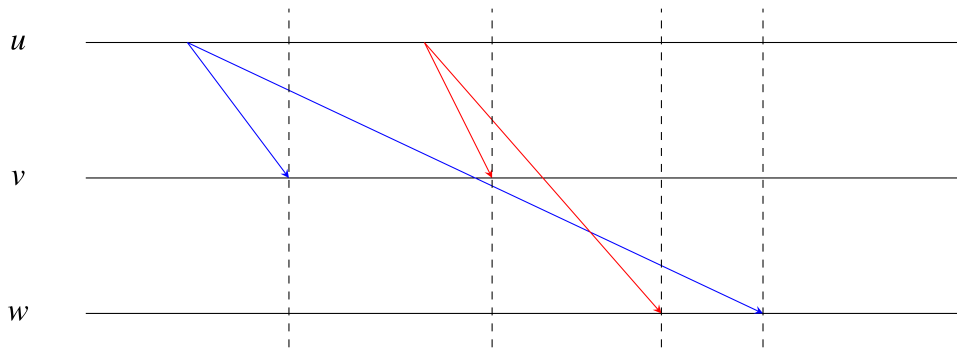


Figure 1.1 – Asynchronous message passing

suppose that the identifiers of the nodes can be encoded in $\Theta(\log n)$ bits.

Also, consistency has to be maintained [86, 87, 110]. Several nodes may keep track of the state of a common variable. Those nodes may have a different view of this variable, that is, the local values of the variable may be different. As this variable is supposed to have only one value, we have to ensure that those views are not too widely different, or not different at all from one node to another. Several models of consistency may be defined, depending on how strict a consistency is required. It will be seen later that consistency is not central to this work, in particular since Dead-reckoning (see Section 1.3.2) is a tool that can easily reduce inconsistencies when the considered variables are positions of moving points.

Another aspect of distributed systems is synchronization. Synchronization may refer to several aspects related to assumptions about time, like clock synchronization, and message passing synchronization. While physical clock synchronization may be important in settings where there is a need of a global reference for nodes to rely on in terms of time, like in WSN [90], our algorithms don't rely on a global clock, but need the messages to be synchronized..

In an arbitrary asynchronous network, because of differences in message transition delays, two events may then be seen by two nodes in a different order, causing consistency problems. On Figure 1.1, an example of message passing in an asynchronous network is given with three nodes; the horizontal lines represent the passage of time, the plain colored lines represent message transitions, and the dashed vertical lines represent the time of arrival of the messages. We can see that u sends two messages at different instants, and that v and w do not receive them in the same order. This can cause problems. For example, let us say that all nodes maintain the value of an integer a that should be the same for everyone, and that at the beginning, v and w agree that $a = 2$. If the first message sent by u says to put a to 0 ($a \leftarrow 0$), and the second message says that a has to be incremented by 1 ($a \leftarrow a + 1$), then v will think that $a = 1$ while w will think that $a = 0$. This shows that in an asynchronous network, specific care has to be taken in order to maintain consistency.

In synchronous networks, additional assumptions are made. The time is divided in *communication rounds*[2], and it is assumed that if a message is sent during a communication round, then the receiver gets the message before the next round starts. Even if no order can be guaranteed for messages sent in the same round, we can see on Figure 1.2 that the general order of the messages is more predictable. While the communication rounds are usually presented (and thought of) as taking place simultaneously on each node like a globally synchronous clock, this does not have to be the case. We only assume that clock deviation and message transmission delays are small enough so that communication rounds can be logically defined.

While this is not ideal when targeting real applications (because real networks are usually

[2]called *pulses* in [142]

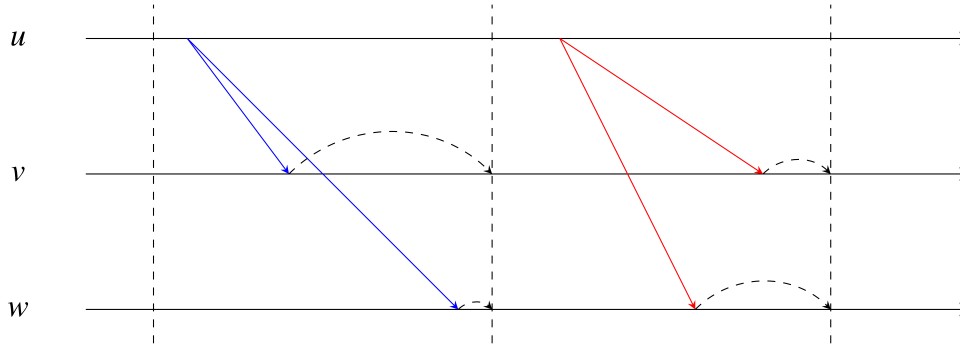


Figure 1.2 – Synchronous message passing

asynchronous), supposing that a system is synchronous makes it much simpler to design an algorithm and analyse its performances. Moreover, there exist synchronizer algorithms, that allow nodes to execute synchronous algorithms even on asynchronous systems [12, 73], so that the interest in synchronous algorithms is high.

The distributed algorithms that we will present in the next chapters target synchronous networks.

Even when ignoring routing and synchronization problems, distributed systems add several difficulties in contrast with centralized systems, because sending messages comes at a cost. Among other problems, bandwidth may limit the number of incoming or outgoing messages for a node, latency may lead a message to arrive to its recipient with a delay, or packet loss may force nodes to cope with the fact that some messages may never arrive to their destination. In our work we focus mainly on problems related to bandwidth.

In the absence of optimization of bandwidth usage, some limitations may occur, usually when increasing the size of the system, depending on the application. For an online game with few players, centralizing information on a single server or letting all players be connected to all others and send all their updates to everyone may work; however, the cost of a powerful server or the bandwidth usage of the same strategies for games with large amounts of players, like Massively Multiplayer Online Games (MMOG), may be prohibitive. Similarly, a small number of moving embedded systems with no battery limit may not need to optimize their amount of communications, but drones usually have limited battery lifetime, and even more powerful autonomous vehicles could be overwhelmed if they have too many messages to handle. This is why improving bandwidth usage is one of the main aspects when searching to improve the scalability of distributed systems.

Overlay Network Also, in a Distributed System, all nodes may not be connected to each other. We will call *connection graph* the graph representing the current state of the connections.

Definition 1.3. *The connection graph $G = (\mathcal{V}, E)$ is a directed graph, with $\mathcal{V}$ the set of nodes, and with $E \subseteq \mathcal{V}^2$ the set of arcs. Arcs are couples from $\mathcal{V}$, and represent the connection status of two nodes: $(u, v) \in E$ means that u is able to send a message to v*

Note that a couple is ordered: if $(u, v) \in E$, the contrary may be false.

Definition 1.4. *If $(u, v) \in E$, we say that v is* adjacent *to u in the connection graph.*

Some models may take G as an entry of the problem, in which case G represents the limitations of the network, and the outgoing arcs of a node u represent the set of nodes to which

u is physically able to communicate. This is the case for example in Mobile Ad hoc Networks (MANETs), where nodes can only communicate wirelessly and with a limited range.

However, in this work, G usually represents an output, that is, an algorithm may reshape G: in this model, nodes may voluntarily drop unneeded connections, and tell other nodes to connect to some specific node by sending its identifier. Each node is physically able to communicate with any other node (for example through the Internet), but may be limited because of a lack of knowledge: a node may send a message to another node only if it knows the ID of the target node.

Let us now get closer to our problematic with a special case of distributed systems.

1.3 Distributed Virtual Environments

The term *Distributed Virtual Environment* (DVE) refers to a specific type of distributed systems where geographically distant users, or players, participate in a common real-time simulation of a virtual world.

Originally, DVEs were studied for military training simulations; however, the main examples of DVEs are now online games. Players control characters[3] that interact with each other, and may modify the shared environment.

One of the differences between a DVE and a classical distributed system like a database, is that the states of the objects in the virtual environment evolve even without changes issued by the players [102] since objects must respect the physics of the game, and non-player characters go about their programmed activities (non-player characters are similar to player characters but follow predetermined behaviors, and are not controlled by human players [143]). In addition, the amount of updates per time unit is generally high, as players are expected to interact a lot with the environment. Those two facts lead to one of the main characteristics of DVEs: changes in the state of the objects are expected to happen all the time, relentlessly.

To keep a vocabulary that is consistent with the other sections, we will use the term *node* instead of player or user. Also, while in some games, players may have control of several characters, our assumptions from Section 1.1.1 mean that we suppose, without loss of generality, that each node is responsible for only one character, and we may use *node* to refer to both.

Also, we will call *update* a message containing information that will change the state of the virtual world. An update is either a request issued by a node[4], for example when the player pushes a button to use an object or to hit an enemy, or a message sent to indicate the change of a continuous value, for example, to indicate the new position of a constantly moving character.

In a DVE, as communications are supposed to go through the Internet, the geographical position of a node in the network is generally irrelevant. Thus, for simplicity, we may refer to the position of a character in the virtual world as the position of the node.

As explained previously, in our work, our main interest relies on the position of nodes, and less on the other values that may make up the state of the various objects, such as remaining life points or amount of carried money.

1.3.1 Challenges

DVE nodes need to communicate in order for everyone to have a sufficient knowledge of the state of the virtual world to be able to display it, and to allow the users to satisfyingly interact

[3] some authors prefer to call them *avatars*

[4] these requests are often called *inputs*, especially in the game-developer communities

with it while making sure the rules of the game are enforced. Two central aspects that need to be optimized in a DVE are consistency and responsiveness.

Consistency As we have seen in Section 1.2, consistency refers to the degree at which each node may have a different view of the system. Inconsistencies arise when two users see different versions of the virtual world. Some small inconsistencies may not be problematic as long as they are not perceptible. However, nowadays, players often communicate with each others using voice communication programs, making inconsistencies more noticeable.

In [102] a definition of consistency is proposed for DVEs: an application respects consistency at an instant t, if any two nodes that received all updates supposed to take effect before t have exactly the same view of the virtual environment. While this definition is satisfactory for the long term, it does not take into account the inconsistencies that may happen temporarily when two nodes receive some updates at different instants. This is why [102] adds the definition of *short-term inconsistencies*, that happen when a node receives an update and applies it before another node receives the same update.

Because of the continuous and evolving aspect of DVEs, inconsistencies may appear differently than in traditional Distributed Systems. We have seen in Section 1.2 an example of inconsistency with a variable a that is seen with a different value by two nodes. In a discrete system, the severity of this situation depends on the number of reads the variable will be subject to, but in a continuous system like a DVE, this may be problematic for the whole time the inconsistency is not corrected (for example, if a represents the position of an object in the virtual world, then the longer two players see the object in a different position, the more this inconsistency will have a bad impact on the game experience). Even worse, the inconsistency may become more and more problematic as time passes: for example, if a represents something like the acceleration of a vehicle, then the difference in perceived position will grow with time. This is one of the reasons that could motivate the use of a consistency measurement called *Time-Space Inconsistency* [147], that takes into account the duration of an inconsistency. This measurement may be averaged like in [91], which could be used to compare different DVEs in terms of consistency.

Responsiveness Responsiveness is related to the real time nature of DVEs. The term refers to the interval between when a user executes an action (for example, pushing the button to shift gears) and when the effects of this action is perceived by the player (the car actually shifting gears). Responsiveness is unsatisfactory when this time delay is noticeable [103].

While responsiveness is much more straightforward to define than consistency, it is quite difficult to determine which level of responsiveness is acceptable. Some games, like turn-based strategy games can handle very low responsiveness without impacting player satisfaction, but others may suffer greatly from it [45].

These two parameters of consistency and responsiveness are usually contradictory: implementing methods to get better consistency usually comes at the cost of responsiveness [52, 129]. Typically, in [102], a method called local-lag is proposed in order to reduce the number of short-term inconsistencies: a delay is added to every request issued by users (for example if a player pushes the button to shift gears, then there will systematically be a slight delay of for example 50ms before it takes effect). In other words, every node will consider locally a state that is slightly from the past, and delay local player inputs accordingly. This augments the probability that every node has received the request when it has to be applied, reducing short-term inconsistencies, but obviously worsening the responsiveness[5].

[5]In fighting games, that are particularly sensible to responsiveness, this type of solution is often frowned upon

When designing a DVE, one has to consider consistency and responsiveness with regards to the targeted network, and thus take into consideration latency and bandwidth limitations.

Latency One difficulty is related to latency, representing the fact that networks are not ideal, and that message transmission is not instantaneous. Latency measures the time between when a message is sent and when the message is received by its recipient. This time has to be taken in consideration in most DVEs. For example, even in recent games, that usually implement some techniques to cope with the problem, users seem to accept to play on networks with latencies up to around 100ms, after which their satisfaction decreases [47, 70, 89].

While differences in latency between nodes are at the root of consistency problems in discrete systems (as it may result in events that are seen in a different order by each node), continuous applications like DVEs are also more sensible to high values of latency. As it takes longer for other nodes to receive updates, this means that nodes will have obsolete views of the state of some objects. This is why latency is at the heart of the trade-off between consistency and responsiveness.

While academic research has been conducted on this trade-off (for example in [13]), most of the ideas to cope with latencies in DVEs come from game developers and are often very application specific [31, 46]. However, a technique called Dead-reckoning, described in Section 1.3.2, is an excellent tool that can address latency issues both in terms of consistency and responsiveness.

Because of this application-specific aspect of latency, in the theoretical context studied in this thesis, we prefer to consider latency as a tangent problem, and our focus is mainly on bandwidth optimization.

Bandwidth Usage Some networks have limited bandwidth capabilities, and may limit the quantity of information that can be transmitted per time unit. In practice, if the network is reaching its maximal bandwidth and is saturated, it results for the end program in a higher latency, as messages need to be queued before being sent. Reducing bandwidth usage of algorithms thus has a positive effect on the latency.

There are mainly two factors that have an impact on bandwidth usage: the number of exchanged messages, and the size of the messages. As in this work, messages are of constant size, our focus relies on the number of exchanged messages. In general, increasing the number of communications between nodes contributes both to responsiveness (changes are transmitted earlier) and consistency (more messages allow a more accurate knowledge of the game's state). On the other hand, bandwidth is costly and limited, and it has even been shown in [101] that too many messages degrade network performance, leading in turn to inconsistencies.

In practice, some games rely on a simple *fixed frequency* strategy, ensuring that each node sends updates at a regular rate to the other nodes [137]. The main flaw of this technique is that the update rate has to be fixed when implementing the application, which often results in either oversampling or undersampling of updates [14]. If the update rate is fixed depending on the strongest consistency need, then in situations where this need is lower, the bandwidth usage will be higher than needed (oversampling), and if on the contrary, the update rate is fixed to reduce bandwidth usage, then it is very likely that situations will appear where consistency is lacking (undersampling). Also, if all nodes send their updates to all other nodes, then this kind of strategy will have a poor scalability, as the number of messages increases quadratically with the number of nodes. Scalability is a concern for DVEs, as some games are intended to be

---

[118].

played by a very large number of participants at the same time (e.g. MMOGs such as World Of Warcraft).

Reducing bandwidth usage by limiting the number of exchanged messages could be achieved with several techniques.

1. *Data compression* like Delta encoding [105, 123] is an implementation trick where only differences between states are sent. This is particularly useful if the state of the game is large, but very dependent on the application. This technique also aims at reducing bandwidth usage by reducing the sizes of the messages, which is of low interest in this work.

2. *Dead-reckoning* (described in more details in Section 1.3.2) is a widely used tool that consists in adding to updates information about how the state of an object is evolving with time. Typically, when maintaining an estimation about the position of an object, updates include not only the position of the object, but also its speed and maybe even its acceleration. This technique improves bandwidth usage by lessening the need for two connected nodes to often send messages to each other.

3. *Interest Management* consists in filtering updates in order to send them only to nodes who might be interested.

   While all of the previously presented constraints seem to make DVEs a lot more complicated to design, other characteristics can be leveraged to ease these restrictions. Because DVE applications are used by humans, with a limited attention capacity, in some cases it may be unnecessary to achieve strict consistency, and better instead to lower the precision required for some elements [20], as certain elements are less important than others. For example, if two nodes see a decorative element in two different states (like seeing a tree in a withered or lush state), it is less of a problem than if the inconsistency is about an objective or an enemy. Also, interactions between characters and/or objects of the environment are usually enabled when they are sufficiently close in the virtual world, so that far away objects are less important. We can see through these remarks that the *interest* and thus the consistency requirements nodes have for other elements may vary depending on the circumstances. Hence the idea of Interest Management.

   Different types of interest management can be identified [27, 94]. It can be zoned (a node receiving information from nodes on the same region as them in the map), based on aura (a node receiving updates from close nodes), based on visibility (a node not receiving updates from nodes outside their line of sight), based on type (a node receiving updates only from nodes of intrinsic interest), or based on a combination of these criteria. Some application-specific approaches may also use the fact that human attention is limited, as in [20], where a set of five interesting nodes is defined at any given time, based on a hybrid interest set management based on distances to other nodes, the time of their last interaction and the angle targeted by the local node. Frequent updates are received from those five nodes, but much less from other nodes. This fairly reduces bandwidth usage, and makes the DVE more scalable to an increased number of nodes.

Of course, combinations of all these techniques can be used. In [30], in combination with Dead-reckoning, an area of interest, similar to aura interest management, is used to modify the threshold of the Dead-reckoning calculations.

Let us now describe Dead-reckoning in more details, a powerful technique that will be used in Chapter 2.

1.3.2 Dead-Reckoning

Dead-Reckoning is a widely used tool, standardized in the Appendix E of [68]. While it can be used for other quantities that evolve with time, Dead-reckoning primarily targets position estimation. Let us consider a node A that estimates the position of another node B. Dead-reckoning consists in adding to the updates that B sends to A, besides information on B's position, additional data about the evolution of this position, for example B's speed and acceleration. Node A uses this information to predict the state of B, extrapolating its movement after each update. By making so that the nodes share the same estimation algorithm, B is able to compute the same estimation as A, so as to know where A estimates B's position. Node B can thus detect if the error on its own position as seen by A is above a given threshold. When this happens, B sends a message to A in order to correct the outdated estimated position (as well as the speed and the acceleration). Thus nodes know their own actual position at any time, and for the other nodes, they only know estimated positions.

Dead-reckoning has several advantages. It can be used to reduce bandwidth usage by sending messages only when needed, but it can also be used to guarantee a bounded error on the positions evaluated by the nodes, and thus improving consistency. It has also further beneficial effects with regards to consistency. As nodes continue to predict the positions of other nodes between updates, their trajectory is smoother, and thus corresponds more accurately to the actual trajectory of the nodes (in conjunction with other techniques for smoothing the trajectory at each update like in [92]). Also, the fact that the nodes use the same prediction algorithms and thus can share information about estimated positions can help for consistency: for example in Chapter 2, the nodes A and B compute their estimated distance as the distance between the two estimated positions, instead of each node computing the distance between its own actual position and the estimated position of the other node, which is more precise but less consistent.

Dead-reckoning can be used to cope with latency, as in [91], where it is shown that reducing the Dead-reckoning threshold (or equivalently increasing the number of exchanged messages) can compensate for the latency with regards to the average number of errors per time unit. Also, Dead-reckoning is flexible with respect to trade-offs between required consistency, and available bandwidth [53], because lowering the Dead-reckoning threshold generates more communications, but improves the accuracy of information (and vice versa).

The main disadvantage of this technique is a higher computational cost than simplistic methods, like the fixed frequency strategy mentioned previously, where nodes send message at a regular rate. However, this cost depends a lot on the prediction scheme, which is generally very simple and, as we supposed in Section 1.2 that local computations are fast with respect to communications, this disadvantage is mitigated in our model. Also, Dead-reckoning is ineffective if nodes are subject to frequent and unpredictable changes in speed, or random movements.

Research on Dead-reckoning can improve bandwidth usage mainly in two ways: get the best prediction possible (for example, in [80], a prediction is proposed that adapts to the shape of the movement of the nodes, and in [41, 51] trajectories that are often used by the nodes are computed beforehand to help the predictions), or improve the update policies (a survey on different update policies is given in [106]). This last aspect often relies on *Interest Management* (see Section 1.3.1).

If the future movements of the nodes are unpredictable, Dead-reckoning is optimal in terms of number of messages for estimating positions. Indeed, when using Dead-reckoning, a node sending updates to another node knows precisely the estimation of the position of this other node, and thus may send updates if and only if the tolerated error between the actual position and the estimated position is exceeded, making it send no more messages than required.

However, if two nodes target at estimating the distance between them instead of their po-

sitions, as in Chapter 2, then none of the two nodes knows the actual distance between them. None of the nodes may thus detect when exactly it is necessary to send a message, making it a harder problem.

1.3.3 Fully distributed DVEs

Even though DVEs are fundamentally distributed, a distinction can be made between centralized and fully distributed DVEs, depending on the presence or not of a central server.

In practice, many online games are based on a client-server architecture. While having a server makes it much easier to achieve consistency [129], this has many disadvantages, because maintaining a server is often expensive, and exposes a single point of failure [123]: server based solutions are thus less scalable. This leads to the incentive to study peer-to-peer solutions, where nodes share the role of the server among themselves, but in this context, bandwidth becomes crucial, as the network capacities of peers are generally low. This is why the solutions explored in this document target bandwidth economy in fully distributed DVEs.

Let us note that in a fully distributed DVE, because of the absence of a server, the nodes will need to communicate directly to each other. The system may then be modelled by a connection graph (see p.20), and each node may keep connections only to a limited set of other nodes.

If the connection graph consists in the complete graph (that is, each nodes is connected to every node), then each node has to maintain $O(n)$ connections, where n is the number of nodes in the DVE. The bandwidth consumption could thus be very high, because in most DVEs, nodes send updates to all other known nodes at regular time intervals, and generally because in most networks, nodes have to periodically send "keep-alive" messages [117] to maintain a connection: the total number of exchanged messages per time step would thus be $O(n^2)$. Also, if each node has to maintain that many connections, it could fill up unnecessary local memory space. It could also be bad for responsiveness: it could take a lot of time for each node to retrieve the state of everyone and handle incoming messages, so that short intervals of time between updates would be difficult to ensure. Also, in this case, at each update the differences between new and out-of-date positions would be higher, which in turn means it would be harder to ensure consistency.

Thus, in the context of fully distributed DVEs, it is of a high importance to have methods to allow nodes to be connected only to relevant nodes.

This problem of constructing sparse connection graphs in a fully distributed context is strongly related to peer-to-peer overlay networks. It is thus useful to look into the classical results of this field.

1.4 Peer-to-peer Overlays

An overlay network is a virtual network built on top of another underlying network. In other words, an overlay is a connection graph built in a system where routing is supposed to work transparently: as explained in Section 1.2, a node u must be physically able to send a message to any other node, but is limited to send message only to the nodes that are adjacent to u in the connection graph, that is, to nodes of which u knows the ID.

This topic has been subject to extensive research in the early 2000's; surveys can be found in [10, 96, 124]. Note however that most of these results are targeted at file sharing: it aims at a system where some nodes posses or own files, while others would like to download them. Let us look at some of the main results while keeping in mind that in our case, nodes have a

high mobility, that is, the states of the objects have to be retrieved regularly, and that nodes need information only about nodes that are close to them in the DVE.

1.4.1 Unstructured Overlays

Historically the first overlays, unstructured overlays do not monitor the connection graph itself, which is thus a somewhat random graph. Because of this, it is needed to implement some sort of graph exploration in order to locate resource. Several ways can be identified.

In the earlier versions of *Gnutella* [39], a simple flooding technique was used to locate a data: the node looking for a data sends a query message to all its adjacent nodes, who in turn transfer the query to their own adjacent nodes. This goes on until a node that possesses the data is found, in which case the identifier of this node is backpropagated to the querying node, which can then directly connect to it in order to download the data. Even if some limiting factors are implemented (a time-to-live is added to each message, so that it stops being propagated after a certain number of hops, and a unique identifier is given to a query so that nodes may stop propagating it if they already got it) this method is clearly not scalable when there are a lot of nodes.

With the popularity of Gnutella, a lot of proposals have been made to improve the scalability of the queries. In [99], it is shown that to locate a file, k simultaneous random walks (k queries are sent, and a node receiving a query transfers it to a random node), induce less messages than flooding. Unfortunately, the delay before the file is found is an order of magnitude higher than with flooding; however, this delay may be reduced by increasing k: according to [99], the delay is inversely proportional to k. In [85], a better routing is proposed.

In Freenet [43], queries are not flooded, but routed in a way reminiscent to a depth-first search. Each node receiving a query checks if it has stored the queried data, and if it does, the query is backtracked to the original sender with the data; if not, the query is forwarded to the adjacent node that is the most likely to have the data. Similarly to Gnutella, queries are accompanied with a unique identifier and a time-to-live number, so as to avoid loops, and to limit the number of hops. Because each node and each data is associated with a key, because queries are routed first to nodes with the key that is the closest to the key of the queried data, and because the data is cached by the intermediate nodes when backtracking a query back to the original node, query routing becomes more and more efficient as time passes for popular data.

While unstructured overlays have been subject to a lot work to make location of data more scalable [81], and while they are often used in practice for file-sharing applications, they are not satisfactory for our problem. Most of the solutions are *best effort* in nature, and fail to give a guarantee that a query is answered successfully even for a data that is available in the network. Similarly, the search of a data may take a long time, requiring to traverse a lot of nodes before finding a node that possesses the data, which in turn is costly in terms of bandwidth usage, especially if data has to be recovered often, like for positions that change regularly.

1.4.2 Structured Overlays

A good example of structured overlays are Distributed Hash Tables (DHT), because they allow to hash keys to some data items: a node that knows the key of a data item it needs to retrieve can start a query in the DHT, to which the DHT will answer with the location of the data item.

Structured overlays, contrary to unstructured overlays, impose stronger properties on the connection graph so as to enforce desirable properties. Typically, the number of connections a node has to maintain is bounded, as is the number of hops needed to find a data starting from any node.

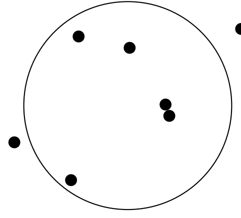


Figure 1.3 – A range query can be used to get all nodes in a certain zone

While it has been considered that structured overlays are an improvement to unstructured overlays, both approaches can actually be complementary; for example propositions have been made to add structured aspects to Gnutella [36, 95]. The main advantage of unstructured overlays, is that it is usually quicker to find a popular, highly replicated data. However, as said previously, unstructured overlays fail to give guarantees and bounds on the time it takes to find a data.

Several DHTs, like Chord [138], Pastry [125] and Tapestry [146], work with circular key-spaces, with each node and each data associated with a key from the space. Information about an object is maintained on the node that has the closest key to the key of the object. A node u is connected to $O(\log n)$ other nodes (with n the total number of nodes in the overlay), in such a way as to be connected to increasingly less nodes, the farther away their keys are away of u's key. This allows to greedily route a query from any node to any other node in a maximum of $O(\log n)$ hops.

For its key-space, CAN [122] uses a (d-dimensional) torus. Each node is responsible for a (d-dimensional) rectangle: each time a node connects to the overlay, it is directed to a random node already in the overlay (using a centralized DNS-like service), which will divide its rectangle in two, giving the new node the responsibility of one of the halves. A node keeps a copy of (or a pointer to) every object that has a key that is situated in its rectangle. Each node connects to the nodes that are responsible for a rectangle that is adjacent to its own rectangle, resulting in $O(d)$ adjacent nodes; a query can be routed greedily in $O(d \cdot n^{\frac{1}{d}})$ hops.

The main problem of this type of solutions is that the keys are usually assigned uniformly to the nodes and stored data items, that is, the values of the items are completely independent of the values of the keys. This is a desired effect for some applications, because it makes sure that the keys in use are spread out uniformly in the key-space, so that nodes end up responsible for approximately the same number of keys, but it makes these solutions unpractical in our context. Classical DHTs map keys to individual objects, so that queries answer questions of the type "what is the value of this object", which in our context translates to "what is the position of this character". As explained previously, one of our difficulties is that nodes need to detect when the character of another node it is not aware of, gets closer to its own character. This means that we need to answer to queries of the form "what characters are situated in this area", that is, queries that are not looking for one specific data item, but for a set of data items satisfying some given criteria.

Fortunately, research has been conducted on range-queriable DHTs. In range-queriable DHTs, the data items are associated with a certain number of attributes, and queries can be made to filter data items according to these attributes. In our case, these attributes would be the positions of the characters; we could then use range queries to get all characters in a certain zone, like on Figure 1.3 (see also Section 1.5.1).

Some solutions for range-queriable DHTs may adapt the key-space to be inherently range-queriable. In [16], Voronoi diagrams are used to build an overlay that maps objects with similar attributes to close-by keys, allowing naturally to do range queries. However, due to a bad scaling

of Voronoi diagrams in higher than two dimensions, this solution is adapted only to objects that have two attributes. In [108], this limitation is overcome by partitioning the attributes in pairs, and by constructing a Voronoi diagram on one of the pairs; when several data items have the same attributes, thus forming a "cluster", another Voronoi diagram on two other attributes is constructed on only the data items of this cluster. This procedure is repeated until a hierarchy of Voronoi diagrams is obtained. Queries are performed top-down, the search space being restricted at each level.

Hashing functions can also be modified so as to allow range queries. In [62], a Locality-Sensitive Hashing is used with Chord to allow single attribute range queries to be executed, but answers are approximate.

Other techniques may construct additional data structures to allow range queries, on top of another DHT. In [140], a quadtree is built on the space of the object attributes: the space is divided into squares, that are divided in successively smaller squares. Each square of the quadtree is associated with a node, and data items are hashed though Chord to the node that is responsible for the square they are located in.

Contrary to uniformly hashed DHTs, range-queriable DHTs may be prone to load balancing problems, that is, they may result in situations where some nodes are responsible for significantly more data items than others [22].

However, load-balancing is not the main issue with these overlays. The main weak-point of structured overlays in general, is that insertions and deletions of nodes into the overlays are costly. In file-sharing applications, this means that transient nodes are not handled effectively, that is, nodes that connect to the system and then disconnect shortly after.

Structured overlays are designed with the assumption that the values of the data items remain constant. While they are *dynamic* in nature, that is, these insertions and deletions of nodes are allowed, they are not *kinetic*, that is, they do not allow for nodes to change their values once they are inserted in the overlay. Therefore, in the context of moving nodes, each time a position changes, a node would have to be removed from the overlay and then reinserted, which may cost unnecessary computations. Structured overlays need thus to be treated with care when dealing with moving objects, either by adapting and improving them, or by using them as a tool alongside other options.

1.4.3 DVE Overlays

Some overlays have already been proposed for mobile points in DVEs [29, 144], which are thus highly related to our problem.

Even if shortcomings of DHTs have been identified in the previous section, it has been proposed to use range-queriable DHTs in several ways to deal with mobile nodes.

Colyseus [21] uses Mercury [22], and separates the DHT in several key-spaces, one for each coordinate. This DHT is used to store some items representing the interests and positions of the node, so that the system may detect if a node needs to track another one. In other words, the DHT is used only as a rendezvous point to notify nodes that are not aware of each other, but that are interested in each other, and to allow them to directly connect and then track each other's position. This lowers the bandwidth usage as nodes do not need to go through the DHT each time they send updates, but a high overhead is still incurred, because changes in positions lead to changes in interests, which imply insertions and deletions of items in the DHT.

Other propositions divide the virtual environment in several regions. In [49, 82] the virtual world is divided statically, that is, regions are defined beforehand, independently of the nodes'

positions; each region is assigned to a superpeer[6], a node that becomes in charge to transfer all updates to the nodes in and around the region. The superpeers are connected through a DHT, that is used when a node changes region. While these approaches are interesting, they divide the virtual environment beforehand, which, similarily to the problem of the choice of an update rate in fixed frequency strategies (see p.23), raises the question of the optimal size of the regions, which is application dependent. Also, the superpeers may be located in any region (not necessary the region they are responsible of), which makes their connections not local, and adds an additional burden on them, because they need to connect to the nodes in the same region as them in addition to the nodes in the region they are responsible of.

These problem are mitigated in solutions like [109], where the regions change according to the current positions of the nodes. The drawback of this kind of solution is a higher overhead, as recomputing the regions and assigning superpeers to them may be costly.

Other solutions like [76] and [130] rely on regular exchanges of information between nodes. While promising on certain practical settings, they come with few theoretical proofs. In order to use these solutions to provide guarantees on the performances, additional work would be needed on each of them individually.

In [130], a concept close to Yao-graphs is used: the space around a node is divided in equally-sized angular sectors, and each node is connected to the closest node to it in each of the sectors. Solutions of this kind will be discussed in Section 3.7.1.

Other geometrical structures can be helpful. In Solipsis [78, 79], convex hulls are used: each node keeps connections with other nodes in such a way as to be situated in the convex hull of its adjacent nodes. It is explained that this allows for a global connectivity of the connection graph. Communications are initiated each time a node notices that it is no longer in the convex hull of its adjacent nodes, so as to restore the property. However, it is shown that Solipsis may still lead to situations where a node gets close to another node without being detected.

The use of Voronoi diagrams seems promising. VON [66] constructs a Voronoi diagram, and on top of all nodes that are at a distance smaller than r[7] (called neighbors), each node is connected to the nodes whose Voronoi regions have a common edge with its own Voronoi region (called enclosing nodes). When a node moves, the outermost neighbors will be in charge to notify the moving node of eventual new neighbors. The problem of this solution is that the Voronoi diagram has to be recomputed at each movement, even if the neighbors and enclosing nodes do not change. Also, in some degenerate cases, the number of enclosing nodes can be arbitrarily higher than the number of neighbors [67], as can be seen on Figure 1.4. Unfortunately, the article that describes VON ([66]) is not proven theoretically, so it is difficult to know the exact bounds; also, it seems that in the simulations, it can happen that a node is not aware of a neighbor for a short amount of time. This last aspect is dealt with in [71], but alas, without theoretical proof either.

Some practical improvements of VON have been proposed, like [37], that takes into account visibility blocking environment elements. Also, the bandwidth consumption can be lowered by using forwarding in conjunction with packing and compression [72].

In [57], the nodes are randomly separated in two sets, and two Voronoi diagrams are maintained on each set. This allows to move all the nodes of one of the sets in parallel, using the nodes of the other set as stationary points to help reconstruct the Voronoi diagram. However, this may lead to problems if nodes of the same set get crowded far away from nodes of the other set.

[6] Called coordinator in [82] and region master in [49].

[7] r is called AOI for Area Of Interest in the article

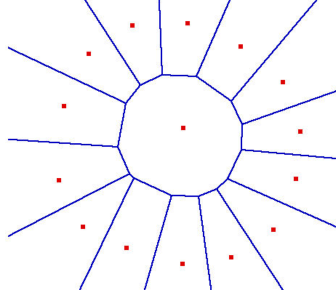


Figure 1.4 – Example of a case in VON where a node has an arbitrary number of enclosing nodes (this figure was taken from [67])

As we can see, a lot of these solutions are based on some geometric structure. Let us now look on some geometrical queries, and the structures that can be built to answer those queries.

1.5 Data Structures for Proximity and Distance Queries

In this section, we present some classical queries that are related to those described in Section 1.1.1, and then, more importantly, we present classical geometrical data structures that are often proposed to serve those queries. As of now, we will consider the centralized setting, with a set of static nodes, that is, their positions do not change with time.

1.5.1 Proximity Queries

Let us denote by p_u the position of node u, and by $d(u, v)$, the distance between nodes u and v (with $u, v \in \mathcal{V}$).

There are several distinct queries that depend on the proximity of the nodes:

- Range queries, mainly orthogonal range queries [98] return all nodes in a given range. In orthogonal range queries, the range is an axis-parallel box. In other words, given $(a_1, a_2, \ldots, a_d)$ and $(b_1, \ldots, b_d)$, with $\forall i, a_i \leq b_i$, the orthogonal range query returns all nodes v with $p_v = (x_1, \ldots, x_d)$ such that $x_1 \in [a_1; b_1] \wedge x_2 \in [a_2; b_2] \wedge \cdots \wedge x_d \in [a_d, b_d]$. Other shapes may be allowed for other types of range queries, like convex shapes or polygones [135].

- Nearest-neighbor [3] returns, for a given node, the node that is closest to it with regards to the distance function d. In other words, given a query node u, it should return a node $v \neq u$ such that $\forall w \in \mathcal{V}, d(u, w) \geq d(u, v)$.

  Approximate nearest neighbor (ANN) [48, 55] is similar, but accepts an error ε: given a query node u, it should return a node v such that $\forall w \in \mathcal{V}, d(u, w) \geq (1 - \varepsilon)d(u, v)$.

  Other variants are the k-nearest neighbor, i.e return the k closest nodes to a query node, [119], and the all near neighbors, i.e return for each node in $\mathcal{V}$ (instead of just for one given node), the node that is closest to it [55].

- Collision detection [50] happens on a set of objects (for example polygons): nodes are thus no longer reduced to just one position, and take up some (small) portion of the virtual environment. Collision detection as a query should return the set of pairs $\{u, v\}$ such that $u \cap v \neq \emptyset$. In most practical settings however, collision detection is rarely used as a query, because the movement of the nodes may be dependent on the advent of collisions: for

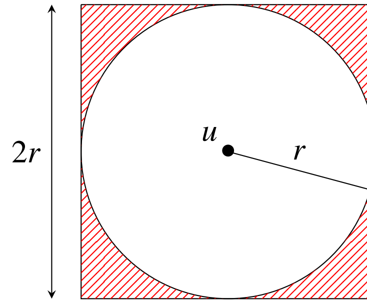


Figure 1.5 – Answering $\text{CloseNodes}_u(r)$ with an orthogonal range query. The nodes which are situated in the red hatched zone have to be filtered out.

example if nodes represent balls, they should bounce off each other, or if they represent cars, they should crash and maybe stop their movement.

Recall the CloseNodes query we have introduced on page 17:

Definition 1.2 ($\text{CloseNodes}_u(r)$)**.** *Given a node $u \in \mathcal{V}$ and a distance $r \in \mathbb{R}$, return all nodes $v \in \mathcal{V}$ such that $d(u, v) \leq r$.*

As mentioned in Section 1.1.1, in Chapter 3, r is a fixed parameter of the query, while in Chapter 4, r is an input to the query, and is thus not known in advance.

A naive approach to answer to the $\text{CloseNodes}_u(r)$ query consists in checking all nodes in $\mathcal{V}$, and keep only those at distance r from u, resulting in $O(n)$ computations. We are thus looking for solutions that answer to the query time sublinear with the number of nodes.

A $\text{CloseNodes}_u(r)$ query can be answered with a range query: the query range just has to be a superset of the ball of radius r centered on the query node u. For example, let us consider nodes that are in a two-dimensional space, and let us use the euclidean distance (the nodes to return with $\text{CloseNodes}_u(r)$ are thus the nodes situated in a circle of radius r centered on u). An orthogonal range query can be used, that returns all nodes within a square with a side of length $2r$ centered on the query node, as shown on Figure 1.5. However, the range query returns additional nodes on top of those needed for $\text{CloseNodes}_u(r)$: on Figure 1.5, the nodes situated in the red hatched zones should be removed from the returned set. A filter has thus to be applied before returning the result of the query. In the worst case, there may be arbitrarily many more nodes in the red hatched zone than in the circle, but as the area of the square is $\frac{4}{\pi}$ times greater than the area of the circle, on average only $\frac{4}{\pi}$ times more nodes are returned before filtering.

On the other hand, the nearest-neighbor and related queries cannot be used directly to answer to CloseNodes. However, as we will see in Section 1.8, some structures used for nearest-neighbor searches are of interest to help find efficient structures for the CloseNodes query [48, 55].

Let us now review some classical structures of the literature that can be used to answer to the CloseNodes query.

1.5.2 Geometrical Data Structures

To help answer to proximity queries, several data structures can be used. Most of the structures studied here can be divided into two broad categories: tree structures, and geometric spanners.

Tree Structures

Trees are inherently well-suited for searching, thanks to the hierarchy they create. Thus there are many structures tailored for proximity queries that take the form of a tree. A tree is an acyclic connected graph in which each element of the tree, which we will call *metanode* (because it may or may not correspond to a node of $\mathcal{V}$) is linked to one or more other metanodes, with one of them being its *parent*. There is only one metanode that does not have a parent, which is called the *root* of the tree.

A *range tree*, as its name suggests, allows to perform range queries. A d-dimensional range tree is defined recursively: a Binary Search Tree is constructed for the values of the positions of the nodes in one of the dimensions, with each of the metanodes in the tree being a $(d-1)$-dimensional range tree. The structure takes $O(n \log^{d-1} n)$ space[8], and range queries are answered in $O(\log^d n + k)$, where k is the number of nodes to return [17].

Multidimensional binary search trees, or *kd-trees* [18], recursively divide the space in two parts, but with the same number of nodes situated in it (except for the parity of the number of nodes). The structure takes $O(n)$ space, and range queries can be answered in $O(d \cdot n^{1-\frac{1}{d}} + k)$ [88].

Quadtrees, as kd-trees, recursively divide the space. Here, however, the space is divided in 2^d equal sized hypercubes. The metanodes in the tree do not correspond to nodes from $\mathcal{V}$, but represent the hypercubes, and they are recursively divided until the hypercubes of the leaves contain at most one node of $\mathcal{V}$. As this may lead, depending on the positions of the nodes, in a lot of unnecessary metanodes (see Figure 1.6), quadtrees may be compressed, resulting in a linear size [44]. Quadtrees are not usually associated with range queries, but we will prove in Section 4.1.3 that they can be used to answer to two-dimensional CloseNodes queries in time $O\left(b_u\left(r + r\sqrt{2}\right)\log\Phi\right)$, with $b_u(x)$ the number of nodes at distance at most x from u (see below, Definition 1.6 p.36 for the definition of Φ, the aspect ratio).

Algorithms with this kind of computation cost that depends on the number of nodes in a ball slightly bigger than the target ball ($b_u\left(r + r\sqrt{2}\right)$ instead of $b_u(r)$), may visit arbitrarily many nodes in the worst case, but visit on average about as many nodes as there are in the target ball. This is disadvantageous in situations where the density of nodes is highly variable: if u is an isolated node, at distance slightly more than r from a huge cluster of nodes, than the number of visited nodes by the $\text{CloseNodes}_u(r)$ query may be high in comparison to the size of the returned set.

However, it is arguable that these kinds of situations are corner cases (for example, in Fortnite, the players are usually homogeneously spread on the map [107]). We usually make assumptions on the number of nodes that are allowed in a small area, by stating that only a constant number of ρ nodes can be situated in a ball of a given small radius. If this is the case, we can assume that for a constant c, $O(b_u(cr)) = O(k)$, where $k = b_u(r)$ is the size of the set returned by the query, so that the above query cost for quadtrees in two dimensions may be simplified to $O(k \log \Phi)$.

The structures we presented are the most common, and some improvements have been done to reduce memory usage or query time [38, 40].

Geometric Spanners

A geometric spanner of a set of nodes, is a graph $G = (\mathcal{V}, E)$ constructed on those nodes, where the edges are weighted with the distance of their extremities (that is if $(u, v) \in E$, then the weight

[8] Recall that the space usage of a data structure counts the number of registers needed to store it in memory.

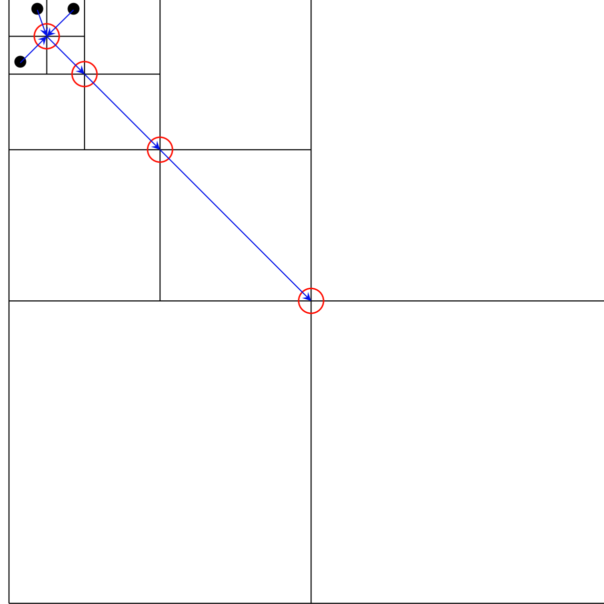


Figure 1.6 – Example of an uncompressed quadtree with a lot of unnecessary metanodes. Metanodes are empty red circle (◯) and nodes are black dots (•). The blue arrows represent the edges in the quadtree.

of (u, v) is $d(u, v)$). The length of a path in G is the sum of the weights of the edges composing it. A spanner has to comply to the following property.

Definition 1.5 (s-spanner)**.** *A graph G is a spanner with stretch factor s, or a s-spanner in short, if the length of a shortest path from any node u to any node v in G is smaller than or equal to $s \cdot d(u, v)$.*

The most obvious spanner for a set of nodes is the complete graph ($E = \mathcal{V}^2$), which has a stretch factor $s = 1$. However the complete graph has a lot of edges ($O(n^2)$ in total), which uses a lot of memory. Thus spanners that have few edges are generally preferred.

Geometric spanners are of high interest for this work, because they can efficiently answer to the $\textsc{CloseNodes}_u(r)$ query: for any node v such that $d(u, v) \leq r$, it is guaranteed that there is a path of length at most $s \cdot r$ from u to v. Thus, a simple breadth first search that stops on nodes that are at distance more than $s \cdot r$ from u in the spanner, is guaranteed to find all nodes needed for $\textsc{CloseNodes}$[9]. This query ends up checking only nodes that are at distance at most $s \cdot r$ from u, and all the outgoing edges from these nodes. The time complexity of $\textsc{CloseNodes}_u(r)$ on a spanner is thus $O(b_u(sr) \cdot \delta)$, where $b_u(sr)$ is the number of nodes in the ball of radius sr centered on u, and δ is the maximum degree of a node in G[10].

As described previously, if the density of the nodes is limited, $b_u(sr)$ is of the same order as $b_u(r)$, which makes it so that spanners can be used to answer to $\textsc{CloseNodes}$ queries in $O(k \cdot \delta)$ time (with, again, k the number of nodes returned by the query).

Let us present some known spanners:

[9]Note that we are here not interested to know which of the paths from u to the nodes returned by the query are the shortest.

[10]It is worth noting that if the spanner maintained the outgoing edges of each node in order of increasing weights, then we could answer the query in $O(b_u(sr))$: during the BFS, when checking the outgoing nodes of the current node v, only those of weight higher than $sr - d$ would need to be checked, with d the distance from v to u in the spanner. This is, however, impractical to maintain in the kinetic setting, as a kinetic sorted list would have to be maintained on the outgoing arcs of each node.

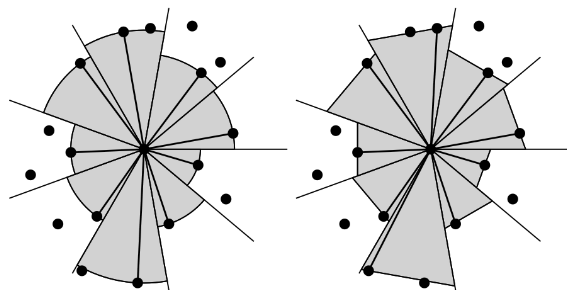


Figure 1.7 – Example of the construction of a Yao graph and a Theta graph in two dimensions on the same set of nodes. Left: connections of a node in the Yao graph; Right: connections of a node in the Theta graph (this figure was taken from [26])

- We have already presented Voronoi diagrams. A Delaunay triangulation is a dual of a Voronoi diagram: it is a graph in which any two nodes that have adjacent Voronoi regions have an edge in the graph. It has been shown that in two dimensions, with euclidean distances, Delaunay Triangulations are spanners of stretch factor 2.5 [77]. We have seen with Figure 1.4, that the maximum degree of a node in a Delaunay Triangulation may be $n-1$; however, the average degree of a node is constant in two dimensions, making it interesting for efficient CloseNodes queries. However, this is no longer the case at higher dimensions, explaining why Delaunay triangulations are rarely studied in this case.

- A Yao graph in two dimensions (resp. in d dimensions) partitions the space around each node in k sectors (resp. cones) of the same angle (resp. solid angle). The nodes are then connected, for each sector, to the node that is closest to it among the nodes that rely in this sector (see Figure 1.7 for a two dimensional example). It is known that with the euclidean distance, this construction leads to a spanner if k is sufficiently large : in two dimensions, for $k > 6$, the stretch factor is at most $1/(1-2\sin(\pi/k))$ [145]. For $k = 6$, we have a stretch factor of 20.4 [114]. With $k = 4$, the graph is also a spanner, but with a very high stretch factor (about 696) [25]. Similarly as with Delaunay triangulations, while the degree of nodes in a Yao graph is constant on average (depending on the number of sectors), the worst degree may go up to $n-1$. However, contrary to Delaunay triangulations, the degree of nodes in Yao graphs remains constant on average at any dimension (although the constant gets bigger in higher dimensions, as more sectors are needed to keep a spanner).

- A Theta graph is a variant of the Yao graph. Instead of connecting the nodes to the closest one in each sector, each sector is associated with a fixed ray (usually the bisector), and the nodes are connected to the node in each sector, whose projection on the ray is the closest one (see Figure 1.7). As for Yao graphs, Theta graphs are spanners when the number of sectors is sufficient. In two dimensions, and again with the euclidean distance, for $k > 6$, we also have a stretch factor of $1/(1-2\sin(\pi/k))$ [126]. For $k = 6$, we have a stretch factor of 2 [24], and for $k = 5$, the stretch factor is of 10 [26]. It has been proven that Theta graphs are still spanners in higher dimensions [112]. The degree of nodes in a Theta graph is also similar to Yao graphs: constant on average, but $n-1$ at worst.

Thus, to the best of our knowledge, in two dimensions the best proven stretch factors are for Delaunay triangulations and Theta graphs. In higher dimensions, Delaunay triangulations are impractical, and Yao and Theta graphs should be considered.

In this work, our interest relies mainly in a specific type of construction that can also lead to

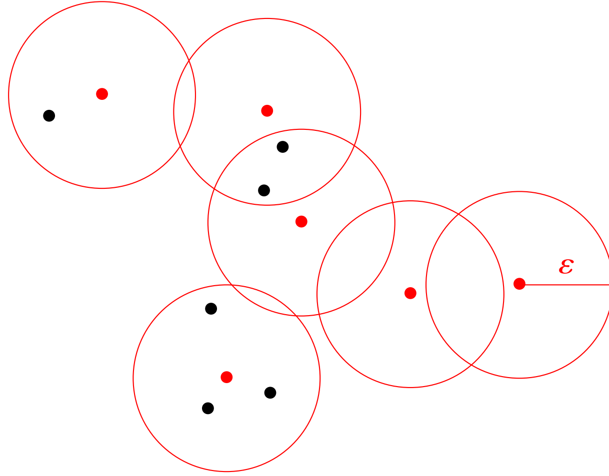


Figure 1.8 – An example of an ε-net. The nodes in red are those that belong to the ε-net.

spanners:

Navigating Nets

One of our focuses relies on hierarchies of ε-nets[11]. This kind of construction has been used in several papers for proximity queries [48, 55, 63, 83].

An ε-net (with a parameter $\varepsilon > 0$) on the set of nodes $\mathcal{V}$ is a subset of $\mathcal{V}$ so that any two nodes from the subset are at distance at least ε from each other (sometimes called the packing property), and so that any node in $\mathcal{V}$ is at most at distance ε from a node from the ε-net (sometimes called the covering property). See Figure 1.8 for an example of an ε-net.

We call navigating net[12] (with a parameter $b > 1$) a succession of b^i-nets. The *level* 0 contains all the nodes from $\mathcal{V}$. The level 1 contains a b-net of the level 0, the level 2 contains a b^2-net of level 1, and so on, until a level is reached where only one node remains, which we will call the root. See Figure 1.9 for an example of a navigating net.

Definition 1.6 (aspect ratio)**.** *We denote by*

$$\Phi = \frac{d_{max}}{d_{min}}$$

the aspect ratio of $\mathcal{V}$, with d_{max} the maximal inter-distance between the nodes, and d_{min} the minimal inter-distance, that is, two value such that $\forall u, v \in \mathcal{V}, d_{min} \leq d(u, v) \leq d_{max}$.

We can see that the navigating net cannot have more than $O(\log \Phi)$ levels. Each node u has thus a maximal level L_u.

The navigating net is then a graph in which each node has an edge with its *parent*. The parent of a node u is a node v so that $L_v > L_u$, and that covers u at level L_u, that is, so that $d(u, v) \leq b^{L_u+1}$.

The navigating net may be enriched with a set of *neighbors* for each node at each level[55]. The neighbors of u at level i are the nodes that are close to u at this level: $\{v \in \mathcal{V} : L_v \geq i \wedge d(u, v) < c \cdot b^i\}$, with c a constant.

It is shown in [55], with $b = 2$ and $c > 4$ that the navigating net is a spanner of stretch factor $\frac{16}{c-4}$. One of our navigating nets (in Section 4.3) has $b \geq 6$ and $c = 2$, and we prove that the stretch factor is 60 when $b = 6$.

[11] ε-nets are called r-nets in [42, 63], Y_r-nets in [48], and discrete centers in [55]

[12] Navigating nets are called net trees in [42] and [63], and the structure from [55] is called DefSpanner.

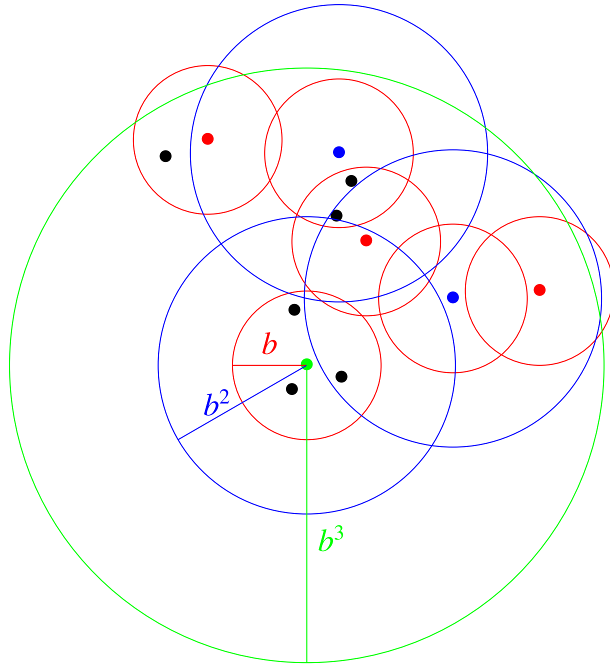


Figure 1.9 – An example of a navigating net, constructed on the same nodes as Figure 1.8. The green node is the only one in level 3. Blue and green nodes are in level two, as well as in level 1 in which there are also red nodes. All nodes are in level 0.

This structure can be used to answer to CloseNodes queries using the properties of the underlying tree. We will see in Chapter 4 that when starting from a node u, the query can be answered by going up and down the hierarchy, resulting in a query time of $O(\log r + b_u(O(r)))$, where $b_u(x)$ is the number of nodes in the ball of center u and of radius x. We may notice that in the worst case, $b_u(O(r))$ may be arbitrarily higher than $b_u(r)$. However, on average, the number of nodes is multiplied only by a constant, and if the metric space is doubling (see Section 1.5.3), as we supposed that there is a minimal inter-distance between the nodes d_{min}, then the worst case time complexity of the query is $O(\log r + k)$, with $k = b_u(r)$, the number of nodes to return.

Navigating nets can also be adapted to the dynamic setting, as shown in [55]. Node insertion and deletion are dealt by traversing the hierarchy top down and/or bottom up, which takes $O(\log \Phi)$ computations.

Unit Disk Graphs

A unit disk graph[13] of unit r on a set of nodes in the plane, is the graph where each node has an edge with every node that is at distance r from it.

Unit disk graphs are used extensively in research about ad-hoc networks, as they model wireless devices that can communicate only with other devices that are close enough [54, 84]. In such settings, the unit disk graphs are usually taken as entries to other problems: for example, in [54], connected dominating sets are constructed, leveraging the fact that the nodes are in a unit disk graph.

We are however mainly interested in unit disk graphs as structures that can be used to answer to CloseNodes$_u(r)$ queries when the radius r of the query is fixed and known in advance, as in Chapter 3. As in the unit disk graph, each node u has an edge with all nodes at distance r, the query is easily answered by returning all nodes that have an edge with u. We are thus interested

[13] Also called euclidian graphs, or geometrical graphs, in other settings.

in taking a sets of nodes and their positions as an entry, and constructing unit disk graphs on those sets of nodes.

In [74], disk graphs, a generalization of unit disk graphs are studied. Taking a dynamic set of nodes and their positions, a structure is built that enables to answer in $O(\log n)$ time if two given nodes are in the same connected component. This is not what we need for our query however, as we want to know if two nodes are directly connected, so that other techniques should be devised.

1.5.3 Doubling Dimension

Many geometrical structures of the literature consider metric spaces that are doubling. A metric space is doubling if there is a doubling dimension $M > 0$ such that any ball of radius r in that metric space can be covered with M balls of radius $\frac{r}{2}$.

All euclidean spaces, and subsets of euclidean spaces are doubling, with a doubling constant depending only on the dimension of the euclidean space [64].

While some research tries to get rid of the so-called "curse of dimensionality" [48], which refers to algorithms that have a cost that grows exponentially with the doubling constant, we target here applications that operate mainly on low-dimensional spaces. Thus, specifically in Chapter 4, we consider spaces with a constant doubling dimension, and the value of M does not appear in our performance measurements.

1.6 Kinetic setting and Related Definitions

In the previous section, we have considered a set of static nodes. From now on we will consider the kinetic setting, in which the nodes of $\mathcal{V}$ follow some kind of movement. To be more precise about these movements, some additional definitions are needed.

1.6.1 Mobility Models

Movement may manifest in different ways, that is, have different limitations. This is why several mobility models have been defined. We identify two main classes of movement models.

Flight Plan model The first movement model, and probably the most used in research on Kinetic Data Structures (KDS in short, see Section 1.7) is the Flight Plan model. In this model, nodes follow continuous trajectories described by functions that are known in advance. These functions are often supposed to be polynomial or algebraic. While the resulting trajectory has to remain continuous, the function describing the movement of a node may change at some points in time, resulting in a piecewise defined function. The functions are called *flight plans*, hence the name of the model.

Black-Box Model While supposing that the trajectories of the nodes follow some known function is good for certain settings where this is the case, because it results in closer to optimal performances, this may not correspond to all practical settings. Sometimes the positions cannot be known in advance, and only regular samples of the positions are observed.

In the Black-Box model (the term was introduced in [55] for KDSs), we consider that the positions of the nodes change at some instants we will call *time steps* (as if there is a "black box" applying the change). We can suppose, without loss of generality, that the time steps are evenly spaced.

The Black-Box model usually comes with additional constraints. Typically, the positions of the nodes are not allowed to change too much at each time step, imposing nodes to remain at a distance d_{mv} from their previous position.

Number of moving nodes In the Black-Box model, the movement model is also characterized by the number of nodes that are allowed to move at the same time. This distinction is useful, as some algorithms may perform better if it is known in advance that only few nodes move. We consider two extreme situations.

- *Low mobility*: only one node is allowed to move at each time step. In other words, at each time step, it is known that only one node has changed its position since the last time step.

- *High mobility*: all the nodes can change their position at each time step.

Also, both in the Flight Plan and Black-Box model, the time it takes to do computations is supposed to be negligible with regards to the movement speeds. In the Flight Plan model, this means that when the trajectories meet certain conditions (for example, two points crossing, or coming within a certain distance of each other), the algorithm may perform some computations while the nodes are supposed to remain stationary until the computations are finished. In the Black-Box model, this means that between two time steps, as many computations can be done as needed, even if the aim is usually to keep them as low as possible.

1.6.2 Parameters

When the nodes are static, d_{min}, d_{max}, and Φ only depend on the positions of the nodes (see Definition 1.6, page 36). In the kinetic setting, however, there are two ways to extend their definitions: either d_{min} and d_{max} depend on the time (and thus represent the observed maximal and minimal distance at each instant), or they do not (and thus represent constraints on the movements that prevent nodes from getting too close or too far apart from each other). We chose the second, more restrictive definition. Let us denote by $d(u, v, t)$ the distance between the positions of nodes u and v at instant t. We may redefine d_{min}, d_{max}, and Φ in the kinetic setting.

Definition 1.7 (aspect ratio Φ)**.** *d_{min} and d_{max} are two given values such that $\forall t, \forall u, v \in \mathcal{V}, d_{min} \leq d(u, v, t) \leq d_{max}$.*

The aspect ratio is given by $\Phi = \frac{d_{max}}{d_{min}}$.

The value of d_{min} may represent either the size of the nodes, so that two nodes may not get too close to each other, or the precision on the positions, in which case two nodes that are at distance less than d_{min} from each other are considered to have the same position (which is how we will treat this kind of nodes in Section 4.3). The value of d_{max} may represent the size of the world in which the nodes are allowed to move.

In the Black-Box model, it is often imposed that d_{mv} is smaller than d_{min} [19] or than a value close to d_{min} [55].

Let us now look at concepts that are useful when maintaining the data structures presented in Section 1.5 under motion of the nodes.

1.7 Kinetic Data Structures

As we have seen in Section 1.5, a lot of classical geometrical problems can be resolved by constructing discrete structures on sets of nodes in an Euclidean space. For example, finding the Nearest Neighbor, getting an Euclidean Minimum Spanning Tree, or constructing a Yao Graph. Kinetic Data Structures (KDS) aim at maintaining this kind of structures in settings where the nodes move.

1.7.1 Flight Plan Model

As mentioned previously, when nodes move, geometrical structures could be maintained by regularly checking the new positions of the nodes, and at each check, removing the nodes that moved and adding them again to the structure. In contexts where the nodes follow continuous movements, this may lead to problems of oversampling and undersampling, as it is usually inefficient to fix a sampling rate in advance (as mentioned in Section 1.3.1). This strategy may also lead to a lot of useless computations, when the structure is still valid with regards to the desired properties even if some node changed slightly its position.

Originally introduced in [14], the main category of KDS uses the Flight Plan model, providing solutions to these problems by detecting when changes should be made to the target structure, and avoiding as much as possible to do any computation in the meanwhile. As explained in Section 1.6.1, it is supposed that each node has a flight plan, a *continuous* function describing the future movement of the node. While these flight plans can change at some points in time, this is allowed to happen only in such a way that the resulting movement remains continuous.

The basic principle to detect when changes should be made, is to use so-called *certificates*. Certificates are predicates involving a *constant* number of nodes, that all together validate the target structure: when the nodes move, as long as all certificates are true, the structure is guaranteed to still be correct. Let us take for example a set of nodes with positions in one dimension. We can get a structure that maintains the order of the nodes, with a certificate between any two consecutive nodes that ascertains that the first has a coordinate strictly smaller than the second. This complies to the requirement of certificates, because as soon as one of the nodes crosses another one so as to change the order of the nodes, one of the certificates becomes false, which enables the structure to detect when a change should be made (see below for a more comprehensive example).

Certificates can be designed in two similar but different ways [61]:

- We could call *absolute* the most straightforward way of defining certificate: given any set of nodes, the set of certificates validates that the structure is correct. In other words, absolute certificates validate the structure independently of anything else.
- Most KDS papers however use what we could call *incremental* certificates: given a valid structure, the certificates certify that the structure remains valid, as long as the certificates remain true (and of course, as long as the nodes comply to the movement model given by the structure). In other words, incremental certificates validate the structure at the condition that the structure was valid before movement. The KDSs we describe in chapters 3 and 4 use incremental certificates.

In the Flight Plan model, as the future movements of the nodes are known, it is possible to compute, for each certificate when it will become invalid. Certificates can thus be put in a priority queue, where the element with the highest priority is the certificates that is invalidated

first. When a certificate becomes invalid, which is called an *event*, the structure is updated following rules that depend on the certificate; if new certificates are to be created, they are added to the priority queue, and if some certificates are no longer needed, they are removed from the queue.

When there is a change in the flight plan of a node u, then the priority queue is updated with the new invalidation times of the certificates concerning u.

In order to measure the quality of a KDS, four factors may be analyzed:

- *responsiveness*: measures the time it takes to update the structure when a certificate becomes invalid. A KDS is *responsive* if the update needs $O(\text{polylog}(n))$ operations.
- *compactness*: measures the total number of certificates in the KDS. A KDS is *compact* if $O(n\,\text{polylog}(n))$ certificates are needed.
- *locality*: measures the impact of a change in the flight plan of a node, by counting the number of certificates associated with each node. A KDS is *local* if each node is involved in $O(\text{polylog}(n))$ certificates. Note that locality implies compactness.
- *efficiency*: measures the overhead of computation induced by the KDS with regards to the target structure. We call *external event* an event that has an effect on the output of the KDS. On the other hand, we call *internal event* an event that leads to changes in the KDS, but without effect on the output of the KDS. See below for an example of internal and external events. The efficiency of a KDS is the ratio of the number of internal events over the number of external events. A KDS is said to be *efficient* if that ratio is $O(\text{polylog}(n))$.

These performance measures have to be put in perspective with the performance of the priority queue of events. In the best known implementation of priority queues, insertions are performed in constant time, but removals require logarithmic time in the size of the queue [28]. This means that when a new certificate is created, no additional cost is caused by adding it to the priority queue. However, when a certificate that fails is treated, it becomes obsolete so that it has to be removed from the priority queue. The cost of removing that certificate from the priority queue is not taken into account when measuring the cost of updating the structure, that is, the responsiveness. This additional cost is logarithmic in the size of the priority queue, which is the total number of certificates, measured by the compactness of the structure. It follows that even for a compact KDS, unless a complexity better than $O(n)$ can be found for the number of certificates in the priority queue, it is useless in the Flight Plan model to have an upper bound on the responsiveness below $O(\log n)$ (some KDSs, like Kinetic Diamond Delaunay Triangulation [2], may have a constant responsiveness).

The responsiveness of a KDS has also to be put into perspective with the cost of building the structure from scratch. Indeed, if the structure has a bad responsiveness but a short building time, it may be more efficient to systematically rebuild the structure at each event.

Example: 1D, Highest Coordinate An example to make the challenges of KDSs clearer, and introduced in [14], consists in a set of moving nodes in one dimension, on which a KDS has to be built that keeps track of the node that has the highest coordinate (in case of a tie, any node with the highest coordinate can be returned by the KDS). On Figure 1.10, a situation with eight nodes $a, b, \dots, h$ is given, where p_u denotes the position of node u; in this situation, for any instant $t \in [t_0; t_3[$, the KDS needs to identify that d is the node with the highest coordinate, and for any instant in $t > t_3$, the KDS should know that it is the node e.

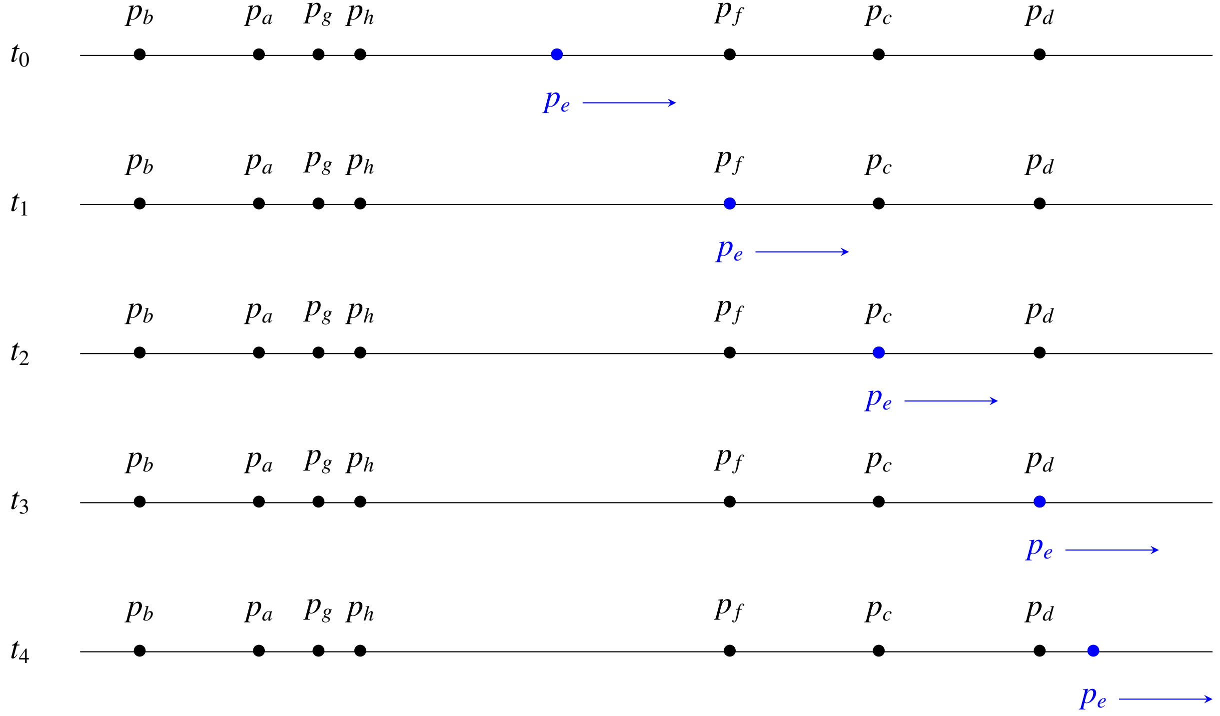


Figure 1.10 – Example of a one dimensional situation with eight nodes $a, b, \ldots, h$, where only e moves. Five noteworthy instants $t_0 < t_1 < t_2 < t_3 < t_4$ are represented.

A naive approach consists in using one certificate for each pair of consecutive nodes, that fails when the nodes cross each other. If we call ORDER(u, v) such a certificate for nodes u and v, Table 1.1 shows the set of certificates used by the KDS for instant t_0 of the example from Figure 1.10.

Table 1.1 – List of naive certificates for the example of Figure 1.10 at instant t_0.

Certificate	**Predicate**
ORDER(b, a)	$p_b < p_a$
ORDER(a, g)	$p_a < p_g$
ORDER(g, h)	$p_g < p_h$
ORDER(h, e)	$p_h < p_e$
ORDER(e, f)	$p_e < p_f$
ORDER(f, c)	$p_f < p_c$
ORDER(c, d)	$p_c < p_d$

An event happens each time a nodes overtakes another one. On Figure 1.10, where only e moves, this happens on instants t_1, t_2, and t_3. As these instants are known in advance in the Flight Plan model, no operation is needed on the KDS in between them. At each event, in order to reestablish the right set of certificates, at most two certificates have to be deleted, and at most two new ones have to be created. For example, at instant t_1, the certificates ORDER(e, f) and ORDER(f, c) have to be removed from the structure, and replaced by ORDER(f, e) and ORDER(e, c). As each of these changes involves an operation on the priority queue of the certificates, which takes $O(\log n)$, and because there is a constant number of them, an event is handled in $O(\log n)$ time, which makes the KDS responsive.

Concerning memory usage, each node is involved in at most two certificates, which makes the KDS local and compact.

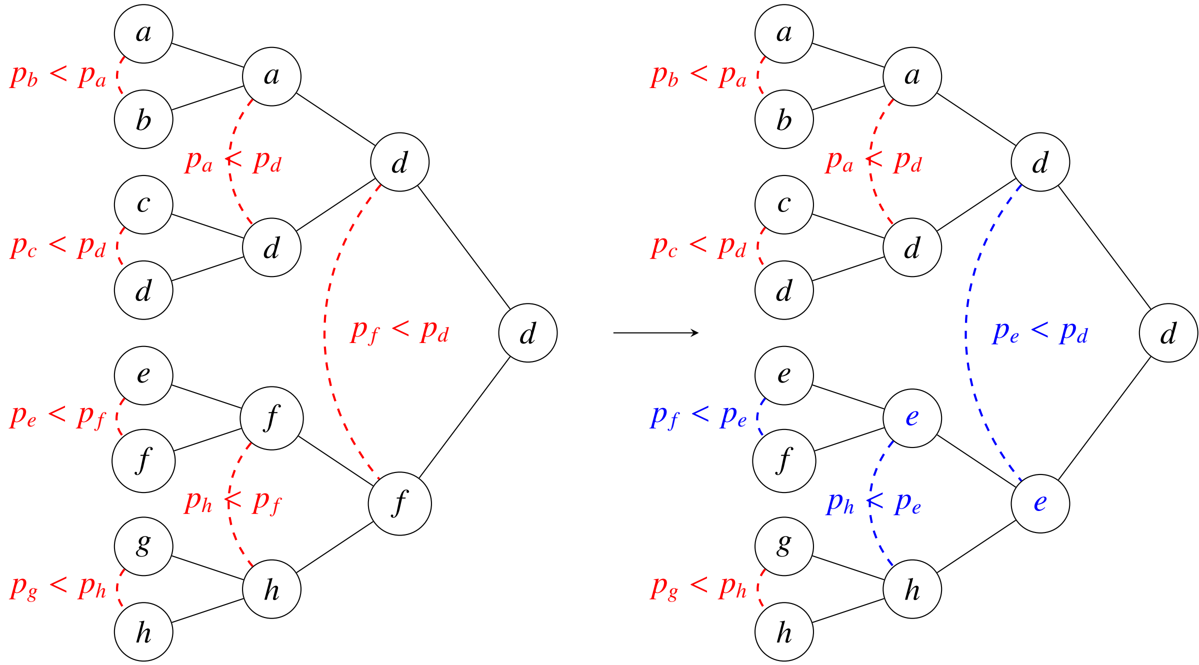


Figure 1.11 – Example of a kinetic tournament; the dashed lines represent the certificates, that compare the positions of two nodes in the same level. On the right is the resulting tournament if node e moves so as to overtake f. The certificates that change accordingly are in blue.

However, the KDS is not efficient. As the output of the KDS is only the node that has the highest coordinate, an external event is an event involving this node; all other events are internal. Thus, among the events shown on Figure 1.10 at instants t_1, t_2, and t_3, only the last one is an external event, as it is the only one that changes the node that has the highest coordinate. In the worst case, e could have started at the left of b, and overtaken all nodes one after the other. This situation would have resulted in $n - 2$ internal events, but only one external event, showing that the KDS is not efficient.

There are however KDSs that validate all quality criteria for this problem, among which the *kinetic tournament*. A kinetic tournament is a tree of *metanodes*, each associated with three values:

- an associated node,
- the “left” child, another metanode,
- the “right” child, also a metanode.

The tree is such that each metanode is associated with the node that has the highest coordinate among the nodes of its children, until only the node with the highest coordinate remains as the root, at level $\log_2 n$. An example of a kinetic tournament, constructed on the nodes of Figure 1.10 at instant t_0, is given on the left side of Figure 1.11, where metanodes are represented by circles, with the associated node written inside, and the children represented as edges of the tree.

Certificates, similar to the certificates of the naive approach, are added between all pairs of *sibling* nodes, that is, the nodes that are associated with metanodes that are children of the same metanode. The certificates of the kinetic tournament on the left of Figure 1.11 (that are also represented by dashed red lines on the figure), are given on Table 1.2

Table 1.2 – List of certificates for the kinetic tournament from Figure 1.11, with the nodes of Figure 1.10 at instant t_0.

Certificate	Predicate
$\textsc{Order}(b, a)$	$p_b < p_a$
$\textsc{Order}(c, d)$	$p_c < p_d$
$\textsc{Order}(e, f)$	$p_e < p_f$
$\textsc{Order}(g, h)$	$p_g < p_h$
$\textsc{Order}(a, d)$	$p_a < p_d$
$\textsc{Order}(h, f)$	$p_h < p_f$
$\textsc{Order}(f, d)$	$p_f < p_d$

An event arises when a node overtakes one of its siblings. On our example, this is what happens at instant t_1, when the node e overtakes f because of the failure of $\textsc{Order}(e, f)$; on the tournament of the left of Figure 1.11, e has to go up one level, and replace f in the next metanode. However, f is also being compared with another sibling h with the certificate $\textsc{Order}(h, f)$, with $p_h < p_e$. Thus e needs to go further up the tournament, until it becomes a sibling of d, which has a higher coordinate. The resulting kinetic tournament is given on the right of the figure, with changes highlighted in blue. In the worst case, a node would have to go up all the $\log n$ levels of the tournament. For each of these changes in the structure, a constant number of certificates are deleted and created, each one incurring $O(\log n)$ computations for updating the priority queue, resulting in a total of $O(\log^2 n)$ computations per event. Thus, the KDS is responsive[14].

Each node is involved in at most $O(\log n)$ certificates, thus the KDS is local and compact.

When a node overtakes another one, it does not necessarily trigger an event, as this happens only with siblings. On our example, after e overtook f, with the kinetic tournament on the right of Figure 1.11, no event arises on instant t_2 when e overtakes c, as those two nodes are not siblings. On instant t_3 however, there is an external event. The proof is trickier than that, but this leads to show that the KDS is efficient.

This example shows that one usually has to be "clever" when designing a KDS that meets all the quality criteria. If a KDS is found that fails on some quality factors, there is usually no a priori and easy way to know if another one can be designed that validates all criteria. This is an additional reason that makes KDSs challenging.

1.7.2 Black-Box Model

Some research papers [42, 55] consider that it is more realistic to use the Black-Box model (see Section 1.6.1) than the Flight Plan model, as the nodes's trajectories are usually described as consecutive measures of their positions, and not as continuous functions of time. In that case, at each time step where the positions of the nodes may change, the structure has to be updated, resulting in some computations.

Most KDSs for the Flight Plan model can be easily adapted to the Black-Box model, provided that linear flight plans are allowed (that is, KDSs where the nodes can move on a line). However, this generally leads to poor performances. Let us consider a valid structure at time t_i, and let t_{i+1} be the next time step. In order to get a valid structure for the new positions at time t_{i+1}, we can give to each node u the linear flight plan that goes from u's position at time t_i

[14]Note also that this is better than the naive approach consisting in rebuilding the tournament from scratch, which takes $O(n)$ time

to its next position at time t_{i+1}. The total cost to update the structure at each time step is then the sum of the costs it takes to compute the trajectories, to reschedule the certificates accordingly to the new failure times, and the cost of updating one certificate multiplied by the total number of failing certificate. Most articles that present KDSs in the Flight Plan model give an upper bound on the total number of events the KDS may have to process, which is usually quadratic. With $O(n^2)$ events on a compact and responsive KDS, we get an upper bound of $O(n \operatorname{polylog} n + n^2 \operatorname{polylog} n)$ on the update time per time step (when supposing that it takes a constant time to compute the failure time of a certificate), which is usually worse than rebuilding the structure from scratch. This kind of upper bound is not tight at all, mostly because the number of events are given for whole executions of the flight plans. Sadly, bounds on the number of events for small increments of movement like in the Black-Box model are rarely given in the literature on the Flight Plan model. Efficiently converting a KDS from the Flight Plan model to the Black-Box model is thus usually not immediate.

With this in mind, some results in the Black-Box model drop the certificates altogether, and systematically do some computations at each time step. While it is proven in [19, 50][15] that those computations are bounded, this may still lead to superfluous computations, when the nodes move in such a way that the structure is checked an rebuilt identically.

Other results use the Black-Box model in conjunction with certificates [42, 55]. While certificates need still to be checked regularly to see if they failed, this allows to separate the cost of verifying the structure and updating it, which could reduce the cost of the computations when the structure does not need to be updated.

Having certificates thus opens up the possibility to use prediction schemes like in [111]. However, maintaining a priority queue as in the Flight Plan model has an additional cost for each time step, caused by removing each certificate that failed. As discussed on page 41 (Section 1.7.1), removing an element from the priority queue has a cost that is logarithmic in the number of elements in the queue. As it is difficult to guarantee that only a limited number of certificates fail at each time step, this adds a cost of $O(n \log n)$ at each time step for compact KDSs.

Nonetheless, we believe that KDSs that use certificates have an advantage over structures that do not use them. While future motions are not known in the Black-Box model, in most practical settings, the movements of the nodes are somewhat predictable when knowing their previous movements. In a way very similar to Dead-reckoning (see Section 1.3.2), an estimate of a node's velocity can be computed, which can be used to predict the failure time of its certificates, as in the Flight Plan model. Thus, the cost of verifying the validity of the certificates at each time step is replaced by the cost of verifying if the prediction is still within some predefined bound, like Dead-reckoning's threshold. In most cases, this should be cheaper, as we expect this to be slightly less computer intensive in practice than the verification of the validity of a certificate (which involves recomputing the predicate associated to that certificate according to the new positions of the nodes), and because the total number of certificates may be an order of magnitude higher than the number of (moving) nodes. Simulations should be conducted to verify this hypothesis, and this is thus a topic for future research.

In Chapter 3 and Chapter 4, we thus focus on the Black-Box model, and provide structures that use certificates.

In the following sections, some results showing how to maintain the structures presented in Section 1.5.2 for moving nodes will be presented. The performances of these solutions can be found in Table 1.4 and Table 1.5 for easier reference and comparison.

[15] Actually, in [19], two solutions are proposed, one of which builds a "sub-KDS" in the Flight Plan model similarly as described in the previous paragraph, and that thus uses certificates. This solution, however, performs worse than the second solution, that doesn't use certificates.

1.7.3 Tree Structures

Several results exist to maintain classical tree structures for moving nodes. Recall that when measuring the performance of a query, k denotes the size of the returned set.

In [15], a kinetic structure is maintained, that allows to construct range trees when movements follow constant degree algebraic flight-plans. It takes $O(n \log^{d-1} n)$ space and has a depth of $O(\log n)$. Range queries can be answered in $O(\log^d n + k)$ (with k the number of nodes returned by the query), and a certificate takes $O(\log^d n)$ to be fixed, for a total of $O(n^2)$ events. In [4], a solution in two dimensions improves upon this for linear flight plans, reducing the space complexity to $O(\frac{n \log n}{\log \log n})$, and with an amortized query time of $O(\log^2 n)$.

Standard kd-trees are not efficient when the nodes move [3], thus some variants have been proposed to have equivalent structures in the kinetic setting. In [7], two variants called pseudo kd-tree and overlapping kd-tree are proposed for two-dimensional pseudo-algebraic flight-plans. Both structures can be used to answer to range queries in $O(n^{1/2+c} + k)$ for a small constant $c > 0$, with certificates that take at most $O(n)$ to be handled, but only $O(\log n)$ on average. In [3], two additional variants are proposed for constant degree algebraic flight-plans. First, rank-based kd-trees use $O(n)$ memory (with $O(1)$ certificate per node) and processes range queries in $O(n^{1-\frac{1}{d}} + k)$ time, events being handled in $O(\log n)$. Rank-based longest-side kd-trees are tailored for two-dimensional ε-approximate range queries, where additional nodes outside the range can be returned[16]. Its size is also linear, answers to the queries in $O(\frac{\log^2 n}{\varepsilon})$ time, and handles events in $O(\log^2 n)$.

Quadtrees are proposed in [50], in the Black-Box model, where positions are updated at regular time steps. It is here supposed that nodes move no more than d_{mv} distance units at each position update, and that no more than ρ nodes can be situated in a ball of radius d_{mv}. The structure does not use certificates, and is updated in time $O(n \log \rho)$ per time step. The quadtree is compressed, and has a linear size. As mentioned in Section 1.5.2, we will prove in Section 4.1.3 that a quadtree can be used to answer to $\textsc{CloseNodes}_u(r)$ queries in a time complexity of $O\left(b_u\left(r + r\sqrt{2}\right)\log \Phi\right)$, where $b_u(x)$ is the number of nodes at distance at most x from u (and with Φ the aspect ratio defined in Definition 1.7, p.39). If we suppose that $b_u(r \cdot O(1)) = O(k)$, then we have a query time of $O(k \log \Phi)$. One of the drawbacks of this structure, is that it is completely rebuilt at each time step, even if very few nodes moved, so that the update complexity does not improve in the low mobility setting compared to the high mobility setting.

A summary of the different kinetic tree structures is given at the top of Table 1.4 and in Table 1.5, where the query time designates the time needed to answer a $\textsc{CloseNodes}_u(r)$ query. We can see that generally, range trees have better query times but higher memory costs than kd-trees. The correction of certificates is usually logarithmic for all the tree structures.

1.7.4 Spanners

As we have seen in Section 1.5.2, the time complexity of $\textsc{CloseNodes}_u(r)$ on a spanner of stretch factor s is $O(b_u(sr) \cdot \delta)$, with $b_u(sr)$ the number of nodes in the ball of radius sr centered on node u, and δ the maximum degree of the spanner. For ease of notation, we will use $s' = s - 1$ (some fractions with a denominator depending on s will tend to infinity when s' tends to zero, that is, when the stretch factor tends to the ideal $s = 1$).

As mentioned previously, Delaunay triangulations lead to good performances in the plane, but to quite poor performances in higher dimensions. This could explain why all the previous

[16] For a query region Q with diameter $\text{diam}(Q)$, the query should return a superset of $Q \cap \mathcal{V}$ so that any returned node u satisfies $d(u, Q) \leq \varepsilon \cdot \text{diam}(Q)$.

work on KDSs for Delaunay triangulations we found in the literature focus only on positions in two dimensions.

As explained in [14], Delaunay triangulations are easy to kinetize, requiring only one certificate per edge. As the number of edges is $O(n)$ in two dimensions, the memory cost is $O(n)$. The degree being $O(n)$ at worst but $O(1)$ on average, the cost of a $\text{CloseNodes}_u(r)$ query is $O(b_u(sr) \cdot n)$ at worst, and $O(b_u(sr))$ on average. An event can be handled in $O(\log n)$ time [60]. The total number of events is nearly cubic: $O(n^2 \lambda(n))$, with $\lambda(n)$ a function that is nearly linear in n in the Flight Plan model [131].

A variant with better performances is presented in [2]. A Delaunay triangulation is constructed, but using a diamond-shaped convex distance function instead of the euclidean distance. The resulting graph is proven to also be a spanner in the plane, with a stretch factor that is a parameter of the KDS. The structure has a size that depends on the given stretch factor s, but can be considered linear ($O(n/s'^2)$) like the euclidean Delaunay triangulation. However, with constant degree polynomial flight plans, the number of events is nearly quadratic, and an event incurs only $O(1)$ computation time to update the spanner, which is better than Delaunay triangulations. Recall however that in the flight-plan model, events are put in a priority queue that has to be updated at each event, and each time a new certificate is created. Thus, the total computation time at each event is $O(\log(n/s'^2))$.

In [19], the Black-Box model is analyzed. When dealing with Delaunay triangulation, it has been noticed that rebuilding the graph (which takes $O(n \log n)$) after each movement is actually hard to outperform [127]. However, in [19], a method is proposed that takes $O(\Phi^2)$ (Φ has been defined in Definition 1.7, page 39) to update the triangulation at each time step, under similar assumptions as in [50]: nodes move no more than d_{mv} distance unit at each position update, and no more than ρ nodes can be situated in a ball of radius d_{mv}. Depending on the value of Φ with respect to n, this can be better. Note also that the structure is updated by treating the nodes one after the other; this means that if it is known in advance that only one node may move at each time step (as explained in Section 1.6.1), the structure may be updated faster: in the paper, it is said that the update of one node is linear in its degree, meaning that the cost of updating the structure in this mobility model of only one moving node is in $O(n)$.

Attempts have been made to distribute Delaunay Triangulations [139]. We may also mention stable Delaunay graphs, subgraphs of Delaunay graphs, that are studied in [6], however it is not mentioned whether they are spanners.

In [120], kinetic versions of the Yao and Theta graphs are given for constant degree polynomial flight plans in two dimensions. Both use $O(n)$ space, and handle events in $O(\log n)$ amortized time. As the degree of those graphs is $O(n)$ at most, but $O(1)$ on average, we get a query time for $\text{CloseNodes}_u(r)$ of $O(b_u(sr) \cdot n)$ at worst, but $O(b_u(sr))$ on average. The maintenance of the Theta graphs leads to a nearly quadratic number of events: $O(n \cdot \lambda(n))$, with again $\lambda(n)$, a function that is nearly linear in n. Yao graphs are a little bit worse with a nearly cubic number of events.

In [119], Theta graphs are kinetized in higher dimensions for constant degree polynomial flight-plans. The structure takes $O(n \log^d n)$ space, and $O(n^2)$ events are handled each in time $O(\lambda(n) \log^{d+1} n)$ on average.

In [1], a variant of Theta graph is proposed, also for constant degree polynomial flight-plans, that is based on what the authors call a Cone-Separated Pairs Decomposition. As in [2], the stretch factor s is a parameter of the structure. The resulting spanner has $O(n/s'^{d-1})$ edges, which can be considered linear like the Theta graph, but a maximum degree of $O(\log^d n)$, resulting in a better worst-case complexity for the $\text{CloseNodes}_u(r)$ query of $O(b_u(sr) \cdot \log^d n)$. The kinetic structure takes $O((n/s'^{d-1}) \log^d n)$ memory, and events are handled in $O(\log^{d+1} n)$ time.

A summary of the different kinetic spanners is given on the bottom of Table 1.4, and in Table 1.5. Note that s is used as the stretch factor of the given structure, that may be different in each solution. As explained previously, for easier notation of the performances of some spanners, we use $s' = s - 1$. Again, the query time designates the time needed to answer a $\textsc{CloseNodes}_u(r)$ query. We can see on the table that spanners have low memory costs, like kd-trees. In terms of query time, let us note that $O(b_u(sr))$ is close to $O(k)$ on average, but higher in the worst case; it then becomes apparent that queries on spanners are not as effective as on trees in the worst case, but close to optimal on average. For better readability, we suppose in Table 1.4 and Table 1.5 that $b_u(r \cdot O(1) + O(1)) = O(k)$, so that this ball size is not shown in the results of query time. The correction of certificates, like for trees, is often logarithmic, albeit often only once amortized. In general, it seems spanners are better than trees on average, but not as good in the worst case.

1.7.5 Navigating Nets

We have presented navigating nets in Section 1.5.2. These tree-like structures have been maintained under motion in previous works.

In [42], a model similar to Black-Box is used. A relaxed version of the navigating net is analyzed in terms of number of combinatorial changes made to the structure and number of created and deleted certificates, with regards to the strict version of the navigating net.

In [55], a full KDS analysis is done on a navigating net, both in the flight-plan model and in the Black-Box model. The navigating net (enriched with neighbors as described in Section 1.5.2) is a spanner of maximum degree $O(\log \Phi / s'^d)$ (recall Definition 1.7, stating that $\Phi = \frac{d_{max}}{d_{min}}$). Thus, the $\textsc{CloseNodes}_u(r)$ query can be answered in $O(b_u(sr) \cdot \log \Phi / s'^d)$. The structure has a linear size of $O(n/s'^d)$, with each node involved in at most $O(\log \Phi / s'^d)$ certificates. In the Flight Plan model (with pseudo-algebraic flight plans), an event incurs $O(\log \Phi / s'^d)$ changes to the navigating net, resulting in events handled in $O(\log \Phi / s'^d + \log(n/s^d))$ time when including updates to the event queue, with a total number of events of $O(n^2 \log \Phi)$. In the Black-Box model, assuming that n nodes can change their level in the navigating net hierarchy, each time step involves $O(n \log \Phi)$ computations. We will see in Section 4.3.1 that if there is only a single node that may move at each time step, then $O(\log \Phi / s'^d)$ computations per time step are sufficient to update the structure. Note that in [55], the same definition of aspect ratio is used as in Definition 1.7 (p.39), with d_{min} being the minimal distance between any two nodes throughout the execution normalized to 1.

1.7.6 Conclusion on Centralized KDSs

This thesis focuses on the Black-Box model. A summary of the different KDSs for the Flight Plan model can be found in Table 1.4[17] (with, at the upper half, the tree-based solutions, and at the lower half, the spanner-based solutions). We have seen in Section 1.7.2, that these solutions can be converted to the Black-Box model, with an upper bound on the time complexity for each time step equal to the correction cost for one certificate multiplied by the total number of events. As can be seen on Table 1.4 no solution yields better results than $O(n^2 \log n)$ computations per time step. While the query times for those structures (usually $O(\log n + k)$ for trees and $O(kn)$ for spanners) are acceptable, which is not a surprise as queries are independent of movement models, the update times leave room for improvement, so that solutions specifically tailored for the Black-Box model are preferred.

[17]Note, as explained previously, that we suppose in Table 1.4 that $b_u(r \cdot O(1) + O(1)) = O(k)$.

Table 1.3 – Summary of the notations used in performance measurements

n	Total number of nodes.
r	Parameter of the $\textsc{CloseNodes}_u(r)$ query.
k	Number of nodes returned by the query.
Φ	Aspect ratio (see Definition 1.6 p.36).
s'	With s the stretch factor of a spanner (see Definition 1.5), $s' = s - 1$.
$\lambda(n)$	Maximum length of a Davenport-Schinzel sequence: this function is nearly linear in n in the Flight Plan model.
ρ	Maximal number of nodes that can be situated in a ball of radius d_{mv} (with d_{mv} the maximal distance a node can move at each time step of the Black-Box model.)

Results that are written specifically for the Black-Box model are shown on Table 1.5[18] (note that the last line shows the result of one of our contribution, which will be presented in Section 1.8). The correction cost refers to the complexity of the updates to the structure at each time step (recall, from page 39, that in the low mobility setting only one node may move at each time step, and all nodes may move in the high mobility setting).

While memory usage is the same for all KDSs, DefSpanner performs the best in query time and in update time for the low mobility setting. For the high mobility setting, the best is not as clear, as the Kinetic Compressed Quadtree can be better if $\rho < \Phi$. While the DefSpanner seems to perform quite well, a more in depth analysis shows that, in some situations, some nodes may need to drastically change their level in the Navigating Net hierarchy, which is responsible for the $\log \Phi$ factor in the update complexity. We try to address this issue in Chapter 4.

1.7.7 The Case of Distributed KDSs

While KDSs are mainly studied as centralized structures, the fact that certificates are *local* conditions that *globally* validate the structure, makes them an interesting opportunity for distributed systems. Local predicates aimed at maintaining global properties are nothing new in the field of distributed systems, but the various tools and measurement techniques introduced in the field of KDS can be beneficial to analyze distributed algorithms.

In the distributed setting, we suppose that the changes in position emanate from the nodes themselves. In the Flight Plan model, this means that the change of flight plan of a node u is an information local to u; any other node would need to receive a message from u in order to know about this change of flight plan. In the Black-Box model, at each time step, each node becomes aware of its new position but not of that of the other nodes. It follows that in the Black-Box model, a node that needs to know if a certificate becomes false needs to receive a message from each of the nodes whose position is involved in that certificate. We suppose that one message is enough for a node to sent its position to another node, which is realistic, as we have supposed previously that a position takes a constant amount of registers in memory.

Sadly, adapting centralized KDSs in the distributed setting, by simply making so that each node maintains the certificates it is involved in, adds complications. Firstly, as the memory is distributed among the nodes, some information required to handle an event might not be directly accessible: if a node needs to retrieve the position of another node without knowing its identifier, a research of that node is needed, the performance of which depends largely on the connection graph. Also, while the number of events does not change (so that the efficiency

[18] Again, with $b_u\,(r \cdot \mathcal{O}(1) + \mathcal{O}(1)) = \mathcal{O}(k)$.

Table 1.4 – Comparison of several structures in the Flight Plan model (see Table 1.3 for a summary of the notations)

	Space	Total memory cost	Query time	Correction cost for one certificate	Total number of events
Kinetic external range-tree [4]	2D	$O(\frac{n \log n}{\log \log n})$	$O(\log n + k)$	$O(\log^2 n)$ amortized	$O(n^2)$
Pseudo and overlapping kd-tree [7]	2D		$O(n^{1/2+c} + k)$ ($c > 0$ a small constant)	$O(n)$ worst case $O(\log n)$ amortized	$O(n^2)$
Rank-based longest-side kd-tree [3]	2D	$O(n)$, $O(\log n)$ certificate per node	$O\left(\frac{\log^2 n}{\varepsilon} + k\right)$ (with an approximation factor of ε)	$O(\log^2 n)$	$O(n^3 \log n)$
Range tree [15]	dD	$O(n \log^{d-1} n)$	$O(\log^d n + k)$	$O(\log^d n)$	$O(n^2)$
Rank-based kd-tree [3]	dD	$O(n)$, $O(1)$ certificate per node	$O(n^{1-1/d} + k)$	$O(\log n)$	$O(n^2)$
Kinetic Delaunay Triangulation [60]	2D	$O(n)$	$O(k \cdot n)$ at worst, $O(k)$ on average	$O(\log n)$	nearly cubic: $O(n^2 \lambda(n))$
Kinetic Diamond Delaunay Triangulation [2]	2D	$O(n/s'^2)$	$O(k/s'^2)$ on average	$O(\log(n/s'^2))$ ($O(1)$ without the priority queue)	nearly quadratic: $O((n/s'^2) \cdot \lambda(n))$
Theta graph [120]	2D	$O(n)$	$O(k \cdot n)$ at worst, $O(k)$ on average	$O(\log n)$ amortized	nearly quadratic: $O(n\lambda(n))$
1-Theta-Yao graph [119]	dD	$O(n \log^d n)$, $O(1)$ certificate per node on average	$O(k \cdot n)$ at worst, $O(k)$ on average	amortized nearly logarithmic: $O((\lambda(n)/n) \cdot \log^{d+1} n)$	$O(n^2)$
CSPD-based spanner [1]	dD	$O((n/s'^{d-1}) \cdot \log^d n)$, $O(1/s'^{d-1})$ certificate per node	$O(k \cdot \log^d n)$ at worst, $O(k/s'^{d-1})$ on average	$O(\log^{d+1} n)$	$O(n^2/s'^{d-1}))$
DefSpanner [55]	dD	$O(n/s'^d)$, $O(\log \Phi/s'^d)$ certificate per node	$O(k \log \Phi/s'^d)$	$O(\log \Phi/s'^d + \log(n/s'^d))$ ($O(\log \Phi/s'^d)$ without the priority queue)	$O(n^2 \log \Phi)$

measurement is the same), and while the local and global memory costs can be of interest in the distributed setting (so that compactness and locality are still relevant), the computation time, that is, the responsiveness as measured in the centralized setting, is usually no longer relevant when considering distributed KDSs. As some computations require messages to be sent, which usually takes an order of magnitude longer than to do local computations, the time performances of distributed algorithms are, in most cases, measured in terms of number of messages and/or number of communication rounds (see Section 1.2), and not in terms of local computations.

One naive solution to convert a centralized KDS to the distributed setting, could be to designate one specific node as a coordinator, that would centralize all computations. The coordinator is tasked to maintain all the certificates of the KDS, and thus needs to know the exact location of all the nodes. In the Flight Plan model, this means that the coordinator needs to be informed of any change in flight plan, and in the Black-Box model, this means that every node needs to send the coordinator its new position at each time step. This technique can actually lead to a surprisingly low number of total exchanged messages: regardless of the total number of certificate, this solution leads to a total of $O(n)$ messages per time step in the Black-Box model (as we supposed that a position can be sent using one message). In comparison, if a KDS is distributed without a coordinator, each node u would need to send its position to each node v such that u and v are involved in a common certificate. Even if the KDS is local, this would lead to $O(n \log n)$ messages, as each node is involved in $O(\log n)$ certificates. However, the solution using a coordinator is not satisfactory, because it overloads one node: the local memory usage for a node and the number of incoming message for one node are $\Omega(n)$, while they can be expected to be $O(\log n)$ in good distributed solutions.

To the best of our knowledge, only one attempt has been made specifically to adapt a kinetic data structure to a distributed setting. In [56], the results of [55] are reused in an asynchronous distributed setting; the same set of certificates is used, each node maintaining a local copy of the certificates that it is involved in. The correction of a certificate failure induces communications between the nodes in order to repair the structure. Two levels of proximity between children and parents of the tree are given, giving the structure extra flexibility, so that some computations can be made in the background when nodes move. The algorithm results, for each node, in a number of connections and a local memory cost of $O(\log \Phi)$. Updates require $O(\log \Phi)$ messages per units of distance a node moves. If we suppose in the Black-Box model, that nodes move only of one distance unit per time step, this results in $O(n \log \Phi)$ messages per time step. The nodes are implicitly treated one after the other, and it is unclear whether this has an effect on the performances, leaving room for investigation.

1.8 Contribution

In this thesis, we have worked on different methods to allow participants of networks of moving points to answer to queries related to their distance, with guarantees on the results, and while minimizing their bandwidth consumption.

1.8.1 Distance Estimation

When two nodes are connected, as we have seen in Section 1.3.2, Dead-reckoning allows them to keep each other informed about their respective state while sending few messages. While the error induced by Dead-reckoning can be measured by different means [8, 147], Dead-reckoning usually aims at bounding the *absolute error* on the nodes' positions. Some research has already been conducted on using Dead-reckoning with some sort of relative threshold [30, 132],

but to the best of our knowledge, no extensive attempt has been made to analyse its effect on the number of exchanged message, and none has been made when targeting only the distance estimation.

In Chapter 2, a synchronous deterministic algorithm is proposed that allows two connected nodes u and v to answer to the ε-Distance query (see Definition 1.1, p.17).

While the proposed algorithm to estimate distances is valid for any movement on a synchronous network, we conduct an extensive theoretical analysis on random movements. Those movements are of two types: either discrete (a random walk on a grid) or continuous (at each communication round, the new position of a moving node may take any value in a circle centered on its previous position). We prove that on these movements, the number of exchanged messages is optimal up to a constant factor (see Section 2.1.3).

These theoretical proofs are complemented with experimental analyses on traces of a real online game, showing that the algorithm behaves better in practice than a strategy that is often used by game developers, consisting in sending updates at regular time intervals.

1.8.2 Kinetic Data Structures for Proximity Queries

We have seen, throughout the previous sections, that the distributed setting adds complications with regards to the centralized setting. We have also seen, when a geometrical data structure has to be maintained on a set of nodes, that making the structure dynamic adds some difficulties, and we have then seen that making the structure kinetic instead is even more difficult.

The work presented in Chapter 3 and Chapter 4 are attempts to go one step further still, by studying *kinetic data structures in a distributed setting*, as well as some related structures in other less stringent settings. Consistency of distance estimation between nodes is considered a separate problem that can be ensured with the algorithm of Chapter 2 or other Dead-reckoning approaches; we thus suppose in Chapter 3 and Chapter 4 that two connected nodes have means to know exactly the distance between each other.

In Chapter 3, we consider a set of *distributed* nodes that move in a *one-dimensional* space, and where each node needs to be able to answer to the CloseNodes$_u(r)$ query for a *fixed* radius r. The nodes move at a constant speed smaller than one half of r distance units per time step. We give a *synchronous* algorithm, denoted by $\mathcal{A}_{f1d}$, that ensures that each node is connected with all nodes that are at distance r from it, where r is given as entry to the algorithm, and is known and identical for all nodes. Thus, an answer to the CloseNodes$_u(r)$ does not require any message nor computation. The structure is maintained in the *high mobility setting* and *Black-Box model* using a constant amount of communication rounds per time step. The number of connections and the local memory cost for a node is $O(b_{max}(r)) = O(k)$ (with k the number of nodes returned by the query, and $b_{max}(r)$ the maximum number of nodes that can be at distance r from a node). In comparison, [56] uses $O(\log \Phi)$ memory space for each node, and as the structure relies on a navigating net, it can answer to CloseNodes$_u(r)$ queries by using $O(\log r)$ communication rounds (which will be proven in Section 4.2.3).

In Chapter 4, we also aim at the CloseNodes$_u(r)$ query, but this time with a *variable* r. Each time a node starts a query, it may give a different value for r.

In Section 4.3, we present results in the *centralized* case, developing the results of [55].

In addition to the CloseNodes$_u(r)$ query that utilizes the spanner property of our navigating net, resulting in $O(b_u(sr) \cdot \log \Phi)$ time to answer to the query, we propose another search algorithm that utilizes the tree structure and results in $O(\log r + b_u(r(2b-1)+1)) = O(\log r + k)$ computations (with r the query range, k the size of the set returned by the query, and $b_u(x)$ the

number of nodes at a distance smaller than x from u). This search algorithm can also be used on the structure from [55].

When the nodes move, the DefSpanners from [55] can lead to situations where a node of low level needs to go up all the hierarchy of the navigating net, resulting in up to $O(\log \Phi)$ changes in the level of the node. We draw upon [48] to avoid this situation, and give a structure that we call *constrained navigating nets* on which we prove that a node may change its level only a constant number of times per time step for small movements. The structure uses $O(n)$ memory space. We give an algorithm, $\mathcal{A}_{cnnptr}$, for the *low mobility setting* in the *Black-Box model* (where only one node moves at each time step), that updates the structure in $O(\log \Phi)$ computations per time step. This performance is similar to DefSpanners, as we prove in Section 4.3.1. See Table 1.5, for a comparison of the different centralized Black-Box solutions (presented throughout Section 1.7) for the CloseNodes$_u(r)$ query. This table shows that our solution is competitive with the other solutions, except for the high mobility setting, where additional work is required to get a better result than the literature.

We also present a *distributed* algorithm, $\mathcal{A}_{cnndist}$, to maintain constrained navigating nets in *synchronous* networks, in the *low mobility setting* and in the *Black-Box model*. This is the first distributed algorithm in this setting. It uses a constant amount of communication rounds per time step, and while it uses $O(n)$ memory for one node at worst, each node needs to track only the positions of $O(\log \Phi)$ other nodes, and the total memory usage of all nodes combined is $O(n)$.

Another advantage of our constrained navigating nets, is that while the performance is the same as [55] in the centralized low mobility setting, we believe that it is possible in the future, to achieve a linear update time of $O(n)$ computations per time step. This would be done by taking into account the movements of nodes move one after the other, and executing the low mobility algorithm on each node, in a specific order.

Table 1.5 – Comparison of cenralized solutions for the CloseNodes query in the Black-Box model

	Total memory cost	Query time	Low mobility correction cost	High mobility correction cost
2D Black-Box Delaunay [19]	$O(n)$	$O(k \cdot n)$ at worst, $O(k)$ on average	$O(n)$	$O(\Phi^2)$ or $O(n \log n)$
Kinetic Compressed Quadtrees [50]	$O(n)$	**2D**: $O(k \cdot \log \Phi)$	$O(n \log \rho)$	$O(n \log \rho)$
DefSpanner [55]	$O(n)$	$O(k \cdot \log \Phi)$ or $O(\log r + k)$	$O(\log \Phi)$	$O(n \log \Phi)$
Chapter 4: $\mathcal{A}_{cnnptr}$	$O(n)$	$O(k \cdot \log \Phi)$ or $O(\log r + k)$	$O(\log \Phi)$	**Conjecture**: $O(n)$

Chapter 2

Distance Estimation

Estimating distances between nodes can be very useful. In many contexts, a node is not interested in knowing the exact state of far away objects; thus, having inexpensive methods for estimating the distances between nodes can help to reduce message exchanges. For example, in DVEs (see Section 1.3), nodes usually interact only with other nodes that are close to them in the virtual world. In addition, in some application-specific cases, distances may be important, for example when an autonomous vehicle needs to get some information on close-by vehicles, or when implementing a spell in a game that heals all allies within a certain range. To the best of our knowledge, no distributed algorithm has been proposed to solve the problem of estimating the distance between users of a DVE, and in the context of mobile objects, most distance estimation methods consider that the objects do not know their own position, and use other means to estimate distance (for example, in [23], external pieces of hardware are used to estimate distance to similarly equipped close-by nodes, and in [104], the number of intermediate nodes needed for message routing is used in a wireless setting to estimate the physical distance between nodes).

The objective of this chapter is *to provide a solution allowing two interconnected nodes aware of their own exact position, but not of the other, to estimate the distance between them, with a condition on the relative error, while guaranteeing that the use of bandwidth is as small as possible*. In particular, it has to be bounded against an ideal algorithm that would send a minimum number of messages, based on a perfect knowledge of the states of all nodes.

Thus, this chapter looks into a problem that is tangent and complementary to the following chapters. Here, we find means for any pair of nodes that are connected to each other (that is, that are able to send messages to each other) to estimate the distance separating them. In the other chapters, we suppose nodes can already access distances, and we find means for them to connect in meaningful ways according to their distance (that is, we aim at minimizing the number of connections, while making it possible for each node to get the set of close-by nodes, see chapters 3 and 4). Thus, the solutions presented in this chapter could be used alongside the algorithms from the other chapters, as a means of estimating distances between two connected nodes.

2.1 Introduction

2.1.1 Related Work

We have seen in Section 1.3, and in particular, in Section 1.3.2, ways for nodes of a DVE to estimate the state of other nodes. Among these works, we identify two main articles related to our objective.

In [102], two techniques are proposed. First, *local-lag* reduces short-term inconsistencies, at the cost of less responsiveness: a delay between the time an operation is issued and the time when the operation becomes effective is added. Secondly, `timewarp` is proposed, an algorithm to ensure consistency. In this algorithm, each node remembers all previous operations and the time at which they were issued. If an operation is received by a node too late, the node rewinds the state of the world, immediately recomputing the current state, using all needed operations. These operations are user initiated, thus, the number of messages is proportional to the number of nodes, and to the length of time. While these solutions garantee high consistency, they are based on heavy communication to ensure it, and thus are not ideal when looking for bandwidth efficient solutions.

In [91], Dead-Reckoning is used to compensate for latencies and message losses on the network. TATSI, the average spatial error on nodes' positions over a time interval, is estimated with no latency or loss of message. Then, under the assumption of a constant acceleration, latencies and message losses are added to the model, and it is shown that the same TATSI can be obtained by lowering the dead-reckoning threshold (thus making DVE nodes send more messages than without latency and message losses). While this article analyses the consistency of the estimations with regards to the number of message exchanges, the basic idea is to compensate for network issues with additional messages. Again, this solution is thus not what we are looking for in terms of bandwidth usage.

To summarize, solutions from the literature are very consuming in term of messages and/or target an *additive bound* on the error. By contrast, this chapter focuses on bounding the *relative error* on distances and keeping the number of message exchanges low.

2.1.2 Limitations and Hypotheses

We use the Black-Box model: the movement of the nodes is synchronous, and we consider that nodes may move (or more generally, that their movement is taken into account) only at specific instants, that we call *time steps*. We also suppose that the nodes are in a synchronous network (see Section 1.2), in such a way that each time step can be in turn divided in subrounds that we will call *communication rounds*. We suppose that latency and clock deviations are small enough so that if a message is sent during a communication round, the recipient of the message is guaranteed to have received it at the start of the next communication round.

We assume that computation time is not an issue inside a communication round: in each communication round, a node may do as many computations, and send as many messages as needed.

Each node is associated with a position, and has access to its exact value (but the node does not have access to other nodes' positions). The message size is enough for nodes to send their current position with one message. Also, we assume that initially, each node knows the exact positions of the other nodes.

2.1.3 Contribution

As seen in Section 1.3.2, when two nodes use Dead-reckoning, each of them knows the position where the other nodes estimate it to be. Thus, a node can send updates only when the tolerated error between its actual position and its estimated position is exceeded, making it an optimal bandwidth strategy. On the other hand, since none of the two nodes knows the actual distance between them, none of them can determine the exact error over the estimated distance, making distance estimation a much harder problem.

We consider deterministic algorithms that allow each node to estimate, at any time, the distances between it and the other node, while having a guarantee on the error. The metric we use is the relative error given in Equation 2.1, where, at each instant t, $d_{act}(t)$ denotes the actual distance between two nodes, and $d_{est}(t)$ denotes their estimated distance,

$$\text{relative error } = \frac{|d_{act}(t) - d_{est}(t)|}{d_{est}(t)}. \tag{2.1}$$

We provide consistency by making sure this error measurement never exceeds ε, the maximum tolerated relative error for any pair of nodes, while minimizing the number of exchanged messages.

That is, Equation 2.2 must always hold, for every pair of nodes,

$$(1 - \varepsilon)d_{est}(t) < d_{act}(t) < (1 + \varepsilon)d_{est}(t). \tag{2.2}$$

We propose an algorithm, called *local change* and denoted by $\mathcal{A}_{lc}$. It relies on the same underlying principle as Dead-reckoning, where position estimations are deterministic and each node computes its own position as seen by other nodes, using the same deterministic algorithm. In $\mathcal{A}_{lc}$, a node Bob sends his actual position p_B to another node Alice as soon as the estimate $\widetilde{p_B}$ of the position of Bob as seen by Alice deviates too much from his actual position, more precisely *as soon as Equation 2.3 is violated*, where $d(p, q)$ denotes the distance between two nodes p and q. In addition, Alice will *immediately respond to Bob by also sending her actual position*.

$$d(p_B(t), \widetilde{p_B}(t)) < d_{est}(t) \times \frac{\varepsilon}{2}. \tag{2.3}$$

To quantify the performance of our algorithm, we compare the number of messages against an oracle with a full knowledge of the current state of the game, called *ideal algorithm* and denoted by $\mathcal{A}_{id}$. In $\mathcal{A}_{id}$, an exchange of messages happens only when, and as soon as Equation 2.2 is violated.

Our results are threefold. First, without any assumption on how nodes move, we prove that with $\mathcal{A}_{lc}$ the maximal error is never overcome: Equation 2.2 is always satisfied (Theorem 2.1, Section 2.2).

Secondly, in the case where movement is limited to the random part based on nodes' actions, which cannot be anticipated by the deterministic prediction algorithm, we prove that, given ε, $\mathcal{A}_{lc}$ is optimal in terms of number of message exchanges up to a constant factor. In sections 2.3 and 2.4, we use two different movement patterns, both of which consisting, at each time step $t \in \mathbb{N}$ (in the Black-Box model), to chose a new position at a distance at most 1 from the last position.

Finally, this theoretical analysis is complemented by experiments in Section 2.5. We first perform experiments on synthetic traces. Then, we use actual traces from Heroes of Newerth [65], to compare $\mathcal{A}_{lc}$ with a *fixed frequency algorithm*, denoted by $\mathcal{A}_{ff}$. $\mathcal{A}_{ff}$ is commonly used in practice in online games. Nodes send messages periodically: after sending a message containing its most recent position to another node, a node waits w time units before sending the next message to the same node. We show that overall, $\mathcal{A}_{lc}$ behaves better while never exceeding the maximal tolerated error.

In summary, the performance (without latency) of $\mathcal{A}_{id}$, $\mathcal{A}_{lc}$, $\mathcal{A}_{ff}$ and `timewarp` [102] in a setting with n interconnected nodes are shown in Table 2.1. Here, the connection graph is supposed to be complete (each node is connected to all $n - 1$ other nodes). T denotes the duration of the experiment. We consider as a reference m_{id}, the (perfect knowledge based) total number of messages sent by $\mathcal{A}_{id}$. In the worst case, $\mathcal{A}_{id}$ would make nodes send one message each time step (when movement is large compared to the distance), thus $m_{id} \leq Tn(n - 1)$.

Note that `timewarp` functions slightly differently than the others: it is intended to ensure strict consistency. The *number of violations* counts, over T time units, the number of distance pairs for which the error is above ε.

Table 2.1 – Performance of the algorithms

	number of messages	maximal error	number of violations
$\mathcal{A}_{id}$	$m_{id} \leq Tn(n-1)$	$\leq \varepsilon$	0
$\mathcal{A}_{lc}$	$O(m_{id})$	$\leq \varepsilon$	0
$\mathcal{A}_{ff}$	$\frac{T}{w}n(n-1)$	0 if $w = 1$ unbounded otherwise	$\Theta(Tn^2)$
`timewarp`	$O(Tn^2)$	0	0

2.2 Algorithm and Movement Models

2.2.1 Model

Let us first assume that $\varepsilon \in]0;1[$. Indeed, $\varepsilon = 0$ means that no error is tolerated, while $\varepsilon = 1$ would accept any estimate on the distance, provided it is larger than half the actual distance, which is not very informative.

We focus on two nodes Alice and Bob. As told in Section 2.1.2, we assume that the network is synchronous, and that local computations do not take time.

We use the Black-Box model, and at any time step $t \in \mathbb{N}$, we denote the positions of both nodes as $p_A(t)$ and $p_B(t)$. A position is a vector whose dimension depends on the virtual world (for example, for a 3D world, a position is described by a vector in $\mathbb{Z}^3$, or $\mathbb{R}^3$ in the case of continuous movement). Each node knows its own actual position, but may not know exactly where the other node is. These positions can change unpredictably, through the actions of users, but only at the time steps (not in-between).

In Section 2.3, we conduct analyses on Random Walks (see definition below), up to 3D. In Section 2.4, we use the Continuous Movement (the definition is also below). As these movements are random, the best possible estimation of the position of other nodes is to assume they remain still, so that a node will estimate that the other nodes are at their last known position.

Random Walk is a discrete movement taking place on a d-dimensional grid. Thus, positions can be represented as values from $\mathbb{Z}^d$. If at time step $t \in \mathbb{N}$, a node following such movement is at position $p = (p_1, p_2, \dots, p_d)$ it has $2d$ neighbors: $(p_1-1, p_2, \dots, p_d)$, $(p_1+1, p_2, \dots, p_d)$, $(p_1, p_2-1, \dots, p_d)$, etc. The movement consists, at each time step, to choose one of the neighbors, each one with probability $\frac{1}{2d}$.

Continuous Movement consists at each time step, to select a value smaller than one, and to add a vector of norm equal to this value, and with a direction randomly chosen. **In 1D**, a moving node adds at each time step, a random number following a uniform distribution on $[-1, 1]$ to their position. **In 2D**, at each time step t, a moving node X chooses ρ_t and θ_t following uniform distributions respectively on $[0, 1]$ and $[0, 2\pi]$, so that $p_X(t+1) = p_X(t) + (\rho_t, \theta_t)$, where (ρ_t, θ_t) is the vector with polar coordinates ρ_t and θ_t. **In 3D**, at each time step t, a moving node chooses ρ_t, θ_t, and φ_t following uniform distributions respectively on $[0, 1]$, $[0, 2\pi]$ and $[0, \pi]$, to add as spherical coordinates.

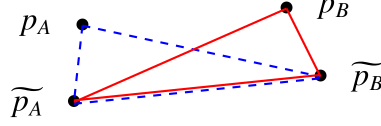


Figure 2.1 – Knowledge of Alice (dashed blue lines) and Bob (continuous red lines)

2.2.2 Algorithm

As explained in Section 2.1, nodes will estimate their distance to each other. To do this, each node will compute a deterministic estimation of the other node's position, in order to get $d_{est}(t)$, i.e. Bob computes $\widetilde{p_A}(t)$, the estimate of the position of Alice, and Alice computes $\widetilde{p_B}(t)$. As they use the same deterministic algorithm, these computations can be replicated, and $\widetilde{p_A}(t)$ and $\widetilde{p_B}(t)$ become a shared knowledge, as shown on Figure 2.1 (even without communication). Thus, we will use the distance between those two (estimated but shared) positions as distance estimate, $d_{est}(t)$. In practice, $\widetilde{p_A}(t)$ is generally based on an extrapolation of Alice's position, speed and acceleration, from the time of the last message exchanged between Alice and Bob.

As pointed out in Section 2.1.3, $\mathcal{A}_{lc}$ makes the nodes send each other updates of the actual position as soon as Equation 2.3 is not satisfied. For this, we use three communication rounds. First, a node detects a message exchange is needed, and sends an update to another node. In the second round, that other node receives the message and answers by sending its own update to the first node, and the third round is used for the first node to receive that update[1]. The procedure is described more precisely in Algorithm 1. The other algorithm, $\mathcal{A}_{id}$, used as a basis for comparison, sends updates as soon as the target inequality (Equation 2.2) becomes false, as depicted in Algorithm 2. As explained previously, $\mathcal{A}_{id}$ is not usable in practice, as it is based on a perfect knowledge of the state of both nodes, which none of them has.

In Theorem 2.1, we prove that $\mathcal{A}_{lc}$ satisfies Equation 2.2, thus its correctness is established. As mentioned in the theorem, this correctness is independent of the movement model, and thus in particular of speed limits.

Theorem 2.1. *Using $\mathcal{A}_{lc}$, Equation 2.2 holds true at any instant (regardless of movement).*

Proof. The following inequalities hold true:

$$\begin{cases} d_{act}(t) - d_{est}(t) \leq d(p_A(t), \widetilde{p_A}(t)) + d(p_B(t), \widetilde{p_B}(t)) & \text{(triangle inequality)} \\ d_{est}(t) - d_{act}(t) \leq d(p_A(t), \widetilde{p_A}(t)) + d(p_B(t), \widetilde{p_B}(t)) & \text{(triangle inequality)} \\ d(p_B(t), \widetilde{p_B}(t)) < \frac{\varepsilon}{2} d_{est}(t) & \text{(by construction)} \\ d(p_A(t), \widetilde{p_A}(t)) < \frac{\varepsilon}{2} d_{est}(t) & \text{(by construction)} \end{cases}$$

so that $|d_{act}(t) - d_{est}(t)| < \varepsilon d_{est}(t)$, which is equivalent to Equation 2.2. □

2.3 Random Walk Movement

In this section, we focus on the 1D case, where nodes move along the integer line. The performance of $\mathcal{A}_{lc}$ is measured by the random variable M, counting the number of message exchanges (a message and its response are counted as one) between two nodes using $\mathcal{A}_{lc}$, before

[1]For clarity, we use three rounds of communication, but two rounds could be enough, by merging the third round of a time step with the first round of the next time step.

Algorithm 1 Local change ($\mathcal{A}_{lc}$), from the point of view of Alice

```
1: Initialization:
2: p_A ← Alice's initial position    (*Actual position of Alice. This is a read-only input to the algorithm*)
3: p̃_A ← Alice's initial position    (*Position of Alice, as estimated by Bob, the other node*)
4: p̃_B ← Bob's initial position    (*Estimated position of Bob*)
5: d_est ← d(p̃_A, p̃_B)    (*Estimated distance. Will always be equal to d(p̃_A, p̃_B)*)
6: procedure TIME_STEP    (*To be executed at each time step*)
7:     update p_A
8:     ===== communication round 1 =====
9:     if d(p_A, p̃_A) ≥ (ε/2) d_est then
10:        p̃_A ← p_A
11:        d_est ← d(p̃_A, p̃_B)
12:        send message (p_A) to Bob
13:    ===== communication round 2 =====
14:    if a message (p_B) has been received from Bob then
15:        RECEIVE_MESSAGE(p_B)
16:        send message (p_A) to Bob
17:    ===== communication round 3 =====
18:    if a message (p_B) has been received from Bob then
19:        RECEIVE_MESSAGE(p_B)
20: procedure RECEIVE_MESSAGE(p_B)
21:    p̃_B ← p_B
22:    d_est ← d(p̃_A, p̃_B)
```

the first message sent with $\mathcal{A}_{id}$. In this setting, our result that $\mathcal{A}_{lc}$ is optimal is formally stated in Theorem 2.6 and Theorem 2.9, by an upper bound on the expectation of M. Note that this upper bound does not hold for a worst-case analysis: M can be infinitely large if nodes come and go, far enough for $\mathcal{A}_{lc}$ to send messages regularly, but not far enough for $\mathcal{A}_{id}$ to send messages.

Let us denote by d_{est} and $\widetilde{p}$ the estimates for $\mathcal{A}_{lc}$. We will consider instants t_i (with $i \geq 1$), defined as the instants at which the i-th round trip of the messages is sent with $\mathcal{A}_{lc}$. Both t_i and M are discrete random variables.

Let $d_0 = d_{act}(0)$. Algorithm $\mathcal{A}_{id}$ generates a message as soon as d_{act} leaves I_{id}, where I_{id} is defined by $I_{id} =]d_0\,(1-\varepsilon)\ ;\ d_0\,(1+\varepsilon)[$. Let $t_{opt} = \min\{t : d_{act}(t) \notin I_{id}\}$ denote the time of the first message sent by $\mathcal{A}_{id}$, then

$$M = \max\{i,\ t_i \leq t_{opt}\}.$$

Let us now define the auxiliary random variable M' : $\min\{i,\ d_{est}(t_i) \notin I_{id}\}$. M' represents the index of the first message of $\mathcal{A}_{lc}$ that is sent when d_{est} is outside of I_{id}.[2] At this instant, by construction, $\mathcal{A}_{id}$ already sent a message. The following proposition states that an upper bound for M' also holds for M.

Proposition 2.2. $M' \geq M$

Proof. By definition of $\mathcal{A}_{lc}$, for every i, $d_{est}(t_i) = d_{act}(t_i)$. Thus, $t_{M'} \in \{t,\ d_{act}(t) \notin I_{id}\}$, so that $t_{M'} \geq t_{opt}$. Since $t_{opt} \geq t_M$, $t_{M'} \geq t_M$ and $M' \geq M$. □

[2]We will first see a setting where only Bob moves, and Alice remains at position 0. In that case, M' represents the index of the first message of $\mathcal{A}_{lc}$ that is sent as Bob's position is outside of I_{id}.

Algorithm 2 The ideal algorithm, $\mathcal{A}_{id}$, from the point of view of Alice

1: Initialization:
2: $\boldsymbol{p_A} \leftarrow$ Alice's initial position (*Actual position of Alice. This is a read-only input to the algorithm*)
3: $\boldsymbol{p_B} \leftarrow$ Bob's initial position (*Actual position of Bob. This is a read-only input to the algorithm*)
4: $\widetilde{p_A} \leftarrow$ Alice's initial position (for both nodes) (*Estimated position of Alice*)
5: $\widetilde{p_B} \leftarrow$ Bob's initial position (for both nodes) (*Estimated position of Bob*)
6: $d_{est} \leftarrow d(\widetilde{p_A}, \widetilde{p_B})$ (for both nodes) (*Estimated distance*)
7: **procedure** TIME_STEP (*To be executed at each time step*)
8: update $\boldsymbol{p_A}$
9: ===== communication round 1 =====
10: **if** $|d_{act}, d_{est}| \geq \varepsilon d_{est}$ **then**
11: $\widetilde{p_A} \leftarrow \boldsymbol{p_A}$
12: send message ($\boldsymbol{p_A}$) to Bob
13: ===== communication round 2 =====
14: **if** a message (p_B) has been received from Bob **then** (*Note that if Alice sent a message during communication round 1, then Bob also sent a message.*)
15: $\widetilde{p_B} \leftarrow \boldsymbol{p_B}$
16: $d_{est} \leftarrow d(\widetilde{p_A}, \widetilde{p_B})$

In the following sections, we will look for upper bounds on the expected value of M', and thus get the same bounds for the expected value of M thanks to Proposition 2.2. For some executions of the Random Walk, M' may be arbitrarily larger than M, but this is mitigated by the fact that we analyze the expected values. However, this opens up the possibility to find better bounds in future works.

2.3.1 1D Case, Only One of the Nodes Moves

Let us start with the case when only one of the two nodes follows a 1D Random Walk[3], as described in Section 2.2. Then, $p_A(t) = 0$ at any instant t and Bob moves on $\mathbb{N}$, starting at distance $d_0 > 0$ from Alice so that $p_B(0) = d_0$ and

$$p_B(t+1) = \begin{cases} p_B(t) + 1 \text{ with probability } \frac{1}{2} \\ p_B(t) - 1 \text{ with probability } \frac{1}{2}. \end{cases}$$

In this section, we will present two bounds based on the same observation: at each message sent by $\mathcal{A}_{lc}$, the new position of Bob can take only two values: one closer, and one farther away from Alice than the previous position; we will call those changes *left* and *right jumps* (exact definitions will be given later). To get the first bound, we analyze the probability for having a high enough number of successive same-direction jumps to ensure that $\mathcal{A}_{id}$ also has to send a message, which will give us an upper bound for M. For the second bound, we use a finer

[3]Note that this movement model is not equivalent to the low mobility setting (see Section 1.6.1): here we impose that one of the nodes does never move, while in the low mobility setting both nodes may move, but not at the same time steps. In a setting where several nodes want to estimate their distance with each other, the movement model of this section imposes that no node moves except for one node (as for each couple of node, only one is allowed to move). This makes the general case, studied later, where both nodes may move, much more interesting in practice.

analysis: noting that, because of the multiplicative nature of the jumps, two opposite jumps end up reducing the distance between Alice and Bob, we will evaluate the probability of having an excess of left jumps high enough to ensure that $\mathcal{A}_{id}$ has to send a message, giving us another bound for M.

Let us first look at the update condition of $\mathcal{A}_{lc}$. Since the movements are 1D, it can be represented by intervals: $\mathcal{A}_{lc}$ generates a message exchange as soon as p_B leaves $\mathcal{I}_{lcB}$.

Definition 2.3. *With* $t \in [\![t_i; t_{i+1}[\![$, $\mathcal{I}_{lcB}(t) = \left]\widetilde{p_B}(t_i) - d_{est}(t_i)\frac{\varepsilon}{2} \; ; \; \widetilde{p_B}(t_i) + d_{est}(t_i)\frac{\varepsilon}{2}\right[$.

As explained in Section 2.2.1, we assume that d_{est} remains constant between two message exchanges in $\mathcal{A}_{lc}$, i.e. $\forall t \in [\![t_i, t_{i+1}[\![$, $d_{est}(t) = d_{est}(t_i)$. As a result, we have the following proposition.

Proposition 2.4. *With only Bob following 1D movements,* $\forall t \in [\![t_i, t_{i+1}[\![$, $\mathcal{A}_{lc}$ *triggers the* $i+1$*-th round trip of messages as soon as* $p_B(t)$ *gets out of* $\mathcal{I}_{lcB}(t) = \left]d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right) \; ; \; d_{est}(t_i)\left(1 + \frac{\varepsilon}{2}\right)\right[$.

Proof. Since Alice does not move and remains at the origin, all messages are generated by Bob and $\forall t \in [\![t_i; t_{i+1}[\![$, $\widetilde{p_B}(t) = d_{est}(t)$. Moreover, since $d_{est}(t) = d_{est}(t_i)$, then for Bob, Equation 2.3 is equivalent to $|p_B(t) - d_{est}(t_i)| < d_{est}(t_i) \times \frac{\varepsilon}{2}$, which in turn is equivalent to $p_B(t) \in \mathcal{I}_{lcB}(t)$. □

We have a similar result for the ideal algorithm:

Proposition 2.5. $\mathcal{A}_{id}$ *sends the first message as soon as* $p_B(t)$ *gets out of* I_{id}.

Proof. As $p_B(t) = d_{act}(t)$, this follows immediately from Equation 2.2. □

First upper bound on M

We provide a first upper bound on the expected value of M, that does not depend on the initial distance between the nodes.

Theorem 2.6. *Let* $\Delta_l = \left\lceil \frac{\log(1-\varepsilon)-\log(1+\varepsilon)}{\log(1-\frac{\varepsilon}{2})} \right\rceil$ *and* $\varepsilon \in \,]0;1[$ *. With two nodes, one of them following a Random Walk, on* $\mathbb{Z}$, $\mathbb{E}[M] \leq \Delta_l \times 2^{\Delta_l}$.

To prove Theorem 2.6, let us first look at the estimated distance. When a message is sent in $\mathcal{A}_{lc}$, $d_{est}(t_{i+1})$ can take only two values, as stated in Proposition 2.7.

Proposition 2.7. $d_{est}(t_{i+1}) = \begin{cases} d_{est}(t_i) - \left\lceil \frac{\varepsilon}{2} d_{est}(t_i) \right\rceil = \left\lfloor d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)\right\rfloor & \textit{(with probability } \frac{1}{2}\textit{)} \\ d_{est}(t_i) + \left\lceil \frac{\varepsilon}{2} d_{est}(t_i) \right\rceil = \left\lceil d_{est}(t_i)\left(1 + \frac{\varepsilon}{2}\right)\right\rceil & \textit{(with probability } \frac{1}{2}\textit{)} \end{cases}$

Proof. By definition of $\mathcal{A}_{lc}$, and since positions of Bob are integers, a message is sent when the position of Bob gets to the first integer position outside of $\mathcal{I}_{lcB}$. The rightmost equalities directly follow the properties of floor and ceiling function. Thus, the two possible positions at time t_{i+1} are at a same distance from $d_{est}(t_i)$ and have therefore the same probability. □

As a result, $d_{est}(t_{i+1})$ can only take two different values depending on $d_{est}(t_i)$, both having the same probability: either $d_{est}(t_{i+1}) = \left\lfloor d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)\right\rfloor$ or $\left\lceil d_{est}(t_i)\left(1 + \frac{\varepsilon}{2}\right)\right\rceil$ (see Figure 2.2). We will call *jump*, this transformation between $d_{est}(t_i)$ and $d_{est}(t_{i+1})$. We will notate this as two functions:

$$l : x \mapsto \left\lfloor x\left(1 - \frac{\varepsilon}{2}\right)\right\rfloor \tag{2.4}$$

and

$$r : x \mapsto \left\lceil x\left(1 + \frac{\varepsilon}{2}\right)\right\rceil. \tag{2.5}$$

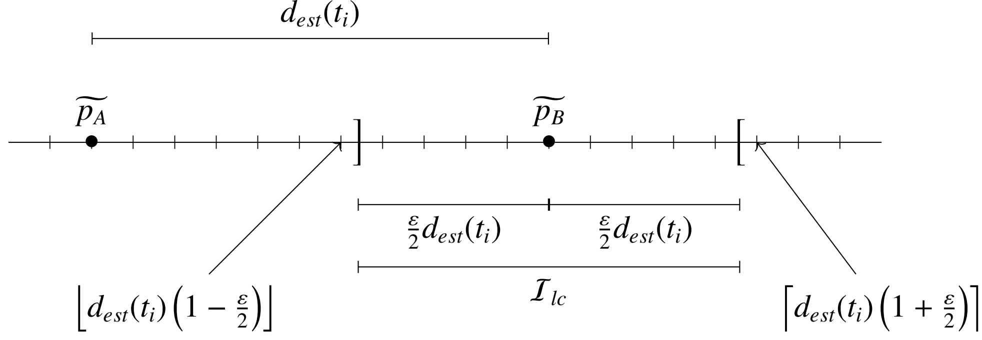


Figure 2.2 – Representation of l-jump and r-jump

To notate which type of jump is applied at instant t_i, we will use

$$m_i = \begin{cases} l \text{ if } d_{est}(t_{i+1}) = l(d_{est}(t_i)) \\ r \text{ if } d_{est}(t_{i+1}) = r(d_{est}(t_i)). \end{cases} \tag{2.6}$$

We may also define Δ_l[4]:

$$\Delta_l = \left\lceil \frac{\log(1-\varepsilon) - \log(1+\varepsilon)}{\log(1-\frac{\varepsilon}{2})} \right\rceil. \tag{2.7}$$

As stated in Lemma 2.8, Δ_l is a number such that Δ_l successive l-jumps[5] ensure Bob gets out of I_{id}, whatever his initial position in the interval I_{id}.

Lemma 2.8. *For all $x \in I_{id}$, $l^{\Delta_l}(x) \leq d_0(1-\varepsilon)$.*

Proof. $x \in I_{id} \Rightarrow x \leq d_0(1+\varepsilon) \Rightarrow l^{\Delta_l}(x) \leq l^{\Delta_l}(d_0(1+\varepsilon))$ since l is increasing, implying that $l^{\Delta_l}(x) \leq d_0(1+\varepsilon)\left(1-\frac{\varepsilon}{2}\right)^{\Delta_l}$ since $\forall x$, $l(x) \leq x\left(1-\frac{\varepsilon}{2}\right)$. Moreover, since, $\Delta_l \geq \frac{\log(1-\varepsilon)-\log(1+\varepsilon)}{\log\left(1-\frac{\varepsilon}{2}\right)}$ and $\log\left(1-\frac{\varepsilon}{2}\right) < 0$, then $(1+\varepsilon)\left(1-\frac{\varepsilon}{2}\right)^{\Delta_l} \leq (1-\varepsilon)$ and $x \in I_{id} \Rightarrow l^{\Delta_l}(x) \leq d_0(1-\varepsilon)$ □

Proof of Theorem 2.6. Let us split the sequence of movements of Bob into *phases* of length Δ_l and let us denote by j the index of the phase containing jumps from $m_{(j-1)\Delta_l}$ to $m_{(j\Delta_l)-1}$. Let us consider the following possible events (i) $\mathcal{S}_j$: there is at least one $i \in [\![(j-1)\Delta_l; j\Delta_l]\!]$ such that $d_{est}(t_i) \notin I_{id}$ and (ii) $\mathcal{S}'_j$: phase j is composed of l-jumps only. In turn, these events can be used to define useful random variables: (i) $X_j = 1$ if $\mathcal{S}_j$ is true, 0 otherwise (ii) $X'_j = 1$ if $\mathcal{S}'_j$ is true, 0 otherwise, (iii) $Y = j$ if $X_j = 1$ and $X_k = 0$ for every $k < j$ and (iv) $Y' = j$ if $X'_j = 1$ and $X'_k = 0$ for every $k < j$. Thus, Y denotes the index of the first phase during which Bob gets out of I_{id}, and Y' denotes the index of the first phase containing only of l-jumps.

If $\mathcal{S}'_j$ is true, then $d_{est}(t_{j\Delta_l}) = l^{\Delta_l}(d_{est}(t_{(j-1)\Delta_l}))$. Thus, by Lemma 2.8, $\mathcal{S}'_j \Rightarrow \mathcal{S}_j$, so that $X'_j = 1 \Rightarrow X_j = 1$.

$$\text{Therefore } Y' = j \Rightarrow X'_j = 1 \Rightarrow X_j = 1 \Rightarrow Y \leq j \text{ and finally } \mathbb{E}[Y] \leq \mathbb{E}[Y']. \tag{2.8}$$

Moreover, we know that Y' follows a geometric distribution with parameter $\mathbb{P}(X'_j = 1) = \frac{1}{2^{\Delta_l}}$ (because each jump has a $\frac{1}{2}$ probability of being l or r), so that $\mathbb{E}[Y'] \leq 2^{\Delta_l}$. Thus, by Equation 2.8, we have $\mathbb{E}[Y] \leq 2^{\Delta_l}$. Since Y denotes the index of the first phase during which Bob gets out of I_{id}, $M' \in [\![(Y-1)\Delta_l; Y\Delta_l]\!]$. In particular, $M' \leq Y\Delta_l$ and $\mathbb{E}[M'] \leq \Delta_l \times 2^{\Delta_l}$. Finally, Proposition 2.2 proves that $\mathbb{E}[M] \leq \Delta_l \times 2^{\Delta_l}$. □

[4] For example, for $\varepsilon = 0.1$, we have $\Delta_l = 4$.

[5] We could have used r-jumps, but values are better with l-jumps (the results are not symmetric : the error being relative, jumps get bigger the farther apart from each other the nodes are). We will later use Δ_r, the number of r-jumps needed to ensure Bob gets out of I_{id}.

Second upper bound on M

Theorem 2.9. *Let Δ_l be defined as previously. If $\varepsilon \in]0;1[$, with two nodes, one of them following a Random Walk, on $\mathbb{Z}$, then* $\mathbb{E}[M] \leq \left\lceil \frac{4}{\pi} \Delta_l^2 \right\rceil \times 8$

We provide a tighter analysis for M, which is formally stated in Theorem 2.9. To establish this result, we no longer consider phases consisting only of l-jumps, but also phases with a sufficient excess of l-jumps. This is because a sequence of an l-jump and a r-jump (in any order) tends to reduce the distance, as proved in Proposition 2.13.

To analyse successive jumps and the difference in numbers of l-jumps and r-jumps, let us define two additional notations[6].

Definition 2.10. *Let $m_{i,j} = m_{j-1} \circ m_{j-2} \circ \cdots \circ m_i$.*

This is a function. Given the estimated distance before the i-th jump, the output of $m_{i,j}$ is the estimated distance before the j-th jump, so that $d_{est}(t_j) = m_{i,j}(d_{est}(t_i))$

Definition 2.11. *Let $\sigma_{i,j} = card(\{k : m_k = l, k \in [\![i, j-1]\!]\}) - card(\{k : m_k = r, k \in [\![i, j-1]\!]\})$.*

This is a number : $\sigma_{i,j}$ denotes the excess in l from m_i to m_{j-1}.

We will need the following result in order to prove Theorem 2.9:

Theorem 2.12. *If $\sigma_{i,j} \geq \Delta_l$, and $x \in I_{id}$, then $m_{i,j}(x) \notin I_{id}$.*

Let us first prove the following properties:

Proposition 2.13. *$\forall p \in \mathbb{N}$, $l \circ r(p) \leq p$, and $r \circ l(p) \leq p$.*

Proof. $l(p) = p - \left\lceil \frac{p\varepsilon}{2} \right\rceil$, and $r(p) = p + \left\lceil \frac{p\varepsilon}{2} \right\rceil$, so that

$$l \circ r(p) = p + \left\lceil \frac{p\varepsilon}{2} \right\rceil - \left\lceil \frac{p\varepsilon}{2} + \left\lceil \frac{p\varepsilon}{2} \right\rceil \frac{\varepsilon}{2} \right\rceil \leq p \text{ since } \left\lceil \frac{p\varepsilon}{2} \right\rceil \leq \left\lceil \frac{p\varepsilon}{2} + \left\lceil \frac{p\varepsilon}{2} \right\rceil \frac{\varepsilon}{2} \right\rceil$$

$$\text{and } r \circ l(p) = p - \left\lceil \frac{p\varepsilon}{2} \right\rceil + \left\lceil \frac{p\varepsilon}{2} - \left\lceil \frac{p\varepsilon}{2} \right\rceil \frac{\varepsilon}{2} \right\rceil \leq p \text{ since } \left\lceil \frac{p\varepsilon}{2} - \left\lceil \frac{p\varepsilon}{2} \right\rceil \frac{\varepsilon}{2} \right\rceil \leq \left\lceil \frac{p\varepsilon}{2} \right\rceil.$$

□

Proposition 2.14. *$\forall (p, q) \in \mathbb{N}^2, \forall s \in \mathbb{N}$ and $\forall f = f_1 \circ f_2 \circ \cdots \circ f_s$, where $f_k = l$ or r for all $1 \leq k \leq s$, if $p \leq q$, then $f(p) \leq f(q)$.*

Proof. The proof is obtained by noting that the ceiling, floor, l and r functions and their compositions are increasing functions. □

Lemma 2.15. *Let $j > i$ and let us assume that $\sigma_{i,j} \geq 0$. $\forall p \in \mathbb{N}$, $m_{i,j}(p) \leq l^{\sigma_{i,j}}(p)$.*

Proof. Let $f = f_1 \circ f_2 \circ \cdots \circ f_s$ where $f_k = l$ or r for all $1 \leq k \leq s$. Let $T : f \mapsto f'$ with $f' = f_1 \circ \cdots \circ f_k \circ f_{k+3} \circ \cdots \circ f_s$ so that $f_{k+1} \circ f_{k+2} = r \circ l$ or $l \circ r$, i.e. T simply consists of removing the first occurrence of $r \circ l$ or $l \circ r$. Then, $f_{k+1} \circ f_{k+2} \circ f_{k+3} \circ \cdots \circ f_s(x) \leq f_{k+3} \circ \cdots \circ f_s(x)$ thanks to Proposition 2.13, and $f_1 \circ \cdots \circ f_s(x) \leq f_1 \circ \cdots \circ f_k \circ f_{k+3} \circ \cdots \circ f_s(x)$ thanks to Proposition 2.14, so that

$$f(x) \leq T(f)(x). \tag{2.9}$$

Let $T^* : f \mapsto f^*$ with f^* being the result of the recursive application of T on f until only ls remain (remember that $\sigma_{i,j} \geq 0$). By Equation 2.9, $f(p) \leq T^*(f)(p)$. As $T^*(m_{i,j}) = l^{\sigma_{i,j}}$, finally $m_{i,j}(p) \leq l^{\sigma_{i,j}}(p)$. □

[6]We use here the standard mathematical notation of $\circ$ for the function composition $g \circ f(x) = g(f(x))$.

Proof of Theorem 2.12. Let $\sigma_{i,j} \geq \Delta_l$ and $p \in I_{id}$. Since $\Delta_l > 0$, $\sigma_{i,j} > 0$. Thus, thanks to Lemma 2.15, $f_{i,j}(x) \leq l^{\sigma_{i,j}}(x) \leq l^{\Delta_l}(x)$ (because $l(x) \leq x$ and $\sigma_{i,j} \geq \Delta_l$). Thus, $f_{i,j}(x) \leq d_0(1-\varepsilon)$ thanks to Lemma 2.8, and by definition of I_{id}, we have $f_{i,j}(p) \notin I_{id}$ □

The following lemma provides a lower bound on the probability of the event $\sigma_{i,j} \geq \Delta_l$, that will be later used to get an upper bound on the expectation of M.

Lemma 2.16. *If $j - i = 2\left\lceil \frac{4}{\pi}\Delta_l^2 \right\rceil$, then $\mathbb{P}(\sigma_{i,j} \geq \Delta_l) \geq \frac{1}{4}$.*

Proof. Let $\Phi = j - i = 2\left\lceil \frac{4}{\pi}\Delta_l^2 \right\rceil$ be the number of jumps between m_i and m_{j-1}, and let $\Lambda = \text{card}(\{k : m_k = l, k \in [\![i, j-1]\!]\})$ be the number of l-jumps between m_i and m_{j-1}. Then, $\sigma_{i,j} \geq \Delta_l \Leftrightarrow 2\Lambda - \Phi \geq \Delta_l \Leftrightarrow \Lambda \geq \frac{\Delta_l + \Phi}{2}$ so that

$$
\begin{aligned}
\mathbb{P}(\sigma_{i,j} \geq \Delta_l) = \mathbb{P}\left(\Lambda \geq \frac{\Delta_l + \Phi}{2}\right) &= \sum_{k=\left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil}^{\Phi} \binom{\Phi}{k} \times \frac{1}{2^\Phi} \text{ because } \mathbb{P}(\Lambda = k) = \binom{\Phi}{k} \times \frac{1}{2^\Phi} \\
&= \frac{1}{2^\Phi}\left(\sum_{k=\frac{\Phi}{2}+1}^{\Phi} \binom{\Phi}{k} - \sum_{k=\frac{\Phi}{2}+1}^{\left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil - 1} \binom{\Phi}{k}\right) \text{ because } \left\lceil \frac{\Delta_l + \Phi}{2} \right\rceil > \frac{\Phi}{2} + 1.
\end{aligned}
$$

Moreover, as Φ is even, $\sum_{k=0}^{\Phi} \binom{\Phi}{k} = 2^\Phi = 2 \times \sum_{k=\frac{\Phi}{2}+1}^{\Phi} \binom{\Phi}{k} + \binom{\Phi}{\frac{\Phi}{2}}$ so that

$$
\begin{aligned}
\mathbb{P}(\sigma_{i,j} \geq \Delta_l) &= \frac{1}{2^\Phi}\left(\frac{2^\Phi - \binom{\Phi}{\frac{\Phi}{2}}}{2} - \sum_{k=\frac{\Phi}{2}+1}^{\left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil - 1} \binom{\Phi}{k}\right) \\
&= \frac{1}{2} - \frac{\binom{\Phi}{\frac{\Phi}{2}}}{2^{\Phi+1}} - \frac{1}{2^\Phi} \sum_{k=\frac{\Phi}{2}}^{\left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil - 1} \binom{\Phi}{k} + \frac{\binom{\Phi}{\frac{\Phi}{2}}}{2^\Phi} \geq \frac{1}{2} - \frac{1}{2^\Phi} \sum_{k=\frac{\Phi}{2}}^{\left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil - 1} \binom{\Phi}{k}.
\end{aligned}
$$

There are $\left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil - \frac{\Phi}{2}$ elements in the remaining sum. Note that $\left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil \leq \frac{\Delta_l+\Phi}{2} + 1 \Rightarrow \left\lceil \frac{\Delta_l+\Phi}{2} \right\rceil - \frac{\Phi}{2} \leq \frac{\Delta_l}{2} + 1$ and that each element of the sum is smaller than the first one since $\binom{\Phi}{\frac{\Phi}{2}} \geq \binom{\Phi}{\frac{\Phi}{2}+n}$. Therefore,

$$
\begin{aligned}
\mathbb{P}(\sigma_{i,j} \geq \Delta_l) &\geq \frac{1}{2} - \frac{1}{2^\Phi} \times \left(\frac{\Delta_l}{2} + 1\right) \times \binom{\Phi}{\frac{\Phi}{2}} \\
&\geq \frac{1}{2} - \frac{1}{2^\Phi} \times \left(\frac{\Delta_l}{2} + 1\right)\left(\frac{2^\Phi}{\sqrt{\frac{\Phi}{2} \times \pi}}\right) \\
&\geq \frac{1}{2}\left(1 - \frac{\Delta_l \sqrt{2}}{\sqrt{\Phi\pi}}\right) + \left(\frac{\sqrt{2}}{\sqrt{\Phi\pi}}\right) \\
&\geq \frac{1}{2}\left(1 - \sqrt{\frac{\pi}{8}} \times \sqrt{\frac{2}{\pi}}\right) + \left(\frac{\sqrt{2}}{\sqrt{\Phi\pi}}\right) \text{ because } \Phi \geq \frac{8}{\pi}\Delta_l^2 \Rightarrow \sqrt{\frac{\pi}{8}} \geq \frac{\Delta_l}{\sqrt{\Phi}} \\
&\geq \frac{1}{4} + \left(\frac{\sqrt{2}}{\sqrt{\Phi\pi}}\right) \geq \frac{1}{4} \text{ because } \Phi > 0.
\end{aligned}
$$

□

Proof of Theorem 2.9. As for the proof of Theorem 2.6, let us split the sequence of Bob movements in *phases* of length Φ and let us denote by j the index of the phase containing jumps $m_{(j-1)\Phi}$ through $m_{(j\Phi)-1}$. Let us consider the following events (i) $\mathcal{S}_j$: there is at least one $i \in [\![(j-1)\Phi; j\Phi]\!]$ such that $d_{est}(t_i) \notin I_{id}$ and (ii) $\mathcal{S}''_j$: either $d_{est}(t_{(j-1)\Phi}) \notin I_{id}$, or $d_{est}(t_{j\Phi}) \notin I_{id}$. These events can in turn be used to define the following random variables (i) $X_j = 1$ if $\mathcal{S}_j$ is true, 0 otherwise (ii) $X''_j = 1$ if $\mathcal{S}''_j$ is true, 0 otherwise (iii) $Y = j$ if $X_j = 1$ and $X_k = 0$ for every $k < j$ and (iv) $Y'' = j$ if $X''_j = 1$ and $X''_k = 0$ for every $k < j$. Thus Y denotes the index of the first phase during which Bob gets out of I_{id}.

If $\mathcal{S}''_j$ is true, then $\mathcal{S}_j$ also holds true. Thus, in a similar way as for Theorem 2.6, $Y'' = j \Rightarrow X''_j = 1 \Rightarrow X_j = 1 \Rightarrow Y \leq j$ and thus $\mathbb{E}[Y] \leq \mathbb{E}[Y'']$. Moreover, by Theorem 2.12 and Lemma 2.16, if $\Phi = \left\lceil \frac{4}{\pi}\Delta_l{}^2 \right\rceil \times 2$, then $\mathbb{P}(\mathcal{S}''_j) \geq \frac{1}{4}$. Note that Y'' follows a geometric distribution with parameter $\mathbb{P}(X''_j = 1)$, so that $\mathbb{E}[Y] \leq \mathbb{E}[Y''] \leq 4$. Since Y denotes the index of the first phase during which Bob gets out of I_{id}, then $M' \in [\![(Y-1)\Phi; Y\Phi]\!]$. In particular, since $M' \leq Y\Phi$, $\mathbb{E}[M'] \leq \Phi \times 4$. By Proposition 2.2, we get $\mathbb{E}[M] \leq \left\lceil \frac{4}{\pi}\Delta_l{}^2 \right\rceil \times 8$. □

Conclusion

In the 1D case, we prove that $\mathbb{E}[M]$ is smaller than both $\Delta_l \times 2^{\Delta_l}$ and $\left\lceil \frac{4}{\pi}\Delta_l{}^2 \right\rceil \times 8$, where $\Delta_l = \left\lceil \frac{\log(1-\varepsilon)-\log(1+\varepsilon)}{\log(1-\frac{\varepsilon}{2})} \right\rceil$. Actually, the choice of the best upper bound depends on values of ε. We can also observe that $\lim_{\varepsilon \to 1} \Delta_l = \infty$, meaning that there is no upper bound on M when ε is close to 1. This is not surprising, since a value of 1 for ε would make the left bound of I_{id} become 0 and $\mathcal{A}_{lc}$ could perform an infinite number of l-jumps before the first message of $\mathcal{A}_{id}$ if d_0 is large enough. Experiments depicted in Section 2.5 indeed show that M can become large when ε gets close to one.

2.3.2 1D Case, Both Nodes Move

In this section, we consider that both nodes move (under the same stochastic movement model) on the integer line $\mathbb{Z}$. Again, we concentrate on a single pair of nodes Alice and Bob but the results apply to any pair of nodes and therefore can be extended to any number of nodes.

At each time step, both Alice and Bob move. Thus, we have, for any $t \geq 1$, for Alice:

$$p_A(t) = \begin{cases} p_A(t-1) + 1 \text{ with probability } \frac{1}{2} \\ p_A(t-1) - 1 \text{ with probability } \frac{1}{2} \end{cases} \tag{2.10}$$

and for Bob:

$$p_B(t) = \begin{cases} p_B(t-1) + 1 \text{ with probability } \frac{1}{2} \\ p_B(t-1) - 1 \text{ with probability } \frac{1}{2}. \end{cases} \tag{2.11}$$

The equality between position and distance ($\widetilde{p_B}(t) = d_{est}(t)$) is no longer valid so that Proposition 2.4 does not hold. However, we will keep $\mathcal{I}_{lcB}(t) = \left]\widetilde{p_B}(t_i) - d_{est}(t_i)\frac{\varepsilon}{2} \,;\, \widetilde{p_B}(t_i) + d_{est}(t_i)\frac{\varepsilon}{2}\right[$, and we will add $\mathcal{I}_{lcA}(t) = \left]\widetilde{p_A}(t_i) - d_{est}(t_i)\frac{\varepsilon}{2} \,;\, \widetilde{p_A}(t_i) + d_{est}(t_i)\frac{\varepsilon}{2}\right[$. The definition of interval I_{id} remains unchanged, and messages are exchanged at t such that $d_{act}(t) \notin I_{id}$ (and not p_B). Theorem 2.17 is an extension of Theorem 2.6 in the case where both nodes move.

Theorem 2.17. *Let Δ_l be defined as previously. If $\varepsilon \in]0; 1[$, then with two nodes following a Random Walk on $\mathbb{Z}$, $\mathbb{E}[M] \leq \Delta_l \times 4^{\Delta_l}$*

Proof. Assume, without loss of generality that Bob remains to the right of Alice, that is, $p_B > p_A$. After the $(i+1)$-th round trip of messages in $\mathcal{A}_{lc}$, i.e. at instant t_{i+1}, one of the four following

events takes place (i) $\mathcal{B}_l$: at instant t_{i+1}, node Bob gets out of $\mathcal{I}_{lcB}$ by getting closer to Alice; (ii) $\mathcal{B}_r$: at instant t_{i+1}, node Bob gets out of $\mathcal{I}_{lcB}$ by getting farther away from Alice; (iii) $\mathcal{A}_l$: at instant t_{i+1}, node Alice gets out of $\mathcal{I}_{lcA}$ by getting farther away from Bob ; (iv) $\mathcal{A}_r$: at instant t_{i+1}, node Alice gets out of $\mathcal{I}_{lcA}$ by getting closer to Bob.

At least one of these events has to hold true: $\mathbb{P}(\mathcal{B}_l \cup \mathcal{B}_r \cup \mathcal{A}_l \cup \mathcal{A}_r) = 1$. Additionally, all four events have the same probability, as both nodes start at the center of their interval at instant t_i. Thus, $\mathbb{P}(\mathcal{B}_l) = \mathbb{P}(\mathcal{B}_r) = \mathbb{P}(\mathcal{A}_l) = \mathbb{P}(\mathcal{A}_r) \geq \frac{1}{4}$.

Let us consider for instance the situation where $\mathcal{B}_l$ is true, i.e. $\widetilde{p_B}(t_{i+1}) = \widetilde{p_B}(t_i) - \left\lceil d_{est}(t_i) \times \frac{\varepsilon}{2} \right\rceil$. When Bob gets out of $\mathcal{I}_{lcB}(t_i)$, as movement is symmetric, Alice has one half probability to be on one side of $\widetilde{p_A}(t_i)$, thus $\mathbb{P}(\widetilde{p_A}(t_{i+1}) \geq \widetilde{p_A}(t_i) | \mathcal{B}_l) \geq \frac{1}{2}$.

Moreover, $\widetilde{p_A}(t_{i+1}) \geq \widetilde{p_A}(t_i) \Rightarrow \widetilde{p_B}(t_{i+1}) - \widetilde{p_A}(t_{i+1}) \leq \widetilde{p_B}(t_i) - \widetilde{p_A}(t_i) - \left\lceil d_{est}(t_i)\frac{\varepsilon}{2} \right\rceil$ by definition of $\mathcal{B}_l$. Therefore, we have that $d_{est}(t_{i+1}) \leq d_{est}(t_i) - \left\lceil d_{est}(t_i)\frac{\varepsilon}{2} \right\rceil$ because Bob is on the right side of Alice. Finally, we get $d_{est}(t_{i+1}) \leq d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)$. Thus, $\mathbb{P}\left(d_{est}(t_{i+1}) \leq d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right) | \mathcal{B}_l\right) \geq \frac{1}{2}$ and by a comparable reasoning on node Alice, we get $\mathbb{P}\left(d_{est}(t_{i+1}) \leq d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right) | \mathcal{A}_r\right) \geq \frac{1}{2}$. Thus, using the law of total probability, we get $\mathbb{P}\left(d_{est}(t_{i+1}) \leq d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)\right) \geq \frac{1}{2} \times \mathbb{P}(\mathcal{B}_l) + \frac{1}{2} \times \mathbb{P}(\mathcal{A}_r) + 0 \times \mathbb{P}(\mathcal{B}_r) + 0 \times \mathbb{P}(\mathcal{A}_l) \geq \frac{1}{4}$ Repeating this operation Δ_l times, we get $\mathbb{P}\left(d_{est}(t_{i+\Delta_l}) \leq d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)^{\Delta_l}\right) \geq \frac{1}{4^{\Delta_l}}$. and similarly to Lemma 2.8, we get $d_{est}(t_i) \in I_{id} \Rightarrow d_{est}(t_i) \leq d_0(1+\varepsilon) \Rightarrow d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)^{\Delta_l} \leq d_0(1+\varepsilon)\left(1 - \frac{\varepsilon}{2}\right)^{\Delta_l}$ and since $\Delta_l \geq \frac{\log(1-\varepsilon) - \log(1+\varepsilon)}{\log\left(1 - \frac{\varepsilon}{2}\right)}$ then $d_{est}(t_i) \in I_{id} \Rightarrow d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)^{\Delta_l} \leq d_0(1 - \varepsilon) \Rightarrow d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right)^{\Delta_l} \notin I_{id}$.

$$\text{Hence, } \forall i, \;\; d_{est}(t_i) \in I_{id} \Rightarrow \mathbb{P}\left(d_{est}(t_{i+\Delta_l}) \notin I_{id}\right) \geq \frac{1}{4^{\Delta_l}}. \tag{2.12}$$

To prove Theorem 2.17, we rely on the same techniques as for Theorem 2.6 and Theorem 2.9, by splitting the sequence of jumps into phases of length Δ_l, and by denoting as j the index of the phase containing jumps $m_{(j-1)\Phi}$ through $m_{j\Phi-1}$. Let us consider the event $\mathcal{S}_j$: there is at least one $i \in [\![(j-1)\Delta_l; j\Delta_l]\!]$ such that $d_{est}(t_i) \notin I_{id}$ and the random variables (i) $X_j = 1$ if $\mathcal{S}_j$ is true, 0 otherwise, (ii) $Y = j$ if $X_j = 1$ and $X_k = 0$ for all $k < j$. By Equation 2.12, if $d_{est}(t_{(j-1)\Delta_l}) \in I_{id}$, then $\mathbb{P}\left(d_{est}(t_{j\Delta_l}) \notin I_{id}\right) \geq \frac{1}{4^{\Delta_l}}$ so that $\mathbb{P}(\mathcal{S}_j) \geq \frac{1}{4^{\Delta_l}}$ and $\mathbb{E}[Y] \leq 4^{\Delta_l}$. Since Y denotes the index of the first phase during which Bob gets out of I_{id}, then $M' \in [\![(Y-1)\Delta_l; Y\Delta_l]\!]$ and in particular, $M' \leq Y\Delta_l$, so $\mathbb{E}[M'] \leq \Delta_l \times 4^{\Delta_l}$. By Proposition 2.2, we finally obtain $\mathbb{E}[M] \leq \Delta_l \times 4^{\Delta_l}$, what achieves the proof of Theorem 2.17. □

2.3.3 2D and 3D Case

As seen in Section 2.2, in a d-D space space, the movement of a node consists in following a Random Walk on a d-D grid. If at an instant t, a node is at position $p = (p_1, p_2, \ldots, p_d)$, then there are $2d$ adjacent positions: $(p_1 - 1, p_2, \ldots, p_d)$, $(p_1 + 1, p_2, \ldots, p_d)$, $(p_1, p_2 - 1, \ldots, p_d)$, $\ldots$, $(p_1, p_2, \ldots, p_d - 1)$, $(p_1, p_2, \ldots, p_d + 1)$. The movement consists, at each time step $t \in \mathbb{N}$, to chose one of those adjacent positions, each with probability $\frac{1}{2d}$.

In 2D, for example, this means that, at each time step, a moving node adds one of the following to his/her position: $(-1, 0)$, $(1, 0)$, $(0, -1)$, or $(0, 1)$.

For our analysis, we will use the L^1 distance (Manhattan distance), that is, for two positions $p = (p_1, p_2)$ and $p' = (p'_1, p'_2)$, the distance is $d(p, p') = |p_1 - p'_1| + |p_2 - p'_2|$. We consider here the general case where both nodes may move.

The bound we find in this section is a generalization of the bound found in 1D, as it relies on the same principle. We will first evaluate the probability that the node that triggers a message exchanges does it by getting farther away from the other node. For this, we observe that a message exchange happens when one of the nodes lands on the borders of an L^1-Ball, and that one of its faces has all its points far away from the other node.

We then evaluate the probability that the two nodes are significantly farther away when they communicate, and use a similar method as previously to get an upper bound for M.

Let us call d the number of dimensions, supposed less than or equal to three. Let us prove that in a d-D space, we have a similar bound than in the 1D case.

Theorem 2.18. *In a d-D Euclidian space, with* $d \leq 3$, *and* $\Delta_r = \left\lceil \frac{\log(1+\varepsilon)-\log(1-\varepsilon)}{\log(1+\frac{\varepsilon}{2})} \right\rceil$, *and with two nodes moving, we have* $\mathbb{E}[M] \leq \Delta_r \times \left(2^{d+1}\right)^{\Delta_r}$

As can be seen in Theorem 2.18, this time, we use Δ_r, the number of r-jumps needed to ensure a message with $\mathcal{A}_{id}$, instead of Δ_l as in the previous section. This is for simplicity of the proofs: we will later use $\mathcal{R}$, the set of positions that lead to a r-jump; the equivalent for l-jumps would be more complicated to define and use.

Let us assume, without loss of generality, that Bob is the node that triggers the $(i+1)$-th message, at instant t_{i+1}.

Let us call $\mathcal{B}_A(t)$ (resp. $\mathcal{B}_B(t)$) the L^1-ball of radius $\left\lceil d_{est}(t)\frac{\varepsilon}{2}\right\rceil$, and of center $\widetilde{p_A}(t)$ (resp. $\widetilde{p_B}(t)$). Thus, $\mathcal{B}_A(t)$ is the set of positions that are at a distance from $\widetilde{p_A}(t)$ less than or equal to $\left\lceil d_{est}(t)\frac{\varepsilon}{2}\right\rceil$ (this is the lower square on Figure 2.3a).

Remark 2.19. *With* $\mathcal{A}_{lc}$, *the* $(i+1)$*-th message is sent when Bob is on the border of* $\mathcal{B}_B(t_i)$.

Proof. With $\mathcal{A}_{lc}$, the $(i+1)$-th message is sent when Bob gets at a position that is at a distance at least $\frac{\varepsilon}{2}d_{est}(t_i)$ from $\widetilde{p_B}(t_i)$. As movement is on integer positions, the first positions satisfying this are all on the border of $\mathcal{B}_B(t_i)$. □

This ball $\mathcal{B}_B$ has 2^d faces of dimension $(d-1)$. We may draw cones, each taking one of these faces as base, and with $\widetilde{p_B}(t_i)$ as the apex: all points of the space will be in only one of the cones, except for points on the borders (see Figure 2.3b for a two-dimensional example, where the borders of the cones are the dashed lines).

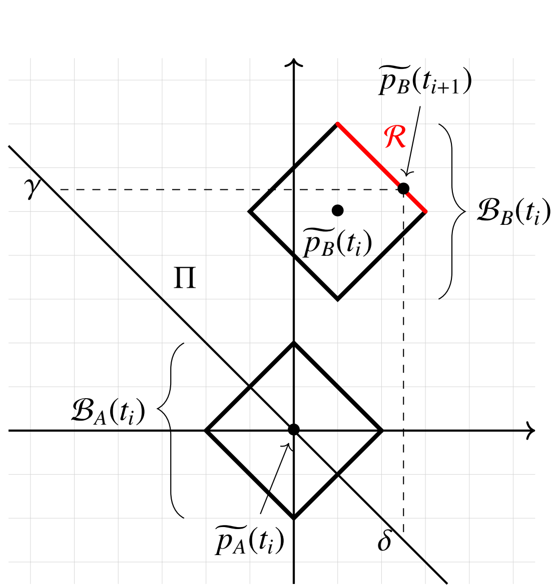
Let us call $\mathcal{R}$ the face that is included in the cone opposing the one containing $\widetilde{p_A}(t_i)$. In the case where $\widetilde{p_A}(t_i)$ is contained in two cones (that is, if $\widetilde{p_A}(t_i)$ and $\widetilde{p_B}(t_i)$ have one coordinate in common), than any of the two opposing faces can be taken for $\mathcal{R}$. $\mathcal{R}$ is represented in red on Figure 2.3b, and as we will see in Lemma 2.20, it is a set of positions p such that $d(p, \widetilde{p_A}(t_i)) = \left\lceil d_{est}(t_i)\left(1+\frac{\varepsilon}{2}\right)\right\rceil$.

Before being able to identify the effect a message has on the estimated distance (which we will do in Lemma 2.23), we analyse how far Bob's estimated position can get from Alice (below in Lemma 2.20).

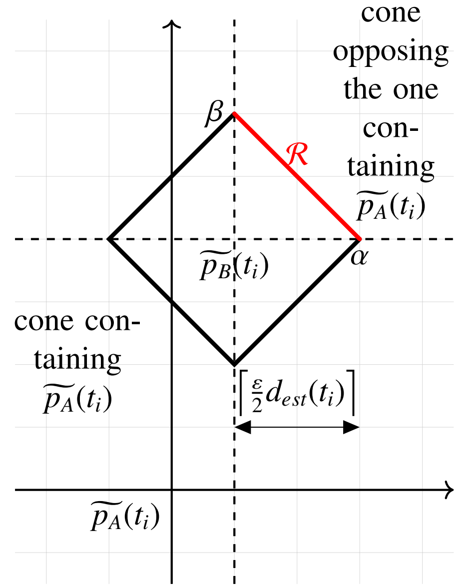
Lemma 2.20. *If* $\widetilde{p_B}(t_i) \neq \widetilde{p_A}(t_i)$, $\mathbb{P}\left(d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1})) \geq \left\lceil d_{est}(t_i)\left(1+\frac{\varepsilon}{2}\right)\right\rceil\right) \geq \frac{1}{2^d}$

Proof. All points of $\mathcal{R}$ are at distance $\left\lceil d_{est}(t_i)\left(1+\frac{\varepsilon}{2}\right)\right\rceil$ of $\widetilde{p_A}(t_i)$ (for this, consider one of the endpoints of the face, like α on Figure 2.3b, for which all coordinates are the same as for $\widetilde{p_B}(t_i)$, except one, where the absolute value is larger by $\left\lceil \frac{\varepsilon}{2}d_{est}(t_i)\right\rceil$). It thus remains to be proven that $\mathbb{P}\left(\widetilde{p_B}(t_{i+1}) \in \mathcal{R}\right) \geq \frac{1}{2^d}$.

By Remark 2.19, $\widetilde{p_B}(t_{i+1})$ is on the border of $\mathcal{B}_B(t_i)$, which is, by definition of the L^1 distance, a hypercube with 2^d faces. As the Random Walk is symmetric, we have a probability of at least $\frac{1}{2^d}$ that Bob sends the $(i+1)$-th message by going on face $\mathcal{R}$. □



(a) When Bob gets on $\mathcal{R}$, half of the possible positions of Alice are farther away.



(b) One of the face of $\mathcal{B}_B$ is always sufficiently far away from $\widetilde{p_A}(t_i)$.

Figure 2.3 – Random walk, two-dimensional case

In Lemma 2.20, the movement of Alice is not taken into account. Let us call Π the hyperplane parallel to $\mathcal{R}$ and containing $\widetilde{p_A}(t_i)$ (see Figure 2.3a for a two-dimensional example).

Remark 2.21. *As Π contains $\widetilde{p_A}(t_i)$, the center of $\mathcal{B}_A$, Π divides $\mathcal{B}_A$ into two halves of the same size.*

Lemma 2.22. *At least half of the points p of $\mathcal{B}_A$ satisfy :*

$$d(p, \widetilde{p_B}(t_{i+1})) \geq d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1})).$$

Proof. By definition of the L^1 distance, and because Π is parallel to $\mathcal{R}$, if we draw, on Π, a polygon connecting d points that are the projections of $\widetilde{p_B}(t_{i+1})$ parallel to the d axes (γ and δ on Figure 2.3a), then all points of Π inside this polygon (including the borders) are all at the same distance to $\widetilde{p_B}(t_{i+1})$.

Also, by definition of $\mathcal{R}$, $\widetilde{p_A}(t_i)$ is inside the polygon. Thus, all points of the polygon are at a distance to $\widetilde{p_B}(t_{i+1})$ equal to $d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1}))$.

If we draw the L^1-ball of center $\widetilde{p_B}(t_{i+1})$ and of radius $d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1}))$, then the polygon is one of the faces of the ball. By Remark 2.21, we have that at least half of the points from $\mathcal{B}_A$ are outside this ball, with a distance to $\widetilde{p_B}(t_{i+1})$ higher than the radius of the ball. □

We can now look at the estimated distance.

Lemma 2.23. *As long as $\widetilde{p_B}(t_i) \neq \widetilde{p_A}(t_i)$, $\mathbb{P}\left(d_{est}(t_{i+1}) \geq \left\lceil d_{est}(t_i)\left(1+\frac{\varepsilon}{2}\right)\right\rceil\right) \geq \frac{1}{2^{d+1}}$*

Proof. As Alice does not get out of $\mathcal{B}_A$, we know that $\widetilde{p_A}(t_{i+1}) \in \mathcal{B}_A$. By Lemma 2.22, and by symmetry of the random movement, $d(\widetilde{p_A}(t_{i+1}), \widetilde{p_B}(t_{i+1})) \geq d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1}))$ with probability $\frac{1}{2}$. Thus, the result is the same as for Lemma 2.20, but with half as much probability. □

As we consider r-jumps, we have to adapt Lemma 2.8 as follows.

Lemma 2.24. *For all $x \in I_{id}$, $r^{\Delta_r}(x) \geq d_0(1+\varepsilon)$.*

Proof. $x \in I_{id} \Rightarrow x \geq d_0(1-\varepsilon) \Rightarrow r^{\Delta_r}(x) \geq r^{\Delta_r}(d_0(1-\varepsilon))$ since r is increasing, implying that $r^{\Delta_r}(x) \geq d_0(1-\varepsilon)\left(1+\frac{\varepsilon}{2}\right)^{\Delta_r}$ since $\forall x, r(x) \geq x\left(1+\frac{\varepsilon}{2}\right)$. Moreover, since, $\Delta_r \geq \frac{\log(1+\varepsilon)-\log(1-\varepsilon)}{\log\left(1+\frac{\varepsilon}{2}\right)}$, then $(1-\varepsilon)\left(1+\frac{\varepsilon}{2}\right)^{\Delta_l} \geq (1+\varepsilon)$ and $x \in I_{id} \Rightarrow r^{\Delta_r}(x) \geq d_0(1+\varepsilon)$ □

We are now ready to prove the main result of this section:

Proof of Theorem 2.18. This proof is very similar to Theorem 2.6. By Lemma 2.23, we know that the probability of having a r-jump (as defined in Equation 2.6) at an instant t_i, is at least $\frac{1}{2^{d+1}}$.

With *phases* of length Δ_r and j the index of the phase containing jumps from $m_{(j-1)\Delta_r}$ to $m_{j\Delta_r-1}$, we have (i) $\mathcal{S}_j$: there is at least one $i \in [\![(j-1)\Delta_r; j\Delta_r]\!]$ such that $d_{est}(t_i) \notin I_{id}$ (ii) $\mathcal{S}'_j$: the phase j is composed only of r-jumps. (iii) $X_j = 1$ if $\mathcal{S}_j$ is true, 0 otherwise (iv) $X'_j = 1$ if $\mathcal{S}'_j$ is true, 0 otherwise (v) $Y = j$ if $X_j = 1$ and $X_k = 0$ for every $k < j$ and (vi) $Y' = j$ if $X'_j = 1$ and $X'_k = 0$ for every $k < j$. Thus, Y denotes the index of the first phase during which Bob gets out of I_{id}.

If $\mathcal{S}'_j$ is true, then $d_{est}(t_{j\Delta_r}) = r^{\Delta_r}(d_{est}(t_{(j-1)\Delta_r}))$. Thus, by Lemma 2.24, $\mathcal{S}'_j \Rightarrow \mathcal{S}_j$, so that $X'_j = 1 \Rightarrow X_j = 1$.

$$\text{Therefore } Y' = j \Rightarrow X'_j = 1 \Rightarrow X_j = 1 \Rightarrow Y \leq j \text{ and finally } \mathbb{E}[Y] \leq \mathbb{E}[Y']. \tag{2.13}$$

Moreover, we know that Y' follows a geometric distribution with parameter $\mathbb{P}(X'_j = 1) \geq \frac{1}{(2^{d+1})^{\Delta_r}}$ (because each jump has at least probability $\frac{1}{2^{d+1}}$ of being r), and $\mathbb{E}[Y'] \leq \left(2^{d+1}\right)^{\Delta_r}$. Thus, by Equation 2.13, we have $\mathbb{E}[Y] \leq \left(2^{d+1}\right)^{\Delta_r}$. Since Y denotes the index of the first phase during which Bob gets out of I_{id}, $M' \in [\![(Y-1)\Delta_r; Y\Delta_r]\!]$. In particular, $M' \leq Y\Delta_r$ and $\mathbb{E}[M'] \leq \Delta_r \times \left(2^{d+1}\right)^{\Delta_r}$. Finally, Proposition 2.2 proves that $\mathbb{E}[M] \leq \Delta_r \times \left(2^{d+1}\right)^{\Delta_r}$. □

Remark 2.25. *If only one node moves, then* $\mathbb{E}[M] \leq \Delta_r \times \left(2^{d}\right)^{\Delta_r}$

Proof. The proof is the same as for Theorem 2.18, noticing that $d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1})) = d_{est}(t_{i+1})$. □

2.4 Continuous Movement, Discrete Time

In this section, we present bounds on M for the Continuous Movement. We consider here only the general case where both nodes may move.

2.4.1 1D Case

As we have seen in Section 2.2, in one dimension, the movement simply consists in adding to the position a random number following a uniform distribution on $[-1, 1]$.

The problem is that when node X gets out of $\mathcal{B}_X(t_i)$, then the next position may take several values: for example, if X got out by the left, then $\widetilde{p_X}(t_i)$ may take any value smaller than the left bound of $\mathcal{B}_X(t_i)$ and higher to this bound minus one (the biggest movement he may have done at the last time step before getting out).

Nevertheless, the equivalent of Theorem 2.17 still holds true in this setting:

Theorem 2.26. *If* $\varepsilon \in]0;1[$ *, then with two nodes following a continuous random movement on* $\mathbb{R}$, $\mathbb{E}[M] \leq \Delta_l \times 4^{\Delta_l}$.

To see this, let us again call m_i the transformation between $d_{est}(t_i)$ and $d_{est}(t_{i+1})$. This time we will get:

$$m_i \in \begin{cases} \mathscr{L} \text{ if } d_{est}(t_{i+1}) \leq d_{est}(t_i)\left(1 - \frac{\varepsilon}{2}\right) \\ \mathscr{R} \text{ if } d_{est}(t_{i+1}) \geq d_{est}(t_i)\left(1 + \frac{\varepsilon}{2}\right) \end{cases} \tag{2.14}$$

By definition of $\mathcal{A}_{lc}$, m_i has to be in either $\mathscr{L}$ or $\mathscr{R}$, and the probability is actually $\frac{1}{2}$ for both cases.

Using this, we have a result comparable to Lemma 2.8:

Lemma 2.27. *For all $x \in I_{id}$, if $j - i \geq \Delta_l$, and all $m_k \in \mathscr{L}$ for $k \in [\![i, j-1]\!]$ then $m_{j-1} \circ m_{j-2} \circ \cdots \circ m_i(x) \leq d_0(1 - \varepsilon)$.*

Proof. Let us call $m_i^{\Delta_l} = m_{j-1} \circ m_{j-2} \circ \cdots \circ m_i$.

Let us assume that $m_k \in \mathscr{L}$. All the m_k are increasing, as m_k is necessary of the form $x \mapsto x\left(1 - \frac{\varepsilon}{2}\right) - a_k$, with $a_k \in [0, 1]$. Thus, we have $x \in I_{id} \Rightarrow x \leq d_0(1+\varepsilon) \Rightarrow m_i^{\Delta_l}(x) \leq m_i^{\Delta_l}(d_0(1+\varepsilon))$, what implies that $m_i^{\Delta_l}(x) \leq d_0(1+\varepsilon)\left(1 - \frac{\varepsilon}{2}\right)^{\Delta_l}$, since $\forall k \in [\![i, j-1]\!]$ and $\forall x$, $m_k(x) \leq x\left(1 - \frac{\varepsilon}{2}\right)$. Moreover, since, $\Delta_l \geq \frac{\log(1-\varepsilon)-\log(1+\varepsilon)}{\log\left(1-\frac{\varepsilon}{2}\right)}$ and $\log\left(1 - \frac{\varepsilon}{2}\right) < 0$, then $(1+\varepsilon)\left(1 - \frac{\varepsilon}{2}\right)^{\Delta_l} \leq (1 - \varepsilon)$ so that $x \in I_{id} \Rightarrow m_i^{\Delta_l}(x) \leq d_0(1 - \varepsilon)$ □

The proof of Theorem 2.26 is then a direct translation of the proof of Theorem 2.17.

2.4.2 2D Case

As we have seen in Section 2.2, in two dimensions, the movement consists in choosing an angle θ between 0 and 2π, and moving a distance ρ between 0 and 1 in that direction. Thus, at each time step t, a moving node X chooses θ_t and ρ_t following continuous distributions respectively on $[0, 2\pi]$ and $[0, 1]$, so that $p_X(t+1) = p_X(t) + (\rho_t, \theta_t)$, where (ρ_t, θ_t) is the vector with polar coordinates ρ_t and θ_t.

Our result is as follows:

Theorem 2.28. *With $\Gamma = 2\,\frac{\log(1+\varepsilon)-\log(1-\varepsilon)}{\log\left(1+\frac{\varepsilon}{\sqrt{2}}+\frac{\varepsilon^2}{4}\right)}$, with two nodes following a random Continuous Movement in two dimensions as defined previously, and implementing $\mathcal{A}_{lc}$ we have: $\mathbb{E}[M] \leq \Gamma \times 8^{\Gamma}$.*

This time again, we will call Bob the node who gets out the first of his set of authorized positions with $\mathcal{A}_{lc}$, meaning that Bob is the node to initiate communication at instant t_{i+1}.

In this setting, we will use the euclidian distance: $\mathcal{B}_B(t_i)$ takes the form of a disk of center $\widetilde{p_B}(t_i)$ and of radius $\frac{\varepsilon}{2}d_{est}$, as represented on Figure 2.4. We will use the same general principle as before, considering only jumps of a single type. Let us call $r_{cm} : x \mapsto x\sqrt{\left(1 + \frac{\varepsilon^2}{4} + \frac{\varepsilon}{\sqrt{2}}\right)}$.

In order to identify r_{cm}-jumps, let us consider the annulus of inner circle $\mathcal{B}_B(t_i)$, and with an outer circle of radius $\frac{\varepsilon}{2}d_{est} + 1$ (see Figure 2.4). As no message is sent by $\mathcal{A}_{lc}$ as long as Bob remains in $\mathcal{B}_B(t_i)$, and as he doesn't move more than one distance unit each moment, this annulus represents the set of positions Bob can take if he triggers a message exchange.

We will call $\mathcal{R}$ the portion of this annulus on the opposite side of $\widetilde{p_A}(t_i)$, (represented as a red hatched zone on Figure 2.4), that deviates not more than $\frac{\pi}{4}$ from the straight line between $\widetilde{p_A}(t_i)$ and $\widetilde{p_B}(t_i)$. More formally, with t the intersection between $\mathcal{B}_B$ and the line $(\widetilde{p_A}(t_i)\widetilde{p_B}(t_i))$, on the opposite side of $\widetilde{p_A}(t_i)$, then $\mathcal{R} = \left\{s, \angle s\widetilde{p_B}(t_i)t \in \left[-\frac{\pi}{4}, \frac{\pi}{4}\right] \text{ and } d(s, \widetilde{p_B}(t_i)) \in \left[\frac{\varepsilon}{2}d_{est}(t_i), \frac{\varepsilon}{2}d_{est}(t_i) + 1\right]\right\}$, where $\angle s\widetilde{p_B}(t_i)t$ denotes the measure of the angle formed by the three points s, $\widetilde{p_B}(t_i)$, and t.

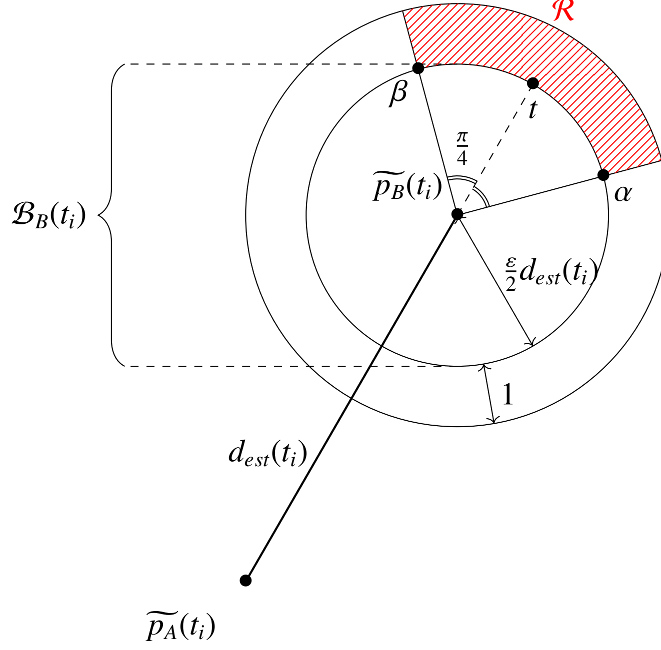


Figure 2.4 – Representation of $\mathcal{B}_B(t_i)$, of $\mathcal{R}$ (the set of points corresponding to an r_{cm}-jump), and other associated values.

We will see in Lemma 2.30 that $\mathcal{R}$ is a set of points corresponding to an r_{cm}-jump, that is, if $\widetilde{p_B}(t_{i+1}) \in \mathcal{R}$, than $d_{est}(t_{i+1}) \geq r_{cm}(d_{est}(t_i))$.

We first identify the probability for Bob to send his position when getting in $\mathcal{R}$ in Lemma 2.29.

Lemma 2.29. *In two dimensions,* $\mathbb{P}(\widetilde{p_B}(t_{i+1}) \in \mathcal{R}) = \frac{1}{4}$.

Proof. As a node does not move more than one distance unit per time unit, the first time step when Bob gets outside of $\mathcal{B}_B(t_i)$, he will be in the annulus. Thus $\widetilde{p_B}(t_{i+1})$ is inside the annulus.

Without loss of generality, let us consider only the movement between Bob's initial position ($p_B(0)$, actually equal to $\widetilde{p_B}(0)$) and the position at time of the first message ($p_B(t_1)$, actually equal to $\widetilde{p_B}(t_1)$). Let us call $T = (p_0, p_1, \ldots, p_{t_1})$ the trajectory taken by Bob to get on $\widetilde{p_B}(t_1)$, with p_t the position Bob had at time step t, where $t \in [\![0, t_1]\!]$. We have $p_0 = p_B(0)$, $p_1 = p_B(1)$, etc., and $p_{t_1} = \widetilde{p_B}(t_1)$.

See Figure 2.5 for a representation of the values.

Let us consider following random variables:

- R, taking the value of $d(p_0, p_{t_i})$.

- Θ, taking the value of the angle between the dashed lines of Figure 2.5, that is, the angle formed by t, p_0, and p_{t_i}, with t the intersection between $\mathcal{B}_B$ and the line $(\widetilde{p_A}(0)p_0)$, on the opposite side of $\widetilde{p_A}(0)$ (similarly to t on Figure 2.4).

As the Random Walk consists in randomly picking an angle θ_t and a distance ρ_t at every time step t, we have that $p_{t_1} = p_0 + (\rho_0, \theta_0) + (\rho_1, \theta_1) + \cdots + (\rho_{t_1-1}, \theta_{t_1-1})$, where (ρ_t, θ_t) is the vector of radius ρ_t and angle θ_t in polar coordinates.

Let us suppose that $t_1 = k$ for some $k > 0$, or in other words, that the trajectory T consists of k hops. Because the θ_t all follow a uniform distribution, for any angle γ, the probability that $p_{t_1} = p_0 + (\rho_0, \theta_0) + (\rho_1, \theta_1) + \cdots + (\rho_{t_1-1}, \theta_{t_1-1})$ is equal to the probability that $p_{t_1} = p_0 + (\rho_0, \theta_0 +$

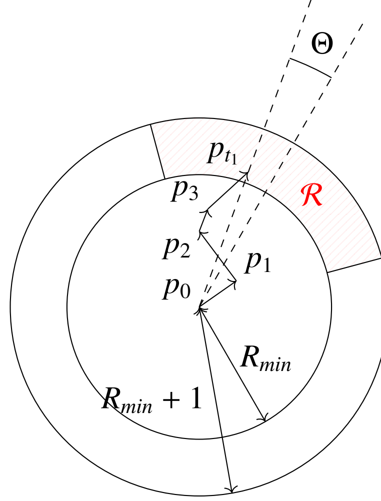


Figure 2.5 – Representation of the values used to compute the probability of a r_{cm}-jump (Lemma 2.29)

$\gamma)+(\rho_1,\theta_1+\gamma)+\cdots+(\rho_{t_1-1},\theta_{t_1-1}+\gamma)$. More exactly, $\mathbb{P}(a \le \Theta \le b) = \mathbb{P}(a+\gamma \le \Theta \le b+\gamma)$ for any γ, regardless of the value of R. As additionally[7], $\int_0^{2\pi} f_\Theta(x)dx = 1$, we get that $\int_{-\frac{\pi}{4}}^{\frac{\pi}{4}} f_\Theta(x)dx = \frac{1}{4}$.

Let us note R_{min} the radius of $\mathcal{B}_B(t_i)$ (that is, $R_{min} = \frac{\varepsilon}{2}d_{est}$). As Bob cannot move of more than one distance unit per time step, we have $\int_{R_{min}}^{R_{min}+1} f_R(x)dx = 1$. Thus, $\int_{R_{min}}^{R_{min}+1}\int_{-\frac{\pi}{4}}^{\frac{\pi}{4}} f_{\Theta,R}(x,y)dxdy = \frac{1}{4}$.

Moreover, we have $\int_{R_{min}}^{R_{min}+1}\int_{-\frac{\pi}{4}}^{\frac{\pi}{4}} f_{\Theta,R}(x,y)dxdy = \mathbb{P}(\widetilde{p_B}(t_1) \in \mathcal{R})$. As we supposed that $t_1 = k$ is true, we have that $\mathbb{P}(\widetilde{p_B}(t_1) \in \mathcal{R} \mid t_1 = k) = \frac{1}{4}$

By the law of total probability, we have that $\sum_0^{+\infty}\left(\mathbb{P}(\widetilde{p_B}(t_1) \in \mathcal{R} \mid t_1 = k) \times \mathbb{P}(t_i = k)\right) = \mathbb{P}(\widetilde{p_B}(t_1)) \in \mathcal{R}) = \frac{1}{4}$.

This remains true if we replace instants $0, 1, 2, \ldots, t_1$ by $t_i, t_i + 1, t_i + 2, \ldots, t_{i+1}$, proving this lemma. □

We may then identify, in Lemma 2.30 and Lemma 2.31, situations where r_{cm} appears.

Lemma 2.30. *With two nodes moving in two dimensions,*

$$\mathbb{P}\left(d_{est}(t_{i+1}) \ge r_{cm}(d_{est}(t_i)) \mid \widetilde{p_B}(t_{i+1}) \in \mathcal{R}\right) \ge \frac{1}{2}.$$

Proof. Let us assume $\widetilde{p_B}(t_{i+1}) \in \mathcal{R}$.

The two points of $\mathcal{R}$ that are closest to $\widetilde{p_A}(t_i)$ are α and β, the two points of the border of $\mathcal{B}_B(t_i)$ so that $\angle\alpha\widetilde{p_B}(t_i)t = \angle t\widetilde{p_B}(t_i)\beta = \frac{\pi}{4}$ (with again t the intersection between $\mathcal{B}_B$ and the line $(\widetilde{p_A}(t_i)\widetilde{p_B}(t_i))$, on the opposite side of $\widetilde{p_A}(t_i)$), see Figure 2.4. Thus, if we call d' the distance between $\widetilde{p_A}(t_i)$ and α (which is also the distance between $\widetilde{p_A}(t_i)$ and β), we have $d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1})) \ge d'$.

As can be seen on Figure 2.6, the value of d' can be given by the law of cosines, relatively to the value of $d_{est}(t_i)$:

$$d' = \sqrt{d_{est}(t_i)^2 + \frac{\varepsilon^2}{4}d_{est}(t_i)^2 - d_{est}(t_i)^2\varepsilon\cos\left(\frac{3\pi}{4}\right)} = d_{est}(t_i)\sqrt{\left(1 + \frac{\varepsilon^2}{4} + \frac{\varepsilon}{\sqrt{2}}\right)}$$

[7] We use here the classical notation of probability density function : for any continuous random variable X, we denote by f_X its density.

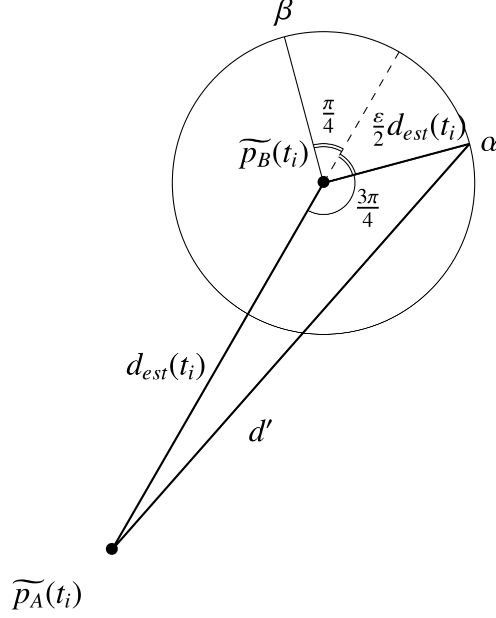


Figure 2.6 – Representation of the different values used to measure an r_{cm}-jump

This corresponds to r_{cm}.

Thus, $\mathbb{P}(d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1})) \geq r_{cm}(d_{est}(t_i)) | \widetilde{p_B}(t_{i+1}) \in \mathcal{R}) = 1$.

We may then notice that, as Alice remains inside $\mathcal{B}_A(t_i)$, the probability that $\widetilde{p_A}(t_{i+1})$ is farther away from $\widetilde{p_B}(t_{i+1})$ than $\widetilde{p_A}(t_i)$ is at least one half. This gives us the final result. □

Lemma 2.31. *With two nodes moving,* $\mathbb{P}\Big(d_{est}(t_{i+1}) \geq r_{cm}(d_{est}(t_i))\Big) \geq \frac{1}{8}$

Proof. The proof of Lemma 2.31 is now immediate with Lemma 2.29, Lemma 2.30, and the law of total probability. □

The last needed property is that successions of r_{cm} will make $\mathcal{A}_{id}$ send a message (recall that $\mathcal{A}_{id}$ sends a message as soon as d_{act} leaves $I_{id} =]d_0\,(1-\varepsilon)\ ;\ d_0\,(1+\varepsilon)[$):

Lemma 2.32. *With* $\Gamma = \frac{\log(1+\varepsilon)-\log(1-\varepsilon)}{\log\left(\sqrt{1+\frac{\varepsilon^2}{4}+\frac{\varepsilon}{\sqrt{2}}}\right)}$, *for all* $x \in I_{id}$, ${r_{cm}}^{\Gamma}(x) \geq d_0(1+\varepsilon)$.

Proof. $x \in I_{id} \Rightarrow x \geq d_0(1-\varepsilon) \Rightarrow {r_{cm}}^{\Gamma}(x) \geq {r_{cm}}^{\Gamma}(d_0(1-\varepsilon))$ since r_{cm} is increasing, so that ${r_{cm}}^{\Gamma}(x) \geq d_0(1-\varepsilon)\left(\sqrt{\left(1+\frac{\varepsilon^2}{4}+\frac{\varepsilon}{\sqrt{2}}\right)}\right)^{\Gamma}$. Moreover, by definition of Γ, $(1-\varepsilon)\left(\sqrt{\left(1+\frac{\varepsilon^2}{4}+\frac{\varepsilon}{\sqrt{2}}\right)}\right)^{\Gamma} \geq (1+\varepsilon)$, so that finally $x \in I_{id} \Rightarrow {r_{cm}}^{\Gamma}(x) \geq d_0(1+\varepsilon)$. □

Proof of Theorem 2.28. We may now prove the theorem with the same reasoning as Theorems 2.6, 2.9 and 2.18.

By Lemma 2.31, we know that the probability of having a jump that increases distance more than r_{cm} at an instant t_i, is at least $\frac{1}{8}$.

With *phases* of length Γ, and

1. $\mathcal{S}_j$: there is at least one $i \in [\![(j-1)\Gamma; j\Gamma]\!]$ such that $d_{est}(t_i) \notin I_{id}$;

2. $\mathcal{S}'_j$: the phase j is composed only of jumps so that the distance increases more than with r_{cm}. That is, for all $i \in [\![(j-1)\Gamma; j\Gamma-1]\!]$, $d_{est}(t_{i+1}) \geq r_{cm}(d_{est}(t_i))$;

3. $X_j = 1$ if $\mathcal{S}_j$ is true, 0 otherwise ;

4. $X'_j = 1$ if $\mathcal{S}'_j$ is true, 0 otherwise ;

5. $Y = j$ if $X_j = 1$ and $X_k = 0$ for every $k < j$;

6. $Y' = j$ if $X'_j = 1$ and $X'_k = 0$ for every $k < j$. Thus, Y denotes the index of the first phase during which Bob gets out of I_{id}.

We know that Y' follows a geometric distribution with parameter $\mathbb{P}(X'_j = 1) \geq \frac{1}{8^\Gamma}$ (because each jump has at least probability $\frac{1}{8}$ of complying to $\mathcal{S}'_j$), and $\mathbb{E}[Y'] \leq 8^\Gamma$. Thus, by Lemma 2.32, we have $\mathbb{E}[Y] \leq 8^\Gamma$. Since Y denotes the index of the first phase during which Bob gets out of I_{id}, $M' \in [\![(Y-1)\Gamma; Y\Gamma]\!]$. In particular, $M' \leq Y\Gamma$ and $\mathbb{E}[M'] \leq \Gamma \times 8^\Gamma$. Finally, Proposition 2.2 proves that $\mathbb{E}[M] \leq \Gamma \times 2^8\Gamma$. □

2.4.3 3D Case

Theorem 2.33. *With* $\Gamma = 2\frac{\log(1+\varepsilon)-\log(1-\varepsilon)}{\log\left(1+\frac{\varepsilon}{\sqrt{2}}+\frac{\varepsilon^2}{4}\right)}$*, with two nodes following a random Continuous Movement in 3D and implementing* $\mathcal{A}_{lc}$*, we have* $\mathbb{E}[M] \leq \Gamma \times 14^\Gamma$.

The reasoning is very similar to the two dimensional case, the main difference being that $\mathcal{B}_B$ is now a sphere. Thus, $\mathcal{R}$ is now a portion of a spherical shell (instead of an annulus). The same definition of $\mathcal{R}$ as in the 2D case is still valid: with t the intersection between $\mathcal{B}_B$ and the line $(\widetilde{p_A}(t_i)\widetilde{p_B}(t_i))$, on the opposite side of $\widetilde{p_A}(t_i)$, then we have the definition $\mathcal{R} = \left\{s, \angle s\widetilde{p_B}(t_i)t \in \left[-\frac{\pi}{4}, \frac{\pi}{4}\right] \text{ and } d(s, \widetilde{p_B}(t_i)) \in \left[\frac{\varepsilon}{2}d_{est}(t_i), \frac{\varepsilon}{2}d_{est}(t_i) + 1\right]\right\}$. It can also be seen as the previous $\mathcal{R}$ from Figure 2.4, but rotated with respect to the line $(\widetilde{p_A}(t_i)\widetilde{p_B}(t_i))$.

Lemma 2.34. *In the 3D case,* $\mathbb{P}(\widetilde{p_B}(t_{i+1}) \in \mathcal{R}) = \frac{2-\sqrt{2}}{4}$.

Proof. The first time step when Bob gets outside of $\mathcal{B}_B(t_i)$, he will be in the spherical shell of inner sphere $\mathcal{B}_B(t_i)$, and with the outer sphere of same center, but with a radius longer of one distance unit ($\frac{\varepsilon}{2}d_{est}(t_i) + 1$).

Let us use the spherical coordinates, centered on $\widetilde{p_B}(t_i)$, and with the z axis pointing towards t. The solid angle covered by $\mathcal{R}$ is the surface, on the unit sphere, of the zone where the colatitude is smaller than $\frac{\pi}{4}$: $\int_0^{\frac{\pi}{4}}\int_0^{2\pi} \sin(\phi)d\theta d\phi = 2\pi\left(1 - \frac{\sqrt{2}}{2}\right)$.

As the whole space is represented by 4π, this means that $\mathcal{R}$ takes $\left(\frac{1}{2} - \frac{\sqrt{2}}{4}\right) \approx 15\%$ of the spherical shell.

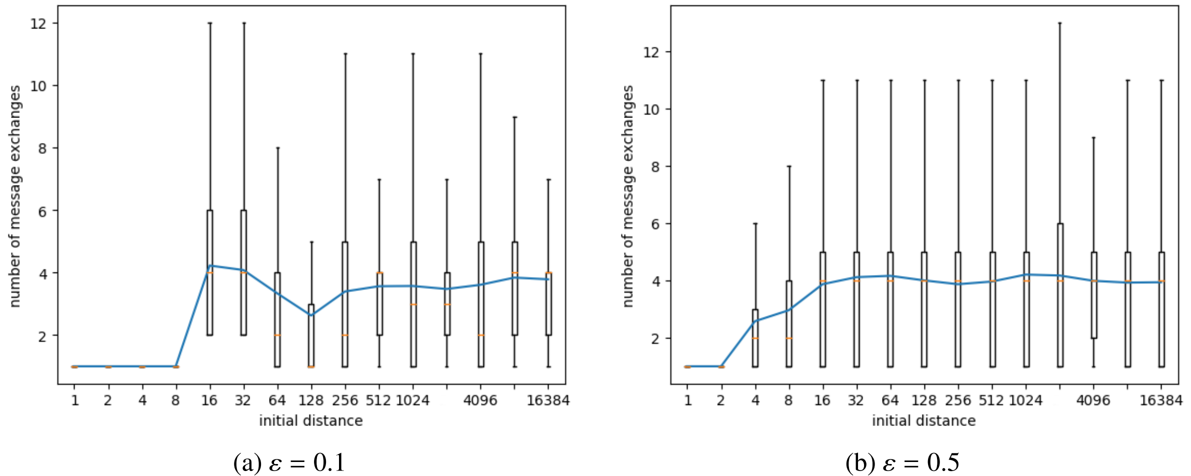
As movement is symmetric with respect to the center of the spherical shell, we can apply the same reasoning as in Lemma 2.29, so that we have probability $\left(\frac{1}{2} - \frac{\sqrt{2}}{4}\right)$ that $\widetilde{p_B}(t_{i+1}) \in \mathcal{R}$. □

Lemma 2.35. *With two nodes moving in 3D,* $\mathbb{P}\left(d_{est}(t_{i+1}) \geq r_{cm}(d_{est}(t_i)) \mid \widetilde{p_B}(t_{i+1}) \in \mathcal{R}\right) \geq \frac{1}{2}$.

Proof. On all planes containing the line $(\widetilde{p_A}(t_i)\widetilde{p_B}(t_i))$, the points of $\mathcal{R}$ closest to $\widetilde{p_A}(t_i)$ follow the construction of α and β on Figure 2.6. Thus, as in the proof of Lemma 2.30, $\mathbb{P}\left(d(\widetilde{p_A}(t_i), \widetilde{p_B}(t_{i+1})) \geq r_{cm}(d_{est}(t_i)) \mid \widetilde{p_B}(t_{i+1}) \in \mathcal{R}\right) = 1$.

For any point x outside $\mathcal{B}_A(t_i)$, more than half of the points y inside $\mathcal{B}_A(t_i)$ satisfy $d(x, y) \geq d(x, \widetilde{p_A}(t_i))$. Thus, as $\widetilde{p_A}(t_{i+1})$ remains inside $\mathcal{B}_A(t_i)$, and $\widetilde{p_B}(t_{i+1}) \in \mathcal{R} \Rightarrow \widetilde{p_B}(t_{i+1}) \notin \mathcal{B}_A(t_i)$, we get our result. □

Proof of Theorem 2.33. To prove this theorem, we first use Lemma 2.34 and Lemma 2.35: $\mathbb{P}\left(d_{est}(t_{i+1}) \geq r_{cm}(d_{est}(t_i))\right) \geq \frac{2-\sqrt{2}}{8}$. Thus, with phases of length Γ, the probability for a phase to contain only jumps that make distance higher than with r_{cm}-jumps is $\left(\frac{8}{2-\sqrt{2}}\right)^\Gamma \approx 14^\Gamma$. Finally, $\mathbb{E}[M] \leq \Gamma \times 14^\Gamma$. □



(a) $\varepsilon = 0.1$

(b) $\varepsilon = 0.5$

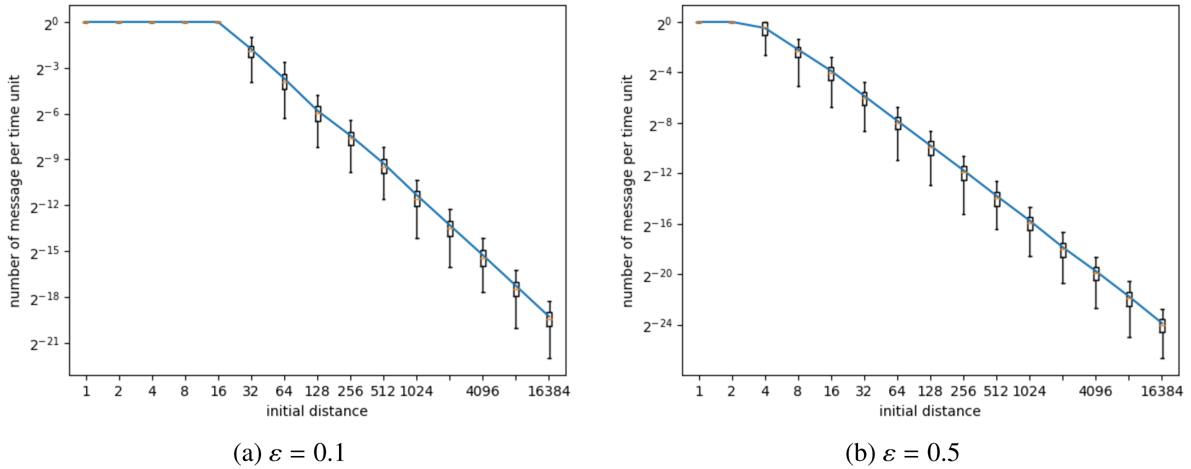
Figure 2.7 – One node moving, Random Walk, 1D: M depending on initial distance

2.5 Experiments

In order to analyze in practice the performance of $\mathcal{A}_{lc}$, we propose simulation results. More precisely, we execute both $\mathcal{A}_{lc}$ and $\mathcal{A}_{id}$ with the same set of random movements (of one or two nodes) and we display M, the number of message exchanges induced by $\mathcal{A}_{lc}$ at the time the first message is induced by $\mathcal{A}_{id}$. We perform simulations for different values of the initial distance (d_0) and maximum error (ε) and for each set of parameters. Everywhere, we repeat the experiments 500 times to account for the stochastic nature of the movements. In all the plots, the blue lines indicate the average value and the boxes indicate $Q1$, the first quartile and $Q3$, the third quartile. The lower whisker takes the values of the lowest reference point that is in the range $[Q1 - 1.5 \times IQR; Q1]$, where $IQR = Q3 - Q1$. Similarly, the upper whisker shows the highest reference point in the $[Q3; Q3 + 1.5 \times IQR]$ range. The results corresponding to the theoretical framework of Random Walk movement considered in Section 2.3 are presented in Section 2.5.1, while we present in Section 2.5.2 simulation results based on actual traces of games of Heroes of Newerth [65]. In particular, we use these traces to compare the behavior of $\mathcal{A}_{lc}$ with the behavior of solutions that are currently implemented in online games and that are based on fixed frequency messages.

2.5.1 Synthetic Traces

The first set of simulations correspond to the setting of Section 2.3.1. In the 1D case, when only one node moves, the evolution of M with the initial distance is depicted in Figure 2.7a ($\varepsilon = 0.1$) and Figure 2.7b ($\varepsilon = 0.5$). As expected, we can observe that M remains bounded and does not depend much on the initial distance (except when the distance is very small with respect to movement amplitudes). Even though constants are smaller than those proved in Theorem 2.6 and Theorem 2.9, the results are in line with the theoretical analysis. Figure 2.8a and Figure 2.8b depict the actual number of messages sent per time step when using $\mathcal{A}_{lc}$, as a function of the initial distance for $\varepsilon = 0.1$ and $\varepsilon = 0.5$. We can observe that the number of messages generated by $\mathcal{A}_{lc}$ quadratically decreases with the distance between the nodes (slope -2 in log-log scale), which is a desirable property, since maintaining an approximate distance should be less expensive when node characters are distant. We also plot the evolution of M with the given maximal tolerated error, ε in Figure 2.9. We can observe that M increases when ε gets close to 1, what suggests that the dependance on ε in our theoretical bounds is unavoidable.



(a) $\varepsilon = 0.1$

(b) $\varepsilon = 0.5$

Figure 2.8 – One node moving, Random Walk, 1D: messages per time unit

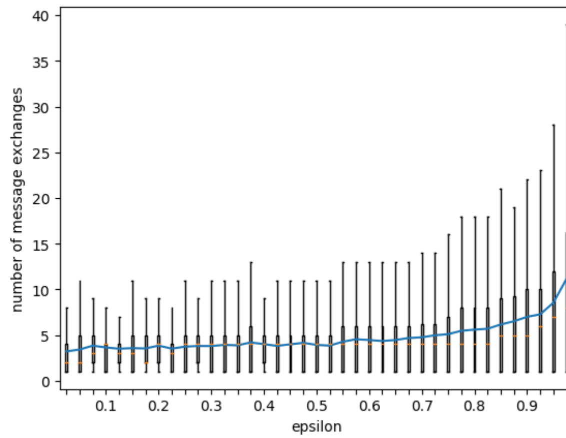


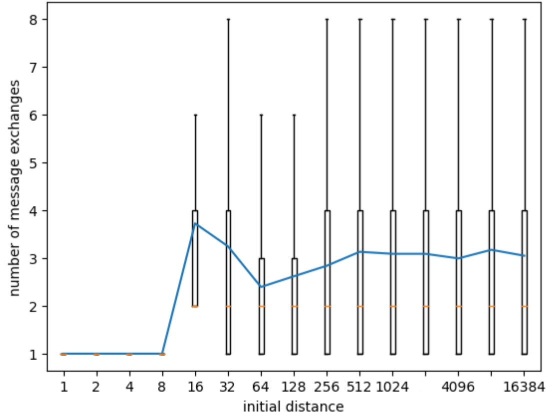
Figure 2.9 – One node moving, Random Walk, 1D: value of M depending on ε, for $d_0 = 400$

When both nodes move, as can be seen on figures 2.10, 2.11, and 2.12, plots show similar behavior as when only one node moves.

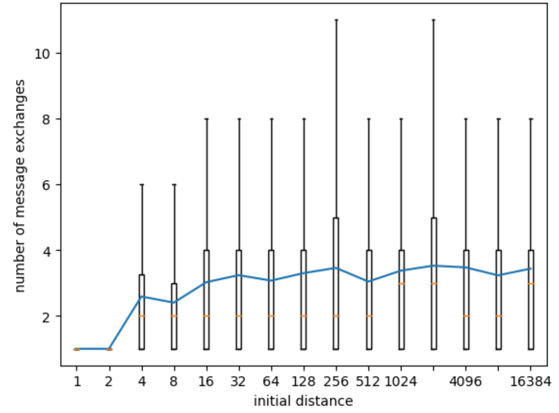
Figures 2.13, 2.14, and 2.15 show results for two nodes moving in a 2D-space. In this case too, M remains bounded when the initial distance changes (from approximately 4 times more messages in 1D to approximately 6 times more messages in 2D). We also observe that the actual number of messages quadratically decreases with the initial distance, as in the 1D case.

In the 3D case, $\mathbb{E}[M]$ still does not depend on the initial distance, as can be seen on Figure 2.16a and Figure 2.16b. Surprisingly, on Figure 2.18, its value appears to remain around the same value, regardless of ε (it seems to no longer grow when ε is close to 1). There are two possibilities explaining this result. Either $\mathbb{E}[M]$ still grows when ε is close to 1, but the steps we took for ε are too large to see it (on Figure 2.18, ε increases by steps of 0.25). In other words, it is possible that the precision of our simulation is not enough to see the dependency $\mathbb{E}[M]$ has on ε. Another possibility is that in the 3D case, $\mathbb{E}[M]$ no longer depends on ε. If this is the case, then it could be possible to get upper bounds on M that do not depend on ε, improving on the results of this chapter. However, as ε is already a constant, the theoretical benefit would be small.

This qualitative analysis is exactly the same when considering Continuous Movement instead of a Random Walk, as can be seen on figures C.1 through C.3 (Appendix C): M has an upper bound, and depends on ε, except maybe for the 3D case, and the number of messages

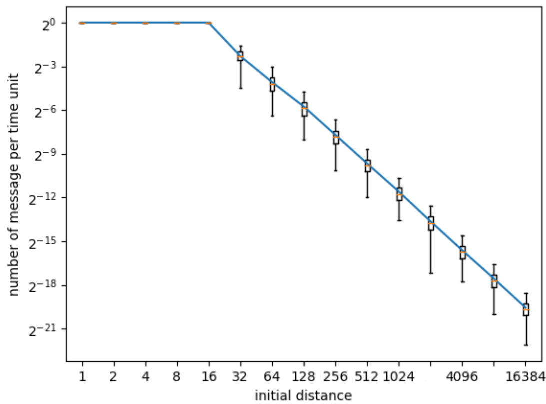


(a) $\varepsilon = 0.1$

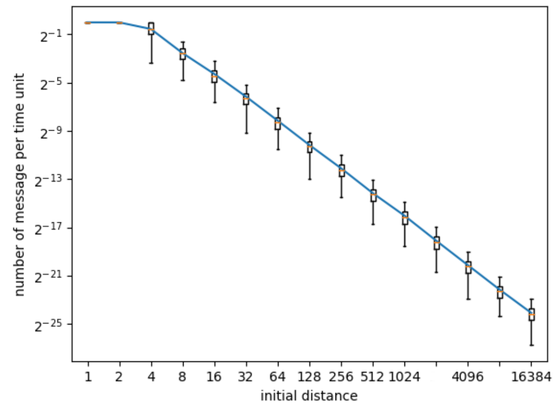


(b) $\varepsilon = 0.5$

Figure 2.10 – Two nodes moving, Random Walk, 1D: M depending on initial distance



(a) $\varepsilon = 0.1$



(b) $\varepsilon = 0.5$

Figure 2.11 – Two nodes moving, Random Walk, 1D: messages per time unit with $\mathcal{A}_{lc}$

generated by $\mathcal{A}_{lc}$ decreases quadratically with the distance.

2.5.2 Actual Traces

Comparison of $\mathcal{A}_{lc}$ with fixed frequency strategies. In order to assess the performance of $\mathcal{A}_{lc}$, we finally compare it to our implementation of the fixed frequency strategy, strategy that is often used in practice in actual games [137], and denoted by $\mathcal{A}_{ff}$. This algorithm does not take a maximal error as parameter, but a fixed wait time w, and nodes send update messages to all other nodes every w time steps.

The traces provided in [33] contain time-stamped information on 98 games of Heroes of Newerth [65] and were used in [32] with the purpose of building mobility models. The files contain the evolution of the positions of 10 players in each game trace: at each time step, we have, for each player, the value of the x coordinate of the position of their character, as well as the y coordinate. As we consider in this experiment a complete connection graph (each node is connected to all the other nodes, see Section 1.2), and because there is a total of 10 characters, a wait time of w induces, on average, $\frac{9\times10}{w}$ messages at each time step.

Even if a smaller w makes information more accurate, $\mathcal{A}_{ff}$ comes without guarantee on maximal error violations, contrarily to $\mathcal{A}_{lc}$. To evaluate the performance of $\mathcal{A}_{ff}$ in terms of

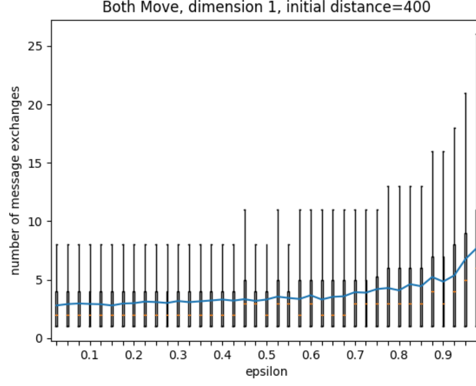
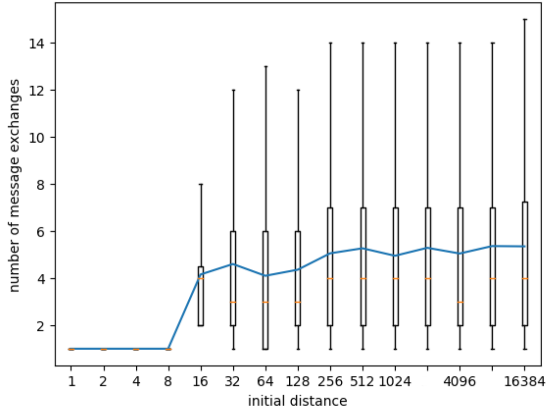
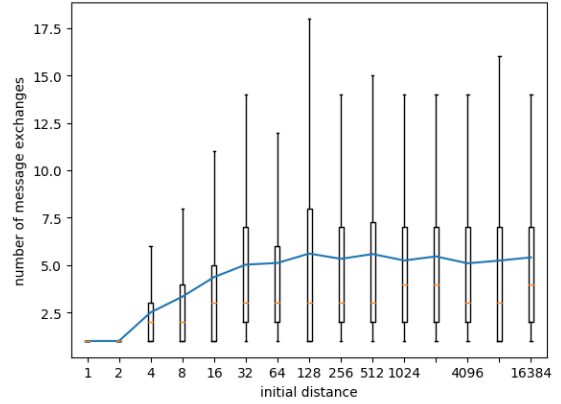


Figure 2.12 – Two nodes moving, Random Walk, 1D: M depending on ε, for $d_0 = 400$



(a) $\varepsilon = 0.1$



(b) $\varepsilon = 0.5$

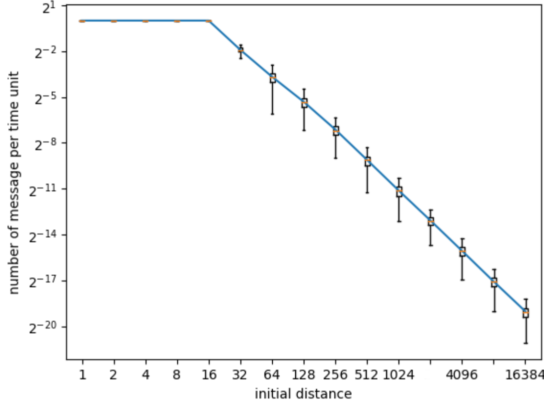
Figure 2.13 – Two nodes moving, Random Walk, 2D: M depending on initial distance

accuracy, we simulated its behavior for several values of ε and w. We counted the *number of violations per time unit*, that is, the number of distance estimates among the nodes that violate Equation 2.2. As there are ten nodes, and each one has an estimate for all nine others, the number of violations has a maximum of 90 for one time unit. Figure 2.19 depicts the number of violations for different values of ε and w. We observe that the number of violations increases very quickly with w.

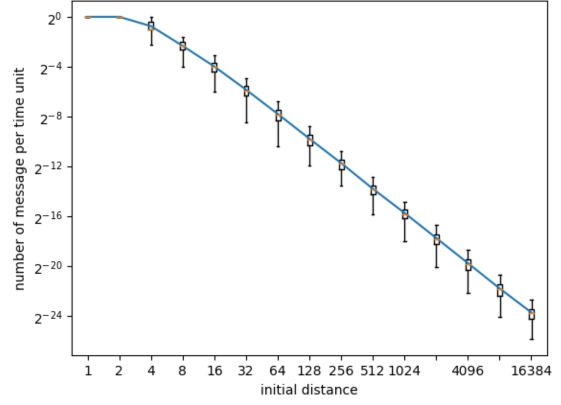
In order to perform a fair comparison between $\mathcal{A}_{lc}$ and $\mathcal{A}_{ff}$, we used the following protocol. First, we ran $\mathcal{A}_{lc}$ for several values of ε, and we measured the resulting average number of messages per time unit. Then, knowing that $\mathcal{A}_{ff}$ induces $\frac{9\times10}{w}$ messages at each time step, we computed the value for w that would result in $\mathcal{A}_{ff}$ sending as much messages on average as $\mathcal{A}_{lc}$. This way, we can compare both algorithms in terms of accuracy (to estimate approximated distance) while they use a similar average message frequency. As w is by definition an integer value, the number of messages sent by both algorithms per time step is not exactly the same, but it is close enough to allow comparison.

The average proportion of violations is shown in bold font in Table 2.2, along with the optimal number of messages, that is, $\mathcal{A}_{id}$, for different values of ε. We can observe that $\mathcal{A}_{lc}$ is far better than $\mathcal{A}_{ff}$ for satisfying Equation 2.2. For instance, it sends only 10.44 messages per time unit for $\varepsilon = 0.1$. With $\mathcal{A}_{ff}$, the only way to ensure Equation 2.2 is by having $w = 1$. This would lead to 90 messages per time unit with $w = 1$, that is, about ten times more than $\mathcal{A}_{lc}$.

Influence of better prediction strategies. As mentioned in Section 1.3.2, Dead-reckoning



(a) $\varepsilon = 0.1$



(b) $\varepsilon = 0.5$

Figure 2.14 – Two nodes moving, Random Walk, 2D: messages per time unit with $\mathcal{A}_{lc}$

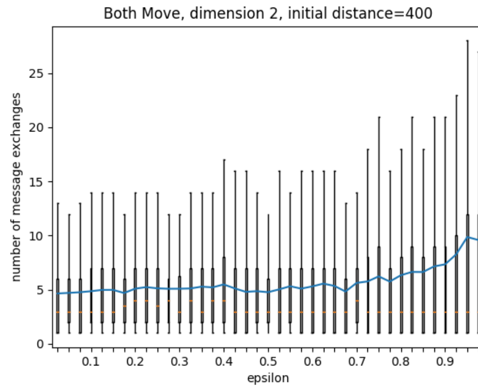


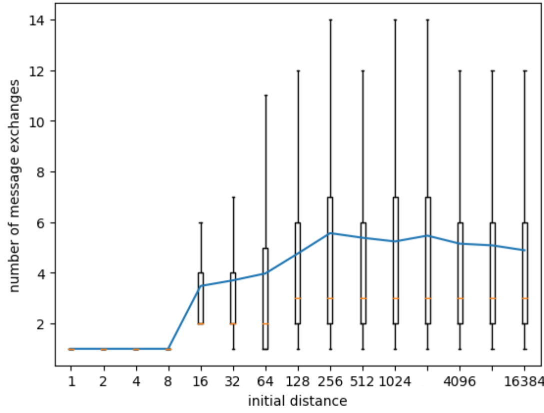
Figure 2.15 – Two nodes moving, Random Walk, 2D: M depending on ε, for $d_0 = 400$

is a popular method for reducing the error on positions of elements of an online game. This is why we wanted to see the benefits induced by Dead-reckoning-like predictions on our algorithm. To do this, as the traces from [33] are in the black-box model, and do not include information on the speed of the nodes, we estimate the speed based on the last two known positions. As in classical dead-reckoning schemes (see Section 1.3.2), this speed is than used to get position estimates by extrapolating the previous known positions. The description of $\mathcal{A}_{lc}$ with dead-reckoning is given in Algorithm 3 (every line that differs from Algorithm 1 is colored in red).

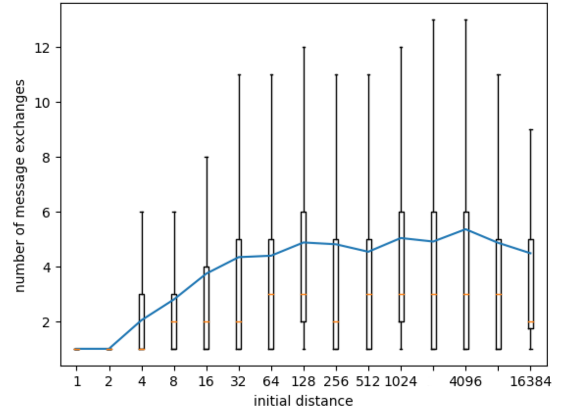
The results of the same experiment as above, but with this prediction algorithm, are shown on Table 2.2, within parentheses.

Table 2.2 – Comparison of $\mathcal{A}_{lc}$ and $\mathcal{A}_{ff}$, **without Dead-reckoning** (with Dead-reckoning)

	$\mathcal{A}_{id}$	$\mathcal{A}_{lc}$		$\mathcal{A}_{ff}$		
ε	msg/time unit	messages per time unit	violations	w	msg/time unit	violations
0.1	**3.26** (2.23)	**10.44** (4.71)	0.0	**9** (19)	**10.00** (4.73)	**2.9%** (5.13%)
0.2	**1.49** (1.24)	**5.41** (3.02)	0.0	**17** (30)	**5.30** (3.00)	**2.74%** (4.66%)
0.3	**0.91** (0.84)	**3.60** (2.26)	0.0	**25** (40)	**3.60** (2.25)	**2.6%** (4.26%)
0.4	**0.63** (0.62)	**2.65** (1.81)	0.0	**34** (50)	**2.65** (1.80)	**2.53%** (3.88%)
0.5	**0.46** (0.46)	**2.07** (1.50)	0.0	**43** (60)	**2.09** (1.50)	**2.42%** (3.51%)

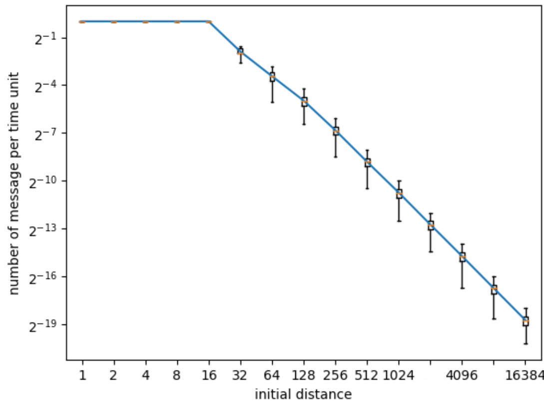


(a) $\varepsilon = 0.1$

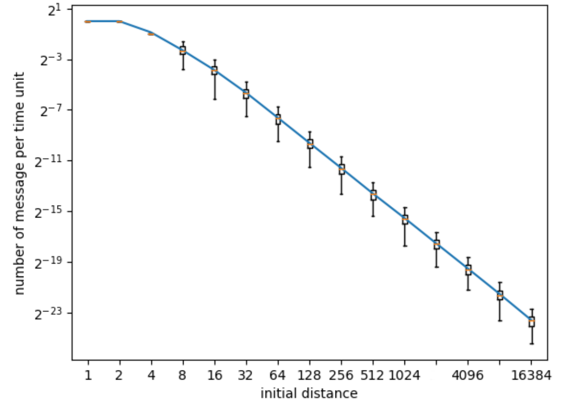


(b) $\varepsilon = 0.5$

Figure 2.16 – Two nodes moving, Random Walk, 3D: M depending on initial distance



(a) for $\varepsilon = 0.1$



(b) $\varepsilon = 0.5$

Figure 2.17 – Two nodes moving, Random Walk, 3D: messages per time unit with $\mathcal{A}_{lc}$

We can observe that the number of message exchanged in $\mathcal{A}_{lc}$ decreases more significantly than $\mathcal{A}_{id}$. Moreover, Dead-reckoning seems to be more beneficial to $\mathcal{A}_{lc}$ than to $\mathcal{A}_{ff}$, as the decrease in message number is not compensated for in terms of violations by the improved prediction precision.

2.6 Conclusion and Future Work

In this chapter, we presented a distributed algorithm $\mathcal{A}_{lc}$, for each node to estimate the distance separating them from each other node, with a relative condition on the error. This type of property is desirable in settings where the proximity between nodes is important, such as online games or networks of moving objects. We proved that (in a restricted setting), this algorithm is optimal in terms of number of message exchanges up a to a constant factor. A summary of our bounds can be found in Table 2.3 (where SM, for Single Move, refers to movement models where only one of the two nodes may move, and BM, for Both Move, the general case where both nodes may move).

We also showed through simulations, based on actual traces of online games, that $\mathcal{A}_{lc}$ performs significantly less communications than the fixed frequency algorithm which is commonly

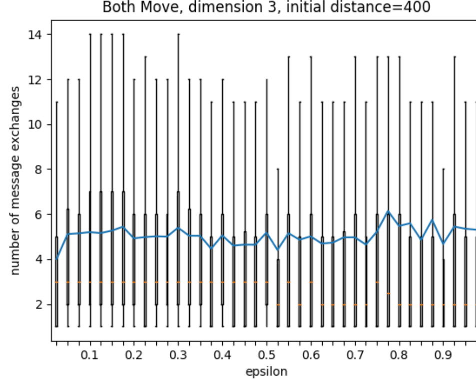


Figure 2.18 – Two nodes moving, Random Walk, 3D: M depending on ε, for $d_0 = 400$

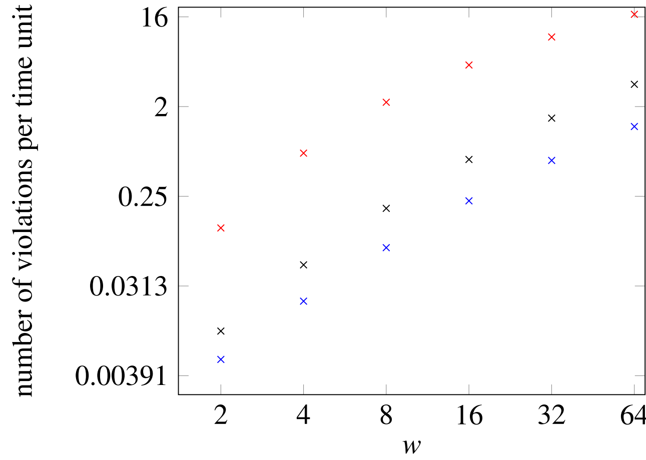


Figure 2.19 – Number of violations with $\mathcal{A}_{ff}$, depending on time to wait between two messages, with $\varepsilon = 0.1$ (red), $\varepsilon = 0.5$ (black), and $\varepsilon = 0.9$ (blue), on log-log scale

used in online games, while bounding the error.

This work opens several perspectives. The first one is to extend the theoretical results proved in this chapter, either by improving the constants or by increasing the scope of the results and to consider more sophisticated prediction algorithms. Another perspective is to extend the set of properties that can be maintained with a (constant) increase in exchanged messages. It was known in the literature that maintaining the positions was possible with no increase in the number of messages and the present chapter showed that a constant increase is enough to maintain relative distances. Extending the class of such properties is highly desirable, both in theory and practice.

In a distributed system with several nodes, each pair of connected nodes may use bandwidth by communicating about their state. In this chapter, we tried to reduce bandwidth consumption by reducing the number of needed messages exchanges between each of these pairs. Another

Table 2.3 – Upper bounds on M

	Random Walk, SM	Random Walk, BM	Continuous Movement, BM
1D case	$\min\left(\Delta_l \times 2^{\Delta_l}\ ;\ \left\lceil \frac{4}{\pi}\Delta_l^2 \right\rceil \times 8\right)$	$\Delta_l \times 4^{\Delta_l}$	$\Delta_l \times 4^{\Delta_l}$
2D case	$\Delta_r \times 4^{\Delta_r}$	$\Delta_r \times 8^{\Delta_r}$	$\Gamma \times 8^{\Gamma}$
3D case	$\Delta_r \times 8^{\Delta_r}$	$\Delta_r \times 16^{\Delta_r}$	$\Gamma \times 14^{\Gamma}$

approach could be to reduce the actual number of connections, that is, try to reduce the number of nodes that are connected while still guaranteeing properties. Chapter 3 and Chapter 4 target this second approach. Also, in these chapters, position and distance estimation are supposed to be separate problems, so that $\mathcal{A}_{lc}$ could be used in conjunction with the other results of this thesis.

Algorithm 3 Local change ($\mathcal{A}_{lc}$) with dead-reckoning-like predictions, from the point of view of Alice

1: Initialization:
2: $\boldsymbol{p_A} \leftarrow$ Alice's initial position (*Actual position of Alice. This is a read-only input to the algorithm*)
3: $\widetilde{p_A} \leftarrow$ Alice's initial position (*Position of Alice, as estimated by Bob, the other node*)
4: $p_A^{-1} \leftarrow$ Alice's initial position; $t_A^{-1} \leftarrow 0$ (*Position, timestamp of previous sent update*)
5: $p_A^{-2} \leftarrow$ Alice's initial position; $t_A^{-2} \leftarrow 0$ (*Position, timestamp of update sent before the previous one*)
6: $\widetilde{p_B} \leftarrow$ Bob's initial position (*Estimated position of Bob*)
7: $p_B^{-1} \leftarrow$ Bob's initial position; $t_B^{-1} \leftarrow 0$ (*Position, timestamp of previous received update*)
8: $p_B^{-2} \leftarrow$ Bob's initial position; $t_B^{-2} \leftarrow 0$ (*Position, timestamp of update received before the previous one*)
9: $d_{est} \leftarrow d(\widetilde{p_A}, \widetilde{p_B})$ (*Estimated distance.*)

10: **procedure** TIME_STEP(i) (*To be executed at each time step i*)
11: update $\boldsymbol{p_A}$
12: $\widetilde{p_A} \leftarrow$ EXTRAPOLATE_POSITION(i, (p_A^{-1}, t_A^{-1}), (p_A^{-2}, t_A^{-2}))
13: $\widetilde{p_B} \leftarrow$ EXTRAPOLATE_POSITION(i, (p_B^{-1}, t_B^{-1}), (p_B^{-2}, t_B^{-2}))
14: $d_{est} \leftarrow d(\widetilde{p_A}, \widetilde{p_B})$
15: ===== communication round 1 =====
16: **if** $d(\boldsymbol{p_A}, \widetilde{p_A}) \geq \frac{\varepsilon}{2} d_{est}$ **then**
17: $p_A^{-2} \leftarrow p_A^{-1}$; $p_A^{-1} \leftarrow \boldsymbol{p_A}$; $t_A^{-2} \leftarrow t_A^{-1}$; $t_A^{-1} \leftarrow i$
18: $\widetilde{p_A} \leftarrow$ EXTRAPOLATE_POSITION(i, (p_A^{-1}, t_A^{-1}), (p_A^{-2}, t_A^{-2}))
19: send message ($\boldsymbol{p_A}$) to Bob

20: ===== communication round 2 =====
21: **if** a message (p_B) has been received from Bob **then**
22: $p_A^{-2} \leftarrow p_A^{-1}$; $p_A^{-1} \leftarrow \boldsymbol{p_A}$; $t_A^{-2} \leftarrow t_A^{-1}$; $t_A^{-1} \leftarrow i$
23: $\widetilde{p_A} \leftarrow$ EXTRAPOLATE_POSITION(i, (p_A^{-1}, t_A^{-1}), (p_A^{-2}, t_A^{-2}))
24: RECEIVE_MESSAGE(p_B, i)
25: send message ($\boldsymbol{p_A}$) to Bob

26: ===== communication round 3 =====
27: **if** a message (p_B) has been received from Bob **then**
28: RECEIVE_MESSAGE(p_B, i)

29: **procedure** EXTRAPOLATE_POSITION(i, (p^{-1}, t^{-1}), (p^{-2}, t^{-2}))
30: $speed \leftarrow \frac{p^{-1} - p^{-2}}{t^{-1} - t^{-2}}$ (*Estimate speed based on previous positions*)
31: **return** $p^{-1} + speed \times (i - t^{-1})$ (*Extrapolate position using that speed*)

32: **procedure** RECEIVE_MESSAGE(p_B, i)
33: $p_B^{-2} \leftarrow p_B^{-1}$; $p_B^{-1} \leftarrow p_B$; $t_B^{-2} \leftarrow t_B^{-1}$; $t_B^{-1} \leftarrow i$
34: $\widetilde{p_B} \leftarrow$ EXTRAPOLATE_POSITION(i, (p_B^{-1}, t_B^{-1}), (p_B^{-2}, t_B^{-2}))
35: $d_{est} \leftarrow d(\widetilde{p_A}, \widetilde{p_B})$

Chapter 3

Distributed Kinetic Data Structure for Proximity Queries in One Dimension: Unit Disc Graphs

3.1 Introduction

Many modern applications can be modeled by sets of distributed nodes that move in a real or virtual environment. In many of those settings, nodes need to retrieve information on other nodes that are close by. For example, players of online video games need to know information about other characters that lie in their character's line of sight, and vehicles may improve security by connecting to other vehicles that are close by. In these use cases, proximity queries (see Section 1.5.1) are often considered.

There is a large literature on static and dynamic data structures for node sets (see Chapter 1). In static models, the set of nodes $\mathcal{V}$ is given and neither the set itself nor the positions of the nodes can change. Dynamic models add a challenge: the structures have to be maintained efficiently when there can be additions of nodes or deletions of nodes to/from $\mathcal{V}$. Updating a dynamic data structure when a single node is added to or removed from the structure should be more efficient than rebuilding an equivalent static data structure from scratch, otherwise the dynamic data structure is pointless.

Designing such a data structure is even more challenging when the nodes are moving, in particular when the movements are not known in advance. The black-box model is considered here (see Section 1.6.1). In this setting, updating the structure when a node moves should be more efficient than simply removing the node from the structure and adding it again with its new position (in the case the structure is also dynamic), and rebuilding the structure from scratch.

Another aspect to consider is that the target proximity query should be treated rapidly: the data structure should support the query in a time complexity smaller than the total number of nodes: otherwise, it is not competitive with a naive strategy that consists in checking all nodes of $\mathcal{V}$, and decide for each whether it should be added to the query result.

Since in this chapter, we consider distributed nodes in a synchronous network, we measure the performance of updates and of query costs in terms of number of needed communication rounds, but also in terms of number of sent messages. The memory cost of the structure will be measured in terms of local memory cost (which corresponds here to the maximal number of connections a node may have with other nodes) and in terms of global cost, that is, the sum, for all nodes in $\mathcal{V}$, of the local memory costs.

3.1.1 Model

Let $\mathcal{V}$ be a set of distributed nodes. Each node $u \in \mathcal{V}$ has a one-dimensional position $p_u(t) \in \mathbb{R}$ at any instant $t \in \mathbb{R}^+$.

We use the Black-box model (see Section 1.7.2): the movements are not known in advance, and nodes may move only at specific, synchronous instants that are called *time steps*. Let us denote by t_i the instant of the i-th time step[1]. Thus, while time is supposed to be continuous, and $t \in \mathbb{R}^+$, we have that for any time step t_i, and any time $t \in [t_i; t_{i+1}[$, $p_u(t) = p_u(t_i)$, so that the movement has a discrete behavior with respect to time.

The time starts at 0. As the first movement happens at the first time step t_1, we note $t_0 = 0$. At time 0, it is supposed that initialization is already completed[2].

At each t_i, the position of a node may move only to a position at most at a distance d_{mv} away from its previous position:

Definition 3.1 (d_{mv})**.** $\forall i \geq 1, \forall u \in \mathcal{V}, p_u(t_i) \in [p_u(t_{i-1}) - d_{mv}; p_u(t_{i-1}) + d_{mv}]$

For simplicity, without loss of generality, we assume that positions are normalized such that $d_{mv} = 1$. The distance between two nodes u and v at instant t is denoted by $d(u, v, t)$.

We do not make any assumption on the number of nodes that move at each time step. We are thus in the high mobility setting (see Section 1.6.1).

Each node has a unique identifier. A node v can send a message to a node u only if v knows u's identifier. Each node $u \in \mathcal{V}$ maintains an address book, a subset of the other nodes[3] in $\mathcal{V}$. The node u may change its address book, either by simply "forgetting" one of the nodes, that is, by removing it from its address book, or by "remembering" a new node, that is, adding a new one to the address book. We suppose a node $v \in \mathcal{V}$ may tell u about a node $w \in \mathcal{V}$, so that u adds w to its address book. We suppose that the sizes of the messages and of the identifiers are so that one message is sufficient for this kind of messages (a node telling to another one that it should add or remove a node from its address book).

Under this definition, it is possible that the contents of the address books are not symmetric: it is possible that u has v in its address book, while v does not know about u, so that u may send messages to v, without v being able to initiate a communication with u.

The union of all address books at instant t defines a connection graph $G(t) = (\mathcal{V}, E(t))$ on the set of nodes (see Definition 1.3, p.20), with $E(t)$ the (directed) edges of the graphs where $(u, v) \in E(t)$ if and only if v is in u's address book at instant t. Because of the changes of address books, E may change with time, hence the notation $E(t)$; however, we are focusing here on the kinetic and non-dynamic setting, and thus we suppose that $\mathcal{V}$ cannot change.

We assume in this chapter that position and distance estimation is perfect: each node has access to the actual position of any node from its address book. This is to allow us to design an algorithm and measure its performance independently of the chosen position and distance estimation techniques. However, this means that in most practical distributed settings, additional messages would have to be exchanged, as nodes need to send positional updates to the nodes that are connected to them. We can however suppose that this number of messages for positional updates is proportional to the number of connections of each node, hence the importance,

[1]Note that the notation differs from how it was used in Chapter 2, where t_i was the time of the i-th exchange of messages. Here t_i is just used as a convenient way to specify a time step, and to be able to refer to the previous and next time step (with t_{i-1} and t_{i+1}).

[2]Note that it would be sufficient to suppose that nodes have a complete and exact knowledge of the positions at initialization, that is, that at $t = 0$, $\forall u, v \in \mathcal{V}$, u knows the value of p_v. Each node would then be able to locally compute a valid initialization, using their identifiers (see later).

[3]In practice, the address book would be a set of identifiers, but for ease of notation, we assume the address book is a set of nodes.

in our performances analysis (Section 3.6), of bounds on the maximal number of connections any node can have.

As mentioned previously, the system is synchronous. It will be supposed that between two position changes, that is, for any $i \geq 0$, between t_i and t_{i+1}, as many communication rounds as needed can be executed, even if we aim at having as few as possible. During any such communication round, a node may execute as many instructions as necessary, and send messages. As explained in Section 1.2, it is guaranteed that the recipient of a message receives it before the beginning of the next communication round.

When time is unambiguous (specifically when some properties should be maintained at any instant t), we may simplify notations by avoiding specifying the instants; for example, we may denote the position of u by p_u, or the connection graph by G.

3.1.2 Objectives

Ideal Objectives

In this chapter, we aim at providing an algorithm maintaining a distributed data structure on moving nodes, such that each node $u \in \mathcal{V}$ can answer to the $\text{CloseNodes}_u(r)$ query, previously defined in Chapter 1: given a node u, return the set of nodes being at most at a distance r from u at the time of the query:

Definition 1.2 ($\text{CloseNodes}_u(r)$)**.** *Given a node $u \in \mathcal{V}$ and a distance $r \in \mathbb{R}$, return all nodes $v \in \mathcal{V}$ such that $d(u, v) \leq r$.*

We consider that r is a *fixed parameter* of the data structure, and we want any node $u \in \mathcal{V}$ to have at any time direct access to the result of $\text{CloseNodes}_u(r)$, without any communication round. In other words, each node u should locally maintain the set of nodes that are at less than r distance units away from u. The structure is thus equivalent to a unit disc graph, with r the "unit" of the disc graph. To the best of our knowledge, no previous work has been published studying the distributed maintenance of unit disc graphs for sets of moving nodes.

The parameter r is fixed, but we suppose the following:

$$r \geq 2d_{mv}. \tag{3.1}$$

The structure to be maintained is the following connection graph:

Definition 3.2 (proximity graph)**.** *The target is the proximity graph $H(t) = (\mathcal{V}, NE(t))$, where $NE(t) = \{(u, v) : d(u, v, t) \leq r\}$.*

This chapter focuses on obtaining a proximity graph for nodes in a synchronous distributed system, where each node is associated to a position in one dimension, moving according to the black-box model.

Approximated Objectives

However, even if communication rounds and local computations can be considered fast with regard to the interval between each movement, we do not suppose that they are instantaneous, hence a small delay can appear between the moment a node moves and the moment when the structure has been corrected accordingly (see Figure 3.3 where the ideal objectives are met only at times that are not hashed in blue). Thus, the structure may have some approximations with regard to the ideal objective.

Because of this delay, the structure provided in this chapter may answer to the CloseNodes query with regard to positions of the nodes slightly from the past: if a node u computes a $\text{CloseNodes}_u(r)$ query after[4] t_i and before the updates associated with the new positions of the nodes at time t_i are finished, then the result may correspond to the positions of the node at time t_{i-1}. However, as the nodes move not more than d_{mv} per time step, the returned set is an approximate version of the CloseNodes query: every node in the returned set is at distance $r + d_{mv}$ from the query node, and any node at distance $r - d_{mv}$ must be in the returned set, as in Definition 3.3.

Definition 3.3 (d_{mv}-$\text{CloseNodes}_u(r)$)**.** *Given a node $u \in \mathcal{V}$ and a distance $r \in \mathbb{R}$, return C a set of nodes such that:*

$$\begin{cases} \forall v \in C, d(u,v) \leq r + d_{mv} \\ \forall v \in \mathcal{V}, d(u,v) \leq r - d_{mv} \implies v \in C \end{cases} \tag{3.2}$$

The proximity graph can be approximated similarly, but we do not need to define the approximation, as the delay can be accounted for by specifying the timestamp of the considered proximity graph.

3.2 The Connection Graph

In this section, we consider any instant t of the execution, and we will thus omit to specify the time in our notations.

In order to achieve our goal, and to get the proximity graph of Definition 3.2, we will ensure that the actual connection graph G is a supergraph of H, the proximity graph.

For G to be a supergraph of H, G needs to have among its arcs the set of *Neighboring edges*, already used in Definition 3.2:

$$NE = \{(u,v) : d(u,v) \leq r\}. \tag{3.3}$$

When maintaining this set of arcs, it is easy to handle the disconnection of a node: as soon as a node u sees that another node v gets at a distance higher than r from u, u may simply remove v from its address book. However, the problem of knowing when a node previously far away gets closer than r is more complicated, so that additional edges are needed on G.

We will associate each node u with two specific nodes called its *lookouts*, one for each directions. The role of these lookouts is to inform u of any node that gets at distance less than r from u. The lookouts are denoted by ℓ_u^+ (for the direction of higher coordinates) and ℓ_u^- (for the direction of lower coordinates). In some specific cases however, u may have no lookout for one or even both directions; in this case, we may note $\ell_u^+ = \perp$ (resp. $\ell_u^- = \perp$) if there is no ℓ_u^+ (resp. ℓ_u^-) lookout.

We will say that ℓ_u^+ is valid if it satisfies:

$$\begin{cases} (\exists v \text{ s.t. } p_u + r < p_v) \implies \ell_u^+ \neq \perp \\ \text{If } \ell_u^+ \neq \perp,\ p_u + r < p_{\ell_u^+} \\ \text{If } \ell_u^+ \neq \perp,\ \nexists v \text{ s.t. } p_u + r < p_v < p_{\ell_u^+} - r \end{cases} \tag{3.4}$$

The same goes for ℓ_u^-, but the other way round:

$$\begin{cases} (\exists v \text{ s.t. } p_v < p_u - r) \implies \ell_u^- \neq \perp \\ \text{Si } \ell_u^- \neq \perp,\ p_{\ell_u^-} < p_u - r \\ \text{Si } \ell_u^- \neq \perp,\ \nexists v \text{ s.t. } p_{\ell_u^-} + r < p_v < p_u - r \end{cases} \tag{3.5}$$

[4]Recall that, as defined in Section 3.1.1, t_i is the instant of the i-th movement of the nodes.

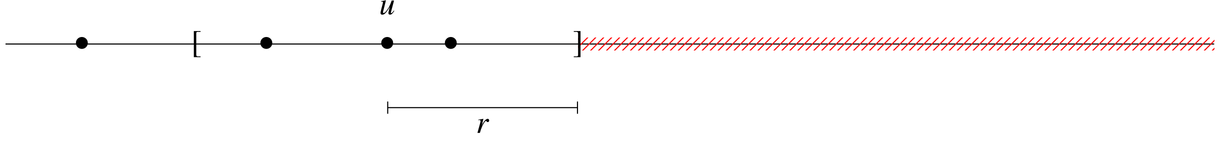


Figure 3.1 – Example of a situation where we must have $\ell_u^+ = \perp$. The interval $]p_u + r; +\infty[$, hatched in red, does not contain any node.

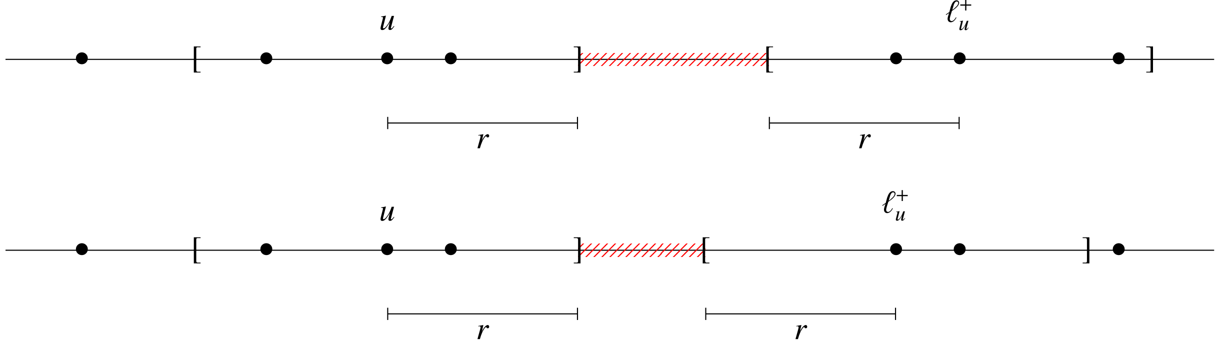


Figure 3.2 – Example of a situation where a node u may have one of two nodes as lookout. In both cases, there is no node in $]p_u + r; p_{\ell_u^+} - r[$, interval represented as red hatched zones.

These definitions of lookouts are such that, when there is no node with a position that is contained in $]p_u + r; +\infty[$, then we must have $\ell_u^+ = \perp$ (and similarly for ℓ_u^-). This situation is represented on Figure 3.1, and is due to the second condition of Equation 3.4. It corresponds to nodes with extreme positions, that is, nodes that are among the furthest on the right or on the left, and thus do not have (and do not need) available lookouts in one of the directions (or in some rare cases, both).

Another important property is that the lookouts are not uniquely defined: at a given moment, there may exist several choices as to which node to take as a lookout for u. For example, on Figure 3.2, the same set of nodes is represented twice, with two possible assignments for ℓ_u^+. This adds some "leniency" to the structure, which is a desirable property to avoid constantly executing updates when the nodes move, as explained in [55], where some elements of the structure are also not uniquely defined.

As told previously, the role of the lookouts is to inform u of any node that gets at distance r from u. Even with this leniency, lookouts are defined in such a way, that if a node v, previously too far away, moves at a distance smaller than r from u, then v was necessarily at distance less than r from one of the lookouts of u. Lookouts are thus able to detect this situation, thanks to their own neighboring edges, and can send messages to u in order for u to add v to its address book. This will be proven later in Lemma 3.10.

Let us suppose that any node $u \in \mathcal{V}$ is assigned valid lookouts, that is, ℓ_u^+ and ℓ_u^- are valid. We add these to the connections of u. In other words, the arcs of G must include the following two sets:

$$LE^+ = \bigcup_{\substack{u \in \mathcal{V}, \\ \ell_u^+ \neq \perp}} \{(u, \ell_u^+)\} \text{ et } LE^- = \bigcup_{\substack{u \in \mathcal{V}, \\ \ell_u^- \neq \perp}} \{(u, \ell_u^-)\}. \quad (3.6)$$

However, as the lookouts need to send messages to the nodes of whom they are the lookout, the reverse arcs have also to be added to G, with SE^+ the reverse of LE^+, and SE^- the reverse of

LE^- :

$$SE^+ = \bigcup_{u \in \mathcal{V}} \{(u, v) : u = \ell_v^+\} \text{ et } SE^- = \bigcup_{u \in \mathcal{V}} \{(u, v) : u = \ell_v^-\}. \quad (3.7)$$

The data structure we defined until now (NE, ℓ_u^+, ℓ_u^-, SE^+, and SE^-) would be sufficient to maintain the structure: each time a lookout u sees that one of its neighbors v (that is, $(u, v) \in NE$) gets too close to one of the nodes w of whom it is the lookout (that is, either $(u, w) \in SE^+$, or $(u, w) \in SE^-$), then u can send messages to w so that w adds v to its address book (the symmetric would be handled by v's lookout). However, this requires the lookouts to verify a lot of conditions in the case where u is the lookout for many other nodes. Another possibility that better spreads the local computation costs, would be for u to send its neighbors to w at each time step, which takes only one communication round, so that w may compute itself which nodes to add to its address book. Instead of this, to get an algorithm that is easier to read, we made the choice to add to the connections of every node u, the nodes that are neighbors of u's lookouts. The arcs of G thus include the following two sets:

$$LNE^+ = \bigcup_{\substack{u \in \mathcal{V}, \\ \ell_u^+ \neq \bot}} \{(u, v) : d(v, \ell_u^+) \leq r\} \text{ et } LNE^- = \bigcup_{\substack{u \in \mathcal{V}, \\ \ell_u^- \neq \bot}} \{(u, v) : d(v, \ell_u^-) \leq r\}. \quad (3.8)$$

The aim is to maintain a connection graph G that includes all of these sets of arcs. Let us define $\mathcal{G}$, the set of connection graphs that fulfill this condition:

Definition 3.4. $\mathcal{G} = \{(\mathcal{V}, E)$ *such that* $\forall u \in \mathcal{V}, \ell_u^+$ *and* ℓ_u^- *are valid and* $NE \cup LE^+ \cup LE^- \cup SE^+ \cup SE^- \cup LNE^+ \cup LNE^- \subseteq E\}$.

3.3 Data Structure

3.3.1 Local Variables

In order to get a connection graph $G \in \mathcal{G}$, each node $u \in \mathcal{V}$ will maintain the following set of local variables, the union of which will be its address book:

- $N_u(t)$, a set of nodes we call the *neighbors*[5] of u. This set is initialized to $N_u(0) = \{v : d(u, v, 0) \leq r\}$. We will say that $N_u(t)$ is valid with respect to the positions of the nodes at instant t' if $N_u(t) = \{v : d(u, v, t') \leq r\}$. Note that if the set of neighbors is valid, we have $u \in N_u(t)$.

- $\ell_u^+(t)$, the *positive lookout* of u, which takes at instant 0 the value of a node that satisfies Equation 3.4.

- $\ell_u^-(t)$, the *negative lookout* of u, which takes at instant 0 the value of a node that satisfies Equation 3.5.

- $S_u^+(t)$, the set of *positive supervised* nodes, that is, the set of nodes of whom u is the positive lookout. This set is initialized to $S_u^+(0) = \{v : \ell_v^+(0) = u\}$, and $S_u^+(t)$ is said to be valid if $S_u^+(t) = \{v : \ell_v^+(t) = u\}$.

- $S_u^-(t)$, the set of *negative supervised* nodes, whose initialization and definition of validity is symmetric to the positive supervised nodes.

[5]Note that this definition differs from the "neighbors" of u in the graph theory sense (where the neighbors are the nodes that are one hop away from u in a graph).

- $L_u^+(t)$, the set of neighbors of $\ell_u^+(t)$, initialized to $L_u^+(0) = \{v : d(v, \ell_u^+(0), 0) \leq r\}$ (that equals to $\emptyset$ if $\ell_u^+(0) = \perp$). Again, we say that $L_u^+(t)$ is valid with regard to the positions of the nodes at instant t' if $L_u^+(t) = \{v : d(v, \ell_u^+(t), t') \leq r\}$. Note that when $L_u^+(t)$ is valid, we have $\ell_u^+(t) \in L_u^+(t)$, and $L_u^+(t) = \emptyset$ if $\ell_u^+(t) = \perp$.

- $L_u^-(t)$, the neighbors of $\ell_u^-(t)$, whose initialisation and definition of validity is symmetric to $L_u^+(t)$.

These notations represent the current values of the variable: $N_u(t)$ is the content of the variable at time t. We may thus omit the time, when we refer to the variable itself, or if the time is unambiguous.

Note also that these notations represent nodes and sets of nodes, independently of their validity. It is thus possible to have for example that $\ell_u^+(t)$ is valid with regard to the positions of the nodes at instant t', with $t \neq t'$. This distinction is important to take into account the delay of updates as described in Section 3.1.2.

The connection graph $G(t)$ is then the directed graph[6] constructed from the local variables of each node at instant t. As these variables correspond to the set of edges used in Definition 3.4, we have the following.

Remark 3.5. *If, for all $u \in \mathcal{V}$, the local variables $N_u(t)$, $\ell_u^+(t)$, $\ell_u^-(t)$, $S_u^+(t)$, $S_u^-(t)$, $L_u^+(t)$, and $L_u^-(t)$ are valid with regard to the positions of the nodes at instant t', then $G(t) \in \mathcal{G}(t')$.*

3.3.2 Certificates

In order to maintain the structure when nodes move, we will use certificates[7], as it is usually done with Kinetic Data Structures (see Section 1.7). Certificates are predicates involving a constant number of nodes, that together validate the structure: in this case, it means that if at instant t, $G(t) \in \mathcal{G}(t)$, and the nodes move according to the model described in Section 3.1.1, and all certificates remain true until $t' > t$, then we have necessarily $G(t') \in \mathcal{G}(t')$. More formally, with $Cert_u(t)$ the set of certificates at time t that involve u:

Definition 3.6 (Certificate Validity)**.**

$$\forall u \in \mathcal{V}, \forall t < t' \in \mathbb{R}, (G(t) \in \mathcal{G}(t) \wedge \forall t'' \in [t, t'], \forall c \in Cert_u(t''), c \text{ } true\ ) \implies G(t') \in \mathcal{G}(t') \tag{3.9}$$

As the movement of the nodes is not known in advance, the certificates have to be reassessed at each instant t_i. As soon as it is detected that one (or several) of them becomes false, the associated nodes execute updates, described in Section 3.4, so as to be again in a state in which the structure and the certificates are valid.

Each node u maintains a set of four different types of certificates:

- Close Neighbor: certifies that a neighbors remains at a distance smaller than r. For each $v \in N_u$, u maintains the certificate $\textsc{CloseNeighbor}(u, v) = d(u, v) \leq r$.

- Potential Neighbor: detects when a node has gotten close enough to become a neighbor. For each $v \in (L_u^+ \cup L_u^-) \setminus N_u$, u maintains the certificate $\textsc{PotentialNeighbor}(u, v) = d(u, v) > r$.

[6]Even though the graph is directed, all arcs except those from L^+ and L^- are actually present in both directions.
[7]*Incremental* certificates to be precise (see Section 1.7.1).

- Distant Lookout: certifies that the lookout remains distant enough. For each $v \in \{\ell_u^+, \ell_u^-\} \setminus \{\bot\}$, u maintains the certificate $\textsc{DistantLookout}(u, v) = d(u, v) > r$.
- Legitimate Lookout: detects when a node gets in between a node and its lookout in such a way that the lookout becomes invalid. For each $v \in N_u \cup L_u^+$, u maintains the certificate $\textsc{LegitimateLookout}^+(u, v) =$

$$\begin{cases} p_v \notin \left]p_u + r\,;\ p_{\ell_u^+} - r\right[& \text{if } \ell_u^+ \neq \bot \\ p_v \leq p_u + r & \text{if } \ell_u^+ = \bot \end{cases}$$

and for each $v \in N_u \cup L_u^-$, u maintains the certificate $\textsc{LegitimateLookout}^-(u, v) =$

$$\begin{cases} p_v \notin \left]p_{\ell_u^-} + r\,;\ p_u - r\right[& \text{if } \ell_u^+ \neq \bot \\ p_u - r \leq p_v & \text{if } \ell_u^+ = \bot. \end{cases}$$

A summary of these certificates can be found on Table 3.1. The column “Certificate” shows the name of the certificate, the column “Predicate” gives the logical definition of the certificate, and the column “Condition” indicates under which conditions the certificate has to be created.

Table 3.1 – List of certificates associated with node u.

Certificate	**Predicate**	**Condition**
$\textsc{CloseNeighbor}(u, v)$	$d(u, v) \leq r$	$v \in N_u$
$\textsc{PotentialNeighbor}(u, v)$	$d(u, v) > r$	$v \in (L_u^+ \cup L_u^-) \setminus N_u$
$\textsc{DistantLookout}(u, v)$	$d(u, v) > r$	$v \in \{\ell_u^+, \ell_u^-\} \setminus \{\bot\}$
$\textsc{LegitimateLookout}^+(u, v)$	$\begin{cases} p_v \notin \left]p_u + r\,;\ p_{\ell_u^+} - r\right[& \text{if } \ell_u^+ \neq \bot \\ p_v \leq p_u + r & \text{if } \ell_u^+ = \bot \end{cases}$	$v \in N_u \cup L_u^+$
$\textsc{LegitimateLookout}^-(u, v)$	$\begin{cases} p_v \notin \left]p_{\ell_u^-} + r\,;\ p_u - r\right[& \text{if } \ell_u^+ \neq \bot \\ p_u - r \leq p_v & \text{if } \ell_u^+ = \bot \end{cases}$	$v \in N_u \cup L_u^-$

Note that these definitions of certificates come with redundancies. Firstly, some of the certificates are replicated symmetrically on the nodes: for example, as every time $v \in N_u$, we are supposed to have $u \in N_v$ (which will be proven with Lemma 3.10) the certificate $\textsc{CloseNeighbor}(u, v)$ and $\textsc{CloseNeighbor}(v, u)$ will correspond to the same predicate, and will thus differ only by the node on which they are executed.

More generally, some certificates have exactly the same predicate definition, but differ only by the nodes on which they apply: for example, both with $v \in N_u$, and $v = \ell_u^+$, $\textsc{PotentialNeighbor}(u, v)$ and $\textsc{DistantLookout}(u, v)$ track the same predicate. This distinction is useful for the algorithm, because the update rules are different depending on the kind of certificate that fails: a failure of $\textsc{PotentialNeighbor}(u, v)$ indicates that the neighbors have to be updated, and a failure of $\textsc{DistantLookout}(u, v)$ indicates that the lookout has to be changed. In some cases, those certificates may actually apply on the same nodes: as $\ell_u^+ \in L_u^+$ (except when $\ell_u^+ = \bot$, we have $\textsc{PotentialNeighbor}(u, \ell_u^+) = \textsc{DistantLookout}(u, \ell_u^+)$). The same goes for example with $\textsc{LegitimateLookout}^+(u, v)$, the predicate of which is redundant with $\textsc{CloseNeighbor}(u, v)$ when $\ell_u^+ = \bot$.

3.4 Updating the Structure

In this section, we present $\mathcal{A}_{f1d}$, the *Fixed-1d* algorithm, that we will use to update the data structure presented above.

The certificates from the previous section are all associated with some update rules. As described in Algorithm 4, at each instant t_i, $\mathcal{A}_{f1d}$ makes so that the nodes check each of their certificates, and execute the operations associated with those that fail. For any $i \geq 0$, the number of communication rounds executed between instant t_i and t_{i+1} is constant.

The updates to execute are described in algorithms 5, 6, and 7. As some of the variables are symmetric (like ℓ^+ and ℓ^- or L^+ and L^-), and with similar associated updates, in order to avoid lengthy redundancies, some of these updates are skipped, like on line 31 of Algorithm 5.

Note that as the positions change only at instants t_i, we have that for any $i \geq 0$, any $u \in \mathcal{V}$, and any $t \in [t_i; t_{i+1}[$, $p_u(t) = p_u(t_i)$.

In Algorithm 4, the conditions to detect events are written explicitly, without invoking the certificates described previously. However, these conditions (lines 4 through 18) are equivalent to the definitions of the certificates from Table 3.1. This is for better readability, and for easier readability of the proofs. As discussed on page 45, in Section 1.7.2 it can be beneficial in some situations, even in the Black-Box model, to directly use certificates, associated with an event-queue. It is straightforward to convert Algorithm 4 for use with an event queue: instead of checking, at each time step, each node that might be involved in a failing certificate, like on lines 4 through 18, a node just has to take all certificates at the top of the priority queue that are failing, and add them to *Fail*, the set of failing certificates.

Algorithm 4 $\mathcal{A}_{f1d}$, the algorithm with continuous time based on lookouts, point of view of u

---

1: At each instant t_i, start the following:
2: $Fail \leftarrow \emptyset$
3: (*On lines 4 through 18, it is checked, for all nodes in u's local data structure, if the associated certificates are valid. The certificates that are not valid are added to *Fail*.*)
4: **for all** $v \in N_u$ so that $d(u, v) > r$ **do**
5: $\quad Fail \leftarrow Fail \cup \{(\text{CloseNeighbor}, u, v)\}$
6: **end for**
7: **for all** $v \in L_u^+ \cup L_u^- \setminus N_u$ so that $d(u, v) \leq r$ **do**
8: $\quad Fail \leftarrow Fail \cup \{(\text{PotentialNeighbor}, u, v)\}$
9: **end for**
10: **for all** $v \in \{\ell_u^+\} \cup \{\ell_u^-\} \setminus \{\perp\}$ so that $d(u, v) \leq r$ **do**
11: $\quad Fail \leftarrow Fail \cup \{(\text{DistantLookout}, u, v)\}$
12: **end for**
13: **for all** $v \in N_u \cup L_u^+$ so that $p_u + r < p_v$ and, if $\ell_u^+ \neq \perp$, so that $p_v < p_{\ell_u^+} - r$ **do**
14: $\quad Fail \leftarrow Fail \cup \{(\text{LegitimateLookout}^+, u, v)\}$
15: **end for**
16: **for all** $v \in N_u \cup L_u^-$ so that $p_v < p_u - r$ and, if $\ell_u^- \neq \perp$, so that $p_{\ell_u^+} + r < p_v$ **do**
17: $\quad Fail \leftarrow Fail \cup \{(\text{LegitimateLookout}^-, u, v)\}$
18: **end for**
19: Correct_Legitimate_and_Distant_Lookout(*Fail*) (*Get a new valid lookout if current one is invalid*)
20: Correct_Close_Neighbor(*Fail*) (*Drop neighbors that are now too far away*)
21: Correct_Potential_Neighbor(*Fail*) (*Get all new neighbors*)

Algorithm 5 Correct_Legitimate_and_Distant_Lookout: update lookouts (point of view of u)

22: **procedure** CORRECT_LEGITIMATE_AND_DISTANT_LOOKOUT($Fail$)
23: ===== round 1: see if lookout has to be changed =====
24: $C \leftarrow \{v : (\text{LEGITIMATELOOKOUT}^+, u, v) \in Fail\}$
25: **if** $C \neq \emptyset$ **then**
26: Let $newLookout \leftarrow \arg\min_{x \in C}(p_x)$
27: **else if** $(\text{DISTANTLOOKOUT}, u, \ell_u^+) \in Fail$ **then**
28: Send REQ(+) to ℓ_u^+ (*Ask the previous lookout to give a new valid lookout*)
29: **else**
30: $newLookout \leftarrow \ell_u^+$ (*$newLookout$ defaults to no change*)
31: [...] (*symmetrical for handling $\text{LEGITIMATELOOKOUT}^-(u, v)$ and $\text{DISTANTLOOKOUT}(u, \ell_u^-)$*)
32: ===== round 2: handle the reception of a REQ =====
33: **if** REQ(+) has been received from w **then**
34: Let $C \leftarrow \{x \in N_u \cup L_u^+ : d(w, x) > r\}$
35: Let $c \leftarrow \arg\min_{x \in C}(p_x)$ (*$c \leftarrow \perp$ if $C = \emptyset$*)
36: Send REPLY(c, +) to w (*Send new lookout to the asking node*)
37: [...] (*symmetrical for handling REQ($-$)*)
38: ===== round 3: update lookout after receiving the response =====
39: **if** REPLY(v, +) has been received **then**
40: $newLookout \leftarrow v$
41: **if** $newLookout \neq \ell_u^+$ **then**
42: Send REMOVE(S^+, u) to ℓ_u^+ (*inform previous lookout*)
43: $\ell_u^+ \leftarrow newLookout$
44: **if** $newLookout = \perp$ **then**
45: $L_u^+ \leftarrow \emptyset$
46: **else**
47: Send ADD(S^+, u) to $newLookout$ (*inform new lookout*)
48: [...] (*symmetrical for handling REPLY(v, $-$) messages, and ℓ_u^-*)
49: ===== round 4: wait one round, and maybe handle messages =====
50: **if** REMOVE(S^+, v) has been received **then**
51: $S_u^+ \leftarrow S_u^+ \setminus \{v\}$
52: **if** ADD(S^+, v) has been received **then**
53: $S_u^+ \leftarrow S_u^+ \cup \{v\}$
54: Send MYNEIGHBORS(w, +) to v for each $w \in N_u$. (*u sends its neighbors to the node that new has u as lookout*)
55: [...] (*symmetrical for handling ADD(S^-, v), and sending MYNEIGHBORS(N_u, $-$)*)
56: ===== round 5: receive neighbors of new lookout =====
57:

$$L_u^+ \leftarrow \begin{cases} L_u^+ & \text{if no MYNEIGHBORS}(w, +) \text{ has been received} \\ \bigcup w & \text{where a message MYNEIGHBORS}(w, +) \text{ has been received otherwise} \end{cases}$$

58: [...] (*symmetrical for handling MYNEIGHBORS(N, $-$)*)

Algorithm 6 Correct_Close_Neighbor: drop points that are no longer neighbors (point of view of u)

```
59: procedure CORRECT_CLOSE_NEIGHBOR(Fail)
60:     for all (CLOSENEIGHBOR, u, v) ∈ Fail do     (*v ∈ N_u ∧ d(u, v, t_i) > r*)
61:         N_u ← N_u \ {v}
62:         Send REMOVE(L+, v) to each w in S+_u
63:         Send REMOVE(L−, v) to each w in S−_u
64:     end for
65:     ===== round 6: receive REMOVE messages =====
66:     if REMOVE(L+, v) has been received then
67:         L+_u ← L+_u \ {v}
68:     if REMOVE(L−, v) has been received then
69:         L−_u ← L−_u \ {v}
```

Algorithm 7 Correct_Potential_Neighbor: add new neighbors (point of view of u)

```
70: procedure CORRECT_POTENTIAL_NEIGHBOR(Fail)
71:     for all (POTENTIALNEIGHBOR, u, v) ∈ Fail do     (*v ∈ (L+_u ∪ L−_u) \ N_u ∧ d(u, v, t_i) ≤ r*)
72:         N_u ← N_u ∪ v
73:         Send ADD(L+, v) to each w ∈ S+_u
74:         Send ADD(L−, v) to each w ∈ S−_u
75:     end for
76:     ===== round 7: receive ADD messages =====
77:     if ADD(L+, v) has been received then
78:         L+_u ← L+_u ∪ {v}
79:     [...]    (*symmetrical for handling ADD(L−, v)*)
```

3.5 Validity

In this section, we prove that we get the proximity graph as defined in Section 3.1.2, albeit with a slight delay:

Theorem 3.7. *With nodes in one dimension that follow the model described in Section 3.1.1, and such that the connection graph is initialized so that $G(0) \in \mathcal{G}(0)$, $\mathcal{A}_{f1d}$ maintains a connection graph G so that for any $i \geq 0$, $G(t_{i+1}) \in \mathcal{G}(t_i)$.*

In other words, the connection graph resulting from the updates initiated at time t_i (and that is obtained at the latest at t_{i+1}) belongs to the set of graphs that are correct with regard to the positions the nodes had at instant t_i. This situation is represented on Figure 3.3. As the nodes do not move between t_i and t_{i+1}, we have, for any $t \in [t_i; t_{i+1}[$, $\mathcal{G}(t) = \mathcal{G}(t_i)$. As the computations that are initiated at time t_i finish at a time $t' \in [t_i; t_{i+1}[$, and the connection graph does not change until the next movement, we have $\forall t \in [t'; t_{i+1}[$, $G = G(t_{i+1})$. Thus, the connection graph is correct only between instant t' and t_{i+1}, which can be summarized in $G(t_{i+1}) \in \mathcal{G}(t_i)$, explaining the delay of Theorem 3.7.

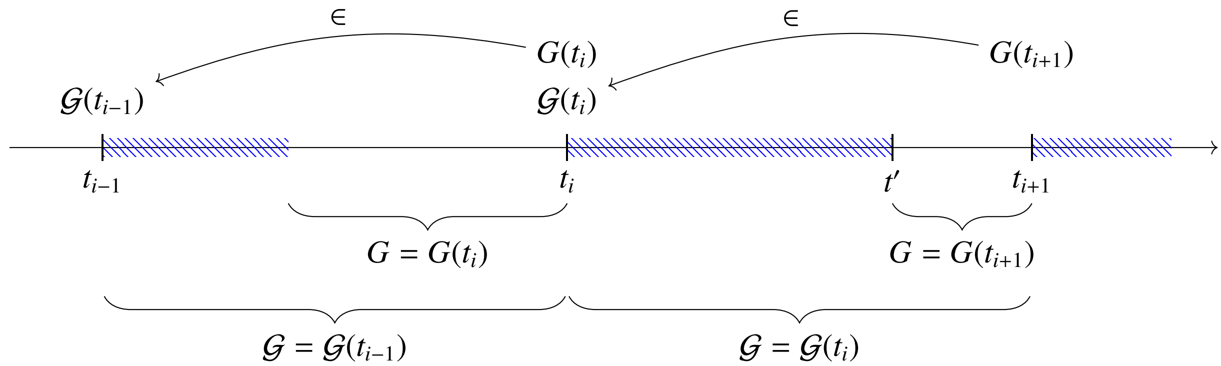


Figure 3.3 – Visualization of the instants of movement and the associated graphs and updates. The blue hashed zones represent the execution of the updates by $\mathcal{A}_{f1d}$.

To prove Theorem 3.7, we proceed in several steps, showing that after the execution of each of the procedures of $\mathcal{A}_{f1d}$, additional properties are verified, until it is proven that the whole connection graph complies to Theorem 3.7. First, we prove in Lemma 3.8 that after the execution of procedure CORRECT_LEGITIMATE_AND_DISTANT_LOOKOUT from Algorithm 5, it is guaranteed that the lookouts (ℓ) and the sets of supervised nodes (S) are valid. Then, in Lemma 3.9, we prove that after the execution of CORRECT_CLOSE_NEIGHBOR from Algorithm 6, all nodes that are no longer neighbors have been filtered out of the sets of neighbors (N) and of the sets of neighbors of the lookouts (L). Finally, we prove in Lemma 3.10 that after execution of CORRECT_POTENTIAL_NEIGHBOR from Algorithm 7, the neighbors and neighbors of lookouts are valid.

As some variables are symmetric, we write only the proofs for the "+" variables. Thus, we prove for example that ℓ_u^+ is valid, but we do not explicitly prove that ℓ_u^- is valid too. The proofs for the symmetric variables are very similar to the proofs below.

Lemma 3.8. *Let t_i be so that $G(t_i) \in \mathcal{G}(t_{i-1})$.*

After the execution of the updates associated with the failure of certificates DISTANTLOOKOUT, LEGITIMATELOOKOUT$^+$ *and* LEGITIMATELOOKOUT$^-$ *of instant t_i, that is, after the execution of procedure* CORRECT_LEGITIMATE_AND_DISTANT_LOOKOUT*, line 19 of Algorithm 4), $\forall u \in \mathcal{V}$, ℓ_u^+ and S_u^+ are valid with regard to the positions of the nodes at time t_i.*

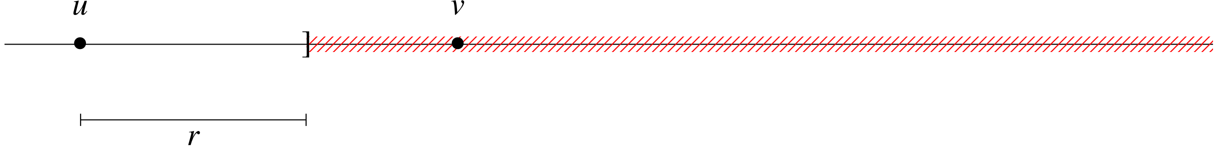


Figure 3.4 – Case 1: We may prove that v was a neighbor of u before the movement.

This remains true until t_{i+1}: $\ell_u^+(t_{i+1})$ and $S_u^+(t_{i+1})$ are valid with regard to the positions of the nodes at time t_i.

Proof. Let us denote by $\tilde{t}$ the instant where $\mathcal{A}_{f1d}$ finishes executing the procedure CORRECT_LEGITIMATE_AND_DISTANT_LOOKOUT. Let u be a node from $\mathcal{V}$. To prove the first part of the lemma, we have to show that $\ell_u^+(\tilde{t})$ and $S_u^+(\tilde{t})$ are valid with regard to the positions of the nodes at time t_i.

First, let us prove that the lookout does not change when it does not need to be changed: if $\ell_u^+(t_i)$ is still valid with regard to the positions of the nodes at instant t_i, then $\ell_u^+(\tilde{t}) = \ell_u^+(t_i)$.

Notice that in $\mathcal{A}_{f1d}$, ℓ_u^+ can change only on line 43 of Algorithm 5. The value affected to the lookout on this line depends on the value of the variable *newLookout*, that can come only from three places.

- Line 30: by default, *newLookout* takes the value of $\ell_u^+(t_i)$, that is, the previous value of the lookout. In other words, if none of the situations below happens, than the lookout does not change.

- Line 26, only if there is at least one $v \in N_u(t_i) \cup L_u^+(t_i)$ so that LEGITIMATELOOKOUT$^+(u, v)$ was false at t_i. Thus this change of lookout occurs only if $\ell_u^+(t_i)$ is no longer valid with regard to the positions of the nodes at instant t_i.

- Line 40, when a REPLY message has been received. In turn, this message has necessarily been sent by ℓ_u^+ (line 36) as a response to a REQ message from u (line 28). As the REQ message is sent only if DISTANTLOOKOUT$(u, \ell_u^+(t_i))$ was false at t_i, this change of lookout also may only happen when the previous lookout is no longer valid.

A first conclusion that can be made from these remarks is that the lookout may change only if it is not valid at time t_i.

Let us now show that if $\ell_u^+(t_i)$ is not valid with regard to the positions of the nodes at instant t_i, then the updates make sure that $\ell_u^+(\tilde{t})$ is. For this, we rely on the assumption stating that $G(t_i) \in \mathcal{G}(t_{i-1})$.

There are only three situations in which $\ell_u^+(t_i)$ is not valid with regard to the positions at t_i, and all three situations are disjoint.

- **Case 1 :** $\ell_u^+(t_i) = \perp$, and $\exists v : p_u(t_i) + r < p_v(t_i)$ (see Figure 3.4).

  At line 24 of Algorithm 5, the set C of candidates for $\ell_u^+(\tilde{t})$ is the set of nodes v such that LEGITIMATELOOKOUT$^+(u, v)$ failed at t_i. As it is supposed in this case that $\ell_u^+(t_i) = \perp$, we have $C = \{v \in N_u(t_i) : p_u(t_i) + r < p_v(t_i)\}$ (see line 13, Algorithm 4; note that $\ell_u^+(t_i) = \perp \wedge G(t_i) \in \mathcal{G}(t_{i-1}) \implies L_u^+(t_i) = \emptyset$).

  However, we have $G(t_i) \in \mathcal{G}(t_{i-1})$, and thus in particular $\ell_u^+(t_i) = \perp$ is valid with regard to the positions of the nodes at time t_{i-1}. Therefore, for any node $v \in \mathcal{V}$, $p_v(t_{i-1}) \leq p_u(t_{i-1}) + r$. Let us take a node v such that $p_u(t_i) + r < p_v(t_i)$. By Definition 3.1, $p_u(t_{i-1}) - d_{mv} + r < p_v(t_{i-1}) + d_{mv}$, and thus, as we have supposed that $r \geq 2d_{mv}$ (Equation 3.1), we have

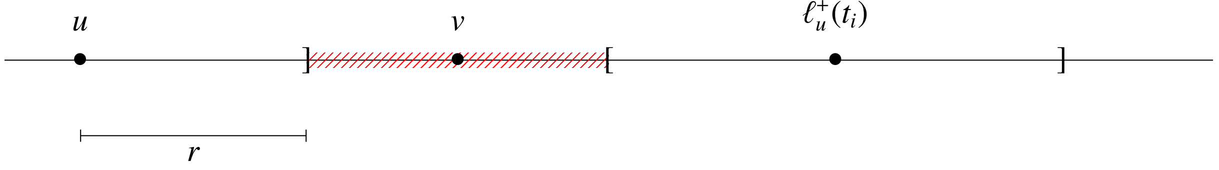


Figure 3.5 – Case 2: We may prove that v was a neighbor either of u or of $\ell_u^+(t_i)$ before the movement.

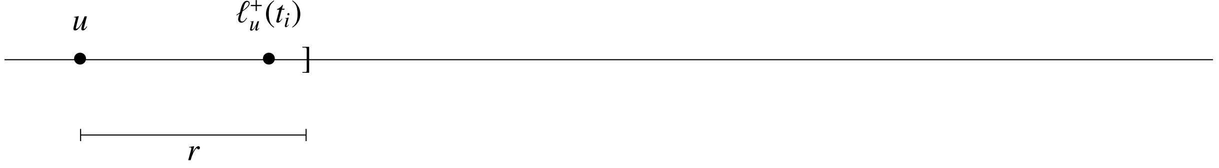


Figure 3.6 – Case 3: We may prove that a new lookout can be found for u in the local variables of $\ell_u^+(t_i)$.

$p_u(t_{i-1}) \le p_v(t_{i-1}) \le p_u(t_{i-1}) + r$, so that $v \in N_u(t_i)$. We thus have $C = \{v \in \mathcal{V} : p_u(t_i) + r < p_v(t_i)\}$.

Thus C is not empty, by supposition of Case 1, and $newLookout \neq \bot$, whereas $\ell_u^+(t_i) = \bot$. It follows that DistantLookout$(u, \ell_u^+(t_i)) \notin Fail$, so that neither line 28 nor line 40 are executed by u, while lines 42 through 47 will be executed. Thus, $\ell_u^+(\tilde{t})$ becomes the node w with the smallest coordinate so that $p_u(t_i) + r < p_w(t_i)$ (as it is the value taken by $newLookout$ at line 26). Thus, there cannot exist a node x so that $p_u(t_i) + r < p_x(t_i) < p_w(t_i) - r$, and thus $\ell_u^+(\tilde{t})$ is a valid lookout for u with regard to the positions of the nodes at time t_i.

- **Case 2 :** $\ell_u^+(t_i) \neq \bot$, and $\exists v : p_u(t_i) + r < p_v(t_i) < p_{\ell_u^+(t_i)}(t_i) - r$ (see Figure 3.5).

  Let $V = \{v \in \mathcal{V} : p_u(t_i) + r < p_v(t_i) < p_{\ell_u^+(t_i)}(t_i) - r\}$. Similarly to Case 1, as $G(t_i) \in \mathcal{G}(t_{i-1})$, and as all nodes did not move more than $d_{mv} \le r/2$ distance units in $]t_{i-1}; t_i]$, for any $v \in V$, we have either $v \in N_u(t_i)$ or $v \in L_u^+(t_i)$. Thus, LegitimateLookout$^+(u, v) \in Fail$, and with C the set computed by u at line 24, we have $C = V$.

  Thus, again, C is not empty. As $\exists v : p_u(t_i) + r < p_v(t_i) < p_{\ell_u^+(t_i)}(t_i) - r$, we necessarily have $d(u, \ell_u^+(t_i), t_i) > r$, so that (DistantLookout$, u, \ell_u^+(t_i)) \notin Fail$ (line 11), and neither line 28 nor line 40 are executed by u, while lines 42 through 47 will be executed, so that $\ell_u^+(\tilde{t})$ takes the value of the node in C that is closest to u. This ensures that there is no node w so that $p_u(t_i) + r < p_w(t_i) < p_{\ell_u^+(\tilde{t})}(t_i) - r$, and thus $\ell_u^+(\tilde{t})$ is valid with regard to the positions of the nodes at time t_i.

- **Case 3 :** $\ell_u^+(t_i) \neq \bot$, and $p_{\ell_u^+(t_i)}(t_i) \le p_u(t_i) + r$ (see Figure 3.6).

  Let us denote by x the lookout of u at time t_i, that is $x = \ell_u^+(t_i)$. We thus suppose that $p_x(t_i) \le p_u(t_i) + r$.

  In this case, we have $]p_u(t_i) + r \,;\, p_x(t_i) - r[= \emptyset$, so that no node v may violate a certificate LegitimateLookout$^+(u, v)$, thus C is empty. Furthermore, as $G(t_i) \in \mathcal{G}(t_{i-1})$, we have $p_u(t_{i-1}) + r < p_x(t_{i-1})$ (as x, the lookout u had at time t_i, is valid with regard to the positions of the nodes at time t_{i-1}). As movements are limited in speed, we thus have $d(u, x, t_i) \le r$, so that (DistantLookout$, u, x) \in Fail$. Thus u sends a message req(+) to x line 28.

The new lookout of u is the node chosen by x on lines 34 and 35, among the nodes of $N_x(t_i) \cup L_x^+(t_i)$ (that is, among the neighbors of u's lookout, and among the neighbors of the lookout of u's lookout) that are at least at distance r from u. Among those nodes, the closest one to u is chosen. To show that this node is a valid lookout for u, let us make a case analysis:

If $\ell_x^+(t_i) \neq \bot$, we can prove two facts:

- Let us show that $p_u(t_i) + r < p_{\ell_x^+(t_i)}(t_i)$. For this, let us first note that at time t_i, x is a valid lookout for u and $\ell_x^+(t_i)$ is a valid lookout for x, both with regard to the positions of the nodes at time t_{i-1}; we thus have $p_u(t_{i-1}) + r < p_x(t_{i-1})$ and $p_x(t_{i-1}) + r < p_{\ell_x^+(t_i)}(t_{i-1})$, so that, as $r \geq 2d_{mv}$, we get $p_u(t_{i-1}) + r + 2d_{mv} < p_{\ell_x^+(t_i)}(t_{i-1})$. As movements are limited in speed, $p_u(t_i) + r < p_{\ell_x^+(t_i)}(t_i)$.
- Let v be a node such that $p_u(t_i) + r < p_v(t_i) < p_{\ell_x^+(t_i)}(t_i) - r$. Let us show that $v \in N_x(t_i) \cup L_x^+(t_i)$. As $G(t_i) \in \mathcal{G}(t_{i-1})$, there does not exist a node w such that $p_u(t_{i-1}) + r < p_w(t_{i-1}) < p_x(t_{i-1}) - r$ (because x is a valid lookout for u) nor such that $p_x(t_{i-1}) + r < p_w(t_{i-1}) < p_{\ell_x^+(t_i)}(t_{i-1}) - r$ (because $\ell_x^+(t_i)$ is a valid lookout for x). Thus, as nodes do not move more than $r/2$ distance units per time step, we have $v \in N_u(t_i) \cup N_x(t_i) \cup N_{\ell_x^+(t_i)}(t_i)$. In the case where $v \in N_u(t_i)$ (and thus $p_v(t_{i-1}) \leq p_u(t_{i-1}) + r$), as $p_u(t_i) + r < p_v(t_i)$, and with $p_u(t_{i-1}) + r < p_x(t_{i-1})$ (x being a valid lookout for u) and $p_x(t_i) \leq p_u(t_i) + r$ (by supposition of case 3), then v and x have exchanged their positions: $p_v(t_{i-1}) \leq p_x(t_{i-1})$, but $p_x(t_i) \leq p_v(t_i)$. Again, as v and x did not move more than $r/2$ distance units, we necessarily had $d(x, v, t_{i-1}) \leq r$, and thus $v \in N_x(t_i)$, by validity of $N_x(t_i)$ with respect to the positions at time t_{i-1}. Thus, we have $v \in N_x(t_i) \cup L_x^+(t_i)$.

Thus, either $\ell_x^+(t_i)$ is a valid lookout for u with respect to the positions at time t_i, or there is a non-empty set V of nodes that are situated in-between u and $\ell_x^+(t_i)$ that make so that $\ell_x^+(t_i)$ is not a valid lookout for u. However, in this case, all nodes of V are in $N_x(t_i) \cup L_x^+(t_i)$, and thus x chooses one of them as new lookout for u on lines 34 and 35. As the chosen node is the node from V with the smallest coordinate, it is a valid lookout for u. Thus, we can conclude that in case 3, $\ell_u^+(\tilde{t})$ is also valid with regard to the positions of the nodes at time t_i.

Concerning the updates of S^+, note that any change of lookout of u comes with a message that is sent to the new lookout v (line 42), that will remove u from S_v^+ and a message to the previous lookout v' (line 47), that will add u to $S_{v'}^+$. Thus, we end up, for any node $u \in \mathcal{V}$, with $S_u^+(\tilde{t}) = \{v : u = \ell_u^+(\tilde{t})\}$.

Finally, we can see that the value of ℓ_u^+ cannot change after the execution of the procedure Correct_Legitimate_and_Distant_Lookout; thus all what has been proved for time $\tilde{t}$ remains valid until t_{i+1}, that is, $\ell_u^+(t_{i+1}) = \ell_u^+(\tilde{t})$, and $S_u^+(t_{i+1}) = S_u^+(\tilde{t})$. Thus, $\ell_u^+(t_{i+1})$ and $S_u^+(t_{i+1})$ are both valid with regard to the positions at time t_i. □

Lemma 3.9. *Let t_i be so that $G(t_i) \in \mathcal{G}(t_{i-1})$.*

After the execution of the updates associated with the failure of certificates CloseNeighbor *of instant t_i, that is, after the execution of procedure* Correct_Close_Neighbor*, line 20 of Algorithm 4), $\forall u \in \mathcal{V}$, we have $\forall v \in N_u$, $d(u, v, t_i) \leq r$, and $\forall v \in L_u^+$, $d(v, \ell_u^+, t_i) \leq r$.*

This remains true until t_{i+1}, that is, $\forall u \in \mathcal{V}$, $\forall v \in N_u(t_{i+1})$, $d(u, v, t_i) \leq r$, and $\forall v \in L_u^+(t_{i+1})$, $d(v, \ell_u^+(t_{i+1}), t_i) \leq r$.

Proof. Let us denote by $\tilde{t}$ the instant where the algorithm $\mathcal{A}_{f1d}$ finishes executing the procedure Correct_Close_Neighbor. Let u be a node from $\mathcal{V}$.

Line 5, for any node $v \in N_u(t_i)$ such that $d(u, v, t_i) > r$, (CloseNeighbor, u, v) is added to *Fail*. Thus, line 61 is executed by u, and v is removed from N_u. We thus have $\forall v \in N_u(\tilde{t})$, $d(u, v, t_i) \leq r$.

Any removal from N_u is complemented, on line 62, with a message sent to the nodes from S_u^+. By Lemma 3.8, we have that, at line 62, $S_u^+ = \{v : \ell_v^+ = u\}$, so that u is also removed from L_v^+ (when v executes line 67). Thus, we also have $\forall v \in L_u^+(\tilde{t})$, $d(v, \ell_u^+(\tilde{t}), t_i) \leq r$.

Finally, we can note that on line 8, only nodes v such that $d(u, v) \leq r$ are added to the list of failing PotentialNeighbor certificates. Thus, any node v added to N_u on line 72 satisfies $d(u, v) \leq r$, and similarly, any node added to L_u^+ satisfies $d(v, \ell_u^+) \leq r$. Hence, the execution of procedure Correct_Potential_Neighbor does not disrupt the property proven for moment $\tilde{t}$, and we have, $\forall u \in \mathcal{V}$, $\forall v \in N_u(t_{i+1})$, $d(u, v, t_i) \leq r$, and $\forall v \in L_u^+(t_{i+1})$, $d(v, \ell_u^+(t_{i+1}), t_i) \leq r$. □

Lemma 3.10. *Let t_i be so that $G(t_i) \in \mathcal{G}(t_{i-1})$.*

After the execution of the updates associated with the failure of certificates PotentialNeighbor *of instant t_i, that is, after the execution of procedure* Correct_Potential_Neighbor*, line 21 of Algorithm 4, $\forall u \in \mathcal{V}$, N_u and L_u^+ are valid with regard to the positions of the nodes at time t_i.*

This remains true until t_{i+1}, that is, $N_u(t_{i+1})$ and $L_u^+(t_{i+1})$ are valid with regard to the positions of the nodes at time t_i.

Proof. Let us denote by $\widetilde{t'}$ the instant where $\mathcal{A}_{f1d}$ *starts* executing the procedure Correct_Potential_Neighbor, and by $\tilde{t}$ the instant where $\mathcal{A}_{f1d}$ *finishes* executing the procedure Correct_Potential_Neighbor. Let u be a node from $\mathcal{V}$.

When the execution of Correct_Potential_Neighbor starts, we have $N_u\left(\widetilde{t'}\right) = \{v \in N_u(t_i) : d(u, v, t_i) \leq r\}$ (by Lemma 3.9, and because no node is added to N_u before the execution of Correct_Close_Neighbor, so that any node in $N_u\left(\widetilde{t'}\right)$ was in $N_u(t_i)$). Thus, to prove the validity of N_u at time $\tilde{t}$, i.e. $N_u(\tilde{t}) = \{v : d(u, v, t_i) \leq r\}$, we need to prove that any node v such that $d(u, v, t_i) \leq r$ and $v \notin N_u(t_i)$ is added to N_u before $\tilde{t}$.

Let us now show that for any node v such that $d(u, v, t_i) \leq r$ and $v \notin N_u(t_i)$, we have $v \in L_u^+(t_i) \cup L_u^-(t_i)$. In this case, we have that (PotentialNeighbor, u, v) is added to *Fail* line 8, and line 72 is executed so that v is added to N_u.

As $G(t_i) \in \mathcal{G}(t_{i-1})$, $v \notin N_u(t_i)$ if and only if $d(u, v, t_{i-1}) > r$. By Definition 3.1 and Equation 3.1, $\forall v \in \mathcal{V}, d(u, v, t_i) \leq r \implies d(u, v, t_{i-1}) \leq r + 2d_{mv} \leq 2r$. Thus, we have $d(u, v, t_i) \leq r \wedge v \notin N_u(t_i) \implies d(u, v, t_{i-1}) \in]r; 2r]$:

$$d(u, v, t_i) \leq r \wedge v \notin N_u(t_i) \implies \begin{cases} p_v(t_{i-1}) \in [p_u(t_{i-1}) - 2r \,;\, p_u(t_{i-1}) - r[\\ \text{or } p_v(t_{i-1}) \in]p_u(t_{i-1}) + r \,;\, p_u(t_{i-1}) + 2r] \end{cases} \tag{3.10}$$

Let us first look at the second case from Equation 3.10, and let us note by I^+ the associated interval, that is, $I^+ =]p_u(t_{i-1}) + r \,;\, p_u(t_{i-1}) + 2r]$. Again, let us denote by x the lookout of u at time t_i, that is $x = \ell_u^+(t_i)$. Because $G(t_i) \in \mathcal{G}(t_{i-1})$, we have that if $x = \perp$, it is impossible that there exists a v such that $p_v(t_{i-1}) \in]p_u(t_{i-1}) + r \,;\, p_u(t_{i-1}) + 2r]$. We can thus suppose that $x \neq \perp$.

For any node $v \in \mathcal{V}$ such that $p_v(t_i) \in I^+$, we have:

- either $v \in L_u^+(t_i)$, and in this case we have obviously $v \in L_u^+(t_i) \cup L_u^-(t_i)$;
- or $v \notin L_u^+(t_i)$. However, the existence of such a node v is in contradiction with $G(t_i) \in \mathcal{G}(t_{i-1})$, as, by validity of x, we cannot have $p_u(t_{i-1}) + r < p_v(t_{i-1}) + r < p_x(t_{i-1})$.

The first case from Equation 3.10 is similar, so that we get what we were looking for: for any node $v \in \mathcal{V}$ such that $d(u, v, t_i) \leq r$ and $v \notin N_u(t_i)$, we have $v \in L_u^+(t_i) \cup L_u^-(t_i)$. As mentioned previously, any such node will be added to N_u on line 72, so that $N_u(\tilde{t})$ is valid with regard to the position at t_i.

As with Lemma 3.9, each addition of a node into N_u is complemented with a message sent to the nodes in L^+ (which is valid by Lemma 3.8), so that we get the validity of $L_u^+(\tilde{t})$.

Finally, as Correct_Potential_Neighbor is the last procedure to be executed, no changes are made to the structure between $\tilde{t}$ and t_{i+1}, and thus we have $N_u(t_{i+1}) = \{v : d(u, v, t_i) \leq r\}$, et $L_u^+(t_{i+1}) = \{v : d(v, \ell_u^+(t_{i+1}), t_i) \leq r\}$. □

Lemma 3.11. *Let t_i be so that $G(t_i) \in \mathcal{G}(t_{i-1})$.*

We have that $\forall u \in \mathcal{V}$, $\ell_u^-(t_{i+1})$, $S_u^-(t_{i+1})$, and $L_u^-(t_{i+1})$ are all valid with regard to the positions of the nodes at time t_i.

Proof. The proof of this lemma is symmetric to the proofs of lemmas 3.8, 3.9 and 3.10. □

Proof of Theorem 3.7. For any $i \in \mathbb{N}$ such that $G(t_i) \in \mathcal{G}(t_{i-1})$, we have after execution of the operations described in Algorithms 4, 5, 6, and 7, that $G(t_{i+1}) \in \mathcal{G}(t_i)$, by Lemmas 3.8, 3.10 and 3.11. As we suppose that $G(0) \in \mathcal{G}(0)$, and because the movements of the nodes start at instant t_1, we have, by induction, that for any $i \geq 0$, $G(t_{i+1}) \in \mathcal{G}(t_i)$. □

Now that we have proven that $\mathcal{A}_{f1d}$ maintains a valid connection graph, we need to analyze the performances of the algorithm.

3.6 Performance Analysis

We will measure the performance of $\mathcal{A}_{f1d}$ by computing several upper bounds on memory usage and message complexity.

As in $\mathcal{A}_{f1d}$, nodes are often connected with other nodes that are situated in a ball of radius r, we will use the following value as a base value for our performance measurements:

$$b_{max} = \max_{u \in \mathcal{V}}(\text{card}(\{v : d(u, v) \leq r\})) \tag{3.11}$$

We make a supposition similar to the *displacement assumption* proposed in [50] : we suppose that the density of nodes that can rely close one to another is limited by a value ρ (recall that $d_{mv} = 1$ is the maximal number of distance units a node may move at each time step).

Definition 3.12. *At any instant t, any ball of radius d_{mv} contains at most ρ nodes from $\mathcal{V}$ (with $\rho \leq n$).*

Here, d_{mv} is used as a reference for scale in terms of distance units. It is reasonable to impose such a limit, as in most real settings, it should not be expected that nodes get arbitrarily “packed” in a small area: for example, nodes could have a non-negligible volume, so that two nodes cannot get too close to each other without inducing a collision. Note however that in the case where such a limit cannot be guaranteed, it is possible to use $\rho = n$.

The value of b_{max} is closely related to ρ: in 1D, we have $b_{max} = \rho \times \frac{r}{d_{mv}}$. We use both values, so that the origin of the costs is more apparent: ρ shows costs related to the movement of the nodes, while b_{max} shows costs related to the CloseNodes$_u(r)$ query.

Table 3.2 compares the performance of $\mathcal{A}_{f1d}$ with several algorithms. We measure the following values:

- The number of messages counts the maximal number of messages that can be sent at each time step by the nodes of $\mathcal{V}$. Here, only messages for the changes of connections and maintenance of the address books are sent: as explained in Section 3.1.1, it is supposed that no messages are needed for the nodes to know the positions of other nodes in their address book.

- The number of communication rounds per time step simply indicates, for the algorithms that use the synchronous model, how many communication rounds are needed to update the structure each time the nodes move.

- The memory usage is the maximal size of the local variables, that is, the sum of the sizes of the sets maintained by each node. Recall, as explained in Section 3.1.1, and as mentioned above, that the number of messages does not include the messages that may be required in practice to be exchanged for position estimation; however, it is supposed that those messages are proportional to the number of other nodes each node is connected to. It so happens that on each of the solutions of Table 3.2, the memory usage of a node is of the same order as the size of its address book (or equivalently, to the number of outgoing arcs of the node in the connection graph). Thus, the memory usage also gives a measurement for the number of messages sent for positional updates.

- The total number of connections counts, as its name suggests, the total size of all address books of the nodes, or equivalently, the number of arcs in the connection graph.

The proofs for the performance values for $\mathcal{A}_{f1d}$ are given below. The other algorithms are the following:

- An ideal algorithm that we denote by $\mathcal{A}_{id}$. In $\mathcal{A}_{id}$, each node u maintains only the target set of the algorithm, that is, N_u. Thus, the memory usage is limited to b_{max} memory entries per node, and the number of connections to $n \times b_{max}$. We suppose that this algorithm needs to send only one message to add or remove one node from N_u (which we believe is impossible in practice). Thus, as nodes do note move more than d_{mv} distance units per time step, no more than 4ρ messages can be needed per node, so that the number of messages per time step is limited to $n \times (4\rho)$ with $\mathcal{A}_{id}$.

- A first naive algorithm, that we denote by $\mathcal{A}_{na1}$. In this algorithm, the nodes are all connected to each others; in other words, the connection graph is the complete directed graph on the set of nodes. As each node has the exact knowledge of the positions of the set of nodes, N_u can be computed locally by u, so that no message needs to be sent to update the sets of neighbors. However, the memory usage is of n entries per node.

- A second naive algorithm, that we denote by $\mathcal{A}_{na2}$. In this algorithm, a node is designated as the *coordinator*, and all the other nodes are connected to this coordinator, similarly to a client/server model. The coordinator has a complete knowledge of the positions of the nodes, and is in charge of telling to any node $u \in \mathcal{V}$ when it should make a change to N_u. It thus follows that updates are handled in one communication round, and the same amount of messages is needed as for $\mathcal{A}_{id}$. The memory usage is of $n + b_{max}$ entries for the coordinator, and of b_{max} entries for any other node (the size of N_u). The total number of connections is as for $\mathcal{A}_{id}$, plus n arcs for the connections to the coordinator.

- D-Spanner, the distributed KDS presented in [56] is also given as comparison. Recall that Φ is the ratio between the highest and lowest possible distance between two nodes (Definition 1.6 p.36). Note however that D-Spanner is not tailored at the $\text{CLOSENODES}_u(r)$ query

for a fixed r, so that additional computations are needed to get the answer to the query. We will see in Section 4.2 that answering to this query needs $O(\log r)$ communication rounds and $O(\log r + k)$ messages.

Table 3.2 – Upper bounds on the performances of the algorithms

	$\mathcal{A}_{f1d}$	$\mathcal{A}_{id}$	$\mathcal{A}_{na1}$	$\mathcal{A}_{na2}$	D-Spanner [56]
Number of messages	$O(nb_{max}\rho) = O(nb_{max})$ (Lemma 3.13)	$n \times (4\rho)$	0	$n \times (4\rho)$	$O(n \log \Phi)$
Number of communication rounds per time step	$O(1)$ (Lemma 3.14)	1	0	1	
Memory usage / address book size	$O(b_{max})$ (Lemma 3.15)	b_{max}	n	n for the coordinator, b_{max} for the others	$O(\log \Phi)$
Total number of connections	$O(nb_{max})$ (Lemma 3.16)	$n \times b_{max}$	n^2	$n(b_{max} + 1)$	$O(n)$

Let us now prove the upper bounds for the performance of $\mathcal{A}_{f1d}$ that are given on Table 3.2.

Lemma 3.13. *The number of messages sent by a node per time step with $\mathcal{A}_{f1d}$ does not exceed $b_{max}(8\rho + 2) + 8$.*

Proof. Each change in the local variables of u is associated with the exchange of some messages:

- The removal of a neighbor from N_u can lead to up to $2b_{max}$ messages, line 62 and line 63 of Algorithm 6, one for each node in S_u^+ and in S_u^-.[8]
- The same goes for the addition of a new neighbor into N_u, which can lead to up to $2b_{max}$ messages.
- A change of lookout can lead to up to $4 + b_{max}$ messages : one REQ message, (line 28 of Algorithm 5), one REPLY message (line 36 of Algorithm 5), one REMOVE message (line 42 of Algorithm 5), and one ADD message (line 47 of Algorithm 5), plus up to b_{max} MYNEIGHBORS messages, one to each node in S_u^+ (line 54 of Algorithm 5).
- For the other local variables, that is, S_u^+, S_u^-, L_u^+, and L_u^-, changes are deduced only from messages received because of one of the above cases. Thus, the messages that are sent because of changes in these other local variables are already counted.

In addition, the number of changes to the local variables of u is limited because of the hypotheses we made:

[8]Note that these messages are not strictly necessary, as the nodes in S_u^+ (resp. S_u^-) could detect when a node from L_u^+ (resp. L_u^-) gets too far away from their lookout. This would however require to add one type of certificate to the list of certificates without drastically changing the performances.

- Because nodes move not more than d_{mv} distance units at each time step, and because this applies both to u and to any node from N_u, by definition of ρ, there cannot be more than 2ρ neighbors to remove from N_u at each time step.
- Similarly, there cannot be more than 2ρ new neighbors to add to N_u at each time step.
- The node u has only two lookouts, and each can change only once per time step.

Thus, the number of messages per time step induced by addition or deletion of nodes to/from N_u is smaller than $4\rho \times 2b_{max}$, while the number of messages induced by changes of lookout is smaller than $2(4 + b_{max})$. When summing all these upper bound, we get a total of $b_{max}(8\rho + 2) + 8$ messages per node at each time step with $\mathcal{A}_{f1d}$. □

The other performance measurements are quite simple to get:

Lemma 3.14. *$\mathcal{A}_{f1d}$ needs 7 communication rounds per time step to finish its updates*

Proof. This is trivial, as the last round, in Algorithm 7, is round number 7. □

Lemma 3.15. *The memory cost for a node is smaller than $5b_{max}$ when using $\mathcal{A}_{f1d}$.*

Proof. The size of the local memory of $u \in \mathcal{V}$ is equal to the sum of the sizes of its local variable, that is $\mathrm{card}(N_u) + \mathrm{card}(\{\ell_u^+\}) + \mathrm{card}(\{\ell_u^-\}) + \mathrm{card}(S_u^+) + \mathrm{card}(S_u^-) + \mathrm{card}(L_u^+) + \mathrm{card}(L_u^-)$.

By Definition 3.11, there cannot be more than b_{max} neighboring nodes in N_u. As the same goes for u's lookout, L_u^+ and L_u^- too cannot contain more than b_{max} elements. At worst, all the nodes from L_u^+ and L_u^- have u as lookout, so that they all are also in S_u^+ and S_u^-. The maximal memory size for u is thus less than $5b_{max}$. □

Lemma 3.16. *The total number of connections when using $\mathcal{A}_{f1d}$ is less than $3nb_{max}$.*

Proof. By Equation 3.4 and Equation 3.5, and by definition of the sets in Section 3.3.1, we have that, for any node u, $S_u^+ \subseteq L_u^-$ and $S_u^- \subseteq L_u^+$. It thus follows, that, as u is connected to a maximum of b_{max} neighbors in N_u, and as much neighbors of ℓ_u^+ and of ℓ_u^-, u doesn't have more than $3nb_{max}$ connections in total. □

3.7 Conclusion and Future Work

In this chapter, we have presented $\mathcal{A}_{f1d}$, a synchronous distributed algorithm that allows nodes to maintain, in the Black-Box model on a line, a structure that depends on the positions of the nodes. This structure enables each node u to have access, at all time, to the answer of the CloseNodes$_u(r)$ query, where r is a fixed parameter of the algorithm. Our algorithm $\mathcal{A}_{f1d}$ needs only a constant number of communication rounds to update the structure at each time step, and the worst case memory usage for one node is of the same order as the worst case number of nodes returned by the CloseNodes$_u(r)$ query, which is optimal.

3.7.1 Possible Extensions

Some elements of this chapter leave room for future research. In this section, we briefly discuss two possible extensions for $\mathcal{A}_{f1d}$, that would allow it to overcome some of the assumptions made in Section 3.1.1.

Algorithm 8 Adaptation of $\mathcal{A}_{f1d}$ so that at each time step, one node may exceed the speed limit d_{mv}.

1: Let u be the node that exceeds the speed limit on time step t_i.
2: Execute the updates as described with $\mathcal{A}_{f1d}$, but with u remaining at its previous position: for any node $v \neq u$, use $p_v(t_i)$, and for u, use $p_u(t_{i-1})$. We thus get a connection graph that is valid with regard to the new positions of all nodes except for u.
3: Keep in memory the previous local variables of u ($N_u(t_{i-1})$, $\ell_u^+(t_{i-1})$, etc.).
4: Insertion of u to its new position:
5: **if** $p_u(t_{i-1}) < p_u(t_i)$ **then**
6: $v \leftarrow \ell_u^+, v' \leftarrow u$ (*After loop, v is the first lookout that is on u's right, and v' a node that has v as lookout*)
7: **while** $p_v(t_i) < p_u(t_i)$ **do**
8: $v' \leftarrow v, v \leftarrow \ell_v^+$
9: By definition of the lookout, and because $v = \ell_{v'}^+$, the neighbors of u and its lookouts are a subset of $N_v \cup N_{v'} \cup \{\ell_v^+, \ell_{v'}^-\}$. The other local variables can then be deduced from this. Compute them.
10: **else**
11: [...] (*symmetrical if u moved to the left*)
12: Deletion of u from its previous position:
13: For any $v \in N_u(t_{i-1})$, remove u from $N_v(t_i)$ and from $L_w^+(t_i)$ and $L_w^-(t_i)$ for any w that has v as lookout.
14: For any node $v \in S_u^+(t_{i-1})$, find $\ell_v^+(t_i)$ in $N_u(t_{i-1}) \cup \{\ell_u^+(t_{i-1})\}$. Symmetrical for node in $S_u^-(t_{i-1})$.

Drop limitations on r and d_{mv}

First, it was supposed that there is a minimal value for r that depends on the movement speed, as stated in Equation 3.1, page 87.

It is possible to tweak $\mathcal{A}_{f1d}$ to still work with lower values for r. Having an algorithm to answer to CloseNodes$_u(r)$ queries, for values of r that are very small compared to the speed of the nodes, may not make much sense at first glance. It means that the answer to the query does not give much information, as the returned set could be very quickly completely out of date.

However, in some settings, the speed of the nodes may be low most of the time, but with the possibility for nodes to temporarily move very quickly. For example, in Heroes of Newerth [65], some objects allow players to teleport short distance of time, or to teleport back to their starting position. It would be interesting, in those settings, to avoid to limit r with a value depending on a very high but rarely happening movement speed.

In the case where only one node can exceed the speed limit at each time step, we may look for an approach similar to dynamic insertion/deletion. This approach is described in Algorithm 8, and consists in three phases. Let us denote by u the node that exceeds the speed limit. First, $\mathcal{A}_{f1d}$ is executed on the new positions of the nodes except for u. Then, in the *insertion* phase, the new local variables for u are looked for. Finally, in the *deletion* phase, u is removed properly from its previous location.

However, Algorithm 8 may lead to very poor performances in the worst case. If all nodes are at distance more than r from each other, the high speed movement of u may lead to the need for up to n communication rounds for the insertion lines 4 trough 11. The memory usage and number of connections is comparable to $\mathcal{A}_{f1d}$, however.

In the case where several nodes may exceed the speed limit at once, conflicts may arise

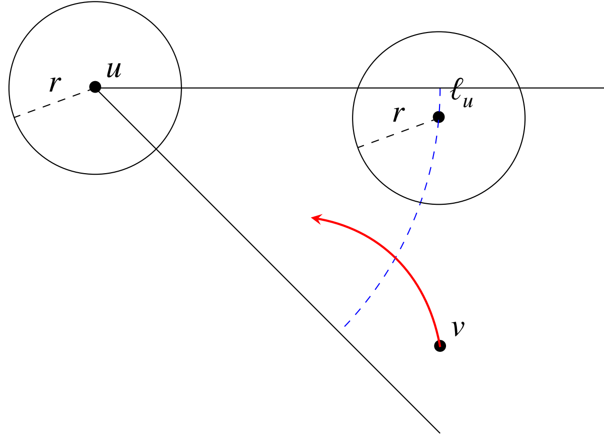


Figure 3.7 – Example of a situation where a node v gets closer to u without becoming neighbor of ℓ_u, the lookout of u in the same sector as v. The red arrow represents v's movement, and the dashed blue line represents the distance between u and ℓ_u.

if the insertion and deletion are not synchronous properly. Nodes would have to execute the associated instruction one after the other, deciding in which order to proceed.

Extending the Method in Higher Dimensions

The results presented in this chapter consider that the nodes have one-dimensional positions. As in most practical applications positions are two or three-dimensional, it is interesting to see if the results can be extended to higher dimensions.

First Approach: Generalize the Algorithm When generalizing $\mathcal{A}_{f1d}$, the set of neighbors remains similar: each node is connected to all nodes at distance r from it, which is no longer a line segment, but a hyperball (on 2D, the neighbors are in a disc like on Figure 3.7). Concerning the lookouts, we may notice that the one-dimensional structure maintained by $\mathcal{A}_{f1d}$ is similar to a Yao graph (see Section 1.5.2). The most immediate generalization would be to divide the space around each node u in k sectors, as for Yao graphs, with a lookout in each sector that is in charge of informing u of any other node that gets close enough to u to become a neighbor.

However, in higher dimensions, some properties change. Firstly, when positions are in one dimension, it is guaranteed that a node v first has to become a neighbor of u's lookout before getting close enough to u so that v should become u's new lookout. With our straightforward generalization of $\mathcal{A}_{f1d}$, where there is only one lookout in each sector, this seems not to be true in two or higher dimensions, as shown on Figure 3.7, where the node v can get close enough to u to become its new lookout, or even its neighbor, without first becoming a neighbor of ℓ_u, the lookout of u that is in the same sector as v. It thus seems difficult to maintain the sets of neighbors using only one lookouts in each sector.

Even maintaining the lookouts themselves seems difficult. As each node would need one lookout per sector, a difficulty arises when ℓ_u, the lookout of u, goes into another sector of u. As can be seen on Figure 3.8, if, among the neighbors and lookouts of ℓ_u none can become the new lookout of u, then the local knowledge of the two nodes is not enough to repair the structure. It thus seems required, in the case of Figure 3.8, for the nodes to explore the connection graph until a new lookout is found. However, there does not seem to be straightforward methods to accomplish this exploration in less than $O(n)$ communication rounds, as the new lookout in the sector left by ℓ_u could be extremely far away.

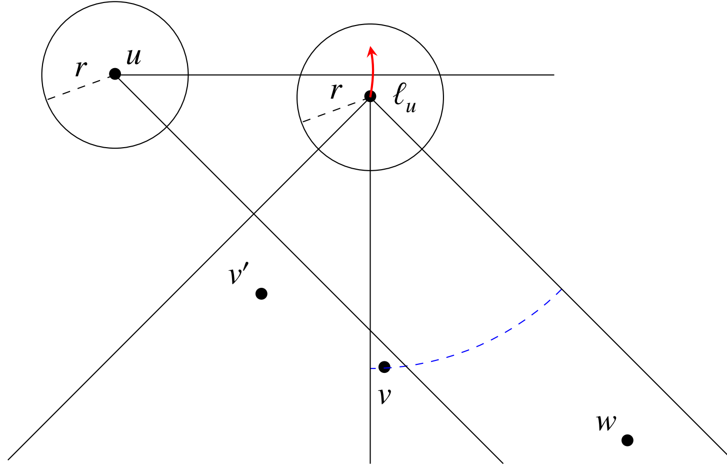


Figure 3.8 – Example of a situation where ℓ_u goes into another sector of u, and where w becomes the only node that can replace ℓ_u as lookout in the sector that it left. Here, neither u nor ℓ_u have w as lookout, as v and v' are ℓ_u's lookouts (as v is closer to ℓ_u than w).

It thus seems that more advanced techniques should be used in higher dimension. Other solutions of the literature that propose kinetic Yao graphs [119, 120] could be considered to help maintaining the lookouts, but as those structures do not involve sets of neighbors, this problem is a perspective for future research.

Second Approach: Run Several Instances of $\mathcal{A}_{f1d}$ Another, simpler approach, consists in running several instances of $\mathcal{A}_{f1d}$, one for each dimension. The set of neighbors of a node u in one of these instances is thus the set of nodes of $\mathcal{V}$ whose projection on the associated dimension is at distance r from u's own projection. In order to answer to the $\textsc{CloseNodes}_u(r)$ query, u would thus have to compute the set of nodes that are actually at distance r from it, out of the union of all its neighbors.

The memory usage is quite high, however. If we call d_{max} the maximal distance there can be between two nodes, it is easy to see that the memory usage of one node is bounded by $d \times 2r \times d_{max}^{d-1} \times \rho'$, with ρ' the maximum number of nodes that can have their position in a unit of volume. See Figure 3.9 for an example of one projection in 2D. As a comparison, the size of the answer to the $\textsc{CloseNodes}_u(r)$ query is bounded by $\pi r^2 \rho'$ in 2D, and by $\frac{4}{3}\pi r^3 \rho'$ in 3D, which is lower, as r should be much smaller than d_{max} in most practical settings.

This lack of straightforward solution for the problem of answering to the $\textsc{CloseNodes}_u(r)$ query in higher dimensions, brings us to study other types of structures, and in particular *navigating nets*. These tree-based structures can be used to answer to $\textsc{CloseNodes}_u(r)$ queries with non-fixed parameters r. This is the main subject of the next chapter.

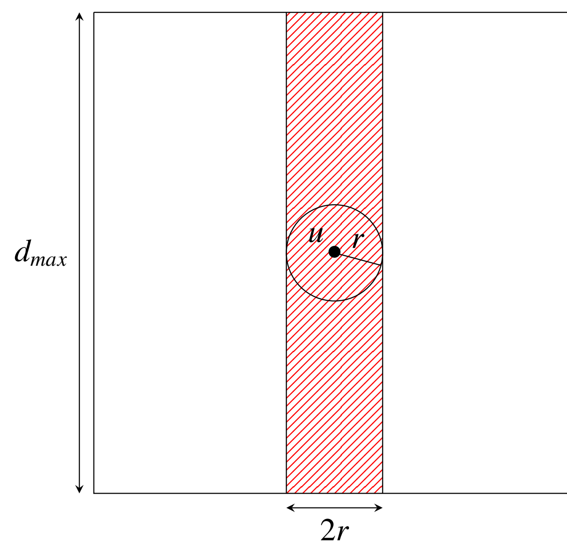


Figure 3.9 – Instance of $\mathcal{A}_{f1d}$ running on one of the projections in 1D. The red hashed zone represents all nodes that are u's neighbors in that instance. The big square represents the whole space.

Chapter 4

Kinetic Data Structures for Proximity Queries in Higher Dimension: Navigating Nets

As in the previous chapter, in this chapter, we aim at providing algorithms and data structures dedicated to moving objects to answer to the $\textsc{CloseNodes}_u(r)$ query: *at time t, given a node u and a distance r, return the set of nodes that are at most at r distance units away from u at the current time t*.

The main difference with previous chapter, is that here r is no longer a predefined constant, but a parameter of the query that may change when a node asks for the query. It is thus no longer possible to maintain at each time the answer to the query as in the previous chapter. We thus focus on a another type of data structure that is naturally adapted to this query: navigating nets.

First, in Section 4.1, we will describe the model that will be used in this chapter, and present some results that could achieve the same goal using other structures than navigating nets. In Section 4.2, we define navigating nets and extensions of navigating nets so as to encompass most uses of similar structures from previous works. In Section 4.3, we present, in the centralized setting, *constrained navigating nets*, that by adding additional constraints to the classical navigating nets, add some useful properties to the structure (described in Section 4.3.2); we then give an algorithm, $\mathcal{A}_{cnn}$, prove that it is able to maintain constrained navigating nets in the low mobility setting, and give some performance measures. Finally, in Section 4.5, we devise future improvements for the results of this chapter, in particular considering the high mobility setting, and the distributed setting.

4.1 Introduction

4.1.1 Model

The suppositions that are made in this chapter are the same as those from the previous chapter. Again, $\mathcal{V}$ is a set of n nodes, and each node $u \in \mathcal{V}$ is associated at each instant $t \in \mathbb{R}^+$ with a position $p_u(t)$. We suppose that the positions are in a metric space with a constant doubling dimension (we may thus have $p_u(t) \in \mathbb{R}^d$, see Section 1.5.3, or more exactly, $p_u(t) \in \{0; a; 2a; \cdots; Ua\}^d$ to take into account the limited size of the memory registers, recall Section 1.1.1). We denote by $d(u, v, t)$ the distance between $p_u(t)$ and $p_v(t)$.

Recall also the following definitions and properties about the nodes' positions:

Definition 1.7 (aspect ratio Φ)**.** *d_{min} and d_{max} are two given values such that $\forall t, \forall u, v \in \mathcal{V}, d_{min} \leq d(u, v, t) \leq d_{max}$.*

The aspect ratio is given by $\Phi = \frac{d_{max}}{d_{min}}$.

The value of d_{max} typically represents the size of the space in which the nodes may move, while d_{min} may represent the size of the nodes or the distance at which they collide.

We use in this chapter the Black-Box model; the movements are not known in advance, and nodes may move only at specific, synchronous instants that we call *time steps*. We will denote by t_i the instant of the i-th time step[1], that is, the moment of the i-th time the nodes may move.

There is a maximum normalized displacement per time unit such that for each point $u \in \mathcal{V}$ and any time step t_i we have $d(p_u(t_i), p_u(t_{i+1})) \leq d_{mv}$, where $d(p_u(t_i), p_u(t_{i+1}))$ denotes the distance between $p_u(t_i)$ and $p_u(t_{i+1})$. For simplicity, we may suppose that d_{mv} is normalized to 1.

The results of this chapter are valid under a relaxed constraint on the minimal distance, already used in the previous chapter (see Definition 3.12, p.101). Instead of strictly limiting that minimal distance, we define a constant ρ, such that a node may have up to ρ nodes at distance d_{mv} from it. Additionally, we suppose that $d_{min} < d_{mv}$.

Definition 4.1. *At any instant t, any ball of radius d_{mv} contains at most ρ nodes from $\mathcal{V}$ (with $\rho \leq n$).*

It is thus possible for two nodes to share the same coordinates. The value of ρ is a measure of density. It can correspond either to a constraint or to an observation. For example, if the nodes represent real objects with a physical volume, then the nodes cannot get too close to each other, naturally limiting the amount of nodes per unit ball. This requirement is similar to the *displacement assumption* proposed in [50].

The query is done on a directed graph $G(t) = (\mathcal{V}, E(t))$, the so-called *connection graph*. The challenging task is to define a connection graph allowing fast answers to the CloseNodes$_u(r)$ query and fast updates when the nodes move at each time step of the Black-Box model. The query is defined as such:

Definition 4.2 (CloseNodes$_u(r)$)**.** *Given a node $u \in \mathcal{V}$ at current time t, and a distance $r \in \mathbb{R}$, return all nodes $v \in \mathcal{V}$ such that $d(u, v, t) \leq r$.*

For convenience, we add the following definition:

Definition 4.3. *We denote by $B_u(r)$ the ball centered in u and of radius r, and by $b_u(r) = card(B_u(r))$ the number of nodes in that ball.*

In other words, CloseNodes$_u(r)$ should return the set $\{v \in \mathcal{V} : p_v \in B_u(r)\}$.

We first focus on the centralized setting, where G is a data structure on a single computer, and where we aim at bounding the amount of memory it uses, as well as the update time to handle the movements of the nodes. We then adapt our results to the distributed setting, where G is, as in the previous chapter, a distributed data structure, where we aim at bounding the local memory usage of each node, and the amount of communication rounds needed to update G when the nodes move.

As seen in Section 1.6.1, we also distinguish two mobility settings.

- In the *low mobility* setting, a single node is allowed to change its position at each time step.
- In the *high mobility* setting, all the nodes can change their position at each time step.

[1] Again, t_i is just used as a convenient way to specify a time step, and to be able to refer to the previous and next time step (with t_{i-1} and t_{i+1}).

4.1.2 Contribution

The contributions of this chapter are twofold.

First, we give a general definition of navigating nets, so as to encompass previous results. We give properties of navigating nets, some of which already discussed in other articles [48, 55], but here we give broader conditions on the parameters of the navigating nets for these properties to be true. In particular, we give in Theorem 4.20 broader conditions than in [55] for navigating nets to be spanners. Navigating nets can thus be used to answer to the $\textsc{CloseNodes}_u(r)$ query in $O(k \cdot \log \Phi)$ time (with $k = \text{card}(B_u(r))$ the size of the set returned by the query), as we have seen in Section 1.5.2 that a spanner may be used to answer to the $\textsc{CloseNodes}_u(r)$ query in $O(k \cdot \delta)$, with δ the maximal degree of the spanner, which is $O(\log \Phi)$ for navigating nets. We also present another search algorithm on navigating nets to answer to the query in $O(\log r + k)$ time (Lemma 4.15).

We then introduce *constrained navigating nets*, a subclass of navigating nets that shows interesting properties. These properties are used to give a centralized algorithm in the low mobility setting, $\mathcal{A}_{cnnptr}$, that updates constrained navigating nets in $O(\log \Phi)$ computations per time step (Theorem 4.52), using $O(n)$ space (Theorem 4.53). This is similar to the DefSpanners from [55], as we prove that they can be adapted to the low mobility setting with $O(\log \Phi)$ computations per time step (Theorem 4.17). With our data structure however, while the cost to compute the changes to make to the data structure at each time step is similar to DefSpanners, the number of actual changes to the data structure is kept small (Theorem 4.37). The centralized results are summarized in Table 4.1.

Table 4.1 – Results for kinetic centralized navigating nets: computations per time step in the Black-Box model.

	Low mobility	High mobility
DefSpanner	$O(\log \Phi)$ (Theorem 4.17)	$O(n \log \Phi)$ [55]
Constrained Navigating Nets ($\mathcal{A}_{cnnptr}$)	$O(\log \Phi)$ (Theorem 4.52)	**Conjecture**: $O(n)$ (see Section 4.5.1)

We also give a synchronous algorithm, $\mathcal{A}_{cnndist}$ to maintain distributed constrained navigating nets using a constant number of communication rounds per time step in the low mobility setting (Theorem 4.62). The memory usage for one node may be $O(n)$ at worst (Theorem 4.64), but a node needs to track the position of only $O(\log \Phi)$ other nodes (Theorem 4.65), and the total memory cost for all the nodes is $O(n)$ (Theorem 4.63). We thus give the first distributed Kinetic Data Structure in the Black-Box model, as it is unclear whether [56], written for the Flight Plan model (a movement model that differs from the Black-Box model, see page 38) can be adapted to the Black-Box model.

Our algorithms are all based on certificates, and as long as all of these certificates are valid, no changes to the structure have to be made.

This chapter is organized as follows.

In Section 4.2, we give the definition of navigating nets, along with some of their properties. Then, in Section 4.3, we present the main results of this chapter. First, in Section 4.3.1, we show that DefSpanners can be easily adapted to the low mobility setting, so as to be updated using only $O(\log \Phi)$ computations per time step. We then give the definition of constrained navigating nets in Section 4.3.2, and show some properties that are the main appeal of these structures. Then, in Section 4.3.3 we explain how to maintain constrained navigating nets under movements of the nodes in the low mobility setting and Black-Box model. We present a first algorithm $\mathcal{A}_{cnn}$ that uses $O(n)$ space, but needs $O(\log^2 \Phi)$ computations per time step, which is

worse than DefSpanners: Section 4.3.4 proves the validity of $\mathcal{A}_{cnn}$, and Section 4.3.5 proves its performance. We then present in Section 4.3.6 some modification to $\mathcal{A}_{cnn}$ to get $\mathcal{A}_{cnnptr}$, the algorithm that needs only $O(\log \Phi)$ computations per time step to maintain constrained navigating nets.

We then move to Section 4.4.3, where we look at the distributed setting, and show that D-Spanners from [56] are ill-adapted to the Black-Box model, and present $\mathcal{A}_{cnndist}$, an algorithm that maintains constrained navigating nets using a constant number of communication rounds per time step. Finally, we conclude in Section 4.5, and give some ideas on how to extend $\mathcal{A}_{cnn}$ to the high mobility setting.

4.1.3 Other Related Structures

We have seen in Section 1.5.2 several data structures that can be used to answer to the CloseNodes query in the centralized setting, and we have seen throughout Section 1.7 a few results about maintaining these structures in the Black-Box model:

- Delaunay triangulations can be used in two dimensions to answer to the CloseNodes query in $O(k \cdot n)$ time (with k the number of nodes returned by the query). Maintaining Delaunay triangulations can be done in the high mobility setting either with $O(\Phi^2)$ or with $O(n \log n)$ computations per time step [19], and with $O(n)$ computations per time step in the low mobility setting.

- Navigating net can be used to answer to the CloseNodes query in $O(k \cdot \log \Phi)$ time. Updating navigating nets takes $O(n \log \Phi)$ computations per time step in the high mobility setting [55], and we will show in Section 4.3.1, once having properly introduced navigating nets, that it takes $O(\log \Phi)$ computations per time step in the low mobility setting.

- Quadtrees can be maintained with $O(n \log \rho)$ computations per time step in the high mobility setting [50] (and as explained in Section 1.7.3, p.46, this cost is not improved in the low mobility setting). We will show in this section how to use quadtrees to answer to CloseNodes queries in two dimensions.

In the distributed setting, to the best of our knowledge, only navigating nets have been studied, in [56]. We will give more details in Section 4.4.1.

Recall that a two-dimensional quadtree is a tree of square-shaped cells. Each cell has four *children*, that is, equal sized, square-shaped cells, that form a partition of the *parent* cell. Some cells however, the *leaves*, do not have children, but maintain links to the nodes whose positions are situated in them. The root of the tree must be a cell that is a superset of the possible positions of the nodes. Each cell is subdivided until the leaves contain at most one node. Note that the borders of the cells should be defined in such a way that a node is contained in only one leaf.

Algorithm 9 describes how to answer to the CloseNodes$_u(r)$ query using a quadtree: we simply descend the hierarchy, keeping only the cells of the quadtree that intersect with the query range $B_u(r)$, until all nodes are found in the leaves.

Let us prove the following (recall that $b_u(x)$ is the number of nodes in the ball of radius x centered in node u):

Theorem 4.4. *In the centralized setting, quadtrees can be used to answer to two-dimensional* CloseNodes$_u(r)$ *queries in* $O\left(b_u\left(r + r\sqrt{2}\right) \log \Phi\right)$ *time.*

As we suppose, by Definition 4.1, that the density of the nodes is limited, we have that $O\left(b_u\left(r + r\sqrt{2}\right)\right) = O(b_u(r))$, and thus, with $k = b_u(r)$ the number of nodes returned by the query, we can simplify the cost from Theorem 4.4, to $O(k \cdot \log \Phi)$.

Algorithm 9 Answer to $\textsc{CloseNodes}_u(r)$ query using a quadtree

1: **function** $\textsc{CloseNodes}(u,r)$
2: $S \leftarrow$ {the root of the quadtree} ; $R \leftarrow \emptyset$
3: **while** $S \neq \emptyset$ **do**
4: Let $s \in S$
5: **if** s is a leaf **then**
6: Add to R every node v such that $p_v \in s \cap B_u(r)$
7: **else**
8: Add to S every children s' of s such that $s' \cap B_u(r) \neq \emptyset$.
9: Remove s from S
10: **return** R

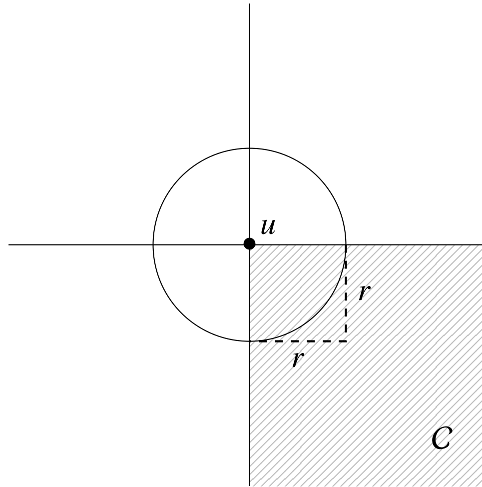


Figure 4.1 – The four sectors used in the proof of Lemma 4.5

To prove Theorem 4.4, we will first define a *big cell* as a leaf of the quadtree whose sides are longer than r, and prove the following lemma:

Lemma 4.5. *In a two-dimensional quadtree, for any $u \in \mathcal{V}$, there is a constant number of big cells which have a non-empty intersection with $B_u(r)$.*

Proof. Let us divide the space around u in four sectors of angle $\frac{\pi}{2}$, as shown on Figure 4.1. Let us call C one of these sectors (hashed in gray on the figure), and let us consider the square in C with sides of length r that has u as a vertex, and in particular, let us consider the two sides of that square opposite of u (those sides are represented with thick dashed lines on Figure 4.1).

By construction, any big cell of the quadtree that is a subset of C, and that has a non-empty intersection with $B_u(r)$ should intersect at least with one of those two sides. However, each side can intersect with at most two big cells, and thus C can intersect with at most four big cells that have a non-empty intersection with $B_u(r)$.

Similarly, the number of big cells that intersect both with $B_u(r)$ and the borders of the sectors is constant. Thus, the total number of big cells that intersect with $B_u(r)$ regardless of sectors is constant. □

Proof of Theorem 4.4. On line 6, Algorithm 9 needs to check if the nodes from the leaves are in $B_u(r)$. Let us denote by L the set of the leaf cells from the quadtree that intersect with $B_u(r)$. All nodes that are checked by Algorithm 9 belong to a cell from L, as each cell of S is either the root or has been added to S on line 8.

The leaves that are not big cells, and thus have a side of length smaller than r are all subsets of $B_u\left(r + r\sqrt{2}\right)$. Thus, by Lemma 4.5, the total number of nodes checked by Algorithm 9 is $O\left(b_u\left(r + r\sqrt{2}\right)\right)$.

As the depth of a quadtree is $O(\log\Phi)$, and as the algorithm starts at the root and descends the hierarchy, we get a total computation time of $O\left(b_u\left(r + r\sqrt{2}\right)\cdot\log\Phi\right)$. □

Table 4.2 – Results for kinetic centralized navigating nets: computations per time step in the Black-Box model.

	Dimension	Query cost	Low mobility	High mobility
Delaunay triangulations	2D	$O(k\cdot n)$ [19]	$O(n)$ [19]	$O(\Phi^2)$ or $O(n\log n)$ [19]
Quadtrees	2D	$O(k\cdot\log\Phi)$ (Theorem 4.4)	$O(n\log\rho)$ [50]	$O(n\log\rho)$ [50]
Navigating nets	any metric space of doubling dimension	$O(k\cdot\log\Phi)$ (Lemma 4.15)	$O(\log\Phi)$ (Theorem 4.17 and Theorem 4.52)	$O(n\log\Phi)$ [55]

The performance of the structures seen in this section are summarized in Table 4.2. All structures have a total memory cost of $O(n)$, which is not represented on the table. We can see that while Quadtrees have the best update time in two dimensions, in the high mobility setting, when ρ is small with regards to Φ, navigating nets are the best for the low mobility setting, while also not being limited to two dimensional positions.

Let us now look at navigating nets, and prove these performances.

4.2 Navigating Nets

Navigating nets have already been introduced in Section 1.5.2. These hierarchical structures are interesting for our problem, as we will see that they can be used to answer efficiently to the CloseNodes$_u(r)$ query. Also, as the definition of the structure depends only on the distance between the nodes, a navigating net can be defined on any metric space, and in particular, regardless of the dimension.

We have seen in Section 1.5.2 that, as with ε-nets (the structures which they are based on), navigating nets have been used on several occasions, but not necessarily under the same name, and sometimes with different parameters. There doesn't seem to be a consensus yet regarding which parts of the definitions of navigating nets should be considered possible to parameterize. In this section, we try to address this problem by giving a definition that encompasses as much as possible other similar structures, that is, with as much parameters as possible.

In Section 1.5.2, navigating nets have been simply described as hierarchies of b^i-nets. Let us redefine navigating nets step by step. In Section 4.2.1, we give a definition of the basic structure of navigating nets. Then, in Section 4.2.2, we will enrich this structure with arcs representing neighbors as in [55]. We will then see in Section 4.2.3 how to use these elements to answer to the CloseNodes$_u(r)$ query.

4.2.1 A Tree of Nodes

The basic structure of a navigating net is that of a tree constructed on the set of nodes. Each node u is the center of several balls of radius b^k, for k ranging from 0 to some integer denoted

L_u, and with b a (fixed) parameter of the navigating net.

The value of k is the *level* of the ball. Recall, as stated in Definition 4.3, that $B_u\left(b^k\right)$ denotes the ball centered on u and of radius b^k. For each level $k \geq 1$, two balls $B_u\left(b^k\right)$ and $B_v\left(b^k\right)$ are allowed to co-exist only if their center-to-center distance is large enough, that is, if $d(u,v) \geq \gamma \cdot b^k$, with γ a fixed parameter of the navigating net[2]. It thus follows :

Lemma 4.6. *For all* $u, v \in \mathcal{V}, \min(L_u, L_v) \leq \left\lfloor \log_b \frac{d(u,v)}{\gamma} \right\rfloor$

Finally, each node u (except one, the root) has a parent $f_u \in \mathcal{V}$. A node v is a valid parent for node u if and only if $L_u < L_v$ and $d(u,v) < \alpha b^{L_u+1}$, with α another fixed parameter of the navigating net.

Note that we need to have $\gamma < \alpha$ (or else it is impossible to construct a navigating net). To summarize, we have three properties as stated in the following definition.

Definition 4.7. *Each node u is assigned an integer* L_u *and a node* f_u *such that:*

- *γ-separation :* $\forall u, v \in \mathcal{V}, \forall k \in \mathbb{N}^* : k \leq \min(L_u, L_v) \implies d(u,v) \geq \gamma \cdot b^k$
- *α-coverage :* $\forall u \in \mathcal{V} : f_u = v \neq u \implies L_u < L_v \wedge d(u,v) < \alpha b^{L_u+1}$
- *unique root :* $\exists! v \in \mathcal{V} : f_v = v$

Together, the properties of unique root and of α-coverage imply that at the highest level of the hierarchy, only one ball may exist, and that it is centered on the root.

On Figure 4.2, an example of three levels of a navigating net is given. We have $L_w = k+2$, $L_v = k+1$, and $L_u = L_x = k$. As both w and v are in level $k+1$, the γ-separation requires that $d(w,v) \geq \gamma \cdot b^{k+1}$, and as all the nodes are in level k, all distance between any two points of $\{u, v, w, x\}$ must be greater than or equal to $\gamma \cdot b^k$. We also have $f_u = v$, $f_v = w$ and $f_x = v$ (w is either the root or its parent is not represented). Thus, α-coverage requires that $d(u,v) < \alpha \cdot b^{k+1}$, $d(v,w) < \alpha \cdot b^{k+2}$, and $d(x,v) < \alpha \cdot b^{k+1}$.

Given a set of nodes, a valid assignment of parents can be obtained with Algorithm 10. With this initialization, all nodes start at level 0, and are kept in the levels above as long as the nodes are not too close to each other.

Algorithm 10 Level and parent initialization

1: **for all** $u \in \mathcal{V}$ **do** $f_u \leftarrow u$, $L_u \leftarrow 0$
2: **end for**
3: $S \leftarrow \mathcal{V}$
4: **for** $k = 1, \ldots, 1 + \left\lfloor \log_b \left(\frac{d_{max}}{\gamma} \right) \right\rfloor$ **do**
5:     **for all** $u \in S$ **do**
6:         **for all** $v \in S : v \neq u, d(u,v) < \gamma b^k$ **do**
7:             $S \leftarrow S \setminus \{v\}; f_v \leftarrow u$
8:         **end for**
9:         $L_u \leftarrow k$
10:     **end for**
11: **end for**

[2]While more general than most definitions of navigating nets, this definition does not encompass the so-called *slack net trees* from [42], where the property of γ-separation is replaced by the following property : $\forall u \in \mathcal{V}, \forall k \leq L_u, \mathrm{card}(\{v \in \mathcal{V} : L_v \geq k \wedge d(u,v) \leq b\}) \leq \gamma$ (note that here, γ is not a distance multiplier, but a maximal number of nodes).

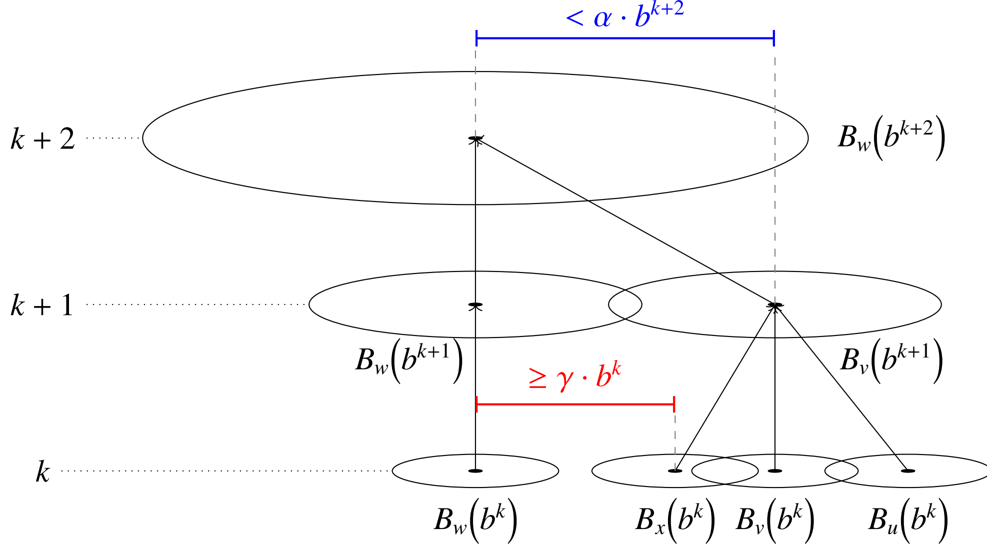


Figure 4.2 – An example of three levels of a navigating net: the balls are drawn as if they were 2D circles seen at an angle, and at each level, only balls at that level are drawn.

This gives a valid assignment of levels and parents. We can see at line 7 that nodes are given parents at distance smaller than γb^k away; as $\gamma < \alpha$, we have that the α-coverage is respected. Concerning γ-separation, note that for each node u that is kept at level k (at line 9), all nodes at distance smaller than γb^k from u remain at level $k-1$ (as they are removed from the set S on line 7), guaranteeing that at each level, the inter-node distance is at least γb^k distance units; it is thus assured that the γ-separation is respected. Finally, the unicity of the root is guaranteed by the fact that the loop stops at level $1 + \left\lfloor \log_b \left(\frac{d_{max}}{\gamma} \right) \right\rfloor$, which is the first level such that the inter-node distance must be strictly greater than d_{max}. As we suppose in our model that d_{max} is the maximal distance, the level in question cannot contain more than one node (and it must contain one, as S is never completely empty).

There is another slightly different way of initializing the levels and parents. In Algorithm 11, we remove nodes from S as soon as a valid parent is found, changing lines 4 and 6.

Algorithm 11 Level and parent initialization, prioritizing low levels

```
1:  for all u ∈ 𝒱 do f_u ← u, L_u ← 0
2:  end for
3:  S ← 𝒱
4:  for k = 1, ..., 1 + ⌊log_b(d_max/α)⌋ do
5:      for all u ∈ S do
6:          for all v ∈ S : v ≠ u, d(u,v) < αb^k do
7:              S ← S \ {v}; f_v ← u
8:          end for
9:          L_u ← k
10:     end for
11: end for
```

This slightly prioritizes lower levels for the nodes, but does not have an impact asymptotically, as the maximum level for a node is still $O(\log \Phi)$. Additionally, we will see that the initialization of Algorithm 10 will be required in Section 4.3 (as it ensures a stronger property than α-coverage). Thus, we will suppose thereafter that our navigating nets are initialized using Algorithm 10.

4.2.2 Enriching the Structure by Adding Neighbors

For better search algorithms, and to enable maintenance of the navigating net under motion of the nodes, additional connections are needed in the navigating net. For each of its levels, a node will be connected to other close-by nodes that we will call its *neighbors*.

While the results on navigating nets closely related to our work mostly use neighbors in their navigating nets, some articles use navigating nets without neighbors, that is, the structure is as defined in Section 4.2.1 [42].

To define these neighbors, we use another constant value c, the neighboring coefficient. For each level k, balls $B_u\left(b^k\right)$ and $B_v\left(b^k\right)$ are neighbors if their center-to-center distance is bounded by $c \cdot b^k$, that is, if $d(u,v) < c \cdot b^k$. We will say that u and v are neighbors at level k if $k \leq \min(L_u, L_v)$, and if $B_u\left(b^k\right)$ and $B_v\left(b^k\right)$ are neighbors.

For node u, the set N_u is a compact representation of the balls that are neighbors of a ball centered in u. Each element of N_u has the form (v,k) where $v \in \mathcal{V}$ and $k \in \mathbb{N}$ is the level at which balls centered in u and v are neighbors. More precisely:

Definition 4.8 (Set of neighbors)**.** $N_u = \{(v,k) \in \mathcal{V} \times \mathbb{N} : 1 \leq k \leq \min(L_u, L_v) \wedge d(u,v) < c \cdot b^k\}$.

Each node is neighbor of itself; in other words, for each level $k \leq L_u$, $(u,k) \in N_u$. Note also that by γ-separation (Definition 4.7), there is a minimum distance between every node at each level; thus, by doubling dimension, the number of neighbors of a node at a given level is always constant.

We may now properly define navigating nets, simply as the graph where the edges represent the parent/child and neighboring relations of the nodes.

Definition 4.9. *Let each node u of $\mathcal{V}$ be associated with an integer L_u, a node f_u, and a set N_u, all satisfying Definition 4.7 and Definition 4.8.*

A navigating net of parameters b, α, γ and c, on $\mathcal{V}$ is the directed graph $(\mathcal{V}, E)$ with the set of edges $E = \{(u,v) : u \in \mathcal{V} \wedge (v = f_u \vee (v,*) \in N_u)\}$ (where $*$ may be any value).

While navigating nets are here defined as directed graphs, we can see that the definition of neighbors is symmetric: if $(u,k) \in N_v$, then we must have $(v,k) \in N_u$. We will later introduce the sets of children, which are the symmetric of parents, and that will be added to our data structure. In practice, the graphs we maintain will thus always have all edges in both sides (that is, if $(u,v) \in E$, then $(v,u) \in E$), so that our structures could be considered as undirected. However, we prefer to explicitly maintain the edges in both ways, so that our definition of G remains consistent in the centralized and distributed setting, as well as with Chapter 3.

As nodes move, new nodes may become neighbors of node u; in order to be able to detect this situation, we will define a set of *potential neighbors* that we will denote by PN_u. In order to define this set, we will need an additional property.

If the constants b, c, and α are chosen carefully, we can prove that any pair of nodes that are neighbors at a level k are necessarily "cousins", that is, either they are neighbors at level $k+1$, or their parents are neighbors at level $k+1$. This property is already given in [55] (in Lemma 4.1) for the values of the constants used in the paper. Here, we generalize it to a wider range of values for b, c, and α:

Lemma 4.10. *Let $u \neq v \in \mathcal{V}$ and $k \in \mathbb{N}$ such that $k \leq L_u \leq L_v$. Assume that $(v,k) \in N_u$. If $b > 1$, $0 < \alpha$, and $c > \frac{2\alpha b}{b-1}$, we have:*

1. *If $k < L_u$, u and v are neighbors at level $k+1$.*

2. *If* $k = L_u < L_v$*,* f_u *and* v *are neighbors at level* $k + 1$.

3. *If* $k = L_u = L_v$*,* f_u *and* f_v *are neighbors at level* $k + 1$.

Proof. As we suppose that u and v are neighbors at level k, we have $d(u, v) < c \cdot b^k$.
Let us analyze each case distinctively:

1. If $k < L_u$, then both nodes u and v are present at level $k + 1$. As $d(u, v) < c \cdot b^k < c \cdot b^{k+1}$, u and v are neighbors also at level $k + 1$.

2. We have $d(f_u, v) \leq d(f_u, u) + d(u, v) \leq \alpha b^{k+1} + cb^k = c \cdot b^{k+1}(\frac{\alpha}{c} + \frac{1}{b})$. Thus, if $c > \frac{\alpha b}{b-1}$, we have $\frac{\alpha}{c} + \frac{1}{b} < 1$, and f_u and v are neighbors at level $k + 1$.

3. $d(f_u, f_v) \leq d(f_u, u) + d(u, v) + d(v, f_v) \leq c \cdot b^{k+1}(\frac{2\alpha}{c} + \frac{1}{b})$. Thus, similarly to the previous case, if $c > \frac{2\alpha b}{b-1}$, then f_u and f_v are neighbors at level $k + 1$.

With $b > 1$, for the lemma to be true, we thus need $c > \frac{2\alpha b}{b-1} > \frac{\alpha b}{b-1}$. □

Note in particular, that if $b = 2$, $c > 4$, and $\alpha = 1$ like in [55], then the conditions of Lemma 4.10 are satisfied. Similarly, we will later use the values $c = 2$, $b \geq 6$, and $\alpha = \frac{b-2}{b}$, which also satifies the conditions of the lemma.

We may then define PN_u, the set of potential neighbors, as the set of "cousin" nodes that are not neighbors of u. v is a potential neighbor of u at level k if the parent of u (or u itself if $L_u \geq k + 1$) and the parent of v (or v itself if $L_v \geq k + 1$) are neighbors at level $k + 1$.

Definition 4.11 (potential neighbors)**.**

$$
\begin{aligned}
PN_u = &\{(v, k \geq 1) : (v, k + 1) \in N_u \vee (f_v, k + 1) \in N_u\} \\
&\cup \{(v, L_u) : (v, L_u + 1) \in N_{f_u} \vee (f_v, L_u + 1) \in N_{f_u}\} \\
&\setminus N_u
\end{aligned}
$$

By Lemma 4.10, it is assured that if two nodes should become neighbors as a consequence of the movements, then they first have to become potential neighbors. This property can then be leveraged in order to update the navigating net under movement of the nodes [55].

Given a set of nodes, if parents and max levels (f_u and L_u) are already assigned (for example with Algorithm 10), a valid assignment of neighbors can be obtained with Algorithm 12. This algorithm goes through all levels of the hierarchy, from top to bottom, and for each node at each level, uses Lemma 4.10 to get the list of the neighbors of that node and of its children one level lower.

To recapitulate, we have seen thus far that a navigating net takes four parameters: b for the level multiplier, γ for the minimum distance among nodes at a given level, α for the maximum parent/child distance, and c for the distance to the neighbors.

4.2.3 Ancestors

In this section, we will see how navigating nets can be used to answer to the CloseNodes$_u(r)$ query. For this, we will extensively use the parent-child relation of the balls. Let us define the *ancestors* of a node u, denoted $ancestors(u)$, as the nodes $u_1, \ldots, u_k$ such that:

- $u_1 = f_u$
- u_k is the root, that is, the unique node v such that $f_v = v$

Algorithm 12 Neighbors initialization

```
1: for all u ∈ 𝒱 do
2:     N_u ← {(u, i) : i ≤ L_u}
3: end for
4: for k = 1 + ⌊log_b(d_max/γ)⌋, ... 2 do   (*Go through all levels, starting at top, except level 1.*)
5:     for all u : L_u ≥ k do   (*Compute neighbors of u and of its children, at level k − 1*)
6:         PN ← ∅
7:         for all v : (v, k) ∈ N_u do   (*Compute potential neighbors of u and of its children*)
8:             for all w : (w, k − 1) ∈ C_v ∨ w = v do
9:                 PN ← PN ∪ {w}
10:            end for
11:        end for
12:        for all w' : (w', k−1) ∈ C_u ∨ w' = u do   (*Get nodes whose neighbors are in PN.*)
13:            for all w ∈ PN do
14:                if d(w, w') < 2 · b^(k−1) then
15:                    N_w' ← N_w' ∪ (w, k − 1)
16:            end for
17:        end for
18:    end for
19: end for
```

- For each $i, 1 \leq i < k$, $u_{i+1} = f_{u_i}$

We also define *the ancestor of u at level ℓ* to be u itself if $L_u \geq \ell$, or the[3] node v with smallest level among the nodes in $ancestors(u)$ whose level is greater than or equal to ℓ. That is, $v \in ancestors(u), L_v \geq \ell$ and for any $v' \neq v$ such that $v' \in ancestors(u), L_{v'} \geq \ell \implies L_{v'} > L_v$.

One useful property is that the distance of a node to any of its ancestors is bounded. The property given in Lemma 4.12 is somewhat similar to the so-called "close-containment property" in [48].

Lemma 4.12. *Let v be the ancestor of u at level ℓ. If $b > 1$, $\alpha \leq \frac{b-1}{b}$, and if the navigating net respects the α-coverage, we have :*

$$d(u, v) < \left(\alpha + \frac{1}{b}\right) b^{\ell} - 1.$$

Proof. In the worst case, $u_1, u_2, \ldots, u_\ell$, the ancestors at level $1, \ldots, \ell$ are distinct nodes. Hence, $u_{i+1} = f_{u_i}$ and thus $d(u_i, u_{i+1}) < \alpha b^{i+1}$. Therefore

$$\begin{aligned}
d(u, v) &\leq d(u, u_1) + d(u_1, u_2) + \ldots + d(u_{\ell-1}, u_\ell) \\
&< \alpha \sum_{i=1}^{\ell} b^i \\
&< \alpha \cdot b^{\ell} + \alpha \left(b \frac{1 - b^{\ell-1}}{1 - b} \right) \\
&< \alpha \cdot b^{\ell} + \frac{b-1}{b} b \left(\frac{b^{\ell-1} - 1}{b - 1} \right) && \text{when supposing that } \alpha \leq \frac{b-1}{b} \\
&< \alpha \cdot b^{\ell} + b^{\ell-1} - 1
\end{aligned}$$

[3] As each node u has only one parent, and such that the level of that parent is strictly greater than the level of u (see Definition 4.7), the ancestor of u at level ℓ is always unique.

$\square$

To find the nodes in the vicinity of a node u, we will look for u', the ancestor of u at a specific level, and then look for all nodes that have u' as ancestor. To look for the ancestor of u, that is, when going up in the hierarchy, one can simply recursively look at the parent of the node until the desired level is reached. However, to be able to go down the hierarchy and look for the set of nodes that have u' as ancestor, nodes have to maintain an additional set of nodes:

Definition 4.13 (children)**.**

$$C_u = \{(v,k) : v \in \mathcal{V} \setminus \{u\} \wedge f_v = u \wedge L_v = k\}$$

Using in concert the ancestors, children and neighbors of the nodes, we can quickly find the nodes that are at distance at most r of any node u. Algorithm 13 describes how to answer to the $\textsc{CloseNodes}_u(r)$ query.

Algorithm 13 Given a distance r and a node u, get the answer to the $\textsc{CloseNodes}_u(r)$ query

```
1: function CloseNodes(u,r)
2:     L ← 1 + ⌈log_b (r/c)⌉
3:     u' ← the ancestor of u at level L    (*might be u itself*)
4:     D ← {v s.t. ∃v' : v' is an ancestor of v ∧ (v', L) ∈ N_u'} ∪ {u}
5:     return {v ∈ D : d(u, v) ≤ r}
```

Lemma 4.14 shows that indeed Algorithm 13 returns the set of nodes at distance at most r of u, as expected.

Lemma 4.14. *If* $b > 1$, $0 < \alpha \leq \frac{b-1}{b}$, *and* $c \geq \frac{2\alpha b+2}{b-1}$, *the function* $\textsc{CloseNodes}(u,r)$ *from Algorithm 13 returns the set* $\{v \in \mathcal{V} : d(u,v) \leq r\}$.

Proof. It is easy to see, on line 5, that every node returned by a call to $\textsc{CloseNodes}(u,r)$ is at distance at most r away from u.

Let v be a node such that $d(u,v) \leq r$. It remains to be proven that v belongs to the set returned by $\textsc{CloseNodes}(u,r)$. Let u', v' be the ancestors at level $L = 1 + \left\lceil \log_b \left(\frac{r}{c}\right) \right\rceil$ of u and v respectively (note that $L_{u'} \geq L$ and $L_{v'} \geq L$). We have $d(u',u) < (\alpha + \frac{1}{b})b^L$, and $d(v',v) < (\alpha + \frac{1}{b})b^L$ by Lemma 4.12.

As $L = 1 + \left\lceil \log_b \left(\frac{r}{c}\right) \right\rceil$, we have $r \leq c \cdot b^{L-1}$, and thus:

$$\begin{aligned} d(u',v') \leq d(u',u) + r + d(v,v') &< 2\left(\alpha + \frac{1}{b}\right)b^L + cb^{L-1} \\ &< cb^L \left(\frac{2\alpha b + 2 + c}{cb}\right) \\ &< cb^L & \text{because } c \geq \frac{2\alpha b + 2}{b-1}. \end{aligned}$$

Therefore, u' and v' are neighbors at level L, so that v is added to the set D on line 4, and as $d(u,v) \leq r$, v belongs to the set returned by $\textsc{ClosedNodes}(u,r)$. $\square$

Thus, the correctness of the algorithm has been proven. Note that with $c = 2$, $\alpha = \frac{b-2}{b}$, and with any $b \geq 6$, as it will later be the case, the conditions of Lemma 4.14 are verified.

We will now analyze its performance. Recall that $b_u(x)$ is the number of nodes in the ball of center u and of radius x. We can measure the performance of the algorithm by bounding the number of nodes that are checked by Algorithm 13.

Lemma 4.15. *In a centralized setting, Algorithm 13 answers to the* CloseNodes *query in* $O(\log r + b_u(O(r)))$ *time.*

Proof. As we can see on line 3, Algortihm 13 first looks for u', the ancestor of u at level L, which takes $O(\log r)$ time.

Then, the algorithms builds D by looking for all nodes v whose ancestor at level L is a neighbor of u'. As each node has only one parent, we have that the building of D on line 4 takes $O(\text{card}(D))$ time. Additionally, with v' the ancestor of v at level L, we have $d(u', v') < c \cdot b^L = O(r)$. By Lemma 4.12, we also have that $d(v, v') \leq (\alpha + \frac{1}{b})b^\ell = O(r)$, and $d(u, u') = O(r)$. Thus, by triangular inequality, all nodes of D are at distance $O(r)$ away from u.

We thus get that Algorithm 13 takes $O(\log r + b_u(O(r)))$ time to complete. □

Let k denote the size of the set returned by the query, that is $k = b_u(r)$. As explained in Chapter 1, if we suppose that $b_u(r \cdot O(1) + O(1)) = O(k)$, then we can write the complexity of Algorithm 13 as $O(\log r + k)$.

Lemma 4.16. *In a synchronous distributed setting, Algorithm 13 answers to the* CloseNodes *query using* $O(\log r)$ *communication rounds and* $O(\log r + k)$ *messages.*

Proof. It takes one communication round to either find the parent of a node or find all children of a node. It thus takes up to $2L = O(\log r)$ communication rounds to execute Algorithm 13. The number of needed messages is similar to the computation complexity in the centralized case (see Lemma 4.15). □

We have thus seen that navigating nets, thanks to their sets of neighbors, enable the structure to be used to efficiently answer to $\text{CloseNodes}_u(r)$ queries. In the next section, we will go a little further still, by adding constraints to the structure, so as to enable efficient maintenance of navigating nets under movements of the nodes.

4.3 Centralized Navigating Nets

In this section, we will study the use of navigating nets in the centralized setting.

4.3.1 Related Work

The main article dealing with navigating nets in the kinetic setting happens to also be the article that introduced the Black-Box model [55]. Their structure, called DefSpanner, is a navigating net[4] with $b = 2$, $\alpha = \gamma = 1$ and $c > 4$. It is shown that the DefSpanner is a spanner, property that is used to answer several geometric queries. The DefSpanner can be maintained both in the dynamic and the kinetic setting when it is supposed that $d_{min} = 1$ (and thus $d_{max} = \Phi$).

The maintenance of the structure under motion of the nodes relies on Lemma 4.10, and uses a list of four types of certificates:

- parent-child certificates, that certify that any node u is at distance strictly smaller than $\alpha \cdot b^{L_u+1}$ from f_u;
- separation certificates, that certify that all neighbors of u at level k are at distance at least $\gamma \cdot b^k$ from u;

[4]Note that in [55] the inequalities are non-strict: for example, $d(u, f_u) \leq \alpha \cdot b^{L_u+1}$, which is slightly different to the strict inequality of Definition 4.7.

- edge certificates, that certify that all neighbors of u at level k are at distance strictly smaller than $c \cdot b^k$;

- and potential neighbor certificates, ensuring that any potential neighbor of u at level k that is not a neighbor of u at that level is at distance at least $c \cdot b^k$ away from u.

Together, the certificates verify the validity of the navigating net: parent-child certificates verify the α-coverage property, separation certificates verify the γ-separation property, given that the sets of neighbors are valid, which is verified by edge certificates (that ensure that the maintained sets of neighbors are subsets of the actual sets of neighbors), and potential neighbor certificates (that ensure, by Lemma 4.10, that the neighbors are supersets of the actual sets of neighbors).

When certificates fail, the structure is updated as follows. In the Black-Box model, it is supposed that $d_{mv} \leq \frac{c}{4-1} \cdot b$, which is true with $d_{mv} = 1$.

- When a parent-child certificate fails, it means that there is a node u getting too far away from its parent; we call u an *orphan*. First, an alternative parent is searched for u among the neighbors of its current parent, and if none is found (that is, if $\forall v \in N_{f_u}, d(u, v) \geq \alpha \cdot b^{L_u+1}$), u is *promoted*, that is, its level is increased by one, and it takes as temporary parent its ancestor at the next level. If u is too far away from that temporary parent, the process is repeated until a valid parent is found. These promotions may never break the γ-separation property, as with the parameters of the DefSpanners, if no alternative parent is found among the neighbors of u's previous or temporary parent, then u's level may be increased without violating the γ-separation.

  When a valid parent is found, the neighbors and potential neighbors of u may be found top down, starting at the new level of u, and using Lemma 4.10: the children of the neighbors of u's ancestor[5] at level ℓ give u's neighbors and potential neighbors at level $\ell - 1$. New certificates are created accordingly.

- When a separation certificate fails between two nodes u and v, the node with lowest level, say u with $L_u = k$ is "demoted", that is, it decreases its level to $k - 1$, and takes v as new parent (which is a valid parent, as $\gamma = \alpha$). Nodes at level $k - 1$ that had u as parent become orphans. These nodes are treated in the same way as nodes created by a failure of parent-child certificate. All certificates involving u at level k are removed from the certificate list, and a new parent-child certificate is added for u and v.

- When an edge certificate (resp. a potential neighbor certificate) fails, the involved nodes are removed (resp. added) to each other's set of neighbors. The edge certificate (resp. potential neighbor certificate) is removed from the certificates list, and a new potential neighbors certificate (resp. edge certificate) is created.

In the Black-Box model, which is the movement model studied in this chapter, updates to the DefSpanner take $O(n \log \Phi)$ computations per time step in the high mobility setting (all nodes are allowed to move at each time step, see p.39), as treating orphans takes $O(\log \Phi)$ time because it is needed to travel all the hierarchy up and down, and because there are $O(\log \Phi)$ levels in the navigating net.

In the article, no result is given for the low mobility setting (when only one node may move at each time step, see p.39). However, as the computation time to repair the structure when a certificate fails is $O(\log \Phi)$, and because each node is involved in $O(\log \Phi)$ certificates, we can

[5] Recall that u's ancestor at level $\ell \leq L_u$ is u itself.

say that the computation time per time step in the low mobility setting is $O(\log^2 \Phi)$. We may however extend the analyses of [55] in the low mobility setting[6], and get a new result improving that update time.

Theorem 4.17. *In the low mobility setting, with $d_{min} = 1$, DefSpanners may be updated with $O(\log \Phi)$ computations per time step.*

Proof. In a navigating net, by γ-separation and doubling dimension, each node has a constant number of neighbors and children per level, and thus also a constant number of potential neighbors. As the nodes have certificates only with their parents, neighbors, and potential neighbors, each node is associated with a constant number of certificates per level, with a maximum total of $O(\log \Phi)$ certificates (per node). We need to prove that the union of all certificates of same type that may get invalidated at one time step do not incur more than $O(\log \Phi)$ computations.

- The cost to repair an edge certificate or a potential neighbor certificate is $O(1)$, as it involves just to delete or add a single node to the two sets of neighbors, and to delete and create one certificate. The newly created certificate is always true by definition. Thus the total cost for those types of certificates is $O(\log \Phi)$.

- Let u be the node that moved. By α-coverage (Definition 4.7), there may be only one parent-child certificate where u is the child, which takes $O(\log \Phi)$ time to be repaired, as it is explained in [55] that any single certificate may be repaired in $O(\log \Phi)$.

  It thus remains to be proven that the time to repair all parent-child certificates where u is the parent is $O(\log \Phi)$. For this, we observe that each time one of the children of u increases its level too much, other children at lower levels cannot increase their level more than a constant number of times. Indeed, let v be a child of u at level k. Before the movement, at time t_i, we have $d(u, v, t_i) < \alpha \cdot b^{k+1}$. After movement, at time t_{i+1}, we have $d(u, v, t_{i+1}) < \alpha \cdot b^{k+1} + d_{mv}$. Let w be a child of u at level $k + 1$. We have $d(v, w, t_{i+1}) < \alpha \cdot b^{k+2} + \alpha \cdot b^{k+1} + d_{mv} < \gamma \cdot b^{k+3}$ (as $d_{mv} = 1$, so that $d_{mv} < b^{k+1}$). Thus, if v's level is increased to $k + 3$, w cannot go up in the hierarchy higher than level $k + 2$ without violating the γ-separation property. As u has a constant number of children at each level, we thus have that the cost of maintaining parent-child certificates is $O(\log \Phi)$.

- As we have explained, to repair a failing separation certificates, one of the two involved nodes decreases its level. By simply putting a higher priority for the node u that moved to decrease its level, only $\log \Phi$ level decreases are needed at maximum, as u cannot decrease its level below 0 (since $d_{min} = 1$). The orphans that appear because of those level changes, that is, the nodes v such that $f_v = u$, and so that L_v is greater or equal to the new level of u, so that u is no longer a valid parent for v (see Definition 4.7, page 115), induce the same computation cost as parent-child certificates. □

The main ideas for our constrained navigating nets in the following section come from [48]. In that article, the authors study navigating nets in the dynamic setting (that is, the nodes do not move, but they may be added to or deleted from the structure), with approximate nearest neighbor queries as the main target (see Section 1.5.1). Insertions and deletions take $O(\log n)$ time[7]. The main achievement of the article is that the bounds do not depend on the aspect ratio Φ.

This time complexity for node insertion is attained thanks to several tricks. First, α is fixed at a sufficiently small value so as to get a property equivalent to Lemma 4.12, that is, so that

[6] and in particular the theorem that is numbered 4.2 in that article

[7] Finding a $(1 + \varepsilon)$-approximate nearest neighbor takes $O(\log n) + (1/\varepsilon)^{O(1)}$.

any node is close enough to any of its ancestors. Secondly, and more importantly, a notion of "ball safety" is introduced, with three levels of safety, so that a ball is safer the closer it is to its parent. This safety is defined in such a way that a ball gets better safety when it is promoted (that is, when the level of the associated node is increased), and so that the safest of balls never need to be promoted. Associated with insertion rules, this ensures that no more than a constant number of promotions of existing balls are needed when inserting a new node in the structure, which is crucial when looking for insertion times that are independent from the aspect ratio. See Section 4.3.2 for a description of our similar property. The article also relies on so-called modified biased skip lists to compute the level and parent of a new node that is to be inserted, but these are not needed in our kinetic setting.

Also, the article introduces the notion of jumps, removing the need for nodes u to maintain balls $B_u\left(b^k\right)$ at each level $k \leq L_u$. The main reason for the use of jumps is to achieve linear space: as all nodes are in level 0, it may seem necessary to avoid storing all balls for each level. We will however see that we can achieve a linear size with a much simpler technique[8].

In [48], nodes do not have a unique parent: instead, for any level $k \leq L_u$, a node u maintains the set of nodes v with $L_v \geq k + 1$ such that $d(u, v) < \alpha \cdot b^{k+1}$ (let us call this set the *candidate parents*). However, their definition of navigating nets is asymptotically equivalent to ours, as the set of candidate parents can be reconstructed in $O(1)$ time using our definition of navigating net, thanks to Lemma 4.10, as in [48], the constants used are $b = 5$, $\alpha = 3/5$, and $c = 2$. Memory is also equivalent, because in metric spaces with bounded doubling dimension, the set of candidate parents of a given level is constant in size.

In [59], the authors reuse the structure from [48] in order to build a spanner.

Limitations of DefSpanners In DefSpanners, the dependency on $\log \Phi$ of the costs has two origins.

First, each node may be involved in up to $O(\log \Phi)$ certificates (one for each level the node appears in), and in some specific situations, even for small movements, all of them may fail at once. It follows that the cost for repairing all these failing certificates cannot be reduced below $O(\log \Phi)$. As by definition of navigating nets, a node may be involved in that many levels, if it has for example a neighbor at each of its levels, this problem seems difficult to avoid.

The second reason for this cost, is that even if taken together, all failing certificates take $O(\log \Phi)$ operations to be treated, as proven in Theorem 4.17, in some cases single failing certificates may need $O(\log \Phi)$ operations to repair the structure. This situation is represented on Figure 4.3, where a node u gets outside of the range of its parent, invalidating a parent-edge certificate. As each ancestor of u is just barely in range of its own parent, the only ancestor that is a valid parent for u is the root, so that u needs to go up all the hierarchy, resulting in $O(\log \Phi)$ computations.

We will see in the next section how to avoid this second problem.

4.3.2 Constrained Navigating Nets

In this section, we reuse concepts from [48], that make it so that when a certificate fails, the associated nodes may change their level only a constant number of times. The main idea is to observe that the scenario from previous section leading a single node to go up all the hierarchy is rare, and that it is possible to detect in advance situations that may lead to that scenario, and to immediately take action, using only a constant number of computations. In other words,

[8]Note however, that our solution is not "truly linear" as defined in the article, that is, our solution hides a dependency on the doubling dimension.

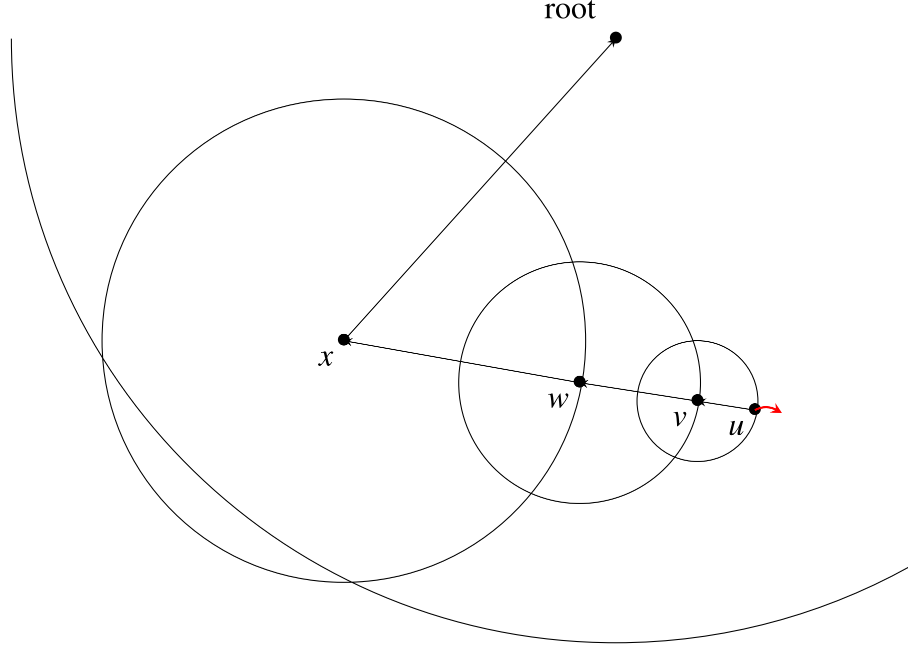


Figure 4.3 – Example of a case where DefSpanner incurs a node to go up all the hierarchy for only a small movement. Thin black arrows represent parent/child relations, the thick red arrow represents a movement of u that could incur $O(\log \Phi)$ computations.

this improvement of navigating nets enable to spread out over time the cost induced by the scenario from Figure 4.3. To achieve this, a property is added to the structure (hence the name *constrained* navigating net) that we call *ancestor invariant*.

In addition to this new property, we add a small improvements over DefSpanners for simplicity. We may thus identify two main differences between our structure and the one from [55]:

- The structure has to comply to an additional property that we call ancestor invariant.
- The parameters are chosen such that two nodes may be neighbors only at one specific level.

Our structure uses the following parameters:

- $b \in \mathbb{N}, b \geq 6$
- $c = 2$
- $\alpha = \frac{b-2}{b} > \beta = \frac{b-3}{b} > \gamma = \frac{b-4}{b}$ (the use of β will be described later).

See Figure 4.4 for an example of the application of α, β, and γ as distance multipliers. We can see that around each node, three "degrees of proximity" for the children may be defined: either v, child of u is such that $d(v,u) < \gamma \cdot b^{L_v+1}$, or $\gamma \cdot b^{L_v+1} \leq d(v,u) < \beta \cdot b^{L_v+1}$, or $\beta \cdot b^{L_v+1} \leq d(v,u) < \alpha \cdot b^{L_v+1}$. We can also see that with $b \geq 6$, the radii grow very fast.

As movements are small ($d_{mv} = 1 \leq b^k$ for $k \geq 0$), a node may only change its "degree of proximity" by one "degree" per time step : if $d(v,u,t_i) < \gamma \cdot b^{L_v+1}$, then we may not have $d(v,u,t_{i+1}) \geq \beta \cdot b^{L_v+1}$ for example. The idea of constrained navigating nets is to reduce the number of nodes v such that $\beta \cdot b^{L_v+1} \leq d(v,f_v) < \alpha \cdot b^{L_v+1}$, so that only few nodes may break the α-coverage property at once.

Before looking at the specific properties of constrained navigating net, let us first make a quick remark concerning potential neighbors.

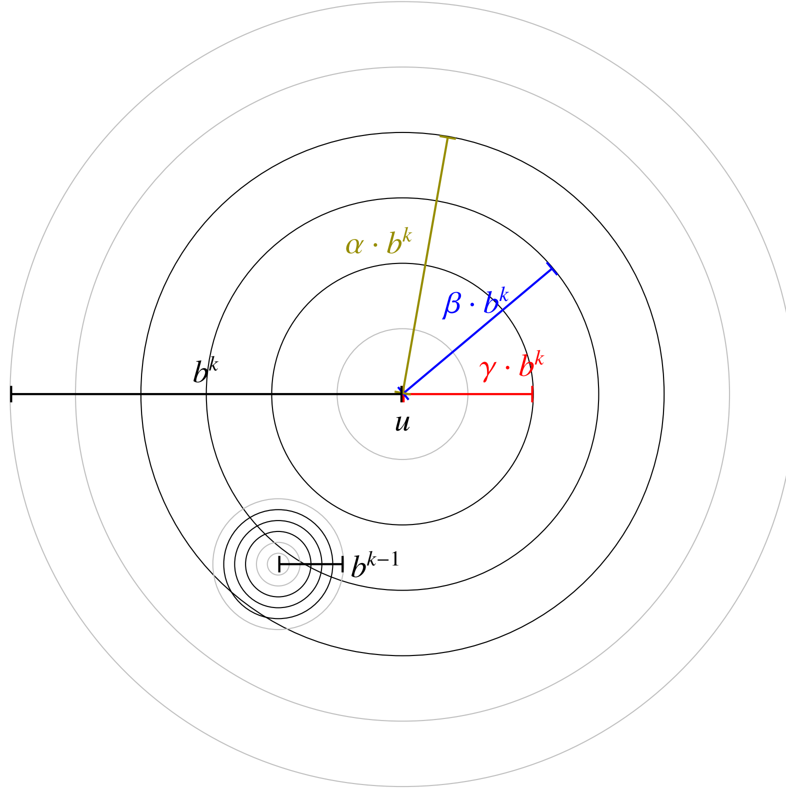


Figure 4.4 – Representation of the values of α, β, and γ.

Usage of Potential Neighbors We have seen in Section 4.2.2, the set PN_u of "cousins" of u, so that each node that becomes a neighbor of u must first become one of those potential neighbors. However, the set PN_u can be used in a slightly stronger way in our model, thanks to Lemma 4.18, which improves on Lemma 4.10. Lemma 4.18 shows that at each time step, this set of potential neighbors includes all nodes that may become neighbors at the next time step. As nodes move of maximum d_{mv} distance units per time step, we thus want PN_u to include at each level k, all nodes at a distance smaller than $c \cdot b^k + 2d_{mv}$ from u. Lemma 4.18 proves that any node at distance $\left(c + \frac{2}{b}\right) b^k$ away from u[9] necessarily satisfies one of the conditions of Definition 4.11, so that $v \in PN_u$.

Lemma 4.18. *Let $u \neq v \in \mathcal{V}$ and $k \in \mathbb{N}$ such that $k \leq L_u \leq L_v$. Assume that $d(u,v) \leq (c + \frac{2}{b})b^k$. If $b > 1$, $0 < \alpha$, and $c > \frac{2\alpha b^2+2}{b^2-b}$, we have:*

1. *If $k < L_u$, u and v are neighbors at level $k+1$.*

2. *If $k = L_u < L_v$, f_u and v are neighbors at level $k+1$.*

3. *If $k = L_u = L_v$, f_u and f_v are neighbors at level $k+1$.*

Proof. Let us analyze each case distinctively:

1. We have $c > \frac{2\alpha b^2+2}{b^2-b} > \frac{2}{b^2-b}$, so that $(c + \frac{2}{b})b^k < c \cdot b^{k+1}$. It follows that $d(u,v) < c \cdot b^{k+1}$, so that u and v are neighbors at level $k+1$.

[9]Note that that with $d_{mv} = 1$, $b \geq 1$ and $k \geq 1$, we have $c \cdot b^k + 2d_{mv} \leq \left(c + \frac{2}{b}\right) b^k$. Also, note that as most of our next results in this chapter focus on the low mobility setting (in which, as only one node moves at a time, the distance between two nodes may not change of more than $1 \cdot d_{mv}$ distance units per time step), it would be enough to prove that $d(u,v) \leq (c + \frac{1}{b})b^k$. The stronger property of Lemma 4.18 opens up several possibilities for future analyses in the high mobility setting (in which the distance between two nodes may change of up to $2 \cdot d_{mv}$ distance units per time step).

2. $d(f_u, v) \leq d(f_u, u) + d(u, v) \leq \alpha b^{k+1} + (c + \frac{2}{b})b^k$. One can check that when $c > \frac{\alpha b^2+2}{b^2-b}$, we get $d(f_u, v) < c \cdot b^{k+1}$, so that f_u and v are neighbors at level $k + 1$.

3. $d(f_u, f_v) \leq d(f_u, u) + d(u, v) + d(v, f_v) \leq 2\alpha b^{k+1} + (c + \frac{2}{b})b^k$. Again, one can check that when $c > \frac{2\alpha b^2+2}{b^2-b}$, we get $d(f_u, f_v) < c \cdot b^{k+1}$, so that f_u and f_v are neighbors at level $k + 1$. □

Note that with $\alpha = \frac{b-2}{b}$ and $b > 1$, we have $\frac{2\alpha b^2+2}{b^2-b} < 2$, so that our choice of parameters satisfies the conditions of Lemma 4.18.

Limiting Levels of Neighbors One very useful property, is that when using specific parameters, two nodes may be neighbors only at one level.

Lemma 4.19. *With $c \leq \gamma \cdot b$, two nodes $u \neq v$, can only be neighbors at level $k = \min(L_u, L_v)$.*

Proof. If two nodes $u \neq v$ are neighbors at level k, then $d(u, v) < c \cdot b^k$. With $c \leq \gamma \cdot b$, we thus have $d(u, v) < \gamma \cdot b^{k+1}$. Thus, by definition of the navigating net, balls $B_u\left(b^{k+1}\right)$ and $B_v\left(b^{k+1}\right)$ are not allowed to both exist. It thus follows that $k = \min(L_u, L_v)$. □

Note that with $\gamma = \frac{b-4}{b}$, $b \geq 6$ and $c = 2$, our choice of parameters satisfies the conditions of Lemma 4.19, so that the result is valid on our constrained navigating net.

The main advantage of this property is that it makes maintenance algorithms simpler, as it removes special cases to be treated. In particular, we will see with Theorem 4.47 (Section 4.3.5) that this property allows to easily prove that the size of the navigating net is linear, which is much simpler than the introduction of jumps and the associated need for an invariant to be maintained as in [48][10].

Another side effects of this property, is that when a navigating net has parameters complying to Lemma 4.19, the sets of neighbors for a node u can be reconstructed simply by looking at the set of children of f_u, which takes a constant amount of computations (as in a doubling space, the set of children of a specific level is of constant size). It follows that Algorithm 13 can be adapted to be executed with similar performances on navigating nets that do not maintain the sets of neighbors, provided that the parameters comply to the conditions of Lemma 4.19.

Note also that thanks to Lemma 4.19, we have that a node can be a potential neighbor of another node only at one specific level. In other words, we cannot have $(v, k) \in PN_u$ and $(v, k') \in PN_u$ with $k \neq k'$.

Spanner In this section, we prove that navigating nets are spanners.

Theorem 4.20. *If $b \geq 1$, $\alpha \leq \frac{b-1}{b}$, and $c > 2\alpha + \frac{2}{b}$, navigating nets are spanners of stretch factor $\left(1 + \frac{4b\alpha+4}{c-2\alpha-\frac{2}{b}}\right)$.*

Proof. Let $u, v \in \mathcal{V}$, and let i be the smallest level where the ancestors of u and v at level i are neighbors. Let u_i and u_{i+1} denote the ancestors of u at level i and $i + 1$ respectively, and let v_i and v_{i+1} be the ancestors of v.

By Lemma 4.12, we have:

$$d(u, u_i) \leq \left(\alpha + \frac{1}{b}\right) \cdot b^i$$
$$d(v, v_i) \leq \left(\alpha + \frac{1}{b}\right) \cdot b^i$$

[10]Note however that in [48], the size if the structure is so-called *truly linear*, that is that it does not hide a constant depending on the doubling dimension of the considered space, whereas our constrained navigating net has a size that depends on the doubling dimension.

By definition of i, we have:

$$d(u_i, v_i) \leq c \cdot b^i$$
$$d(u_{i-1}, v_{i-1}) > c \cdot b^{i-1}$$

Thus we have, by triangular inequality:

$$\begin{aligned} d(u_i, v_i) &\leq d(u_i, u) + d(u, v) + d(v, v_i) \\ &\leq d(u, v) + 2\left(\alpha + \frac{1}{b}\right) b^i \end{aligned} \tag{4.1}$$

We also have:

$$\begin{aligned} & d(u_{i-1}, v_{i-1}) \leq d(u_{i-1}, u) + d(u, v) + d(v, v_{i-1}) \\ \Longrightarrow\ & c \cdot b^{i-1} - 2\left(\alpha + \frac{1}{b}\right) b^{i-1} \leq d(u, v) \\ \Longrightarrow\ & b^{i-1}\left(c - 2\left(\alpha + \frac{1}{b}\right)\right) \leq d(u, v) \\ \Longrightarrow\ & 4\left(\alpha + \frac{1}{b}\right) b^i \leq d(u, v) \frac{4\left(\alpha + \frac{1}{b}\right) b^i}{b^{i-1}\left(c - 2\left(\alpha + \frac{1}{b}\right)\right)} \qquad \text{because } c > 2\alpha + \frac{2}{b} \\ \Longrightarrow\ & 4\left(\alpha + \frac{1}{b}\right) b^i \leq d(u, v) \frac{4b\left(\alpha + \frac{1}{b}\right)}{c - 2\left(\alpha + \frac{1}{b}\right)} \end{aligned}$$

Let $\Lambda(u, v)$ be the length of the shortest path from u to v in our structure. We may see that the proof of Lemma 4.12 uses the sum of the intermediate edges. The same reasoning can thus also give:

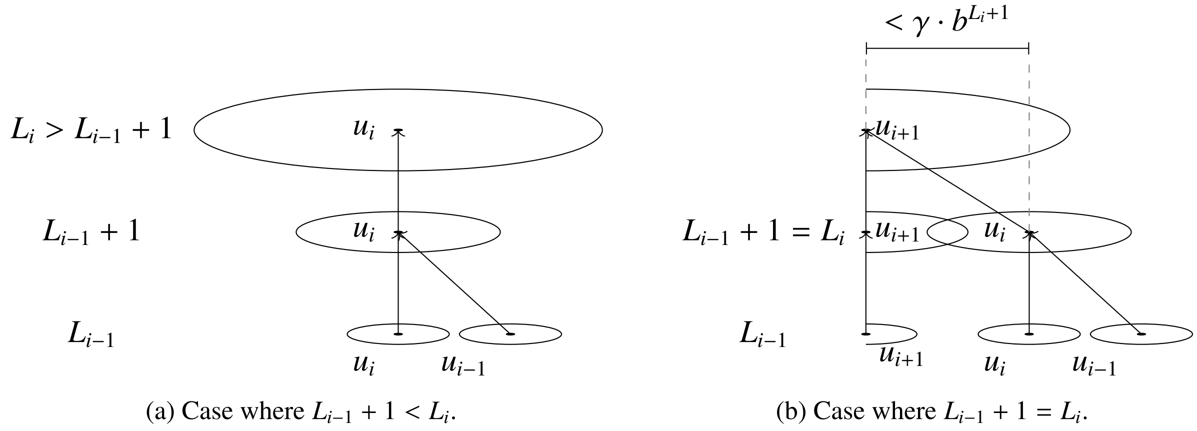
$$\Lambda(u, u_i) \leq \left(\alpha + \frac{1}{b}\right) b^i$$
$$\Lambda(v, v_i) \leq \left(\alpha + \frac{1}{b}\right) b^i$$

We thus have:

$$\begin{aligned} \Lambda(u, v) &\leq \Lambda(u, u_i) + d(u_i, v_i) + \Lambda(v_i, v) \\ &\leq d(u, v) + 4\left(\alpha + \frac{1}{b}\right) b^i \quad \text{(by Equation 4.1)} \\ &\leq d(u, v) + d(u, v) \frac{4b\left(\alpha + \frac{1}{b}\right)}{c - 2\left(\alpha + \frac{1}{b}\right)} \\ &\leq d(u, v)\left(1 + \frac{4b\left(\alpha + \frac{1}{b}\right)}{c - 2\left(\alpha + \frac{1}{b}\right)}\right) \end{aligned}$$

□

Note that with $\alpha = \frac{b-2}{b}$, $b \geq 6$, and $c = 2$, the conditions of Theorem 4.20 are met. While these conditions are not met for DefSpanners (where $\alpha = 1$), it is proven in [55] that these structures have a stretch factor of $1 + \frac{16}{c-4}$ (with $c > 4$). This means that for high values of c, stretch factors close to 1 can be obtained for DefSpanners. For $\alpha = 2/3$, $b = 6$, and $c = 2$, the stretch factor given by Theorem 4.20 becomes 61, which is thus higher than what can be achieved with DefSpanners.



(a) Case where $L_{i-1} + 1 < L_i$.

(b) Case where $L_{i-1} + 1 = L_i$.

Figure 4.5 – Representation of the two cases of Definition 4.21

Ancestor Invariant Let $u_1, \ldots, u_k$ be the ancestors of node u (ordered by increasing level), with k the level of the last ancestor of u, that is, the level of the root. To simplify the definitions, let $u_0 = u$, and $L_0 < L_1 < \ldots < L_k$ the levels $L_{u_0}, L_{u_1}, \ldots, L_{u_k}$.

Let us define $\gamma a(u)$, the *γ-ancestor* of u.

Definition 4.21 (γ-ancestor)**.** *$\gamma a(u) = u_i$ iff $1 \leq i \leq k$ is the smallest i such that either :*

- *$L_{i-1} + 1 < L_i$ or*
- *$L_{i-1} + 1 = L_i$ and $d(u_i, f_{u_i}) < \gamma \cdot b^{L_i+1}$.*

In the case where no ancestor of u satisfies one of these properties, we say that u does not have a γ-ancestor, and we note $\gamma a(u) = \bot$.

This definition is equivalent to saying that the γ-ancestor u_i is the first ancestor of u such that the distance between u_i and the ancestor at level $L_i + 1$ is smaller than $\gamma \cdot b^{L_i+1}$. Indeed, if $L_{i-1} + 1 < L_i$, it means that the ancestor at level $L_i + 1$ is u_i, in which case this distance is 0. See Figure 4.5 for an example.

The only node that doesn't have a γ-ancestor is the root, as by definition of the ancestors page 118, the root v has only one ancestor $v_1 = v$ that does not satisfy either of the two conditions of Definition 4.21.

Definition 4.22 (α-ancestor)**.** *$\alpha a(u) = u_i$ iff $1 \leq i < k$ is the smallest i such that :*

- *$L_{i-1} + 1 = L_i$ and $\beta \cdot b^{L_i+1} \leq d(u_i, f_{u_i}) < \alpha \cdot b^{L_i+1}$*

In the case where no ancestor of u satisfies this property, we say that u does not have a α-ancestor, and we note $\alpha a(u) = \bot$.

Let us set that $L_{\alpha a(u)} = +\infty$ when $\alpha a(u) = \bot$. The main addition in constrained navigating nets is the following property:

Definition 4.23 (Ancestor invariant)**.** *$\forall u \in \mathcal{V}$, $\beta \cdot b^{L_u+1} \leq d(u, f_u) < \alpha \cdot b^{L_u+1} \implies L_{\gamma a(u)} < L_{\alpha a(u)}$*

In the definition above, as $L_\bot = +\infty$, if a non-root node does not have an α-ancestor, then it respects the ancestor invariant. Note also that even if the root does not have a γ-ancestor, it satisfies the ancestor invariant, as when u is the root, $d(u, f_u) = 0$ and $\beta \cdot b^{L_u+1} > 0$.

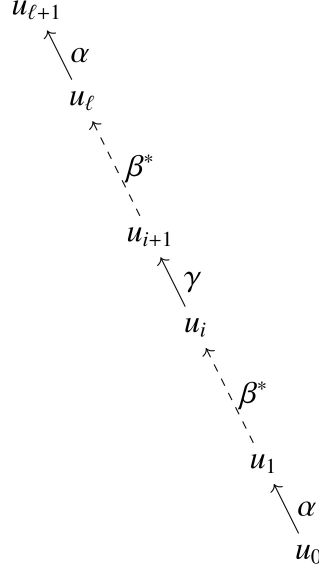


Figure 4.6 – Graphic representation of the ancestor invariant.

A graphical representation of the ancestor invariant can be found on Figure 4.6. Arrows with a solid line represent a parent/child relation, and have labels representing their role in the ancestor invariant. u_i and u_{i+1} have an arrow with a γ, meaning that $d(u_i, u_{i+1}) < \gamma \cdot b^{L_{u_i}+1}$, or that $L_{u_{i+1}} > L_{u_i} + 1$ (so that u_i can be a γ ancestor for its descendants). u_0 and u_1 have an α arrow, meaning that $\beta \cdot b^{L_{u_0}+1} \leq d(u_0, u_1) < \alpha \cdot b^{L_{u_0}+1}$, and that $L_{u_1} = L_{u_0} + 1$ (so that u_0 can be an α ancestor for its descendants). A β means that neither of the others is verified: if there is a β arrow from v to w, it means that $f_v = w$, with $\gamma \cdot b^{L_v+1} \leq d(v, w) < \beta \cdot b^{L_v+1}$ and $L_w = L_v + 1$. Arrows with dashed lines represent an arbitrary number of successive parent/child relations.

We can see on Figure 4.6 that arrows with a β label are neutral with regards to the ancestor invariant, as there can be arbitrarily many without impacting the invariant. We can also see that between each α in the hierarchy, there must be a γ reflecting Definition 4.23.

Note that by assigning levels and parent as in Algorithm 10, the invariant is satisfied, simply because all nodes u satisfy $d(u, f_u) < \gamma \cdot b^{L_u+1}$. The invariant has no use when one is interested in proximity queries. However, maintaining the invariant allows to repair the structure with a constant number of changes per node after a (small) movement.

4.3.3 Maintain Constrained Navigating Nets When Nodes Move

Data Structure

The data structure contains, for each node u of $\mathcal{V}$, the following variables:

- L_u the highest level of u,
- f_u the parent of u,
- C_u the set of children of u,
- N_u the set of neighbors of u.

Let us make some remarks regarding these variables.

As in the previous chapter, these notations refer to the current values of the variables. If time is ambiguous, we may refer to the values of these variables at specific instants; for example, $N_u(t)$ refers to the content of the set N_u at instant t.

With only levels, parents, and children, thanks to Lemma 4.10, the sets of neighbors at a level k of a node u could be recomputed in constant time (as a node has only a constant number of neighbors and children at a given level). One may notice that in our algorithms, the sets of neighbors are always used only at specific levels at a time (N_u is never used as a whole, only for example $\{(v, L) \in N_u : L = k\}$, the subset of neighbors at level k of u). This means that N_u is actually optional in the data structure, as it could be recomputed each time it is needed.

The same goes for PN_u, which we however did not include in the data structure. In our algorithm, we never explicitly use PN_u (except at initialization), and instead directly recompute it using the other variables of the data structure. However, we may use PN_u in our proofs as mathematical notation.

Certificates

The correctness of the data structure is ascertained with certificates, each one involving two nodes and satisfied whenever the distance between these two nodes is above or below a certain threshold. We denote by $Cert(t)$ the set of certificates of the structure at time t (or just $Cert$ if time is unambiguous). These certificates are then associated with updates to execute when they fail, resulting in the kinetic algorithm for constrained navigating nets. For ease of notation, we will call that algorithm $\mathcal{A}_{cnn}$. Algorithms 14, 15, and 16 give the instructions of $\mathcal{A}_{cnn}$.

The set $Cert$ contains five kinds of certificates.

- $\texttt{Separation}_{\gamma,k}(u, v)$: $d(u, v) \geq \gamma b^k$, for neighbors at level k.
- $\texttt{ShortEdge}_{c,k}(u, v)$: $d(u, v) < cb^k$, for neighbors at level k.
- $\texttt{LongEdge}_{c,k}(u, v)$: $d(u, v) \geq cb^k$, for potential neighbors at level k.
- $\texttt{Cover}_{\tau,k}(u, v)$: $d(u, v) < \tau b^{k+1}$, for a non-root node u and its parent f_u. τ will be either α, β or γ. Note that b is raised to the power of $k + 1$; k is intended to be the level of u and v is expected to be equal to f_u.
- $\texttt{SCover}_{\tau,k}(u, v)$: $d(u, v) \geq \tau b^{k+1}$, for a non-root node u and its parent f_u. τ will be either γ or β. Again, k is intended to be the level of u and v is expected to be equal to f_u.

A summary of these certificates can be found on Table 4.3. The column "Certificate" shows the name of the certificate, the column "Predicate" gives the logical definition of the certificate, and the column "Condition" indicates under which conditions the certificate exists.

Certificate	**Predicate**	**Condition**
$\texttt{Separation}_{\gamma,k}(u, v)$	$d(u, v) \geq \gamma b^k$	$(v, k) \in N_u$
$\texttt{ShortEdge}_{c,k}(u, v)$	$d(u, v) < cb^k$	$(v, k) \in N_u$
$\texttt{LongEdge}_{c,k}(u, v)$	$d(u, v) \geq cb^k$	$(v, k) \in PN_u$
$\texttt{Cover}_{\tau,k}(u, v)$	$d(u, v) < \tau b^{k+1}$	$v = f_u \neq u$, $k = L_u$, $\tau = \textsc{RoundDistance}(u, v, k)$ (see Algorithm 14)
$\texttt{SCover}_{\tau,k}(u, v)$	$d(u, v) \geq \tau b^{k+1}$	$v = f_u \neq u$, $k = L_u$, $\tau = \gamma$ if $\textsc{RoundDistance}(u, v, k) = \beta$, β if $\textsc{RoundDistance}(u, v, k) = \alpha$

Table 4.3 – List of certificates associated with node u.

For the certificates ShortEdge, LongEdge, and Separation, we define the symmetric certificates as being the same certificates, but inverting the nodes. For example, the symmetric of $\texttt{Separation}_{\gamma,k}(u, v)$ is $\texttt{Separation}_{\gamma,k}(v, u)$. Note that if a certificate is in *Cert* than its symmetric is in *Cert* too.

Algorithm to Maintain Centralized Navigating Nets in the Black-Box Model: $\mathcal{A}_{cnn}$

The centralized algorithm that updates a constrained navigating nets in the Black-Box model, $\mathcal{A}_{cnn}$, is described in Algorithms 14, 15, and 16.

For better readability, we make the following abuses of notation:

- As $\gamma < \beta < \alpha$, and because these parameters represent successive "ranks" of closeness between parents and children, we sometimes write $\tau - 1$ or $\tau + 1$ when $\tau \in \{\gamma, \beta, \alpha\}$ to refer to the "next" or "previous" rank. For example, when $\tau = \beta$, $\tau + 1$ refers to α, and $\tau - 1$ refers to γ.
- As a result, we sometimes add, replace, or remove certificates of the form $\texttt{SCover}_{\tau-1,k}(u, v)$ where $\tau = \gamma$. These certificates do not exist, and these instructions should be ignored. This allows to avoid writing cumbersome conditions on the value of τ.

The main loop of $\mathcal{A}_{cnn}$ is described on lines 16 through 19. Failing certificates are treated in three steps: the Separation certificates first, then the ShortEdge and LongEdge certificates, and finally the SCover and Cover certificates. Each of these steps are done in order of decreasing level, that is, certificates with a high level parameter k are always treated before certificates of same type but with lower k. When two certificates are of same type and same level, the order of execution may be arbitrary. However, in the case of Separation certificates (as can be seen on line 35, where the execution depends on whether $u \prec v$) some priorities are applied depending on an order on the nodes.

This order (with the symbol $\prec$) may actually be chosen as arbitrary: in Section 4.3.4, we prove that the algorithm is valid regardless of how the order of the nodes is computed. However, some performance results depend on a nicely chosen order: in Section 4.3.5, we prove that when only one node moves at each time step (the low mobility setting, see p.39), choosing a simple order in which the node that moved is of higher order than other nodes, leads to nodes that cannot change their level more than a constant number of times per time step. Also, we will explain in Section 4.5.1 why we believe that choosing an order that depends on previous operations on the nodes can lead to a linear computation time when all nodes may move at each time step (the high mobility setting, see p.39).

The updates associated with each failure of certificate are then described in Algorithms 15, and 16:

- When a Separation certificate fails between two nodes u and v at level L (see Figure 4.7 and line 35 of Algorithm 16), one of the two nodes is demoted, that is, its level gets decreased. The node that gets demoted is the one with the lowest level, or any of both if they have the same level (the choice is arbitrary). Let us suppose that u is the one to get demoted; u is removed from level L, and it takes v as new parent. Because of this demotion, every child of u at level $L - 1$ becomes an "orphan", as u is no longer a valid parent for them as it breaks the α-coverage property (see Definition 4.7). Each orphan is either given a new parent through the Redirect function (line 38) if a node at level L is found that is at distance γb^L from that orphan, like w and u' on Figure 4.7, or its level is increased though the Promote function if none is found (line 39), like w' on

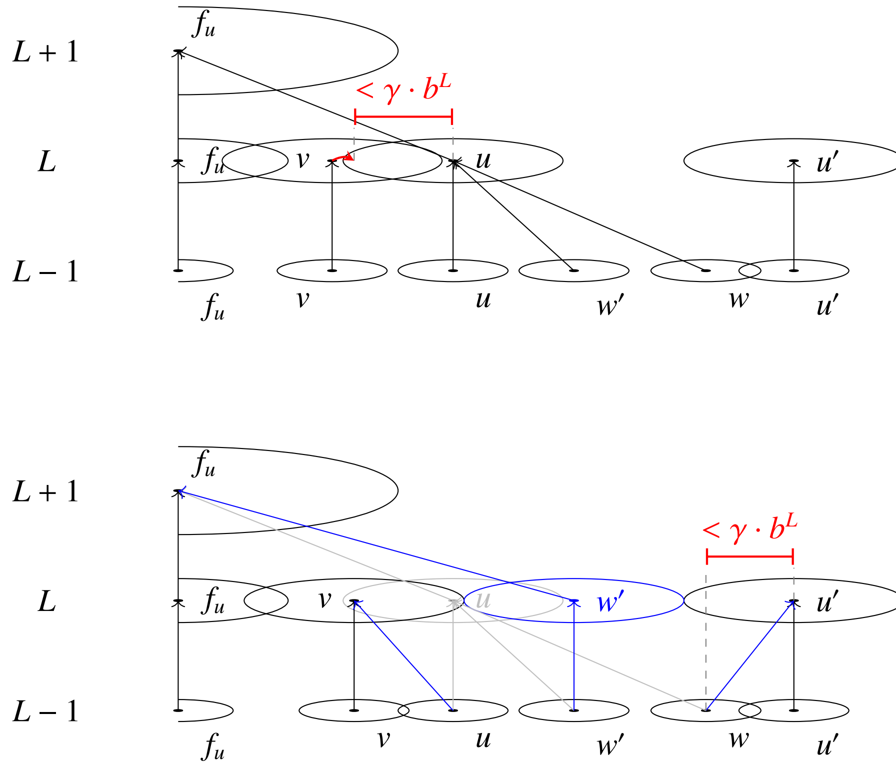


Figure 4.7 – An example of a failure of a `Separation` certificate: at the top, the situation of the nodes before the updates, and at the bottom the result of the updates from $\mathcal{A}_{cnn}$. The circles represent here the balls of radius $\gamma \cdot b^k$ where k is the level (either $L-1$, L, or $L+1$).

Figure 4.7. Note that if w is redirected to a new parent, then we have $d(w, f_w) < \gamma \cdot b^{L_w+1}$, which corresponds to the second condition of Definition 4.21, and thus does not break the ancestor invariant property for w's descendants. Similarly, if w' is promoted to level L, as w' doesn't have any children at level $L-1$ yet, for any child x of w', we have that $L_x + 1 < L_{w'}$, which corresponds to the first condition of Definition 4.21. Orphans are thus treated in such a way that the ancestor invariant property is never broken for any descendent of w.

- When a `ShortEdge` or `LongEdge` certificate fails (line 47 and line 50), then the sets of neighbors and the list of certificates are updated accordingly: when a `ShortEdge` certificates fails between two nodes, both are removed from their respective sets of neighbors, and when a `LongEdge` certificate fails, the two nodes are added to each other's sets of neighbors.

- When a `SCover` certificate fails (line 53), nothing else then changing the values of the certificates has to be done. When a parent v and its child u fail a $\texttt{SCover}_{\tau,L}(u, v)$ certificate, then their $\texttt{Cover}_{\tau+1/b,L}(u, v)$ certificate is replaced by a $\texttt{Cover}_{\tau,L}(u, v)$ in *Cert* certificate. Similarly, the $\texttt{SCover}_{\tau,L}(u, v)$ certificate is replaced by a $\texttt{SCover}_{\tau-1,L}(u, v)$ certificate, but this may happen only if $\tau = \beta$, as there are no $\texttt{SCover}_{\alpha,L}$ certificates, and as there are no values for τ below γ.

- When a `Cover` certificate fails between u at level L and its parent v, the action that is taken is different depending on the distance threshold that has been overtaken between u and v. If the distance between u and v has become greater than $\alpha \cdot b^{L+1}$ (as represented on the top of Figure 4.8, and described on line 68 of Algorithm 16), it means that v is

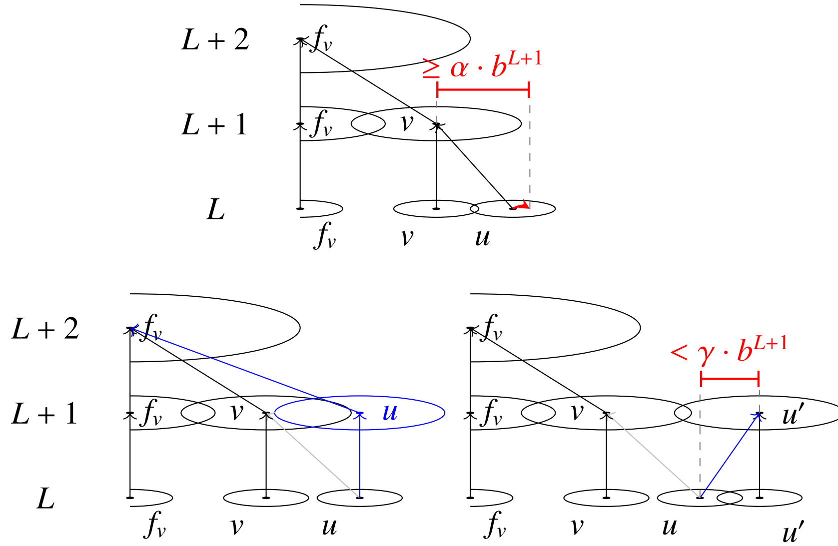


Figure 4.8 – An example of a failure of a $\texttt{Cover}_{\alpha,L}(u,v)$ certificate: at the top, the situation of the nodes before the updates, and at the bottom, two possible results for the updates from $\mathcal{A}_{cnn}$. The circles represent here the balls of radius $\alpha \cdot b^k$ where k is the level (either L, $L+1$, or $L+2$).

no longer a valid parent for u with regards to the α-coverage (Definition 4.7); there are two ways u may be treated, like an orphan described above, and as represented on the bottom of Figure 4.8: either a new parent u', at distance $\gamma \cdot b^{L+1}$ of u is found (bottom right of Figure 4.8), or u is promoted so as to not break the ancestor invariant for all its descendants (bottom left of Figure 4.8). If the threshold of distance between u and v is either $\beta \cdot b^{L+1}$ or $\gamma \cdot b^{L+1}$ (line 57), then there is a risk that the ancestor invariant is broken. In both cases, the algorithm thus looks for the α-ancestor of u, and if its level is lower than the level of u's γ-ancestor, then that α-ancestor is treated as an orphan: either it is redirected to a new parent, or promoted to a new level, similarly to what happens to u on Figure 4.8. Finally, if the threshold is $\beta \cdot b^{L+1}$, u also gets treated as an orphan (lines 61 through 61 of Algorithm 16), again, as on Figure 4.8.

These main actions are complemented with instructions to keep all sets up to date, and to add and remove certificates accordingly. For example, lines 21 though 33 describe additional instructions that should be executed each time changes are performed on the sets of neighbors and of children.

One interesting property, is that each time a node u that was the α-ancestor of another node v is either redirected to a new parent with the Redirect function or promoted through the Promote function, then u no longer is the α-ancestor of v, and may even become its γ-ancestor. This is represented in Figure 4.9 and Figure 4.10.

We will see in Section 4.3.4 that $\mathcal{A}_{cnn}$ maintains a valid structure. We will then show in Section 4.3.5 that updates are done in $O(\log^2 \Phi)$ time, and that the structure takes $O(n)$ memory. We will then show in Section 4.3.6 how to improve the update time to $O(\log \Phi)$ with a simple trick, and in Section 4.4.3 adapt $\mathcal{A}_{cnn}$ to the distributed setting.

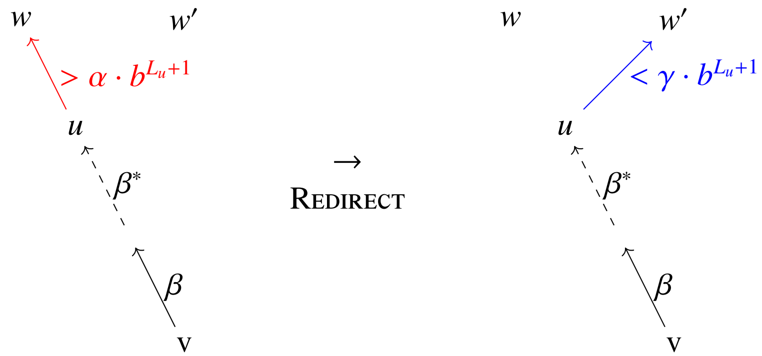


Figure 4.9 – Example of usage of the Redirect function.

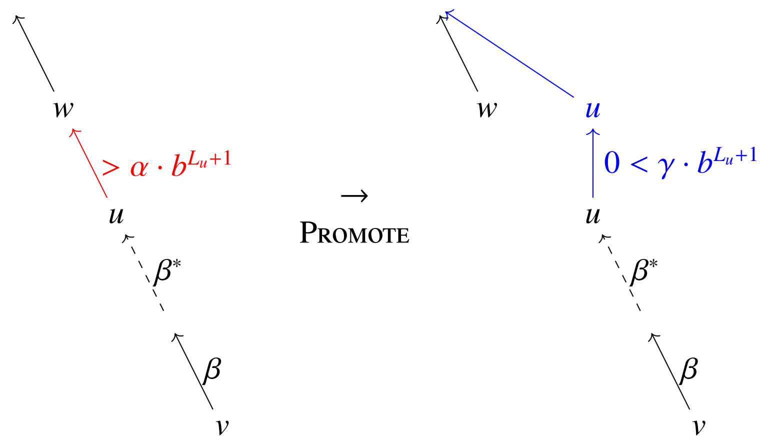


Figure 4.10 – Example of usage of the Promote function.

Algorithm 14 $\mathcal{A}_{cnn}$: certificates initialization and maintenance of the data structure

1: **function** RoundDistance(u, v, k) (*Computes the value associated with the distance to the parent.*)

2: **return** τ where $\tau = \begin{cases} \gamma & \text{if } d(u,v) < \gamma b^{k+1} \\ \beta & \text{if } \gamma b^{k+1} \leq d(u,v) < \beta b^{k+1} \\ \alpha & \text{if } \beta b^{k+1} \leq d(u,v) \end{cases}$

3: Initialization:

4: **for all** node $u \in \mathcal{V}$ **do**

5: $L_u \in \mathbb{N}$ (*largest k s.t. u is the center of a ball of radius b^k, see Algorithm 10*)

6: $f_u \in \mathcal{V}$ (*parent of u, a node $v \neq u$ except for the root, see Algorithm 10*)

7: $N_u = \{(v, k) : d(u, v) < cb^k \wedge 0 < k \leq \min(L_u, L_v)\}$ (*$(v, k) \in N_u$ if u and v are neighbors at level k*)

8: $C_u = \{(v, k) : f_v = u \wedge L_v = k\}$ (*$(v, k) \in C_u$ if the level of v is k and u is the parent of v*)

9: **let** $PN_u = \{(v, k) : k > 0 \wedge \exists (w, k+1) \in N_u \cup N_{f_u}, ((v, k) \in C_w \vee (v, k) = (w, k)) \wedge d(u, v) \geq cb^k\}$ (*Set of cousins. Note that this set is used only at initialization.*)

10: $Cert_u \leftarrow \{\texttt{Separation}_{\gamma,k}(u, v), \texttt{ShortEdge}_{c,k}(u, v) : (v, k) \in N_u \setminus \{(u, *)\}\} \cup \{\texttt{Long-Edge}_{c,k}(u, v) : (v, k) \in PN_u\}$

11: $\tau \leftarrow$ RoundDistance(u, f_u, L_u); $Cert_u \leftarrow Cert_u \cup \{\texttt{Cover}_{\tau,L_u}, \texttt{SCover}_{\tau-1/b,L_u}\}$

12: **end for**

13: $Cert \leftarrow \bigcup_{u \in \mathcal{V}} Cert_u$

14: Main loop:

15: **for each** time t_i where a node moves **do**

16: Resolve all false certificates of $Cert$ one by one, according to the instructions of algorithms 15 and 16:

17: first `Separation`, in order of decreasing level,

18: then `ShortEdge` and `LongEdge`, in order of decreasing level,

19: and finally `Cover` and `SCover` certificates, in order of decreasing level.

20: **end for**

Algorithm 15 $\mathcal{A}_{cnn}$: management of certificates of type `Separation`, `LongEdge`, and `ShortEdge`

21: (*Each time neighbors and children are updated, the following should be executed:*)
22: **when** (v, k) is added to/*removed from* N_u **do**
23: **if** $v \neq u$ **then** add/*remove* $\texttt{Separation}_{\gamma,k}(u, v)$, $\texttt{ShortEdge}_{c,k}(u, v)$ to/*from* *Cert* **end if**
24: **if** $v \neq u$ **then** remove/*add* $\texttt{LongEdge}_{c,k}(u, v)$ from/*to* *Cert* **end if**
25: **for each** $\{u', v'\} : (u', k-1) \in C_u \cup \{u, k-1\}, (v', k-1) \in C_v \cup \{v, k-1\} : u' \neq v'$ **do**
26: add/*remove* $\texttt{LongEdge}_{c,k-1}(u', v')$ to/*from* *Cert* (*add if $(v', k-1) \notin N_{u'} \setminus \{(u', *)\}$, remove if certificate exists*)
27: **end for**
28: **end when**
29: **when** (v, k) is added to/*removed from* C_u **do**
30: **for each** $v' : \exists(w, k+1) \in N_u, (v', k) \in C_w \cup \{w, k\} : v \neq v'$ **do**
31: add/*remove* $\texttt{LongEdge}_{c,k}(v, v')$ to/*from* *Cert* (*add if $(v', k) \notin N_v \setminus \{(v, *)\}$, remove if exists*)
32: **end for**
33: **end when**

34: (*For simplicity, in the following, the certificate that fails is implicitly removed, as well as its symmetric. Also, each time a certificate is said to be created at level 0, except $\texttt{Cover}_{\gamma,0}$, it should not be created.*)

35: **when** $\texttt{Separation}_{\gamma,L}(u, v)$ fails with $L_u \leq L_v$ and $u < v$ **do** (*$L = L_u$. Note that the node with lowest order will be the one to reduce its level.*)
36: $PF \leftarrow \{w \neq u : (w, L) \in N_u\}$ (*Potential parent for level $L-1$ children of u*)
37: **for each** $w : (w, L-1) \in C_u$ **do** (*$L_w = L - 1$ and $f_w = u$*)
38: **if** $\exists w' \in PF : d(w, w') < \gamma b^L$ **then** Redirect(w, u, w')
39: **else** Promote(w, u); $PF \leftarrow PF \cup \{w\}$
40: **end for**
41: remove $\texttt{Cover}_{*,*}(u, f_u)$, $\texttt{SCover}_{*,*}(u, f_u)$ from *Cert*; $C_{f_u} \leftarrow C_{f_u} \setminus \{(u, L)\}$
42: **for each** $w : (w, L) \in N_u$ **do** $N_w \leftarrow N_w \setminus \{(u, L)\}$ **end for**
43: $N_u \leftarrow N_u \setminus \{(w, L) \in N_u\}$
44: $L_u \leftarrow L_u - 1$; $f_u \leftarrow v$; $C_v \leftarrow C_v \cup (u, L_u)$ add $\texttt{Cover}_{\gamma,L_u}(u, v)$ to *Cert*
45: remove $\texttt{Separation}_{\gamma,L}(u, *)$, $\texttt{Separation}_{\gamma,L}(*, u)$, $\texttt{LongEdge}_{c,L}(u, *)$, $\texttt{LongEdge}_{c,L}(*, u)$
46: **end when**
47: **when** $\texttt{ShortEdge}_{c,L}(u, v)$ fails with $L_u \leq L_v$ **do** (*$L = L_u$*)
48: $N_u \leftarrow N_u \setminus (v, L)$; $N_v \leftarrow N_v \setminus (u, L)$;
49: **end when**
50: **when** $\texttt{LongEdge}_{c,L}(u, v)$ fails with $L_u \leq L_v$ **do** (*$L = L_u$*)
51: $N_u \leftarrow N_u \cup (v, L)$; $N_v \leftarrow N_v \cup (u, L)$
52: **end when**

Algorithm 16 $\mathcal{A}_{cnn}$: management of certificates of type `Cover` and `SCover`

53: **when** $\texttt{SCover}_{\tau,L}(u,v)$ fails with $\tau \in \{\gamma,\beta\}$ **do** (*$L = L_u$ and $v = f_u$*)
54: **if** $\tau = \beta$ **then** replace $\texttt{SCover}_{\beta,L}(u,v)$ by $\texttt{SCover}_{\gamma,L}(u,v)$ in $Cert$ **end if**
55: replace $\texttt{Cover}_{\tau+1/b,L}(u,v)$ by $\texttt{Cover}_{\tau,L}(u,v)$ in $Cert$
56: **end when**
57: **when** $\texttt{Cover}_{\tau,L}(u,v)$ fails with $\tau \in \{\gamma,\beta\}$ and $L \geq 1$ **do** (*$L = L_u$ and $v = f_u$*)
58: **if** $L_{\alpha a(u)} < L_{\gamma a(u)}$ **then**
59: **if** $\exists (f', L_{\alpha a(u)} + 1) \in N_{f_{\alpha a(u)}} : d(\alpha a(u), f') \leq \gamma b^{L_{\alpha a(u)}+1}$ **then**
60: $\textsc{Redirect}(\alpha a(u), f_{\alpha a(u)}, f')$ **else** $\textsc{Promote}(\alpha a(u), f_{\alpha a(u)})$ **end if**
61: **if** $\tau = \beta$ **then**
62: **if** $\exists v' : (v', L+1) \in N_v \wedge d(u,v') \leq \gamma b^{L+1}$ **then**
63: $\textsc{Redirect}(u,v,v')$ **else** $\textsc{Promote}(u,v)$ **end if**
64: **else**
65: replace $\texttt{Cover}_{\tau,L}(u,v)$ by $\texttt{Cover}_{\tau+1/b,L}(u,v)$ in $Cert$
66: replace $\texttt{SCover}_{\tau-1/b,L}(u,v)$ by $\texttt{SCover}_{\tau,L}(u,v)$ in $Cert$ (*replace if it exists, add otherwise*)
67: **end when**
68: **when** $\texttt{Cover}_{\alpha,L}(u,v)$ fails or $\texttt{Cover}_{\gamma,L=0}(u,v)$ fails **do**
69: **if** $\exists v' : (v', L+1) \in N_v \wedge d(u,v') \leq \gamma b^{L+1}$ **then**
70: $\textsc{Redirect}(u,v,v')$ **else** $\textsc{Promote}(u,v)$ **end if**
71: **end when**
72: **procedure** $\textsc{Redirect}(v, oldf, newf)$ (*Change v's parent to $newf$*)
73: $C_{oldf} \leftarrow C_{oldf} \setminus \{(v, L_v)\}$; remove $\texttt{Cover}_{*,*}(v, oldf)$, $\texttt{SCover}_{*,*}(v, oldf)$ from $Cert$;
74: $f_v \leftarrow newf$; $C_{newf} \leftarrow C_{newf} \cup (v, L_v)$; add $\texttt{Cover}_{\gamma,L_v}(v, newf)$ to $Cert$
75: **procedure** $\textsc{Promote}(v, oldf)$ (*Increase L_v once*)
76: **if** $L_{oldf} \geq L_v + 2$ **then** $newf \leftarrow oldf$ **else** $newf \leftarrow f_{oldf}$; **end if**
77: **if** $L_v \neq 0$ **then**
78: **if** $L_{\alpha a(oldf)} < L_{\gamma a(oldf)}$ **then** $\tau' \leftarrow \beta$ **else** $\tau' \leftarrow \textsc{RoundDistance}(v, newf, L_v + 1)$
79: **else**
80: $\tau' \leftarrow \textsc{RoundDistance}(oldf, newf, L_{oldf})$ (*May lead to avoidable updates*)
81: $C_{oldf} \leftarrow C_{oldf} \setminus \{(v, L_v)\}$; remove $\texttt{Cover}_{*,L_v}(v, oldf)$; add $\texttt{Cover}_{\tau',L_v+1}(v, newf)$ in $Cert$
82: remove $\texttt{SCover}_{*,L_v}(v, oldf)$; add $\texttt{SCover}_{\tau'-1/b,L_v+1}(v, newf)$ in $Cert$ (*The new certificate may fail immediately*)
83: $L_v \leftarrow L_v + 1$; $f_v \leftarrow newf$
84: $W \leftarrow \{(w, L_v) \in C_{w'} \cup \{(f_v, L_v)\} : (w', L_v + 1) \in N_{f_v} \wedge d(v,w) < cb^{L_v}\}$ (*$(f_v, L_v) \notin W$ if $d(v, f_v) \geq cb^{L_v}$*)
85: $N_v \leftarrow N_v \cup W \cup \{(v, L_v)\}$
86: **for each** $(w, L_v) \in W$ **do** $N_w \leftarrow N_w \cup \{(v, L_v)\}$ **end for**
87: $C_{newf} \leftarrow C_{newf} \cup (v, L_v)$
88: **if** $newf$ is the root and $L_{newf} = L_v$ **then** increment L_{newf} **end if**

4.3.4 Validity of the Algorithm in the Low Mobility Setting

In this section, we will prove that $\mathcal{A}_{cnn}$ maintains a valid structure when nodes respect the movement constraints in the low mobility setting. In other words, we suppose in this section, that discrete instants t_i may be identified, so that at t_i, only one node may move (of maximum $d_{mv} = 1$ distance units) and invalidate the structure. No other node may move until instant t_{i+1}.

Let us denote by $G = \{(u, f_u, L_u, N_u, C_u) : u \in \mathcal{V}\}$ the structure resulting from $\mathcal{A}_{cnn}$. We define its validity as follows.

Definition 4.24 (Valid Structure)**.** *The structure G is valid if it has the following properties:*

- *unique root:* $\exists! v \in \mathcal{V} : f_v = v$ *(Definition 4.7),*
- *γ-separation:* $\forall u, v \in \mathcal{V}, \forall k \in \mathbb{N}^* : k \leq \min(L_u, L_v) \implies d(u, v) \geq \gamma \cdot b^k$ *(Definition 4.7),*
- *correctness of neighbors: for each node u,* $N_u = \{(v, k) \in \mathcal{V} \times \mathbb{N} : 1 \leq k \leq \min(L_u, L_v) \wedge d(u, v) < c \cdot b^k\}$ *(Definition 4.8),*
- *α-coverage:* $\forall u \in \mathcal{V} : f_u = v \neq u \implies L_u < L_v \wedge d(u, v) < \alpha b^{L_u+1}$ *(Definition 4.7),*
- *ancestor invariant:* $\forall u \in \mathcal{V},\ \beta \cdot b^{L_u+1} \leq d(u, f_u) < \alpha \cdot b^{L_u+1} \implies L_{\gamma a(u)} < L_{\alpha a(u)}$ *(Definition 4.23),*
- *correctness of children, that is, for each node u,* $C_u = \{(v, k) : v \in \mathcal{V} \backslash \{u\} \wedge f_v = u \wedge L_v = k\}$ *(Definition 4.13).*

Similarly to the previous chapter, we denote by $\mathcal{G}(t)$ the set of possible values for G that are valid with regards to the positions the nodes have at time t. Our theorem is formulated in a similar way as Theorem 3.7 from the previous chapter.

Theorem 4.25. *In the low mobility setting, assuming that the structure is correctly initialized, that is,* $G(0) \in \mathcal{G}(0)$*, and that all certificates of Table 4.3 are in Cert at time* 0*, then when using* $\mathcal{A}_{cnn}$*,* $\forall i \geq 0, G(t_{i+1}) \in \mathcal{G}(t_i)$*.*

Proof. The proof of this theorem uses the lemmas below, we give here the general outline of the proof.

The proof is by induction on i. We prove that, given $G(t_i) \in \mathcal{G}(t_{i-1})$ and a corresponding set $Cert(t_i)$ of certificates as described on Table 4.3, the result of the instructions of $\mathcal{A}_{cnn}$ applied after the movement of time t_i result in a structure, $G(t_{i+1})$, that is valid with regards to these new positions, in other words, $G(t_{i+1}) \in \mathcal{G}(t_i)$.

To prove that between t_i and t_{i+1} the structure is corrected, we prove that the three phases of $\mathcal{A}_{cnn}$ (lines 17, 18 and 19 of Algorithm 14) each add properties from Definition 4.24, needed for the validity of G, without invalidating the properties from the previous phases. This is done in lemmas 4.30, 4.32, and 4.35. We then note that the certificates are correctly added to and removed from *Cert*, proving in Lemma 4.36 that the list of certificates is also correctly updated at time t_{i+1}. □

In order to prove this step by step, we will use an additional definition, that of structure coherency, a weaker version of structure validity (note that it just as validity minus all properties depending on distances). This property should be true at the start of any update, as it ensures that the variables of the structure are always coherent with one another.

Definition 4.26 (Coherent Structure)**.** *The structure G is coherent if it has the following properties:*

- *unique root,*
- $\forall u \in \mathcal{V}, f_u \neq u \implies L_{f_u} > L_u$,
- $\forall u \in \mathcal{V}, \forall (v,k) \in N_u, k \leq \min(L_u, L_v)$.
- *correctness of children.*

We will suppose for the rest of this section, that $G(t_i) \in \mathcal{G}(t_{i-1})$, and that the set of certificates in $Cert_{t_i}$ corresponds to the list of certificates given on Table 4.3, according to the positions of the nodes at time t_{i-1}.

Throughout our proof, we will use the following, that shows that even after a movement, if a node u gets its level increased to a level ℓ and takes its ancestor of level $\ell+1$ as a new parent, then this ancestor can be used to reconstruct u's new neighbors using Lemma 4.10.

Lemma 4.27. *At any instant $t \in [t_i; t_{i+1}[$, let $u \in \mathcal{V}$ and $\ell = L_u(t) \geq 1$. Let v be a node such that $d(u,v,t) < c \cdot b^\ell$, and let v' be the node that is, at time t, the ancestor of v at level $\ell+1$ (which may be v itself if $L_v \geq \ell$). We have:*

$$d(f_u(t), v', t) < c \cdot b^{\ell+1}$$
$$d(f_u(t), v', t_{i-1}) < c \cdot b^{\ell+1}$$

Proof. Note that a node w can decrease its level only on line 44, when a $\texttt{Separation}_{\gamma,L}(w,w')$ certificate fails, in which case the new parent for w is w', so that $d(w,w',t_{i-1}) < \gamma \cdot b^L + d_{mv} < \alpha \cdot b^L$. Also, a node w can increase its level only on line 83, in which case the new parent for w was, at time t_i, the ancestor at level $\ell+1$ in $G(t_i)$.

Thus, as $G(t_i) \in \mathcal{G}(t_{i-1})$, the structure respects the α-coverage with respect to the distances between the nodes at time t_{i-1}, and we may apply Lemma 4.12 (knowing that $f_u(t)$ is the ancestor of u at level $\ell+1$):

$$d(u, f_u(t), t_{i-1}) < \left(\alpha + \frac{1}{b}\right) b^{\ell+1} - 1$$
$$d(v, v', t_{i-1}) < \left(\alpha + \frac{1}{b}\right) b^{\ell+1} - 1$$

By triangular inequality, we have $d(f_u(t), v', t) \leq c \cdot b^\ell + d(u, f_u(t), t) + d(v, v', t)$, and as only one node may have moved at time t_i, we have:

$$d(f_u(t), v', t) < c \cdot b^\ell + 2\left(\alpha + \frac{1}{b}\right) b^{\ell+1} - 2 + d_{mv} \tag{4.2}$$

Note that:

$$\begin{aligned}
& 2\frac{b-1}{b-1} \leq c && \text{as } c = 2 \\
\implies & \frac{2\alpha b + 2}{b-1} \leq c && \text{as } \alpha = \frac{b-2}{b} \\
\implies & c \cdot b^\ell + 2\alpha b^{\ell+1} + 2b^\ell \leq c \cdot b^{\ell+1} \\
\implies & c \cdot b^\ell + 2\left(\alpha + \frac{1}{b}\right) b^{\ell+1} - 2 + d_{mv} \leq c \cdot b^{\ell+1} - d_{mv} && \text{as } d_{mv} \leq 1
\end{aligned}$$

Combining this inequality with that from Equation 4.2, and recalling that only one node may have moved at time t_i, we get our result. ◻

Before analyzing the three steps of the algorithm and of our proof, let us note the following:

Remark 4.28. *At instant t_i, before any update is executed, $G(t_i)$ is coherent.*

Proof. This is trivial: as long as no change is made to the structure, any property that does not depend on the distance between the nodes, such as the uniqueness of the root and the level relations between nodes, remains true. □

First phase: Separation certificates

As can be seen line 17 of Algorithm 14, all certificates of type `Separation` are resolved first. We define the first phase as the execution of line 17, that is, the correction of all `Separation` certificates until no invalid certificate of that type is left in $Cert$.

We will first make the following remark:

Remark 4.29. *When a node v is promoted during the first phase with function* Promote*, this promotion never breaks the γ-separation property, that is, with $L_v = k$ before the call of* Promote*, $\forall w \in \mathcal{V} : L_w \geq k$, $d(v, w) \geq \gamma \cdot b^{k+1}$.*

Proof. In the first phase, Promote can be called only on line 39. By the conditions from lines 36 to 40 (noting that $w \in C_u \iff u = f_w$ by Remark 4.28), we need to prove that any node u such that $L_u \geq L_v + 1$ and $d(u, v, t_i) < \gamma b^{L_v+1}$ satisfies $(u, L_v + 1) \in N_{f_v}$, except for previously promoted nodes. We have that such a node u sees $d(u, v, t_{i-1}) < \gamma b^{L_v+1} + d_{mv} < c \cdot b^{L_v+1}$, thus $(u, L_v + 1) \in N_{f_v}$ as it is supposed that $G(t_i) \in \mathcal{G}(t_{i-1})$. □

We will now prove that the first phase restores the γ-separation:

Lemma 4.30. *After the first phase, that is, once all certificates of type* `Separation` *are resolved between t_i and t_{i+1}, the structure validates following properties:*

- *unique root,*
- *γ-separation.*

In addition, G is coherent.

Proof. First, note that the only nodes that may decrease their level, on line 44, are nodes u that fail a `Separation` certificate with another node v on level L_u (by the conditions of line 35, and by Lemma 4.19). In addition, the only nodes that may increase their level, through the function Promote on line 39, are u's children.

By Remark 4.28, there is a unique root before the correction of `Separation` certificates, without any other node at its maximal level, thus its level is not decreased. It follows that the uniqueness of the root is maintained after correction of `Separation` certificates.

Let u and v be two nodes that violate the γ-separation at t_i, that is, such that there is a level $0 < k \leq \min(L_u(t_i), L_v(t_i))$ with $d(u, v, t_i) < \gamma \cdot b^k$. As only one node may move at t_i, but not more than d_{mv} distance units, we have $d(u, v, t_{i-1}) < \gamma \cdot b^k + d_{mv} < c \cdot b^k$ (as $d_{mv} = 1$). Thus, because $G(t_i) \in \mathcal{G}(t_{i-1})$, we have that u and v are neighbors at t_i, so that both $\texttt{Separation}_{\gamma,k}(u, v)$ and $\texttt{Separation}_{\gamma,k}(v, u)$ are in $Cert(t_i)$ and are failing.

Let $k = \min(L_u(t_i), L_v(t_i))$. As $d(u, v, t_{i-1}) \geq \gamma \cdot b^k$ (because $G(t_i) \in \mathcal{G}(t_{i-1})$), and because of the limited movement, we also have $d(u, v, t_i) \geq \gamma \cdot b^k - d_{mv} \geq \gamma \cdot b^k - 1$. Thus, if $k \geq 1$, we have $d(u, v, t_i) \geq \gamma \cdot b^{k-1}$. In other words, u and v may violate the γ-separation only at level $\min(L_u(t_i), L_v(t_i))$.

When executing line 44 of Algorithm 15, for the first certificate that is treated among $\texttt{Separation}_{\gamma,k}(u, v)$ and $\texttt{Separation}_{\gamma,k}(v, u)$, either u or v decreases its maximum level by 1, restoring the γ-separation property. As explained on line 34, as the two certificates are symmetric the one that is not treated first is removed from $Cert$.

Note that as we proved that having $G(t_i) \in \mathcal{G}(t_{i-1})$ and that the γ-separation fails implies that $\texttt{Separation}_{\gamma,k}(u, v)$ and $\texttt{Separation}_{\gamma,k}(v, u)$ fail, by contraposition, we have that if none of the certificates fails, then the γ-separation property is still verified.

Also, by Remark 4.29, no promotion induces a failure of the γ-separation property, so that no error is introduced in the structure either, finally proving that the γ-separation property is verified at the end of the first phase.

Finally, one can check that neighbors, children, and levels are updated properly on lines 41 through 45, and in the functions Redirect and Promote so that the structure is coherent after finishing treating the `Separation` certificates. □

Second phase: Long- and ShortEdge certificates

As can be seen on line 18 of Algorithm 14, after correcting certificates of type `Separation`, the certificates of type `LongEdge` and `ShortEdge` are treated. Similarly to the previous phase, we define the second phase as the execution of line 18, that is, the correction of all `ShortEdge` and `LongEdge` certificates until no invalid certificate of those types is left in $Cert$.

Let us first make the following remark:

Remark 4.31. *Let us call t' the instant where the first phase finishes.*

If Promote(v, u) *is called on time $t \in [t_i; t'[$ with $L_v(t) = k$, then, with $newf$ the node that becomes v's new parent on line 83, the ball $B_{newf}\left(b^{k+2}\right)$ already existed at time t_i; in other words, we have $L_{newf}(t_i) \geq k + 2$.*

Proof. To prove this result, we will prove that $newf$ cannot have decreased its level below level $k + 2$ between t_i and t.

During the first phase, Promote(v, u) is called only on children of a node $u = f_v(t)$ that fails a $\texttt{Separation}_{\gamma,k+1}(u, u')$ certificate with another node u' on level $k+1$, with $L_u(t) \leq L_{u'}(t)$. Thus, by Lemma 4.19 $L_u(t) = k + 1$, so that by line 76, $newf = f_u(t)$. As only one node may move at each time step, either u or u' moved at time t_i.

- If u moved, either $f_u(t) = f_u(t_i)$, with $L_{f_u(t_i)}(t_i) \geq k + 2$, or u changed its parent on line 44 because of a failure of $\texttt{Separation}_{\gamma,k+2}(u, w)$ certificate on level $k+2$ with a node w with $L_w(t_i) \geq k+2$, so that $f_u(t) = w$ (because w can fail a `Separation` certificate only with u). The node u can only have changed its parent on line 44, as the only other places where u could have changes its parent are in functions Redirect and Promote, which would imply that the parent of u failed a `Separation` certificate with another node than u, which is impossible as only u moved. Thus, if u moved, $L_{newf}(t_i) \geq k + 2$.

- If u' moved, we have that $f_u(t) = f_u(t_i)$ because all failing certificates must involve u', so that u may not have changed its parent yet at time t. As $L_{f_u(t_i)}(t_i) \geq k+2$ by Definition 4.7 and because $G(t_i) \in \mathcal{G}(t_{i-1})$, we get $L_{newf}(t_i) \geq k + 2$.

□

Lemma 4.32. *Let us denote by t'' the instant when the second phase ends, that is, the instant when all certificates of type* `LongEdge` *and* `ShortEdge` *are resolved between t_i and t_{i+1}. At time t'', the structure validates following properties:*

- *unique root,*
- *γ-separation,*
- *correctness of neighbors.*

In addition, $G(t'')$ is coherent, and $Cert(t'')$ does not contain any invalid `Separation` *certificate.*

Proof. As failures of `LongEdge` and `ShortEdge` do not involve changes in maximal levels, by Lemma 4.30, the root remains unique, the γ-separation remains true, and the structure remains coherent. Similarly, no `Separation` certificate is violated or created.

Let us denote by t' the instant where $\mathcal{A}_{cnn}$ starts executing the instructions for correcting `LongEdge` and `ShortEdge` certificates (we thus have $t_i < t' < t''$). Let u and v be two nodes with $k = \min(L_u(t''), L_v(t''))$, such that $d(u, v, t_i) < c \cdot b^k$. Let us prove that $(v, k) \in N_u(t'')$ (which, by Lemma 4.19 proves the neighbors correctness).

If we have $(v, k) \in N_u(t_i)$, then no `LongEdge` or `ShortEdge` certificate involving v and u is violated, so that $(v, k) \in N_u(t'')$.

Let us then look at the other case, that is, $(v, k) \notin N_u(t_i)$. As $G(t_i) \in \mathcal{G}(t_{i-1})$, and because only one node may move of d_{mv} distance units, we have $d(u, v, t_{i-1}) < c \cdot b^k + d_{mv} \leq (c + \frac{2}{b})b^k$ for $k \geq 1$.

- If $k \leq \min(L_u(t_i), L_v(t_i))$, by Lemma 4.18, we have $v \in PN_u(t_i)$, so both $\texttt{LongEdge}_{c,k}(u, v)$ and $\texttt{LongEdge}_{c,k}(v, u)$ are in $Cert(t_i)$. We need to prove that they are still in $Cert(t')$. During the first phase, these certificates may have been removed from $Cert$ only on four places.
  - Line 24, if u and v were added to each other's set of neighbors, which, during the first phase, may have happened only when a node was promoted, on level $\min(L_u(t_i), L_v(t_i)) + 1 > k$.
  - Line 26, if u and v's parents were removed from each other's neighborhood, which may have happened only if u and v failed a `Separation` certificate at level k, which is impossible because of the speed limit, and because $(v, k) \notin N_u(t_i)$.
  - Line 31, if u and v were no longer potential neighbors because one of them changed its parent. During the first phase, this may have happened only on line 41, if one of them failed a separation certificate, which we just showed is not possible, or on line 73 or 81 if one of their parents failed a separation certificate. In these last two cases, as the lists of children and of neighbors are recreated on level $k + 1$ on lines 74 and 85, the lines 26 and 31 are executed, which adds the $\texttt{LongEdge}_{c,k}(u, v)$ and $\texttt{LongEdge}_{c,k}(v, u)$ certificates back to $Cert$.
  - Line 45, if u and v failed a `Separation` certificate at level k, which we already proved is impossible.

  Thus, if $k \leq \min(L_u(t_i), L_v(t_i))$, lines 50 through 52 are executed and v is added to N_u, so that $(v, k) \in N_u(t'')$.

- Finally, in the case where $k > \min(L_u(t_i), L_v(t_i))$, it means u or v (or both) have increased their level since time t_i (which may have happened only on line 83, in a call of Promote following a failure of `Separation` certificate). Thus, we have $\texttt{LongEdge}_{c,k}(u, v) \notin Cert(t_i)$, so that we need to prove that a $\texttt{LongEdge}_{c,k}(u, v)$ certificate has been added to

the structure during the first phase. As the level change of u and/or v necessarily implies that u and/or v are added to their new parent's children, lines 30 through 32 are executed. By Lemma 4.27, at that moment, f_u and f_v are neighbors, thus line 31 is executed for u and v, adding a $\texttt{LongEdge}_{c,k}(u, v)$ certificate to the certificate list.

It is simpler to prove that if u and v satisfy $d(u, v, t_i) \geq c \cdot b^k$ implies $v \notin N_u(t'')$. If $v \notin N_u(t_i)$, then, again, no `LongEdge` or `ShortEdge` certificate is violated, so that we get the result. If $v \in N_u(t_i)$, we have that $\texttt{ShortEdge}_{c,k}(u, v)$ and $\texttt{ShortEdge}_{c,k}(v, u)$ are in $Cert(t_i)$ and are failing. They are still in $Cert(t')$, because they can be removed only on line 23, when they are removed from each other's neighborhood, which would mean that $v \notin N_u(t'')$ anyway. Thus, lines 47 though 49 are executed for one of the two certificates, and v is removed from u's neighbors, so that $v \notin N_u(t'')$. □

Third phase: Cover and SCover certificates

Again, we define the third phase as the execution of line 19, that is, the correction of all `Cover` and `SCover` certificates until no certificate of those types is left in $Cert$.

Remark 4.33. *Every time a node at level ℓ changes parent with the* Redirect *function, the new parent is always at distance smaller than $\gamma \cdot b^{\ell+1}$ from it.*

Proof. Every time the function Redirect is called (on line 60, 63, or 70) for a node u with $L_u = \ell$, it is under the condition that a node v was found such that $d(u, v) < \gamma \cdot b^{\ell+1}$. □

Remark 4.34. *If the ancestor invariant is valid, then the ancestor invariant remains valid after promoting a node u of level k such that $\beta \cdot b^{k+1} \leq d(u, f_u) < \alpha \cdot b^{k+1}$.*

Proof. Let us call $oldf$ the parent of u before promotion, and $newf = f_{oldf}$ the parent after promotion. As the ancestor invariant is valid, there are two possibilities for the distance between $oldf$ and $newf$:

- $d(oldf, newf) < \gamma \cdot b^{k+2}$. We have $d(u, newf) < \gamma \cdot b^{k+2} + \alpha \cdot b^{k+1} < \beta \cdot b^{k+2}$, and thus the ancestor invariant remains true.

- $\gamma \cdot b^{k+2} \geq d(oldf, newf) < \beta \cdot b^{k+2}$. Thus $oldf$ was neither a α-ancestor nor a γ-ancestor for u before promotion. As the ancestor invariant was supposed to be true before the promotion, the two ancestors are above $newf$'s level, so that the invariant remains true after promotion. □

Lemma 4.35. *After the third phase, that is, at time t_{i+1} once all certificates of type* `Cover` *and* `SCover` *are resolved between t_i and t_{i+1}, we have $G(t_{i+1}) \in \mathcal{G}(t_i)$.*

Proof. Let us look at the properties for $G(t_{i+1})$'s validity one by one. We will denote by t' the instant where $\mathcal{A}_{cnn}$ finishes the second phase, that is, the moment just before the instructions for correcting `Cover` and `SCover` are executed (with t_{i+1} being the end of the third phase).

Unique root. We need to prove that when a node u that is not the root is assigned a new parent during the third phase, this new parent is never u itself. When treating `Cover` and `SCov-er` certificates, the only changes in parents can happen on line 74, in the function Redirect and on line 83, in the function Promote.

In the first case (assignment of a new parent in function Redirect), f_u is given the value of $newf$, which can come from two places: either line 60, where u is the α-ancestor of another node, or line 70, where u just failed a $\texttt{Cover}_{\alpha,L}$ certificate. In both cases, $newf$ is a neighbor of f_u. As the structure is coherent, $u \notin N_{f_u}(t_i)$, so that $f_u(t_{i+1}) \neq u$.

In the second case (assignment of a new parent in function PROMOTE), the new value for f_u is either f_u itself if $L_{f_u} \geq L_u + 2$, or f_{f_u} (line 76). Thus, as the structure is coherent, $f_u(t_{i+1}) \neq u$.

Correctness of neighbors. Any increase in level of a node v (on line 83) is associated with the addition of neighbors at v's new level on line 85. Those neighbors are, by line 84, the children of the neighbors of v's new parent, which by Lemma 4.27 include all required neighbors. Thus the correctness of neighbors is kept true during the third phase.

γ-separation. During the third phase, changes of level can happen only on line 83, during a call of function PROMOTE. Each call of PROMOTE (lines 60, 63, and 70) is preceded by a verification ascertaining that no neighbor of the parent of the node to be promoted is too close from it.

Let t be an instant where a call to function PROMOTE is made. It remains to be proven that a node v that would be too close to promote u without violating the γ-separation, that is, such that $d(u, v, t) < \gamma \cdot b^{L_u(t)+1}$, is necessarily a neighbor of $f_u(t)$ at time t. As $f_u(t)$ was an ancestor of u at time t_i, we have, by Lemma 4.12, that $d(u, f_u(t), t) < (\alpha + \frac{1}{b})b^{L_u+1} - 1 + d_{mv}$. By triangular inequality, with our values of γ, α, b and c (see p. 125), we get that $d(v, f_u(t), t) < c \cdot b^{L_u(t)+1}$, meaning that v and $f_u(t)$ are indeed neighbors at time t, by correctness of neighbors.

Thus, the γ-separation is kept true during the third phase, and no `Separation` certificate is created that is immediately false.

α-coverage. Knowing that certificates are treated in order of decreasing levels, we will prove by induction on each level k that this property is corrected during the third phase.

First, for the induction basis, let us prove that at time t', at the level of the root, the α-coverage property is true. Any increase in the level of a node that may have been operated previously goes through the PROMOTE function, which ends at line 88, ensuring that the root is the only node at the highest level. Thus, at the highest level of the constrained navigating net, the α-coverage property is trivially satisfied, as by definition, with u the root, $f_u = u$.

Let us now look at a level $k \neq 0$ that is not the highest level, and such that for any node of level $k' > k$ the α-coverage property is true, and prove that after executing the operations associated with the failure of certificates at level k, the α-coverage is true at level k. Let us call t the moment where the algorithm starts treating the certificates of level k. By definition of the α-coverage (Definition 4.7), any node u of level $L_u(t) = k$ that does not satisfy the α-coverage either is such that $L_{f_u(t)}(t) \leq L_u(t)$, or is too far away from $f_u(t)$.

- Let us prove that the first case is impossible. As $G(t')$ is coherent, we have, for any non-root node $v \in \mathcal{V}$, that $L_{f_v(t')}(t') > L_v(t')$, and that, for any $(w, k+1) \in N_{f_v(t')}(t')$, $L_w(t') > L_v(t')$. Thus, when a node v changes its parent either in function REDIRECT, the new parent always satisfies $L_{f_v} > L_v$. If $k + 1$ is not the level of the root, we have, by coherency of the structure, that $L_{f_{f_v}}(t') > L_v(t') + 1$, and thus, the PROMOTE function preserves that $L_{newf} > L_v$, with $newf$ the new parent for u. If $k + 1$ is the level of the root, in which case $f_v(t)$ is the root, as we proved that the highest level contains only the root, then line 88 is executed, that also guarantees that $L_{newf} > L_v$.

- Let us now prove that if there is a node u with $L_u(t) = k$ such that $d(u, f_u(t), t) \geq \alpha \cdot b^{k+1}$, then a failing $\texttt{Cover}_{\alpha,k}(u, f_u(t))$ certificate will be treated during this third phase. By Remark 4.33, and as a parent can change only in functions REDIRECT and PROMOTE, if $d(u, f_u(t), t) \geq \alpha \cdot b^{k+1}$, then either $f_u(t) = f_u(t_i)$ and u got farther away from its parent, or u got promoted previously with the PROMOTE function. For the first case, we have $d(u, f_u(t), t_i) \geq \alpha \cdot b^{k+1} - d_{mv} \geq \beta \cdot b^{k+1}$, and thus $\texttt{Cover}_{\alpha,k}(u, f_u(t)) \in Cert(t_i)$. In the second case, when u got promoted, lines 78 and 81 of function PROMOTE ensure that a $\texttt{Cover}_{\tau',L_u(t)}(u, f_u(t))$ certificate was created with $\tau' = \alpha$ or β, which fails because of the

distance between u and $f_u(t)$. In the case where $\tau' = \beta$, the failing certificate is treated on line 57, adding a $\texttt{Cover}_{\alpha,k}(u, f_u(t))$ certificate on line 65.

For any level ℓ, when a failing $\texttt{Cover}_{\alpha,\ell}(u, f_u(t))$ certificate is treated, u either gets through REDIRECT a new parent so as to satisfy, by Remark 4.33, the α-coverage, or u gets promoted. In that last case, if $newf$, the new parent of u is such that $d(u, newf, t) \geq \alpha \cdot b^{L_{newf}}$, then on line 78 a $\texttt{Cover}_{\alpha,\ell+1}(u, newf)$ is created with the help of function ROUNDDISTANCE; the certificate is immediately false, and as $\ell + 1 > k$, it will be treated immediately. This process is repeated until a level ℓ is found such that $d(u, newf, t) < \alpha \cdot b^{\ell}$, possibly increasing the level of the root in the process[11]. Thus u will eventually satisfy the α-coverage.

Finally, we can see that this process does not invalidate the α-coverage for any node of level higher than k. Indeed, any promotion of a node v that would result in invalidating the α-coverage for v, incurs the creation of a failing $\texttt{Cover}_{\alpha,L_{f_v}}(v, f_v)$ property, which would, similarly as above, eventually correct the property.

The procedure is similar at level 0, except for the fact that any node u of that level satisfies $d(u, f_u) < \gamma \cdot b^1$, and if it is not the case, then u is immediately redirected to another parent or promoted on lines 68 through 71. In the case of a promotion, a $\texttt{Cover}_{\tau',1}$ certificate is created, depending on the value of τ' at line 80. If $\tau' = \gamma$, then it is replaced by a $\texttt{Cover}_{\beta,1}$ certificate on line 55, which, as described above will eventually restore the α-coverage as described above.

Thus, by induction, after all Cover certificates have been treated, the structure satisfies the α-coverage property.

Ancestor invariant. As for the α-coverage, we will prove this by induction. Let us call by *toptree at and above k* the subgraph of our navigating net induced by all nodes u such that $L_u \geq k$.

First, for the induction basis, it is easy to see that the toptree at and above the maximal level complies to the ancestor invariant, as it contains only one node, the root.

Let us now consider at a time t during the third phase, a level k that is not the highest level, such that the toptree at and above $k + 1$ complies to the ancestor invariant, and such that $Cert(t)$ contains no failing certificate with a level higher than k. We will prove that after executing the updates associated with failing certificates of level k, that is, once no failing certificates remain at levels k and above, the toptree at and above k complies to the ancestor invariant.

By supposition, at time t, the toptree at and above k can invalidate the ancestor invariant only because some node u of level k is such that $\beta \cdot b^{k+1} \leq d(u, f_u(t), t) < \alpha \cdot b^{k+1}$ and $L_{\alpha a(u)}(t) < L_{\gamma a(u)}(t)$. When comparing $G(t_i)$ and $G(t)$, there are three possible reasons for that distance between u and $f_u(t)$.

- We can have $f_u(t) = f_u(t_i)$ and $\beta \cdot b^{k+1} \leq d(u, f_u, t_i) < \alpha \cdot b^{k+1}$, in other words, the parent of u did not change since t_i, but the distance between u and f_u remains within the same bounds. We will show below that in this case it is impossible to have $L_{\alpha a(u)}(t) < L_{\gamma a(u)}(t)$.

- We can also have $f_u(t) = f_u(t_i)$ but with $\gamma \cdot b^{k+1} \leq d(u, f_u, t_i) < \beta \cdot b^{k+1}$, in other words, u's parent did not change, but the distance between u and f_u increased sufficiently to break the previous upper bound. We will prove below that if this is the case, then a $\texttt{Cover}_{\beta,k}(u, f_u)$ fails, and the operations associated with that failure restore the ancestor invariant.

- Finally, it is possible that $f_u(t) \neq f_u(t_i)$. In other words, u changed its parent since t_i, and got attached to a parent at distance more than $\beta \cdot b^{k+1}$ away from it. We will prove, as for the previous case, that a failing $\texttt{Cover}_{\beta,k}(u, f_u)$ certificate restores the ancestor invariant.

[11]Note that we will see in the next section that a constant number of level increases per node is enough, thanks to the ancestor invariant.

First case: let us suppose that $f_u(t) = f_u(t_i)$, that at time t_i we had $\beta \cdot b^{k+1} \leq d(u, f_u, t_i) < \alpha \cdot b^{k+1}$, and at time t we have $\beta \cdot b^{k+1} \leq d(u, f_u, t) < \alpha \cdot b^{k+1}$, and let us suppose that $L_{\alpha a(u)}(t) < L_{\gamma a(u)}(t)$. As $G(t_i) \in \mathcal{G}(t_{i-1})$, we had $L_{\alpha a(u)}(t_i) > L_{\gamma a(u)}(t_i)$, which means that either the α-ancestor, or the γ-ancestor of u changed. By Definition 4.21 and Definition 4.22, a change of one of those ancestors is caused either by the change of level of an ancestor of u, which can happen only in functions Redirect and Promote, or by a change of distance between one of the ancestors and its parent.

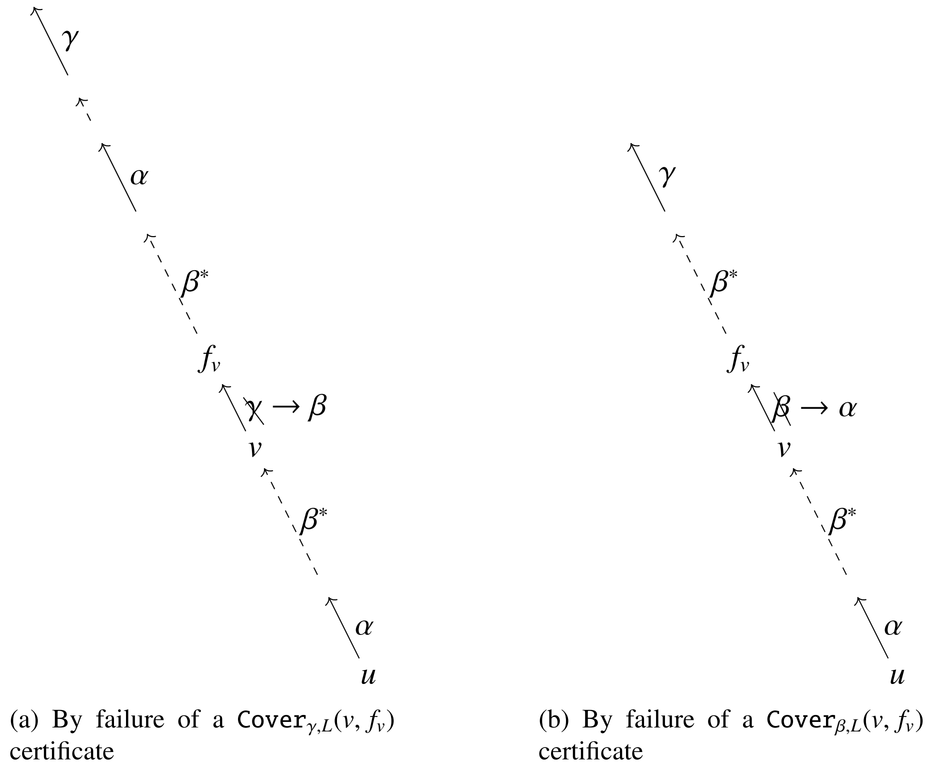
First, let us see that the inequality $L_{\alpha a(u)}(t) < L_{\gamma a(u)}(t)$ cannot be caused by a call to the Redirect function, as by Remark 4.33, a redirection can only increase the level of u's α-ancestor (if Redirect is applied to the α-ancestor, the new α-ancestor is higher up in the hierarchy), and can only decrease the level of u's γ-ancestor (if Redirect is applied to a node below the γ-ancestor, that node becomes the nex γ-ancestor). Concerning the function Promote, a node v that gets promoted from level ℓ to level $\ell + 1$ does not get any new children in the function, so that all its children are at a level below ℓ: after promotion, $\forall (v', \ell') \in C_v, L_{v'} \leq \ell - 1$. It follows that if v, the promoted node, is both an ancestor of u and a descendent of u's γ-ancestor, than v becomes u's new γ-ancestor after promotion (as v satisfies the first condition of Definition 4.21). As any other ancestor of u at a level below i did not change with the call to Promote, this function cannot cause the α-ancestor of u to become below the γ-ancestor. Thus, neither function Redirect nor function Promote applied on any node v can cause a u to have $L_{\alpha a(u)}(t) < L_{\gamma a(u)}(t)$.

Let us now prove that the inequality $L_{\alpha a(u)}(t) < L_{\gamma a(u)}(t)$ cannot be caused by the change of distance between an ancestor v of u and its parent f_v, thanks to $\mathcal{A}_{cnn}$. For this, we can see that only two situations can lead to the inequality, both represented graphically in Figure 4.11, using the same representation as on Figure 4.6[12].

- First, the situation represented on Figure 4.11a, if u's γ-ancestors, let us call it v, gets at a distance higher than $\gamma \cdot b^{L_v+1}$ from f_v, then v may no longer satisfy the conditions of Definition 4.21 (if we also have $L_{f_v} = L_v + 1$), so that the new γ-ancestor of u becomes an ancestor of v, potentially breaking the ancestor invariant. This change of distance, however, goes with the failure of a $\texttt{Cover}_{\gamma,L}(v, f_v)$ certificate, which on line 60 calls either Redirect or Promote on v's α-ancestor, which in this case is also u's α-ancestor. Let us call w that α-ancestor. If Redirect is called, by Remark 4.33, w becomes u and v's new γ-ancestor, restoring the ancestor invariant. In the case Promote is called, as w's level is increased but not the level of its children, w becomes u and v's new γ-ancestor, restoring the ancestor invariant for u. In addition, as it is supposed that the toptree at and above level k satisfies the ancestor invariant, and because by definition of w, we have $\beta \cdot b^{L_w+1} \leq d(w, f_w, t) < \alpha \cdot b^{L_w+1}$, we have by Remark 4.34 that the toptree at and above level $k + 1$ keeps the validity of the ancestor invariant.

- Second, the situation represented on Figure 4.11b, an ancestor v of u below u's γ-ancestor (that is, such that $L_v < L_{\gamma a(u)}$) that is not u's α-ancestor (that is, such that $\gamma \cdot b^{L_v+1} \leq d(v, f_v) < \beta \cdot b^{L_v+1}$ and $L_{f_v} = L_v + 1$) might get at a distance higher than $\beta \cdot b^{L_v+1}$ from f_v, which breaks the ancestor invariant. However, if that is the case, then a $\texttt{Cover}_{\beta,L}(v, f_v)$ certificate fails, which on line 63 calls either the function Redirect or Promote on v. As above, by Remark 4.33 and Remark 4.34, v thus becomes u's new γ-ancestor, restoring the ancestor invariant for u while keeping the property for the toptree at and above level $k + 1$.

[12]Arrows with a solid line represent a parent/child relation, and have labels representing their role with regards to the ancestor invariant: γ if the child can be a γ-ancestor for its descendants, α if it can be a α-ancestor, and β if it can be neither. Arrows with dashed lines represent successive parent/child relations. See p.130 for more details.



(a) By failure of a $\texttt{Cover}_{\gamma,L}(v, f_v)$ certificate

(b) By failure of a $\texttt{Cover}_{\beta,L}(v, f_v)$ certificate

Figure 4.11 – Graphical representation of the movements of a node that could invalidate the ancestor invariant.

This finishes the proof of the first case, as we cannot have $L_{\alpha a(u)}(t) < L_{\gamma a(u)}(t)$.

Second case: let us suppose that u's parent did not change since t_i, but that the distance between u and f_u increased: $f_u(t) = f_u(t_i)$, with $\gamma \cdot b^{k+1} \leq d(u, f_u, t_i) < \beta \cdot b^{k+1}$, and $\beta \cdot b^{k+1} \leq d(u, f_u(t), t) < \alpha \cdot b^{k+1}$. There is a risk that the ancestor invariant fails as represented graphically on Figure 4.12. We have that *Cert* contains a failing $\texttt{Cover}_{\beta,L_u}(u, f_u)$ certificate, which on line 63 calls either the function Redirect or Promote on u's α-ancestor. Again, by Remark 4.33 and Remark 4.34, the ancestor invariant is restored for u, while keeping the property for the toptree at and above level $k + 1$.

By the rules of the failing $\texttt{Cover}_{\beta,L_u}(u, f_u)$ certificate, u is also redirected or promoted on line 63. If $k = L_u(t) > 0$, the condition on line 78 of the Promote function ensures that the ancestor invariant is kept: indeed, if u does not comply to the ancestor invariant from Defini-

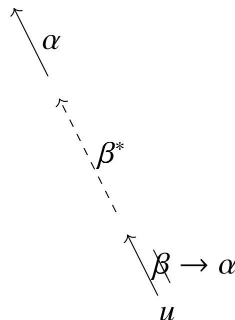


Figure 4.12 – Graphical representation of a problem of ancestor invariant in the second case.

tion 4.23, then a $\texttt{Cover}_{\beta,\ell}(u, newf)$ is created, with ℓ the new level of u and $newf$ its new parent, which fails immediately, and redirecting or promoting again u until the ancestor invariant is true again (we will however later see that a constant number of promotions is enough).

If $k = L_u(t) = 0$ on the other hand line 78 is not executed, and the new `Cover` certificate depends on the distance between $oldf = f_u(t)$ and its new parent $newf = f_{f_u(t)}(t)$. As u does not get any children from the promotion, if $d(u, newf, t) < \beta \cdot b^1$, then the promotion does not have any impact on the ancestor invariant. Remains to be proven that if $d(u, newf, t) \geq \beta \cdot b^2$, then any problem of ancestor invariant will be corrected. We will distinguish the cases according to the distance between $oldf$ and $newf$.

- If $d(oldf, newf, t) < \beta \cdot b^2$, then line 80 initializes τ' either to β, or to γ. In this second case, the failing $\texttt{Cover}_{\gamma,1}(u, newf)$ certificate will be replaced by a β on line 65; thus in both cases, a failing $\texttt{Cover}_{\beta,1}(u, newf)$ certificate will eventually be added to $Cert$, which as its level is higher than 0, will restore the ancestor invariant as proven previously.

- If $d(oldf, newf, t) \geq \beta \cdot b^2$, then, as u and $oldf$ end up with the same parent, the ancestor invariant is true for u if and only if it is true for $oldf$. As we supposed that the toptree at and above level $k + 1 = 1$ complies to the ancestor invariant, the property is true for u.

Third case: let us suppose that $f_u(t) \neq f_u(t_i)$, which means that either function Redirect or Promote has been called on u between time t_i and t, that is, before the third phase. Let us also suppose that $\beta \cdot b^{k+1} \leq d(u, f_u, t)$. By Remark 4.33, the last call to one of the two functions must have been a call to Promote. As seen in the previous case, line 78 ensures that a call of Promote always adds certificates to $Cert$ that end up restoring the ancestor invariant.

We thus have proven that after all certificates of level k have been treated in the third phase, the toptree at and above level k complies to the ancestor invariant. By induction, we get that at the end of the third phase, the whole structure complies to the property.

Correctness of children. It is easy to see that each change of parent is associated with the corresponding updates to the sets of children, ensuring the correctness of children.

□

Certificate list

Lemma 4.36. *Once all failing certificates are resolved between t_i and t_{i+1}, all certificates of Table 4.3 are in Cert.*

Proof. Knowing thanks to Lemma 4.35 that the structure is valid after all failing certificates have been resolved, it remains to see that all needed non-failing certificates are added to $Cert$.

- All certificates of type `Separation` and `ShortEdge`, that depend only on the sets of neighbors, are added to $Cert$ on lines 22 through 28, at each change of neighbor. The `LongEdge` certificates, that depend on the potential neighbors, and thus both on the neighbor and parent/child relations of the nodes, are added to $Cert$ at each change either of neighbors or of children on lines 21 through 33.

- The `Cover` and `SCover` certificates depend on the parent/child relations of the nodes, but must have the right distance parameter. Changes of distances for parent/child pairs that remain parent/child are treated in the associated failing certificates, lines 55, 65, and 66. Changes of parents happen only in the Redirect and Promote functions. In the Redirect, only a `Cover` certificate with a distance threshold of γ is created, which by Remark 4.33 is the needed threshold. In function Promote, certificates are added lines 81 and 82,

with a parameter τ' that depends on previous conditions. If this parameter is not the right parameter for the actual distance between the nodes, then the created certificates are failing, and treated like a change of distance between a preexisting pair of parent/child, which we just showed is eventually corrected. □

Put all together, by induction on the time steps, these lemmas prove Theorem 4.25. Note that none of these proofs involve d_{min} or d_{max}. Indeed, $\mathcal{A}_{cnn}$ is still valid even if there are no limits on the distances between the nodes; we will however see in the next section that those constraints are needed for the performance analysis.

Now that the validity of our algorithm has been settled, we will look at its performance.

4.3.5 Performance Analysis in the Low Mobility Setting

In this section, we prove three results about $\mathcal{A}_{cnn}$'s performance, again in the low mobility setting. First, we prove in Theorem 4.37 that the levels of the nodes do not change drastically: at each time step, a node cannot change its level more than a constant number of times. This property represents the main difference with the DefSpanner from [55], as in that structure, nodes may need to go up all the hierarchy in one time step in order to repair the structure. This paves the way for future results improving the update time in the high mobility setting, where all nodes may move at each time step. We then show in Lemma 4.42 that in the low mobility setting, $\mathcal{A}_{cnn}$ repairs the structure in $O(\log^2 \Phi)$ computations per time step, which, sadly, is worse than for a DefSpanner (see Theorem 4.17). Finally, we will show in Theorem 4.3.5 that the total memory space of a constrained navigating net is linear, as for a DefSpanner.

Level changes

First, we prove that if the order of the nodes is so that the node that moved has a higher order than the nodes of lower levels that didn't move, then at each moment, each node changes its level only a constant number of times, which is stated formally in Theorem 4.37.

Theorem 4.37. *If u is the only node that moved at t_i, and the order on the nodes guarantees that for all $v \in \mathcal{V}$ such that $L_v(t_i) < L_u(t_i)$, we have $v \prec u$, then $\mathcal{A}_{cnn}$ ensures that each node does not change its level more than $O(1)$ times between t_i and t_{i+1}.*

Proof. As for the validity proof, we proceed by analysing each phase separately: we show in lemmas 4.38, 4.39 and 4.41 that in each phase, the number of times a node may change its level is constant. □

For the rest of this section, we suppose that, with u the (only) node that moved at t_i, for all $v \in \mathcal{V}$ such that $L_v(t_i) < L_u(t_i)$, we have $v \prec u$.

Let us first look at the first phase.

Lemma 4.38. *During the first phase, that is, during the handling of* `Separation` *certificates, a node may increase its level only once.*

During the first phase, a node may decrease its level only once.

Proof. Let us first look at u, the node that moved at t_i. For u to increase its level in the first phase, a `Separation` certificate involving $f_u(t_i)$ must fail. As $G(t_i) \in \mathcal{G}(t_{i-1})$, we have that $f_u(t_i) \neq u$, except if u is the root in which case there is no other node than u at level $L_u(t_i)$. Thus (and because u is the only node that moved) no `Separation` certificate involving $f_u(t_i)$ can fail, and u cannot increase its level if that level was not decreased beforehand.

Let us now suppose that u decreases its level once, and let us show that u cannot increase or decrease its level again afterwards, during the first phase.

- First, when u decreases its level for the first time, line 44, by the failure of a $\texttt{Separation}_{\gamma,L_u(t_i)}(u,v)$ certificate with another node v, then two important things happen: v becomes u's new parent, line 44, and the symmetric $\texttt{Separation}_{\gamma,L_u(t_i)}(v,u)$ certificate is removed from *Cert* line 45 also, so that so that the new parent of u cannot fail a `Separation` certificate afterwards (as any other node than u did not move), and u cannot increase its level.

- We may then see that if u fails any other $\texttt{Separation}_{\gamma,k}(u,v)$ certificate at a level $k < L_u(t_i)$, then by supposition, $v \prec u$, so that v will decrease its level instead of u, and thus u cannot decrease its level again.

Let us now look at a node v that did not move, that is, $v \neq u$. Let us first suppose that v decreased its level once.

- Similarly to the previous case, when v decreases its level, than u becomes v new parent. Thus, if $L_v(t_i) < L_u(t_i)$, so that by supposition, any node w at level $L_v(t_i)$ that may fail a `Separation` certificate with u has $w \prec u$, so that v may not increase its level.

  If $L_v(t_i) = L_u(t_i)$, however, its level can increase once, but as this implies that a failing $\texttt{Separation}_{\gamma,L_u(t_i)}(u,w)$ with another node w has been treated, which on line 45 removes all `Separation` certificates involving u, the only node that moved, v can neither increase nor decrease its level for the rest of the first phase.

- As u is the only node that moved, v may fail a `Separation` certificate only with u. In addition, by Lemma 4.19, u and v may fail that kind of certificate only on one level. Thus, once v decreased its level, it cannot decrease its level again.

Let us now suppose that v increased its level once.

- As `Separation` certificates are treated in order of decreasing levels during the first phase, once v increased its level, no failing `Separation` certificates can remain at the level of its new parent, and thus v cannot increase its level a again.

- The node v can increase its level only through the Promote function, which by Remark 4.29 does not break the γ-separation property. No `Separation` certificate involving v can thus fail at its new level, and v cannot decrease its level again for the rest of the first phase.

Thus putting all together, we have proven that during the first phase, a node can increase and decrease its level only once (both at worst). □

The second phase is trivial.

Lemma 4.39. *During the second phase, that is, during the handling of* `ShortEdge` *and* `LongEdge` *certificates, no node may change its level.*

Proof. As failures of `LongEdge` and `ShortEdge` do not involve changes in maximal levels, no node may change its level during the second phase. □

Let us now look at the third and last phase. We will first make a generalization of Lemma 4.12, showing that because of the limited movement speed, the distance between a node and one of its ancestors does not change drastically.

Remark 4.40. *Let $t \in [t_i; t_{i+1}[$. Let v be any node, and let us suppose that at time t, w is the ancestor of v at level ℓ, and w' is the ancestor of v at level $\ell - 1$. Let us also suppose that $d(w', w, t) < \tau \cdot b^\ell$ (with $\tau \in \{\alpha, \beta, \gamma\}$).*

We have $d(v, w, t) < \left(\tau + \frac{1}{b}\right) b^\ell$.

Proof. Let $v_1, v_2, \dots v_\ell = w$ be the successive ancestors of v at time t. For any v_i, we have one of the following.

- v_i and v_{i+1} were v's ancestors at time t_i, in which case $d(v_i, v_{i+1}, t_{i-1}) < \alpha \cdot b^{L_{v_i}(t)+1}$ (as $G(t_i) \in \mathcal{G}(t_{i-1})$).
- An ancestor v_i of v got redirected at an instant between t_i and t, in which case, by Remark 4.33, $d(v_i, f_{v_i}(t), t) < \gamma \cdot b^{L_{v_i}(t)+1}$, and thus we have $d(v_i, f_{v_i}(t), t_{i-1}) = d(v_i, v_{i+1}, t_{i-1}) < \gamma \cdot b^{L_{v_i}(t)+1} + d_{mv} < \alpha \cdot b^{L_{v_i}(t)+1}$.
- An ancestor v_i of v saw its level decreased by the failure of a `Separation` certificate, in which case the new parent of v_i is at distance $\gamma \cdot b^{L_{v_i}(t)+1}$ away from it (see line 44 of $\mathcal{A}_{cnn}$), so that we also get $d(v_i, v_{i+1}, t_{i-1}) < \alpha \cdot b^{L_{v_i}(t)+1}$.

Note that if one of the ancestors of v has been promoted between t_i and t, then that ancestor was assigned, on line 83 of the algorithm, a new parent that already was its ancestor.

In any case, we have, for any ancestor v_i of v, that $d(v_i, v_{i+1}, t_{i-1}) < \alpha \cdot b^{L_{v_i}(t)+1}$. We can thus apply the same sum as in the proof of Lemma 4.12, and we get that $d(v, w, t_{i-1}) < \tau \cdot b^\ell + b^{\ell-1} - 1$. As one of the nodes may have moved, we have:

$$\begin{aligned} d(v, w, t) &< \tau \cdot b^\ell + b^{\ell-1} - 1 + d_{mv} \\ &< \tau \cdot b^\ell + b^{\ell-1} \qquad \text{as } d_{mv} \leq 1 \\ &< \left(\tau + \frac{1}{b}\right) b^\ell. \end{aligned}$$

□

In particular, we may apply Remark 4.40 to get the following:

- if $d(w', w, t) < \gamma \cdot b^\ell$, then $d(v, w, t) < \beta \cdot b^\ell$, and
- if $d(w', w, t) < \beta \cdot b^\ell$, then $d(v, w, t) < \alpha \cdot b^\ell$.

Lemma 4.41. *During the third phase, that is, during the handling of* `Cover` *and* `SCover` *certificates, a node may increase its level only three times (two for a node v such that $L_v(t_i) > 0$). No node may decrease its level.*

Proof. No `Cover` or `SCover` failure involves the reduction of the level of a node. It thus remains only to prove that that the level increases of nodes are limited during the third phase.

During the third phase, a node v may be promoted only on three places: on line 60, if v is the α-ancestor of another node, or on line 63 or 70 if v fails a `Cover` certificate. Note that if v is promoted, as no new children are assigned to v after the promotion, v cannot be the α-ancestor of another node. Line 60 is thus never executed twice on the same node, and it is enough to look only at `Cover` certificates involving v.

We have seen by induction, during the proof of Lemma 4.35, that when a failing `Cover` certificate is treated at level k, then the toptree at and above level $k + 1$ (the subgraph of the constrained navigating net induced by all nodes u such that $L_u \geq k$) validates the α-coverage and the ancestor invariant. Let us thus suppose that v got promoted once to level $k + 1$, by a

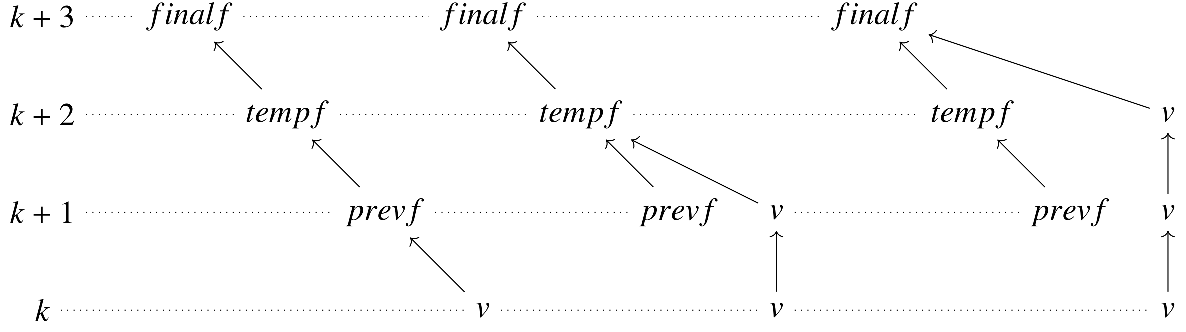


Figure 4.13 – Graphical representation of the different parents of v at time t (on the left), at time t' (in the middle), and after the third promotion (on the right). Dotted lines are there to help recognizing the levels of the nodes.

call to Promote at instant t, and prove that if v gets promoted a second time at instant t', then that second promotion never creates `Cover` certificates that are false. Let us call $prevf = f_v(t)$ the parent v had before that first promotion[13], $tempf = f_v(t')$ the parent v got after that first promotion and before that possible second promotion, and let us call $finalf$ the parent v would get after that second promotion. See Figure 4.13 for a graphical representation of these parents. Note that we may have $prevf = tempf = finalf$, in which case the distance between them is 0.

Let us suppose that $k > 0$. By the conditions of line 78, when v got promoted to level $k+1$, there are only two conditions under which the $\texttt{Cover}_{\tau,k+1}(v, tempf)$ certificate that was created as part of that promotion may fail. Either

$$\begin{cases} L_{\alpha a(prevf)}(t) < L_{\gamma a(prevf)}(t), \text{ and} \\ d(v, tempf, t_i) \geq \beta \cdot b^{k+2}, \end{cases} \tag{4.3}$$

in which case a failing $\texttt{Cover}_{\beta,k+1}(v, tempf)$ was created; or

$$\begin{cases} L_{\alpha a(prevf)}(t) > L_{\gamma a(prevf)}(t), \text{ and} \\ d(v, tempf, t_i) \geq \alpha \cdot b^{k+2}, \end{cases} \tag{4.4}$$

in which case a failing $\texttt{Cover}_{\alpha,k+1}(v, tempf)$ was created.

Let us now prove that in both cases, the $\texttt{Cover}_{*,k+2}(v, finalf)$ certificate that is created by the second promotion cannot fail. Again, we have that this certificate can fail only if either Equation 4.5 or Equation 4.6 is true.

$$\begin{cases} L_{\alpha a(tempf)}(t') < L_{\gamma a(tempf)}(t'), \text{ and} \\ d(v, finalf, t_i) \geq \beta \cdot b^{k+3}. \end{cases} \tag{4.5}$$

$$\begin{cases} L_{\alpha a(tempf)}(t') > L_{\gamma a(tempf)}(t'), \text{ and} \\ d(v, finalf, t_i) \geq \alpha \cdot b^{k+3}. \end{cases} \tag{4.6}$$

First case: let us suppose that Equation 4.3 is true, and that a failing $\texttt{Cover}_{\beta,k+1}(v, tempf)$ certificate is to be handled. First, we can see that as $L_{\alpha a(prevf)}(t) < L_{\gamma a(prevf)}(t)$, we cannot have $d(tempf, finalf, t_i) < \gamma \cdot b^{k+3}$, thus, as the toptree at and above level $k+1$ complies to the α-coverage, there are two sub-cases.

[13]Note that that first promotion may have happened during the first phase, in which case it is possible that $prevf$ decreased its level after the promotion.

- If $\gamma \cdot b^{k+3} \leq d(tempf, finalf, t_i) < \beta \cdot b^{k+3}$, then when the failing $\texttt{Cover}_{\beta,k+1}(v, tempf)$ is treated, the α-ancestor of $tempf$ gets promoted or redirected (line 60), before the second promotion of v (line 63), and thus, at time t', we have that $L_{\alpha a(tempf)}(t') > L_{\gamma a(tempf)}(t')$. Additionally, as $tempf$ and $finalf$ are ancestors of v at time t', we have, by Remark 4.40 that $d(v, finalf, t_i) < \beta \cdot b^{k+3}$. Thus, neither Equation 4.5 nor Equation 4.6 can be true, and no further promotion of v can happen.

- If $\beta \cdot b^{k+3} \leq d(tempf, finalf, t_i) < \alpha \cdot b^{k+3}$, then $tempf$ is the α-ancestor of v at the time of the handling of the $\texttt{Cover}_{\beta,k+1}(v, tempf)$. Thus, similarly to the previous sub-case, just before the second promotion of v, $tempf$ gets promoted, so that when function Promote is called for v, $tempf$ is v's ancestor both at level $k+2$ and $k+3$, so that by Remark 4.40, $d(v, finalf, t_i) < \frac{1}{b} \cdot b^{k+3}$. Thus, neither Equation 4.5 nor Equation 4.6 can be true, and no further promotion of v can happen.

  Note that in this case, $tempf$ cannot have been promoted beforehand (or else, $prevf = tempf$, in which case we cannot have $\beta \cdot b^{k+2} \leq d(v, tempf, t_i)$), so that we don't need to count $tempf$'s promotion towards the maximal number of promotions per nodes.

Second case: let us suppose that Equation 4.4 is true, and that there is a failing $\texttt{Cover}_{\alpha,k+1}(v, tempf)$ certificate to be handled. We have that $d(v, tempf, t_i) \geq \alpha \cdot b^{k+2} \leq (\beta + \frac{1}{b}) \cdot b^{k+2}$. Thus, as at time t, $prevf$ and $tempf$ are the ancestors of v at level $k+1$ and $k+2$ respectively, by contraposition of Remark 4.40, we get that $d(prevf, tempf, t_i) \geq \beta \cdot b^{k+2}$. As the toptree at and above level $k+1$ complies to the α-coverage, we have that $\beta \cdot b^{k+2} \leq d(prevf, tempf, t_i) < \alpha \cdot b^{k+2}$. Thus, ad the toptree satisfies also the ancestor invariant, we cannot have $\beta \cdot b^{k+3} \leq d(tempf, finalf, t_i)$. We have again two sub-cases.

- If $d(tempf, finalf, t_i) < \gamma \cdot b^{k+3}$, then by Remark 4.40, as $tempf$ and $finalf$ are v's ancestors, we have that $d(v, finalf, t_i) < \beta \cdot b^{k+3}$, and thus neither Equation 4.5 nor Equation 4.6 can be true, and no further promotion of v can happen.

- If $\gamma \cdot b^{k+3} \leq d(tempf, finalf, t_i) < \beta \cdot b^{k+3}$, then by Remark 4.40, $d(v, finalf, t_i) < \alpha \cdot b^{k+3}$, and by ancestor invariant, $L_{\alpha a(tempf)}(t') > L_{\gamma a(tempf)}(t')$. Again, neither Equation 4.5 nor Equation 4.6 can be true.

We have thus proven that a node v at level $k > 0$ cannot be promoted more than twice during the third phase.

For level $k = 0$, if v gets promoted once, then the next promotion of v happens at level 1. Thus, the same proof as above can be used to show that v cannot be promoted more than three times during the third phase. □

Update time

Knowing, from previous section that changes of levels happen only a constant number of times, we prove that the update time is polylogarithmic in Theorem 4.42.

Recall that Φ, the aspect ratio, is so that $\Phi = \frac{d_{max}}{d_{min}}$, with d_{max} (resp. d_{min}), the maximum (resp. minimum) distance possible between two points. As explained in Section 4.1.1, our algorithms work under the relaxed assumption on minimal distance that only a constant ρ of nodes can be situated in a ball of radius $d_{mv} = 1 > d_{min}$. Similarly to this[14], we could remove the dependency on d_{min} of our performance results, and replace Φ by $\frac{d_{max}}{d_{mv}} = d_{max}$ in the following, however, in order to be able to compare our algorithms with related results, we prefer to use Φ.

[14]In both cases, this is a result of treating the level 0 as a special case, where it is guaranteed that thanks to Definition 4.1, only a constant number of nodes can be children of a given node at level 0.

Theorem 4.42. *In the low mobility setting, when ρ is a constant, and the metric space has a constant doubling dimension, the computations induced by $\mathcal{A}_{cnn}$ every t_i take $O(\log^2 \Phi)$ time.*

Proof. Computations in $\mathcal{A}_{cnn}$ are limited to corrections of certificates. We will see in Lemma 4.44 that there is a logarithmic number of certificates that fail at t_i. Combined with Lemma 4.45, this means that only a logarithmic number of certificates have to be treated in total between instants t_i and t_{i+1}.

As by Lemma 4.46 each of these certificates takes $O(\log \Phi)$ time to be treated, we get our result. □

First, we see that only a logarithmic number of certificates may fail because of the movement, in Lemma 4.44. But for this, we need to prove that the size of all the needed sets associated with one node u is logarithmic in the aspect ratio, in Lemma 4.43

Lemma 4.43. *When ρ is a constant, and the metric space has a constant doubling dimension, for any $u \in \mathcal{V}$, $card(\{L_u\} \cup \{f_u\} \cup N_u \cup C_u \cup PN_u) = O(\log \Phi)$.*

Proof. For each level $k \geq 1$, there is a minimum distance between nodes, by γ-separation: $\forall v_1, v_2$ so that $L_{v_1} \geq k$ and $L_{v_2} \geq k$, $d(v_1, v_2) \geq \gamma \cdot b^k$. As $\forall (v,k) \in N_u, d(u,v) < c \cdot b^k$, each neighbor of u at a given level k is in a ball of radius $c \cdot b^k$. As, by doubling dimension, this ball can be covered by a constant number of balls of radius $\gamma \cdot b^k$, there is only a constant number[15] of neighbors for u at each level k. Similarly, there is only a constant number of children for u at level k (as $\forall (v,k) \in C_u, d(u,v) < \alpha \cdot b^{k+1}$) and there is only a constant number of potential neighbors at level k (as $\forall (v,k) \in PN_u, d(u,v) < (2\alpha + c)b^{k+1}$).

For level $k = 0$, by Definition 4.8, Definition 4.11, and Definition 4.13, the sets of neighbors, potential neighbors and children are empty at level 0. As ρ is a constant, any node u may also have only a constant number of children at level 0.

As the number of levels a node can be present in is $O(\log \Phi)$, we get our result. □

Lemma 4.44. *In the low mobility setting, when ρ is a constant, and the metric space has a constant doubling dimension, only $O(\log \Phi)$ certificates may fail at time t_i.*

Proof. This lemma is a direct consequence of Lemma 4.43, as with u the point that moved, any certificate that fails after the movement must involve u. □

Lemma 4.45. *In the low mobility setting, when ρ is a constant, and the metric space has a constant doubling dimension, the total number of certificates that are created false between t_i and t_{i+1} is $O(\log \Phi)$.*

Proof. Let us look at the different types of certificates separately.

- `Separation`(u, v) and `ShortEdge`(u, v) certificates are created only on line 23, when u and v become neighbors. In turn, u and v may be added to each other's neighbors set only on line 51, through the failure of a `LongEdge` certificate (which by definition happens only when u and v are close enough), or on line 86 or line 85, under the condition, line 84 that u and v are close enough.

  Thus, no `Separation` or `ShortEdge` certificate is created false between t_i and t_{i+1}.

- `LongEdge`(u, v) certificates are created on line 26 and 31, under the condition that u and v are not already neighbors. Thus no `LongEdge` certificate is created false, except if one of these two lines is executed at a time where u and v are close enough to be neighbors, but

[15]Note that this number depends on the doubling constant

have not been added yet to each other's set of neighbors (which will eventually happen by Lemma 4.32). We have however seen in the proof of Lemma 4.32, that for any two nodes that are in that situation, a `LongEdge`(u, v) certificate already exists.

Thus, no new `LongEdge` certificate is created false between t_i and t_{i+1}.

- Concerning `Cover` certificate, we can see that the failure of any certificate may create only a constant number of `Cover` certificates. If a `Separation` certificate fails, then a `Cover` certificate is created for the node that decreases its level, and for all of its children, that are, as we have seen in the proof of Lemma 4.43, constant in number. If a `Cover` certificate fails, only one other `Cover` certificate is created. Thus, the $O(\log \Phi)$ certificates that may fail at time t_i by Lemma 4.44 each induces only a constant number of created failing certificates.

  The failure of a `Cover`(v, f_v) certificate either ends with Redirect being called on v, which creates only valid `Cover` certificates (by Remark 4.33), or with a call to Promote on v, which increases the level of v node. Thus, as by Theorem 4.37, v can change its level only a constant number of time, each of the potentially $O(\log \Phi)$ failing `Cover` certificates, only lead in total to a constant number of failing `Cover` certificates.

  Thus, only $O(\log \Phi)$ `Cover` certificates are created false between t_i and t_{i+1}.

- Each time an $SCov$ certificate is created, a `Cover` certificate is also created.

  Thus, only $O(\log \Phi)$ `SCover` certificates are created false between t_i and t_{i+1}.

□

We then finally look at the cost to repair each certificate, which finishes the proof of Theorem 4.42.

Lemma 4.46. *When the metric space has a constant doubling dimension, the computations made by $\mathcal{A}_{cnn}$ for any failing certificate are in $O(\log \Phi)$ time.*

Proof. Almost all operations executed when treating a failing certificate are constant in time. We have seen in the proof of Lemma 4.43, that the neighbors, children, and potential neighbors of a node are constant in number at a given level. If all sets involving different levels are stored in a structure that indexes each level to the list of nodes in the set at that level, then all nodes of a given level can be retrieved in constant time.

The only operations for handling failing certificates in $\mathcal{A}_{cnn}$ that do not involve these sets, and that are not trivially constant in time, are the search for α-ancestors and γ-ancestors on lines 58 and 78, which take $O(\log \Phi)$ time. As these searches are always done once or twice per failing certificate, we get our result. □

Structure size

We suppose in this section that nodes respect the constraint of minimal distance, as stated in Definition 4.1, and show that the total memory space of a constrained navigating net is linear.

Note that the total memory size is the sum of the sizes of all variables plus the total number of certificates. As the presence of `Separation`, `ShortEdge`, `Cover` and `SCover` certificates depends on the presence of neighbors, parents, and children, these certificates are already included when counting the sizes of the variables. However, `LongEdge` certificates are conditioned by the fact that two nodes are potential neighbors; thus, to count the total memory size, we have to take into consideration the sizes of the PN_u sets.

Let us then prove that the total memory space is linear with the total number of nodes:

Theorem 4.47. *When ρ is a constant, and the metric space has a constant doubling dimension, the total memory usage of $\mathcal{A}_{cnn}$ is $O(n)$.*

Proof. For any node u, there are five variables associated with u: L_u, f_u, N_u, C_u, PN_u. Both f_u and L_u have a constant memory cost, so their total cost is linear.

First, let us prove that the total size of neighbors is linear in size, in other words, let us prove that $\sum_{u\in\mathcal{V}} \text{card}(N_u) = O(n)$. Let $N_u(+) = \{(v,k) \in N_u : L_v \geq L_u\}$.

Note, as stated in the proof of Lemma 4.43, that there is only a constant number of neighbors per level. By Lemma 4.19, all nodes from $N_u(+)$ are neighbors of u at level L_u, and thus $\text{card}(N_u(+)) = O(1)$. Thus, we have $\sum_{u\in\mathcal{V}} \text{card}(N_u(+)) = O(n)$.

Note that for any nodes u and v that are neighbors at some level k, that is, such that $(u,k) \in N_v$ and $(v,k) \in N_u$, either $(v,k) \in N_u(+)$ or $(u,k) \in N_v(+)$ (or eventually both). Thus, we have $\sum_{u\in\mathcal{V}} \text{card}(N_u) \leq 2 \cdot \sum_{u\in\mathcal{V}} \text{card}(N_u(+))$. Finally, this leads to $\sum_{u\in\mathcal{V}} \text{card}(N_u) = O(n)$.

As each node has only one parent, the total cost of children is linear too.

Only the size of potential neighbors remains to be analyzed. Note that by Definition 4.11, each pair (u,v) of potential neighbors at a level k is associated with a pair (u',v') of neighbors at level $k+1$, so that either u' and v' are parents of u and v, or they are the same nodes. We have seen in the proof of Lemma 4.43 that the children of a node at a given level are constant in number. It follows that the number of possible pairs of potential neighbors associated with a given pair of neighbors is also constant in size (even if this constant is raised to the square). As we have already proven that the total size of neighbors is linear, we thus get that the total size of potential neighbors is linear too. □

Similarly, thanks the the doubling dimension, we may see that the size of the variables associated with a node u (that is, $\text{card}(\{L_u\} \cup \{f_u\} \cup N_u \cup C_u \cup PN_u)$) is $O(\log \Phi)$. Sadly, in the distributed setting, the local memory usage of the nodes will be higher, as some additional sets will be added to the data structure (see Theorem 4.63, Section 4.4.3).

4.3.6 Improving $\mathcal{A}_{cnn}$ for a better update time

We have seen in Theorem 4.42 that the update time for $\mathcal{A}_{cnn}$ takes $O(\log^2 \Phi)$ computations per time step. As stated in Theorem 4.17, the DefSpanner, another navigating net, can be maintained with $O(\log \Phi)$ computations per time step. We will see in this section that it is possible to improve $\mathcal{A}_{cnn}$ so as to also get $O(\log \Phi)$ on constrained navigating nets, by adding pointers in the data structure. We will call $\mathcal{A}_{cnnptr}$ the resulting improved algorithm.

First, let us see why $\mathcal{A}_{cnn}$ may need $\Theta(\log^2 \Phi)$ computations per time step.

The principle of the proof of Theorem 4.42 is to see that there are $O(\log \Phi)$ certificates to handle at each time step, and that each of these certificates takes $O(\log \Phi)$ computations to be treated. Let us see a situation in which $\mathcal{A}_{cnn}$ would create $\Theta(\log \Phi)$ failing certificates in one time step t_i that all need $\Theta(\log \Phi)$ computations to be updated. See Figure 4.14 for a graphical representation. Let u be the node that moved at t_i. If u fails a $\texttt{Separation}_{\gamma,k}(u, v_k)$ certificate at each level $k < L_u$ with a node v_k, then each v_k will decrease its level as $v_k \prec u$. If additionally, each v_k has a child v'_k at level $k-1$, then these children will need to be promoted, and will be assigned as new parent f_{v_k}, the parent of v_k.

If $\beta \cdot b^{k+1} \leq d\left(v'_k, f_{v_k}\right) < \alpha \cdot b^{k+1}$, then $\mathcal{A}_{cnn}$ needs to check whether $L_{\alpha a(v_k)} < L_{\gamma a(v_k)}$, which may take $\Theta(\log \Phi)$ computations if both u's γ-ancestor and α-ancestor are situated high up in the hierarchy.

As this may happen at each of u's levels, we may get the need to do up to $\Theta(\log \Phi)$ of those searches for α-ancestors or γ-ancestors, leading to a total cost of $\Theta(\log^2 \Phi)$.

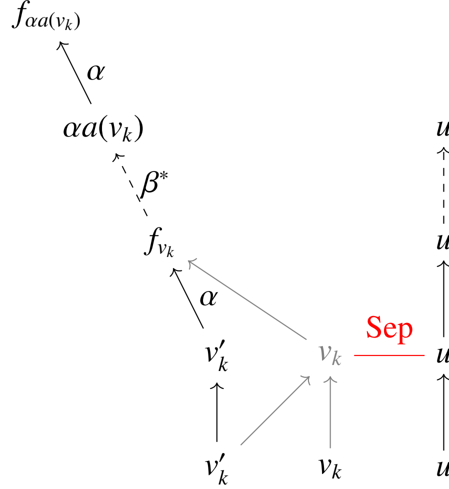


Figure 4.14 – Graphical representation of a part of a situation that leads to $O(\log^2 \Phi)$ computations with $\mathcal{A}_{cnn}$. The red line marked with "Sep" represents a failing `Separation` certificate. Arrows represent parent/child relations, and they are gray if they are removed because of the failing `Separation` certificate.

As it seems impossible to find a better bound on the number of failing certificates (if the root moves, it may invalidate a certificate in each of its $\Theta(\log \Phi)$ levels), we reduce, in order to improve that update time per time step, the cost of treating the certificates. In particular, we lower the cost of searching the ancestors of the nodes in a modified version of $\mathcal{A}_{cnn}$, that we call $\mathcal{A}_{cnnptr}$. In $\mathcal{A}_{cnnptr}$, each node u maintains a couple $pointer_u$ that is reset at each time step, and recomputed when a node looks for its γ- or α-ancestor. This ensures that each time we have found the γ- or α-ancestor of a node, there is no need to look for it a second time. The value of $pointer_u$ is computed in function findLower, described in Algorithm 17.

At the start of each time step, $pointer_u$ has an "empty" value $(NULL, -1)$. With a call to findLower, we may get $pointer_u = (v, k)$, where v is either the γ- or α-ancestor u had last time a change was made to $pointer_u$, whichever was lowest in the hierarchy, and k is the level of the node that was both a child of v and an ancestor of u. This level k allows to compute in constant time if v can be the γ-ancestor or the α-ancestor of u, as Definition 4.21 and Definition 4.22 depend on that level k: v is the γ-ancestor of u iff $L_v \geq k + 2$ or $d(v, f_v) < \gamma b^{L_v+1}$; v is the α-ancestor of u iff $L_v = k + 1$ and $\beta b^{L_v+1} \leq d(v, f_v)$. Recall that if a call is made to Promote on v, then v becomes the γ-ancestor of u, as L_v gets increased so that $L_v \geq k+2$: thus a node stored in $pointer_u$ that was previously u's α-ancestor may have become u's γ-ancestor by promotion of v, explaining the necessity to be able to rapidly compute if the previously stored node is γ- or α-ancestor. At the end of each time step (and at time 0), $pointer_u$ is reset to $(NULL, -1)$.

The function findLower does not necessarily return exact results: because of changes in the hierarchy, a node u may see its γ- or α-ancestor change, so that $pointer_u$ no longer points to the right node. We will see however, in Lemma 4.48, that the function returns results that can be used for maintaining correct constrained navigating nets.

In order to take into account function findLower, two additional changes have to be made, as explained in Algortihm 18. The whole algorithm $\mathcal{A}_{cnnptr}$ can be found in Appendix D.

As has been done for $\mathcal{A}_{cnn}$ in Theorem 4.25, we first need to prove that $\mathcal{A}_{cnnptr}$ yields valid constrained navigating nets, which we will do in Theorem 4.49. Let us first show in Lemma 4.48, that a call to findLower on node f_u can be used to know for a node u whether $L_{\alpha a(u)} < L_{\gamma a(u)}$.

Algorithm 17 Function FINDLOWER, used to search the first ancestor w of a node v, such that w is either the γ-ancestor or the α-ancestor of v. The input node u should be such that $u = f_v$.

```
1: function findLower(u, k)     (*Recursive function: to get the ancestor of a node v, a call to
   findLower(f_v, L_v) should be made.*)
2:    if L_u ≥ k + 2, or if RoundDistance(u, f_u, k + 1) = γ or α then    (*Check whether u is
   the γ- or α-ancestor of its descendants. RoundDistance, is given in Algorithm 14, line 1.*)
3:        return (u, k)
4:    else if pointer_u ≠ (NULL, −1) then
5:        return pointer_u
6:    else
7:        pointer_u ← findLower(f_u, L_u)
8:        return pointer_u
```

Algorithm 18 $\mathcal{A}_{cnnptr}$: changes to use function FINDLOWER.

```
1: replace lines 58 through 60 of Algorithm 16 with the following:
2: (anc, k) ← findLower(f_u, L_u)
3: if L_anc = k + 1 and RoundDistance(anc, f_anc, L_anc) = α then
4:    if ∃(f′, L_anc + 1) ∈ N_{f_anc} : d(anc, f′) ≤ γb^{L_anc+1} then
5:    Redirect(anc, f_anc, f′) else Promote(anc, f_anc)  end if

6: replace line 78 of Algorithm 16 with the following:
7: if RoundDistance(v, newf, L_v + 1) = γ (resp. β) then
8:    τ′ ← γ (resp. β)
9: else   (*RoundDistance(v, newf, L_v + 1) = α*)
10:   (anc, k) ← findLower(f_v, L_v)
11:   if L_anc = k + 1 and RoundDistance(anc, f_anc, L_anc) = α then τ′ ← β else τ′ ← α end if
```

Lemma 4.48. *If $L_{\alpha a(u)} < L_{\gamma a(u)}$ when there is a call to function* findLower *on f_u and L_u in algorithm $\mathcal{A}_{cnnptr}$, then the function returns (v, k) with $v = \alpha a(u)$, and with k the level of the child of v that is also an ancestor of u.*

Proof. Let us call u the node that is given as entry to findLower, and v either the γ- or α-ancestor of u, whichever is lowest in the hierarchy.

First, we can notice that as with $\mathcal{A}_{cnn}$, the structure remains coherent throughout the execution of $\mathcal{A}_{cnnptr}$ (see Definition 4.26). Thus, by Definition 4.21 and Definition 4.22, if for any ancestor w of u, $pointer_w = (NULL, -1)$ findLower trivially returns v.

As can be seen in Algorithm 17, changes to $pointer_u$ happen only if $pointer_u = (NULL, -1)$. As also any other value for $pointer_u$ immediately stops the search for ancestor and returns the content of $pointer_u$, it remains to be proven that when its value is set, then that value complies to the lemma for any future call of findLower until all the updates of the current time step have been done.

As long as no ancestors of u changes, $pointer_u$ remains valid. Ancestors may change only in two ways.

- A call to the Redirect function on a node w that is both ancestor of u and descendant of v may make it so that v is no longer ancestor of u. However, by Remark 4.33, w becomes the γ-ancestor of u, thus any subsequent call to Redirect on u cannot happen with $L_{\alpha a(u)} < L_{\gamma a(u)}$.
- A call to Promote on a node w that is both ancestor of u and descendant of v makes so that w becomes u's new γ-ancestor, and again, any subsequent call to Redirect on u cannot happen with $L_{\alpha a(u)} < L_{\gamma a(u)}$.

□

Theorem 4.49. *In the low mobility setting, assuming that the structure is correctly initialized, that is, $G(0) \in \mathcal{G}(0)$, and that all certificates of Table 4.3 are in Cert at time 0, then when using $\mathcal{A}_{cnnptr}$, $\forall i \geq 0, G(t_{i+1}) \in \mathcal{G}(t_i)$.*

Proof. The only difference between $\mathcal{A}_{cnn}$ and $\mathcal{A}_{cnnptr}$ relies in the changes pointed out in Algorithm 18.

We can see that when $L_{\alpha a(u)} < L_{\gamma a(u)}$, the instructions of Algorithm 18 are equivalent to those of $\mathcal{A}_{cnn}$ if findLower returns the α-ancestor, which is the case by Lemma 4.48.

When $L_{\alpha a(u)} > L_{\gamma a(u)}$, $\mathcal{A}_{cnn}$ either does nothing or creates valid certificates. $\mathcal{A}_{cnnptr}$ however, if findLower does not return the correct ancestor, may call Redirect or Promote on a node v that is not u's α-ancestor (line 5 of Algorithm 18), or create a false $\texttt{Cover}_{\beta,L_u}(u, f_u)$ certificate (line 11 of Algorithm 18, and line 81 of Algorithm 16), which in turn also calls Redirect or Promote on nodes u and v (line 57 of Algorithm 16). All these calls happen by failure of `Cover` certificates, and thus happen during the third phase of the algorithm, so that, by Lemma 4.35, they keep the validity of the hierarchy.

We have thus proven that in any case, the changes mentioned in Algorithm 18 maintain the validity of the constrained navigating net. □

Concerning the performance, we can see that the only additional memory cost are the values of $pointer_u$, which take a constant size per node. The memory usage of $\mathcal{A}_{cnnptr}$ is thus $O(n)$ as for $\mathcal{A}_{cnn}$. The computation cost, however, is better than $\mathcal{A}_{cnn}$, as stated in Theorem 4.52.

To prove Theorem 4.52, let us first make a computational remark in Remark 4.50, that will be used in Lemma 4.51 to show that the total number of edges from nodes close to the node u that moved to the root is $O(\log \Phi)$.

Remark 4.50. *Let $t \in [t_i, t_{i+1}[$, so that $G(t_i) \in \mathcal{G}(t_{i-1})$. Let u and v be two nodes such that $L_u(t) \geq k$, $L_v(t) \geq k$ and $d(u, v, t) < (\gamma + \alpha) \cdot b^k$.*

For node w that is the ancestor of v at level $k' > k$ at time t, we have $d(u, w, t) < (\gamma + \alpha) \cdot b^{k'}$.

Proof. As $G(t_i) \in \mathcal{G}(t_{i-1})$, we have $d(v, f_v, t_{i-1}) < \alpha \cdot b^{k+1}$, so that $d(v, f_v, t_i) < \alpha \cdot b^{k+1} + 1$ (as $d_{mv} = 1$), and thus:

$$\begin{aligned} d(u, f_v, t) &< \alpha \cdot b^{k+1} + 1 + (\gamma + \alpha) b^k \\ &< \left(\alpha + \frac{\gamma}{b} + \frac{\alpha}{b}\right) b^{k+1} + 1. \end{aligned}$$

As $b \geq 6$, $\alpha = \frac{b-2}{b}$, and $\gamma = \frac{b-4}{b}$, we get $d(u, f_v, t) < (\gamma + \alpha) \cdot b^{k+1}$. By induction on the ancestors of v, we get the result. □

Lemma 4.51. *Let $G(t_i) \in \mathcal{G}(t_{i-1})$. Let $u \in \mathcal{V}$, and let $U = \{v \in \mathcal{V} : L_v(t_i) \leq L_u(t_i) \wedge d(u, v, t_i) < (\gamma + \alpha) \cdot b^{L_v(t_i)}\}$. When the metric space has a constant doubling dimension, the total number of parent/children edges that occur in the paths from the nodes of U to the root is $O(\log \Phi)$.*

Proof. By doubling dimension, the subset of U of nodes at a level $k \leq L_u(t_i)$ contains a constant number of nodes. By Remark 4.50, all ancestors at level k of nodes from U belong to U. We thus have that the number of parent/children edges of the paths from all nodes of U to their ancestor at level $L_u(t_i)$ is $O(\log \Phi)$.

In addition, as there is a constant number of nodes in U at level $L_u(t_i)$, the total number of parent/children edges from them to the root is also $O(\log \Phi)$. □

Theorem 4.52. *If u is the only node that moved at t_i, and the order on the nodes guarantees that for all $v \in \mathcal{V}$ such that $L_v(t_i) < L_u(t_i)$, we have $v \prec u$, then the computations induced by $\mathcal{A}_{cnnptr}$ every t_i take $O(\log \Phi)$ time.*

Proof. As we have seen in the proof of Lemma 4.46, all operations of $\mathcal{A}_{cnn}$ take a constant number of time except the search for α-ancestors and γ-ancestors. The same goes for $\mathcal{A}_{cnnptr}$, where only a call to function FINDLOWER may take more than a constant time to execute.

Here, when FINDLOWER is called on a node v, any further call of FINDLOWER on v takes a constant time, as we then have $pointer_v \neq (NULL, -1)$. Let us call u the node that moved at time t_i.

The function FINDLOWER can be called on a node v only in three circumstances.

- Node v is promoted because a `Separation`(f_v, w) certificate fails with another node w (when `Separation` fails, all children of the node that decreases its level are either redirected or promoted). As only one node may have moved, we have either $u = f_v$ or $u = w$. In both cases, we have by triangular inequality $d(v, u, t_i) < (\gamma + \alpha) \cdot b^{L_v(t_i)}$. Thus, by Lemma 4.51, the total cost of FINDLOWER in this situation is $O(\log \Phi)$.

- If a `Cover`(v, f_v) certificate fails, than either it was because of u's movement, so that $u = v$ or $u = f_v$, or it was because of a previous promotion of v, which happened in the same conditions as the previous case. Again, we have $d(v, u, t_i) < (\gamma + \alpha) \cdot b^{L_v(t_i)}$, and by Lemma 4.51, the total cost of FINDLOWER in this situation is $O(\log \Phi)$.

- Node v may be promoted because it is returned by another call to FINDLOWER on a node w that fails a `Cover`(w, f_w) certificate. We have $L_w(t_i) \leq L_u(t_i)$, and thus v is the ancestor of w with $d(w, u, t_i) < (\gamma + \alpha) \cdot b^{L_w(t_i)}$. Again, the total cost of FINDLOWER in this situation is $O(\log \Phi)$.

Finally, $\mathcal{A}_{cnnptr}$ does some computations in addition to $\mathcal{A}_{cnn}$, as at each time step, for each node u, $pointer_u$ should be reset to $(NULL, -1)$. This can be done in $O(\log \Phi)$ time: by the above, the number of nodes u that change $pointer_u$ is $O(\log \Phi)$ and thus by storing in an array each node u, the additional cost of this step is $O(\log \Phi)$ (and $O(\log \Phi)$ memory, which is lower than $O(n)$). □

Now that the update time has been analyzed for $\mathcal{A}_{cnnptr}$, let us show that the memory cost is the same as for $\mathcal{A}_{cnn}$.

Theorem 4.53. *For a given size of the space, when ρ is a constant, the total memory usage of $\mathcal{A}_{cnnptr}$ is $O(n)$.*

Proof. For any node u, the size of $pointer_u$ is constant, and thus the total size of the pointers is $O(n)$. As all other elements of the data structure of $\mathcal{A}_{cnnptr}$ are also in the data structure of $\mathcal{A}_{cnn}$, we have by Theorem 4.47 that the total memory needed for $\mathcal{A}_{cnnptr}$ is $O(n)$. □

4.4 Distributed Navigating Nets

In this section, we look at the distributed setting. We make similar assumptions as in Chapter 3.

The network is supposed to be synchronous, and at each time step, as many synchronization rounds can be executed as needed before the next time step. The aim of synchronous algorithms, however, is to need as few communication rounds per time step as possible. Any sent messages is supposed to be received before the next synchronization round. Computational power is not of concern, as we suppose that each node can make as many computations as needed during each synchronization round.

Each node has a unique identifier, and maintains a set of nodes that we call address book. A node may send messages only to nodes from its address book. Identifiers may be sent, so that a node u may send a message to tell another node v to add any node w to v's address book, provided w is in u's address book at the time the message is sent.

In the distributed setting, the connection graph G represents the ability for nodes to send messages, and $G(t) = (\mathcal{V}, E(t))$ is defined by the union of the address books: $(u, v) \in E(t)$ if and only if v is in u's address book at time t. We assume that distance estimation is perfect, so that each node knows at all time the exact distance to all other nodes in their address book.

4.4.1 Related Work

To the best of our knowledge, only one article has been published to maintain navigating nets in a distributed setting. In [56], a structure similar to the DefSpanner from [55] is presented, denoted as D-Spanner.

A D-Spanner is a navigating net with $b = 2$ and $\gamma = 1$, as in a DefSpanner. *True parents* are defined such that if a node at level k is at distance at most b^{k+1} away from its parent, then that parent is a true parent. A D-Spanner may accept *relaxed parents*, that is, parents that are slightly farther away, effectively making it so that α is a constant such that $\alpha > 2$. The value of c is fixed to a constant such that $c > 4 + \alpha$. While relaxed parents are allowed in the structure, the aim is to make it so that all parents are true parents: as soon as a node gets a relaxed parent, the update algorithm tries to find a true parent for that node.

It is shown that a D-Spanner can be maintained in a distributed setting. The same set of certificates as in the DefSpanner are used (see Section 4.3.1), and each node maintains a local copy of all certificates it is involved in. When certificates fail, the structure is updated as in [55],

with the difference that instead of immediately looking for a true parents for a node that gets too far away from its true parent, its parent instead becomes a relaxed parent.

- When a parent-child certificate fails, the former parent becomes a relaxed parent.
- When a separation certificate fails between two nodes u and v, the node with lowest level, say u with $L_u = k$ decreases its level, and all children of u at level $k - 1$ take v as relaxed parent.
- When an edge certificate or potential neighbor certificate fails, as with the DefSpanner, the neighbors are respectively added or removed as in [55].

Nodes with relaxed parents are treated as orphans from DefSpanners: a node u of level k looks for an alternative parent among the neighbors at level $k + 1$ of its relaxed parent f_u, and if none is found and f_u itself has a true parent, then u is promoted to level $k + 1$, taking that grandparent (of level $k + 2$) as relaxed parent; if none of this is the case, u has to wait until f_u gets a true parent to be promoted (no deadlock is possible as this dependency goes only in one direction). This is repeated until a true parent is found, but is done in parallel: once failing certificates are corrected, the nodes may continue to move, and relaxed parents are treated in the background. In the flight plan model, this means that if the computation and communication speed is low in comparison to the movement speed and to α, then a node with a relaxed parent gets a true parent before getting too far away from its relaxed parent.

In terms of number of messages, D-Spanners incur an amortized cost of $O(\log \Phi)$ messages per distance unit a node moves in the flight plan model. However, the number of needed communication rounds, which may be different than the number of messages, is not discussed. Also the "parallel" updates of relaxed parents leave room for interpretation in the Black-Box model, as it is unclear how the associated computations should be allotted to the time steps. This is the topic of the discussions in the next section.

4.4.2 Adapting D-Spanners to the Black-Box Model

We identify two difficulties when adapting D-Spanners to synchronous networks with the Black-Box model. First, there is a trade-off between allowed movement speed and computational efficiency, as allowing too few communication rounds per time step for correcting the relaxed parents may lead to strong requirements on the maximal movement speed (or else nodes may get too far away from their relaxed parent before the algorithm finds a true parent for them). Second, it may happen that coordination is needed when several nodes have relaxed parents, which may cost additional communication rounds to correct relaxed parents.

Trade-off Between Computational Efficiency and Allowed Movement Speed

We can observe, for a node u that has a relaxed parent, that looking for an alternative parent and promoting u one level takes $O(1)$ communication rounds, as the list of neighbors of the relaxed parent need to be retrieved. As a node may have to go up all the hierarchy when a parent-child certificate fails (as seen on Figure 4.3, p.125), the total cost to repair a relaxed parent is $O(\log \Phi)$ communication rounds. Correcting relaxed parents thus has a non-negligible cost. As our goal is to have as few communication rounds per time step as possible, it is tempting to reduce the number of times each node can be promoted per time step, and repair the relaxed parents in parallel on several time steps. However, the less communication rounds are allotted for relaxed parents per time step, the more time steps it will take for a node to get a true parent. In turn,

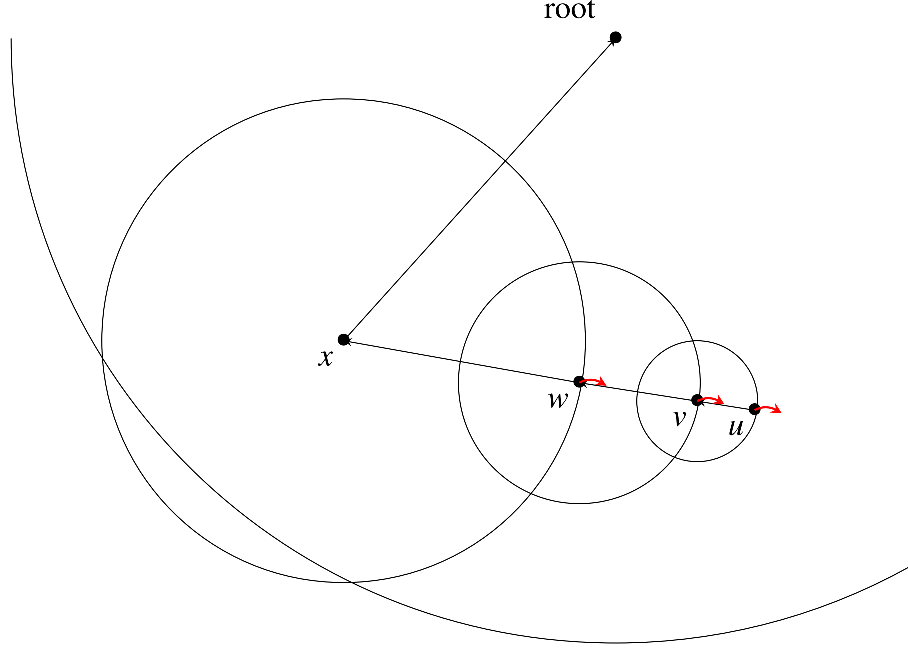


Figure 4.15 – Example of a case where a node in a D-Spanner needs to wait $\log \Phi$ time steps before getting promoted. The red arrow represent movements of the nodes that happen at the same time step.

to avoid nodes to get too far away from their relaxed parents in the meantime, this means that lower movement speeds can be accepted by the structure.

To see this trade-off between less communication rounds and higher allowed movement speed, let us suppose, in the high mobility setting, that at each time step, a node with a relaxed parent may be promoted only once. We can see in Theorem 4.54 that the associated speed limit is quite restrictive, as it is inversely proportional to Φ.

Theorem 4.54. *In the high mobility setting, if a node may be promoted only once per time step, it is necessary to have* $d_{mv} \leq \frac{(\alpha-1)b}{2(\log \Phi - 1)}$ *in order to maintain a D-Spanner.*

Proof. If a node u at level 0 has a relaxed parent v, and all its ancestors at each level also have relaxed parents, then u has to wait for each of its ancestors to get true parents before being able to get promoted. This situation may happen if all of u's ancestors fail parent-child certificates at the same time, as represented on Figure 4.15 (which is similar to Figure 4.3): all of u's ancestors move away from their respective parent at the same time step.

The node u may thus have to wait $\log \Phi - 1$ time steps before being able to be promoted. During that wait time, u should not get at a distance more than $\alpha \cdot b^1$ from its parent v. As v becomes a relaxed parent when $d(u, v) > b^1$, the distance should not increase of more than $\frac{(\alpha-1)b}{\log \Phi - 1}$ distance units per time step. As both u and v may move of d_{mv} distance units per time step in the high mobility setting, we need $d_{mv} \leq \frac{(\alpha-1)b}{2(\log \Phi - 1)}$ □

While other update schedules for relaxed parents may be devised for D-Spanners, we propose in the next section an adaptation of $\mathcal{A}_{cnn}$ to distributed networks in the low mobility setting that gets rid altogether of the trade-off between movement speed and number of needed communication rounds per time step.

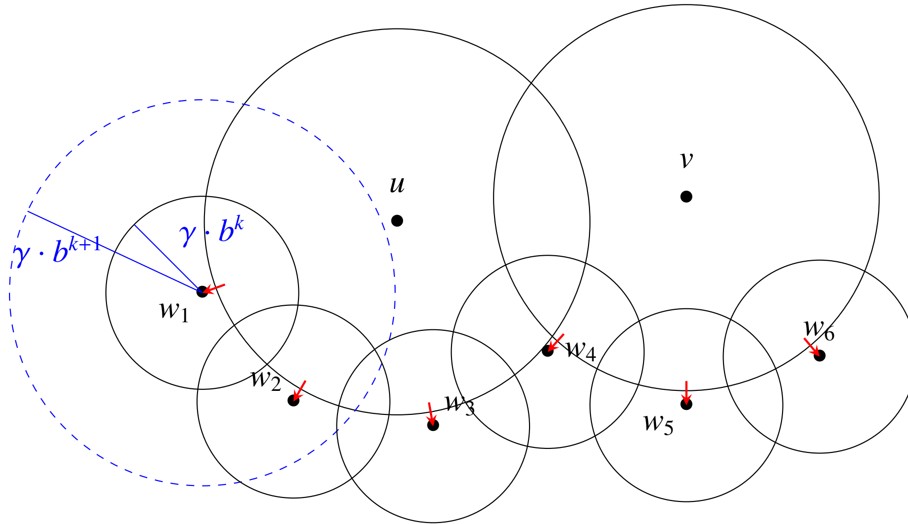


Figure 4.16 – Example where coordination is needed among the nodes in order to promote them in parallel. The dashed blue circle represents the ball centered on w_1 in case it gets promoted to level $k + 1$. After the movement represented by the short red arrows, u is the relaxed parent of w_1, w_2, and w_3, and v is the relaxed parent of w_4, w_5, and w_6.

Coordination Among Nodes

Another problem that arises with D-Spanners is when two nodes u and v at the same level k have a relaxed parent but no alternative true parent (so that both u and v are looking to be promoted according to the rules of the update algorithm to D-Spanners), but are close to each other in such a way that $d(u, v) < \gamma \cdot b^{k+1}$ (so that both nodes cannot get promoted, as this would break the γ-separation). In this case, only one promotion is enough: if u gets promoted for example, then u is close enough to v to become its true parent. However, if the promotions of the nodes is treated in parallel, coordination is required for the nodes to decide which one should get promoted.

This problem of coordination might happen at a bigger scale. Let us give an example on Figure 4.16. Let us take a set of m nodes $w_1, w_2, \ldots, w_m$, all at the same level k (such that for any $1 \leq i \leq m$, $L_{w_i} = k$), all with relaxed parents, and such that for any $1 \leq i \leq m - 1$, $d(w_i, w_{i+1}) < \gamma \cdot b^{k+1}$. In the high mobility setting, this may happen if all nodes go away from their true parents at the same time step as represented by the red arrows on Figure 4.16. In the low mobility setting, this may happen too if the nodes are promoted only a constant number of times per time step (with w_1 starting at level k, w_2 starting at level $k - 1$, and so on, even if the nodes move one after the other, the situation of Figure 4.16 can appear).

On Figure 4.16, deciding which nodes among $w_1, \ldots, w_6$ to promote is equivalent to computing a maximal independent set (MIS)[16]: let us call P the set of nodes with relaxed parents at level k; if we create an undirected graph on P with edges between each node in P that are at distance $\gamma \cdot b^{k+1}$ from each other, then we may promote all nodes that are in an MIS in that graph.

Computing MISs in distributed networks is a classical problem. Without a node to centralize the computation, no algorithm seems to exist to compute MISs in $O(1)$ communication rounds. For example, Luby's algorithm needs $O(\log n)$ rounds with high probability [97].

In the next section, we will see that our algorithm $\mathcal{A}_{cnn}$ for constrained navigating nets can be

[16] An MIS is a subset of the nodes in a graph such that no two nodes in the MIS are adjacent in the graph, and such that any node of the graph is adjacent to at least on node in the MIS.

adapted to synchronous networks in the low mobility setting, such that only a constant number of communication rounds are needed per time step. The speed limit for the nodes is the same as previously, that is, $d_{mv} = 1$, which does not depend on the aspect ratio Φ, and thus avoids the trade-off between movement speed and needed number of communication rounds per time step as explained previously. Also, no MIS needs to be computed, as the set of nodes that may change their local data structure at a time step t_i are all adjacent in the connection graph $G(t_i)$ to the node u that moved at t_i, so that all computations can be centralized on u.

4.4.3 Adapting $\mathcal{A}_{cnn}$ to the Distributed Setting: $\mathcal{A}_{cnndist}$

In this section, we show that $\mathcal{A}_{cnn}$ can be adapted to synchronous distributed networks in the low mobility setting, so as to use only a constant number of communication rounds per time step (Theorem 4.62), while using $O(n)$ local memory space for a node at worst, but $O(n)$ memory in total for all nodes. We denote by $\mathcal{A}_{cnndist}$ the distributed algorithm.

As seen in Section 4.3.6, the cost for a node u to check whether $L_{\alpha a(u)} < L_{\gamma a(u)}$ can be high if it is necessary to search for $\alpha a(u)$ and $\gamma a(u)$. If this has to be done in a synchronous network, it may take $O(\log \Phi)$ communication rounds. To avoid this cost, we will do in a similar way as with $\mathcal{A}_{cnnptr}$: each node u will add to its local data structure a pointer $pointer_u$ to the node that may take a long time to retrieve. This time however, $pointer_u$ is not reset at each time step, and a consistent definition can be given.

Definition 4.55. *$pointer_u = (v, k)$, where v is either the γ- or α-ancestor of u, whichever is lowest in the hierarchy, and k is the level of the node that is both a child of v and an ancestor of u.*

As explained in Section 4.3.6, the value of k is useful to compute whether v is the γ-ancestor or the α-ancestor of u: v is the γ-ancestor of u iff $L_v \geq k + 2$ or $d(v, f_v) < \gamma b^{L_v+1}$; v is the α-ancestor of u iff $L_v = k + 1$ and $\beta b^{L_v+1} \leq d(v, f_v)$. Similarly to the modifications to $\mathcal{A}_{cnn}$ explained in Algorithm 18, p. 159, to take into account the values maintained in $pointer_u$, the changes outlined in Algorithm 19 should be applied in $\mathcal{A}_{cnndist}$.

Algorithm 19 $\mathcal{A}_{cnndist}$: changes to use $pointer_u$.

1: **replace lines 58 through 60 of Algorithm 16 with the following:**
2: $(anc, k) \leftarrow pointer_u$
3: **if** $L_{anc} = k + 1$ and RoundDistance(anc, f_{anc}, L_{anc}) $= \alpha$ **then**
4:     **if** $\exists (f', L_{anc} + 1) \in N_{f_{anc}} : d(anc, f') \leq \gamma b^{L_{anc}+1}$ **then**
5:     Redirect(anc, f_{anc}, f') **else** Promote(anc, f_{anc}) **end if**

6: **replace line 78 of Algorithm 16 with the following:**
7: **if** RoundDistance(v, $newf$, $L_v + 1$) $= \gamma$ (resp. β) **then**
8:     $\tau' \leftarrow \gamma$ (resp. β)
9: **else** (*RoundDistance(v, $newf$, $L_v + 1$) $= \alpha$*)
10:     $(anc, k) \leftarrow pointer_v$
11:     **if** $L_{anc} = k + 1$ and RoundDistance(anc, f_{anc}, L_{anc}) $= \alpha$ **then** $\tau' \leftarrow \beta$ **else** $\tau' \leftarrow \alpha$ **end if**

A node thus has access at all time to the lowest node among its α-ancestor and γ-ancestor without needing additional communication rounds.

Always maintaining the values of those pointers adds a difficulty however: a change of distance between a node v and its parent may change $pointer_w$ for each node w that is a descendant of v. An example can be seen on Figure 4.17: if $pointer_w = u$ for several nodes w,

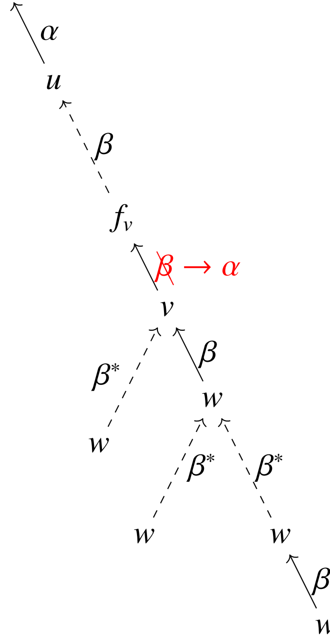


Figure 4.17 – Graphical representation of the movements of a node that could change the α-ancestor for many nodes.

and $d(v, f_v, t_{i-1}) < \beta \cdot b^{L_v+1}$ and $d(v, f_v, t_i) \geq \beta \cdot b^{L_v+1}$ for a node v that is both ancestor of the nodes w and descendant of u, then all the nodes w must change their α-ancestor, which at time t_i becomes v. Maintaining the pointer of each node may thus cost up to $O(\log \Phi)$ communication rounds per time step, as it may be needed to go down all the hierarchy to get all descendants that changed their ancestors.

To avoid this problem, each node u maintains in addition the (sometimes empty) list of all nodes w such that $pointer_w = u$. We will denote that list by ST_u (because it is a subtree of the navigating net). Thus, if for example v becomes the α-ancestor of some of its descendants, the nodes that need to change their α-ancestor are all in $ST_{\alpha a(v)} \cup ST_{\gamma a(v)}$, and may thus be retrieved in a constant number of communication rounds.

Definition 4.56. $ST_u = \{v \in \mathcal{V} : pointer_v = u\}$

The local data structure of a node u thus contains:

- L_u the highest level of u,
- f_u the parent of u,
- C_u the set of children of u,
- N_u the set of neighbors of u,
- $pointer_u$, the pointer given by Definition 4.55,
- ST_u the set of nodes v such that $pointer_v = u$.

The main idea of $\mathcal{A}_{cnndist}$ is to centralize at each time step t_i all computations on u, the node that moved at time t_i. The node u is thus in charge of retrieving all information on the nodes that will have to change their local data structure, and then telling them those changes.

The description of $\mathcal{A}_{cnndist}$ is similar to the centralized $\mathcal{A}_{cnn}$ (see Algorithms 14, 15, and 16), except for the main loop that needs to change in order to explicitly centralize the operations on u, and handle the maintenance of $pointer_v$ and ST_v for any node v. This is described in Algorithm 20. We call *neighboring graph* the graph on the nodes in $\mathcal{V}$, with the edges defined by the parent, children, neighbors, and pointers of the nodes (but not the subtrees ST). We say that u and v are at distance x in the neighbouring graph, if the number of edges needed to go from u to v in the neighboring graph is smaller or equal to x.

Algorithm 20 $\mathcal{A}_{cnndist}$, from the point of view of a node u.

1: Main loop:
2: **for each** time t_i where u moves **do**
3: Get the positions and data structure of all nodes at distance 10 from u in the neighboring graph.
4: Resolve locally all false certificates of $Cert$, according to the instructions of algorithms 15 and 16, using the retrieved positions:
5: first `Separation`, in order of decreasing level,
6: then `ShortEdge` and `LongEdge`, in order of decreasing level,
7: and finally `Cover` and `SCover` certificates, in order of decreasing level.
8: Each time a $\texttt{Cover}_{\beta,L}(u', v')$ fails, for each node $w \in ST_{\alpha a(u')} \cup ST_{\gamma a(u')}$ that is a descendant of u' or u' itself, $pointer_w \leftarrow (u', L-1)$, $ST_{\alpha a(u')} \leftarrow ST_{\alpha a(u')} \setminus w$, and $ST_{\gamma a(u')} \leftarrow ST_{\gamma a(u')} \setminus w$.
9: Each time a $\texttt{SCover}_{\gamma,L}(u', v')$ fails, or a node u' gets promoted from level L to level $L+1$, for each node $w \in ST_{\alpha a(u')} \cup ST_{\gamma a(u')}$ that is a descendant of u' or u' itself, $pointer_w \leftarrow (u', L-1)$, $ST_{\alpha a(u')} \leftarrow ST_{\alpha a(u')} \setminus w$, and $ST_{\gamma a(u')} \leftarrow ST_{\gamma a(u')} \setminus w$.
10: Send to each node the changes to its local data structure, and to its certificate list. (*This takes one communication round.*)
11: **end for**

As usual, we have to prove that $\mathcal{A}_{cnndist}$ is valid. For this, we need to prove that u retrieves all information needed for the updates of all certificates that fail at time t_i. With u the node that moved, we thus need to prove that all nodes affected by $\mathcal{A}_{cnn}$ are at distance 10 from u in the neighboring graph, which we will do in Lemma 4.60.

First, let us prove an upper bound on the distance from u of the nodes that are affected by a correction of `Separation` certificates.

Remark 4.57. *When u moves at time t_i, and a* `Separation` *certificate at level L fails, the nodes that change their local data structure or their certificates list in $\mathcal{A}_{cnn}$ are at distance 3 from u in the neighboring graph.*

Proof. As u is the only node that moves, all failing `Separation` certificates involve u. Let us call v another node that fails that certificate (that is a $\texttt{Separation}_{\gamma,k}(u, v)$ fails).

The only nodes that may change their local data structure or their certificates list are:

- u and v;
- neighbors of u and v at level L (on Algorithm 15 p.137, line 43);
- children of u and v at level $L-1$ (on Algorithm 15, because of line 43, we execute line 26);
- nodes that are children at level $L-1$ of neighbors at level L of u or v (as u or v may change their children on line 44, which gets line 31 executed).

As the distance between u and v in the neighboring graph is 1, all those nodes are at distance at most 3 from u. □

A similar result can be proven for `ShortEdge` and `LongEdge` certificates.

Remark 4.58. *When u moves at time t_i, and a* `ShortEdge` *or* `LongEdge` *certificate at level L fails, the nodes that change their local data structure or their certificates list in $\mathcal{A}_{cnn}$ are at distance 2 from u in the neighboring graph.*

Proof. Let v be a node that fails a `ShortEdge`(u, v) or `LongEdge`(u, v) certificate with u.

The only nodes that may change their local data structure or their certificates list are u and v, and children of u and v at level $L - 1$ (as on Algorithm 15, because of line 48 and 51, we execute line 26). Among those, the farthest from u are the children of v, which are at distance 2 from u. □

For `Cover` certificates, we may see that the only changes to the data structure and to the certificates list happen through promotions of the nodes. Let us see that a single promotion affects nodes that are close in the neighboring graph to the node that gets promoted.

Remark 4.59. *When a node v is promoted, the nodes that change their local data structure or their certificates list in $\mathcal{A}_{cnn}$ were at distance 5 from v in the neighboring graph just before the promotion.*

Proof. When v gets promoted, similarly to what was described in the proof of Remark 4.57, changes are made to the data structure between lines 81 and 87 of Algorithm 16. In particular, on line 85, nodes are added to v's neighborhood, which triggers line 26 of Algorithm 15. Line 26 may affect a child of a child of a neighbor of $newf$, and as $newf$ was the parent of the parent of v before the promotion, that node is at distance 5 from v. □

Finally, we can prove Lemma 4.60.

Lemma 4.60. *When u moves at time t_i, the nodes that change their local data structure or their certificates list in $\mathcal{A}_{cnn}$ are at distance 10 from u in the neighboring graph at time t_i.*

Proof. By Remark 4.57 and Remark 4.58, all nodes affected by `Separation`, `ShortEdge` and `LongEdge` certificates are at distance less than 3 from u.

Thanks to Lemma 4.38 (p.150), Lemma 4.39, and Lemma 4.41, we have that a node v may not be promoted more than 4 times between t_i and t_{i+1}. As at each promotion, v gets as new parent the node that was previously the parent of its parent (which was at distance 1 from v), each promotion affects nodes that were one step further away from v at time t_i. By Remark 4.59, the node affected by these promotions are thus at distance 8 from v at time t_i. As the first of these promotions may occur on line 39 of Algorithm 15 (p.137), on a node w that is a child of a node that fails a `Separation` certificate with u, the nodes affected by promotions between t_i and t_{i+1} are at distance 10 from u at time t_i. This is represented on Figure 4.18: w is at distance 2 from u. In the worst case, w may get promoted 4 times, and we know thanks to Remark 4.59 that the affected nodes, like x on Figure 4.18, are at most at distance 5 away from w just before its last promotion (represented in blue on the figure). We may thus see that x is at most at distance 10 away from u.

As we may see on Algorithm 16 that the only changes that happen when treating failing `Cover` and `SCover` certificates are through promotions, we get that all updates that happen in $\mathcal{A}_{cnn}$ affect nodes at distance less than 10 from u. □

We can then easily deduce from Lemma 4.60 that $\mathcal{A}_{cnndist}$ is valid in Theorem 4.61.

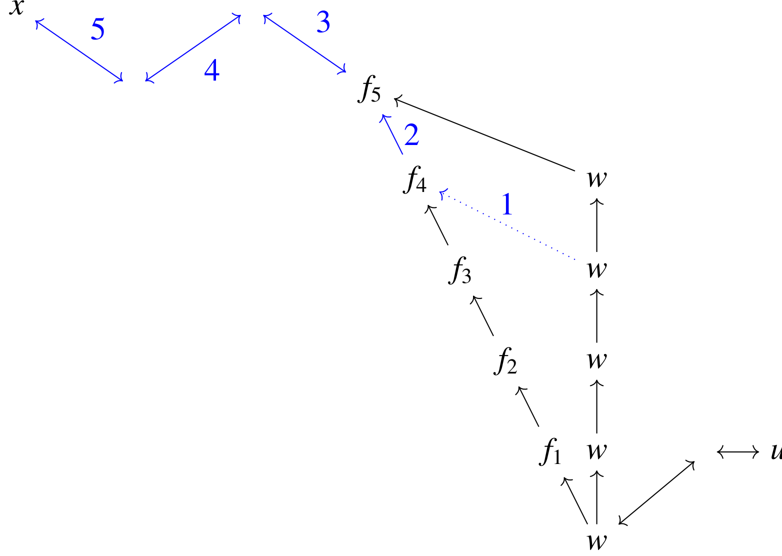


Figure 4.18 – Graphical representation of a node x that could be at distance 10 from u.

Theorem 4.61. *With synchronous distributed networks in the low mobility setting, assuming that the structure is correctly initialized, that is, $G(0) \in \mathcal{G}(0)$, and that all certificates of Table 4.3 are in Cert at time 0, then when using $\mathcal{A}_{cnndist}$, $\forall i \geq 0, G(t_{i+1}) \in \mathcal{G}(t_i)$.*

Proof. As all computations are centralized on u, the node that moved at time t_i, as those computations are identical to the centralized $\mathcal{A}_{cnn}$, and as by Lemma 4.60, u retrieves all needed information about the nodes that are affected by the changes at time t_i (on line 3 of Algorithm 20), we get the result by Theorem 4.25 (p.139). □

In terms of performance, as said previously, the algorithm needs a constant number of communication rounds per time step.

Theorem 4.62. *$\mathcal{A}_{cnndist}$ uses a constant amount of communication rounds per time step.*

Proof. This is trivial: as 10 communication rounds are needed on line 3 of Algorithm 20, and one additional round is needed on line 10, the total usage of communication rounds is 11 per time step. □

Let us now look at the memory usage. We show in Theorem 4.63 that the data structure (presented on page 167) takes $O(n)$ memory space per node, but that the amortized memory cost of the data structure is constant.

Theorem 4.63. *When ρ is constant, the distributed data structure of $\mathcal{A}_{cnndist}$ uses $O(n)$ memory space for each node, but the total cost for all nodes is $O(n)$.*

Proof. We have proven in Lemma 4.43 (p.155) that the centralized data structure takes $O(\log \Phi)$ per node. However, a node may be neighbor of another node at only one level by Lemma 4.19 (p.127), and similarly, a node may child only at one level. Thus, we have that for a node u, the memory cost of $\{L_u\} \cup \{f_u\} \cup C_u \cup N_u$ is $O(n)$.

In $\mathcal{A}_{cnndist}$, a node u adds $pointer_u$ to its local data structure, which takes a constant amount of memory space, and ST_u, which may, at the worst case, take up to $O(n)$ space, in the case where all nodes point to u. Thus, the data structure takes $O(n)$ memory space for each node.

As each node points only to one node with $pointer_u$, the total memory space of $\bigcup_{u\in\mathcal{V}} ST_u$ is $O(n)$. As we have seen in Theorem 4.47 (p.157) that the total size of the centralized structure is $O(n)$, we get that all the elements of the distributed structure other than ST_u, also take $O(n)$. □

Now that we showed the memory cost of the structure itself, it remains to be proven that $\mathcal{A}_{cnndist}$ doesn't need too much additional memory when updating that structure. We thus want to measure the memory used by the node u in Algorithm 20.

Theorem 4.64. *When ρ is constant, the memory usage of $\mathcal{A}_{cnndist}$ is $O(n)$.*

Proof. We have shown in Theorem 4.63 that the total cost of the data structure is $O(n)$. We have also seen through Remark 4.57, Remark 4.58, and Remark 4.59 that when a certificate fails, the affected nodes are close to u, and the sets of neighbors and children are affected only at specific levels: when u retrieves the local data structure of a node v because there is a certificate between u and v at level L, then u needs to store v's neighbors and children only at levels L' such that $|L - L'|$ is constant. Thus the usage of memory for u in Algorithm 20 is $O(n)$. □

To finish our analysis, let us recall that in most practical settings, it is costly to track in real time the positions of other nodes. While the data structure of $\mathcal{A}_{cnndist}$ uses up to $O(n)$ memory space per node, the number of positions a node needs to track is actually lower. Indeed, we may notice that $\mathcal{A}_{cnndist}$ can be executed when a node u knows only the parent/child relations of the nodes in ST_u, so that u does not need to track the positions of the nodes in ST_u.

Theorem 4.65. *When ρ is constant, the number of positions a node needs to track may be reduced to $O(\log \Phi)$.*

Proof. In Algorithm 20, the values of the subtrees ST are used only on lines 8 and 9, to find the descendants of a node u' that are in $ST_{\alpha a(u')} \cup ST_{\gamma a(u')}$. As $u' \in ST_{\alpha a(u')} \cup ST_{\gamma a(u')}$, the node u that centralizes the information may compute those descendants knowing only the parent/child relations of the nodes in $ST_{\alpha a(u')} \cup ST_{\gamma a(u')}$. As the memory cost of the local data structure of a node w without ST_w is $O(\log \Phi)$, we get our result. □

4.5 Conclusion and Possible Extensions

We have given in this Chapter a general definition of navigating nets, and an extension with interesting properties that we call constrained navigating nets.

We have given an algorithm, $\mathcal{A}_{cnnptr}$, to maintain constrained navigating nets under movements in the Black-Box model and in the low mobility setting, and have proven that this algorithm takes $O(n)$ memory space, and that during updates, a node may not change its level more than a constant number of times at each time step. The update time is $O(\log \Phi)$ computations per time step, as with a DefSpanner, a structure presented in [55].

We have also given a distributed algorithm for synchronous networks that maintains constrained navigating net in the low mobility setting. The algorithms needs a constant number of communication rounds per time step, and it uses $O(n)$ memory space for nodes at worst, but $O(n)$ in total (which is constant on average).

Our work opens several perspectives. First, the most immediate question is how our techniques would perform in the high mobility setting. We believe that it is possible to adapt $\mathcal{A}_{cnn}$ to the high mobility setting, achieving interesting update costs per time step. This is the subject of the discussions in Section 4.5.1. For the distributed setting, the main drawback of $\mathcal{A}_{cnndist}$ is its worst-case memory cost per node. It could be interesting to see if one could reduce that memory cost without affecting too much the needed number of communication rounds per time step.

More generally, it seems that it is still needed to look for different techniques. After the foundations of [55] to maintain navigating nets, the main addition of [56] was to allow some

nodes to have relaxed parents, that can be repaired later in parallel. Here, the two main extensions, are the ability to avoid nodes to be neighbors at more than one level, and the property of ancestor invariant. One of the main drawbacks of our method is the intricacy of the algorithms, which makes them difficult to understand. In practice, simpler algorithms, even if slightly less cost efficient are often preferred. Thus it would be still interesting to find different techniques using other concepts to maintain navigating nets, or other structures to answer efficiently to the $\text{CloseNodes}_u(r)$ query.

4.5.1 Adapting $\mathcal{A}_{cnn}$ to the High Mobility Setting

An immediate way to adapt $\mathcal{A}_{cnn}$ to the (centralized) high mobility setting is to take in consideration the movements of the nodes one after the other, in an arbitrary order, and run $\mathcal{A}_{cnnptr}$ on these single movements as in the low mobility setting. As there are n nodes, and as we have seen (Theorem 4.52) that $\mathcal{A}_{cnnptr}$ takes $O(\log \Phi)$ time to update the structure in the low mobility setting, this leads to a total update cost of $O(n \log \Phi)$.

We believe however that it is possible to adapt $\mathcal{A}_{cnn}$ to the high mobility setting, so as to obtain an update time of $O(n)$ per time step, and thus improving the performance of DefSpanners [55] (that need $O(n \log \Phi)$ computations per time step). This result would be optimal: as all nodes may move at each time step, maintaining the structure is necessarily $\Omega(n)$.

As for $\mathcal{A}_{cnnptr}$ and $\mathcal{A}_{cnndist}$, to enable the nodes to find their α- and γ-ancestors in $O(1)$ computations, we store for each node u a pointer $pointer_u$ to the lowest of these ancestors, as given in Definition 4.55 (p.166). Here however (see Algorithm 21), the values of the pointers are updated lazily in each time step; while we conjecture that at the end of the updates of each time step, all $pointer_u$ comply to Definition 4.55, there may be some instant $t \in [t_i; t_{i+1}[$ where some $pointer_u$ is not exactly true (see Conjecture 4.69).

In the data structure, we thus have for each node u:

- L_u the highest level of u,
- f_u the parent of u,
- C_u the set of children of u,
- N_u the set of neighbors of u,
- $pointer_u = (v, k)$, the pointer to an ancestor v, and k the level of the node that is both a child of v and an ancestor of u.

The list of certificates is the same as for $\mathcal{A}_{cnn}$ (see Section 4.3.3, p.131), but we add the two following certificate:

- $\texttt{PointerEdge}_k(u, v)$:

$$\begin{cases} pointer_u = (v, k) & \text{if } L_v \geq k+2 \vee d(v, f_v) < \gamma \cdot b^{k+2} \\ & \text{or } L_v = k+1 \wedge \beta \cdot b^{k+2} \leq d(v, f_v) < \alpha \cdot b^{k+2} \\ \gamma a(u) = \gamma a(v) & \text{otherwise} \end{cases}$$

  for $v = f_u$ and $k = L_u$, and

When a failing $\texttt{PointerEdge}_*(u, v)$ certificate is treated, u needs either to take v as new ancestor (if v is either the γ- or α-ancestor of u), or to copy v's ancestor. Also, each time a node

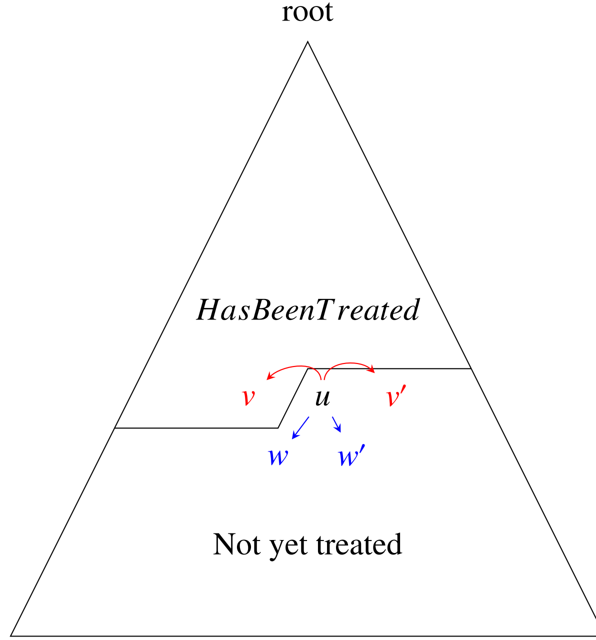


Figure 4.19 – Graphical representation of $HasBeenTreated$. u, the node currently being treated, may have neighbors (in red on the figure) that are in $HasBeenTreated$ (as v) and others that are not yet in $HasBeenTreated$ (as v'). However, none of its children (w, and w', in blue on the figure) are in $HasBeenTreated$.

is promoted, it would obtain its new pointers in a similar way, by copying the pointer of its new parent if that new parent is not the γ- or α-ancestor.

To take into account the values of those pointers, as in $\mathcal{A}_{cnndist}$, the changes outlined in Algorithm 19 (p. 166) should also be used in the high mobility setting.

In the high mobility setting, the algorithm moves the nodes one after the other, and corrects all failing certificates after each movement, except for `PointerEdge` certificates, that are treated lazily (see line 8 of Algorithm 21). The instructions to treat other failing certificates are the same as for $\mathcal{A}_{cnn}$.

In order to know which nodes have already been moved at each time step, we maintain two sets of nodes.

Definition 4.66. *The set $HasBeenTreated$ contains all nodes that have already been moved during the current time step.*

When there is a failure of `Separation` certificates, some nodes that did not move yet may decrease their level, the second set covers that case.

Definition 4.67. *The set HasBeenDemoted contains all nodes that have decreased their level, but that have not moved yet during the current time step.*

A graphical representation of the order of execution and of the set $HasBeenTreated$is given in Figure 4.19. These two sets are then used to define the order $\prec$ of the nodes for the correction of `Separation` certificates.

Definition 4.68. *We define priority as such (with ∞^- a value strictly higher than any other value except ∞) :*

$$priority(u) = \begin{cases} \infty & \text{if } u \in HasBeenTreated \\ \infty^- & \text{if } u \in HasBeenDemoted \\ L_u & \text{otherwise} \end{cases} \tag{4.7}$$

We define $u \prec v$ if and only if $priority(u) \leq priority(v)$.

This definition of $\prec$ prevents a node from decreasing its level several times without moving itself. Also, as nodes are moved in order of decreasing levels, it ensures that the conditions of Theorem 4.37 (p.150) are respected.

The procedure to execute at each time step is described in Algorithm 21.

Algorithm 21 Adaptation of $\mathcal{A}_{cnn}$ to the high mobility setting

1: At each time step t_i, do the following:
2: **for each** $u \in \mathcal{V}$ **do** $p_u \leftarrow p_u(t_{i-1})$ **end for**
3: $HasBeenTreated \leftarrow \emptyset$; $HasBeenDemoted \leftarrow \emptyset$
4: **while** $HasBeenTreated \neq \mathcal{V}$ **do**
5: Take a node $u \notin HasBeenTreated$ that maximises $priority(u)$
6: $p_u \leftarrow p_u(t_i)$
7: Correct false certificates (following instructions from algorithms 15 and 16) in the following order:
8: All `PointerEdge`$_k$ certificates such that $k \geq L_u$, and all `PointerEdge`$_k(*, u)$ certificates. (*We ensure all ancestor pointers are correct at levels above current level, and for all children of u. All other pointers will be updated later.*)
9: All `Separation` certificates
10: All `ShortEdge` and `LongEdge` certificates
11: All `Cover` and `SCover` certificates
12: **for each** $v \in \mathcal{V}$ that decreased its level **do** add v to $HasBeenDemoted$ **end for**
13: add u to $HasBeenTreated$ and remove u from $HasBeenDemoted$

14: **when** `PointerEdge`$_k(u, v)$ fails **do** (*Contrary to other certificates, `PointerEdge` certificates are not removed when failing.*)
15: **if** $L_v \geq k + 2 \lor d(v, f_v) < \gamma \cdot b^{k+2} \lor (L_v = k + 1 \land \beta \cdot b^{k+2} \leq d(v, f_v) < \alpha \cdot b^{k+2})$ **then**
16: $pointer_u \leftarrow (v, L_u)$
17: **else**
18: $pointer_u \leftarrow pointer_v$
19: **end when**

20: Each time a node u gets a new parent $newf$, remove any `PointerEdge`$_*(u, *)$ certificate, and create a `PointerEdge`$_{L_u}(u, newf)$ certificate. (*That certificate will fail immediately, and by the instructions above, $pointer_u$ will be set in $O(1)$ computations.*)

To get our result in the high mobility setting, it remains to prove that Algorithm 21 maintains a valid constrained navigating net, and that it does not need more than $O(n)$ computations per time step.

To prove the validity of the algorithm, as the validity of $\mathcal{A}_{cnn}$ has been proven in the low mobility setting, and as Algorithm 21 makes it so that each node moves one after the other, it remains only to be proven that the lazy update of the pointers to the α-ancestors and γ-ancestors is enough for the validity of the structure. This result is given in Conjecture 4.69, along with some arguments that could be used in the future to prove the conjecture.

Let us call *treated tree* the subgraph of the connection graph G induced by the nodes in $HasBeenTreated$. We say that the treated tree is valid if it complies to the properties of valid navigating nets (see Definition 4.24, p. 139), and so that for any $u \in HasBeenTreated$, $pointer_u$ complies to the definition of the pointers (see Definition 4.55, p.166).

Conjecture 4.69. *When using Algorithm 21 in the high mobility setting, each time a node u is treated and moved, when u is added to $HasBeenTreated$, the treated tree is valid, even if there were some invalid $pointer_v$ for nodes v such that $L_v < L_u$ during the computations.*

ideas for the proof. To prove this conjecture, it is needed to show that in all situations where an invalid $pointer_v$ is used, nothing "bad" happens to the structure.

Let $pointer_v = (anc, k)$. There are only three situations in which the value of anc is incorrect:

- (First case) $L_{\gamma a(v)} < L_{anc}$ (but $L_{anc} < L_{\alpha a(v)}$).
- (Second case) $L_{\alpha a(v)} < L_{anc}$ (but $L_{anc} < L_{\gamma a(v)}$).
- (Third case) Both of the above cases: $L_{\gamma a(v)} < L_{anc}$ and $L_{\alpha a(v)} < L_{anc}$.

First case: the correct value for anc should be $\gamma a(v)$, but $pointer_v$ actually points to another node higher in the hierarchy. In Algorithm 19, p.166, the value of $pointer_v$ is used only in two situations.

- Line 1 through 5. Here, the intent is to redirect or promote $\alpha a(v)$ if $L_{\alpha a(v)} < L_{\gamma a(v)}$. As we have $L_{\gamma a(v)} < L_{\alpha a(v)}$ by supposition, if RoundDistance(anc, f_{anc}, L_{anc}) is either γ or β, then the outcome is the same as if $pointer_v$ was valid. If RoundDistance(anc, f_{anc}, L_{anc}) is α, then anc gets redirected or promoted. This, however, may happen only in situation where $L_{anc} > L_u$ (recall that u is the node that moved), as line 1 through 5 are executed during a failure of `Cover` certificate, so that $f_v = u$. Thus $anc \in HasBeenTreated$, which means anc can be promoted, as the ancestor invariant is true for anc and its ancestors.
- The value of $pointer_v$ is also used lines 6 through 11. Again, if RoundDistance(anc, f_{anc}, L_{anc}) is either γ or β, then the outcome is the same as if $pointer_v$ was valid. If RoundDistance(anc, f_{anc}, L_{anc}) is α however, line 11 may lead to an erroneous creation, for a node w child of v, of a failing $\texttt{Cover}_{\beta,*}(w,*)$ certificate (as the value of τ' is used line 81 of Algorithm 16, p.138). In turn, when treating that failing $\texttt{Cover}_{\beta,*}(w,*)$ certificate, the node pointed by $pointer_w$ will get redirected or promoted. We need to prove that in this case, $pointer_w = anc$, which means subsequent calls to RoundDistance(anc, f_{anc}, L_{anc}) result in γ. However w also may get promoted, and it remains to be proven that this additional promotion does not cause other problems.

Second case: here, we have a node w such that $w = \alpha a(v)$, and $L_w < L_{anc}$. However, as the connection graph was valid at the end of the last time step, it means that previously, w was not the α-ancestor of v. Two ways come to mind as to how w may have become v's α-ancestor.

- The node w got farther away from its parent, breaking a $\texttt{Cover}_{\beta,L_w}(w, f_w)$ certificate. This resulted in the redirection or the promotion of w (line 63 of Algorithm 16), which contradicts the supposition that $w = \alpha a(v)$ (as by Remark 4.33 p. 144, and because a promotion increases w's level, w should have become v's γ-ancestor).
- The node w was promoted previously, getting as new parent a node $newf$ such that $\beta b^{L_w(+)1} \leq d(w, newf)$. However, as at that time $pointer_w$ was valid, this means that by line 11 of Algorithm 18 (p.159), a failing $\texttt{Cover}_{\beta,L_w}(w, f_w)$ was created, which again, would have caused w to be redirected or promoted, and contradicts the supposition that $w = \alpha a(v)$.

Third case: this case is similar to the two previous cases, depending on whether $L_{\alpha a(v)} < L_{\gamma a(v)}$ or $L_{\gamma a(v)} < L_{\alpha a(v)}$.

□

Concerning the performance of Algorithm 21, as we maintain $pointer_u$ for all node u, all corrections of certificates need only a constant time to finish. It remains to be proven that only a linear number of certificates may fail in total over all the n moving nodes.

For this, we may see that in Lemma 4.44 (p.155), the value of $\mathcal{O}(\log \Phi)$ stems from an upper bound on $\mathcal{O}(L_u)$. We can thus improve the result.

Lemma 4.70. *When node u moves, only $\mathcal{O}(L_u)$ certificates may fail.*

Proof. The proof is an immediate consequence of Lemma 4.43 (p.155), when noticing that $\mathcal{O}(\log \Phi)$ can be replaced by $\mathcal{O}(L_u)$. □

With Lemma 4.70, and as Algorithm 21 relies on moving the nodes one after the other, which is equivalent to several successive movements equivalent to the low mobility setting, we may get in Lemma 4.71 a result similar to Lemma 4.45 (p. 155).

Lemma 4.71. *When node u moves and the failing certificates are treated, only $\mathcal{O}(L_u)$ failing certificates need to be treated in total.*

As by Theorem 4.47, we have for each instant t, $\sum_{u \in \mathcal{V}} L_u(t) = \mathcal{O}(n)$, we then would like to use Lemma 4.71 to show that, over all $\mathcal{O}(n)$ movements at a time step t_i, only $\mathcal{O}(n)$ failing certificates are treated. As each certificate needs constant time to be corrected, we would get the result of a total update time of $\mathcal{O}(n)$ computations per time step in the high mobility setting.

However, one final hurdle stands in the way: while Theorem 4.47 is true at each moment in time, Lemma 4.71 is about the level the node that moves has *when it is moving*, and as the nodes move one after the other, a slightly stronger result is needed.

Conjecture 4.72. *Let $u_h \in \mathcal{V}$, and let $m_h \in [t_i; t_{i+1}[$ be the moment at which u_h is moved in Algorithm 21. We need to prove that $\sum_{u_h \in \mathcal{V}} L_{u_h}(m_h) = \mathcal{O}(n)$.*

ideas for the proof. We propose to prove that between t_i and t_{i+1}, a node can decrease or increase its level only a constant number of times. We would thus have that for any time $t \in [t_i, t_{i+1}[$, and for any node u_h, the difference between $L_{u_h}(t)$ and $L_{u_h}(t_i)$ is constant, so that by Theorem 4.47, we would get $\sum_{u_h \in \mathcal{V}} L_u(m_h) = \sum_{u_h \in \mathcal{V}} (L_{u_h}(t_i) + \mathcal{O}(1)) = \mathcal{O}(n)$.

First, it is easy to show that once a node is in *HasBeenDemoted*, it cannot fail a `Separation` certificate with another node from *HasBeenDemoted* (as *HasBeenDemoted* contains all nodes that decreased their level once, and as the movement speed is too low for two nodes to fail `Separation` certificates on two levels during the same time step). Thus a node can decrease its level only once between t_i and t_{i+1}.

Concerning the increase of levels, we can see that a node u can increase its level only because of three reasons.

The first reason is that u or its current parent moves. As u moves only once, that movement may cause only a constant number of promotions of u by Theorem 4.37 (p.150). We may then prove that u can fail a *Cov* certificate because of the movement of is current parent f_u only once: if that movement imposes the promotion of u, than either $L_u \leq L_{f_u}$, so that u keeps the same parent (which cannot move a second time), or it gets as new parent a node $newf$ with $L_{newf} \geq L_{f_u} + 1$, and thus $newf \in HasBeenTreated$, as nodes are added to $HasBeenTreated$ in order of decreasing level.

Another reason is the promotion of u because u is the α-ancestor of another node. Let us note with k the level of u before promotion. After promotion, we have that $L_u = k + 1$, but that u does not have any children at level k. Thus, for u to become the α-ancestor of a node again, it would require u to get a new child v at level k through promotion, such that $d(u, v) \geq \beta \cdot b^{k+1}$.

Thus, v must have been a descendant of u at time t_i. However, by Lemma 4.12, we have $d(u, v) < \left(\alpha + \frac{1}{b}\right) b^k < \beta \cdot b^{k+1}$.

Finally, u with $L_u = k$ can increase its level because its parent f_u fails a `Separation` certificate and decreases its level. If $L_{f_u} > k + 1$, then by Lemma 4.19 (p.127) and by the rules of priority, f_u does not decrease its level. For u to increase its level more than a m times through failures of `Separation` certificates, it would require for each of its m first ancestors to fail a `Separation` certificate with a different node. All of those nodes would need to move before u moves, and thus, these nodes must be at a level higher than $k + m$, which, if m is high enough, implies that all these nodes do not respect the γ-separation.

As we have seen with different arguments that a node may change its level through the same cause only a constant number of times, the main element that remains to be proven is that those causes may not be combined to changes a node's level more often. □

While some technical details need to be correctly proven, we explained in this section the broad ideas that could be used to prove that constrained navigating nets could be maintained with $\mathcal{O}(n)$ computations per time step in the high mobility setting.

Chapter 5

Conclusion

In this thesis we presented algorithms to allow moving nodes to answer to queries related to their distances.

In Chapter 2, we presented $\mathcal{A}_{lc}$, a synchronized distributed algorithm allowing two connected nodes to estimate the distance between them while having a guarantee on the error. We then showed that $\mathcal{A}_{lc}$ is optimal in terms of number of message exchanges up to a constant factor, when the nodes follow some random movement patterns. These results are complemented with experiments, showing that $\mathcal{A}_{lc}$ sends less messages in practice than a simple strategy that is often used in online video games, consisting in sending message at regular time intervals.

We then focused our attention on the CloseNodes in the Black-Box model, where time steps can be identified, so that nodes move only at each time step, and not more than d_{mv} distance units away from their previous position.

Definition 1.2 (CloseNodes$_u(r)$)**.** *Given a node* $u \in \mathcal{V}$ *and a distance* $r \in \mathbb{R}$*, return all nodes* $v \in \mathcal{V}$ *such that* $d(u, v) \leq r$*.*

In Chapter 3, we presented $\mathcal{A}_{f1d}$, a synchronized distributed algorithm that allows nodes with positions in one dimension, to answer to the CloseNodes(r) query when the value r is fixed and known by the algorithm, and such that $r \geq d_{mv}$. Nodes locally maintain the result of the query so that it can be answered on $O(1)$ time. $\mathcal{A}_{f1d}$ needs only a constant number of communication rounds per time step, and the local memory usage for each nodes is $O(b_{max})$, where b_{max} is the maximum number of nodes that can be at distance r from a node, which is optimal for solutions that maintain the result of the query.

Finally, in Chapter 4, we presented constrained navigating nets, a special case of navigating nets that have interesting properties. We showed that navigating nets can be used to answer to the CloseNodes$_u(r)$ query in $O(\log r + k)$ computations, where k is the number of nodes returned by the query. We gave $\mathcal{A}_{cnnptr}$, a centralized algorithm to maintain constrained navigating nets for movements in the low mobility setting. Our algorithm uses $O(n)$ space. Updates need $O(\log \Phi)$ computations per time step, which is competitive with the DefSpanners from [55]. We also presented $\mathcal{A}_{cnndist}$, a synchronized algorithm that maintains distributed constrained navigating nets in the low mobility setting, using only a constant amount of communication rounds per time step. The memory cost of $\mathcal{A}_{cnndist}$ is $O(n)$ per node at worst, but each nodes needs to track the position of only $O(\log \Phi)$ other nodes, and the memory cost for all the nodes is $O(n)$ in total.

We have seen that our work opens up several perspectives.

With additional work, we could perhaps improve the constants of Chapter 2, and show that the number of messages compares even better to the optimal. We could also perhaps devise a more sophisticated algorithm, in which nodes reduce their messages by exchanging information

with more than one node. Also, it could be interesting to see if our algorithm can be used to maintain other values than the distance, like number of hit points in a video games, or remaining battery power in ad-hoc networks.

Concerning the results of Chapter 3, we could try to drop the limitation imposing that $r \geq 2d_{mv}$, as this could help in settings where the maximal speed of a node is very high, but where the nodes usually move at a much lower speed (for example when characters of a video game may teleport). Using $\mathcal{A}_{f1d}$ as a basis for settings where the positions of the nodes are in higher dimensions seems difficult however, and we believe that the results of Chapter 4 are better suited for that.

Finally, the results of Chapter 4 prove particularly interesting for future work. We believe that little work remains to adapt our algorithm to the high mobility setting, so as to get an update time of $O(n)$ computations per time step, which is better than the $O(n \log \Phi)$ computations per time step from [55]. Another direction for future work could also be to implement $\mathcal{A}_{cnnptr}$ and $\mathcal{A}_{cnndist}$ and run simulations to measure the performance of our solution in practice. We already implemented a previous version of $\mathcal{A}_{cnn}$[1], but some parts need to be adapted to the current version presented in Chapter 4.

Another interesting perspective would be to see how our different results may be combined. On one side, the results of Chapter 2 aim at reducing the number of exchanged message for two nodes that are connected, and on the other side, the algorithms from Chapter 3 and Chapter 4 aim at reducing the number of other nodes each node is connected to. These are two complementary approaches to reduce the global bandwidth usage in distributed settings.

In Chapter 4, the certificates are all based on distances: it could be interesting to study how much our distance estimation technique from Chapter 2 is usable for maintaining a constrained navigating net. As updates in $\mathcal{A}_{lc}$ may cause the estimated distances to change abruptly, we would have to study how using that algorithm relates to the speed limits required for the use of $\mathcal{A}_{cnn}$.

[1] Available at `https://github.com/KDS-NET`.

Bibliography

[1] Mohammad Ali Abam and Mark de Berg. "Kinetic Spanners in Rd". In: *Discrete & Computational Geometry* 45.4 (June 1, 2011), pp. 723–736. ISSN: 1432-0444. DOI: 10.1007/s00454-011-9343-y. URL: https://doi.org/10.1007/s00454-011-9343-y (visited on 10/15/2020).

[2] Mohammad Ali Abam, Mark de Berg, and Joachim Gudmundsson. "A simple and efficient kinetic spanner". In: *Computational Geometry*. Special Issue on 24th Annual Symposium on Computational Geometry (SoCG'08) 43.3 (Apr. 1, 2010), pp. 251–256. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2009.01.008. URL: http://www.sciencedirect.com/science/article/pii/S0925772109000601 (visited on 11/05/2020).

[3] Mohammad Ali Abam, Mark de Berg, and Bettina Speckmann. "Kinetic kd-Trees and Longest-Side kd-Trees". In: *SIAM Journal on Computing* 39.4 (Jan. 1, 2010). Publisher: Society for Industrial and Applied Mathematics, pp. 1219–1232. ISSN: 0097-5397. DOI: 10.1137/070710731. URL: https://epubs.siam.org/doi/10.1137/070710731 (visited on 10/26/2021).

[4] Pankaj K Agarwal, Lars Arge, and Jeff Erickson. "Indexing Moving Points". In: *Journal of Computer and System Sciences*. Special Issue on PODS 2000 66.1 (Feb. 1, 2003), pp. 207–243. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(02)00035-1. URL: https://www.sciencedirect.com/science/article/pii/S0022000002000351 (visited on 10/28/2021).

[5] Pankaj K. Agarwal. "Range Searching". In: *Handbook of Discrete and Computational Geometry*. 3rd ed. Num Pages: 36. Chapman and Hall/CRC, 2017. ISBN: 978-1-315-11960-1.

[6] Pankaj K. Agarwal, Jie Gao, Leonidas Guibas, Haim Kaplan, Vladlen Koltun, Natan Rubin, and Micha Sharir. "Kinetic stable Delaunay graphs". In: *Proceedings of the twenty-sixth annual symposium on Computational geometry*. SoCG '10. New York, NY, USA: Association for Computing Machinery, June 13, 2010, pp. 127–136. ISBN: 978-1-4503-0016-2. DOI: 10.1145/1810959.1810984. URL: https://doi.org/10.1145/1810959.1810984 (visited on 10/29/2021).

[7] Pankaj K. Agarwal, Jie Gao, and Leonidas J. Guibas. "Kinetic Medians and kd-Trees". In: *Algorithms — ESA 2002*. Ed. by Rolf Möhring and Rajeev Raman. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 5–17. ISBN: 978-3-540-45749-7. DOI: 10.1007/3-540-45749-6_5.

[8] Sudhir Aggarwal, Hemant Banavar, Amit Khandelwal, Sarit Mukherjee, and Sampath Rangarajan. "Accuracy in dead-reckoning based distributed multi-player games". In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. NetGames '04. New York, NY, USA: Association for Computing Machinery,

Aug. 30, 2004, pp. 161–165. ISBN: 978-1-58113-942-6. DOI: 10.1145/1016540.1016559. URL: https://doi.org/10.1145/1016540.1016559 (visited on 10/15/2020).

[9] Farhan Ahammed, Javid Taheri, Albert Y. Zomaya, and Max Ott. "VLOCI: Using Distance Measurements to Improve the Accuracy of Location Coordinates in GPS-Equipped VANETs". In: *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Ed. by Patrick Sénac, Max Ott, and Aruna Seneviratne. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Berlin, Heidelberg: Springer, 2012, pp. 149–161. ISBN: 978-3-642-29154-8. DOI: 10.1007/978-3-642-29154-8_13.

[10] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. "A survey of peer-to-peer content distribution technologies". In: *ACM Computing Surveys* 36.4 (Dec. 2004), pp. 335–371. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/1041680.1041681. URL: https://dl.acm.org/doi/10.1145/1041680.1041681 (visited on 10/18/2021).

[11] Vishal Kumar Arora, Vishal Sharma, and Monika Sachdeva. "A multiple pheromone ant colony optimization scheme for energy-efficient wireless sensor networks". In: *Soft Computing* 24.1 (Jan. 1, 2020), pp. 543–553. ISSN: 1433-7479. DOI: 10.1007/s00500-019-03933-4. URL: https://doi.org/10.1007/s00500-019-03933-4 (visited on 10/05/2021).

[12] Baruch Awerbuch. "Complexity of network synchronization". In: *Journal of the ACM* 32.4 (Oct. 1, 1985), pp. 804–823. ISSN: 0004-5411. DOI: 10.1145/4221.4227. URL: https://doi.org/10.1145/4221.4227 (visited on 10/04/2021).

[13] F. Baiardi, L. Ricci, and L. Genovali. "Improving Responsiveness By Locality In Distributed Virtual Environments". In: *ECMS 2007 Proceedings edited by: I. Zelinka, Z. Oplatkova, A. Orsoni*. 21st Conference on Modelling and Simulation. ECMS, June 4, 2007, pp. 195–200. ISBN: 978-0-9553018-2-7. DOI: 10.7148/2007-0195. URL: http://www.scs-europe.net/dlib/2007/2007-0195.htm (visited on 10/08/2021).

[14] Julien Basch, Leonidas J Guibas, and John Hershberger. "Data Structures for Mobile Data". In: *Journal of Algorithms* 31.1 (Apr. 1, 1999), pp. 1–28. ISSN: 0196-6774. DOI: 10.1006/jagm.1998.0988. URL: http://www.sciencedirect.com/science/article/pii/S0196677498909889 (visited on 10/15/2020).

[15] Julien Basch, Leonidas J. Guibas, and Li Zhang. "Proximity problems on moving points". In: *Proceedings of the thirteenth annual symposium on Computational geometry*. SCG '97. New York, NY, USA: Association for Computing Machinery, Aug. 1, 1997, pp. 344–351. ISBN: 978-0-89791-878-7. DOI: 10.1145/262839.262998. URL: https://doi.org/10.1145/262839.262998 (visited on 10/26/2021).

[16] Olivier Beaumont, Anne-Marie Kermarrec, Loris Marchal, and Etienne Riviere. "VoroNet: A scalable object network based on Voronoi tessellations". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007 IEEE International Parallel and Distributed Processing Symposium. ISSN: 1530-2075. Mar. 2007, pp. 1–10. DOI: 10.1109/IPDPS.2007.370210.

[17] Jon Louis Bentley. "Decomposable searching problems". In: *Information Processing Letters* 8.5 (June 11, 1979), pp. 244–251. ISSN: 0020-0190. DOI: 10.1016/0020-0190(79)90117-0. URL: https://www.sciencedirect.com/science/article/pii/0020019079901170 (visited on 11/04/2021).

[18] Jon Louis Bentley. “Multidimensional binary search trees used for associative searching”. In: *Communications of the ACM* 18.9 (Sept. 1, 1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: https://doi.org/10.1145/361002.361007 (visited on 11/04/2021).

[19] Mark de Berg, Marcel Roeloffzen, and Bettina Speckmann. “Kinetic convex hulls and delaunay triangulations in the black-box model”. In: *Proceedings of the twenty-seventh annual symposium on Computational geometry*. SoCG ’11. New York, NY, USA: Association for Computing Machinery, June 13, 2011, pp. 244–253. ISBN: 978-1-4503-0682-9. DOI: 10.1145/1998196.1998233. URL: https://doi.org/10.1145/1998196.1998233 (visited on 10/15/2020).

[20] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. “Donnybrook: enabling large-scale, high-speed, peer-to-peer games”. In: *ACM SIGCOMM Computer Communication Review* 38.4 (Aug. 17, 2008), pp. 389–400. ISSN: 0146-4833. DOI: 10.1145/1402946.1403002. URL: https://doi.org/10.1145/1402946.1403002 (visited on 06/24/2020).

[21] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. “Colyseus: a distributed architecture for online multiplayer games”. In: *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*. NSDI’06. USA: USENIX Association, May 8, 2006, p. 12. (Visited on 10/21/2021).

[22] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. “Mercury: supporting scalable multi-attribute range queries”. In: *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. SIGCOMM ’04. New York, NY, USA: Association for Computing Machinery, Aug. 30, 2004, pp. 353–366. ISBN: 978-1-58113-862-7. DOI: 10.1145/1015467.1015507. URL: https://doi.org/10.1145/1015467.1015507 (visited on 10/21/2021).

[23] Urs Bischoff, Martin Strohbach, Mike Hazas, and Gerd Kortuem. “Constraint-Based Distance Estimation in Ad-Hoc Wireless Sensor Networks”. In: *Wireless Sensor Networks*. Ed. by Kay Römer, Holger Karl, and Friedemann Mattern. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 54–68. ISBN: 978-3-540-32159-0. DOI: 10.1007/11669463_7.

[24] Nicolas Bonichon, Cyril Gavoille, Nicolas Hanusse, and David Ilcinkas. “Connections between Theta-Graphs, Delaunay Triangulations, and Orthogonal Surfaces”. In: *Graph Theoretic Concepts in Computer Science*. Ed. by Dimitrios M. Thilikos. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 266–278. ISBN: 978-3-642-16926-7. DOI: 10.1007/978-3-642-16926-7_25.

[25] Prosenjit Bose, Mirela Damian, Karim Douïeb, Joseph O’Rourke, Ben Seamone, Michiel Smid, and Stefanie Wuhrer. “$\pi/2$-Angle Yao Graphs Are Spanners”. In: *Algorithms and Computation*. International Symposium on Algorithms and Computation. Springer, Berlin, Heidelberg, Dec. 15, 2010, pp. 446–457. DOI: 10.1007/978-3-642-17514-5_38. URL: https://link.springer.com/chapter/10.1007/978-3-642-17514-5_38 (visited on 11/04/2021).

[26] Prosenjit Bose, Pat Morin, André van Renssen, and Sander Verdonschot. “The θ5-Graph is a Spanner”. In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Andreas Brandstädt, Klaus Jansen, and Rüdiger Reischuk. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 100–114. ISBN: 978-3-642-45043-3. DOI: 10.1007/978-3-642-45043-3_10.

[27] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. "Comparing interest management algorithms for massively multiplayer games". In: *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. NetGames '06. New York, NY, USA: Association for Computing Machinery, Oct. 30, 2006, 6–es. ISBN: 978-1-59593-589-2. DOI: 10.1145/1230040.1230069. URL: https://doi.org/10.1145/1230040.1230069 (visited on 10/08/2021).

[28] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. "Strict fibonacci heaps". In: *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. STOC '12. New York, NY, USA: Association for Computing Machinery, May 19, 2012, pp. 1177–1184. ISBN: 978-1-4503-1245-5. DOI: 10.1145/2213977.2214082. URL: https://doi.org/10.1145/2213977.2214082 (visited on 01/27/2022).

[29] Eliya Buyukkaya, Maha Abdallah, and Gwendal Simon. "A survey of peer-to-peer overlay approaches for networked virtual environments". In: *Peer-to-Peer Networking and Applications* 8.2 (Mar. 1, 2015), pp. 276–300. ISSN: 1936-6450. DOI: 10.1007/s12083-013-0231-5. URL: https://doi.org/10.1007/s12083-013-0231-5 (visited on 10/15/2020).

[30] Wentong Cai, F.B.S. Lee, and L. Chen. "An auto-adaptive dead reckoning algorithm for distributed interactive simulation". In: *Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99. (Cat. No.PR00155)*. Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99. (Cat. No.PR00155). May 1999, pp. 82–89. DOI: 10.1109/PADS.1999.766164.

[31] Tony Cannon. "Fight the lag!" In: *Game Developer Magazine* 19.9 (Sept. 2012), pp. 7–13. URL: http://twvideo01.ubm-us.net/o1/vault/GD_Mag_Archives/GDM_September_2012.pdf (visited on 07/08/2019).

[32] Emanuele Carlini and Alessandro Lulli. "A Spatial Analysis of Multiplayer Online Battle Arena Mobility Traces". In: *Euro-Par 2017: Parallel Processing Workshops*. Ed. by Dora B. Heras, Luc Bougé, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 496–506. ISBN: 978-3-319-75178-8. DOI: 10.1007/978-3-319-75178-8_40.

[33] Emanuele Carlini and Alessandro Lulli. *Data used for submission entitled "A spatial analysis of Multiplayer Online Battle Arena mobility traces"*. Type: dataset. May 26, 2017. DOI: 10.5281/zenodo.583600. URL: https://zenodo.org/record/583600 (visited on 10/08/2021).

[34] Marcus Carter, Kyle Moore, Jane Mavoa, Heather Horst, and luke gaspard luke. "Situating the Appeal of Fortnite Within Children's Changing Play Cultures". In: *Games and Culture* 15.4 (June 1, 2020). Publisher: SAGE Publications, pp. 453–471. ISSN: 1555-4120. DOI: 10.1177/1555412020913771. URL: https://doi.org/10.1177/1555412020913771 (visited on 09/29/2021).

[35] Armando Castañeda, Pierre Fraigniaud, Ami Paz, Sergio Rajsbaum, Matthieu Roy, and Corentin Travers. "A topological perspective on distributed network algorithms". In: *Theoretical Computer Science* 849 (Jan. 6, 2021), pp. 121–137. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2020.10.012. URL: https://www.sciencedirect.com/science/article/pii/S0304397520305831 (visited on 10/05/2021).

[36] Miguel Castro, Manuel Costa, and Antony Rowstron. "Should we build Gnutella on a structured overlay?" In: *ACM SIGCOMM Computer Communication Review* 34.1 (Jan. 1, 2004), pp. 131–136. ISSN: 0146-4833. DOI: 10.1145/972374.972397. URL: https://doi.org/10.1145/972374.972397 (visited on 10/18/2021).

[37] Şafak Burak Çevikbaş and Veysi İşler. "Phaneros: Visibility-based framework for massive peer-to-peer virtual environments". In: *Computer Animation and Virtual Worlds* 30.1 (2019). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cav.1808, e1808. ISSN: 1546-427X. DOI: 10.1002/cav.1808. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cav.1808 (visited on 06/29/2020).

[38] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. "Orthogonal range searching on the RAM, revisited". In: *Proceedings of the twenty-seventh annual symposium on Computational geometry*. SoCG '11. New York, NY, USA: Association for Computing Machinery, June 13, 2011, pp. 1–10. ISBN: 978-1-4503-0682-9. DOI: 10.1145/1998196.1998198. URL: https://doi.org/10.1145/1998196.1998198 (visited on 11/04/2021).

[39] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. "Making gnutella-like P2P systems scalable". In: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. SIGCOMM '03. New York, NY, USA: Association for Computing Machinery, Aug. 25, 2003, pp. 407–418. ISBN: 978-1-58113-735-4. DOI: 10.1145/863955.864000. URL: https://doi.org/10.1145/863955.864000 (visited on 10/20/2021).

[40] Bernard Chazelle. "Lower bounds for orthogonal range searching: I. The reporting case". In: *Journal of the ACM* 37.2 (Apr. 1, 1990), pp. 200–212. ISSN: 0004-5411. DOI: 10.1145/77600.77614. URL: https://doi.org/10.1145/77600.77614 (visited on 11/04/2021).

[41] Youfu Chen and Elvis S. Liu. "A Path-Assisted Dead Reckoning Algorithm for Distributed Virtual Environments". In: *2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT). ISSN: 1550-6525. Oct. 2015, pp. 108–111. DOI: 10.1109/DS-RT.2015.29.

[42] Minkyoung Cho, David M. Mount, and Eunhui Park. "Maintaining Nets and Net Trees under Incremental Motion". In: *Algorithms and Computation*. Ed. by Yingfei Dong, Ding-Zhu Du, and Oscar Ibarra. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 1134–1143. ISBN: 978-3-642-10631-6. DOI: 10.1007/978-3-642-10631-6_114.

[43] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". In: *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Ed. by Hannes Federrath. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 46–66. ISBN: 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_4. URL: https://doi.org/10.1007/3-540-44702-4_4 (visited on 10/18/2021).

[44] Kenneth L. Clarkson. "Fast algorithms for the all nearest neighbors problem". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. 24th Annual Symposium on Foundations of Computer Science (sfcs 1983). ISSN: 0272-5428. Nov. 1983, pp. 226–232. DOI: 10.1109/SFCS.1983.16.

[45] Mark Claypool and Kajal Claypool. "Latency and player actions in online games". In: *Communications of the ACM* 49.11 (Nov. 1, 2006), pp. 40–45. ISSN: 0001-0782. DOI: 10.1145/1167838.1167860. URL: https://doi.org/10.1145/1167838.1167860 (visited on 10/08/2021).

[46] *Client-side Prediction for Smooth Multiplayer Gameplay*. KinematicSoup Technologies Inc. URL: https://www.kinematicsoup.com/news/2017/5/30/multiplayerprediction (visited on 06/24/2020).

[47] Victor Clincy and Brandon Wilgor. "Subjective Evaluation of Latency and Packet Loss in a Cloud-Based Game". In: *2013 10th International Conference on Information Technology: New Generations*. 2013 10th International Conference on Information Technology: New Generations. Apr. 2013, pp. 473–476. DOI: 10.1109/ITNG.2013.79.

[48] Richard Cole and Lee-Ad Gottlieb. "Searching dynamic point sets in spaces with bounded doubling dimension". In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing*. STOC '06. New York, NY, USA: Association for Computing Machinery, May 21, 2006, pp. 574–583. ISBN: 978-1-59593-134-4. DOI: 10.1145/1132516.1132599. URL: https://doi.org/10.1145/1132516.1132599 (visited on 11/09/2020).

[49] Gennaro Cordasco, Rosario De Chiara, Ugo Erra, and Vittorio Scarano. "Some considerations on the design of a P2P infrastructure for massive simulations". In: *2009 International Conference on Ultra Modern Telecommunications Workshops*. 2009 International Conference on Ultra Modern Telecommunications Workshops. ISSN: 2157-023X. Oct. 2009, pp. 1–7. DOI: 10.1109/ICUMT.2009.5345386.

[50] Mark De Berg, Marcel Roeloffzen, and Bettina Speckmann. "Kinetic compressed quadtrees in the black-box model with applications to collision detection for low-density scenes". In: *Proceedings of the 20th Annual European conference on Algorithms*. ESA'12. Berlin, Heidelberg: Springer-Verlag, Sept. 10, 2012, pp. 383–394. ISBN: 978-3-642-33089-6. DOI: 10.1007/978-3-642-33090-2_34. URL: https://doi.org/10.1007/978-3-642-33090-2_34 (visited on 07/21/2021).

[51] Declan Delaney, Tomas E. Ward, and Seamus McLoone. "On reducing entity state update packets in distributed interactive simulations using a hybrid model". In: Proceedings 21st IASTED International Multi-conference on Applied Informatics. Ed. by D. Delaney, T. Ward, and S. McLoone. Innsbruck, Austria: IASTED, 2003. URL: http://mural.maynoothuniversity.ie/280/ (visited on 06/24/2020).

[52] Cédric Fleury, Thierry Duval, Valérie Gouranton, and Bruno Arnaldi. "A new adaptive data distribution model for consistency maintenance in collaborative virtual environments". In: *Proceedings of the 16th Eurographics conference on Virtual Environments & Second Joint Virtual Reality*. EGVE - JVRC'10. Goslar, DEU: Eurographics Association, Sept. 27, 2010, pp. 29–36. ISBN: 978-3-905674-30-9. (Visited on 10/13/2021).

[53] Deborah A. Fullford. "Distributed interactive simulation: its past, present, and future". In: *Proceedings of the 28th conference on Winter simulation*. WSC '96. USA: IEEE Computer Society, Nov. 8, 1996, pp. 179–185. ISBN: 978-0-7803-3383-3. DOI: 10.1145/256562.256601. URL: https://doi.org/10.1145/256562.256601 (visited on 10/08/2021).

[54] Stefan Funke, Alexander Kesselman, Ulrich Meyer, and Michael Segal. "A simple improved distributed algorithm for minimum CDS in unit disk graphs". In: *ACM Transactions on Sensor Networks* 2.3 (Aug. 1, 2006), pp. 444–453. ISSN: 1550-4859. DOI: 10.1145/1167935.1167941. URL: https://doi.org/10.1145/1167935.1167941 (visited on 04/05/2022).

[55] Jie Gao, Leonidas J. Guibas, and An Nguyen. "Deformable spanners and applications". In: *Computational Geometry*. Special Issue on the 20th ACM Symposium on Computational Geometry 35.1 (Aug. 1, 2006), pp. 2–19. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2005.10.001. URL: http://www.sciencedirect.com/science/article/pii/S0925772105000994 (visited on 10/15/2020).

[56] Jie Gao, Leonidas J. Guibas, and An Nguyen. "Distributed Proximity Maintenance in Ad Hoc Mobile Networks". In: *Distributed Computing in Sensor Systems*. Ed. by Viktor K. Prasanna, Sitharama S. Iyengar, Paul G. Spirakis, and Matt Welsh. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 4–19. ISBN: 978-3-540-31671-8. DOI: 10.1007/11502593_4.

[57] Mohsen Ghaffari, Behnoosh Hariri, and Shervin Shirmohammadi. "A delaunay triangulation architecture supporting churn and user mobility in MMVEs". In: *Proceedings of the 18th international workshop on Network and operating systems support for digital audio and video*. NOSSDAV '09. Williamsburg, VA, USA: Association for Computing Machinery, June 3, 2009, pp. 61–66. ISBN: 978-1-60558-433-1. DOI: 10.1145/1542245.1542260. URL: https://doi.org/10.1145/1542245.1542260 (visited on 06/29/2020).

[58] Benyamin Ghojogh and Saber Salehkaleybar. "Distributed voting in beep model". In: *Signal Processing* 177 (Dec. 1, 2020), p. 107732. ISSN: 0165-1684. DOI: 10.1016/j.sigpro.2020.107732. URL: https://www.sciencedirect.com/science/article/pii/S0165168420302759 (visited on 10/05/2021).

[59] Lee-Ad Gottlieb and Liam Roditty. "An Optimal Dynamic Spanner for Doubling Metric Spaces". In: *Algorithms - ESA 2008*. Ed. by Dan Halperin and Kurt Mehlhorn. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 478–489. ISBN: 978-3-540-87744-8. DOI: 10.1007/978-3-540-87744-8_40.

[60] Leonidas J. Guibas, Joseph S. B. Mitchell, and Thomas Roos. "Voronoi diagrams of moving points in the plane". In: *Graph-Theoretic Concepts in Computer Science*. Ed. by Gunther Schmidt and Rudolf Berghammer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1992, pp. 113–125. ISBN: 978-3-540-46735-9. DOI: 10.1007/3-540-55121-2_11.

[61] Leonidas J. Guibas and Marcel Roeloffzen. "Modeling motion". In: *Handbook of Discrete and Computational Geometry*. 3rd ed. Num Pages: 20. Chapman and Hall/CRC, 2017. ISBN: 978-1-315-11960-1.

[62] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. "Approximate Range Selection Queries in Peer-to-Peer Systems". In: Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003). 2003, p. 11.

[63] Sariel Har-Peled and Manor Mendel. "Fast Construction of Nets in Low-Dimensional Metrics and Their Applications". In: *SIAM Journal on Computing* 35.5 (Jan. 2006), pp. 1148–1184. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/S0097539704446281. URL: http://epubs.siam.org/doi/10.1137/S0097539704446281 (visited on 10/16/2020).

[64] Juha Heinonen. “Quasisymmetric Maps: Basic Theory I”. In: *Lectures on Analysis on Metric Spaces*. Ed. by Juha Heinonen. Universitext. New York, NY: Springer, 2001, pp. 78–87. ISBN: 978-1-4613-0131-8. DOI: 10.1007/978-1-4613-0131-8_10. URL: https://doi.org/10.1007/978-1-4613-0131-8_10 (visited on 09/28/2022).

[65] *Heroes of Newerth - Home*. URL: http://www.heroesofnewerth.com/ (visited on 10/08/2021).

[66] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. “VON: a scalable peer-to-peer network for virtual environments”. In: *IEEE Network* 20.4 (July 2006). Conference Name: IEEE Network, pp. 22–31. ISSN: 1558-156X. DOI: 10.1109/MNET.2006.1668400.

[67] Shun-Yun Hu and Guan-Ming Liao. “Scalable peer-to-peer networked virtual environment”. In: *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. NetGames ’04. New York, NY, USA: Association for Computing Machinery, Aug. 30, 2004, pp. 129–133. ISBN: 978-1-58113-942-6. DOI: 10.1145/1016540.1016552. URL: https://doi.org/10.1145/1016540.1016552 (visited on 04/08/2021).

[68] “IEEE Standard for Distributed Interactive Simulation–Application Protocols”. In: *IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995)* (Dec. 2012). Conference Name: IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995), pp. 1–747. DOI: 10.1109/IEEESTD.2012.6387564.

[69] Piotr Indyk and Rajeev Motwani. “Approximate nearest neighbors: towards removing the curse of dimensionality”. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. STOC ’98. New York, NY, USA: Association for Computing Machinery, May 23, 1998, pp. 604–613. ISBN: 978-0-89791-962-3. DOI: 10.1145/276698.276876. URL: https://doi.org/10.1145/276698.276876 (visited on 09/20/2022).

[70] Chunzhen Jiang, Aritra Kundu, Shengmei Liu, Robert Salay, Xiokun Xu, and Mark Claypool. “A Survey of Player Opinions of Network Latency in Online Games”. In: (2020), p. 12.

[71] Jehn-Ruey Jiang, Jiun-Shiang Chiou, and Shun-Yun Hu. “Enhancing Neighborship Consistency for Peer-to-Peer Distributed Virtual Environments”. In: *27th International Conference on Distributed Computing Systems Workshops (ICDCSW’07)*. 27th International Conference on Distributed Computing Systems Workshops (ICDCSW’07). ISSN: 1545-0678. June 2007, pp. 71–71. DOI: 10.1109/ICDCSW.2007.101.

[72] Jehn-Ruey Jiang, Yu-Li Huang, and Shun-Yun Hu. “Scalable AOI-Cast for Peer-to-Peer Networked Virtual Environments”. In: *2008 The 28th International Conference on Distributed Computing Systems Workshops*. 2008 The 28th International Conference on Distributed Computing Systems Workshops. ISSN: 2332-5666. June 2008, pp. 447–452. DOI: 10.1109/ICDCS.Workshops.2008.80.

[73] C. Johnen, L.O. Alima, A.K. Datta, and S. Tixeuil. “Self-stabilizing neighborhood synchronizer in tree networks”. In: *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*. Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003). ISSN: 1063-6927. June 1999, pp. 487–494. DOI: 10.1109/ICDCS.1999.776551.

[74] Haim Kaplan, Alexander Kauer, Katharina Klost, Kristin Knorr, Wolfgang Mulzer, Liam Roditty, and Paul Seiferth. "Dynamic Connectivity in Disk Graphs". In: *arXiv:2106.14935 [cs]* (June 28, 2021). arXiv: 2106.14935. URL: http://arxiv.org/abs/2106.14935 (visited on 04/05/2022).

[75] Menelaos I. Karavelas and Leonidas J. Guibas. "Static and kinetic geometric spanners with applications". In: *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*. SODA '01. USA: Society for Industrial and Applied Mathematics, Jan. 9, 2001, pp. 168–176. ISBN: 978-0-89871-490-6. (Visited on 07/07/2021).

[76] Yoshihiro Kawahara, Tomonori Aoyama, and Hiroyuki Morikawa. "A Peer-to-Peer Message Exchange Scheme for Large-Scale Networked Virtual Environments". In: *Telecommunication Systems* 25.3 (Mar. 1, 2004), pp. 353–370. ISSN: 1572-9451. DOI: 10.1023/B:TELS.0000014789.70171.fd. URL: https://doi.org/10.1023/B:TELS.0000014789.70171.fd (visited on 10/21/2021).

[77] J. Mark Keil and Carl A. Gutwin. "Classes of graphs which approximate the complete euclidean graph". In: *Discrete & Computational Geometry* 7.1 (Jan. 1, 1992), pp. 13–28. ISSN: 1432-0444. DOI: 10.1007/BF02187821. URL: https://doi.org/10.1007/BF02187821 (visited on 11/10/2021).

[78] J. Keller and G. Simon. "Toward a peer-to-peer shared virtual reality". In: *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*. Proceedings 22nd International Conference on Distributed Computing Systems Workshops. July 2002, pp. 695–700. DOI: 10.1109/ICDCSW.2002.1030849.

[79] Joaquín Keller and Gwendal Simon. "Solipsis: A Massively Multi-Participant Virtual World." In: PDPTA. Vol. 3. 2003, pp. 262–268.

[80] Vasily Y. Kharitonov. "Motion-aware adaptive dead reckoning algorithm for collaborative virtual environments". In: *Proceedings of the 11th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and its Applications in Industry*. VRCAI '12. New York, NY, USA: Association for Computing Machinery, Dec. 2, 2012, pp. 255–261. ISBN: 978-1-4503-1825-9. DOI: 10.1145/2407516.2407577. URL: https://doi.org/10.1145/2407516.2407577 (visited on 10/15/2020).

[81] Elahe Khatibi and Mohsen Sharifi. "Resource discovery mechanisms in pure unstructured peer-to-peer systems: a comprehensive survey". In: *Peer-to-Peer Networking and Applications* 14.2 (Mar. 1, 2021), pp. 729–746. ISSN: 1936-6450. DOI: 10.1007/s12083-020-01027-9. URL: https://doi.org/10.1007/s12083-020-01027-9 (visited on 10/15/2021).

[82] B. Knutsson, Honghui Lu, Wei Xu, and B. Hopkins. "Peer-to-peer support for massively multiplayer games". In: *IEEE INFOCOM 2004*. IEEE INFOCOM 2004. Vol. 1. ISSN: 0743-166X. Mar. 2004, p. 107. DOI: 10.1109/INFCOM.2004.1354485.

[83] Robert Krauthgamer and James R. Lee. "Navigating nets: simple algorithms for proximity search". In: *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. SODA '04. USA: Society for Industrial and Applied Mathematics, Jan. 11, 2004, pp. 798–807. ISBN: 978-0-89871-558-3. (Visited on 05/12/2021).

[84] Fabian Kuhn, Rogert Wattenhofer, and Aaron Zollinger. "Ad-hoc networks beyond unit disk graphs". In: *Proceedings of the 2003 joint workshop on Foundations of mobile computing*. DIALM-POMC '03. New York, NY, USA: Association for Computing Machinery, Sept. 19, 2003, pp. 69–78. ISBN: 978-1-58113-765-1. DOI: 10.1145/941079.941089. URL: https://doi.org/10.1145/941079.941089 (visited on 04/05/2022).

[85] A. Kumar, J. Xu, and E.W. Zegura. “Efficient and scalable query routing for unstructured peer-to-peer networks”. In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Vol. 2. ISSN: 0743-166X. Mar. 2005, 1162–1173 vol. 2. DOI: 10.1109/INFCOM.2005.1498343.

[86] Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (Sept. 1979). Conference Name: IEEE Transactions on Computers, pp. 690–691. ISSN: 1557-9956. DOI: 10.1109/TC.1979.1675439.

[87] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Concurrency: the Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, Oct. 4, 2019, pp. 179–196. ISBN: 978-1-4503-7270-1. URL: https://doi.org/10.1145/3335772.3335934 (visited on 10/04/2021).

[88] D. T. Lee and C. K. Wong. “Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees”. In: *Acta Informatica* 9.1 (Mar. 1, 1977), pp. 23–29. ISSN: 1432-0525. DOI: 10.1007/BF00263763. URL: https://doi.org/10.1007/BF00263763 (visited on 11/04/2021).

[89] Youngki Lee, Sharad Agarwal, Chris Butcher, and Jitu Padhye. “Measurement and Estimation of Network QoS Among Peer Xbox 360 Game Players”. In: *Passive and Active Network Measurement*. Ed. by Mark Claypool and Steve Uhlig. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 41–50. ISBN: 978-3-540-79232-1. DOI: 10.1007/978-3-540-79232-1_5.

[90] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. “Optimal clock synchronization in networks”. In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. SenSys ’09. New York, NY, USA: Association for Computing Machinery, Nov. 4, 2009, pp. 225–238. ISBN: 978-1-60558-519-2. DOI: 10.1145/1644038.1644061. URL: https://doi.org/10.1145/1644038.1644061 (visited on 10/12/2021).

[91] Zengxiang Li, Xueyan Tang, Wentong Cai, and Xiaorong Li. “Compensatory dead-reckoning-based update scheduling for distributed virtual environments”. In: *SIMULATION* 89.10 (Oct. 1, 2013). Publisher: SAGE Publications Ltd STM, pp. 1272–1287. ISSN: 0037-5497. DOI: 10.1177/0037549712470857. URL: https://doi.org/10.1177/0037549712470857 (visited on 10/11/2021).

[92] Kuo-Chi Lin, Morgan Wang, Jie Wang, and Daniel E. Schab. “Smoothing a dead reckoning image in distributed interactive simulation”. In: *Journal of Aircraft* 33.2 (1996). Publisher: American Institute of Aeronautics and Astronautics _eprint: https://doi.org/10.2514/3.46962 pp. 450–452. DOI: 10.2514/3.46962. URL: https://doi.org/10.2514/3.46962 (visited on 10/18/2021).

[93] Richard J Lipton and Robert Endre Tarjan. “Applications of a separator theorem for planar graphs”. In: 18th Annual IEEE Symposium on the Foundations of Computer Science. 1977.

[94] Elvis S. Liu and Georgios K. Theodoropoulos. “Interest management for distributed virtual environments: A survey”. In: *ACM Computing Surveys* 46.4 (Mar. 1, 2014), 51:1–51:42. ISSN: 0360-0300. DOI: 10.1145/2535417. URL: https://doi.org/10.1145/2535417 (visited on 10/01/2021).

[95] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. “The Case for a Hybrid P2P Search Infrastructure”. In: *Peer-to-Peer Systems III*. Ed. by Geoffrey M. Voelker and Scott Shenker. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 141–150. ISBN: 978-3-540-30183-7. DOI: 10.1007/978-3-540-30183-7_14.

[96] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. “A survey and comparison of peer-to-peer overlay network schemes”. In: *IEEE Communications Surveys Tutorials* 7.2 (2005). Conference Name: IEEE Communications Surveys Tutorials, pp. 72–93. ISSN: 1553-877X. DOI: 10.1109/COMST.2005.1610546.

[97] Michael Luby. “A Simple Parallel Algorithm for the Maximal Independent Set Problem”. In: *SIAM Journal on Computing* 15.4 (Nov. 1986). Publisher: Society for Industrial and Applied Mathematics, pp. 1036–1053. ISSN: 0097-5397. DOI: 10.1137/0215074. URL: https://epubs.siam.org/doi/abs/10.1137/0215074 (visited on 06/07/2022).

[98] George S. Lueker. “A data structure for orthogonal range queries”. In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. 19th Annual Symposium on Foundations of Computer Science (sfcs 1978). ISSN: 0272-5428. Oct. 1978, pp. 28–34. DOI: 10.1109/SFCS.1978.1.

[99] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. “Search and replication in unstructured peer-to-peer networks”. In: *Proceedings of the 16th international conference on Supercomputing*. ICS ’02. New York, NY, USA: Association for Computing Machinery, June 22, 2002, pp. 84–95. ISBN: 978-1-58113-483-4. DOI: 10.1145/514191.514206. URL: https://doi.org/10.1145/514191.514206 (visited on 10/20/2021).

[100] Nancy A. Lynch. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996. 904 pp. ISBN: 978-0-08-050470-4.

[101] Damien Marshall, Seamus McLoone, Tomas E. Ward, and Declan Delaney. “Does Reducing Packet Transmission Rates Help to Improve Consistency within Distributed Interactive Applications?” In: CGAMES06. Dublin Institute of Technology, Ireland, Nov. 2006. URL: http://mural.maynoothuniversity.ie/1283/ (visited on 10/15/2020).

[102] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. “Local-lag and timewarp: providing consistency for replicated continuous applications”. In: *IEEE Transactions on Multimedia* 6.1 (Feb. 2004). Conference Name: IEEE Transactions on Multimedia, pp. 47–57. ISSN: 1941-0077. DOI: 10.1109/TMM.2003.819751.

[103] Martin Mauve. “Consistency in replicated continuous interactive media”. In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. CSCW ’00. New York, NY, USA: Association for Computing Machinery, Dec. 1, 2000, pp. 181–190. ISBN: 978-1-58113-222-9. DOI: 10.1145/358916.358989. URL: https://doi.org/10.1145/358916.358989 (visited on 10/16/2021).

[104] Sabrina Merkel, Sanaz Mostaghim, and Hartmut Schmeck. “Hop count based distance estimation in mobile ad hoc networks – Challenges and consequences”. In: *Ad Hoc Networks*. Smart solutions for mobility supported distributed and embedded systems 15 (Apr. 1, 2014), pp. 39–52. ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2013.08.013. URL: https://www.sciencedirect.com/science/article/pii/S1570870513001765 (visited on 07/30/2021).

[105] mikolalysenko. *Replication in networked games: Overview (Part 1)*. 0 FPS. Feb. 10, 2014. URL: https://0fps.net/2014/02/10/replication-in-networked-games-overview-part-1/ (visited on 07/05/2019).

[106] Jeremy R. Millar, Douglas D. Hodson, Gilbert L. Peterson, and Darryl K. Ahner. "Consistency and fairness in real-time distributed virtual environments: Paradigms and relationships". In: *Journal of Simulation* 11.3 (Aug. 1, 2017). Publisher: Taylor & Francis, pp. 295–302. ISSN: 1747-7778. DOI: 10.1057/s41273-016-0035-8. URL: https://orsociety.tandfonline.com/doi/abs/10.1057/s41273-016-0035-8 (visited on 10/15/2020).

[107] Philipp Moll, Mathias Lux, Sebastian Theuermann, and Hermann Hellwagner. "A Network Traffic and Player Movement Model to Improve Networking for Competitive Online Games". In: *2018 16th Annual Workshop on Network and Systems Support for Games (NetGames)*. 2018 16th Annual Workshop on Network and Systems Support for Games (NetGames). ISSN: 2156-8146. June 2018, pp. 1–6. DOI: 10.1109/NetGames.2018.8463390.

[108] M. Mordacchini, L. Ricci, L. Ferrucci, M. Albano, and R. Baraglia. "Hivory: Range Queries on Hierarchical Voronoi Overlays". In: *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*. 2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P). ISSN: 2161-3567. Aug. 2010, pp. 1–10. DOI: 10.1109/P2P.2010.5569973.

[109] P. Morillo, W. Moncho, J. M. Orduña, and J. Duato. "Providing Full Awareness to Distributed Virtual Environments Based on Peer-to-Peer Architectures". In: *Advances in Computer Graphics*. Ed. by Tomoyuki Nishita, Qunsheng Peng, and Hans-Peter Seidel. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 336–347. ISBN: 978-3-540-35639-4. DOI: 10.1007/11784203_29.

[110] David Mosberger. "Memory consistency models". In: *ACM SIGOPS Operating Systems Review* 27.1 (Jan. 1, 1993), pp. 18–26. ISSN: 0163-5980. DOI: 10.1145/160551.160553. URL: https://doi.org/10.1145/160551.160553 (visited on 10/13/2021).

[111] David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. "A computational framework for incremental motion". In: *Proceedings of the twentieth annual symposium on Computational geometry*. SCG '04. New York, NY, USA: Association for Computing Machinery, June 8, 2004, pp. 200–209. ISBN: 978-1-58113-885-6. DOI: 10.1145/997817.997849. URL: https://doi.org/10.1145/997817.997849 (visited on 10/15/2020).

[112] Giri Narasimhan and Michiel Smid. *Geometric Spanner Networks*. Google-Books-ID: SY4cNZWc4GAC. Cambridge University Press, 2007. 500 pp. ISBN: 978-0-521-81513-0.

[113] Ashraf Nasr and Samir A. Elsagheer Mohamed. "Accurate Distance Estimation for VANET Using Nanointegrated Devices". In: 2012 (June 28, 2012). Publisher: Scientific Research Publishing. DOI: 10.4236/opj.2012.22015. URL: http://www.scirp.org/journal/PaperInformation.aspx?PaperID=20402 (visited on 09/14/2021).

[114] Joseph O'Rourke. "The Yao Graph Y_6 is a Spanner". In: *Computer Science: Faculty Publications* (June 1, 2010). URL: https://scholarworks.smith.edu/csc_facpubs/33.

[115] Mark H. Overmars. *The Design of Dynamic Data Structures*. Google-Books-ID: Z8R9zaUQ6xAC. Springer Science & Business Media, 1983. 194 pp. ISBN: 978-3-540-12330-9.

[116] David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000. ISBN: 0-89871-464-8.

[117] Richard Price, Peter Tiňo, and Georgios Theodoropoulos. "Still Alive: Extending Keep-Alive Intervals in P2P Overlay Networks". In: *Mobile Networks and Applications* 17.3 (June 1, 2012), pp. 378–394. ISSN: 1572-8153. DOI: 10.1007/s11036-011-0317-3. URL: https://doi.org/10.1007/s11036-011-0317-3 (visited on 11/23/2021).

[118] Ricky Pusch. *Explaining how fighting games use delay-based and rollback netcode*. Ars Technica. Oct. 18, 2019. URL: https://arstechnica.com/gaming/2019/10/explaining-how-fighting-games-use-delay-based-and-rollback-netcode/ (visited on 10/16/2021).

[119] Zahed Rahmati, Mohammad Ali Abam, Valerie King, and Sue Whitesides. "Kinetic k-Semi-Yao graph and its applications". In: *Computational Geometry*. Canadian Conference on Computational Geometry 77 (Mar. 1, 2019), pp. 10–26. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2015.11.001. URL: http://www.sciencedirect.com/science/article/pii/S0925772115001248 (visited on 10/15/2020).

[120] Zahed Rahmati, Mohammad Ali Abam, Valerie King, Sue Whitesides, and Alireza Zarei. "A simple, faster method for kinetic proximity problems". In: *Computational Geometry* 48.4 (May 1, 2015), pp. 342–359. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2014.12.002. URL: http://www.sciencedirect.com/science/article/pii/S0925772114001278 (visited on 10/15/2020).

[121] Asim Rasheed, Saira Gillani, Sana Ajmal, and Amir Qayyum. "Vehicular Ad Hoc Network (VANET): A Survey, Challenges, and Applications". In: *Vehicular Ad-Hoc Networks for Smart Cities*. Ed. by Anis Laouiti, Amir Qayyum, and Mohamad Naufal Mohamad Saad. Advances in Intelligent Systems and Computing. Singapore: Springer, 2017, pp. 39–51. ISBN: 978-981-10-3503-6. DOI: 10.1007/978-981-10-3503-6_4.

[122] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. "A scalable content-addressable network". In: *ACM SIGCOMM Computer Communication Review* 31.4 (Aug. 27, 2001), pp. 161–172. ISSN: 0146-4833. DOI: 10.1145/964723.383072. URL: https://doi.org/10.1145/964723.383072 (visited on 09/09/2021).

[123] Laura Ricci and Emanuele Carlini. "Distributed Virtual Environments: From client server to cloud and P2P architectures". In: *2012 International Conference on High Performance Computing Simulation (HPCS)*. 2012 International Conference on High Performance Computing Simulation (HPCS). July 2012, pp. 8–17. DOI: 10.1109/HPCSim.2012.6266885.

[124] John Risson and Tim Moors. "Survey of research towards robust peer-to-peer networks: Search methods". In: *Computer Networks* 50.17 (Dec. 5, 2006), pp. 3485–3521. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2006.02.001. URL: https://www.sciencedirect.com/science/article/pii/S1389128606000223 (visited on 09/15/2021).

[125] Antony Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems". In: *Middleware 2001*. Ed. by Rachid Guerraoui. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 329–350. ISBN: 978-3-540-45518-9. DOI: 10.1007/3-540-45518-3_18.

[126] Jim Ruppert and Raimund Seidel. "Approximating the d-dimensional complete Euclidean graph". In: Proceedings of the 3rd Canadian Conference on Computational Geometry (CCCG 1991). 1991, pp. 207–210.

[127] Daniel Russel. "Kinetic data structures in practice". PhD thesis. STANFORD UNIVERSITY, Mar. 2007.

[128] Nicola Santoro. *Design and Analysis of Distributed Algorithms*. John Wiley & Sons, 2007. 610 pp. ISBN: 978-0-471-71997-7.

[129] Hermann Schloss, Jean Botev, Markus Esch, Alex Höhfeld, Ingo Scholtes, and Peter Sturm. "Elastic Consistency in Decentralized Distributed Virtual Environments". In: *2008 International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution*. 2008 International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution. Nov. 2008, pp. 249–252. DOI: 10.1109/AXMEDIS.2008.26.

[130] Arne Schmieg, Michael Stieler, Sebastian Jeckel, Patric Kabus, Bettina Kemme, and Alejandro Buchmann. "pSense - Maintaining a Dynamic Localized Peer-to-Peer Structure for Position Based Multicast in Games". In: *2008 Eighth International Conference on Peer-to-Peer Computing*. 2008 Eighth International Conference on Peer-to-Peer Computing. ISSN: 2161-3567. Sept. 2008, pp. 247–256. DOI: 10.1109/P2P.2008.20.

[131] Micha Sharir, Mîkā Šārîr, and Pankaj K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Google-Books-ID: HSZhIHxHXJAC. Cambridge University Press, May 26, 1995. 388 pp. ISBN: 978-0-521-47025-4.

[132] Kwang-Hyun Shim and Jong-Sung Kim. "A dead reckoning algorithm with variable threshold scheme in networked virtual environment". In: *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236)*. 2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236). Vol. 2. ISSN: 1062-922X. Oct. 2001, 1113–1118 vol.2. DOI: 10.1109/ICSMC.2001.973069.

[133] Ajit Singh, Mukesh Kumar, Rahul Rishi, and D. K. Madan. "A Relative Study of MANET and VANET: Its Applications, Broadcasting Approaches and Challenging Issues". In: *Advances in Networks and Communications*. Ed. by Natarajan Meghanathan, Brajesh Kumar Kaushik, and Dhinaharan Nagamalai. Communications in Computer and Information Science. Berlin, Heidelberg: Springer, 2011, pp. 627–632. ISBN: 978-3-642-17878-8. DOI: 10.1007/978-3-642-17878-8_63.

[134] Shio Kumar Singh, M.P Singh, and D K Singh. "Routing Protocols in Wireless Sensor Networks - A Survey". In: *International Journal of Computer Science & Engineering Survey* 1.2 (Nov. 29, 2010), pp. 63–83. ISSN: 09763252. DOI: 10.5121/ijcses.2010.1206. URL: http://www.airccse.org/journal/ijcses/papers/1110ijcses06.pdf (visited on 10/05/2021).

[135] S. Sioutas, D. Sofotassios, K. Tsichlas, D. Sotiropoulos, and P. Vlamos. "Canonical Polygon Queries on the Plane: A New Approach". In: *Journal of Computers* 4.9 (Sept. 1, 2009), pp. 913–919. (Visited on 11/02/2021).

[136] Steven S. Skiena. *The Algorithm Design Manual*. London: Springer London, 2008. ISBN: 978-1-84800-069-8 978-1-84800-070-4. DOI: 10.1007/978-1-84800-070-4. URL: http://link.springer.com/10.1007/978-1-84800-070-4 (visited on 01/03/2022).

[137] *Source Multiplayer Networking - Valve Developer Community*. URL: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (visited on 08/27/2019).

[138] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (Aug. 27, 2001), pp. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: https://doi.org/10.1145/964723.383071 (visited on 09/15/2021).

[139] Yoo Taewon, Choi Sunghee, Lee Hyonik, and Lee Jinwon. "Distributed Kinetic Delaunay Triangulation". In: *Proceedings of the Korean Information Science Society Conference* (2005). Publisher: Korean Institute of Information Scientists and Engineers, pp. 964–966. ISSN: 1598-5164. URL: https://www.koreascience.or.kr/article/CFKO200507523374001.page (visited on 11/06/2021).

[140] Egemen Tanin, Aaron Harwood, and Hanan Samet. "Indexing Distributed Complex Data for Complex Queries". In: Proceedings of the 2004 annual national conference on Digital government research. 2004, p. 10.

[141] Vladyslav Taran, Oleg Alienin, Sergii Stirenko, Yuri Gordienko, and A. Rojbi. "Performance evaluation of distributed computing environments with Hadoop and Spark frameworks". In: *2017 IEEE International Young Scientists Forum on Applied Physics and Engineering (YSF)*. 2017 IEEE International Young Scientists Forum on Applied Physics and Engineering (YSF). Oct. 2017, pp. 80–83. DOI: 10.1109/YSF.2017.8126655.

[142] Gerard Tel. *Introduction to Distributed Algorithms*. 2nd Edition. Nov. 2000. ISBN: 978-0-521-79483-1.

[143] Henrik Warpefelt and Harko Verhagen. "Towards an updated typology of non-player character roles". In: *Proceedings of the 8th International Conference on Game and Entertainment Technologies (GET 2015)*. 2015, pp. 1–9.

[144] Amir Yahyavi and Bettina Kemme. "Peer-to-peer architectures for massively multiplayer online games: A Survey". In: *ACM Computing Surveys* 46.1 (July 11, 2013), 9:1–9:51. ISSN: 0360-0300. DOI: 10.1145/2522968.2522977. URL: https://doi.org/10.1145/2522968.2522977 (visited on 10/21/2021).

[145] Andrew Chi-Chih Yao. "On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems". In: *SIAM Journal on Computing* 11.4 (Nov. 1, 1982). Publisher: Society for Industrial and Applied Mathematics, pp. 721–736. ISSN: 0097-5397. DOI: 10.1137/0211059. URL: https://epubs.siam.org/doi/abs/10.1137/0211059 (visited on 10/25/2021).

[146] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. "Tapestry: a resilient global-scale overlay for service deployment". In: *IEEE Journal on Selected Areas in Communications* 22.1 (Jan. 2004). Conference Name: IEEE Journal on Selected Areas in Communications, pp. 41–53. ISSN: 1558-0008. DOI: 10.1109/JSAC.2003.818784.

[147] Suiping Zhou, Wentong Cai, Bu-Sung Lee, and Stephen J. Turner. "Time-space consistency in large-scale distributed virtual environments". In: *ACM Transactions on Modeling and Computer Simulation* 14.1 (Jan. 1, 2004), pp. 31–47. ISSN: 1049-3301. DOI: 10.1145/974734.974736. URL: https://doi.org/10.1145/974734.974736 (visited on 10/08/2021).

[148] Ming Zhu, Jiannong Cao, Deming Pang, Zongjian He, and Ming Xu. “SDN-Based Routing for Efficient Message Propagation in VANET”. In: *Wireless Algorithms, Systems, and Applications*. Ed. by Kuai Xu and Haojin Zhu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 788–797. ISBN: 978-3-319-21837-3. DOI: 10.1007/978-3-319-21837-3_77.

Appendix A

Glossary

In this appendix, we regroup some of the terms that are defined and used in the document.

- Adjacent (p.20): if a node u is able to send messages to a node v (in other words, if $(u, v) \in E$), then v is said to be adjacent to u in the connection graph.

- Black-Box model (p.38, and Section 1.7.2): movement model in which evenly spaced *time steps* may be identified, in such a way that the nodes move only at those time steps, and can be considered motionless between two consecutive time steps.

- Certificate (p.40): certificates are predicates used in Kinetic Data Structures (KDSs). These predicates involve a constant number of nodes, and ascertain that the KDS is valid: each time a certificate fails because of the movements of the nodes, operations have to be made to update the structure with regards to the new positions of the nodes.

- Connection Graph (p.20): the graph representing the current status of the connections of the nodes.

- Doubling Dimension (p.38): a metric space is doubling if there is a doubling dimension M such that any ball of radius r in that metric space can be covered with M balls of radius $\frac{r}{2}$. We call M the doubling dimension of that metric space.

- Dynamic (p.29): used to describe a system that allows to insert and delete elements of the system. Typically, if a distributed system is dynamic, it allows nodes to enter and leave the system. Note that assumptions can still be made as to how many elements may be inserted/deleted at once.

- Event (p.41): the failure of a certificate in a KDS.

- Flight plan model (p.38, and Section 1.7.1): movement model in which nodes follow continuous trajectories that are known in advance. The trajectories may however change at some points in time. It is assumed that movement speeds are small with respect to computation speed; in other words, when the trajectories meet certain conditions (for example, two points crossing, or coming within a certain distance of each other), the algorithm may perform some computations, while the nodes remain stationary until the computations are finished.

- High mobility setting (p.39): the default setting in the Black-Box model: at each time step, all nodes may move.

- Kinetic (p.29): used to describe a system that allows elements to continuously change their values throughout the execution. Kinetic Data Structures (see Section 1.7) are the best examples of kinetic systems.

- Low mobility setting (p.39): a setting in the Black-Box model in which at each time step, only one node may move.

- Metric space: a set of possible positions such that a distance can be computed between any two positions of the set.

- Node (p.18): a computer in the distributed setting, or a point in the centralized setting. We assume that each node is associated with an unique position that may evolve with the time (see p.21).

- Update (p.21): a message containing information about the new state of an object.

Appendix B

Notations

Let us complement previous appendix with the main notations used throughout the document:

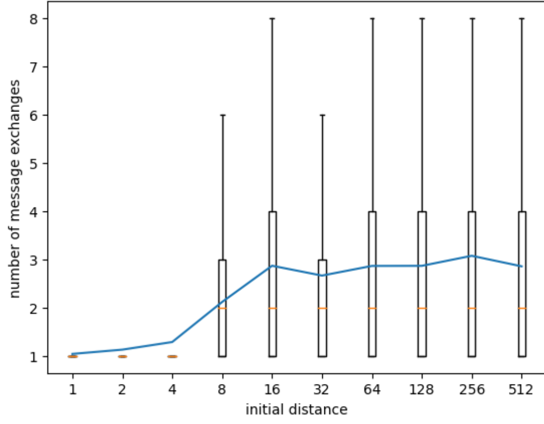
- $\text{card}(A)$: cardinality of the set A, that is, the number of elements in the set A.
- $\mathcal{V}$: the set of nodes in the distributed system.
- $p_u(t)$: the position of node u at instant t.
- U: number of possible values for one coordinate of the position of a node.
- a: the accuracy with which positions can be stored in memory. We have for any $u \in \mathcal{V}$, and any moment t, $p_u(t) \in [0; a; 2a; \cdots; Ua]^d$.
- d: the number of dimensions for the positions of the nodes in $\mathcal{V}$. Note that it is usually considered that $d = O(1)$.
- $d(u, v, t)$: the distance separating the positions of u and v at instant t.
- d_{min}: minimal distance. $\forall t, \forall u, v \in \mathcal{V}, d_{min} \leq d(u, v, t)$.
- d_{max}: maximal distance. $\forall t, \forall u, v \in \mathcal{V}, d(u, v, t) \leq d_{max}$.
- d_{mv}: maximal distance a node may move at a given instant in the Black-box model (see Section 1.6.1).
- Φ: the aspect ratio. $\Phi = \frac{d_{max}}{d_{min}}$ (see Definition 1.7, p.39)
- $\text{CloseNodes}_u(r)$: query that should return all nodes within a radius of r of u (see Definition 1.2, p.17).
- r: the parameter of the CloseNodes query. Note that some algorithms may be tailored at one specific value for r, while others accept any value for r.
- $B_u(r, t)$: the ball of radius r centered on $p_u(t)$.
- $b_u(r, t)$: the number of nodes whose positions are in $B_u(r, t)$.
- $b_{max}(r)$: the maximum number of nodes that can be at distance r from a node. $b_{max}(r) = \max_{u \in \mathcal{V}, t \in \mathbb{R}} b_u(r, t)$
- ρ: maximum number of nodes that can be situated in a ball of radius d_{mv}.

- $G(t)$: the connection graph at instant t.
- $E(t)$: the set of arcs in $G(t)$.
- $\delta(t)$: maximal degree of a node in $G(t)$, that is, the maximal number of outgoing arcs for any node $u \in \mathcal{V}$.
- k: size of the returned set when performing a query. For example, if an algorithm can answer to a query in time $O(\log n + k)$, it means that the algorithm needs to check at most a logarithmic number of nodes in addition to the nodes that are to be returned.
- s: the stretch factor of a spanner with respect to the complete euclidean graph (see Definition 1.5, p.34).
- $Cert_u(t)$: the set of certificates at time t that involve u.
- t_i : in the Black-Box model (see Section 1.6.1), t_i is the i-th time step, that is, the i-th change of the positions of the nodes from $\mathcal{V}$.

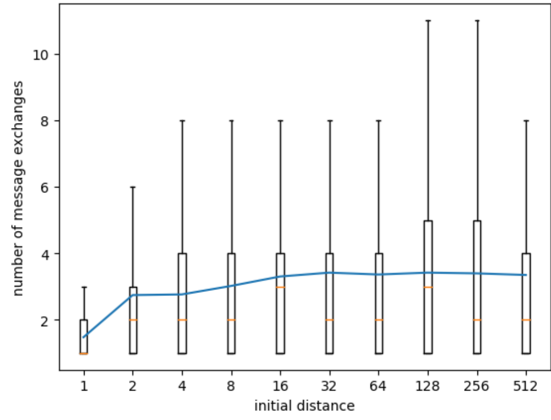
Appendix C

Figures for Continuous Movement

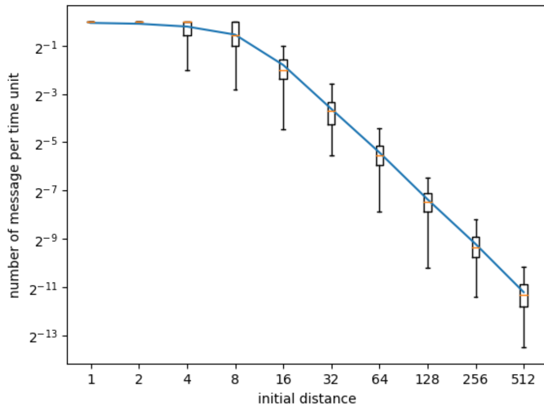
This appendix contains some additional figures analyzing results of our distance estimation algorithm $\mathcal{A}_{lc}$ on synthetic traces (see Section 2.5.1). See figures C.1, C.2, and C.3.



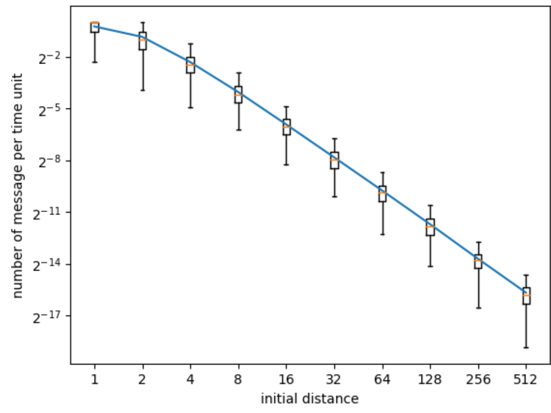
(a) M depending on initial distance, for $\varepsilon = 0.1$



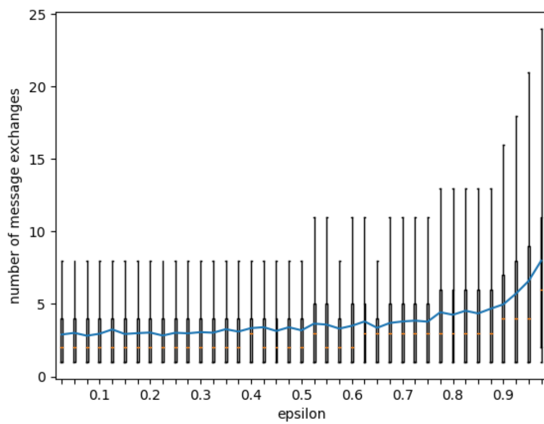
(b) M depending on initial distance, for $\varepsilon = 0.5$



(c) Messages per time unit with $\mathcal{A}_{lc}$, for $\varepsilon = 0.1$

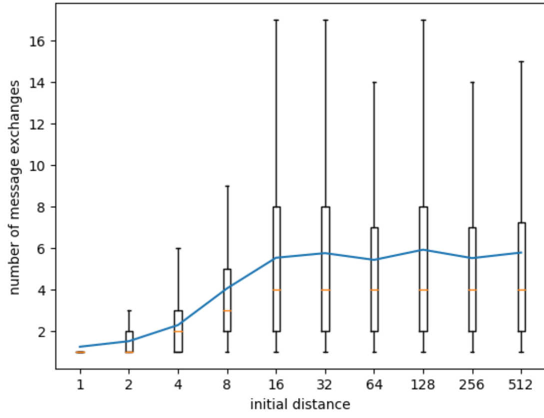


(d) Messages per time unit with $\mathcal{A}_{lc}$, for $\varepsilon = 0.5$

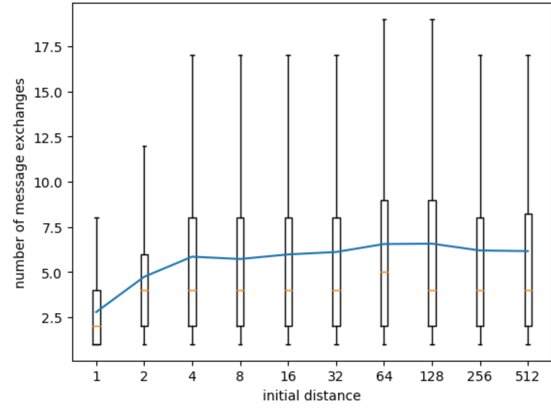


(e) M depending on ε, for $d_0 = 400$

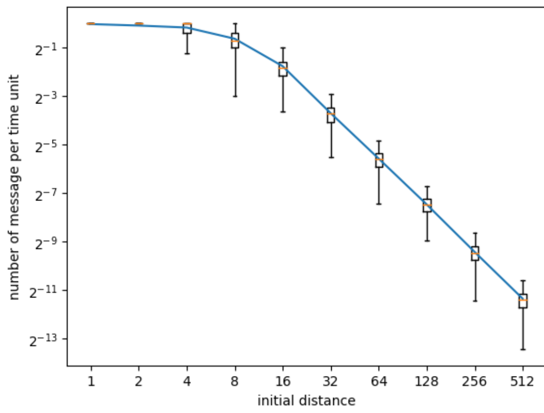
Figure C.1 – Values in the 1D case when both nodes follow a continuous movement



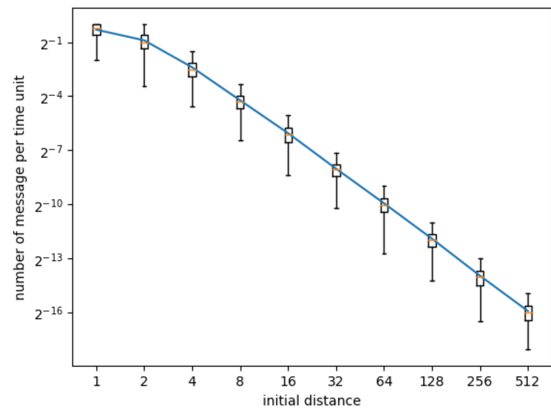
(a) M depending on initial distance, for $\varepsilon = 0.1$



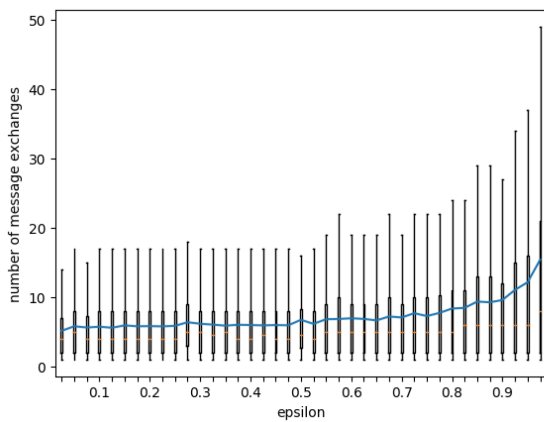
(b) M depending on initial distance, for $\varepsilon = 0.5$



(c) Messages per time unit with $\mathcal{A}_{lc}$, for $\varepsilon = 0.1$

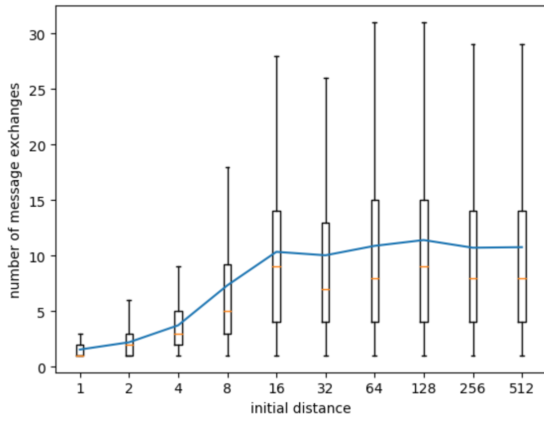


(d) Messages per time unit with $\mathcal{A}_{lc}$, for $\varepsilon = 0.5$

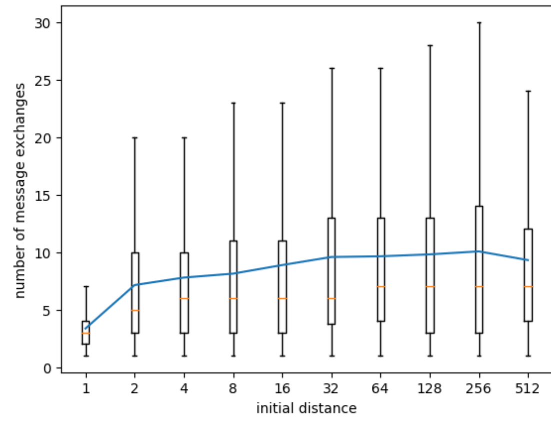


(e) M depending on ε, for $d_0 = 400$

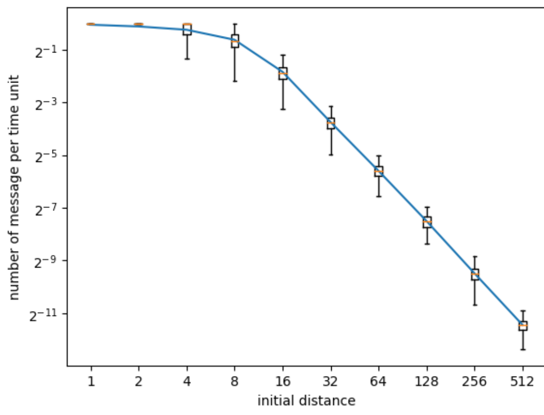
Figure C.2 – Values in the 2D case when both nodes follow a continuous movement



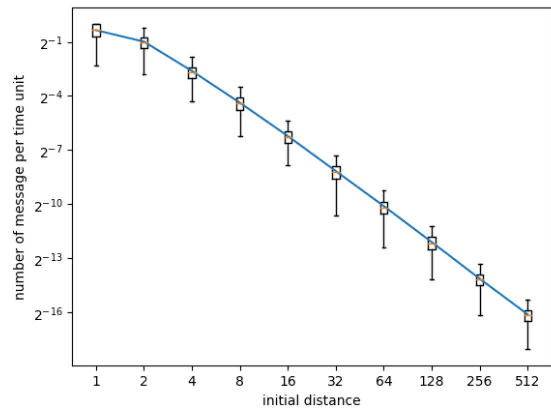
(a) M depending on initial distance, for $\varepsilon = 0.1$



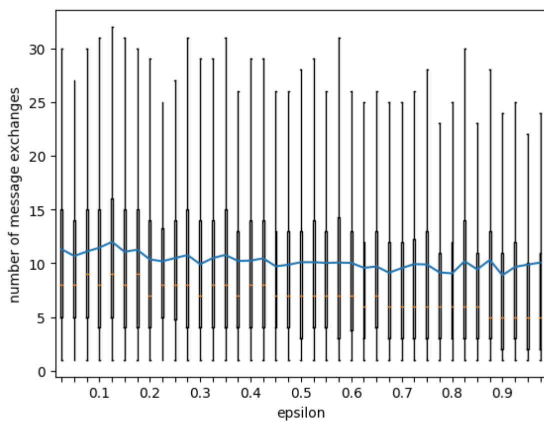
(b) M depending on initial distance, for $\varepsilon = 0.5$



(c) Messages per time unit with $\mathcal{A}_{lc}$, for $\varepsilon = 0.1$



(d) Messages per time unit with $\mathcal{A}_{lc}$, for $\varepsilon = 0.5$



(e) M depending on ε, for $d_0 = 400$

Figure C.3 – Values in the 3D case when both nodes follow a continuous movement

Appendix D

Full description of $\mathcal{A}_{cnnptr}$

The full description of $\mathcal{A}_{cnnptr}$ can be found in Algorithms 22, 23, and 24. Every line that differs from $\mathcal{A}_{cnn}$ is colored in red (line 12, line 21, lines 23 through 30, lines 68 through 71, and lines 89 through 93).

Algorithm 22 $\mathcal{A}_{cnnptr}$: certificates initialization and maintenance of the data structure

1: **function** RoundDistance(u, v, k) (*Computes the value associated with the distance to the parent.*)

2: **return** τ where $\tau = \begin{cases} \gamma & \text{if } d(u,v) < \gamma b^{k+1} \\ \beta & \text{if } \gamma b^{k+1} \leq d(u,v) < \beta b^{k+1} \\ \alpha & \text{if } \beta b^{k+1} \leq d(u,v) \end{cases}$

3: Initialization:

4: **for all** node $u \in \mathcal{V}$ **do**

5: $L_u \in \mathbb{N}$ (*largest k s.t. u is the center of a ball of radius b^k, see Algorithm 10*)

6: $f_u \in \mathcal{V}$ (*parent of u, a node $v \neq u$ except for the root, see Algorithm 10*)

7: $N_u = \{(v,k) : d(u,v) < cb^k \wedge 0 < k \leq \min(L_u, L_v)\}$ (*$(v,k) \in N_u$ if u and v are neighbors at level k*)

8: $C_u = \{(v,k) : f_v = u \wedge L_v = k\}$ (*$(v,k) \in C_u$ if the level of v is k and u is the parent of v*)

9: **let** $PN_u = \{(v,k) : k > 0 \wedge \exists (w, k+1) \in N_u \cup N_{f_u}, ((v,k) \in C_w \vee (v,k) = (w,k)) \wedge d(u,v) \geq cb^k\}$ (*Set of cousins. Note that this set is used only at initialization.*)

10: $Cert_u \leftarrow \{\texttt{Separation}_{\gamma,k}(u,v), \texttt{ShortEdge}_{c,k}(u,v) : (v,k) \in N_u \setminus \{(u,*)\}\} \cup \{\texttt{LongEdge}_{c,k}(u,v) : (v,k) \in PN_u\}$

11: $\tau \leftarrow$ RoundDistance(u, f_u, L_u); $Cert_u \leftarrow Cert_u \cup \{\texttt{Cover}_{\tau,L_u}, \texttt{SCover}_{\tau-1/b,L_u}\}$

12: $pointer_u \leftarrow (NULL, -1)$ (*$pointer_u$ contains (v,k), where v is an ancestor of u, and k the level of one of its children (see p.158)*)

13: **end for**

14: $Cert \leftarrow \bigcup_{u \in \mathcal{V}} Cert_u$

15: Main loop:

16: **for each** time t_i where a node moves **do**

17: Resolve all false certificates of $Cert$, according to the instructions of algorithms 15 and 16:

18: first `Separation`, in order of decreasing level,

19: then `ShortEdge` and `LongEdge`, in order of decreasing level,

20: then `Cover` and `SCover` certificates, in order of decreasing level.

21: $\forall u$, reset $pointer_u$ to $(NULL, -1)$.

22: **end for**

23: **function** findLower(u, k) (*recursive function, with u the next node, and k the level of the previous node*)

24: **if** $L_u \geq k+2$, or if RoundDistance(u, f_u, $k+1$) $= \gamma$ or α **then** (*The definition of RoundDistance is the same as for $\mathcal{A}_{cnn}$, see Algorithm 14, line 1.*)

25: **return** (u, k)

26: **else if** $pointer_u \neq (NULL, -1)$ **then**

27: **return** $pointer_u$

28: **else**

29: $pointer_u \leftarrow$ findLower(f_u, L_u)

30: **return** $pointer_u$

Algorithm 23 $\mathcal{A}_{cnnptr}$: management of certificates of type `Separation`, `LongEdge`, and `ShortEdge`

31: (*Each time neighbors and children are updated, the following should be executed:*)
32: **when** (v, k) is added to/*removed from* N_u **do**
33: **if** $v \neq u$ **then** add/*remove* $\texttt{Separation}_{\gamma,k}(u, v)$, $\texttt{ShortEdge}_{c,k}(u, v)$ to/*from Cert* **end if**
34: **if** $v \neq u$ **then** remove/*add* $\texttt{LongEdge}_{c,k}(u, v)$ from/*to Cert* **end if**
35: **for each** $\{u', v'\} : (u', k-1) \in C_u \cup \{u, k-1\}, (v', k-1) \in C_v \cup \{v, k-1\} : u' \neq v'$ **do**
36: add/*remove* $\texttt{LongEdge}_{c,k-1}(u', v')$ to/*from Cert* (*add if $(v', k-1) \notin N_{u'} \setminus \{(u', *)\}$, remove if certificate exists*)
37: **end for**
38: **end when**
39: **when** (v, k) is added to/*removed from* C_u **do**
40: **for each** $v' : \exists (w, k+1) \in N_u, (v', k) \in C_w \cup \{w, k\} : v \neq v'$ **do**
41: add/*remove* $\texttt{LongEdge}_{c,k}(v, v')$ to/*from Cert* (*add if $(v', k) \notin N_v \setminus \{(v, *)\}$, remove if exists*)
42: **end for**
43: **end when**

44: (*For simplicity, in the following, the certificate that fails is implicitly removed, as well as its symmetric. Also, each time a certificate is said to be created at level 0, except $\texttt{Cover}_{\gamma,0}$, it should not be created.*)

45: **when** $\texttt{Separation}_{\gamma,L}(u, v)$ fails with $L_u \leq L_v$ and $u \prec v$ **do** (*$L = L_u$. Note that the node with lowest order will be the one to reduce its level.*)
46: $PF \leftarrow \{w \neq u : (w, L) \in N_u\}$ (*Potential parent for level $L-1$ children of u*)
47: **for each** $w : (w, L-1) \in C_u$ **do** (*$L_w = L-1$ and $f_w = u$*)
48: **if** $\exists w' \in PF : d(w, w') < \gamma b^L$ **then** Redirect(w, u, w')
49: **else** Promote(w, u); $PF \leftarrow PF \cup \{w\}$
50: **end for**
51: remove $\texttt{Cover}_{*,*}(u, f_u)$, $\texttt{SCover}_{*,*}(u, f_u)$ from *Cert*; $C_{f_u} \leftarrow C_{f_u} \setminus \{(u, L)\}$
52: **for each** $w : (w, L) \in N_u$ **do** $N_w \leftarrow N_w \setminus \{(u, L)\}$ **end for**
53: $N_u \leftarrow N_u \setminus \{(w, L) \in N_u\}$
54: $L_u \leftarrow L_u - 1$; $f_u \leftarrow v$; $C_v \leftarrow C_v \cup (u, L_u)$ add $\texttt{Cover}_{\gamma,L_u}(u, v)$ to *Cert*
55: remove $\texttt{Separation}_{\gamma,L}(u, *)$, $\texttt{Separation}_{\gamma,L}(*, u)$, $\texttt{LongEdge}_{c,L}(u, *)$, $\texttt{LongEdge}_{c,L}(*, u)$
56: **end when**
57: **when** $\texttt{ShortEdge}_{c,L}(u, v)$ fails with $L_u \leq L_v$ **do** (*$L = L_u$*)
58: $N_u \leftarrow N_u \setminus (v, L)$; $N_v \leftarrow N_v \setminus (u, L)$;
59: **end when**
60: **when** $\texttt{LongEdge}_{c,L}(u, v)$ fails with $L_u \leq L_v$ **do** (*$L = L_u$*)
61: $N_u \leftarrow N_u \cup (v, L)$; $N_v \leftarrow N_v \cup (u, L)$
62: **end when**

Algorithm 24 $\mathcal{A}_{cnnptr}$: management of certificates of type `Cover` and `SCover`

63: **when** $\texttt{SCover}_{\tau,L}(u,v)$ fails with $\tau \in \{\gamma,\beta\}$ **do** (*$L = L_u$ and $v = f_u$*)
64: **if** $\tau = \beta$ **then** replace $\texttt{SCover}_{\beta,L}(u,v)$ by $\texttt{SCover}_{\gamma,L}(u,v)$ in $Cert$ **end if**
65: replace $\texttt{Cover}_{\tau+1/b,L}(u,v)$ by $\texttt{Cover}_{\tau,L}(u,v)$ in $Cert$
66: **end when**
67: **when** $\texttt{Cover}_{\tau,L}(u,v)$ fails with $\tau \in \{\gamma,\beta\}$ and $L \geq 1$ **do** (*$L = L_u$ and $v = f_u$*)
68: $(anc, k) \leftarrow \text{FINDLOWER}(f_u, L_u)$
69: **if** $L_{anc} = k+1$ and $\text{ROUNDDISTANCE}(anc, f_{anc}, L_{anc}) = \alpha$ **then**
70: **if** $\exists (f', L_{anc}+1) \in N_{f_{anc}} : d(anc, f') \leq \gamma b^{L_{anc}+1}$ **then**
71: $\text{REDIRECT}(anc, f_{anc}, f')$ **else** $\text{PROMOTE}(anc, f_{anc})$ **end if**
72: **if** $\tau = \beta$ **then**
73: **if** $\exists v' : (v', L+1) \in N_v \wedge d(u,v') \leq \gamma b^{L+1}$ **then**
74: $\text{REDIRECT}(u,v,v')$ **else** $\text{PROMOTE}(u,v)$ **end if**
75: **else**
76: replace $\texttt{Cover}_{\tau,L}(u,v)$ by $\texttt{Cover}_{\tau+1/b,L}(u,v)$ in $Cert$
77: replace $\texttt{SCover}_{\tau-1/b,L}(u,v)$ by $\texttt{SCover}_{\tau,L}(u,v)$ in $Cert$ (*replace if it exists, add otherwise*)
78: **end when**
79: **when** $\texttt{Cover}_{\alpha,L}(u,v)$ fails or $\texttt{Cover}_{\gamma,L=0}(u,v)$ fails **do**
80: **if** $\exists v' : (v', L+1) \in N_v \wedge d(u,v') \leq \gamma b^{L+1}$ **then**
81: $\text{REDIRECT}(u,v,v')$ **else** $\text{PROMOTE}(u,v)$ **end if**
82: **end when**
83: **procedure** $\text{REDIRECT}(v, oldf, newf)$
84: $C_{oldf} \leftarrow C_{oldf} \setminus \{(v, L_v)\}$; remove $\texttt{Cover}_{*,*}(v, oldf)$, $\texttt{SCover}_{*,*}(v, oldf)$ from $Cert$;
85: $f_v \leftarrow newf$; $C_{newf} \leftarrow C_{newf} \cup (v, L_v)$; add $\texttt{Cover}_{\gamma,L_v}(v, newf)$ to $Cert$
86: **procedure** $\text{PROMOTE}(v, oldf)$
87: **if** $L_{oldf} \geq L_v + 2$ **then** $newf \leftarrow oldf$ **else** $newf \leftarrow f_{oldf}$; **end if**
88: **if** $L_v \neq 0$ **then**
89: **if** $\text{ROUNDDISTANCE}(v, newf, L_v+1) = \gamma$ (resp. β) **then**
90: $\tau' \leftarrow \gamma$ (resp. β)
91: **else** (*$\text{ROUNDDISTANCE}(v, newf, L_v+1) = \alpha$*)
92: $(anc, k) \leftarrow \text{FINDLOWER}(f_v, L_v)$
93: **if** $L_{anc} = k+1$ and $\text{ROUNDDISTANCE}(anc, f_{anc}, L_{anc}) = \alpha$ **then** $\tau' \leftarrow \beta$ **else** $\tau' \leftarrow \alpha$ **end if**
94: **else**
95: $\tau' \leftarrow \text{ROUNDDISTANCE}(oldf, newf, L_{oldf})$ (*May lead to avoidable updates*)
96: $C_{oldf} \leftarrow C_{oldf} \setminus \{(v, L_v)\}$; remove $\texttt{Cover}_{*,L_v}(v, oldf)$; add $\texttt{Cover}_{\tau',L_v+1}(v, newf)$ in $Cert$
97: remove $\texttt{SCover}_{*,L_v}(v, oldf)$; add $\texttt{SCover}_{\tau'-1/b,L_v+1}(v, newf)$ in $Cert$ (*The new certificate may fail immediately*)
98: $L_v \leftarrow L_v + 1$; $f_v \leftarrow newf$
99: $W \leftarrow \{(w, L_v) \in C_{w'} \cup \{(f_v, L_v)\} : (w', L_v+1) \in N_{f_v} \wedge d(v,w) < cb^{L_v}\}$ (*$(f_v, L_v) \notin W$ if $d(v, f_v) \geq cb^{L_v}$*)
100: $N_v \leftarrow N_v \cup W \cup \{(v, L_v)\}$
101: **for each** $(w, L_v) \in W$ **do** $N_w \leftarrow N_w \cup \{(v, L_v)\}$ **end for**
102: $C_{newf} \leftarrow C_{newf} \cup (v, L_v)$
103: **if** $newf$ is the root and $L_{newf} = L_v$ **then** increment L_{newf} **end if**