Contributions to temporal graph theory and mobility-related problems

Jason Schoeters

université de BORDEAUX

LABORATOIRE BORDELAIS DE RECHERCHE EN INFORMATIQUE LaBRI

THESIS PRESENTED

TO OBTAIN THE DEGREE OF

DOCTOR OF

THE UNIVERSITY OF BORDEAUX

Doctoral School of Mathematics and Computer Science

Specialization Computer Science

Jason Schoeters

CONTRIBUTIONS TO TEMPORAL GRAPH THEORY AND MOBILITY-RELATED PROBLEMS

Under supervision of Arnaud Casteigts

Presented on the 29th of March, 2021

Jury members:

Quentin Bramas Associate Professor, University of Strasbourg, France ..... Examinator

Arnaud Casteigts Associate Professor, University of Bordeaux, France ..... Advisor

Paola Flocchini Professor, University of Ottawa, Canada ..... Reviewer

Nicolas Hanusse Senior Researcher, CNRS, France ..... President

Ralf Klasing Senior Researcher, CNRS, France ..... Examinator

Rolf Niedermeier Professor, TU Berlin, Germany ..... Reviewer

Eric Sanlaville Professor, University of Le Havre, France ..... Reviewer

Title: Contributions to temporal graph theory and mobility-related problems
Abstract: In this thesis, we are interested in research questions that pertain respectively to temporal graphs, to mobility, as well as to the interaction between the two. The problem we consider on temporal graphs is motivated by a 20-year old open question, namely what the analog definition of a spanning tree in temporal graphs is. Our main result in this topic is to show that, even though sparse spanners do not exist in general temporal graphs, sparse spanners exist in significant particular cases. On the other end of the field of dynamic networks, we study the design of physical movements, which led us to consider a discrete model of acceleration called racetrack and to revisit the traveling salesperson problem (TSP). The questions of movement design on one hand, and temporal graphs on the other, end up being in strong interaction when considering the execution of distributed algorithms in a MANET scenario. In this context, the third contribution consists of a software package proposing mobility models that induce temporal graph properties in the resulting communication network.
Keywords: temporal graph theory, motion planning, algorithms

Titre : Contributions à la théorie des graphes temporels et aux problèmes de mobilité
Résumé : Dans cette thèse, nous nous intéressons à des questions de recherche qui concernent respectivement les graphes temporels, la mobilité, ainsi que l'interaction entre les deux. Le problème que nous considérons sur les graphes temporels est motivé par une question ouverte depuis 20 ans, à savoir quelle est la définition analogique d'un arbre couvrant dans les graphes temporels. Notre principal résultat dans ce sujet est de montrer que, même si des spanners peu denses n'existent pas dans les graphes temporels en général, ces spanners existent cependant dans des cas particuliers significatifs. À l'autre bout du champ des réseaux dynamiques, nous étudions la conception des mouvements physiques, en considérant un modèle d'accélération discret appelé racetrack dans le contexte du problème du voyageur de commerce (TSP). Les questions de conception de mouvement d'une part, et de graphes temporels d'autre part, sont en forte interaction lorsque l'on considère l'exécution d'algorithmes distribués dans un scénario MANET. Dans ce contexte, la troisième contribution consiste en un progiciel proposant des modèles de mobilité qui induisent des propriétés de graphe temporel dans le réseau de communication résultant.
Mots clés : Théorie des graphes temporels, planification de mouvement, algorithmique

---

Research unit
Univ. Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR5800, F-33400 Talence, France

Contents

Acknowledgements

My acknowledgements are given in three languages. Starting with English, concerning mostly work-related mentions, then French, for Elodie and friends, and finishing with Dutch, for family. Some people may be mentioned more than once.

My advisor Arnaud Casteigts deserves my first thanks. Having guided and advised me over these last years may not have always been easy. I consider myself lucky to have had an advisor who would always make time for his students. I hope to be as curious, creative, and successful a researcher as Arnaud.

I had the honor of having a high-quality multi-disciplined jury who examined this document as well as my defense. Thank you to the members composing this jury, being Quentin Bramas, Arnaud Casteigts, Paola Flocchini, Nicolas Hanusse, Ralf Klasing, Rolf Niedermeier, and Eric Sanlaville. Thanks to Isabelle Bordas for the organization.

I had the pleasure to have worked with the following researchers, engineers and students: Joseph G. Peters, Mathieu Raffinot, Rémi Laplace, Luiz Fernando Afra Brito, Luis Fredes, Valentin Pasquale, Clément Larue and Ladislav Stacho. Joseph stands out among these people, as he acted almost as a second advisor during my time spent as a visiting graduate student at Vancouver's Simon Fraser University.

Others who had a notable influence on my three years as a graduate student, include Lamine Lamali, Ralf Klasing, Cyril Gavoille, Marc Zeitoun and Sofian Maabout.

In terms of teaching, I'd like to thank Guy Melançon who introduced me to computer science, as well as Carole Blanc for being an exceptional teacher and role model.

A special thanks to Brady Haran, creator of Numberphile, a YouTube channel that explores topics from a variety of fields of mathematics, of which I am a big fan.

Thanks to coffee for the energy, to Stack Overflow for the daily troubleshooting, and to "Jason of tomorrow" to whom I delegated most of the work.

Le plus grand merci est pour Elodie Gaudry, ma compagne. Même après toutes ces années, je passerais une thèse pour l'impressionner. Il va falloir trouver autre chose d'impressionnant maintenant, comme une medaille olympique, un prix Nobel, ou encore une boîte remplie de chocolats.
J'en profite pour remercier sa famille également, notamment Nadia et Christophe Gaudry, ainsi que Hélène et Hugo Van Espen, pour de bons moments passé en bonne compagnie, et souvent autour de bons repas.
Un merci va à mes cobureaux Rohan Fossé et Paul Ouvrard. Entre le code à Roro, et les preuves à Paulo, il y avait de quoi s'entraider, et de quoi rigoler. Luis Fredes est un cobureau informel, car techniquement un mur nous sépare (à démolir ?). La légende dit que la bière a meilleur goût une fois que l'on est docteur, mais plus d'expériences sont nécessaires avant de pouvoir attaquer cette conjecture.
Un autre merci va à Clément Larue, qui étudie des arbres n'ayant pas de sommets ou d'arêtes, mais qui ont néanmoins des feuilles. Ceci n'a bien entendu aucun sens, ce que je répète sans cesse pendant nos réunions hebdomadaires à discuter de l'état de l'art.
Merci à Rocket League, Duvel et au Taco'Tac de Talence d'avoir sponsorisé ces réunions.
J'aimerais remercier également d'autres doctorants (ou docteurs entre temps), que j'ai eu le plaisir de connaître ou de rencontrer au cours des dernières années : Karim Alami, Karim Aderghal, Trang Ngo, Christelle Al Hasrouty, Simon Da Silva, Mathias Lacaud, Alexandre Blanché, Henri Derijcke, Dimitri Lajou, Tobias Castanet, Antonin Lentz, Yessin Neggaz, Mohammed Senhaji, Théo Pierron, Giovanni Farina, Yackolley Amoussou-Guenou, Gewu Bu, Kawtar Lasri, Guillaume Marques et Peter Bradshaw.
Bizarre que cela puisse paraître, j'ai également des amis avec qui je peux parler d'autre chose que de science et de recherche. Merci à eux de me permettre de m'échapper à des moments : Teddy Jamin, Lucille Bozetto, Anique Van De Put, Faycel Grine, Florian Marcel, Rodney Embola, Pierre-Louis Euvrard, Dimitri Ranchou, Nicolas Marcy et Daniela Cortés Robles.
À l'autre bout du monde, ce rôle d'ami a été pris par Larissa Nobre Campos et Giampaolo Lepore, ainsi que par Mitsuki et Marek (dont je n'ai pas les noms de famille).
Le plus petit merci va à mon chat, Freddie (qui n'a pas de nom de famille).

Hartelijk dank aan mijn dichte familie, bij wie ik altijd welkom ben wanneer ik België en omstreken bezoek, en vice versa, wie altijd welkom zijn bij mij thuis:
Mijn ouders Dominique Bastien en Gerrit Donny.
Rudy Schoeters en Sabine Smeulders, mijn tweede paar ouders.
Mijn drie “kleine” zussen Jessie, Lotte, en Hanne Schoeters.
Yvette Vermeylen, mijn meter en onthaalmoeder.
Goede vrienden Steven Dams en Julie Van Agtmael.
En mijn vierpotige vriend Max, die misschien de volgende doctor in de familie zal zijn.

Voor mijn moeder.

Chapter 1

Introduction

I still keep asking these "how" and "why" questions.
Occasionally, I find an answer.

— Stephen Hawking

The study of dynamic networks has gained increasingly more interest over recent years, motivated mainly by emerging technological contexts like vehicular networks, robots, and drones. Other applications include social networks analysis and influence spreading, and more recently epidemics analysis such as trying to understand the spread of infection (for example concerning COVID-19). The presence of communication or interaction links in such networks may vary over time due to a variety of reasons, ranging from the removal of physical Ethernet cables and security measures to the effects of social distancing. Concerning Mobile Ad hoc Networks (MANETs), in which mobile agents are able to communicate when in communication range of each other, controlling mobility of these agents is key for controlling the entirety of the network. Optimal control over a mobile agent's movements in itself has long been studied in physics and control theory, and has several applications in robotics concerning autonomy and automation, so as to minimize any needed human intervenience.

In this thesis, we are interested in several aspects of these topics. More precisely, we consider research questions that pertain respectively to temporal graphs, to mobility, as well as to the interaction between the two. The problem we consider on temporal graphs is motivated by a 20-year old open question, namely what the analog definition of a spanning tree in temporal graphs is. It turns out that this notion has puzzled researchers ever since, due to the fact that even analogs of sparse spanners do not exist in general

temporal graphs. Our main result in this topic is to show that sparse spanners however exist in significant particular cases. On the other end of the field of dynamic networks, motivated mostly by the particular case of MANETs, we study the design of physical movements and how this may be approached in an algorithmic fashion, instead of through typical control theory methods. This led us to consider a discrete model of acceleration called `racetrack` and to revisit the traveling salesperson problem (TSP) with this additional constraint. While seemingly unrelated, the questions of movement design on one hand, and temporal graphs on the other, end up being in strong interaction when considering the execution of distributed algorithms in a MANET scenario. In an effort to connect the two ends of the spectrum, our third contribution considers the interaction between the two, by means of the design of collective movements that induce temporal graph properties in the resulting communication network. This third topic is developed as a software package, available for researchers in distributed computing. We now describe in more details each of these contributions.

1.1 Summary of contributions

1.1.1 Preliminaries

In Chapter 2, we start by giving some basic definitions, standard notations, and well-known results from graph theory, computational complexity, algorithmics and motion planning. These are necessary for the understanding of the following more precise results in this document. Basic concepts from temporal graph theory are introduced here as well.

1.1.2 Temporal cliques admit sparse spanners

More and more standard graph theory structures, as well as their corresponding computational problems, are being extended into temporal graph problems, and as such are often redefined over time rather than at any given instant. It is even possible for a relatively simple structure to be naturally extended in a dynamic environment and become a relatively complex structure. An example of this is a spanning tree. Indeed, although the size of a spanning tree in any standard graph is considered trivial (as well as its computation), several open questions surround the subject of spanners in temporal graphs, including their size and computation, even when restrained to simple topologies.

Inspired by a question from Kempe, Kleinberg, and Kumar in [60], previous work on temporal spanners (in particular, [11]) has shown that there exist some infinite families

of temporal graphs admitting no sparse temporal spanner, where *sparse* refers to a subquadratic number of edges. In Chapter 3, we present our results on temporal spanners in complete temporal graphs. More precisely, we provide (among other results) the first positive answer in this line of research, showing that temporal cliques always admits a spanner of size $O(n \log n)$ edges. We prove this through a multi-step constructive algorithm. This work has been published in the proceedings of ICALP 2019 [25] and the full version is currently in minor revision for the JCSS journal.

1.1.3 VectorTSP: A Traveling Salesperson Problem with Racetrack-like acceleration constraints

Mobility models often do not take into account acceleration (and inertia) forces, and as a result are not adapted to deal with situations in which these may play a significant role. The few models that do propose acceleration are often of a continuous nature, which may be more prone to the fields of control theory and analytic functions. An old paper-and-pencil game named Racetrack, defined by Martin Gardner in [45], proposes some simple, yet intriguingly accurate mobility constraints for a player's race car, in which velocity, acceleration and inertia all are simulated through a small set of rules, which specifies that the velocity of the vehicule can only be modified by some discrete amount in each time step, in each dimension. Being by definition discrete, these constraints are naturally inclined to algorithmic investigation.

In Chapter 4, we present the Vector Traveling Salesman Problem, an adaptation of the standard Traveling Salesman Problem, in which we add the Racetrack acceleration constraints. Even though these Racetrack constraints have been investigated already in the setting of trajectory optimization, we obtain a number of interesting results when applied to the visit of multiple points in a non-predetermined order (*i.e.*, the TSP). Our results include insights and comparisons (differences even) between standard Euclidean TSP and VectorTSP's respective solutions. We include a proof of NP-hardness, reducing Exact Set Cover to VectorTSP, as well as a reduction from VectorTSP to Group TSP. Through an adapted TSP algorithm, using an A* oracle, we demonstrate that VectorTSP is genuinely distinct from other known versions of TSP and we quantify the gap experimentally. This work has been published in the proceedings of ALGOSENSORS 2020 [27] and the full version is about to be submitted to a journal.

1.1.4 Temporal graph properties induced by collective mobility

Standard graph properties can result in a multitude of possible extensions when considered in a temporal context. A good example is the standard notion of connectivity, extending to temporal connectivity, constant connectivity or windowed connectivity, to name but a few. In fact, a whole hierarchy of different types of temporal connectivity has been identified by Casteigts *et al.* in [21] (and recently extended by Casteigts in [18]), identifying over a dozen temporal properties that had been studied and used in the distributed computing and networking literature (*e.g.* see Altisen *et al.*'s recent work [4]). These properties often correspond to necessary or sufficient conditions for some algorithm or problem.

In Chapter 5, we propose multiple mobility models for MANET, which induce and/or exclude several temporal properties on the underlying temporal graph, through governing the collective movements of mobile entities. This contribution is more practical and offered as a software package coded in Java using the JBotSim library, a simulation library for distributed algorithms in dynamic networks (first presented by Casteigts *et al.* in [23]). Users are able to use this package as a black box test environment to visualize, as well as verify, their distributed algorithms relying on a specific temporal property.

1.1.5 Conclusion

We conclude in Chapter 6 with a collection of open questions, remarks and proposed directions for future work concerning the multiple research items presented in this thesis.

1.2 Order of chapters

The order in which the chapters are read, after Chapter 2, is up to the reader, each chapter being self-contained. The order chosen however, starts with the more abstract and theoretical works and finishes with the more practical results. It also happens to be the (more or less) chronological order in which the presented research was conducted.

Chapter 1

Introduction (en français)

Je continue à poser ces questions de "comment" et "pourquoi". Parfois, je trouve une réponse.

— Stephen Hawking

L'étude des réseaux dynamiques suscite de plus en plus d'intérêt au cours des dernières années, motivée principalement par des technologies émergentes comme les réseaux véhiculaires, les robots, et les drones. D'autres applications sont l'analyse des réseaux sociaux et la propagation de l'influence, et plus récemment l'analyse des épidémies comme essayer de comprendre la propagation de l'infection (par exemple concernant le COVID-19). La présence de liens de communication ou d'interaction dans ces réseaux peut varier au cours du temps pour diverses raisons, allant de la suppression des câbles Ethernet physiques jusqu'aux mesures de sécurité comme la distanciation sociale. Concernant les réseaux mobiles ad hoc (MANET), dans lesquels les agents mobiles sont capables de communiquer lorsqu'ils sont à portée de communication les uns des autres, le contrôle de la mobilité de ces agents est essentiel pour contrôler l'ensemble du réseau. Le contrôle optimal des mouvements d'un agent mobile en lui-même a longtemps été étudié en physique et en théorie du contrôle, et a plusieurs applications en robotique concernant l'autonomie et l'automatisation, afin de minimiser toute intervention humaine nécessaire.

Dans cette thèse, nous nous intéressons à plusieurs aspects de ces sujets. Plus précisément, nous considérons des questions de recherche qui concernent respectivement les graphes temporels, la mobilité, ainsi que l'interaction entre les deux. Le problème que nous considérons sur les graphes temporels est motivé par une question ouverte depuis 20 ans, à savoir quelle est la définition analogique d'un arbre couvrant dans les graphes

temporels. Cette notion a intrigué les chercheurs, du fait que même les analogues des spanners peu denses n'existent pas dans les graphes temporels en général. Notre principal résultat dans ce sujet est de montrer que des spanners peu denses existent cependant dans des cas particuliers significatifs. À l'autre bout du champ des réseaux dynamiques, motivés principalement par le cas particulier des MANET, nous étudions la conception des mouvements physiques et comment cela peut être abordé de manière algorithmique, plutôt que par des méthodes classiques de la théorie du contrôle. Ceci nous a conduit à considérer un modèle d'accélération discret appelé `racetrack` et à revoir le problème du voyageur de commerce (TSP) avec cette contrainte supplémentaire. Bien qu'apparemment sans rapport, les questions de conception de mouvement d'une part, et de graphes temporels d'autre part, finissent par être en forte interaction lorsque l'on considère l'exécution d'algorithmes distribués dans un scénario MANET. Dans un effort de connecter les deux extrémités du spectre, notre troisième contribution considère l'interaction entre les deux, au moyen de la conception de mouvements collectifs qui induisent des propriétés de graphe temporel dans le réseau de communication résultant. Ce troisième thème est développé sous forme de progiciel, disponible pour les chercheurs en informatique distribuée. Nous décrivons maintenant plus en détail chacune de ces contributions.

1.1 Résumé des contributions

1.1.1 Préliminaries

Dans Chapitre 2, nous commençons par donner quelques définitions de base, des notations standard et des résultats bien connus de la théorie des graphes, de la complexité de calcul, de l'algorithmique et de la planification de mouvement. Celles-ci sont nécessaires pour comprendre les résultats plus poussés dans ce document. Les concepts de base de la théorie des graphes temporels sont également introduits ici.

1.1.2 Les cliques temporelles admettent des spanners peu denses

De plus en plus de structures de théorie des graphes standards, ainsi que leurs problèmes de calcul correspondants, sont étendues à des problèmes de graphes temporels, et en tant que telles sont souvent redéfinies au fil du temps plutôt qu'à un instant donné. Il est même possible qu'une structure relativement simple soit naturellement étendue dans un environnement dynamique et devienne une structure relativement complexe. Un exemple de ceci est un arbre couvrant. En effet, bien que la taille d'un arbre couvrant dans tout

graphe standard soit considérée comme triviale (ainsi que son calcul), plusieurs questions ouvertes entourent le sujet des spanners dans les graphes temporels, y compris leur taille et leur calcul, même lorsqu'ils sont limités à des topologies simples.

Inspiré d'une question de Kempe, Kleinberg et Kumar dans [60], des travaux antérieurs sur les spanners temporels (en particulier, [11]) ont montré qu'il existe des familles infinies de graphes temporels n'admettant pas de spanner temporel peu dense, où *peu dense* fait référence à un nombre sous-quadratique d'arêtes. Dans Chapitre 3, nous présentons nos résultats sur les spanners temporels dans des graphes temporels complets. Plus précisément, nous fournissons (entre autres résultats) la première réponse positive dans cette ligne de recherche, montrant que les cliques temporelles admettent toujours un spanner de taille $O(n \log n)$ arêtes. Nous le prouvons grâce à un algorithme constructif en plusieurs étapes. Ce travail a été publié dans les actes de ICALP 2019 [25] et la version complète est actuellement en révision mineure pour le journal JCSS.

1.1.3 VectorTSP: un problème de voyageur de commerce avec des contraintes d'accélération Racetrack

Les modèles de mobilité ne prennent souvent pas en compte les forces d'accélération (et d'inertie) et ne sont donc pas adaptés pour faire face à des situations dans lesquelles celles-ci peuvent jouer un rôle significatif. Les quelques modèles qui proposent une accélération sont souvent de nature continue, ce qui peut être plus enclin aux domaines de la théorie du contrôle et des fonctions analytiques. Un vieux jeu de papier et crayon nommé Racetrack, défini par Martin Gardner dans [45], propose des contraintes de mobilité simples mais étonnamment précises pour la voiture de course d'un joueur, dans lesquelles la vitesse, l'accélération et l'inertie sont toutes simulées à travers un petit ensemble de règles, qui spécifie que la vitesse du véhicule ne peut être modifiée que d'une certaine quantité discrète à chaque pas de temps, dans chaque dimension. Étant par définition discrètes, ces contraintes sont naturellement portées à l'investigation algorithmique.

Dans Chapitre 4, nous présentons le Vector Travelling Salesman Problem, une adaptation du Travelling Salesman Problem standard, dans lequel nous ajoutons les contraintes d'accélération Racetrack. Même si ces contraintes Racetrack ont déjà été étudiées dans le cadre de l'optimisation de trajectoire, nous obtenons un certain nombre de résultats intéressants lorsqu'ils sont appliqués à la visite de plusieurs points dans un ordre non prédéterminé (*i.e.*, le TSP). Nos résultats incluent des aperçus et des comparaisons (même des différences) entre les solutions respectives du TSP euclidien standard et de VectorTSP. Nous incluons une preuve de NP-complétude, réduisant le problème de couverture par

ensembles exacte à VectorTSP, ainsi qu'une réduction de VectorTSP à Groupe TSP. Grâce à un algorithme TSP adapté, utilisant un oracle A*, nous démontrons que VectorTSP est véritablement distinct des autres versions connues de TSP et nous quantifions l'écart expérimentalement. Ce travail a été publié dans les actes d'ALGOSENSORS 2020 [27] et la version complète est sur le point d'être soumise à un journal.

1.1.4 Propriétés de graphes temporels induits par la mobilité collective

Les propriétés de graphe standard peuvent donner lieu à une multitude d'extensions possibles lorsqu'elles sont considérées dans un contexte temporel. Un bon exemple est la notion standard de connectivité, qui s'étend à la connectivité temporelle, à la connectivité constante ou à la connectivité fenêtrée, pour n'en citer que quelques-uns. En fait, toute une hiérarchie de différents types de connectivité temporelle a été identifiée dans [21] (et récemment étendue dans [18]), identifiant plus d'une douzaine de propriétés temporelles qui sont étudiées et utilisées dans l'informatique distribuée et les réseaux. Ces propriétés correspondent souvent à des conditions nécessaires ou suffisantes pour un algorithme ou un problème.

Dans Chapitre 5, nous proposons de multiples modèles de mobilité pour MANET, qui induisent et/ou excluent plusieurs propriétés temporelles sur le graphe temporel sous-jacent, en gouvernant les mouvements collectifs des entités mobiles. Cette contribution est plus pratique et proposée sous forme de progiciel codé en Java à l'aide de la bibliothèque JBotSim, une bibliothèque de simulation d'algorithmes distribués dans des réseaux dynamiques. Les utilisateurs peuvent utiliser ce package comme un environnement de test de boîte noire pour visualiser, ainsi que vérifier, leurs algorithmes distribués qui s'appuyent sur une propriété temporelle spécifique.

1.1.5 Conclusion

Nous concluons dans Chapitre 6 en regroupant les questions ouvertes, ainsi que des remarques ou directions intéressantes pour du futur travail concernant les sujets de recherche présentés dans cette thèse.

1.2 Ordre des chapitres

L'ordre dans lequel les chapitres sont lus, après Chapitre 2, appartient au lecteur, chaque chapitre étant indépendant. Cependant, l'ordre choisi commence par les travaux les plus

abstraits et théoriques et se termine par les résultats les plus pratiques. Il se trouve également qu'il s'agit de l'ordre (plus ou moins) chronologique dans lequel la recherche présentée a été menée.

Chapter 2

Preliminaries

Computer science is no more about computers than astronomy is about telescopes.

— Edsger Dijkstra

Contents

In this chapter, we present necessary basic definitions, notations and known results for the general understanding of this document's methods and results. More specific information is given in the concerned chapters. We start by presenting several properties and structures from graph theory. Some of these properties and structures are then revisited in the context of temporal graphs, as most of these are affected by the addition of the dimension of time. We then present some notations surrounding algorithmic analysis and important classes of computational complexity, with basic examples. Finally, we give some standard methods and algorithmic aspects surrounding motion planning.

2.1 Standard graphs

Here, we review some main concepts of **graph theory**, the study of **graphs**. We will commonly call graphs **standard graphs** or **static graphs** to differentiate them from temporal graphs. When the context is clear enough, we will just call these graphs.

2.1.1 Definitions

We recall the definition of a graph. To be precise, an undirected, unweighted, and simple graph. Generalizations are discussed later in Section 2.1.3, and the particular generalization of temporal graphs in Section 2.2.

Definition 1. *A graph is a pair (V, E), where V is a set of vertices, and E a collection of (unordered) pairs of vertices, called edges.*

Informally, a graph is composed of some dots with some lines connecting these dots (see Figure 2.1). These dots represent the **vertices**, and the lines represent the **edges** (also called nodes and links respectively in some contexts). When generally speaking about a graph, one often uses n as the number of vertices, and m as the number of edges.

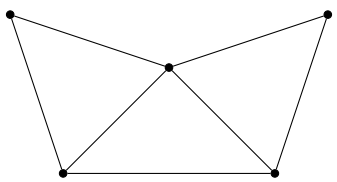


Figure 2.1: Example of a graph with 5 vertices and 7 edges.

An edge $e = \{u, v\}$ induces a symmetric relation over its **endpoints** $u, v \in V$, and one says that these vertices are **adjacent** or **neighbors**. In a similar manner, edge $e = \{u, v\}$ is said to be **incident** to u and to v, as well as to any other edge containing u or v.

The **degree** of a vertex u, denoted $deg(u)$, is its number of incident edges, or equivalently here, its number of neighbors.

A **path** joining two vertices u and v (also called the **endpoints** of the path) is a sequence of distinct adjacent vertices $(u, u_0, u_1, ..., u_k, v)$, or equivalently, a sequence of distinct incident edges $(\{u, u_0\}, \{u_0, u_1\}, ..., \{u_k, v\})$. The length of a path is the number of edges it contains. Note that a path of length 1 is simply an edge.

The **distance** between two vertices u and v, noted $d(u, v)$, is defined as the length of a shortest path between u and v. If there is no path between u and v, then $d(u, v) = \infty$.

If there exists at least one path between every pair of vertices of the graph, the graph is said to be **connected**. If there exists no path between some pair of vertices, then the graph is **disconnected**.

A **cycle** is a path in which the first and last vertices are equal, so $\{u, u_0, u_1, ..., u_k, u\}$.

A **subgraph** G' of a graph $G = (V, E)$ is composed of a subset V' of V and a subset E' of E.

Generally speaking, we use the term **structure** to identify a type of subgraph in a graph, whereas the term **property** mostly designates a feature or characteristic of either a vertex, an edge, a subgraph, or a graph.

We give two last examples of structures in graphs, which we will use as examples or references during the rest of this chapter, and which are related to the subjects studied thereafter. The first of these is a spanning tree.

Definition 2. *A **spanning tree** of a graph $G = (V, E)$ is a connected subgraph $T = (V, E' \subseteq E)$ which admits no cycles.*

The term **spanning** refers to the fact that a structure contains all vertices. Note how removing an edge from a spanning tree T results in a disconnected graph. We thus obtain the following fact.

Fact 1. *A spanning tree T is a minimal subgraph of the graph G that maintains connectivity.*

In fact, the size of the spanning tree (so its number of edges) is also minimum, since all spanning trees have the same number of edges (namely, $n - 1$ edges).

Another useful structure for the rest of this section is that of a Hamiltonian cycle.

Definition 3. *A **Hamiltonian cycle** of a graph $G = (V, E)$ is a cycle which visits each vertex exactly once.*

Many more properties and structures can be defined, some more complex than others (such as the treewidth of a graph, which is outside of the scope of this document).

2.1.2 Graph classes

Many different types of graphs exist, and can be grouped in distinct sets according to some shared **properties** (also called graph invariants), which can then be subdivided into several subsets, and so on. Such sets are often referred to as **graph classes** or **graph families**. Note that graph classes can be of infinite size, for example the class of all connected graphs.

Many such graph classes exist for the purpose of adapting graphs to specific situations. Another purpose of some families of graphs is that they represent the graphs in which some computational problem may be solved easily, or reversely, be hard to solve. More on this in Section 2.3.

Following are basic examples of graph classes (see Figure 2.2).

A **path** graph is defined as a connected graph with exactly two vertices of degree 1 (its endpoints), and the other vertices of degree 2.

Remark 1. *These path graphs are not to be confused with the concept of a path between two vertices, presented before. Indeed, although similar, the former is a graph family and the latter a structure found in graphs. The same holds for cycles and cycle graphs.*

A **cycle** graph is a connected graph in which all vertices have degree 2. It is thus equivalent to a path graph in which the endpoints are identical (thus removing it from the path family since it has no more vertices of degree 1).

A **star** is a connected graph of n vertices with one vertex having degree $n - 1$, and the other vertices having degree 1.

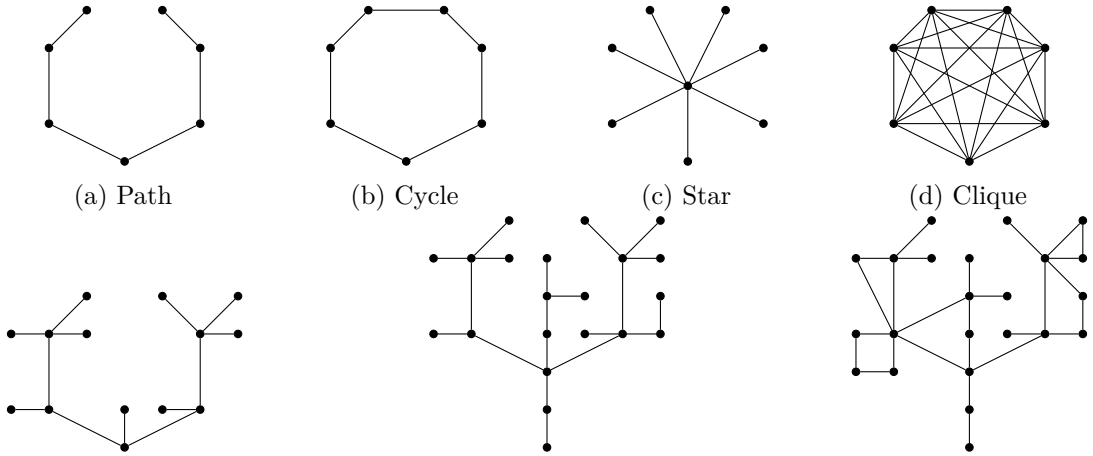
A **clique**, or **complete** graph, is a connected graph of n vertices with all vertices having degree $n - 1$.

A **caterpillar** is a path graph in which one adds any number of vertices of degree 1 adjacent to the path (or rather to the vertices of the path). These added vertices are sometimes referred to as **satellite** vertices.

A **tree** is a connected graph without cycles. Vertices of degree 1 in a tree graph are referred to as **leaves**. Note how the only spanning tree of a tree graph is itself.

A **forest** is a (possibly disconnected) graph without cycles. Informally, a forest graph can be seen as a tree graph in which one removes some edges. Equivalently, a forest graph can be created by taking some tree graphs and referring to them as one graph.

As a last example, a **cactus** is a connected graph in which any two cycles have no edge in common. Informally, a cactus can be seen as an inflated tree.



(a) Path (b) Cycle (c) Star (d) Clique

(e) Caterpillar (f) Tree (g) Cactus

Figure 2.2: Example graphs of multiple graph classes.

2.1.2.1 Hierarchy of graph classes

It is useful to compare and organize different graph classes for a multitude of reasons. Citing *Information System on Graph Classes and their Inclusions* [35] (also known as ISGCI or `graphclasses.org`), it can be useful to:

- Check the relation between graph classes and get a witness for a result
- Draw clear inclusion diagrams
- Find the P/NP boundary for a problem (more on P/NP in Section 2.3)
- Find references on classes, inclusions and algorithms

We give an example of such a hierarchy surrounding the presented classes (see Figure 2.3).

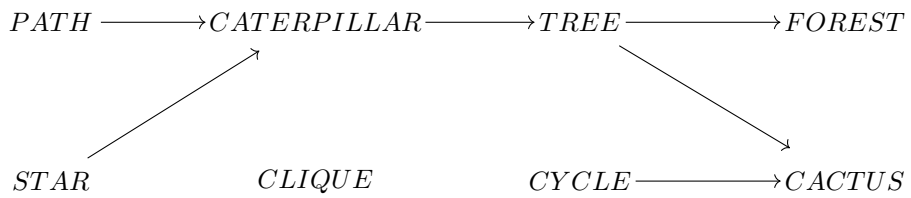


Figure 2.3: An example of a hierarchy surrounding some simple graph classes.

In the presented hierarchy, the set denoted by $PATH$ refers to the family of paths, the one denoted by $STAR$ to the family of stars, *etc.* Inclusion of some set X in a set Y is portrayed by an arrow from X to Y. Multiple relations are thus made clear through this hierarchy. We give some of these relations:

- Path graphs are included in caterpillar graphs. Indeed, a caterpillar without any satellite vertices is still a caterpillar, but in particular, is also a path.
- Star graphs are tree graphs, as star graphs admit no cycles.
- Cycle graphs can be constructed as a cactus as follows. Take the tree graph consisting of one edge. Now, inflate it to a cycle of any size. This shows cycles are a special case of cacti. Also, among the presented classes, cycle graphs are only comparable to cactus graphs.
- Complete graphs (or cliques) are not comparable to any of the presented graph classes. Indeed, cliques are no special case of paths, stars, *etc.*, nor vice versa. Note however that there is a (very small) overlap between cliques and cycles, concerning the cycle of order 3.

Note how all the classes present in Figure 2.3 are naturally included in the set of all graphs, even though this inclusion is not portrayed.

2.1.3 Generalizations of graphs

It is possible to **generalize** graphs to represent situations that standard graphs might not. In terms of generalizing graphs, multiple models exist, and one could probably invent many more.

Following are multiple examples of graph generalizations (see Figure 2.4).

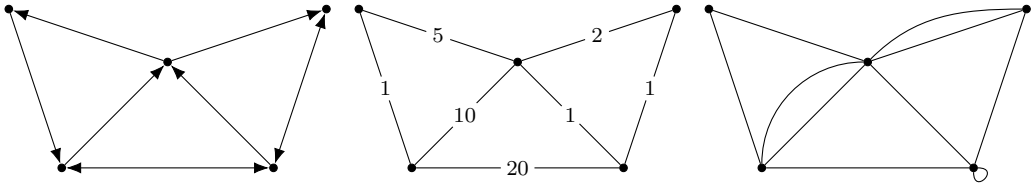


Figure 2.4: Examples of a directed graph, a weighted graph, and a non-simple graph respectively.

One model is that of a **directed** graph, where edges may no longer be symmetric between endpoints and thus may have a direction, from a **source** vertex, towards a **destination** vertex. Such edges are then named directed edges, or simply **arcs**. This model may more accurately represent road maps with possible one-way streets for example.

Another model is a graph on which one can label each edge e with a number, called the **weight** of the edge, denoted $w(e)$. Such a graph is then called a **weighted** graph. The weight of an edge can represent, for example, the capacity of a communication link, or its latency.

Finally, one might consider allowing **loops**, *i.e.* edges that join a vertex u to itself, or **multi-edges**, *i.e.* more than one edge joining a pair of vertices. When no loops or multi-edges are present in a graph, one calls such a graph a **simple** graph, otherwise the graph is called **non-simple**.

Often, graphs can be seen as special cases of these generalizations. Regarding the examples of generalizations given, an undirected, unweighted, simple graph can be seen as:

- a directed graph where all arcs are bidirectional.
- a weighted graph with all weights equal to 1 (or 0 depending on the context).
- a non-simple graph without loops or multi-edges.

2.2 Temporal graphs

Temporal graphs are another generalization of graphs, although an important one concerning this document's work.

Temporal graphs are also known under the names of dynamic graphs [51], evolving graphs [44], time-varying graphs [21], stream graphs and link streams [64], and possibly more.

For most purposes, these represent the same generalization of graphs (the main issue arises when time is continuous, see *e.g.* [19]).

Informally, a temporal graph is a graph on which edges can appear and disappear. We will use a simple definition which suits us regarding the work presented in this document, and rely on a combination of terminologies from [21] and [90].

2.2.1 Definitions

Definition 4. *A temporal graph is a pair (G, λ), where $G = (V, E)$ is a graph, and $\lambda : E \to 2^{\mathbb{N}}$ a mapping that assigns to each edge of G a (non-empty) set of discrete presence times.*

A temporal graph is thus a graph on which edges can appear and disappear over some defined timeline, also called the **lifetime** of the graph. The lifetime of the graph may either be finite or infinite. In this document, time is **discrete**, unless explicitly stated otherwise, and the lifetime can thus be considered as a subinterval of the natural numbers.

Temporal graphs can be seen as a generalization of graphs, because any graph is simply some temporal graph whose edges are always present.

One may visualize a temporal graph $\mathcal{G} = (G, \lambda)$ in the following manner. Take the induced static graph G, also called the **footprint** of $\mathcal{G}$ or underlying graph, and add **time labels** (also called time stamps in some works) on each appearing edge corresponding to the labeling λ (see Figure 2.5).

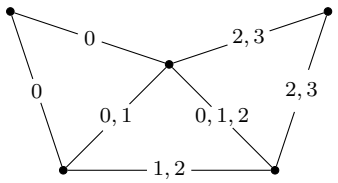
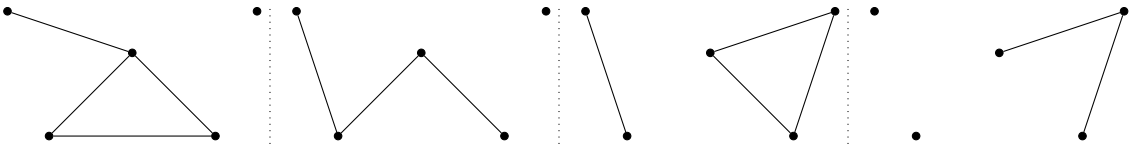


Figure 2.5: Example of a temporal graph, represented through the footprint with time labels on present edges.

These labels thus represent the moment(s) when the corresponding edge is present. Reversely, if some label t is not marked on an edge, it means this edge is not present at time t of the lifetime of the graph. Edges that are not drawn, are simply never present.

Another way to visualize temporal graphs is to look at the corresponding sequence of **snapshots**, *i.e.* static graphs corresponding to each time step t in the lifetime of the graph (see Figure 2.6).



(a) G_0 (b) G_1 (c) G_2 (d) G_3

Figure 2.6: Example of a temporal graph $\mathcal{G}$ (the same one as in Figure 2.5), represented as a sequence of static graphs (G_0, G_1, G_2, G_3).

Different visualizations of temporal graphs add different points of view which can be helpful for reasoning on the graph in question. In this document, we rely mainly on the representation using time labels, although some temporal concepts presented are easier to define or better observed in the sequence of snapshots.

Some notations, properties and structures from static graphs directly adapt to temporal graphs (such as adjacency, incidence, *etc.*), others take on some new meaning, and possibly multiple adaptations. The addition of time also makes way for completely new structures to be studied. Below are the most basic and important structures to understand the rest of this document.

The **eventual footprint** of an infinite lifetime temporal graph is its footprint without non-recurring edges (edges which no longer reappear after some point in time).

A **time edge** (also called a contact) (e, t) is the edge $e = \{u, v\}$ at time $t \in \lambda(e)$. At time t, vertices u and v are thus adjacent, or neighbors.

A **temporal path**, or simply **journey**, from vertex u to vertex v is a sequence of time edges with non-decreasing time labels $((\{u, u_1\}, t_1), (\{u_1, u_2\}, t_2), ..., (\{u_{k-1}, v\}, t_k))$. An example of a journey is given on both models of temporal graph visualization in Figure 2.7.

We say a vertex u can reach vertex v through a journey (or reversely, a vertex v can be reached by vertex u), if a journey from u to v exists.

Even when time edges are symmetrical, *i.e.* the existence of time edge $(\{u, v\}, t)$ implies the existence of time edge $(\{v, u\}, t)$, this is not necessarily the case for journeys. Indeed, unlike paths in static graphs, the existence of a journey from some u to v, does not imply the existence of a journey from v to u. A simple example is the journey given from u to v in Figure 2.7, as no journey exists from v to u.

Two different versions of journeys can be considered. We say journeys are **strict** if only increasing time labels are allowed on the journey's time edges. Allowing for non-decreasing

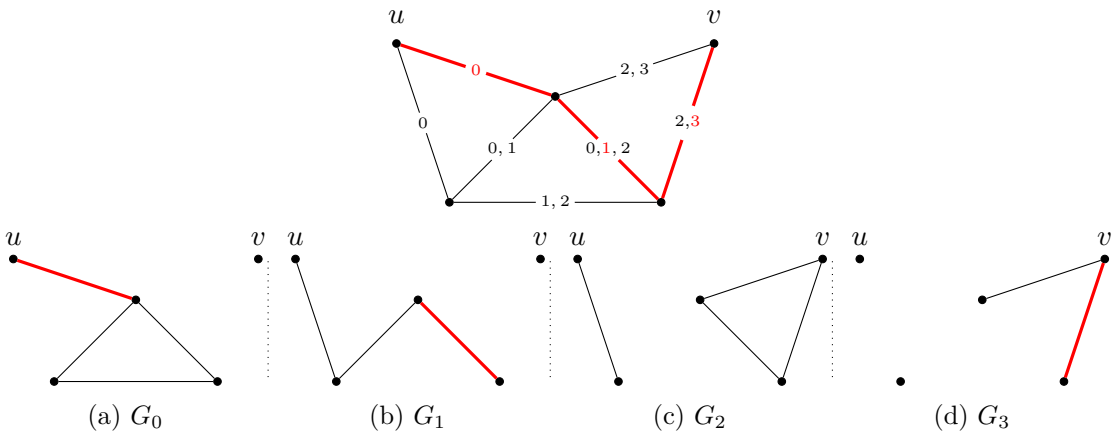


Figure 2.7: Example of a journey from vertex u to vertex v (marked in bold and red).

time labels results in **non-strict** journeys. The journey given in Figure 2.7 is a strict journey.

The **distance** between two vertices u and v can now be defined as, again, the length of a shortest journey between these vertices. However, the term "length of a shortest journey" can now possibly take on multiple meanings. Does it relate to the amount of time edges it contains, or rather the time it takes between the first time edge and the last? Or is it related to the overall arrival time, independently of the amount of edges and total travel time? All these notions and variations of shortest journeys are studied in [20, 74, 15].

A temporal graph is said to be **temporally connected** (also noted $\mathcal{TC}$) if there exists a journey (of the considered type) between each pair of vertices (in both directions).

We note that, somewhat counter intuitively, a temporal graph with a connected footprint is not necessarily temporally connected. Indeed, see again Figure 2.7. This example graph has a connected footprint, however it is not temporally connected, since, as stated before, there exists no journey from v to u.

Temporal connectivity was considered in an early paper by Awerbuch and Even [10] (1984), and studied from a graph-theoretical point of view in the early 2000's in a number of seminal works including Kempe, Kleinberg, and Kumar [61], and Bui-Xuan, Ferreira, and Jarry [15] (see also [74] for an early study of graphs with time-dependent delays on the edges). More recently, it has been the subject of algorithmic studies, such as [3, 11, 86, 22, 85], which consider algorithms for computing structures related to temporal connectivity, and [41, 90] whose work focuses on restricting temporal connectivity

(epidemics, malware spreading, *etc.*). Broad reviews on these topics can be found, *e.g.*, in [21, 54, 69], although the list is non-exhaustive and the literature is rapidly evolving.

A **round trip** is a journey in which the first and last vertices are equal. Note that in most cases, and in contrast to its static counterpart, a round trip only allows for traversal in one direction.

A **temporal subgraph** $\mathcal{G}'$ of a temporal graph $\mathcal{G} = ((V, E), \lambda)$ is composed of a subgraph (V', E') of the footprint (V, E), and for every edge $e \in E'$, a subset $\lambda'(e)$ of $\lambda(e)$.

Lastly, let us look at what happens to the spanning tree in temporal graphs. The spanning tree in static graphs is a tree, meaning it contains no cycles. So one might look into how to extend the concept of a spanning tree to a structure which contains no round trips for example. However, in Chapter 3, we focus on a different extension of the spanning tree. As stated in Fact 1, a spanning tree can be seen as a (minimum) subgraph preserving connectivity in a static graph. We take this point of view of a spanning tree, and adapt it so as to preserve temporal connectivity.

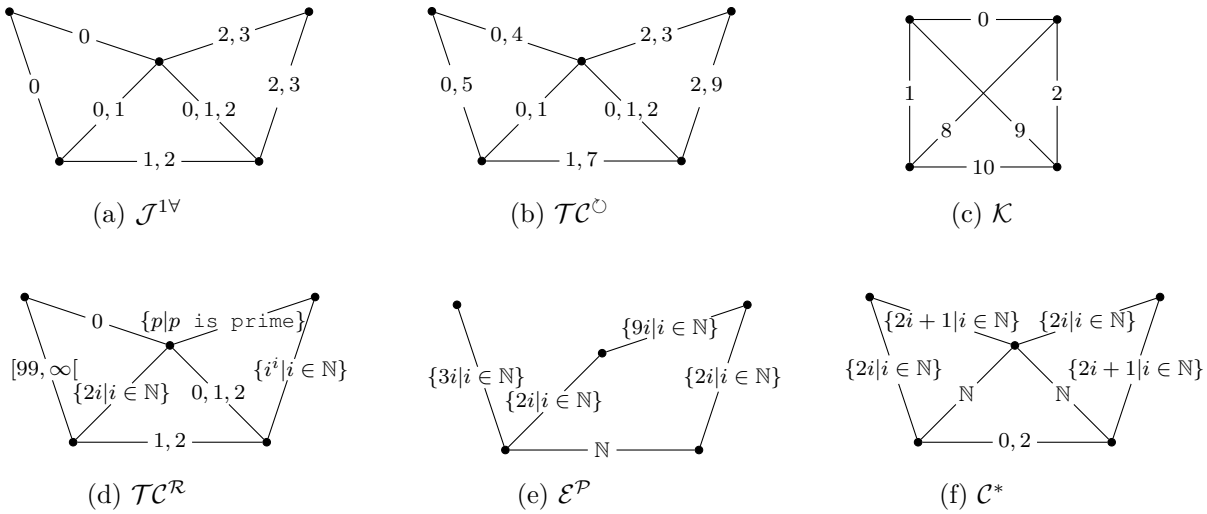
Definition 5. *A **temporal spanner** of a temporal graph $\mathcal{G} = ((V, E), \lambda)$ is a temporally connected subgraph $\mathcal{S} = ((V, E' \subseteq E), \lambda' \subseteq \lambda)$.*

We remark how a spanning tree in static graphs is minimum in size by default. This is not the case however for a temporal spanner (so size in terms of number of edges or time edges), unless we specifically consider a **minimum** temporal spanner.

2.2.2 Temporal graph classes

With the introduction of temporal graphs, various new and natural temporal graph properties can be defined, such as temporal connectivity. The literature quickly got filled with several temporal properties that turned out to be of use, such as the necessary or sufficient condition for some algorithm in distributed computing and networking, to function correctly in temporal graphs. In [21], a dozen temporal properties were identified that have been effectively exploited in the distributed computing and networking literature. These were extended more recently in [18], and renamed using mnemonic symbols. These in turn induce several classes of temporal graphs in which these properties are satisfied (or not). Following are some examples of these temporal graph classes. More context about these classes is given in Chapter 5.

Some of these properties are, by definition, satisfiable in some finite amount of time steps, and thus the evolution of the temporal graph after the property has been satisfied does



(a) $\mathcal{J}^{1\forall}$ (b) $\mathcal{TC}^{\circlearrowleft}$ (c) $\mathcal{K}$

(d) $\mathcal{TC}^{\mathcal{R}}$ (e) $\mathcal{E}^{\mathcal{P}}$ (f) $\mathcal{C}^{*}$

Figure 2.8: Some temporal graphs respecting some temporal properties.

not impact the property itself. An example of such a property is temporal connectivity, as once all vertices have been able to connect over time, the evolution of the graph afterwards no longer matters in regard of this property. In the following, such properties are referred to as **finite properties**.

The property (or class) $\mathcal{J}^{1\forall}$ is when the graph has a **temporal source**, *i.e.* when at least one vertex can reach all the others through a journey.

When every vertex can reach every other vertex, and be reached from that vertex *afterwards*, the graph is said to be **round trip connected**, noted $\mathcal{TC}^{\circlearrowleft}$.

The class $\mathcal{K}$ includes temporal graphs having a complete graph as a footprint. These graphs are simply called **temporal cliques**. In Chapter 3, we focus on the case of temporal cliques, to study and solve a long-standing open question on the existence of sparse temporal spanners.

Other properties are **recurrent**, in the sense that these properties need to be satisfied infinitely often. Such properties can of course only occur in infinite lifetime graphs.

Recurrent temporal connectivity is the most basic of recurrent properties. Noted $\mathcal{TC}^{\mathcal{R}}$, it denotes the property of any vertex being able to reach any other vertex of the graph through a journey, starting after any point in time.

Among the more specific properties, the class $\mathcal{E}^{\mathcal{P}}$ contains the temporal graphs with **periodic edges**, meaning that any appearing edge in the lifetime of the graph, reappears

in a periodic fashion. Additionally, temporal graphs in this class must have a connected footprint.

As a last example, the class $\mathcal{C}^*$ represents the property of having **always-connected snapshots**. So, at any point in time, the snapshot of the temporal graph is connected.

More temporal properties are presented in Chapter 5, where we present mobility models for MANET to induce some of these properties in the corresponding interactions graph.

2.2.2.1 Hierarchy of temporal graph classes

A hierarchy concerning temporal graphs has been presented in [21] by Casteigts *et al.*, in an effort to unify most properties found in the studies on dynamic networks. This hierarchy is mostly focused on connectivity in temporal graphs (see Figure 2.9).

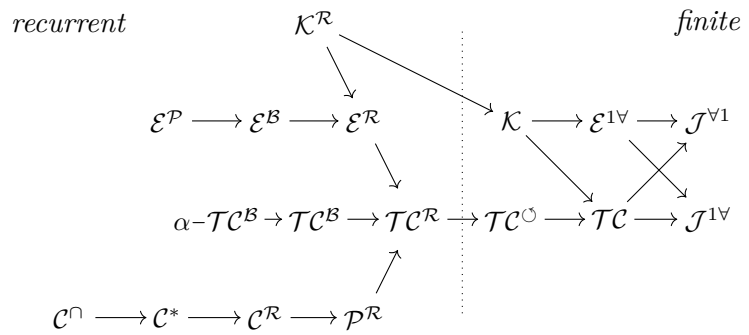


Figure 2.9: Hierarchy surrounding connectivity in temporal graphs (figure from [18]).

Numerous classes in this hierarchy are not presented in this chapter, but are not necessary to understand some relationships between the presented classes:

- Relations between finite and recurrent properties exist. In particular, classes based on recurrent properties are special cases (subsets) of those based on finite properties, for example $\mathcal{TC}^{\mathcal{R}} \subset \mathcal{TC}$.
- On the far right side of the hierarchy, one can find the most general classes of temporal graphs, such as $\mathcal{J}^{1\forall}$. Indeed, all the other classes are included (either directly or indirectly) in $\mathcal{J}^{1\forall}$.
- On the other end, one can find classes such as $\mathcal{E}^{\mathcal{P}}$ and $\mathcal{C}^*$, which are very specific classes, as almost no others are included in them.

2.2.3 Generalizations of temporal graphs

Outside of the scope of this document, but worth mentioning, is the possibility to generalize even further the concept of temporal graphs. Indeed, one can generalize in the same manner as shown for static graphs, such as considering directed edges or arcs, instead of bidirectional edges.

As discussed in [18], investigating the relations between directed analogues of these classes can have a significant impact. For example, with undirected edges, the repetition of journeys from a vertex u to a vertex v eventually creates backward journeys from v to u, which explains why no recurrent version of $\mathcal{J}^{1\forall}$ was defined (since it would amount to $\mathcal{TC}^{\mathcal{R}}$). This type of "reversibility" argument does not apply to directed networks.

Rainbow structures have some relation to temporal structures. For example, a strict journey can be seen as a rainbow path (*i.e.* a path in a graph with colored edges, using distinct colors for each edge) in which the "colors" must obey some given order. Some work on rainbow Hamiltonian paths and cycles in complete graphs has been done in [50, 5]. However, although similar in some aspects, temporal graphs often have multiple labels, whereas edges in a rainbow setting typically only have one color. Also, rainbow structures are often considered in a graph which has a proper edge-coloring (whereas locally distinct labels on temporal graphs are generally not assumed).

2.3 Computational complexity

This section defines basic computational problems, which one might try to solve in graphs or temporal graphs (or other settings). A way to solve such problems is through some algorithm which, when given the problem as an input, computes a solution. An important area of theoretical computer science is interested in, first of all, if such an algorithm exists, and secondly, how much computational resources (computation time and/or space) such an algorithm would need to find a solution to the problem. We will go over the main related concepts being of interest to this document.

2.3.1 Computational problems

As discussed in 2.1 and 2.2, many structures and properties reside in graphs, which can be used to define some graph classes. A natural question one can ask is whether a given graph belongs to some given class or not. The answer to this question is either yes or no.

A question such as this one, for which there is only a yes or no answer, is referred to as a **decision problem**.

The **input** of a decision problem is the given object (*e.g.* a graph) for which the question is asked. The **output** is yes or no. The input is said to be a **positive instance** of the problem if the output is yes, and a **negative instance** if the answer is no.

A decision problem that naturally corresponds to the spanning tree in standard graphs is the following:

Definition 6. *Spanning tree decision problem*

INPUT: *A graph G.*
QUESTION: *Does G contain a spanning tree?*
OUTPUT: *Yes/no.*

Fact 2. *The decision problem of whether a given graph G contains a spanning tree, is equivalent to the decision problem of whether the graph G is connected, since only connected graphs admit spanning trees, and vice-versa, graphs admitting spanning trees are all connected.*

The problem can be modified easily to ask the same question regarding the existence of a temporal spanner in a given temporal graph. Similarly, this problem would be equivalent to the problem of deciding whether or not the input temporal graph is temporally connected, since only temporally connected graphs admit temporal spanners. A slightly different problem, which is not equivalent to the temporal connectivity problem, is the following:

Definition 7. *Temporal spanner decision problem*

INPUT: *A temporal graph $\mathcal{G}$ and an integer k.*
QUESTION: *Does $\mathcal{G}$ contain a temporal spanner of size k?*
OUTPUT: *Yes/no.*

This decision problem has a strong connection to our results presented in Chapter 3.

Other recent theses focusing on adapting classical graph problems in temporal graphs include Hendrik Molter's thesis [71] and Mathilde Vernet's thesis [84].

In **optimization problems**, one would like to obtain a solution corresponding to the optimization (so maximization or minimization) of some parameter. Optimization problems thus do not output *yes* or *no*.

The straightforward optimization problem regarding a spanning tree in standard graphs is of no interest, since a spanning tree is always of size exactly $n - 1$ edges.

In the case of weighted graphs however, a spanning tree can be minimized according to the sum of the weights of its edges. A minimum weight spanning tree is usually referred to as a **minimum spanning tree** or simply **MST**.

Similarly as with the spanning tree, the straightforward Hamiltonian cycle optimization problem makes little sense, but in weighted graphs the problem is known as the **Traveling salesman problem**, or **TSP**.

Definition 8. *Traveling salesman problem*

INPUT: *A weighted graph G.*
OUTPUT: *A Hamiltonian cycle of G of minimum weight.*

A Hamiltonian cycle in a weighted graph is commonly referred to as a **TSP tour**. In Chapter 4, we present and study a new version of this well-known optimization problem.

Since (most) decision and optimization versions of a problem are more or less the same problem, when referring to a problem (such as the temporal spanner problem or the traveling salesman problem) we tend to refer to both decision and optimization versions of the problem, unless specifically stated otherwise.

2.3.2 Algorithms

An algorithm is a finite sequence of well-defined instructions aiming to solve a given problem. These instructions can be the creation of **variables** in which it stores a value in computer memory, some changes of these variables, some basic operations between values such as a sum or multiplication, *etc.*.

For the sake of illustration, consider Algorithm 1, which shows a version of a well-known graph traversal algorithm named **Breadth First Search**, or simply **BFS** which is used extensively in our work in Chapter 4.

This BFS algorithm solves the problem of whether the given graph G is connected. As stated in Fact 2, since this problem is equivalent to the spanning tree decision problem, Algorithm 1 also solves the spanning tree decision problem.

In terms of memory, it uses some variables, namely u, v and Q, the latter being a queue data structure. **Data structures** allow for ease of storing and retrieving data. Data

Algorithm 1 : `Breadth First Search (connectivity version).`

`Input:` A graph $G = (V, E)$
`Question:` Is G connected?
`Output:` Yes/no

```
1: Queue Q
2: Vertex u = random vertex of V
3: mark u
4: Q.enqueue(u)
5: while Q is not empty do
6:     v = Q.dequeue()
7:     for each neighbor w of v do
8:         if w is not marked then
9:             mark w
10:            Q.enqueue(w)
11: if all vertices in V are visited then
12:    return yes
13: return no
```

structures can take on many forms, such as queues, stacks, linked lists, hash tables, priority queues, *etc.*.

Algorithms may use **loops** to repeat lines of code, such as the **while** loop used to repeat instructions 6 to 10, or the **for** loop repeating lines 9 and 10. Conditional instructions, marked by **if** are only executed if the corresponding condition is met, such as line 10 which is only executed if vertex w is not yet visited.

Finally, the algorithm finishes by **returning** some value. In this algorithm, it either returns `yes` or `no`. It is possible for an algorithm to not finish with a return, in which case the algorithm may finish after having executed its last line of instructions, or may run for some indefinite period of time (such as the mobility models proposed in Chapter 5).

2.3.3 Complexity

Some problems are harder to solve than others. Mostly, comparisons are in terms of execution time needed to solve a problem (and sometimes in terms of memory space needed) when the input scales in size.

For simplicity, the **running time** of an algorithm can be interpreted as the number of instructions executed by the algorithm (each having a unitary cost).

In complexity theory, given a problem with an input of size n, one is interested in the running time of algorithms solving the problem. In particular, the term with the highest growth in n is of interest, since considering the scaling of n will eventually make other terms negligible. Multiplicative factors which do not depend on n are often ignored as well, at least when comparing complexities having different growth.

Let's start with some definitions, which we will then explain in more detail, and for which we will give quick examples regarding some presented problems.

Definition 9. *An algorithm's asymptotic time complexity is $O(f(n))$ if there exists some constant c and some n' such that for all $n > n'$, the algorithm runs in time at most $c \cdot f(n)$.*

Definition 10. *An algorithm's asymptotic time complexity is $\Omega(f(n))$ if there exists some constant c and an n-vertex graph (for arbitrarily large n), such that the algorithm runs in time at least $c \cdot f(n)$ on said graph.*

Definition 11. *An algorithm's asymptotic time complexity is $\Theta(f(n))$ if the algorithm's asymptotic time complexity is $O(f(n))$ and $\Omega(f(n))$.*

The O notation extends to the problem that is solved by the analyzed algorithm. This means that if for some problem P of input size n, one obtains an algorithm solving P in time $O(f(n))$, then the problem itself is said to be solvable in time $O(f(n))$. (The same holds for Ω and Θ if the algorithm is optimal.)

As an example, consider again Algorithm 1. Through proper analysis of the algorithm, one may find an upper bound on this algorithm's running time, depending on n, the number of vertices of the given graph. One may find the upper bound $O(n^9)$ for example. Such a bound is said to be **loose**, in the sense that while technically correct, there exists a better, or **tighter** bound. Indeed, regarding Algorithm 1, a tighter bound may be $O(n^2)$ or even $O(n + m)$, where m represents the number of edges in the given graph. A tight Ω bound on Algorithm 1's execution time is $\Omega(n + m)$ as well, which is attained on any connected graph. By definition, this algorithm thus runs in time $\Theta(n + m)$.

As a second example, consider an algorithm solving the Hamiltonian cycle problem, by generating all possible permutations of vertices and checking whether one corresponds to a Hamiltonian cycle. Such an algorithm is commonly referred to as a **brute force** algorithm or a **naive** algorithm, in which one simply tests all possible solutions. Due to the fact that there are $n!$ possible permutations of vertices, the time complexity of this particular algorithm is $\Theta(n!)$, which is prohibitive. However, a better algorithm,

named the Bellman-Held-Karp algorithm, was proposed in [53]. This algorithm solves the Hamiltonian cycle decision problem (as well as TSP) in time $O(2^n n^2)$, through dynamic programming.

In addition to these three standard asymptotic complexity notations, in this document we will use two other notations. The first is used to designate a loose upper bound (as opposed to O designating a potentially tight upper bound).

Definition 12. *An algorithm's asymptotic time complexity is $o(f(n))$ if for every constant c, there exists some n' such that for all $n > n'$, the algorithm runs in time at most $c \cdot f(n)$.*

The second is a big O notation which ignores logarithmic factors as well, on top of ignoring constant factors.

Definition 13. *An algorithm's asymptotic time complexity is $\tilde{\mathrm{O}}(f(n))$ if there exists some constant k such that the algorithm runs in time at most $O(f(n) \cdot \log^k f(n)$.*

The asymptotic notations O, Ω, Θ, o and $\tilde{\mathrm{O}}$ are not solely used for time complexity. These notations adapt to anything which may be bounded asymptotically, such as space complexity or the size of a solution, *e.g.* in Chapter 3 these notations are used to quantify the asymptotic size of a resulting spanner, and in Chapter 4 these are used to denote the length of a TSP tour. Such a quantity is called **polynomial** when it is $O(n^c)$ for some constant c.

2.3.4 Reductions among problems

Generally speaking, theoretical computer scientists often consider polynomial time to be an "acceptable" running time, whereas superpolynomial (anything larger than polynomial) time is not. Corresponding problems are said to be **tractable** or **intractable** respectively (sometimes they are simply said to be easy or hard respectively). Observe that polynomial time is closed under composition, which is a convenient fact to keep in mind when designing polynomial time algorithms.

Polynomial reductions are useful to prove a problem's intractability. Multiple versions of polynomial reductions exist, but the one we present and use in this document is the polynomial many-one reduction.

Definition 14. *A **polynomial many-one reduction** from problem A to problem B, is a polynomial time algorithm that takes as an input any input a for A, and outputs an input b for B. Additionally, a is a positive instance of A, if and only if b is a positive instance of B.*

For the rest of this document, a polynomial many-one reduction may simply be called reduction or polynomial reduction.

To show why such a reduction allows for transferring the intractability of one problem to another, suppose one can reduce a problem A which is known to be intractable to some other problem B. Since problem A is intractable, no polynomial algorithm is known to solve it. Reducing it to B is done in polynomial time. Now, if B is solvable in polynomial time, then the reduction followed by the algorithm used for B constitutes a polynomial algorithm to solve A, which is a contradiction. Problem B must therefore be intractable as well.

Of course, if problem B does turn out to be solvable in polynomial time, then problem A will be as well.

2.3.5 Complexity classes

Distinct problems, and algorithms solving them, may scale in a similar manner when their input scales. Such problems can be classified, and a multitude of distinct **complexity classes** can be defined accordingly.

We focus on the standard complexity classes related to time complexity, most relevant to this document's work (although in 2.4.4 we will mention space complexity classes not presented here).

Definition 15. *The complexity class P (for **polynomial time**) contains all problems which can be solved through a **polynomial time algorithm**.*

Class P includes the spanning tree problem, since we have given Algorithm 1 which solves it in time $O(n^2)$.

The Hamiltonian cycle problem is not known to be in P, since either there exists no polynomial algorithm for it, or it hasn't been found yet (the best known bound being $O(2^n n^2)$).

Definition 16. *The complexity class NP contains all problems which can be **verified** in polynomial time.*

By verified in polynomial time, we mean that any positive instance can be checked by some algorithm in polynomial time, given some **proof** (called **certificate** in this context).

To give an example, the Hamiltonian cycle problem is in NP, since the Hamiltonian cycle itself is a proof that the graph contains one. Indeed, all one has to do is to check if the given Hamiltonian cycle is indeed a cycle, if it is Hamiltonian (so visiting each vertex exactly once), and of course if it is a subgraph of the input graph. All this can be done in polynomial time. NP has another equivalent definition.

Definition 17. *The complexity class NP (for **non-deterministic polynomial time**) contains all problems which can be solved by a polynomial **non-deterministic** algorithm.*

Formally, this definition relies on **non-deterministic Turing machines**, which can be seen as an abstract model of a standard computer. Without going into details, non-determinism essentially allows an algorithm to search for a solution to a problem in multiple directions, or multiple branches, at once. A standard computer only supports **deterministic** algorithms (such as the ones we've seen thus far), meaning that it can only execute one instruction at a time, and thus only test one of such branches at a time.

Definition 18. *The complexity class NP-hard contains all problems for which there exists a polynomial reduction from all problems in NP.*

In other words, if all problems in NP are able to be reduced to some problem A, then A is NP-hard. As presented in 2.3.4, reducing some problem A to another problem B makes it so that problem B is at least as hard as A. Now, if *all* problems A in NP are reducible to some problem B, then B is seen as a relatively "harder" problem than any problem A, since solving it allows for solving any problem A in turn.

Definition 19. *The complexity class NP-complete contains all NP problems which are NP-hard.*

The first problem to have been shown NP-complete is the **satisfiability problem** (usually referred to as **SAT**), which given a boolean formula (boolean variables joined by `and` and `or` operations), asks if some assignment of the variables exists in which the boolean formula resolves to `True`. The problem is in NP since a given solution/proof can be verified in polynomial time. The problem is shown NP-hard by Cook [32] and later by Levin [66] in a different manner, through a non-trivial and groundbreaking theorem, now referred to as the Cook-Levin theorem.

Ever since, SAT has been the mother of all NP-complete problems. Since all one had to do to show some other problem was NP-complete was to find a reduction from SAT to the other problem, which in turn could serve for other reductions, *etc.*

Many problems have since been shown to be NP-complete, including the Hamiltonian cycle problem [59]. Whenever a new problem is studied which is in NP, it is natural to ask if it is NP-complete, and if so, to exhibit a reduction that proves it (which is what we did for our version of TSP in Chapter 4).

One of the reasons why NP-complete problems constitute such an important class, is that solving one of these problems in polynomial time would indirectly solve the famous millennium problem of P versus NP (discussed below).

Many more complexity classes exist outside of the ones presented, such as EXPTIME, PSPACE, *etc.* as well as circuit complexity classes such as AC, NC, *etc.* and parameterized complexity classes such as FPT, W[1], *etc.* All these classes are sometimes jokingly referred to as the complexity zoo, and are detailed on `complexityzoo.net`.

2.3.5.1 Complexity hierarchy

A well-known hierarchy arises from the definitions of P, NP, NP-hard and NP-complete (see Figure 2.10). In this hierarchy, "bubbles" are typically used to represent classes and inclusions/intersections, rather than the inclusion arcs used in Figure 2.3 and Figure 2.9.

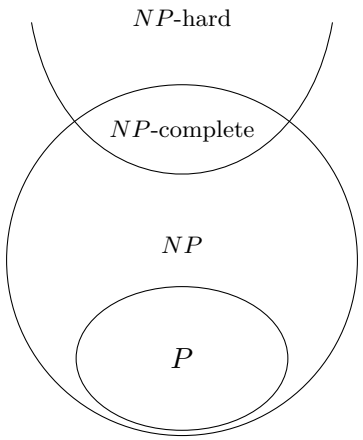


Figure 2.10: The usual depiction of the complexity hierarchy surrounding classes P, NP, NP-hard and NP-complete (assuming $P \neq NP$).

The main information to take away from Figure 2.10 is the following:

- All problems in P are contained in NP, due to the observation that such a problem's positive instances can be "checked" with no proof at all in polynomial time, simply by using the problem's corresponding polynomial algorithm.

- The class NP-hard is depicted as open on one end, because the class is defined through a lower bound, unlike P and NP.

The famous **conjecture** about **P versus NP**, formally introduced by Cook in 1971 [32] and later in 1973 by Levin [66], asks the question whether P equals NP. Although inclusion of P in NP is clear, it is unknown whether the inclusion is strict or not. Indeed, some problems in NP exist for which either no polynomial algorithm exists, or for which we simply haven't found one yet (such as the Hamiltonian cycle problem). The general consensus is that $P \neq NP$, and many results are actually conditional upon this conjecture, although unproven.

The conjecture is one of the famous seven millennium prize problems in mathematics [56]. Proving or disproving any of these conjectures is worth a 1 million US dollars prize. Thus far, only one of the seven conjectures has been solved. Concerning P versus NP, if it does turn out $P = NP$, it could radically change discrete mathematics and computer science, as well as have impacts on cryptography, life sciences, logistics and many more.

2.4 Motion planning and related algorithms

2.4.1 Definitions

From an abstract point of view, **motion planning** decision problems can be stated as follows.

Definition 20. *Motion planning decision problem*

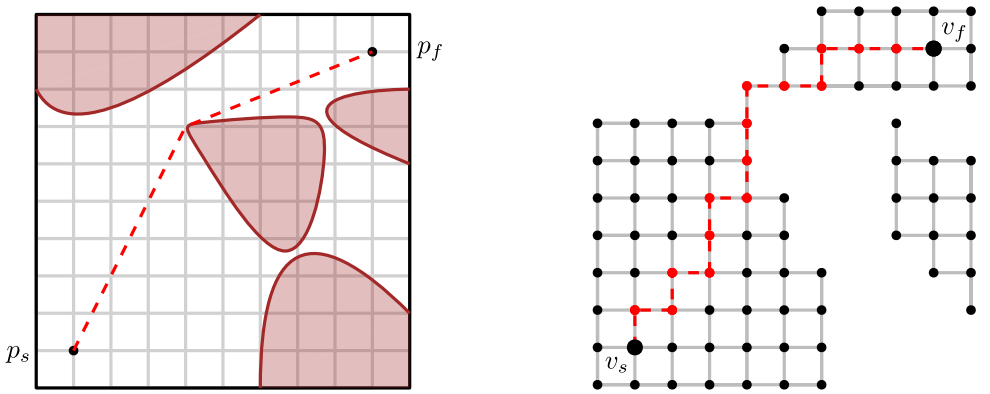
> ***INPUT:*** *a mobile entity m with some mobility constraints, a (potentially infinite) environment E, a starting position $p_s \in E$, a finishing position $p_f \in E$, some obstacles $O \subset E$*
> ***QUESTION:*** *can m reach p_f, starting from p_s?*
> ***OUTPUT:*** *yes/no*

The exact definition depends on how the environment and the obstacles are modeled, and what the constraints are. As a simple example, let us consider a robot r which can move up to a speed of one meter per second in any direction, and which can change directions instantly. The robot's size is often considered negligible. An example of the environment E can be a square plane of side length 10 meters in the real domain $\mathbb{R}^2$, with the starting position for $p_s = (1, 1)$ and finishing position $p_f = (9, 9)$ (see Figure 2.11a).

Now, depending on the obstacles, the output might be no if some obstacles, such as a wall or a river, separates the robot from its finishing position. If no obstacles are present however, the answer is trivially yes.

The optimization version typically aims to find a solution path that optimizes some parameter concerning the mobile entity. The most natural parameters to minimize could be the total distance traveled by the entity, or the total amount of travel time it needs to get to p_f.

In the above example, if no objects are blocking a direct path between $p_s = (1, 1)$ and $p_f = (9, 9)$, the shortest path would be of length (or travel time) $\sqrt{8^2 + 8^2} = 8\sqrt{2} \approx 11.31$. However, as we can see in Figure 2.11a, an obstacle is present on the direct path, resulting in the most direct path costing ≈ 12.09.



(a) Mothion planning problem, with optimal path of cost ≈ 12.09 (seconds or meters) passing via location $(4, 7)$.

(b) Configuration space in a discretized version, with optimal path of cost 16. The graph is disconnected.

Figure 2.11: An example motion planning problem for robot r, with $p_s = (1, 1)$, $p_f = (9, 9)$ and some obstacles, and a corresponding configuration space. Optimal paths are shown in red and dashed.

Closely related to motion planning are **reachability** problems and **pathfinding** problems. All three capture the general idea of a system with a starting state and a goal state, attainable (or not) through some allowed operations on the system. Motion planning represents the problem in the particular setting of a mobile entity. Reachability is the term used mainly in automata, Petri nets, graphs and other abstract settings. The term pathfinding is also used in the setting of graphs and maze-solving. As it turns out though, most motion planning problems can actually be transformed into a graph problem, via the creation of a configuration space.

2.4.2 Configuration space

A **navigation mesh** is the result of a specific transformation from a motion planning instance into a graph. Here, we're interested in a specific navigation mesh, commonly called the configuration space, phase space, state graph, *etc.* A **configuration space** corresponding to some motion planning problem is a (possibly directed and/or weighted) graph constructed by considering all possible **configurations** in which the mobile entity can possibly find itself (possibly infinitely many states). These configurations constitute the configuration space's vertices. Arcs are added between vertices if the mobile entity is capable of going from the source vertex's configuration to the destination vertex's configuration in one time unit. One may add a weight to the arc representing the cost of using this arc **w.r.t.** some parameter to optimize.

An example on how to construct the configuration space for a discrete version of the robot example (see again Figure 2.11) is as follows. Consider that the robot can only change directions every second, in one of only four directions (east, west, north and south), at only one speed, say 1 meter per second. The robot's configuration space's vertices are then the set $V = \{v_{xy} | 0 \leq x, y \leq 10\}$ for any (x, y) which isn't located on an obstacle. One might define $v_s = v_{11}$ as the vertex representing p_s, and $v_f = v_{99}$ representing p_f. Its configuration space's arcs are $\{(v_{xy}, v_{x'y'})\}$ if $|x - x'| + |y - y'| = 1$ and if no object is placed between these configurations (see Figure 2.11b). One may add an implicit weight to these arcs of 1 (meter or second).

To remain realistic, the discretization of a entity's mobility constraints should be chosen carefully. Indeed, some discretizations may alter the output of some motion planning problems. A discretization removing too much of the initial power of the entity's mobility constraints can make it so no path exists between v_s and v_f, even if it a path exists between p_s and p_f. The cost of an optimal path (whether it be distance, travel time, ...) may also be altered, although one may bound the possible error from the optimum result.

As can be observed in Figure 2.11, an optimal path is present for the robot between p_s and p_f, passing by position $(4, 6)$. This path is only barely present in the configuration space. Indeed, if the obstacle close to $(4, 6)$ were only a bit larger, covering $(4, 6)$, then no path would exist between v_s and v_f, even though one would still exist between p_s and p_f. This is the case for the small gaps between obstacles near $(7, 3.5)$, as well as near $(8, 7)$, which are not present in the configuration space (in fact, the graph is disconnected due to this). To avoid such problems, one might have chosen a smaller time unit, such as 0.1 seconds, maintaining paths between close obstacles, at the cost of a larger configuration

space. In terms of the cost of an optimal path, in the original setting it costs ≈ 12.09, while afterwards it costs exactly 16. In fact, through this specific discretization, costs of paths have a worst case multiplicative error of $\sqrt{2}$ (assuming optimal paths were maintained).

We also note here that some very large (or infinite) configuration spaces which may be too large to generate and store at once, may simply be generated on the fly, as they are traversed by some algorithm (see Section 2.4.3).

More techniques exist to transform a given motion planning problem into a more tangible and solvable structure. For example, some other navigation mesh, known as **triangulation** creates a specific type of graph which might have been a better way to treat our example problem with the robot, since the resulting graph typically doesn't distort optimal paths' distances. Also, in triangulation, little to no discretization needs to be applied to the entity's mobility constraints. However, we choose to present the configuration space technique, since the configuration space is the go to navigation mesh concerning work (including ours in Chapter 4) on the Racetrack model (see Section 2.4.4). For more on the broad subject of different techniques and navigation meshes, see Botea *et al.*'s survey [13], covering also real-time search and multi-agent pathfinding. For the specific triangulation navigation mesh, the reader is referred to [36] by Demyen *et al.* (Also, in French, I recommend Cyril Gavoille's lecture notes for "Techniques Algorithmiques et Programmation" [47] which, among other subjects including TSP and A*, covers the subject of navigation meshes.)

2.4.3 Graph algorithms

We presented how a given instance of a motion planning problem can be transformed into a graph. Now, the problem is how to get from vertex v_s in this graph, representing p_s, to vertex v_f, representing p_f. (The term pathfinding is now appropriate since we consider the setting of graphs.)

Multiple algorithms on the topic of graph traversal or graph search can be applied. We'll go over the most well known ones. We start by giving algorithms which are able to solve any pathfinding problem, whether it be a decision, optimization or computational problems. These algorithms are breadth first search, Dijkstra's algorithm and A* (pronounce A star), algorithms which are able to compute optimal paths, also called simply **shortest paths**. Afterwards, we'll also give a quick overview of other algorithms of notice which do not necessarily result in shortest paths.

Breadth first search (or **BFS**) is an algorithm that traverses a graph, prioritizing the visit of adjacent vertices over the visit of distant vertices. Partially presented in 2.3.2, we present a more generic form, which can be adapted to a multitude of problems needing to traverse a given graph (including the spanning tree decision problem).

Algorithm 2 : `Generic Breadth First Search.`

`Input :` A graph $G = (V, E)$ and starting vertex v_s

```
1: Queue Q
2: mark v_s
3: Q.enqueue(v_s)
4: while Q is not empty do
5:     v = Q.dequeue()
6:     for each neighbor w of v do
7:         if w is not marked then
8:             mark w
9:             Q.enqueue(w)
```

BFS starts at some vertex, which in our motion planning problems, will correspond to p_s. The algorithm marks this vertex as visited, and stores all of its neighbors which aren't yet marked visited in a queue data structure. It then dequeues an element from that queue and repeats. It isn't hard to prove that this process will visit the entirety of the given graph G if G is connected. Also, it will visit p_f if and only if it is reachable from p_s. A simple condition can be added between lines 5 and 6 to check if the dequeued vertex represents p_f. If so, then for the decision problem one can return `true`. To obtain the shortest path, one would also need to add a way to retrace the path taken to get to vertex p_f. This can be done by adding a **parent** vertex to traversed vertices. For the weighted versions, there's a potential problem with BFS: it doesn't take into account any potential weights on arcs. This actually doesn't create any problems for graphs with no weights, or for graphs in which all arcs have the same weights. Indeed, BFS always returns a path using the minimum amount of arcs. However, if the graph contains different weights on arcs, the optimal path in terms of minimum sum of weights on arcs may not correspond to the path using the minimum amount of arcs. Our robot example does not use any different weights on arcs, as by construction, any arc has a cost of distance 1 or time 1 for the robot. One can of course easily create a context in which different weights are needed, such as taking into account different terrain, such as as muddy terrain, or uphill and downhill terrain, *etc.*

The BFS algorithm runs in time $O(n + m)$, where n and m represent the number of vertices and edges of the graph respectively.

For completion, we present **Dijkstra's** algorithm, even though our work and results presented in Chapter 4 have no need for it, since our configuration space will be a graph with the same weights on all arcs.
In [37], Edsger W. Dijkstra presented an elegant modification of BFS, distinct enough to be known as Dijkstra's algorithm nowadays, which is able to produce results for graphs with differently weighted arcs (but with positive weights), unlike BFS.

Algorithm 3 : `Generic Dijkstra's algorithm.`

`Input:` A graph $G = (V, E)$ and starting vertex v_s

```
1:  List L
2:  for each vertex u in V do
3:      d[u] = ∞
4:      L.add(u)
5:  d[v_s] = 0
6:  mark v_s as visited
7:  while L is not empty do
8:      v = element in L with minimum d[v]
9:      L.remove(v)
10:     mark v as visited
11:     for each neighbor w of v do
12:         if w is not visited then
13:             d' = d[v] + w(v, w)
14:             if d' < d[w] then
15:                 d[w] = d'
```

As shown in Algorithm 3, some similar concepts from BFS are used, but in a way so as that the cost (or sum of weights) necessary to get to a vertex from the starting vertex, may be updated if a better path has been found with less total cost over the course of the algorithm. It does this through a data structure which allows the algorithm to recover an element in the data structure according to some criteria (line 8). This is done with a list here, but can be implemented with other data structures as well, such as a priority queue (used in Algorithm 4). As with BFS, some small modifications suffice to adapt this algorithm to motion planning problems.

Dijkstra's algorithm runs in time $O((n + m) \log n)$ when using a priority queue, and in time $O(m + n \log n)$ when using a Fibonacci heap.

The **A star** algorithm (**A***) is basically a BFS or a Dijkstra's algorithm, in which one is able to **guide** the search towards p_f. For this, one needs to find a suitable **guiding heuristic**, denoted $g(v)$, which is able to estimate the cost needed to go from vertex v to v_f. This guiding heuristic doesn't need to be exact, although it does need to respect one

important property: not **overestimating** the actual cost between v and v_f. If it does not respect this property, it may not return an optimal path.

For the example in Figure 2.11, a suitable guiding heuristic may be the Euclidean distance or even the Manhattan distance.

Algorithm 4 : `Generic A star.`

`Input:` A graph $G = (V, E)$, starting vertex v_s and finishing vertex v_f

```
1: PriorityQueue Q
2: v_s.cost = 0
3: Q.enqueue(v_s, 0)
4: while Q is not empty do
5:     v = Q.dequeue()
6:     if v is not visited then
7:         mark v as visited
8:         for each neighbor w of v do
9:             if w is not visited then
10:                w.cost = v.cost +w(v, w)
11:                c = w.cost +g(w)
12:                Q.enqueue(w, c)
```

Generally speaking, A* shares Dijkstra's time complexity, although some better bounds may be found depending on the context, guiding heuristic, *etc.*. In practice though, this algorithm is much faster than BFS or Dijkstra's algorithm.

Other algorithms of notice, which do not generally result in an optimal path, are random walk, the greedy algorithm and Depth First Search (DFS). These algorithms may still be useful regarding other settings, such as for geographic routing in sensor networks due to low memory usage or no need for global knowledge. Another usage may simply be in video games, to make enemies' movement naive or unpredictable.

Lastly, worth noting, is that some general techniques exist to make the resulting configuration space paths more realistic and less costly. In our configuration space example with the robot, it can be useful to round sharp turns in a computed path using the B-spline technique for example. If the mobile entity's constraints are discrete by nature however, such as the Racetrack model, one has no need for such techniques.

2.4.4 Racetrack

In a recreative column of the *Scientific American* in 1973 [45], Martin Gardner presented a paper-and-pencil game known as **Racetrack**. Other names for this game include Vector

Racer, Graph Racer, Vector Rally, Vector Race, Vektorrennen (in German) and Le Zip (in French). The game is played in **rounds**, by multiple players, in which they compete to finish first. This can in fact be seen as a motion planning optimization problem under the guise of a multiplayer game. Any Racetrack game is thus a motion planning problem, with fixed mobility constraints. The natural parameter to minimize is the amount of rounds needed for a mobile entity to finish.

The mobility constraints are as follows (see also Figure 2.12). In each step, the mobile entity moves according to a discrete-coordinate **vector**, initially the zero vector $(0, 0)$. This vector is allowed to be modified at each step (before moving accordingly), but must obey the rule that at step $i + 1$ it cannot differ from the vector at step i by more than one unit in each dimension.

These constraints allow for a rather simple but effective way to simulate a mobile entity with velocity, acceleration and inertia forces, such as (originally) a race car, or a drone.

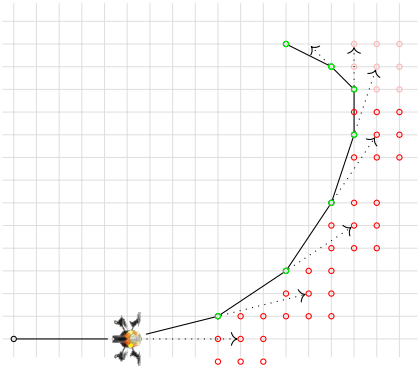


Figure 2.12: Example trajectory of a drone with the Racetrack mobility constraints, trying to turn to the left. Due to a significant initial velocity to the right, it takes a few rounds to effectively turn.

Since these constraints, by definition, allow for only a finite amount of positions in a finite environment, an $L \times H$ environment E can be given as a two dimensional array, representing every discrete position while also precising whether it lays on an obstacle or not. By convention and for ease of computational analysis, $N = \texttt{max}(L, H)$ and E is given as an $N \times N$ array.

The original Racetrack problem can now be formulated as follows.

Definition 21. *original Racetrack problem*

> ***INPUT:*** *A mobile entity m respecting the Racetrack mobility constraints, an $N \times N$ array representing the environment E, a starting position $p_s \in E$, a finishing position $p_f \in E$*
> ***OUTPUT:*** *Minimum amount of rounds necessary for m to reach p_f, starting from p_s.*

Since the term "path" doesn't reflect the concept of acceleration, we define the term **trajectory** to be a solution path in Racetrack instances.

The creation and traversal of a configuration space is a natural solution for a Racetrack problem. The simple mobility constraints have no need to be further discretized, as they already propose a set of simple options every discrete round.

So no optimal paths are lost, and their costs remain unchanged, after transforming a Racetrack instance into a configuration space graph.

A **configuration space** corresponding to a given Racetrack instance is the following directed graph. The configurations are of the form (x, y, dx, dy), where (x, y) represents the entity's position, and (dx, dy) its velocity vector. x and y are trivially bounded by $O(N)$ each, as well as dx and dy. However, observe that after k rounds of only accelerating, a total distance of $1 + 2 + 3 + ... + k = k(k + 1)/2 = O(k^2)$ can be crossed by the Racetrack vehicle in each dimension. Since the environment is bounded by $N \times N$, the vehicle can reach a top speed of at most $O(\sqrt{N})$ in each dimension, so dx and dy can be bounded tighter than $O(N)$, namely by $O(\sqrt{N})$. As a result, the configuration space contains a total of $O(N \times N \times \sqrt{N} \times \sqrt{N}) = O(N^3)$ vertices. Since each configuration has nine possible next configurations, the amount of edges is also bounded by $O(N^3)$. The resulting graph is thus polynomial in size *w.r.t* to the Racetrack instance's input environment.

Any presented graph traversal algorithm (see Section 2.4.3) can then be applied to obtain an optimal trajectory. Since these algorithms run in linear time on the graph, and the graph is polynomial in size, the complete algorithm of creating a configuration space and traversing it runs in polynomial time. The original Racetrack problem thus belongs to complexity class P.

In [55], Holzer and McKenzie present some more precise complexity results, alongside interesting reductions to and from Racetrack. Indeed, inside of complexity class P exist some more complexity classes surrounding the amount of memory space needed, which are classes we did not discuss. These classes can be considered outside of the scope of this thesis. In short, Racetrack's complexity depends on whether obstacles' borders are

accessible or not (for example, in Figure 2.11 we suppose they are). If they are, then Racetrack is equivalent to the grid graph reachability problem, and is thus a problem in L (problems needing a logarithmic amount of memory space). If obstacles' borders are not accessible, then Racetrack is equivalent to directed graph reachability, which is known to be NL-complete (somewhat similar in relation to L, as NP-complete is to P). The authors finish by proving that verifying if a given strategy for Racetrack is winning is P-hard (so among the hardest problems of P), which contradicts the popular "conjecture" stating that all interesting and fun games are NP-hard.

In [12], Bekos *et al.* are interested in specific environments of Racetrack games, such as the Indianapolis track, a simple rectangular track of size $L \times H$ of some width W. The mobile entity has to do a complete lap as fast as possible. In this specific track, the authors were able to show that one can do significantly better than a polynomial algorithm *w.r.t.* the size of the environment $L \times H$, by proposing an algorithm which runs in time $O(W^5)$, so only polynomial in the width of the Indianapolis track, whatever the size $L \times H$. They further generalize their algorithm for generalized tracks of Indianapolis, in which any number of orthogonally placed tracks of width W can be treated. The authors finish by proposing algorithms which only have access to a **limited view** of the racetrack, which essentially allows for faster computing of a solution, but potentially results in non-optimal solutions. This is similar to how we decided to accelerate our proposed algorithm for experimental results in Chapter 4.

Chapter 3

Temporal cliques admit sparse spanners

The worst thing you can do to a problem is solve it completely.

— Daniel Kleitman

Contents

This chapter is based on our work on temporal spanners, which was published in the proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP) 2019 [25]. The full version is currently in minor revision for the Journal of Computer and System Sciences (JCSS). This work is available on ArXiv [26], and a short French version was presented at national conference ALGOTEL [24], where it was awarded the best paper award.

Let $G = (V, E)$ be an undirected graph on n vertices and $\lambda : E \to 2^{\mathbb{N}}$ a mapping that assigns to every edge a non-empty set of positive integer labels. These labels can be seen as discrete times when the edge is present. Such a labeled graph $\mathcal{G} = (G, \lambda)$ is *temporally connected* if a path exists with non-decreasing times from every vertex to every other vertex. In a seminal paper, Kempe, Kleinberg, and Kumar [61] asked whether, given such a temporal graph, a *sparse* subset of edges can always be found whose labels suffice to preserve temporal connectivity—a *temporal spanner*. Axiotis and Fotakis [11] answered negatively by exhibiting a family of $\Theta(n^2)$-dense temporal graphs which admit no temporal spanner of density $o(n^2)$. The natural question is then whether sparse temporal spanners can always be found in at least *some* classes of dense graphs.

In this chapter, we answer this question affirmatively, by showing that if the underlying graph G is complete, then one can always find temporal spanners of density $O(n \log n)$. The best known result for complete graphs so far was that they admit spanners of density $\binom{n}{2} - \lfloor n/4 \rfloor = O(n^2)$. Our result is the first *positive* answer as to the existence of $o(n^2)$ sparse spanners in adversarial instances of temporal graphs since the original question by Kempe *et al.*, focusing here on complete graphs. The proofs are constructive and directly adaptable as an algorithm.

3.1 Introduction

3.1.1 Sparse Temporal Spanners and Related Work

In the last section of [61] (conference version [60]), Kempe, Kleinberg, and Kumar ask

> "Given a temporally connected network $G = (V, E)$ on n nodes, is there a set $E' \subseteq E$ consisting of $O(n)$ edges so that the temporal network on the subgraph (V, E') is also temporally connected? In other words, do all temporal networks have sparse subgraphs preserving this basic connectivity property?"

Here, Kempe *et al.* consider a model where each edge has a single label, thus the edges are identified with their labels, but the discussion is more general. What they are asking, essentially, is whether an analogue of spanning tree exists for temporal networks when the labels are *already fixed*. They answer immediately (and negatively) for the particular case of $O(n)$ density, by showing that hypercubes labeled in a certain way need all of their edges to achieve temporal connectivity, thus some temporal graphs of density $\Theta(n \log n)$ cannot be sparsified. The more general question, asking whether $o(n^2)$-sparse spanners always exist in dense temporal graphs remained open for more than a decade, and was eventually settled, again negatively, by Axiotis and Fotakis [11]. The proof in [11] exhibits an infinite family of temporally connected graphs with $\Theta(n^2)$ edges that do not admit $o(n^2)$-sparse spanners. Their construction can be adapted for strict and non-strict journeys.

On the positive side, Akrida *et al.* [3] show that, if the underlying graph G is a complete graph and every edge is assigned a single globally-unique label, then it is always possible to find a temporal spanner of density $\binom{n}{2} - \lfloor n/4 \rfloor$ edges (leaving however the asymptotic density unchanged). Akrida *et al.* [3] also prove that if the label of every edge in G is chosen uniformly at random (from an appropriate interval), then almost surely the graph admits a temporal spanner with $O(n \log n)$ edges. Both [11] and [3] include further results related to the (in-)approximability of finding a *minimum* temporal spanner, which is out of the scope of this chapter and document.

By its nature, the problem of finding a temporal spanner in a temporal network seems to be significantly different from its classical (*i.e.*, non-temporal) version, whether this version considers a static graph (see *e.g.* [30, 72, 78]) or the current network topology of an updated dynamic graph (see *e.g.* [9, 40, 49]). The essential difference is that spanning trees always exist in standard (connected) graphs, thus one typically focuses on the tradeoff between the density of a solution and a quality parameter like the *stretch factor*, rather than to the very existence of a sparse spanner.

3.1.2 Contributions

In this chapter, we establish that temporal graphs built on top of complete graphs *unconditionally* admit $\Theta(n \log n)$-sparse temporal spanners when *non-strict* journeys are allowed. Furthermore, such a spanner can be computed in polynomial time. The case of strict journeys requires more discussion. Kempe *et al.* observed in [61] that if every edge of a complete graph is given the same label, then this graph is temporally connected, but no multi-hop *strict* journey exist, thus none of the edges can be removed, and the problem is trivially a no-instance. To make the problem interesting when only strict journeys are allowed, one should constrain λ to avoid such a pathological situation. In the present case, we require that a sub-labeling of one label per edge exists in which any two adjacent edges have different labels. This formulation slightly generalizes the single-label global-unicity assumption made in [3] (although essentially equivalent regarding temporal connectivity) and eliminates the distinction between strict and non-strict journeys. Under this restriction, we establish that all temporal graphs whose underlying graph is complete admit a $O(n \log n)$-sparse temporal spanner. Moreover, if the restricted labeling is given, then the spanner can be computed in polynomial time. (The problem of deciding whether a general labeling admits such a sub-labeling is not discussed here; it might be computationally hard.)

Our proofs are constructive. We start by observing that the above two settings one-way reduce to the setting where every edge has a single label and two adjacent edges have different labels. The reduction is "one-way" in the sense that the transformed instance may have less feasible journeys than the original instance, but all of these journeys correspond to valid journeys in the original instance, so a temporal spanner computed in the transformed instance is valid in the original instance. As a result, the main algorithm takes as input a complete graph $\mathcal{G}$ with single, locally distinct labels, and computes a $O(n \log n)$-sparse temporal spanner of $\mathcal{G}$ in polynomial time.

In summary, we give the first positive answer to the question of whether sparse temporal spanners always exist in a class of dense graphs, focusing here on the case of complete graphs. This answer complements the negative answer by Axiotis and Fotakis [11] and motivates more investigation to understand where the transition occurs between their negative result (no sparse spanners exist in some dense temporal graphs) and our positive result (they essentially always exist in complete graphs). Our algorithms are based on a number of original techniques, which we think may be of more general interest for problems related to temporal connectivity.

The chapter is organized as follows. In Section 3.2, we define the model and notations, and describe the one-way reductions that allows us to concentrate subsequently on single and distinct labels. We also mention two basic techniques of interest for this problem, although they are not used subsequently in the presented results, namely the *sub-clique* technique (from [3]) and the *pivoting* technique (introduced here). In Section 3.3 we introduce the main concepts and techniques used in our algorithms, namely *delegation*, *dismountability*, and *k-hop dismountability*, whose purpose is to recursively self-reduce the problem to smaller graph instances. These techniques are subsequently combined into a more sophisticated algorithm that successfully computes a temporal spanner of $O(n \log n)$ edges. The first step, presented in Section 3.4, is called *fireworks* and results in a spanner of density (essentially) $\binom{n}{2}/2$. Then, in Section 3.5, we exploit a particular dichotomy in the structure of the residual instance, which allows us to sparsify the graph down to $O(n \log n)$ edges. In Section 3.6, we review the main components of the algorithm, showing that its running time complexity is polynomial. In Chapter 6, we present a few open questions and conclude with some remarks.

3.2 Definitions and basic results

3.2.1 Model and definitions

Let $G = (V, E)$ be an undirected graph and $\lambda : E \to 2^{\mathbb{N}}$ a mapping that assigns to every edge of E a non-empty set of integer labels. These labels can be seen as discrete times when the edge is present. In this chapter, we refer to the resulting graph $\mathcal{G} = (G, \lambda)$ as a *temporal graph* (other models and terminologies exist, many of them being equivalent for the considered problem). If λ is single-valued and locally injective (*i.e.*, adjacent edges have different labels), then we say that λ is *simple*, and by extension, a temporal graph is simple if its labeling is simple.

A temporal path in $\mathcal{G}$ (also called *journey*), is a finite sequence of k triplets $\mathcal{J} = \{(u_i, u_{i+1}, t_i)\}$ such that $(u_1, \ldots, u_{k+1})$ is a path in G and for all $1 \leq i < k$, $\{u_i, u_{i+1}\} \in E$, $t_i \in \lambda(\{u_i, u_{i+1}\})$ and $t_{i+1} \geq t_i$. Strict temporal path (strict journeys) are defined analogously by requiring that $t_{i+1} > t_i$. We say that a vertex u can *reach* a vertex v iff a journey exists from u to v (strict or non-strict, depending on the context). If every vertex can reach every other vertex, then $\mathcal{G}$ is *temporally connected*. Finally, observe that the distinction between strict and non-strict journeys does not exist in simple temporal graphs, as all the journeys are strict.

In general, one can define a *temporal spanner* of $\mathcal{G} = ((V, E), \lambda)$ as a temporal graph $\mathcal{G}' = ((V', E'), \lambda')$ such that $V = V'$, $E' \subseteq E$ and $\lambda' : E' \to 2^{\mathbb{N}}$ with $\lambda'(e) \subseteq \lambda(e)$ for all $e \in E'$. We call $\mathcal{G}'$ a *valid* spanner if $\mathcal{G}'$ is temporally connected. Observe that, if $\mathcal{G}$ is simple, then spanners are fully determined by the chosen subset of edges $E' \subseteq E$ (as in the above citation from [61]). Thus, in such cases, we say that E' itself *is* the spanner. Many of these notions are analogous to the ones considered in [3, 11, 61], although they are not referred to as "spanners" in these works.

Finally, when the underlying graph G is a complete graph, we call $\mathcal{G}$ a temporal clique. An example of a (valid) temporal spanner of a *simple temporal clique* is shown in Figure 3.1.

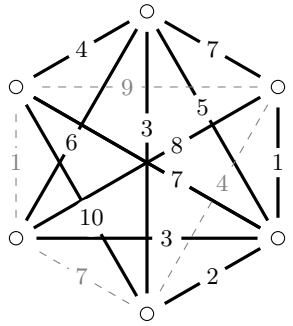


Figure 3.1: Example of a simple temporal clique and one of its temporal spanners (edges in bold). This spanner is not minimum (nor even minimal) and the reader may try to remove further edges.

3.2.2 Generality of simple labelings

We claimed in Section 3.1.2 that if non-strict journeys are allowed, then one can transform a temporal clique $\mathcal{G} = (G, \lambda)$ with *unrestricted* labeling λ into a clique $\mathcal{H} = (G, \lambda_H)$ with *simple* labeling such that any valid temporal spanner of $\mathcal{H}$ induces a valid temporal spanner of $\mathcal{G}$. (As explained, the converse is false, but this is not a problem.) The reduction proceeds in two steps: (1) For every edge e, restrict $\lambda(e)$ to a single label chosen arbitrarily; and (2) Whenever k edges adjacent to a same vertex have the same label l, these edges are relabeled with a unique value in the interval $[l, l + k - 1]$ (arbitrarily) and the value of all the other labels in the graph which are larger than l are shifted by k. It is not difficult to see that if a journey exists in $\mathcal{H}$, then the same sequence of edges allows for a (possibly non-strict) journey in $\mathcal{G}$. A small example of such a process is given below.

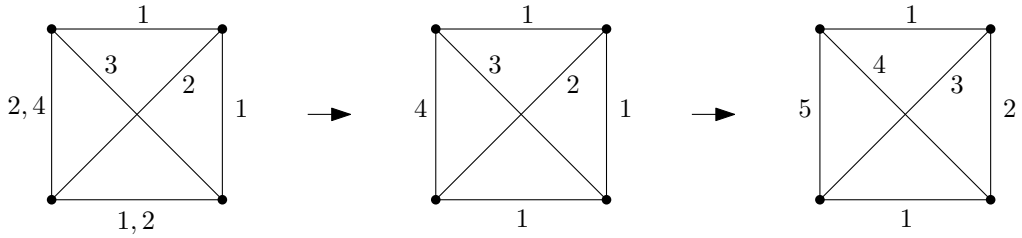


Figure 3.2: Example transformation of a temporal clique (with non-strict journeys) into a temporal clique with simple labelling. First, take one label per edge, then, determine an order between locally equal labels (top right vertex with two edges having label 1) and shift the larger labels.

As for the second claim of the introduction, if strict journeys are the only ones allowed, then as explained, we require the existence of a *simple* sub-labeling of λ (whose computation is not discussed). Here, it is even more direct that any journey based on the sub-labeling is a fortiori available in the complete instance. Based on these arguments, the rest of the chapter focuses on simple temporal cliques, and we sometimes drop the adjective "simple" when it is clear from the context. Consequently, the input is a temporal clique $\mathcal{G} = (G, \lambda)$, where $G = K_n$ is the complete graph on n vertices, and λ a simple labelling of the edges, which may be represented as a permutation π of $(1, 2, 3, ..., \binom{n}{2})$.

3.2.3 Basic techniques

We present here two basic sparsification techniques. The first is from previous works, the second is original. These techniques are not used subsequently, thus they are presented here rather than in Section 3.3. However, they are relevant to the problem in general, and may carry some insights.

3.2.3.1 The subclique technique

So far, the only existing approach for sparsifying simple temporal cliques is that of Akrida *et al.* [3], who prove that one can always remove $\lfloor n/4 \rfloor$ edges without breaking temporal connectivity. Their approach is as follows. First, it is established that if $n = 4$, then it is always possible to remove at least *one* edge. Then, as $n \to \infty$, one can arbitrarily partition the input clique into (essentially) $n/4$ subcliques of 4 vertices each, and remove an edge from each subclique. The edges *between* subcliques are kept, thus the impact of each removal is limited to the corresponding subclique and the resulting graph is temporally

connected. Before moving to other techniques, let us observe that this technique can be greatly improved as follows.

Observation 1 (Improving the subclique technique)**.** The type of partitioning used in [3] is *node-disjoint*, but a key observation is that the same argument actually holds when applied to *edge-disjoint* subcliques, with significant consequences. Indeed, by Wilson's Theorem [87], the number of edge-disjoint cliques on 4 vertices in a complete graph on n vertices is $\lfloor n^2/12 \rfloor$, possibly with a few vertices remaining. (More generally, as $c \to 1$, graphs with minimum degree at least cn have $\lfloor \binom{n}{2}/\binom{k}{2} \rfloor$ disjoint copies of K_k for all k.) The immediate consequence is that one can remove $\lfloor n^2/12 \rfloor = \Theta(n^2)$ edges by the same technique as in [3].

Although this technique allows us to remove $\Theta(n^2)$ edges, it seems unlikely that purely *structural* techniques like this one will lead to spanners of $o(n^2)$ edges. The techniques we develop in this chapter are different in essence and consider the interplay of timestamps at a finer scale.

3.2.3.2 Pivotability

Another natural approach that one might think of is inspired by Kosaraju's principle for testing strong connectivity in a directed graph (see [2]). This principle relies on finding a vertex that all the other vertices can reach (through directed paths) and that can reach all these vertices in return. This condition is sufficient in standard graphs because paths are transitive. In the temporal setting, transitivity does not hold, but we can define a temporal analogue as follows. A *pivot vertex* p is a vertex such that all other vertices can reach p by some time t (through journeys) and p can reach all other vertices *after* time t. The union of the tree of (incoming) journeys towards p and the tree of (outgoing) journeys from p forms a temporal spanner with at most $2(n-1)$ edges. Such a spanner is illustrated on Figure 3.3. Unfortunately, arbitrarily large non-pivotable cliques may exist (see Section 3.3.2.1 for an example of infinite family). However, experiments suggest that they may exist asymptotically almost surely in random temporal cliques (where an instance corresponds to a random permutations of the time interval $[\binom{n}{2}]$).

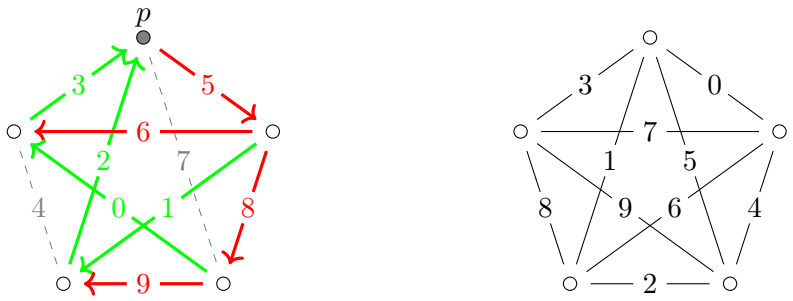


Figure 3.3: Examples of pivotable graph (left) and non-pivotable graph (right). The (light) green edges in the pivotable graph belong to the tree of incoming journeys to pivot vertex p (with $t = 4$); the (darker) red edges belong to the tree of outgoing journeys; the dashed edges belong to neither.

3.3 Delegation and Dismountability

This section introduces a number of basic techniques which are subsequently used (and extended) in Sections 3.4 and 3.5. Given a vertex v, write $e^-(v)$ for the edge with smallest label incident with v, and $e^+(v)$ analogously for the largest label.

Lemma 1. *Given a temporal clique $\mathcal{G}$, if $\{u, v\} = e^-(v)$, then u can reach all vertices through v. Symmetrically, if $\{u, w\} = e^+(w)$, then all vertices can reach u through w.*

Proof. If $\{u, v\} = e^-(v)$, then v has a direct edge with every other vertex after that time, thus a journey exist from u to every vertex through v. (A symmetrical argument applies in the second case.) □

Observe that Lemma 1 holds only when the underlying graph is complete. This property makes it possible for a vertex u to *delegate* its emissions to a vertex v, *i.e.*, exploit the fact that v can still reach all the other vertices *after* interacting with u, thus none of u's other edges are required for reaching the other vertices. By a symmetrical argument, u can delegate its receptions to a vertex w if w can be reached by all the other vertices *before* interacting with u, so the other edges of u are not needed for being reached by other vertices.

The delegation concept suggests an interesting technique to construct temporal spanners. We say that a vertex u in a temporal clique $\mathcal{G}$ is *dismountable* if there exist two other vertices v and w such that $\{u, v\} = e^-(v)$ and $\{u, w\} = e^+(w)$, i.e., u can delegate both its emissions and its receptions. The existence of such a vertex enables a self-reduction of the spanner construction as follows: select $e^-(v)$ and $e^+(w)$ for future

inclusion in the spanner, then recurse on the smaller clique $\mathcal{G}[V \setminus u]$, as illustrated on Figure 3.5. More precisely:

Theorem 1 (Dismountability)**.** *Let $\mathcal{G}$ be a temporal clique, and let u, v, w be three vertices in $\mathcal{G}$ such that $\{u, v\} = e^-(v)$ and $\{u, w\} = e^+(w)$. Let S' be a temporal spanner of $\mathcal{G}[V \setminus u]$. Then $S = S' \cup \{\{u, v\}, \{u, w\}\}$ is a temporal spanner of $\mathcal{G}$.*

Proof. Since $\{u, v\} = e^-(v)$, all edges incident with v in S' have a larger label than $\{u, v\}$, thus u can reach all the vertices through v using only the edges of S'. A symmetrical argument implies that all vertices in $\mathcal{G}$ can reach u through w using only $\{u, w\}$ and the edges of S'. □

We call a graph *dismountable* if it contains a dismountable vertex. It is said to be *fully dismountable* if one can find an ordering of V that allows for a recursive dismounting of the graph until the residual instance has two vertices. An example of fully dismountable graph is given in Figure 3.4.

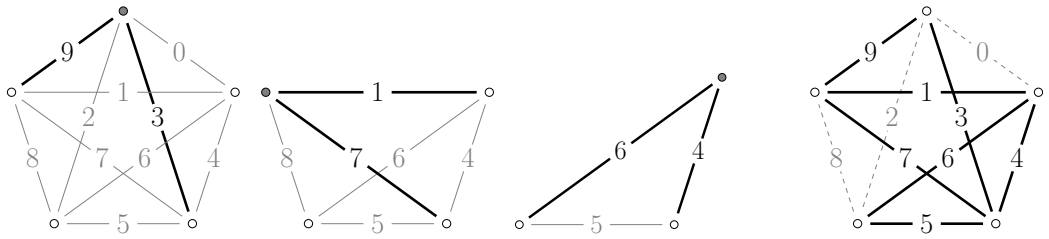


Figure 3.4: Example of a fully dismountable graph and the resulting spanner.

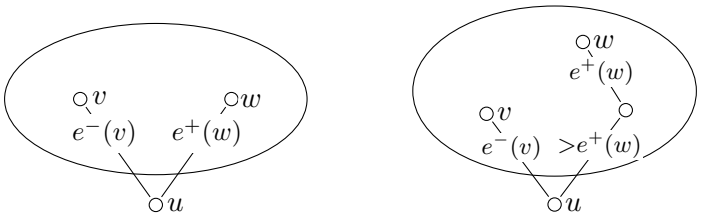
Fact 3 (Spanners based on dismountability)**.** *If a graph can be fully dismounted, then the resulting spanner will have $2(n-2)+1 = 2n-3$ edges.*

Unfortunately, one can design arbitrarily large temporal cliques which are not fully dismountable (see Subsection 3.3.2).

3.3.1 k-hop delegation and k-hop dismountability

The dismountability technique can be generalized to multi-hop journeys. Let J be a journey from vertex u to vertex v through vertices $u = u_0, u_1, \ldots, u_k = v$ with $\{u_{k-1}, u_k\} = e^-(v)$. The key observation is that u can delegate its emissions to v even though $\{u_{i-1}, u_i\} \neq e^-(u_i)$ for some i. Indeed, it is sufficient that the *last* edge of a

journey from u to v is the minimum (at v) in order to delegate u's emissions to v. Symmetrically, it is sufficient that the *first* edge of a journey from w to u is $e^+(w)$ in order to delegate u's receptions to w. Accordingly, a vertex u is called *k-hop dismountable* if one can find two other vertices v and w (possibly identical if $k > 1$) such that there are journeys of *at most* k hops (1) from u to v that arrives at v through $e^-(v)$, and (2) from w to u that leaves w through $e^+(w)$. See Figure 3.5 for an illustration.



(a) Dismountability principle. (b) Example of 2-hop dismountability.

Figure 3.5: Illustration of the principle of dismountability and k-hop dismountability.

Temporal spanners can be obtained in a similar way to 1-hop dismountability by selecting all of the edges involved in these journeys for inclusion in the spanner. However, only the edges adjacent to the dismounted vertex are removed in the recursion, thus some edges used in a multi-hop journey may be selected several times over the recursion (at our advantage). We can then state an analogue fact for k-hop dismountability as follows.

Fact 4. *If a temporal graph $\mathcal{G}$ is fully k-hop dismountable, then this process yields a temporal spanner with at most $2k(n-2)+1 \simeq 2kn$ edges.*

Unfortunately, again, there exist arbitrarily large graphs which are not k-hop dismountable for any k (the same counter-example as for 1-hop dismountability applies, see Section 3.3.2.2). Nonetheless, k-hop dismountability is a core component in the more sophisticated techniques presented next.

3.3.2 Adversarial Families

3.3.2.1 Non-pivotable Graphs

We explain how to construct non-pivotable graphs of arbitrary sizes. The construction ensures that there is a time t before which no vertex can be reached by all vertices, and after which no vertex can reach all vertices. The choice of t does not matter, as moving it forward or backward could only worsen one of the direction. Thus, the simple existence of such a t rules out the existence of a pivot vertex. The construction is first presented

with respect to the 6-vertex clique of Figure 3.6, then we explain how to generalize it. Thus, let $n = 6$. In this case, let $t = 7$ and let us consider the two periods $[0, 7]$ and $[8, 14]$.

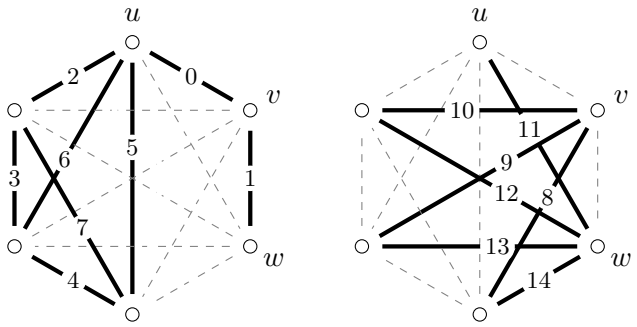


Figure 3.6: Example of a non-pivotable clique seen as the union of two specific subgraphs which represent the periods $[0, 7]$ (left) and $[8, 14]$ (right).

Looking at the graph, observe that none of the vertices can be reached by all the others during the first period. This is true because (1) the only vertex that w can reach is v, making v the only candidate, and (2) none of the other vertices (except u) can reach v. Similarly, none of the vertices can reach all the others in the second period. This is true because (1) u can only be reached by w, making w the only candidate, and (2) w cannot reach v in the second period.

The construction can be generalized to any larger value of n by choosing three vertices to play the same role as u, v, and w. The graph of the first period corresponds to a subclique on $n - 2$ vertices (including u, but not v and w), plus the two edges $\{u, v\}$ and $\{v, w\}$. Thus $t = \binom{n-2}{2} + 1$. The labeling assigns 0 to $\{u, v\}$, 1 to $\{v, w\}$, and all values in $[3, t]$ to the edges of the subclique (arbitrarily). By the same argument as above, none of the vertices can be reached by all others during the first period. As for the second period, the labeling must ensure that all the edges of w have a larger label than all the edges of v, and that $\{u, w\}$ is assigned the smallest label among the edges incident to w, which results in direct applicability of the same argument as above. Thus, none of the vertices can reach all the others during the second period.

Remark 2. *One may be tempted to make both periods overlap by one unit, so that the edge labeled t can be used for both directions. However, this would have no decisive effect on the construction.*

3.3.2.2 Non-dismountable Graphs

We explain how to construct arbitrarily large simple temporal cliques which are not k-hop dismountable for all k (non-dismountable, for short). To start, consider a 4-vertex clique in which the local relations among labels is the same as in Figure 3.7. Such a clique is non-dismountable, because none of the vertices can (1) reach a vertex u through $e^-(u)$ and (2) be reached from a vertex v through $e^+(v)$. An arbitrary large clique can be built

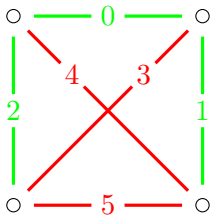


Figure 3.7: Example of a non-dismountable clique on 4 vertices.

by making copies of this clique and assigning labels so that the local minima and maxima in each copy behave as in the original clique. An example with 8 vertices (thus 28 edges) is shown in Figure 3.8. All the other edges are assigned an intermediate label, *i.e.*, whose value is between the local minima and maxima, which prevents journeys between different cliques that arrive through a minimum edge (or leave through a maximum edge). This construction can be generalized to any number of vertices which is a multiple of 4.

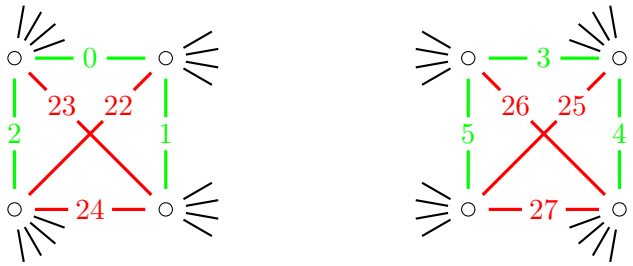


Figure 3.8: Example of a non-dismountable clique on 8 vertices.

Remark 3. *The given construction for non-dismountable graphs makes it possible to find a pivot vertex. However, experiments that we conducted suggest that there exist instances of arbitrary sizes which are neither pivotable nor dismountable.*

3.4 The Fireworks Technique

In this section, we present an algorithm called *fireworks*, which exploits a system of multi-hop delegations among vertices. In particular, we take advantage of *one-sided delegations*, in which a vertex delegates only its emissions, or only its receptions. The combination of many such delegations is shown to lead to the removal of essentially half of the edges of the input clique. The residual instance has a particular structure that is exploited in Section 3.5 to obtain the final $O(n \log n)$-sparse spanners.

3.4.1 Forward Fireworks

The purpose of fireworks is to mutualize several one-sided delegations in a transitive way, so that many vertices do not need to reach the other vertices directly. Given a temporal clique $\mathcal{G} = (G, \lambda)$ with $G = (V, E)$, define the directed graph $G^- = (V, E^-)$ such that $(u, v) \in E^-$ iff $\{u, v\} = e^-(v)$, except that, if $e^-(u) = e^-(v)$ for some u and v, only one of the arcs is included (chosen arbitrarily).

Lemma 2. *Directed paths in G^- correspond to journeys in $\mathcal{G}$.*

Proof (by contradiction).. Let $(u_0, u_1), (u_1, u_2), ..., (u_{k-1}, u_k)$ be a directed path in G^- and suppose that the corresponding path in $\mathcal{G}$ is *not* a journey. Then it must be the case that the label of an edge (u_{i-1}, u_i) is greater than the label on the adjacent edge (u_i, u_{i+1}) for some i. Then $\{u_{i-1}, u_i\} \neq e^-(u_i)$ which is impossible. □

By construction, E^- induces a disjoint set of *out-trees* (one source, possibly several sinks). We transform E^- into a disjoint set $\mathcal{T}^- = (V, E_T^-)$ of *in-trees* (one sink, possibly several sources) as follows, see also Figure 3.9 for an illustration. Let E_T^- be initialized as a copy of E^-. For every v with outdegree at least 2 in E^-, let $(v, u_1), ..., (v, u_\ell)$ be its out-arcs with (v, u_ℓ) being the one with the *largest* label. For every $i < \ell$, if u_i is a sink vertex, then flip the direction of (v, u_i) in E_T^- (*i.e.*, replace (v, u_i) by (u_i, v) in E_T^-); otherwise remove (v, u_i) from E_T^-. Let $\mathcal{T}^- = (V, E_T^-)$ be the resulting set of in-trees $\mathcal{T}_1^-, ..., \mathcal{T}_k^-$ (containing possibly more in-trees than the number of initial out-trees).

Fact 5. *The set of in-trees $\mathcal{T}^- = (V, E_T^-)$ has the following properties:*

1. *Directed paths in $\mathcal{T}^-$ correspond to journeys in $\mathcal{G}$.*
2. *Every vertex belongs to exactly one tree.*

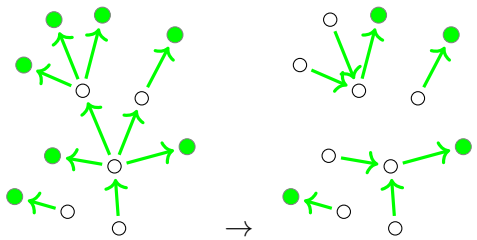


Figure 3.9: Example of transformation from a disjoint set of out-trees (V, E^-) to a disjoint set of in-trees (V, E_T^-). The colored vertices represent sink vertices.

3. *Every tree contains at least two vertices.*

4. *There is a unique sink in each tree.*

5. *The unique arc incident with a sink s corresponds to $e^-(s)$.*

Fact 5.1 follows from Lemma 2 because an arc (v, u_i) is only replaced by (u_i, v) if the label of (v, u_i) is less than the label of another arc (v, u_ℓ), so $(u_i, v), (v, u_\ell)$ is a journey in $\mathcal{G}$. Observe that some of the journeys induced by the arcs of $\mathcal{T}^-$ may include intermediate hops where the arc's label is not locally minimum for its head endpoint. However, as already discussed in Section 3.3, a delegation only requires that the label of the last hop of a journey be locally minimum, and that is the case here (Fact 5.5).

The motivation behind this construction is that all the vertices in each in-tree are able to delegate their emissions to the corresponding sink vertex. For this reason, the sink vertex will be called an *emitter* in the rest of this work. An important consequence of our construction is that the number of emitters in $\mathcal{T}^-$ cannot exceed half of the total number of vertices.

Lemma 3. *The number of emitters in $\mathcal{T}^-$ is at most $n/2$*

Proof. After the transformation from E^- to E_T^-, there is only one emitter in each in-tree $\mathcal{T}_i^- \in \mathcal{T}^-$ (Fact 5.4), and there are at most $n/2$ trees because each one contains at least 2 vertices (Fact 5.3). □

We are now ready to define a temporal spanner based on $\mathcal{T}^-$, which consists of the union of all edges involved in an in-tree and all edges incident with at least one emitter. More formally, let $S_T^- = \{\{u, v\} \in E : (u, v) \in \mathcal{T}^-\} \cup \{\{u, v\} \in E : u \text{ is an emitter}\}$.

Theorem 2. *S_T^- is a temporal spanner of $\mathcal{G}$.*

Proof. By Fact 5, every vertex v of $\mathcal{G}$ that is a non-emitter in $\mathcal{T}^-$ can reach an emitter s through an edge $e^-(s)$. Furthermore, the inclusion of all edges incident to a vertex s that is an emitter in $\mathcal{T}^-$ ensures that v can still reach all other vertices afterwards and so can s. Therefore, every vertex can reach all other vertices by using only edges from S_T^-. □

We call this type of spanner a *forward fireworks cover*. An example is given in Figure 3.10, the corresponding journeys being depicted on the left side.

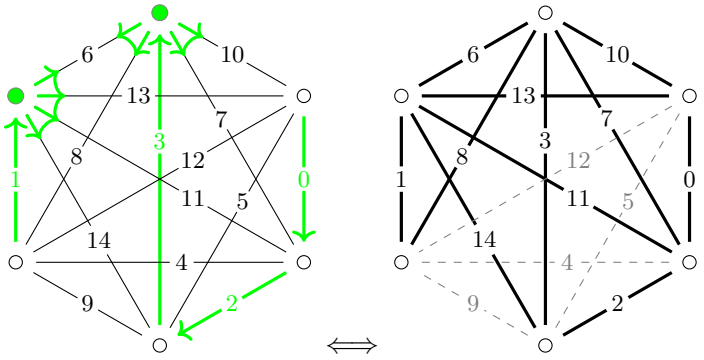


Figure 3.10: Example of forward fireworks cover and the resulting spanner.

Theorem 3. *Forward fireworks covers have at most* $3/4\binom{n}{2} + O(n)$ *edges.*

Proof. Let S_T^- be a forward fireworks cover based on a set of in-trees $\mathcal{T}^-$. Each non-emitter in $\mathcal{T}^-$ has only one out-arc which becomes one edge in S_T^-, thus overall $\mathcal{T}^-$ contributes less than n edges to S_T^-. Now, every emitter has an edge to every other vertex in S_T^-, and there are at most $n/2$ emitters in $\mathcal{T}^-$ by Lemma 3. Note that the edges between emitters are selected twice but should be counted only once. Thus in the end, there are at most $n(n/2) - n^2/8$ edges, plus the edges of the in-trees (that is $O(n)$ edges). □

Before moving to Section 3.4.2, we establish a small technical lemma that will be used in Section 3.5, but is worth being stated here as it pertains to the structure of the in-trees.

Lemma 4. *Every non-emitter vertex v can reach a vertex v' in the same in-tree $\mathcal{T}_i^-$ (emitter or not) using a journey of length* at most two *that arrives at v' through $e^-(v')$.*

Proof. Before the transformation from E^- to E_T^-, all arcs $(u, v) \in E^-$ are such that $\{u, v\} = e^-(v)$, thus all the journeys in E^- have the required property. Then, some arcs

are flipped by the transformation (replaced by oppositely directed arcs). Now, if an arc (v, u) is flipped, then its head u is a sink in E^- and its tail v is not, thus no arc of the form (w, v) is flipped. As a result, at least one arc in any two consecutive arcs in a journey in $\mathcal{T}_i^-$ is the minimum edge of its head. □

3.4.2 Backward Fireworks

A symmetrical concept of fireworks can be defined based on the edges $\{u, v\} = e^+(v)$ of a temporal clique $\mathcal{G} = (G, \lambda)$. All arguments developed in the context of forward fireworks can be adapted in a symmetrical way, so we will omit most of the details. First, we build a directed graph $G^+ = (V, E^+)$ which is a disjoint set of *in-trees*. By an analogous transformation as above, this set is then converted into a disjoint set $\mathcal{T}^+ = (V, E_T^+)$ of *out-trees* each of which contains only one source which we call a *collector*. The collector s of an out-tree can reach all of the other vertices in this tree by journeys that leave s through its edge $e^+(s)$, thereby guaranteeing that every other vertex that reaches s can subsequently reach all other vertices in the tree. The following lemma is obtained by symmetrical arguments.

Lemma 5. *The number of collectors in $\mathcal{T}^+$ is at most $n/2$*

Finally, we can build a temporal spanner $S_T^+ = \{\{u, v\} : (u, v) \in \mathcal{T}^+\} \cup \{\{u, v\} : u$ is a collector$\}$ which we call a *backward fireworks cover*, and prove the following results by symmetrical arguments to the ones in Section 3.4.1.

Theorem 4. *S_T^+ is a temporal spanner of the temporal clique $\mathcal{G}$.*

Theorem 5. *Backward fireworks covers have at most $3/4\binom{n}{2} + O(n)$ edges.*

An example of a backward fireworks cover is given on Figure 3.11. Finally, we establish a symmetrical property as in Lemma 4, to be used also in Section 3.5.

Lemma 6. *For every non-collector vertex v in an out-tree $\mathcal{T}_i^+ \in \mathcal{T}^+$, there exists a vertex v' in $\mathcal{T}_i^+$ (collector or not), such that a journey of length* at most two *from v' to v exists, leaving v' through $e^+(v')$.*

3.4.3 Bidirectional Fireworks

A forward fireworks cover makes it possible to identify a subset of vertices, the *emitters*, such that every vertex can reach at least one emitter u through $e^-(u)$ and u can reach

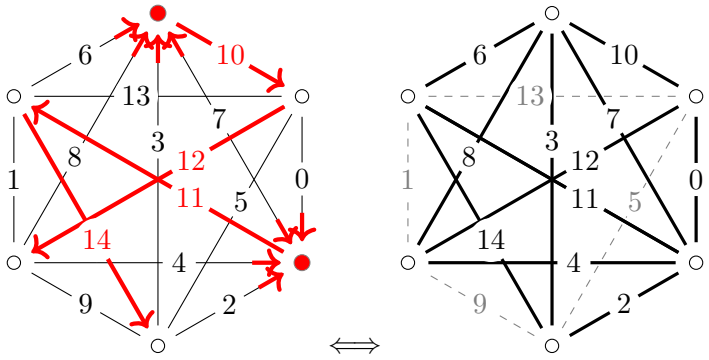


Figure 3.11: Example of a backward fireworks cover and the resulting spanner (p:65).

every other vertex afterwards *through a single edge*. Similarly, a backward fireworks cover makes it possible to identify a subset of vertices, the *collectors*, such that every vertex can be reached by at least one collector v through $e^+(v)$ and v can be reached by every other vertex before this *through a single edge*. Combining both ideas, we can define a sparser spanner in which we do not need to include *all* of the edges incident with emitters and collectors, but only the edges *between* emitters and collectors (plus, of course, the edges used for reaching an emitter and for being reached by a collector).

Precisely, let $\mathcal{T}^-$ be the disjoint set of in-trees obtained during the construction of a forward fireworks cover (see Figure 3.9), and let $\mathcal{T}^+$ be the disjoint set of out-trees obtained during the construction of a backward fireworks cover. Let X^- be the set of emitters (one per in-tree in $\mathcal{T}^-$) and let X^+ be the set of collectors (one per out-tree in $\mathcal{T}^+$). The two sets can overlap, as a vertex may happen to be both an emitter in some tree in $\mathcal{T}^-$ and a collector in some tree in $\mathcal{T}^+$, which is not a problem. Let $H = (X^- \cup X^+, E_H)$ be the graph such that $E_H = \{\{u, v\} \in E : u \in X^-, v \in X^+\}$; in other words, H is the subgraph of G that connects all emitters with all collectors. Finally, let $S = \{\{u, v\} : (u, v) \in \mathcal{T}^- \cup \mathcal{T}^+\} \cup E_H$. We call S a bidirectional fireworks cover (or simply a *fireworks cover*). An illustration is given in Figure 3.12.

Theorem 6. *S is a temporal spanner of $\mathcal{G}$.*

Proof. Every non-emitter vertex can reach at least one emitter u through $e^-(u)$. Every emitter can reach *all* collectors afterwards. Every vertex can be reached by a collector v through $e^+(v)$. □

Theorem 7. *Bidirectional fireworks covers have at most $\binom{n}{2}/2 + O(n)$ edges.*

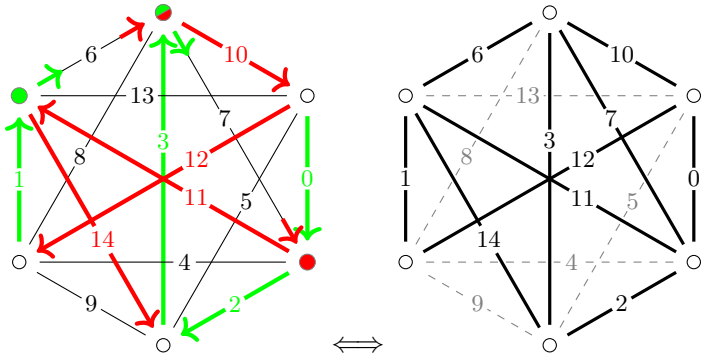


Figure 3.12: A bidirectional fireworks cover and the corresponding spanner. The forward component and the emitters are depicted in (light) green; the backward component and the collectors are depicted in (darker) red. The top vertex is both an emitter and a collector.

Proof. The number of edges in $\mathcal{T}^-$ and $\mathcal{T}^+$ is linear in n. By Lemma 3, the number of emitters is at most $n/2$, and so is the number of collectors by Lemma 5. Some vertices may be both emitter and collector; however, the number of edges is maximized when X^- and X^+ are disjoint, *i.e.*, H is a complete bipartite graph with $n/2$ vertices in each part. Thus, the spanner contains at most $n^2/4$ edges plus the edges of $\mathcal{T}^-$ and $\mathcal{T}^+$. □

3.5 Recursing or sparsifying

After applying the fireworks technique, one is left with a residual instance (or spanner) made of all the edges between emitters X^- and collectors X^+, together with all the edges corresponding to the arcs of $\mathcal{T}^-$ and $\mathcal{T}^+$, these edges being denoted S^- and S^+ for simplicity. As we will see, the algorithm may recurse several times due to dismountability, thus it is worth mentioning that variables $\mathcal{G}$ and V refer to the instance of the current recursion. The algorithm considers two cases, depending on the outcome of the fireworks procedure. Either $X^- \cup X^+ \neq V$ (Case 1) or $X^- \cup X^+ = V$ (Case 2).

Case 1 ($X^- \cup X^+ \neq V$)**.** In this configuration, at least one vertex v is neither emitter nor collector. By Lemma 4, there exists a journey of length at most two from v that arrives at some vertex $u \neq v$ through $e^-(u)$. Similarly, by Lemma 6, there is a journey of length at most two from some vertex $w \neq v$ to v, leaving w through $e^+(w)$. As a result, v is 2-hop dismountable (see Section 3.3). One can thus select the corresponding edges (at most four) for future inclusion in the spanner and recurse on $\mathcal{G}[V \setminus v]$; that is, re-apply

the fireworks technique from scratch. Then, either the recursion keeps entering Case 1 and dismounts the graph completely, or it eventually enters Case 2.

Case 2 ($X^- \cup X^+ = V$)**.** Both X^- and X^+ have size at most $n/2$ (Lemma 3 and 5), thus if their union is V, then both sets must be disjoint and of size exactly $n/2$. As a result, the graph which connects vertices of X^- to vertices of X^+ (called H in Section 3.4) is a complete bipartite graph. In fact, H possesses even more structure; in particular, both S^- and S^+ are perfect matchings—by contradiction, if this is not the case, then at least one of the in-tree (out-tree) contains more than one edge, resulting in strictly less emitters (collectors) than $n/2$. Furthermore, every vertex is either an emitter or a collector, thus each of these edges connects an emitter with a collector, implying that the residual instance is H itself. Now, recall that every edge in S^- is locally minimum for the corresponding emitter (Fact 5.5), and every edge in S^+ is locally maximum for the corresponding collector. We then have the following stronger property.

Lemma 7. *If the minimum edge of an emitter is not also the minimum edge of the corresponding collector in H, then the residual instance is 2-hop dismountable. The same holds if the maximum edge of a collector is not also the maximum edge of the corresponding emitter in H.*

Proof. Let us prove this for minimum edges (a symmetrical argument applies for maximum edges). Consider an emitter u whose minimum edge $\{u, v\} = e^-(u)$ leads to collector v such that $\{u, v\} \neq e^-(v)$ in the bipartite graph. Then an edge with smaller label exists between v and another emitter u', which creates a 2-hop journey from u' to u arriving through $e^-(u)$, implying that u' can delegate its emissions to u. Moreover, emitters and collectors are disjoint, thus u' is not a collector. As a result, u' already delegates its receptions to a collector (through a direct edge), thus u' is 2-hop dismountable. □

Lemma 7 implies that either a vertex v is 2-hop dismountable and the algorithm can recurse as in Case 1, or the edges of the matchings are minimum (resp. maximum) *on both sides*. An example of the latter case is given in Figure 3.13.

In summary, either the algorithm recurses until the input clique is fully dismounted (through Case 1 or Case 2), resulting in a $O(n)$-dense spanner (Fact 4), or the *recursion stops* and the residual instance is sparsified further by a dedicated procedure, described now.

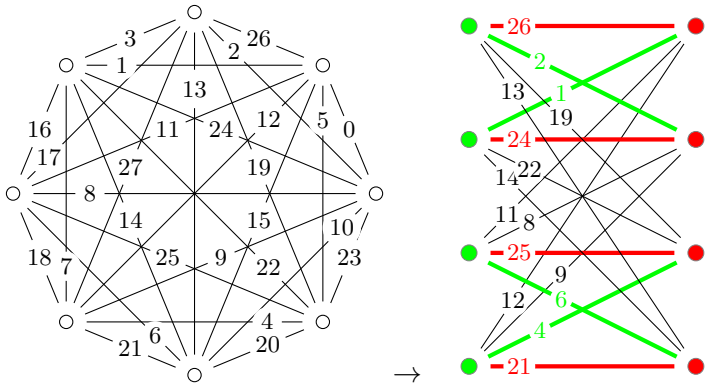


Figure 3.13: Temporal clique for which the output of the fireworks technique results in the second case: every vertex is either an emitter or a collector and the edges of the matchings are minimum (*resp.* maximum) on both sides. The minimum edges are depicted in light green and maximum edges in dark red.

3.5.1 Sparsifying the Residual Instance

For simplicity, the sparsification of the residual instance is considered as a separate problem. The input is a labeled complete bipartite graph $B = (X^-, X^+, E_B)$ where X^- is the set of emitters, X^+ is the set of collectors, and the labels are inherited from $\mathcal{G}$. There are two perfect matchings S^- and S^+ in B such that the labels of the edges in S^- (resp. S^+) are minimum (resp. maximum) locally to both of their endpoints (Lemma 7). The objective is to remove as many edges as possible from E_B, while preserving S^-, S^+, and the fact that every emitter can reach *all* collectors by a journey. Indeed, these three properties ensure temporal connectivity of the graph (using the same arguments as in Theorem 6).

While both S^- and S^+ are matchings, our algorithm effectively exploits this property with respect to S^+ as follows.

Fact 6. *If an emitter can reach another emitter, then it can reach the corresponding collector by adding to its journey the corresponding edge of* S^+.

This property makes it possible to reduce the task of reaching some collectors to that of reaching the corresponding emitter in S^+. It is however impossible for an emitter u to make a complete delegation to another emitter v, because the existence of a journey from u to v arriving through $e^-(v)$ would contradict the fact that S^- is also a matching. For this reason, when a journey from emitter u arrives at emitter v, some of v's edges have already disappeared. Nevertheless, the algorithm exploits such *partial* delegations, while

paying extra edges for the missed opportunities (contained within a logarithmic factor). This is done by means of an iterative procedure called *layered delegations*, described over the remaining of this section. Note the term iterative, not recursive; from now on, the instance has a fixed vertex set and it is sparsified until the final bound is reached.

3.5.1.1 Layered Delegations

The algorithm proceeds by *eliminating* half of the emitters in each step j, while selecting a set S_j of edges for inclusion in the spanner, so that the eliminated emitters can reach all collectors by a mixture of direct edges and indirect journeys through other emitters (partial delegations). The set of non-eliminated emitters at step j (called *alive*) is denoted by X_j^-, with $X_1^- = X^-$. The set of collectors X^+ is invariant over the execution. We denote by $k = n/2$ be the initial degree of the emitters in B (one edge shared with each collector), and by $e^i(v)$ the edge with the i^{th} smallest label (label of *rank* i) locally to a vertex v, in particular $e^1(v) = e^-(v)$ and $e^k(v) = e^+(v)$.

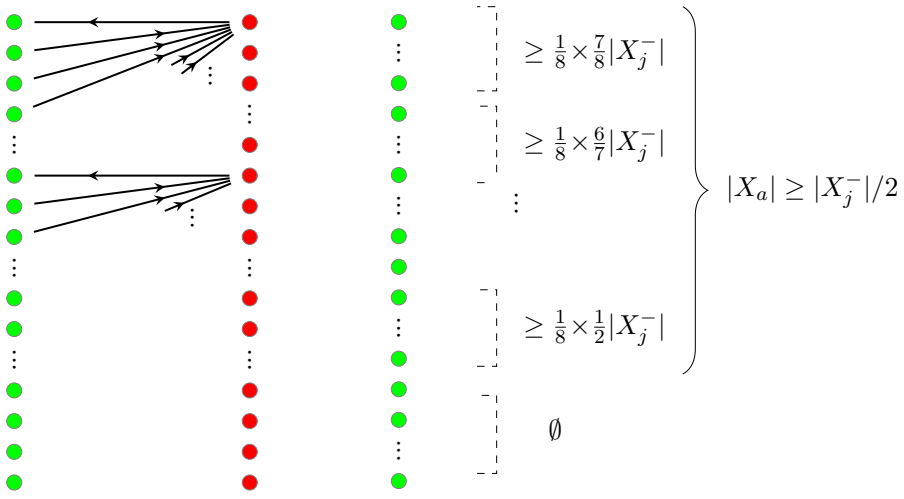
The k ranks are partitioned into subintervals of doubling size $\mathcal{I}_j = [2^{j+2} - 7, 2^{j+3} - 8]$, where j denotes the current step of the iteration, ranging from 1 to $\log_2 k - 3$. For simplicity, assume that k is a power of two, we explain below how to adapt the algorithm when this is not the case. For example, if $k = 128$, then $\mathcal{I}_1 = [1, 8], \mathcal{I}_2 = [9, 24], \mathcal{I}_3 = [25, 56]$, and $\mathcal{I}_4 = [57, 120]$. Computation step j is made with respect to the subgraph $B_j = (X_j^-, X^+, E_j)$ where $E_j = \{e^i(v) \in E_B : i \in \mathcal{I}_j, v \in X_j^-\}$, namely the edges of the currently alive emitters, whose ranks are in the interval $\mathcal{I}_j$.

Lemma 8. *In each step j, X_j^- can be split into two sets X_a and X_b such that $|X_a| \geq |X_b|$ and every vertex in X_a can reach a vertex in X_b through a 2-hop journey (within B_j).*

Proof. This proof is illustrated on Figure 3.14 for the particular case that $j = 1$ (first step). The main idea is to show that the average degree of collectors in B_j forces the existence of sufficiently many two-hop journeys among emitters. To start, observe that if a collector v shares an edge with d emitters in B_j (we say that these emitters *meet* at v), then the emitter whose edge with v has the largest label can be reached by the $d - 1$ other emitters through two-hop journeys. The proof proceeds by showing that, in each step j, the distribution of degrees over collectors forces the existence of sufficiently many such meetings among emitters. Here, the size of the first interval $\mathcal{I}_j$ matters, as if one starts with intervals of size only 2 or 4 (say), then the density of edges remains insufficient for the argument to apply, and starting with 8 (in fact, any constant power of two) does not impact the asymptotic cost, as we will see. Also observe that the doubling

size of the rank intervals cancels out the halving size of X_j^- over the steps, leading to an average degree for collectors that remains constant over the steps (namely, 8).

The generic calculation relative to step j is itself based on an iterative argument that one should be careful not to mistake with the outer loop varying j. Thus, keeping j fixed for the rest of the proof, X_a and X_b are built iteratively as follows: identify the collector c with highest degree and add all the corresponding emitters to X_a except for the one whose edge with c has largest label, which is added to X_b; eliminate these emitters and repeat until $X_a \geq X_j^-/2$, then add the remaining emitters to X_b. To see why this works (and always terminates), observe that the average degree of 8 for collectors forces at least one collector to be of degree 8. In fact, by the pigeon hole principle, this property remains true as long as the number of emitters not being processed yet (*i.e.*, in $X_j^- \setminus (X_a \cup X_b)$) is larger than $7/8 \cdot |X_j^-|$, which guarantees that X_a has size at least $1/8 \cdot 7/8 \cdot |X_j^-|$ when the number of non processed emitters goes below that threshold. An analogue argument forces at least one collector to be of degree 7 so long as the number of non processed emitters is above $6/8 \cdot |X_j^-|$, resulting in at least $1/8 \cdot 6/7 \cdot |X_j^-|$ more emitters in X_a when the next threshold is attained (indeed, six emitters out of the seven considered enter X_a). By iterating this argument, the size of X_a ends up being at least a fraction of X_j^- equal to $1/8 \cdot (7/8 + 6/7 + 5/6 + 4/5 + 3/4 + 2/3 + 1/2) \simeq 0.66 \geq 0.5$. □



(a) Emitters "meeting" at collectors.

(b) Calculation of the size of X_a with respect to the set of emitters X_j^-.

Figure 3.14: Illustration of the method used in the proof of Lemma 8.

Remark 4. *The computation of X_a (described in the proof) proceeds by repeatedly considering the largest degree d of a collector and assigning $d-1$ of the corresponding*

emitters to X_a and one to X_b; it is therefore a greedy algorithm. The process is to be stopped whenever X_a reaches half the size of X_j^-. If X_a exceeds this threshold during step j, then some emitters can be arbitrarily transferred from X_a to X_b to preserve the fact that $|X_{j+1}^-|$ is a power of two. The case that $|X_1^-| = k$ is not a power of two is addressed similarly after the first iteration, in order to set the size of X_b to the highest power of two below k.

How X_a and X_b are then used: When an emitter u in X_a can reach another emitter v in X_b, the corresponding journey arrives at v through some edge $e^i(v)$ with $i \in \mathcal{I}_j$. We say that u partially delegates its emissions to v in the sense that all collectors that v can reach after this time can *de facto* be reached from u (through v), the other collectors being possibly no longer reachable from v after this time. Thus, the delegation is *partial*.

Lemma 9. *If an emitter u makes a partial delegation to v in step j, then the number of collectors that u may no longer reach through v is at most $2^{j+3} - 8$.*

Proof. This number is the largest value in the current interval; it corresponds to the largest rank of the edge through which the journey from u may have arrived at v. All the edges whose rank locally to v is larger than $2^{j+1} - 2$ can still be used and thus the corresponding collectors are still reachable. □

A partial delegation from u to v in step j implies the removal of u from the set of emitters, the selection of the two edges of the journey from u to v, and the selection of at most $2^{j+3} - 8$ direct edges between u and the missed collectors. This implies the following fact.

Fact 7. *In each step j, at most 2^{j+3} edges are selected relative to every eliminated emitter.*

More globally, let J_j be the edges used in all the delegation journeys from vertices in X_a to vertices in X_b in step j, and D_j the union of direct edges towards missed collectors. Let $S_j = J_j \cup D_j$. The algorithm thus consists of selecting all the edges in S_j for inclusion into the spanner. Then X_{j+1}^- is set to X_b and the iteration proceeds with the next step. The computation goes for j ranging from 1 to $\log_2 k - 3$, which leaves exactly *eight* final emitters alive. All the remaining edges of these emitters (call them S_{last}) are finally selected. Overall, the final spanner is the union of all selected edges, plus the edges corresponding to the two initial matchings, *i.e.*, $S = (\cup_j S_j) \cup S_{last} \cup S^- \cup S^+$.

Theorem 8. *S is a temporal spanner of the complete bipartite graph B and it is made of $\Theta(n \log n)$ edges.*

Proof. The key observation for establishing *validity* of the spanner is that eliminated emitters reach all collectors either directly or through an emitter that can still reach this collector *afterwards*. This property applies transitively (thanks to the disjoint and increasing intervals) until eight emitters remain, all the edges of which are selected for simplicity. Therefore, every initial emitter can reach all collectors. The rest of the arguments are the same as in the proof of Theorem 6: all vertices in the input clique can reach at least one emitter u through $e^-(u)$, and be reached by at least one collector v through $e^+(v)$.

Regarding the size of the spanner, in step j, $\frac{k}{2^j}$ emitters are eliminated and at most 2^{j+3} edges are selected for each of them (Fact 7), amounting to at most $8k = 4n$ edges. The number of iterations is $\Theta(\log k) = \Theta(\log n)$. Finally, the sets S_{last}, S^-, and S^+ each contain $\Theta(n)$ edges (and S^+ is actually included in S_{last}). □

Corollary 1. *Simple temporal cliques always admit $O(n \log n)$-sparse spanners.*

Proof. In each recursion of the global algorithm, either the residual instance of the fireworks procedure is 2-hop dismountable and the algorithm recurses on a smaller instance induced by a removed vertex, after selecting a *constant* number of edges, or the algorithm computes a $\Theta(n \log n)$-sparse spanner of the residual instance through the layered delegation process. Let n_1 be the number of times the graph is 2-hop dismounted and $n_2 = n - n_1$ be the number of vertices of the residual instance when the layered delegation process begins (if applicable, 0 otherwise). The resulting spanner has $\Theta(n_1) + \Theta(n_2 \log n_2) = O(n \log n)$ many edges. □

3.6 Time complexity

This short section reviews the cost in time of the main components involved in the algorithm. This discussion is by no means a detailed analysis, its purpose is rather to sustain the claim that the overall algorithm runs in polynomial time. To start, the input is a temporal clique $\mathcal{G} = (G, \lambda)$, where $G = K_n$ is the complete graph on n vertices, and λ a simple labelling of the edges, which may be represented as a permutation π of $(1, 2, 3, ..., \binom{n}{2})$. The global algorithm is portrayed in Figure 3.15. Observe that whenever the algorithm recurses, the number of vertices is reduced by one, and in each recursion the

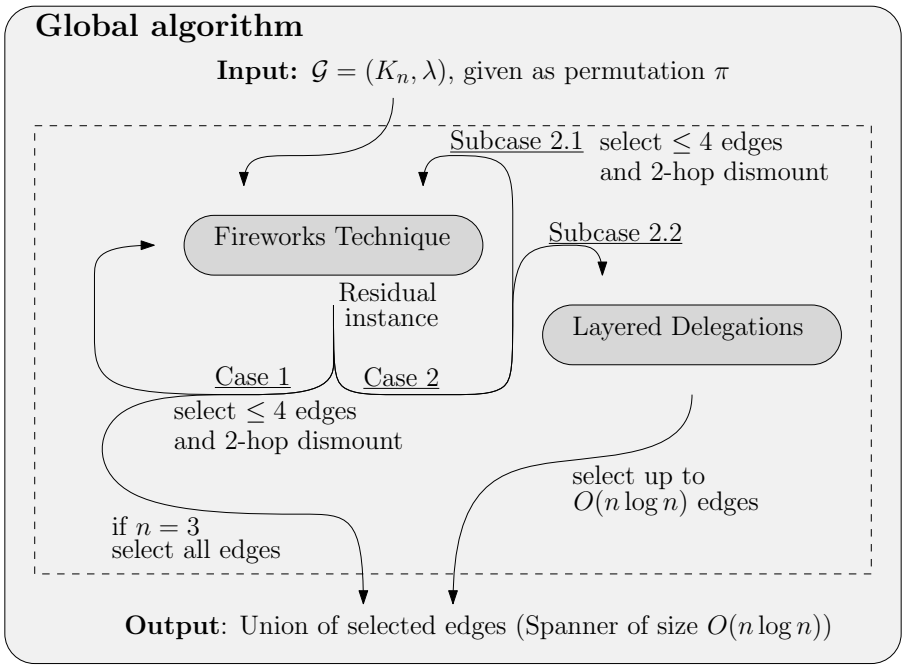


Figure 3.15: The global algorithm, using the fireworks technique, dismounting, and layered delegations.

fireworks process is run twice (forward and backward), possibly followed by the layered delegation processes, in which case the algorithm subsequently terminates. The fireworks process will thus run less than $2n$ times and the delegation process at most one time. The fireworks process first identifies the edges which are minimum (maximum) for at least one vertex. Then, it transforms this structure by means of a set of local operations consisting of flipping edges (at most once) or discarding them. As for layered delegations, the main operation is the composition of the delegation sets X_a and X_b, which is done a logarithmic number of times by a greedy procedure whose main operation is to examine the degrees of all collectors to detect the local maximum among their labels. In light of these observations, we hope that it is clear to the reader that the overall running time is polynomial.

3.7 Conclusion

In this chapter, we established that sparse temporal spanners always exist in temporal cliques, proving constructively that one can find $O(n \log n)$ edges that suffice to preserve temporal connectivity. Our results hold for non-strict journeys with single or multiple

labels on each edge, and strict journeys with single or multiple labels on each edge with the property that there is a subset of locally exclusive single labels. Our results give the first positive answer to the question of whether any class of dense graphs always has sparse temporal spanners.

To prove our results, we introduced several techniques (pivotability, delegation, dismountability and k-hop dismountability, forward and backward fireworks, partial delegation, and layered delegations), all of which are original and some of which might be of independent interest. Whether some of these techniques can be used for more general classes of graphs is an open question. Delegation and dismounting rely explicitly on the graph being complete; however, refined versions of these techniques like partial delegation might have wider applicability.

We decide to discuss the open questions regarding temporal spanners in 6.

3.7.1 Transition between chapters

Changing the setting and the context a bit, the next chapter will consider an original problem, first studied during my thesis, surrounding motion planning and the well-known optimization problem TSP. The current chapter and the next one may thus at first seem quite unrelated, but we remind the reader that Chapter 5 can be considered as a link between the two subjects, since we'll be interested in temporal graph properties induced through mobile agents' controlled mobility.

Chapter 4

VectorTSP: A Traveling Salesperson Problem with Racetrack-like acceleration constraints

Nothing happens until something moves.

— Albert Einstein

Contents

This chapter is based on our work on VectorTSP, recently presented and published in the proceedings of Algorithms for Sensor Networks (ALGOSENSORS) 2020 [27]. The full version is about to be submitted to a journal. This work is available on ArXiv as well [28].

We study a new version of the Euclidean TSP called VectorTSP (VTSP for short) where a mobile entity is allowed to move according to a set of physical constraints inspired from the pen-and-pencil game [45] (also known as *Vector Racer*). In contrast to other versions of TSP accounting for physical constraints, such as Dubins TSP, the spirit of this model is that (1) no speed limitations apply, and (2) inertia depends on the current velocity. As such, this model is closer to typical models considered in path planning problems, although applied here to the visit of n cities in a non-predetermined order.

We motivate and introduce the VectorTSP problem, discussing fundamental differences with previous versions of TSP. In particular, an optimal visit order for ETSP may not be optimal for VTSP. We show that VectorTSP is NP-hard, and in the other direction, that VectorTSP reduces to GroupTSP in polynomial time (although with a significant blow-up in size). On the algorithmic side, we formulate the search for a solution as an interactive scheme between a high-level algorithm and a *trajectory oracle*, the former being responsible for computing the visit order and the latter for computing the cost (or the trajectory) for a given visit order. We present algorithms for both, and we demonstrate and quantify through experiments that this approach frequently finds a better solution than the optimal trajectory realizing an optimal ETSP tour, which legitimates the problem itself and (we hope) motivates further algorithmic developments.

4.1 Introduction

The problem of visiting a given set of places and returning to the starting point, while minimizing the total cost, is known as the Traveling Salesperson Problem (TSP, for short). The problem was independently formulated by Hamilton and Kirkman in the 1800s and has been extensively studied since. Many versions of this problem exist, motivated by applications in various areas, such as delivery planning, stock cutting, and DNA reconstruction. In the classical version, an instance of the problem is specified as a graph whose vertices represent the *cities* (places to be visited) and weights on the edges represent the cost of moving from one city to another (the move is impossible if the

edge does no exist). One is asked to find the minimum cost tour (optimization version) or to decide whether a tour having at most some cost exists (decision version) subject to the constraint that every city is visited *exactly* once. Karp proved in 1972 that the Hamiltonian Cycle problem is NP-hard, which implies that TSP is NP-hard [59]. TSP was subsequently shown to be inapproximable (unless $P = NP$) by Orponen and Manilla in 1990 [75]. On the positive side, while the trivial algorithm has a factorial running time (essentially, evaluating all permutations of the visit order), Held and Karp presented a dynamic programming algorithm [53] running in time $O(n^2 2^n)$, which as of today remains the fastest we known.

In many cases, the problem is restricted to more tractable settings. In Metric TSP, the costs must respect the triangle inequality, namely $cost(u, v) \leq cost(u, w) + cost(w, v)$ for all u, v, w, and the constraint of visiting a city exactly once is relaxed (or equivalently, it is not, but the instance is turned into a complete graph where the weight of every edge uv is the cost of a shortest *path* from u to v in the original instance). Metric TSP was shown to be approximable within factor 1.5 by Christofides [31]. Whether the factor is optimal is unknown, although it cannot be less than 1.0045 (unless $P = NP$) and so no PTAS exists for Metric TSP [76]. A particular case of Metric TSP is when the cities are points in the plane, and weights are the Euclidean distance between them, known as the Euclidean TSP (ETSP, for short). This problem, although still NP-hard (see Papadimitriou [77] and Garey *et al.* [46]), was shown to admit a PTAS by Arora [8] and Mitchell [70].

An attempt to add physical constraints to the ETSP is Dubins TSP (DTSP). This version of TSP, which is also NP-hard (Le Ny *et al.* [65]), accounts for inertia through bounding by a fixed radius the curvature of a trajectory. This approach offers an elegant (*i.e.* purely geometrical) abstraction to the problem. However, it does not account for speed variations; for example, it does not enable sharper turns when the speed is low, nor does it account for inertia beyond a fixed speed. More realistic models have been considered beyond TSP, such as in the context of the path planning problem, where one aims to find an optimal trajectory between two given points (with obstacles), while satisfying constraints on acceleration/inertia. More generally, the literature on *kinodynamics* is vast (see, e.g. [16, 17, 39] for some relevant examples). The constraints are often formulated in terms of the considered space's dimensions, a bounded acceleration and a bounded speed. The positions may either be considered in a discrete domain or continuous domain, the latter being more related to the fields of control theory and analytic functions. In constrast, the discrete domain is naturally prone to algorithmic investigation.

Savla *et al.* presented a simple but effective algorithm, adapting every other segment of an ETSP solution to obey the curvature constraint [82]. In the same paper, the authors prove there exists a constant C such that for any instance on n points with optimal ETSP tour length L, the optimal DTSP tour for this instance has length L' with $L \leq L' \leq L + Cn$. A DTSP trajectory is often thought of as some vehicle on which the wheels cannot be turned more than some angle, *e.g.* a car, thus resulting in its trajectory being curvature-constraint. A DTSP trajectory can however also be thought of as an entity moving at some constant speed, which creates some constant and fixed inertia force, thus resulting in its trajectory being curvature-constraint. Note that DubinsTSP also has the property that the optimal visit order may be different from that of the ETSP [65].

Considering Racetrack, presented in 2.4.4, Bekos *et al.* [12] present efficient algorithms for simpler tracks of uniform width, as well as algorithms for when the parts of the track are known as soon as they become visible during the race. The results mentionned above are all under the assumption the Racetrack instance is encoded as a 2-dimensional array. A parameter to this problem is whether or not the entity is allowed to touch the defined boundaries, as this changes the hardness of the problem. Indeed, Holzer and McKenzie have proven in 2010 that the non-touching variant is NL-complete, and the touching variant is in L [55]. They also proved that the reachability problem with a given time unit limit is NL-complete (also called single-player Racetrack), regardless of touching, as well as deciding the existence of a winning strategy in Gardner's original two-player game is P-complete [55]. Erickson states a more natural input representation than the array is a set of line segments that delimit the boundary of the Racetrack. Erickson only considers the non-touching variant. Under these assumptions, he proves the single-player Racetrack problem belongs to PSPACE, and computing an optimal racing strategy needs exponential time, as any explicit description of the optimal path could have exponential complexity.

4.1.1 Contributions

Defining a version of TSP based on a racetrack-like physical model is quite natural. Consider, for instance, a scenario involving a spacecraft in a simplified physical setting (*i.e.* non-relativized and without gravity), where no speed limit applies and acceleration constraints are identical in all directions. Finding the best tour visiting a given set of planets, or asking whether such a tour can be performed in a given time are indeed natural questions and objectives. Another, perhaps more realistic, scenario involves a

drone taking aerial pictures of a set of locations. Despite an extensive literature, the TSP problem does not seem to have been investigated from the point of view of pure acceleration. (Anecdotally, there exists a TSP heuristics called "racetrack" [89], which does not relate to such models, nor to acceleration in general.)

In this chapter, we introduce a version of the Traveling Salesperson Problem called VectorTSP (or VTSP), in which a vehicle must visit a given set of points in some Euclidean space and return to the starting point, subject to racetrack-like constraints. The quality of a solution is the *number* of vectors (equivalently, of configurations) it uses. We start by presenting a generalized racetrack physical model, in Section 3.2.1, and reviewing some of its algorithmic features, including known techniques based on the graph of configurations. Then, we define the VTSP problem in a quite general setting, where the space may be discrete or continuous, in an arbitrary number of dimensions (namely, $\mathbb{Z}^d$ or $\mathbb{R}^d$). An instance may be parameterized by two additional parameters: the maximum speed at which a city is considered as visited (visit speed ν), the speed being otherwise unbounded; and the maximum distance at which a city is considered as visited (visit distance α). These parameters correspond to natural motivations. For example, if the aforementioned space mission consists of dropping or collecting passengers in given "city", then the vehicle might need to slow down (or stop) at visit time; if it consists of making quick measurements, then the visit speed is unconstrained and some distance from the visited city may even be tolerated.

In Section 4.3, we make a number of general observations about VTSP. In particular, optimizing the racetrack trajectory of an optimal ETSP tour may not result in an optimal VTSP solution: the visit order is impacted by acceleration. Another key observation is that even if the speed is unbounded, one can easily compute a loose bound on the maximal speed to be considered in the search for an optimal solution, with important consequences on the computational complexity of the problem. In fact, we prove that VTSP is NP-hard under a natural parameterization (and therefore, in general), and in the other direction, it polynomially reduces to GroupTSP, however with a significant blow-up in the input size. On the algorithmic side, we present in Section 4.4 a modular approach to address VTSP based on an interactive scheme between a high-level algorithm and a trajectory oracle. The first is responsible for exploring the space of possible visit orders, while making queries to the second for knowing the cost (or full trajectory) associated with a given visit order. We present algorithms for both. The high-level algorithm adapts a known heuristic for ETSP, trying to gradually improve the solution through generating a set of 2-permutations (swaps of two cities) until a local optimum is found. As for the oracle,

we present an algorithm which adapts the A* framework to multipoint paths in the configuration space, using an original cost function based on unidimensional projections of the cities coordinates.

In Section 4.5, we present a few experimental results based on this algorithmic framework. Beyond demonstrating the practicality of our algorithms, our results motivate the problem itself, by showing empirical evidence that the optimum trajectory resulting from an optimal ETSP tour is unlikely to be optimal for VTSP, and so, in most natural settings. In particular, the probability that our algorithm improves upon such a trajectory seems to approach 1 as the number of cities increase in a fixed area.

4.2 Model and definitions

In this section, we present a generalized version of the racetrack model, highlighting some of its algorithmic features. Then, we define VectorTSP in generality, making observations and presenting preliminary results that are used in the subsequent sections.

4.2.1 Generalized Racetrack model

Let us consider a mobile entity (hereafter, the *vehicle*), moving in a discrete or continuous Euclidean space $\mathbb{S}$ of some dimension d (for example, $\mathbb{S} = \mathbb{Z}^2$ or $\mathbb{S} = \mathbb{R}^3$). The state of the vehicle at any time is given by a *configuration* c, which is a couple containing a position $pos(c)$ and a velocity $vel(c)$, both encoded as elements of $\mathbb{S}$. For example, if $\mathbb{S} = \mathbb{Z}^2$, then a configuration c is of the form $((x, y), (dx, dy))$. Furthermore, we write $speed(c)$ for $||vel(c)||$. Given a configuration c, the set of configurations being reachable from c in a single time step, *i.e.*, the successors of c, is written as $\texttt{succ}(c)$ and is model-dependent.

The original model presented by Gardner [45] corresponds to the case that $\mathbb{S} = \mathbb{Z}^2$, and given two configurations c_i and c_j, written as above, $c_j \in \texttt{succ}(c_i)$ if and only if $x_j = x_i + dx_i + \chi$ and $dx_j = x_j - x_i$, and $y_j = y_i + dy_i + \psi$ and $dy_j = y_j - y_i$, where $\chi, \psi \in \{-1, 0, 1\}$. In other words, the velocity of a configuration corresponds to the difference between its position and the position of the previous configuration, and this difference may only vary by one unit in each dimension in one time step. In the following, we refer to this model as the default setting, or sometimes 9-successor model, since we sometimes consider the case that at most one dimension can change in one time step, which will be referred to as the 5-successor model. These models can be naturally

extended to continuous space, by considering that the set of successors is infinite, typically amounting to choosing a point in a d-sphere, as illustrated on Figure 4.1.

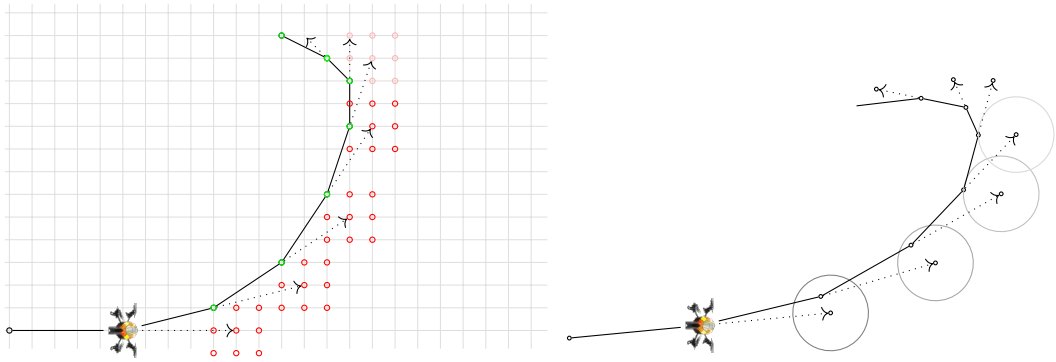


Figure 4.1: Discrete and continuous space racetrack models (left and right, respectively).

Definition 22 (Trajectory)**.** *A trajectory (of length k) is a sequence of configurations $c_1, c_2, ..., c_k$. It is called* valid *if $c_{i+1} \in succ(c_i)$ for all $i < k$.*

We define the inverse c^{-1} of a configuration c as the configuration that represents the same movement in the opposite direction. For example, if $\mathbb{S} = \mathbb{Z}^2$ and $c = ((x, y), (dx, dy))$, then $c^{-1} = ((x + dx, y + dy), (-dx, -dy))$. A successor function is *symmetrical* if $c_j \in \mathtt{succ}(c_i)$ if and only if $c_i^{-1} \in \mathtt{succ}(c_j^{-1})$. Intuitively, this implies that if $(c_1, c_2, \ldots, c_k)$ is a valid trajectory, then $(c_k^{-1}, \ldots, c_2^{-1}, c_1^{-1})$ is also a valid trajectory: the trajectory is *reversible*. All the successor functions considered in this chapter are symmetrical; however, it could make sense in general to consider non-symmetrical successor functions, for example if the vehicle can decelerate faster than it can accelerate.

4.2.1.1 Configuration space

The concept of *configuration space* is a powerful and natural tool in the study of racetrack-like problems. This concept was rediscovered many times and is now considered as folklore. The idea is to consider the graph of configurations induced by the successor function as follows.

Definition 23 (Configuration graph)**.** *Let $\mathcal{C}$ be the set of all possible configurations, then the* configuration graph *is the directed graph $G(\mathcal{C}) = (V, E)$ where $V = \mathcal{C}$ and $E = \{(c_i, c_j) \subseteq \mathcal{C}^2 : c_j \in succ(c_i)\}$.*

The configuration graph $G(\mathcal{C})$ is particularly useful when the number of successors of a configuration is bounded by a constant. In this case, $G(\mathcal{C})$ is sparse and one can search for optimal trajectories within it, using standard algorithms like breadth-first search (BFS). For example, in a $L \times L$ subspace of $\mathbb{Z}^2$, there are at most L^2 possible positions and at most $O(L)$ possible velocities (the speed cannot exceed $\sqrt{L}$ in each dimension without getting out of bounds [42]), thus $G(\mathcal{C})$ has $\Theta(L^3)$-many vertices and edges. More generally:

Observation 2 (Folklore)**.** *A breadth-first search (BFS) in a $L \times L$ subspace of $\mathbb{Z}^2$ can find an optimum trajectory between two given configurations in time $O(L^3)$. A similar observation leads to time $O(L^{9/2})$ in $\mathbb{Z}^3$, and more generally $O(L^{3d/2})$ in dimension d.*

Note that the presence of obstacles (if any) results only in the graph having possibly less vertices and edges. (We do not consider obstacles in this work.)

4.2.2 Definition of VectorTSP

Informally, VectorTSP is defined as the problem of finding a minimum length trajectory (optimization version), or deciding if a trajectory of at most a given length exists (decision version), which visits a given set of unordered cities (points) in some Euclidean space, subject to racetrack-like physical constraints. As explained in the introduction, we consider additional parameters to the problem, which are (1) *Visit speed* ν: maximum speed at which a city is visited; (2) *Visit distance* α: maximum distance at which a city is visited; and (3) *Vector completion* β: $(true/false)$ whether the visit distance is evaluated only at the coordinates of the configurations, or also in-between configurations. See Figure 4.2 for a small example in which a city is considered visited only for some specific values of these parameters. In general, we say a trajectory visits a city, but in particular, one can pinpoint a specific vector (or configuration) fo such a trajectory which visits the city. The first two parameters' context are already discussed in the introduction. The visit distance is actually similar in spirit to the *TSP with neighborhood* [7]. The third parameter is more technical, although it could be motivated by having a specific action (sensing, taking pictures, etc.) being realized only at periodic times, for minimizing energy consumption.

We are now ready to define VectorTSP. For simplicity, the definitions rely on discrete space ($\mathbb{S} = \mathbb{Z}^d$), to avoid technical issues with the representation of real numbers, in particular their impact on the input size. Similarly, we require the parameters ν and α to be integers and β to be a boolean. However, the problem might be adaptable to

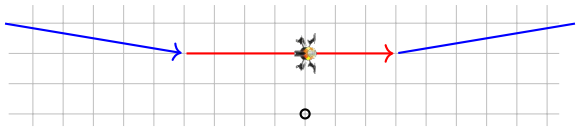


Figure 4.2: A trajectory visiting some city $p = (x, y)$, if $\nu \geq 7$, $\alpha \geq 2$, and $\beta = false$. If either $\nu < 7$, $\alpha < 2$, or $\beta = true$, the city is not visited. More precisely, the red vector corresponding to configuration $(x + 3, y + 2, 7, 0)$ visits the city.

continuous space without much complications, possibly with the use of a *real RAM* abstraction [79].

Definition 24. VECTORTSP *(decision version)*

> ***Input:*** *A set of n cities (points) $P \subseteq \mathbb{Z}^d$, a distinguished city $p_0 \in P$, two integer parameters ν and α, a boolean parameter β, a polynomial-time-computable successor function* `succ`*, a positive integer k, and a trivial bound Δ encoded in unary.*
>
> ***Question:*** *Does there exist a valid trajectory $\mathcal{T} = (c_1, \ldots, c_k)$ of length k that visits all the cities in P, with $pos(c_1) = pos(c_k) = p_0$ and $speed(c_1) = speed(c_k) = 0$.*

The role of parameter Δ is to guarantee that the length of the optimal trajectory is polynomially bounded in the size of the input. Without it, an instance of even two cities could be artificially hard due to the sole distance between them [55, 42]. As we will see, one can always find a (possibly sub-optimal) solution trajectory of $poly(L)$ configurations, where L is the maximum distance between two points in any dimension, and similarly, a solution trajectory must have length at least $\sqrt{L}$. Therefore, writing $\Delta = \texttt{unary}(\lfloor\sqrt{L}\rfloor)$ in the input is sufficient. The optimization version is defined analogously.

Definition 25. VECTORTSP *(optimization version)*

> ***Input:*** *A set of n cities (points) $P \subseteq \mathbb{Z}^d$, a distinguished city $p_0 \in P$, two integer parameters ν and α, a boolean parameter β, a polynomial-time-computable successor function* `succ`*, and a trivial bound Δ encoded in unary.*
> ***Output:*** *Find a valid trajectory $T = (c_1, \ldots, c_k)$ of minimum length visiting all the cities in P, with $pos(c_1) = pos(c_k) = p_0$ and $speed(c_1) = speed(c_k) = 0$.*

Tour *vs.* trajectory (terminology): In the Euclidean TSP, the term *tour* denotes both the visit order and the actual path realizing the visit, because both coincide. In VECTORTSP, a given visit order could be realized by many possible trajectories. To avoid ambiguities,

we always refer to a visit order (*i.e.,* a permutation π of P) as a *tour*, while reserving the term *trajectory* for the actual sequence of racetrack configurations. Furthermore, we denote by `racetrack(`π`)` an optimal (*i.e.*, min-length) racetrack trajectory realizing a given tour π (irrespective of the quality of π).

Default setting: In the rest of the chapter, we call *default setting* the 9-successor model in two dimensional discrete space ($\mathbb{S} = \mathbb{Z}^2$), with unrestricted visit speed ($\nu = \infty$), zero visit distance ($\alpha = 0$), and non-restricted vector completion ($\beta = false$). Most of the results are however transposable to other values of the parameters and to higher dimensions.

4.3 Preliminary results

In this section we make general observations about VECTORTSP, some of which are used in the subsequent sections. In particular, we highlight those properties which are distinct from Euclidean TSP.

Fact 8. *The starting city has an impact on the cost of an optimal solution.*

Figure 4.3: Minimal example showing the starting point has an impact on the solution. On the left, the starting point is the leftmost city, while on the right, the starting point is the middle city.

Example. This can be seen on a small example (see Figure 4.3), with $P = \{(0,0),(1,0),(2,0)\}$ in the default setting. Starting at $(0,0)$, a solution exists with 7 configurations (*i.e.,* 6 vectors), namely $T = (((0,0),(0,0)), ((1,0),(1,0)), ((2,0),(1,0)), ((2,0),(0,0)), ((1,0),(-1,0)), ((0,0),(-1,0)), ((0,0),(0,0)))$. In contrast, if the tour starts at $(1,0)$, the vehicle will have to decelerate three times instead of two, which gives a trajectory of 8 configurations (7 vectors). □

This fact is the reason why an input instance of VECTORTSP is also parameterized by a starting city $p_0 \in P$. More generally, the cost of traveling between two given cities is impacted by the previous and subsequent positions of the vehicle and cannot be captured

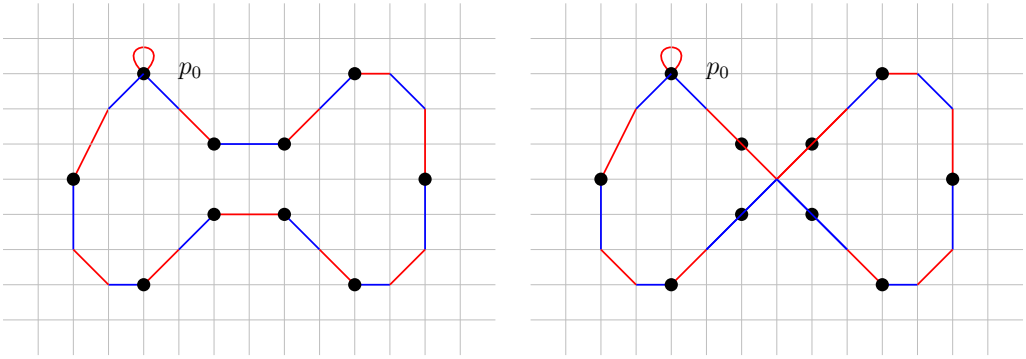


Figure 4.4: Small example showing `racetrack(`π`)` (on the left, using 22 vectors) may not be an optimal solution (on the right, using 20 vectors).

by a fixed cost, which is why VTSP does not straightforwardly reduce to classical TSP. The following fact strengthens the distinctive features of VTSP, showing that it does not straightforwardly reduce to ETSP either.

Lemma 10. *Let $\mathcal{I}$ be a* VTSP *instance on a set of cities P, in the default setting. Let π be an optimal tour for an ETSP instance on the same set of cities P, then* `racetrack(`π`)` *may not be an optimal solution to $\mathcal{I}$.*

Proof. Consider the example given in Figure 4.4, where the trajectories alternate between dashed red and plain blue vectors. On the left picture, the trajectory corresponds to an optimal realization of the optimal ETSP tour π, starting and ending at p_0 (whence the final deceleration loop). It it not hard to see that this trajectory is indeed optimal for π. In contrast, an optimal VTSP trajectory visiting the same cities (right picture) would use two configurations less, based on a non-optimal tour π' for ETSP. This lemma is also true if maximum visiting speed $\nu = 0$ (see Figure 4.5). □

Hence, solving VTSP does not reduce to optimizing the trajectory of an optimal ETSP solution: the visit order is impacted. Furthermore, we observe the following two properties on Figure 4.4 and Figure 4.5 respectively, distinguishing VectorTSP even further from EuclideanTSP.

Fact 9. *An optimal* VTSP *solution may self-cross.*

Fact 10. *The clockwise or counter-clockwise visit order may not be an optimal* VTSP *solution for cities placed on the vertices of a convex hull.*

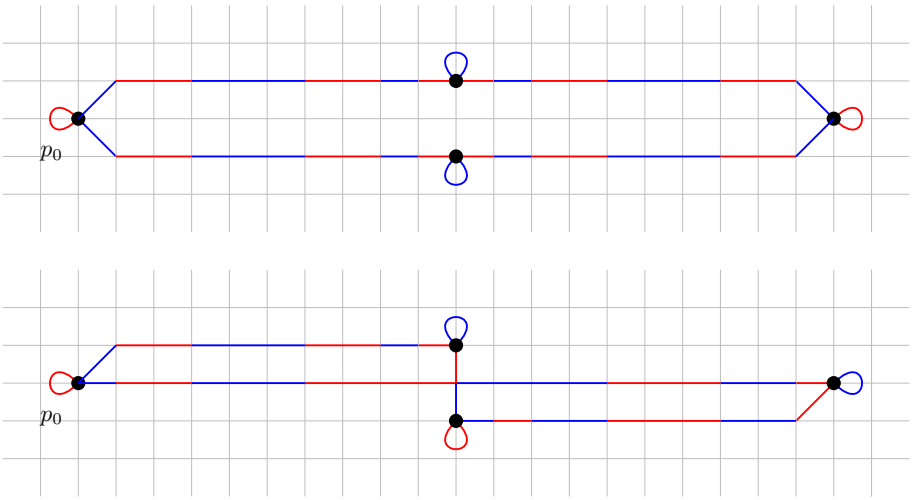


Figure 4.5: Small example showing `racetrack(π)` (on the left, using 28 vectors) may not be an optimal solution (on the right, using 26 vectors), even if maximum visiting speed $\nu = 0$.

4.3.1 The configuration space can be bounded

The spirit of the racetrack model is to focus on acceleration only, without bounding the speed. Nonetheless, we show here that a VECTORTSP trajectory in general (and an optimal one in particular) can always be found within a certain subgraph of the configuration graph, whose size is polynomially bounded in the size of the input. These results are formulated in the default setting for any discrete d-dimensional space.

Lemma 11 (Bounds on the solution length)**.** *Let P be a set of cities and L be the largest distance in any dimension (over all d dimensions) between two cities of P. Then a solution trajectory must contain at least $\sqrt{L}$ configurations. Furthermore, there always exists a solution trajectory of $O(L^d)$ configurations.*

Proof. The lower bound follows from the fact that it takes at least $\sqrt{L}$ configurations to travel a distance of L (starting at speed 0), the latter being a lower bound on the total distance to be traveled. The upper bound can be obtained by exploring all the points of the d-dimensional rectangular hull containing the cities in P at unit speed, which amounts to $O(L^d)$ configurations. □

Lemma 12 (Bounds on the configuration graph)**.** *An (optimal) trajectory for VTSP can be found in a subgraph of the configuration graph with polynomially many vertices and edges (in the size of the input), namely $O(L^{(d^2)})$.*

Proof. First observe that if there exists a trajectory of $O(L^d)$ configurations, then this bound also applies to an optimal trajectory. Now, we know that a trajectory corresponds to a path in $G(\mathcal{C})$, thus an optimal trajectory can be found within the subgraph of $G(\mathcal{C})$ induced by the vertices at distance at most $O(L^d)$ from the starting point, which consists of $O(L^{(d^2)})$ vertices in total. □

4.3.2 A glimpse at computational complexity

Here, we present polynomial time transformations from VectorTSP to other NP-hard problems and vice versa. Even after significant time spent trying to obtain a natural reduction from a closely related NP-hard problem, *e.g.* EuclideanTSP or DubinsTSP, no such a result was found. At some point we tried to adapt Le Ny *et al.*'s proof of DubinsTSP's NP-hardness (see [65]), where the authors use Papadimitrou's proof of EuclideanTSP's NP-hardness as somewhat of a black box. However, it turns out that a key property of Le Ny *et al.*'s proof is that an optimal EuclideanTSP tour is of cost (or distance) at most the cost of an optimal tour of DubinsTSP, which unfortunately is not the case for VectorTSP. Eventually, we adapted Papadimitrou's proof directly, in which we establish a reduction from ExactCover, which is less natural, as well as technical and lengthy. Our main modifications of Papadimitriou's work are the careful repositioning and distancing of cities so as to force the VectorTSP trajectory to use an exact amount of vectors between some pairs of cities, and to avoid the VectorTSP trajectory being able to visit other cities than the predetermined visit order using the same amount of vectors. Through this, we establish NP-hardness of a particular parameterization of VectorTSP (and thus, of the general problem) where the visit speed ν is zero.

We also present a general reduction from VectorTSP to GroupTSP. This reduction relies crucially on Lemma 12 above.

4.3.2.1 NP-hardness of VectorTSP

The proof goes through a number of intermediate steps before ending in the main theorem stating VectorTSP is NP-hard, being Theorem 10.

Let us first recall the definition of ExactCover. Let $\mathcal{U}$ be a set of m elements (the *universe*), the problem ExactCover takes as input a set $\mathcal{F} = \{F_i\}$ of n subsets of $\mathcal{U}$, and asks if there exists $\mathcal{F}' \subseteq \mathcal{F}$ such that all sets in $\mathcal{F}'$ are *disjoint* and $\mathcal{F}'$ covers all the elements of $\mathcal{U}$. For example, if $\mathcal{U} = \{1, 2, 3\}$ and $\mathcal{F} = \{\{1, 2\}, \{3\}, \{2, 3\}\}$, then $\mathcal{F}' = \{\{1, 2\}, \{3\}\}$ is a valid solution, but $\{\{1, 2\}, \{2, 3\}\}$ is not.

Given an instance $\mathcal{I}$ of ExactCover, the proof shows how to construct an instance $\mathcal{I}$' of VTSP such that $\mathcal{I}$ admits a solution if and only if there is a valid trajectory visiting all the cities of $\mathcal{I}$' using at most a certain number of configurations. We first give the high-level ideas of the proof, which are in common with that of Papadimitriou's proof for ETSP. Then, we explain the details of the adaptation to VTSP (with visit speed $\nu = 0$).

The instance $\mathcal{I}'$ is composed of several types of gadgets, representing respectively the subsets $F_i \in \mathcal{F}$ and the elements of $\mathcal{U}$ (with some repetition). For each F_i, a subset gadget C_i is created which consists of a number of cities placed horizontally (wavy horizontal segments in Figure 4.6). For now, it is sufficient to know that each gadget can be traversed optimally in exactly two possible ways (without considering direction), which ultimately corresponds to including (traversal 1) or excluding (traversal 2) subset F_i in the ExactCover solution. The C_i's are located one below the other, starting with C_1 at the top. Between every two consecutive gadgets C_i and C_{i+1}, copies of *element* gadgets are placed for each element in $\mathcal{U}$, thus the element gadgets H_{ij} are indexed by both $1 \leq i \leq n-1$ and $1 \leq j \leq m$ (see Figure 4.6). The element gadgets are also made of a number of cities, whose particular organization is described later on. Finally, every subset gadget C_i above or below an element gadget representing element j is slightly modified in a way that represents whether F_i contains element j or not.

Intuitively, a tour visiting all the cities must choose between inclusion or exclusion of each F_i (*i.e.*, traversal 1 or 2 for each C_i). An element $j \in \mathcal{U}$ is considered as covered by a subset F_i if C_i does *not* visit any of the adjacent element gadgets representing j. Each element gadget $H_{i,j}$ must be visited either from above (from C_i) or from below (from C_{i+1}). Now, the number of subset gadget is n, the number of element gadgets for each element is $n-1$ (one between every two consecutive subset gadgets), and the construction guarantees that at most one element gadget for each element $j \in \mathcal{U}$ is visited from a subset gadget C_i (or the tour is non-optimal). These three properties collectively imply that for each element $j \in \mathcal{U}$, there is exactly one subset gadget C_i that does not visit any of the element gadgets representing j.

In summary, the tour proceeds from the top left corner through the C_is (in order), visiting all the $H_{i,j}$ through local detours. So long as a C_i visits a $H_{i,j}$ (thus, from above), this means that element j has not yet been selected in the ExactCover solution. Element j is covered by subset F_i in the ExactCover solution if C_i is the first subset gadget that does *not* visit the corresponding $H_{i,j}$ (which must eventually happen), after which all the $H_{i,k<j}$ will necessarily be visited (i.e. not covered again) from below by the corresponding C_{k+1}. The details of the construction specify the internal organization of

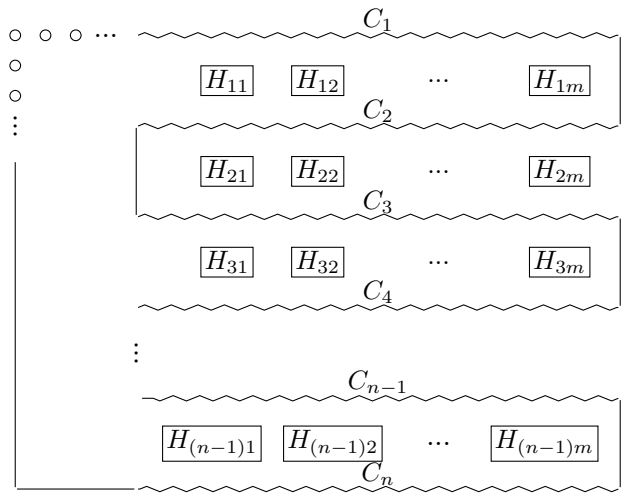


Figure 4.6: Papadimitriou's high-level construction

each gadget (positions of the cities composing it), and the spacing between the cities, in such a way that a tour is optimal if and only if it obeys this global traversal without shortcutting in non-authorized ways. In particular, the local configuration of C_i above or below element gadgets makes it impossible for C_i to avoid the visit of $H_{i,j}$ unless $j \in F_i$ (or unless j has already been covered by another subset, i.e. $H_{i-1,j}$ is not yet visited).

Setting the visiting speed $\nu = 0$ is crucial for controlling (almost canceling) the impact of acceleration, so as to force the optimal trajectory to follow the same pattern as in Papadimitriou's proof. Admittedly, the spirit of the VTSP problem is undermined by such a proof, which remains unsatisfatory and motivates Open question 7 in Chapter 6. The details of our adaptation specify the corresponding intra-gadget spacing between cities and the spacing between the gadgets. Most of the consecutive cities in the tour are actually separated by only one or two space units, which cancels out the benefits of accelerating. The few exceptions are between subset gadgets and the adjacent element gadgets, where the speed can get arbitrarily large depending on the distance chosen. We choose a distance close to the original distance of 20 units, resulting in a maximum speed of 5 space units. The proportions in the spacing imply that this has no impact on the visit order *w.r.t.* Papadimitriou's tour.

4.3.2.2 Technical aspects

This section describes how to reduce an ExactCover instance to a VTSP instance with visit speed $\nu = 0$. For simplicity, it is first formulated in the 5-successor model, *i.e.,* the speed can change only in one dimension at a time (Theorem 9). This constraint is subsequently relaxed to the default settings, including the 9-successor racetrack model, through a geometrical trick, resulting in Theorem 10.

The following definitions are from Papadimitriou [77]. A subset P' of the set of cities is an *a-component* if for all $p \in P'$ we have $\texttt{min}(\texttt{cost}(p, p') : p' \notin P') \geq a$ and $\texttt{max}(\texttt{cost}(p, p') : p' \in P') < a$, and P' is maximal *w.r.t.* these properties. A *k-trajectory* for a set of cities is a set of k, not closed trajectories visiting all cities. A valid trajectory for a VTSP instance is thus a closed (or cyclic) 1-trajectory. A subset of cities is *a-compact* if, for all positive integers k, an optimal k-trajectory has cost less than the cost of an optimal $(k+1)$-trajectory plus a. Note that a-components are trivially a-compact.

Lemma 13 (Papadimitriou [77])**.** *Suppose we have N a-components $P_1, ..., P_N \in P$, such that the cost to connect any two components through a trajectory is at least $2a$, and P_0, the remaining part of P, is a-compact. Suppose that any optimal 1-trajectory of this instance does not contain any vectors between any two a-components. Let $K_1, ..., K_N$ be the costs of the optimal 1-trajectories of $P_1, ..., P_N$ and K_0 the cost of the optimal $(N+1)$-trajectory of P_0. If there is a 1-trajectory T of P consisting of the union of an optimal $(N+1)$-trajectory of P_0, N optimal 1-trajectories of $P_1, ..., P_N$ and $2N$ trajectories of cost a connecting a-components to P_0, then T is optimal. If no such 1-trajectory exists, the optimal 1-trajectory of P has a cost greater than $K = K_0 + K_1 + ... + K_N + 2Na$.*

Consider the *1-chain* structure presented in Figure 4.7. This structure is composed of cities positioned on a line, at distance one from one another. 1-chains can bend at 90 degrees angles, and only one optimal 1-trajectory exists, with a cost of $2(n-1)$ vectors for a 1-chain of length n.

Next, consider the structure in Figure 4.8, referred to as a *2-chain*. The distance between the leftmost (or rightmost) city and its nearby cities is $\sqrt{2}$. The closest distance between other cities is 2. The important thing to notice here is there exists only two distinct optimal 1-trajectories, denoted as mode 1 and mode 2, both of a cost of $3n + 11$ for a 2-chain of length n.

Observation 3. *Among all 1-trajectories for H (see Figure 4.9) having as endpoints two of the cities $A, A', B, B', C, C', D, D'$, there are 4 optimal 1-trajectories, namely those with endpoints (A, A'), (B, B'), (C, C'), (D, D'), which all have a cost of 77 vectors.*

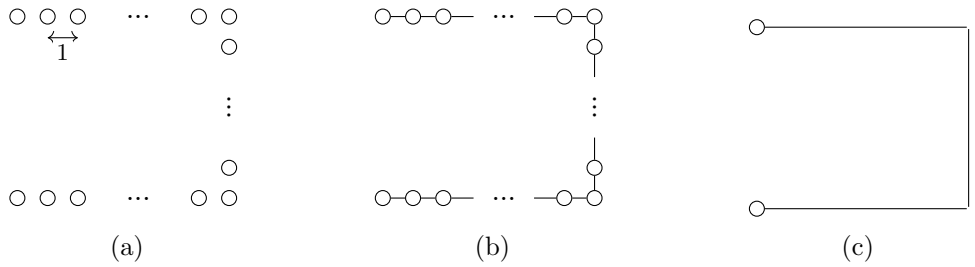


Figure 4.7: 1-chain structure which turns 90° twice. The distance between consecutive cities is 1. The optimal visit order is shown in (b). We abbreaviate a 1-chain schematically as shown in (c).

Figure 4.8: 2-chain structure (a). A 2-chain has precisely two optimal 1-trajectories, (b) and (c). We abbreviate a 2-chain schematically as shown in (d).

We are now ready to prove Theorem 9 using the above definitions and gadgets.

Theorem 9. ExactCover *reduces in polynomial time to* VectorTSP *with default settings, with maximum visiting speed* $\nu = 0$ *and the* 5*-successor model.*

Proof. The aforementioned structures are combined to construct a VTSP instance from a given Exact Cover instance. Construct the structure shown in Figure 4.10, where n is the number of subsets given in the corresponding Exact Cover instance, and m the number of elements in the universe.

The 2-chains represent the subsets in Exact Cover, and H structures indirectly represent the elements in the universe. Finally, for every 2-chain C_i, replace the cities positioned directly above or below an H, by one of two structures, depending on the elements in C_i's corresponding subset. If the subset contains the element corresponding to the above (or below) H, then replace by structure A (see Figure 4.11), otherwise

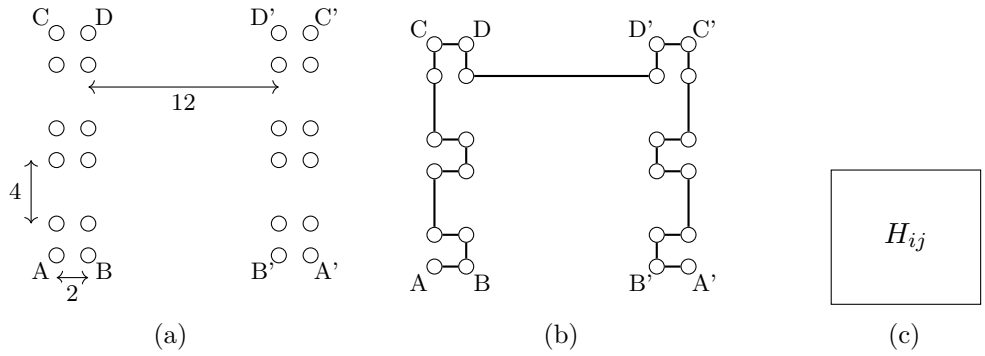


Figure 4.9: Structure H. The distance between A and B is 2, between A and C 14, and between A and A' 16. An optimal 1-trajectory in H is shown in (b). We abbreviate an H structure schematically as shown in (c).

replace by structure B (see Figure 4.12). The idea is to make it costly to visit an H above or below from a structure A traversed in mode 1.

We observe that now the optimal cost to connect two k-paths between some 2-chain C_i and some H_{ij} (or $H_{(i-1)j}$) is 10 vectors, whereas the optimal cost to connect any two k-paths between two H_{ij}, is at least 40 vectors. Also, this optimal cost of 10 vectors between some 2-chain C_i and some H_{ij}, can only be attained by a trajectory on a straight vertical line, thanks to the precise distance of 25. Deviating even the slightest bit from the vertical line would result in a non-optimal cost. The construction of the VTSP instance is now complete. It should be clear that an optimal 1-trajectory must have Q and R as endpoints. This construction meets the hypotheses of Lemma 13 with $a = 10$, $N = m(n-1)$, $K_1 = ... = K_N = 77$ and $K_0 = 1257mn + 4m + 557n + 24p + 1464$, where p is the sum of cardinalities of all given subsets of the Exact Cover instance.
We examine when this structure has an optimal 1-trajectory T, as described in the lemma. T traverses all 1-chains in the obvious way, and all 2-chains in one of the two traversals. Since its portion on P_0 has to be optimal, T must visit a component H from any configuration B encountered, and it must return (by Observation 3) to the symmetric city of B, since its portion on H must be optimal, too. If T encounters a configuration A and the corresponding chain is traversed in traversal 2, T will also visit a component H. However, if the corresponding chain is traversed in traversal 1, T will traverse A without visiting any configuration H, since all trajectories connecting P_0 and H components must be of cost a. Moreover this must happen exactly once for each column of the structure, since there are $n-1$ copies of H and n structures A or B in each column. Hence, if we

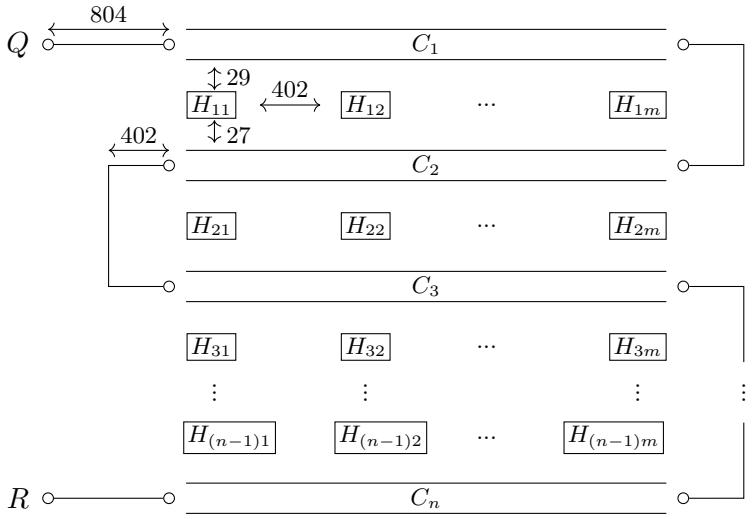


Figure 4.10: Construction of the VTSP instance.

consider the fact that C_j is traversed in traversal 1 (*resp.* traversal 2) to mean that the corresponding subset is (*resp.* is not) contained in the Exact Cover solution, we see that the existence of a 1-trajectory T, as described in Lemma 13, implies the Exact Cover instance admits a solution. Conversely, if the Exact Cover instance admits a solution, we assign, as above, traversals to the chains according to whether or not the corresponding subset is included in the solution. It is then possible to exhibit a 1-trajectory T meeting the requirements of Lemma 13. Hence the structure at hand has a 1-trajectory of cost no more than $K = 1354mn - 93m + 557n + 24p + 1464$ if and only if the given instance of Exact Cover is solvable. Finally, to obtain a valid VTSP trajectory, connect both endpoints Q and R in Figure 4.10 with a 1-chain, and increase K accordingly. □

Theorem 10. ExactCover *reduces in polynomial time to* VectorTSP *with default settings, except for maximum visiting speed* $\nu = 0$.

Proof. The proof for the 9-successor model is the same as for the 5-successor model, except that the whole created VTSP instance $\mathcal{I}'$ is tilted by 45° (the direction does not matter), and distances are scaled by $\sqrt{2}$. The value of K is unchanged. This modification transposes the limitations of the 5-successor model to the 9-successor model. Indeed, due to the careful choice of distances involved, if one wishes to remain optimal while visiting the cities, one needs to only consider the outermost accelerations (diagonals) of

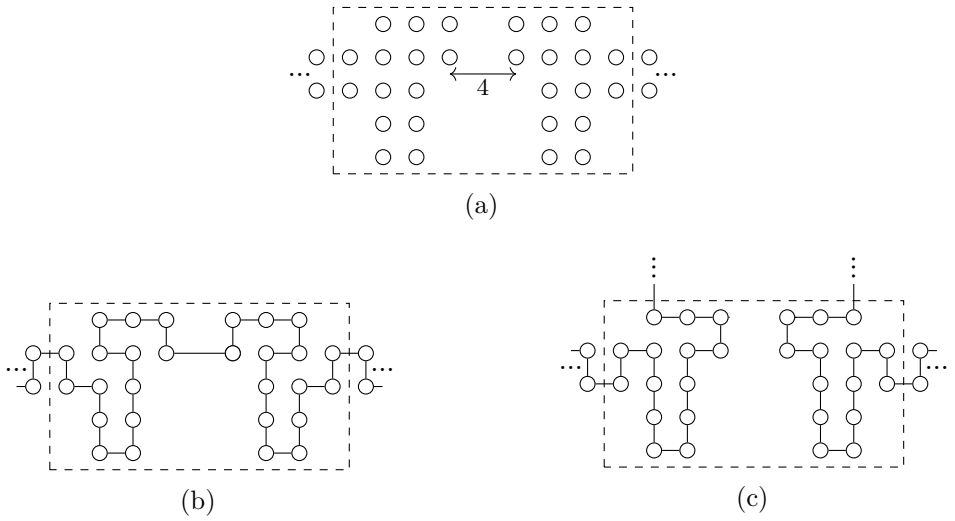


Figure 4.11: Structure A (see (a)). Visiting structure A in mode 1 makes it costly to visit an H structure above or below (see (b)). Visiting structure A in mode 2 however, makes it less costly to visit an H structure above (see (c)) or below.

the 9-successor version, as well as the null speed before turning (since different diagonals in the 9-successor model cannot directly succeed one another). □

Note that a similar geometrical trick might be used to adapt the proof to further settings, such as continuous space with the continuous d-sphere successor function, such as depicted in Figure 4.1 (for $\mathbb{R}^2$). Also, intuitively, one may simply remove the null vectors, so as to consider a maximum visiting speed $\nu = 1$.

4.3.2.3 Transformation from VectorTSP to GroupTSP

Here, we show that VectorTSP admits a natural polynomial-time reduction to the so-called GroupTSP (also known as SetTSP or GeneralizedTSP), where the input is a set of cities partitioned into groups, and the goal is to visit at least one city in each group.

Lemma 14. *VectorTSP admits a natural polynomial-time reduction to GroupTSP.*

Proof. Let $\mathcal{I}$ be the original VTSP instance and n the number of cities in $\mathcal{I}$. Each city in $\mathcal{I}$ can be visited in a number of different ways, each corresponding to a different configuration in $\mathcal{C}$ (the set of all possible configurations). The strategy is to create a city in $\mathcal{I}$' for each configuration that visits at least one city in $\mathcal{I}$, and group them according

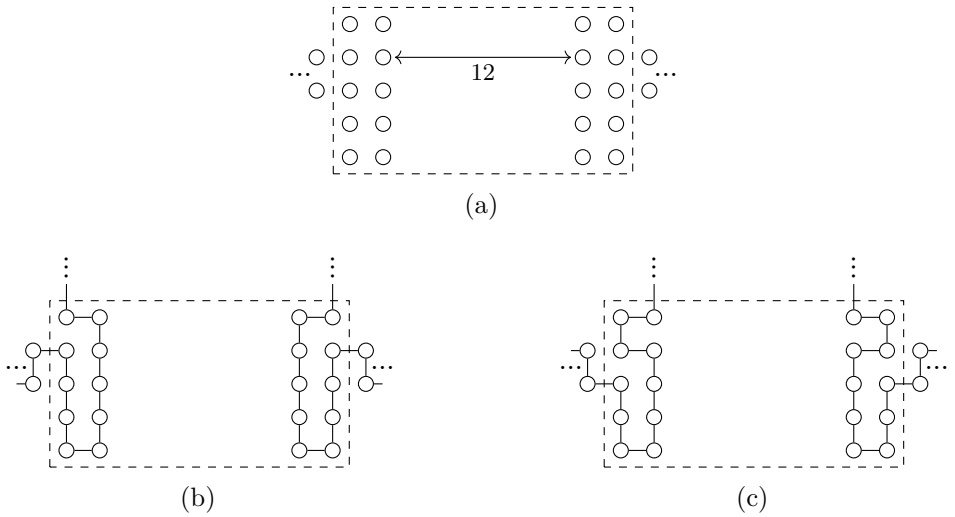


Figure 4.12: Structure B (see (a)). Visiting structure B in any mode makes it advantageous to visit an H structure above or below (see (b) and c).

to which city of $\mathcal{I}$ they visit (the other configurations are discarded). Thus, visiting a city in each group of $\mathcal{I}$' corresponds to visiting all cities in $\mathcal{I}$. Depending on the parameters of the model (visit speed, visit distance, vector completion), it may happen that a same configuration visits several cities in $\mathcal{I}$, which implies that the groups may overlap; however, Noon and Bean show in [73] that a GTSP instance with overlapping groups can be transformed into one with mutually exclusive groups at the cost of creating k copies of a city when it appears originally in k different groups. Thus we proceed without worrying about overlaps. Let X be the set of cities in $\mathcal{I}$, and $\mathcal{C}(x) \subseteq \mathcal{C}$ be the configurations which visit city $x \in X$. Instance $\mathcal{I}$' is defined by creating a city for each configuration in $\cup_{x \in X} \mathcal{C}(x)$ and a group for each $\mathcal{C}(x)$. An arc is added between all couples (c_1, c_2) of cities in $\mathcal{I}$' such that c_1 and c_2 belong to different groups; the weight of this arc is the distance between c_1 and c_2 in the configuration graph. Thus, a trajectory using k configurations to visit all the cities in $\mathcal{I}$ corresponds to a tour of cost k visiting at least one city in each group in $\mathcal{I}$'. The fact that the reduction is polynomial (both in time and space) results from the facts that (1) there is a polynomial number of relevant configurations (Lemma 12), each one being copied at most n times; and (2) the distance between two configurations in the configuration graph can be computed in polynomial time (Observation 2). □

Note that this reduction is general in terms of the parameters: any combination ν, α, and β only impacts the set of vectors that visit each city. Lemma 14 implies the following corollary.

Corollary 2 (Through [73])**.** *VectorTSP admits a natural polynomial-time reduction to AsymmetricTSP.*

Indeed, Noon and Bean [73] present a polynomial time reduction from GTSP to ATSP. This is done by adding arcs of cost zero inside every group, creating a cycle visiting all nodes in the group. Suppose *w.l.o.g.* the created cycle $x_1, x_2, ..., x_k, x_1$, then for all arcs (x_i, y_j), with y_j some node in another group, replace by arcs (x_{i-1}, y_j). Outgoing arcs of node x_1 get replaced on node x_k. This effectively shifts all outgoing arcs in each group, such that an optimal tour is forced to visit all nodes in a group before visiting another. In turn, we have the following corollary.

Corollary 3 (Through [58])**.** *VectorTSP admits a natural polynomial-time reduction to SymmetricTSP.*

Kanellakis and Papadimitriou proved any asymmetric TSP instance such as we can obtain in Corollary 2, can be transformed into a symmetric TSP instance. The main idea is to simulate an ATSP as follows. Add three nodes i_1, i_2 and i_3 for every node i in the ATSP. Add (undirected) edges (i_1, i_2) and (i_2, i_3) of cost 0 to i_2. Add (undirected) edges (i_3, j_1) if $(i, j) \in E$ of the ATSP, with the same cost. This transformation triples the amount of nodes of the instance. This was later improved to only double the amount of nodes, first by Jonker and Volgenant [57] and later by Kumar and Li [63].

4.4 Algorithms

In this section, we present an algorithmic framework for finding acceptable solutions to VTSP in practical polynomial time. It is based on an interaction between a high-level part that decides the visit order (tour), and a trajectory oracle that evaluates its cost.

4.4.1 Exploring visit orders (`FlipVTSP`)

A classical heuristic for ETSP is the so-called `2-opt` algorithm [33], also known as `Flip`. It is a local search algorithm which starts with an arbitrary tour π. In each step, all the possible 2-permutations (*i.e.,* swaps of two cities, or simply flips) of the current tour π are generated. If such a flip π' improves upon π, it is selected and the algorithm recurses on

π'. Eventually, the algorithm finds a local optimum whose quality is commonly admitted to be of reasonable quality, albeit without guarantees (the name `2-opt` does not reflects an approximation ratio, it stands for 2-permutation local optimality). Adapting this algorithm seems like a natural option for the high-level part of our framework.

The main differences between our algorithm, called `FlipVTSP`, and its ETSP analogue are that (1) the cost of a tour is not evaluated in terms of distance, but in terms of the required number of racetrack configurations (through calls to the oracle); (2) the tours involving self-crosses are not discarded (see Fact 9); and (3) the number of recursions is polynomially bounded because new tours are considered only in case of improvement, and the length of a trajectory is itself polynomially bounded (Lemma 11). The resulting tour is a local optimum with respect to 2-permutations, also known as a 2-optimal tour. For completeness, the algorithm is given by Algorithm 5.

Algorithm 5 : `2-opt`.

Input: a set P of cities.
Output: a 2-optimal tour w.r.t. the racetrack model.

```
1:  π_opt ← init(P)
2:  C_opt ← oracle(π_opt)                          ▷ Without limited view (optimal)
3:  improved ← true
4:  while improved do
5:      improved ← false
6:      for each city i (except starting city) do
7:          for each other city j (except starting city) do
8:              π_test ← flip(π_opt, i, j)
9:              C_test ← oracle(π_test)             ▷ With limited view (faster)
10:             if C_test < C_opt then
11:                 π_opt ← π_test
12:                 C_opt ← C_test
13:                 improved ← true
14:                 break
15:         if improved then
16:             break
17: return π_opt
```

Theorem 11. *One can find a 2-optimal tour for VTSP in time $O(n^2 L^d \tau(n, L))$, where n is the number of cities, L the largest distance between cities in a dimension, d the number of dimensions, and $\tau(n, L)$ the running time complexity of the oracle for computing the cost of an optimal racetrack trajectory visiting the n cities.*

Proof. As explained in (the proof of) Lemma 11, if the visit order is not imposed, then one can easily find a trajectory of length $O(L^d)$ that visits all the cities, through walking over the entire area (rectangle hull containing the cities). Let π be the order in which the cities are visited by such a walk, shifted circularly so as to set the starting city to the desired one. This tour is the one returned by the `init()` function. Then C_{opt} is accordingly initialized with cost $O(L^d)$ in line 2. The factor L^d in the complexity formula then follows from the fact that the main loop iterates only if a shorter trajectory is found, which can occur at most as many times as the length of the initial trajectory. Then, in each iteration, up to $O(n^2)$ flips are generated (at constant time), with a nested call to the oracle. All the other operations take constant time under the standard arithmetic abstractions. □

4.4.2 Optimal racetrack given a fixed visit order (`Multipoint A*`)

Here, we discuss the problem of computing an optimal racetrack trajectory that visits a set of points *in a given order*. A previous work of interest is Bekos *et al.* [12], which addresses the problem of computing an optimal racetrack trajectory in a so-called "Indianapolis" track, where the track has a certain width and right-angle turns. This particular setting limits the maximum speed at the turns, which makes it possible to decompose the computation in a dynamic programming fashion. In contrast, the space is open in VTSP, with no simple way to bound the maximum speed. Therefore, we propose a different strategy based on searching for an optimal path in the configuration graph using A*.

The problem: Given an ordered sequence of points $\pi = (p_1, p_2, \ldots, p_n)$, compute (the cost of) an optimal trajectory realizing π, *i.e.,* visiting the points in order, starting at p_1 and ending at p_n at zero speeds. (In the particular case of VTSP, p_1 and p_n coincide.)

Finding the optimal trajectory between *two* configurations already suggests the use of path-finding algorithms like BFS, Dijkstra, or A* (see e.g. [83] and [12]). The difficulty in our case is to force the path to visit all the intermediary points in order, despite the fact that the space is open. Our contribution here is to design a cost function that guides A* through these constraints. In general, A* explores the search space by generating a set of successors of the current "position" (in our case, configuration) and estimate the cost of each successor using a problem-specific function. The successors are then inserted into a data structure (in general, a priority queue) which makes it easy to continue exploration from the position which is globally the best estimated. The great feature of

A* is that it is guaranteed to find an optimal path, provided that the cost function does not over-estimate the actual cost, and so, is as fast as the estimation is precise.

4.4.2.1 Cost estimation

For simplicity, we first present how the estimation works relative to the entire tour. Then we explain how to generalize it for estimating an arbitrary intermediate configuration in the trajectory (i.e. one that has already visited a certain number of cities and is located at a given position with given velocity). The key insight is that the optimal trajectory, whatever it be, must obey some pattern in each dimension. Consider, for example, the tour $\pi = \{(5, 10), (10, 12), (14, 7), (8, 1),$
$(3, 5), (5, 10)\}$ shown on Figure 4.13. In the x-dimension, the vehicle must move at least from 1 to 3, then stop at a *turning point*, change direction, and travel towards 5, then stop and change direction again, and travel back to 1. Thus, any trajectory realizing π can be divided into *at least* three subtrajectories in the x-dimension, whose cost is *at least* the cost of traveling along these segments, starting and ending at speed 0 at the turning points. Thus, in the above example, the vehicle must travel at least along distances 9, 11, and 2 (with zero speed at the endpoints), which gives a cost of at least 16 (*i.e.,* 6, 7, and 3, respectively). The same analysis can be performed in each dimension; then, the actual cost must be *at least the maximum* value among these costs, which is therefore the value we consider as estimation.

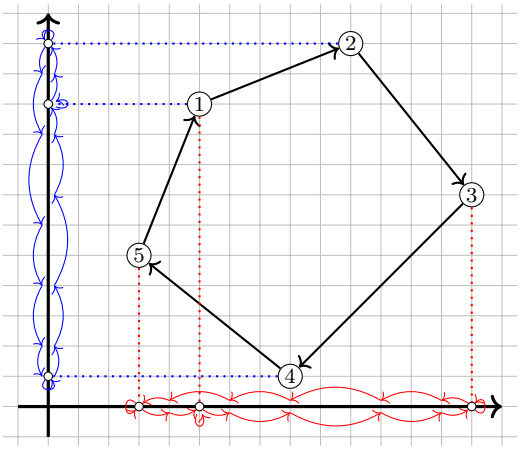


Figure 4.13: Projection in each dimension.

In general, the configurations whose estimation is required by A* are more general than the above case. In particular, it has an arbitrary position and velocity, and the vehicle may have already visited a number of cities. Therefore, the number of visited

cities is stored along a configuration, and the dimensional cost is evaluated against the remaining sub-tour. The only technical difference is that one must carefully take into account the current position and velocity when determining where the next turning point is in the dimensional projection, which however poses no significant difficulty. Concretely, a case-based study of the initial configuration with respect to the first turning point, allows one to self-reduce the estimation to the particular case that the initial speed is zero (possibly at a different starting position). Consequently, the total cost amounts to a sum of costs between consecutive pairs of turning points with zero speed at these points.

Lemma 15. *The cost estimation of a subtour $\pi' = c, p_i, ..., p_n$, where c is the current configuration and $p_i, \ldots, p_n$ is a suffix of π can be computed in $O(n)$ time.*

Proof. As explained, the subtour is first reduced to a subtour $\pi'' = p_{i-1}, p_i, \ldots, p_n$. The turning points in π'' are easily identified through a pass over π''. Their number is at most n because they are a subset of the points in π''. Finally, the cost between each pair of selected turning points can be computed in constant time [12] (if one neglects the encoding size of an integer representing a coordinate). □

The reader is referred to [12] for more on computing the cost between two configurations in one dimension. Let us now discuss the running time complexity of the resulting algorithm. In general, A* can have an exponential running time in the solution depth (thus, length of the trajectory). It is however possible, in our case, to make it polynomial.

Theorem 12. *The A* oracle runs in polynomial time, more precisely in time $\widetilde{O}(L^{(d^2)}n^2)$.*

Proof. A "configuration" of the A* algorithm (let us call it a state, to avoid ambiguity) is made of a racetrack configuration c together with a number k of visited cities. There are at most $O(L^{(d^2)})$ configurations (Lemma 12) and n cities, thus A* will perform at most $O(L^{(d^2)}n)$ iterations, provided that it does not explore a state twice. Given that the states are easily orderable, the later condition can be enforced by storing all the visited states in an ordered collection that is searchable and insertable in logarithmic time (whence the $\widetilde{O}$ notation). Finally, each state is estimated in $O(n)$ time (Lemma 15). □

The combined use of `FlipVTSP` and `Multipoint A*` thus runs in polynomial time (Theorem 11 and Theorem 12). We follow with the details of computing a cost in 1 dimension, and afterwards we present a way to make the oracle algorithm even faster if one is willing to trade optimality for performance.

4.4.2.2 Uni-dimensional cost estimation

We present here how to efficiently compute an exact cost between projections in one dimension x and x'.

Lemma 16. *The exact cost from x to x', with initial speed $dx = 0$, can be computed in constant time.*

Proof. Suppose *w.l.o.g.* $x \leq x'$. One basically has some distance $d = x' - x$ to cross, starting and stopping at zero speed. Consider only vectors that accelerate in the first half of the segment of length d, and only vectors that decelerate in the second half, such as shown in Figure 4.14. If the corresponding trajectories line up perfectly in the middle of the segment, the combined trajectories is trivially an optimal trajectory *w.r.t.* the number of vectors used. Note this would have been the case for Figure 4.14 if distance d was 12, instead of 13. If this does not line up perfectly however, consider the distance d' separating the last vectors before attaining the middle of the segment (so in Figure 4.14, $d' = 1$). If it is possible to cross this distance with one vector that isn't too small *w.r.t.* adjacent vectors, then doing so will trivially result in an optimal trajectory. If the vector is too small however, then one can insert it somewhere else in the trajectory (see the green vector in Figure4.14). The resulting trajectory is again trivially optimal. If the distance d' is too large to cross with only 1 vector, respecting adjacent vectors' lengths, then 2 vectors should suffice, and can again be inserted in the trajectory. Indeed, through a proof by contradiction, at most 2 vectors should be necessary to cross distance d', or else one could have added one more accelerating vector to the left trajectory, and one more decelerating vector to the right trajectory. Return the total cost, without forgetting the last "self-looping" vector, needed to stop at zero speed (in Figure 4.14, this last self-looping vector is shown in green as well).

This can be done in time $O(\sqrt{n})$ with a simple while loop, although one can do better (in constant time), through Algorithm 6. □

We found that the computation of this cost with $dx = 0$, *w.r.t.* distance d, seems to correspond with the integer sequence A027434, found on the On-line Encyclopedia of Integer Sequences [67]. An elegant closed form of this sequence is given as $\lceil 2\sqrt{d} \rceil$.

Corollary 4. *The exact cost going from x to x', with initial speed dx, can be computed in constant time.*

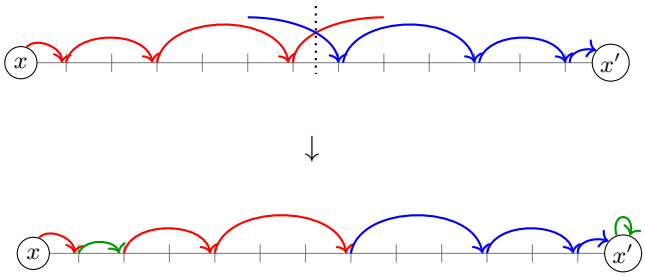


Figure 4.14: Example on $d = x' - x = 13$ on how to compute a partial cost. Vectors are presented as arcs for visibility.

Algorithm 6 : `one-dimensional cost computation`

Input: a distance $d = x' - x$
Output: an optimal cost for traversing d, starting and stopping at zero speed.

```
if d == 0 then
    return 0
r ← ⌊(√(4d+1) − 1)/2⌋
d' ← d − r(r + 1)
if d' == 0 then
    return 2r + 1
if d' <= r + 1 then
    return 2r + 2
return 2r + 3
```

Proof. Considering different speeds doesn't add much difficulty. Indeed, all cases can be reduced to a case with speed $dx = 0$ (treated in Lemma 16). If $dx < 0$, slow down to zero speed, add the necessary amount of vectors to do so, $-dx$, to the cost, and compute the rest of the cost from this new $x = x - \frac{dx(dx-1)}{2}$ position with speed $dx = 0$. If $dx > 0$ and very large (large enough to bypass x', even when continuously decelerating), then also slow down to zero speed. Lastly, if $dx > 0$, but not so large as to bypass x' by only decelerating, add $-dx$ to the cost, and compute the rest of the cost with new $x = x - \frac{dx(dx+1)}{2}$ position and $dx = 0$. □

4.4.2.3 A faster heuristic using limited views.

Our presented A* algorithm always finds the optimum, but in practice, it only scales up to medium-sized instances (see Section 4.5). If one is willing to lose some precision, then a simple trick (also used in the indianapolis case [12]) can be used to scale linearly with the number of cities. The idea is to compute limited sequential sections of the trajectory and

glue them together subsequently. Concretely, given a tour $\pi = p_1, ..., p_n$, the limited view heuristic runs A* on a sliding window of fixed length l (typically 5 or 6) over π. For each offset i of the window, the trajectory is computed from p_i to p_{i+l} (p_n, if less than l cities remain). Then, of the computed trajectory, only the subtrajectory T_i from p_i to p_{i+1} is retained, the offset advances to $i + 1$ and A* is run again, using the last configuration of T_i as initial configuration. Finally, the algorithm returns the concatenation of the T_is.

4.5 Experiments

In this section, we present a few experiments with the goal to (1) validate the algorithmic framework described in Section 4.4, and (2) motivate the VTSP problem itself, by quantifying the discrepancy between ETSP and VTSP. The instances were generated by distributing cities uniformly at random within a given square area. For each instance, `Concorde` [6] was used to obtain the reference optimal ETSP tour π. The optimal trajectory T realizing this tour was computed using `Multipoint A*` (with complete view). Then, `FlipVTSP` explored the possible flips (with limited view) until a local optimum is found. An example is shown on Figure 4.15 (right), resulting from 2 flips on an optimal ETSP tour (left). Finding these flips is left as an exercise.

(a) Optimal realization of an optimal ETSP tour (128 vectors)

(b) Local optima in `FlipVTSP` (120 vectors)

Figure 4.15: Example of tour improvement.

Such an outcome is not rare. Figures 4.16, 4.17 and 4.18 show some measures when varying (1) the number of cities in a fixed area; (2) the size of the area for a fixed number of cities; and (3) both at constant density. The plots show the likelihood of at least one improving flip happening, as well as the average number of flips, tested on 100 random instances. For performance, only the flips which did not deteriorate the EuclideanTSP tour distance by too much were considered (15 %, empirically). Thus, the plots tend

to *under-estimate* the impact of VTSP (they already do so, by considering only *local* optima, and *limited view* in the flip phase).

The results suggest that an optimal ETSP tour becomes less likely to be optimal for VTSP as the number of cities increases (in a fixed area). The size of the area for a fixed number of cities (here, 10) does not seem to have a significant impact. Somewhat logically, scaling both parameters simultaneously (at constant density) seems to favor VTSP as well. Further experiments should be performed for a finer understanding. However, these results are sufficient to confirm that VTSP is a specific problem.

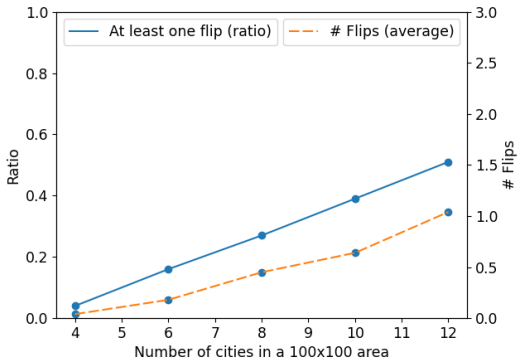


Figure 4.16: Varying the number of cities within a fixed 100×100 area.

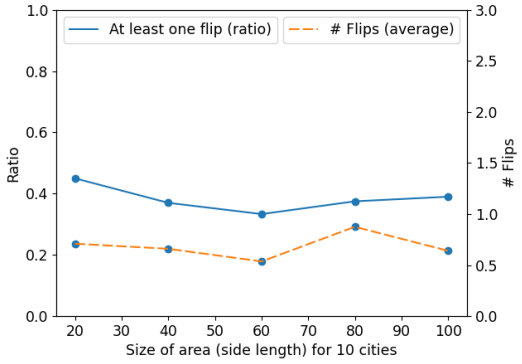


Figure 4.17: Varying the size of the area, considering 10 cities.

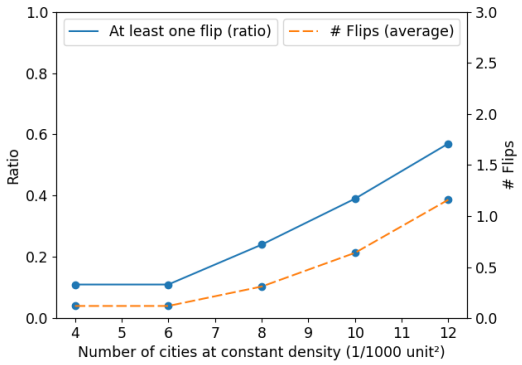


Figure 4.18: Varying the number of cities and the size of the area, but keeping constant density of 1 city per 1000 unit^2.

4.6 Conclusion

We explored a new version of the Traveling Salesman Problem in which we added mobility constraints so as to represent a highly mobile entity needing to visit a set of locations. We call this problem VectorTSP. The constraints are taken from a pencil-and-paper game named Racetrack, which aptly represents acceleration, speed and inertia forces of the visiting vehicle through a set of simple and discrete rules. We gave some insights into this problem, some of which distinguish it from the EuclideanTSP. VectorTSP is shown NP-hard through a reduction from Exact Set Cover, and reversely, we gave a natural reduction from VectorTSP to multiple other TSP. In terms of algorithms, we adapted a simple heuristic known as 2-opt, which uses an oracle. We proposed as an implementation for this oracle, a multi-point A* algorithm for which we give an admissible cost estimation function which computes exact uni-dimensional costs. Finally, experiments seem to imply that VectorTSP becomes more and more distinguished from EuclideanTSP as the number of cities grows larger, although more experiments should be performed for a better understanding. We hope that the results from this work will motivate future investigations on this problem.

Open questions and future work are discussed in Chapter 6.

4.6.1 Transition between chapters

In the next chapter, we remove the visiting of a given set of cities, and instead of considering one mobile entity, we consider multiple entities which together need to induce

some desired temporal graph property on the induced temporal communications graph. The next chapter is thus where the two domains of temporal graph theory from Chapter 3, and motion planning from this chapter, interact strongly in the form of a mobile ad hoc network or MANET.

Chapter 5

Temporal properties induced by collective mobility

I spend almost as much time figuring out what's wrong with my computer as I do actually using it.

— Clifford Stoll

Contents

The content of this chapter is published as a software package, available online on my personal page[1], as well as on GitHub[2].

[1] `https://www.labri.fr/perso/jschoete/index.php?id=mobility-models`

[2] `https://github.com/jschoete/mobilitymodels`

In this chapter, we are interested in proposing a software package allowing the generation of temporal graphs containing (or excluding) certain temporal properties, induced through the collective mobility of a swarm of mobile agents (also referred to as a MANET). More precisely, we provide a number of mobility models that control how the agents move, each model inducing a particular set of properties. This package relies on JBotSim [23], a Java simulation library for distributed algorithms in dynamic networks. We first present the motivation surrounding this subject, before presenting our proposed mobility models. We finish with some examples on how a user may wish to use this package.

5.1 Motivation

We first define all temporal graph properties, and give some context surrounding each property. In particular, some distributed algorithms rely heavily on the presence of certain properties. We'll then explain how the collective mobility of a MANET induces an **interaction graph** (also called **communications graph** or simply graph when the context is clear), which, if the mobility is significant, is in fact a temporal graph. Controlling this mobility to ensure certain temporal properties in turn creates, among other uses, testing environments for distributed algorithms.

5.1.1 Properties and context

In this subsection, we review in detail some of the main temporal properties (or classes) identified in temporal graphs (some of these are already presented in Section 2.2.2). In [21], a dozen temporal properties were identified that have been effectively exploited in the distributed computing and networking literature. These were extended more recently in [18], and renamed using mnemonic symbols (see Figure 5.1). These in turn define several classes of temporal graphs in which these properties are satisfied.

We focus on recurrent temporal properties because distributed algorithms typically execute over networks of infinite lifetime. Going from the more general properties, and working our way up to the more precise classes, we have:

- $\mathcal{TC}^{\mathcal{R}}$: Every vertex can reach all others starting at any time (although it might take a while to do so). Arguably the most basic recurrent property, it allows for previously mentioned problems to be solved at any time in such a temporal graph. This property is often assumed to be present in mobile ad hoc networks, as it

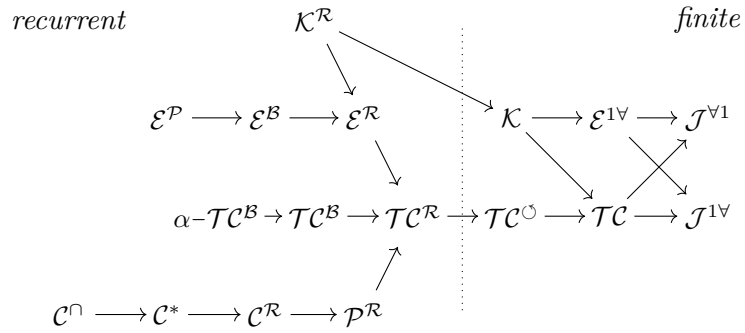


Figure 5.1: Hierarchy surrounding temporal connectivity (figure from [18]). (Same Figure as used in Chapter 2.)

captures the basic ability for the nodes to influence each other recurrently. Note that the footprint of a graph inducing property $\mathcal{TC}^{\mathcal{R}}$ is connected, and since all the following properties are included in the class $\mathcal{TC}^{\mathcal{R}}$, all of these classes' graphs will have connected footprints as well.

- $\mathcal{TC}^{\mathcal{B}}$: Every vertex can reach all others in a bounded period of time, infinitely often. This class corresponds in essence to classical assumptions made in static distributed computing, in particular when the communication is asynchronous.
- α-$\mathcal{TC}^{\mathcal{B}}$: Every vertex can reach all others, by a journey which stops at most α time steps at each intermediate vertex. This property manifests with high probability in a wide range of edge-markovian temporal graphs, and it's a sufficient condition to stop re-transmission after some time in a parsimonious broadcast.
- $\mathcal{K}^{\mathcal{R}}$: All vertices are connected by some time edge to all other vertices, infinitely often. $\mathcal{K}^{\mathcal{R}}$ may be used to represent population protocols' communication schedules.
- $\mathcal{E}^{\mathcal{R}}$: Appearing edges reappear infinitely often. This class has strong connections with classes $\mathcal{E}^{\mathcal{B}}$ and $\mathcal{E}^{\mathcal{P}}$ and as a consequence results on these classes are often related. In [1], Aaron *et al.* show that the dynamic map visitation problem is inapproximable in $\mathcal{E}^{\mathcal{R}}$, approximable in $\mathcal{E}^{\mathcal{B}}$, and tractable in $\mathcal{E}^{\mathcal{P}}$. Casteigts *et al.* also present several relations of foremost, shortest and fastest broadcasts with these classes in [20].
- $\mathcal{E}^{\mathcal{B}}$: Appearing edges reappear infinitely often, each in a bounded time period.
- $\mathcal{E}^{\mathcal{P}}$: Appearing edges reappear infinitely often, in a periodic time period.

- $\mathcal{P}^{\mathcal{R}}$: For all pairs of vertices, there will always be a future snapshot in which they are connected by a path. In [80], Ramathan *et al.* show some (un)feasability results related to $\mathcal{TC}^{\mathcal{R}}$, $\mathcal{P}^{\mathcal{R}}$, and $\mathcal{C}^{\mathcal{R}}$, depending on the knowledge of the temporal graph the vertices have.
- $\mathcal{C}^{\mathcal{R}}$: There will always be a future snapshot which is connected (in the static sense).
- $\mathcal{C}^{*}$: Every snapshot is connected. This is a sufficient condition for information dissemination from any vertex to all, in an optimal amount of rounds.
- $\mathcal{C}^{\cap}$: For all time windows of some given length T, there exists a persistent connected subgraph, *i.e.* the graph is connected through some subgraph of edges which do not disappear during this time window. This property creates some stability, which allows for the speeding up of many distributed algorithms, some by a factor of up to T, as shown by Kuhn *et al.* in [62].

Some other classes which are of interest include the classes studied by Gomez-Calzado *et al.* in [48] where some latency function was considered on the edges of the graph. Also, recently Altisen *et al.* in [4] introduced the class $\mathcal{TC}^{\mathcal{Q}}$ which is positioned somewhere between $\mathcal{TC}^{\mathcal{R}}$ and $\mathcal{TC}^{\mathcal{B}}$, and which proved to be useful for characterizing some conditions on their self-stabilizing algorithms. Classes of temporal graphs may also be defined by restricting the footprint to belong to a particular class of (standard) graphs, which was done in [90].

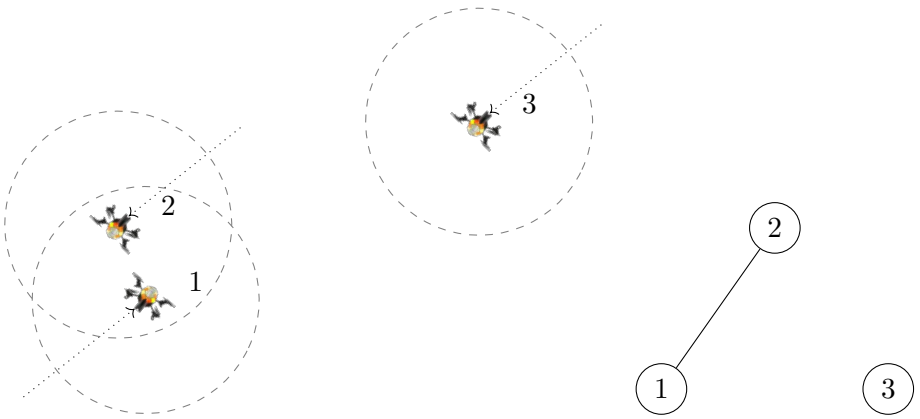
In the above list of classes, one might differentiate classes considering strict journeys and non-strict journeys respectively, depending on the context. Concerning our proposed mobility models, the two are practically the same, so we will not differentiate between these classes.

Remark 5. *Two inclusion relations are missing from Figure 5.1. We leave this as an exercice for the reader.*

5.1.2 Properties through mobility

In the setting of Mobile Ad Hoc Networks (MANET or simply swarms), mobility of the swarm's entities play a significant role in the structure of the communications graph, since these simple entities often have a short communication range (see Figure 5.2).

As shown, the inclusion of an agent a inside of another agent b's communication range, makes it so agent b can send messages to agent a. In this document, the communication



(a) Entities 1 and 2 pass each other and are able to communicate. Entity 3 is not in range to communicate.

(b) Corresponding communication graph. Vertices 1 and 2 are connected, while 3 is isolated.

Figure 5.2: Some mobile entities with limited communication range, with the graph of communication.

ranges for agents are considered to be identical for all agents, meaning that if one agent is able to send messages to some other agent, then the reverse is true also. In this case, we say agents are able to **communicate**.

The communications graph is constructed by creating a vertex per agent, and adding an edge between vertices when their corresponding agents are able to communicate. Edges disappear when agents exit each other's communication range. Note how this graph is in fact a temporal graph due to the possible appearance and disappearance of edges over time. For example, in Figure 5.2 surely edge $\{1, 2\}$ will disappear, and edge $\{1, 3\}$ appear in the near future. Such a communications graph is also known as a **unit disk graph**.

Mobility models are defined as an algorithm controlling mobile agents according to their mobility constraints, to achieve a certain objective, such as exploration, surveillance, delivery or other typical robot or swarm objectives. Our use of mobility models has as objective to create, as well as maintain some recurrent temporal properties on the graph.

A multitude of mobility models are present in the literature under various names (such as mobility strategies), and for various purposes. See for example [81, 34, 14, 88, 43] as well as some control theory papers [38, 68]. In fact, one paper already gives a mobility model for ensuring $\mathcal{C}^*$, namely Michael *et al.*'s control theory based distributed mobility model for mobile robot networks [68].

To obtain temporal properties on the graph through a mobility model, we use an algorithm which has access to all mobile agents, at all points in time, and may order these agents to move as it pleases (obeying only some given mobility constraints).

Remark 6. *Due to the inclusions shown in the temporal connectivity hierarchy (see Figure 5.1), one may technically only need to construct mobility models for the most specialized properties $\mathcal{K}^{\mathcal{R}}$, $\mathcal{E}^{\mathcal{P}}$, α-$\mathcal{TC}^{\mathcal{B}}$ and $\mathcal{C}^{\cap}$ (the leftmost ones), since these are included in all other properties. We however create mobility models for intermediate classes (such as $\mathcal{E}^{\mathcal{B}}$) which ensure that any more precise property (so $\mathcal{E}^{\mathcal{P}}$ for $\mathcal{E}^{\mathcal{B}}$) are* ***not*** *present in the graph. Outside of the added challenge, this may be justified as a security measure, for example maybe one would like $\mathcal{E}^{\mathcal{B}}$ to be present in the MANET's graph for some algorithm to function correctly, however the presence of $\mathcal{E}^{\mathcal{P}}$ would allow some adversary to corrupt the MANET.*

5.2 Proposed models

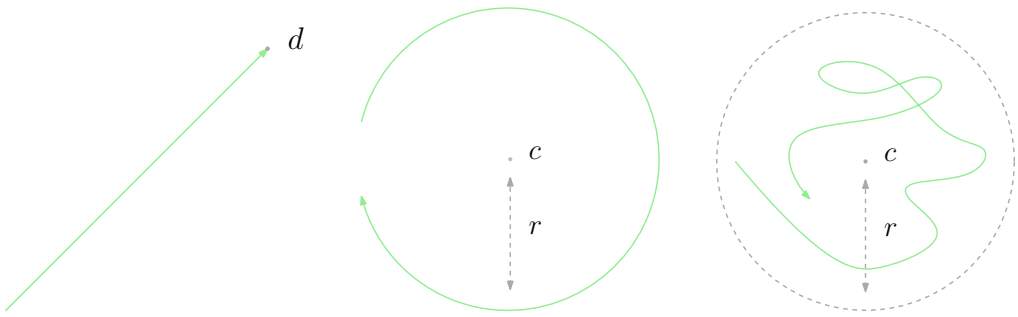
Before actually presenting our mobility models for MANET, we first present what we call "atomic blocks" in the following, which are very basic mobility models for individual agents. These can also be seen as states for agents to be in. Afterwards, we construct mobility models for MANET by manipulating agents through these atomic blocks.
We then present our mobility models, leaving out implementation details. We prove correctness of the mobility models concerning the property they induce (and exclude). We finish by discussing the different mobility constraints which we choose to implement in the software package.

5.2.1 Atomic blocks

We define three atomic blocks (see Figure 5.3), which are meant to be basic mobility models which should be executable by the agents, independently from any mobility constraints one may impose on the agents. They are defined as follows on the plane:

- **Messenger**: the agent makes straight for its destination d. If the agent is already arrived at d, the agent simply comes to a stop for as long as it is in this state and its destination d does not change.

  Variations include limiting the maximum speed at which the agent moves towards its destination. Also, messenger may be used to keep agents at a stop, by simple



(a) Messenger. (b) Roundabout. (c) Swarm.

Figure 5.3: The three atomic blocks used for our mobility models. The path or trajectory of an agent is shown as a green arrow.

setting their destination to their current position (taking into account possible inertia forces at play).

- **Roundabout**: the agent turns (clockwise or counterclockwise) in circles of radius r around center c. Any agent not already at distance r from c first makes for some destination which is (through messenger).

  Variations include forcing a certain low speed on the turning vehicle. Stops are simulated through messenger.

- **Swarm**: The movement of a swarm agent is random, as long as it remains within distance r from the swarm's center c. Any agent outside of the swarm first makes for a destination inside the swarm (through messenger).

  Variations include different forms of swarms, so not necessarily a disk.

5.2.2 Mobility models

Due to Remark 6, in the following, whenever we present a mobility model for some class $\mathcal{P}$, it should be understood as a mobility model for $\mathcal{P} \setminus \mathcal{P}'$ for all (identified) subclasses $\mathcal{P}'$ of $\mathcal{P}$, unless stated otherwise.

We present mobility models inducing properties $\mathcal{TC}^{\mathcal{R}}$, $\mathcal{K}^{\mathcal{R}}$, $\mathcal{C}^{\mathcal{R}}$, $\mathcal{P}^{\mathcal{R}}$, $\mathcal{C}^{*}$, $\mathcal{TC}^{\mathcal{B}}$, $\mathcal{E}^{\mathcal{P}}$, $\mathcal{E}^{\mathcal{B}}$, and $\mathcal{E}^{\mathcal{R}}$ (in this order) for MANET composed of agents with communication range `comrange`. Only one mobility model per property will be presented in this document, although some properties have multiple mobility models.

We often start mobility models by **initializing** agents. With this, we mean to arrange the agents in a certain position (and/or atomic block) so as to start the mobility model.

If the mobility model has the power to do so before communication between agents starts, then the resulting graph admits the desired property. If however the mobility model does not have control of the starting position of the agents (for example the MANET starts at some precise position corresponding to a warehouse or some other practical starting position), then we use the messenger atomic block to move each agent to its initial position. Properties may then not be valid on the graph as a whole, due to unavoidable interactions at the start, but in this case we ensure the properties are satisfied eventually.

Definition 26. *A property $\mathcal{P}$ is **eventually** present in temporal graph $\mathcal{G} = (G_0, G_1, ...)$, if there exists a time t such that the temporal subgraph $\mathcal{G}' = (G_t, G_{t+1}, ...)$ respects property $\mathcal{P}$.*

Concerning eventual inclusion, $\mathcal{E}^{\mathcal{R}}$ has the following useful property, due to which all presented mobility models ensure inclusion in $\mathcal{E}^{\mathcal{R}}$ eventually.

Fact 11. *Any $\mathcal{TC}^{\mathcal{R}}$ graph is eventually in $\mathcal{E}^{\mathcal{R}}$.*

The reader may like to visualize the presented mobility models through our software package, available here[3]. This is not necessary, as the explanations below should suffice. If the reader wishes to visualize the mobility models, we recommend executing:

```
1 public static void main(String[] args) {
2     new MobilityModel(new TCR_swarm(), new Racetrack());
3 }
```

Once executed, this will open up a window with the created MANET, which by default is paused. To resume, use the right click menu. The agents will start moving according to their mobility model. One may at any point accelerate or slow down JBotSim's clock speed through the right click menu as well.

The reader may afterwards modify `TCR_swarm()` for other properties presented. For now, `Racetrack()` may be left untouched, as it is a constraint, presented in Section 5.2.3.

5.2.2.1 Swarm mobility model

First, let's take a look at a mobility model for arguably the most basic of temporal properties, namely $\mathcal{TC}^{\mathcal{R}}$. Although this class has the most subclasses which we would like

[3] `https://www.labri.fr/perso/jschoete/index.php?id=mobility-models` or `https://github.com/jschoete/mobilitymodels`

to exclude, the mobility model we present is fairly straightforward. The **swarm** mobility model for $\mathcal{TC}^{\mathcal{R}}$ (denoted $\mathcal{TC}^{\mathcal{R}}$-`swarm`):

1. Initialize half of the agents in the swarm atomic block with some center c and some radius r. We refer to this swarm as swarm 1.
2. Initialize the other half in the swarm atomic block also, but with center c' such that $||c - c'||^4 > 2(r + \texttt{comrange})$, and radius r. We refer to this swarm as swarm 2.
3. Initialize one agent in swarm 1 as the messenger atomic block, with destination $d = c'$. Refer to this agent as the messenger.
4. Whenever the messenger attains its destination, it stays in the corresponding swarm for a random amount of time, before setting its destination to the other swarm's center.

Lemma 17. *$\mathcal{TC}^{\mathcal{R}}$-`swarm` induces a temporal graph with property $\mathcal{TC}^{\mathcal{R}}$.*

Proof. Due to the infinite lifetime of the MANET, and the (possibly small) chance to either directly interact with an agent in one's own swarm, or indirectly interact send a message through the messenger to the other swarm, $\mathcal{TC}^{\mathcal{R}}$ is maintained. Since the messenger is allowed to stay in a swarm indefinitely long, $\mathcal{TC}^{\mathcal{B}}$ is not present between agents from different swarms. The precise distancing of the swarms makes $\mathcal{P}^{\mathcal{R}}$ impossible between agents from different swarms. The swarm mobility model does however respect $\mathcal{E}^{\mathcal{R}}$, which is something we wanted to avoid. A simple rule may be added so as to keep two messengers instead of one up to a certain point in time, after which only one remains, but this seemed somewhat of an artificial rule. Also, due to Fact 11, any mobility model will not avoid eventual inclusion in $\mathcal{E}^{\mathcal{R}}$. □

A first very simple modification to this swarm mobility model, so as to obtain a mobility model for $\mathcal{K}^{\mathcal{R}}$, is to remove one of the swarms completely, as well as the messenger. This is the $\mathcal{K}^{\mathcal{R}}$-`swarm` mobility model.

Lemma 18. *$\mathcal{K}^{\mathcal{R}}$-`swarm` induces a temporal graph with property $\mathcal{K}^{\mathcal{R}}$.*

Proof. Due to all agents moving randomly in the swarm, and the infinite lifetime of the MANET, all possible pairs of agents will communicate infinitely often. □

[4] The Euclidean distance between p_i and p_o.

Next is a modification of $\mathcal{TC}^{\mathcal{R}}$-swarm which induces $\mathcal{C}^{\mathcal{R}}$ in the communications graph. Simply move the swarms closer together, so that $2r + \texttt{comrange} < ||c - c'|| \leq 2(r + \texttt{comrange})$ (*i.e.*, the swarms' boundaries are separated by distance d with $\texttt{comrange} < d \leq 2\texttt{comrange}$). The result is $\mathcal{C}^{\mathcal{R}}$-swarm.

Lemma 19. *$\mathcal{C}^{\mathcal{R}}$-swarm induces a temporal graph with property $\mathcal{C}^{\mathcal{R}}$.*

Proof. Whenever the messenger moves in between swarms, there's a chance it may connect the whole graph in some snapshots. This (possibly small) chance is sufficient to ensure $\mathcal{C}^{\mathcal{R}}$, although one may add additional rules so as to increase the odds of this happening. $\mathcal{C}^*$ is not satisfied here, since the graph is disconnected most of the time. □

Take $\mathcal{C}^{\mathcal{R}}$-swarm and add a third swarm. Make sure all three swarms are still all distanced from each other as stated for $\mathcal{C}^{\mathcal{R}}$-swarm. This is possible in the plane with disk-shaped swarms, for example in a triangular fashion. Suppose a third of all agents are in each swarm. Now, instead of going back and forth between only two swarms, the messenger has a choice to make (at random) as to which of the other two swarms it visits. We refer to this mobility model as $\mathcal{P}^{\mathcal{R}}$-swarm.

Lemma 20. *$\mathcal{P}^{\mathcal{R}}$-swarm induces a temporal graph with property $\mathcal{P}^{\mathcal{R}}$.*

Proof. Similarly to $\mathcal{C}^{\mathcal{R}}$-swarm, the messenger may connect the subgraph induced by the two swarms it moves between, and thus create paths between all vertices corresponding to the agents. Since the messenger has the possibility to do this between any pair of the three swarms, $\mathcal{P}^{\mathcal{R}}$ is ensured in the graph. We ensure $\mathcal{C}^{\mathcal{R}}$ is not satisfied, since the graph may never be connected as a whole in any given snapshot, due to the messenger only being able to connect two of the three swarms. Also, we ensure $\mathcal{K}^{\mathcal{R}}$ is not present in the graph (this is the missing inclusion from Remark 5). Even though paths between all pairs of vertices occur infinitely often, edges between all pairs of vertices do not, as no edges appear between vertices representing agents from different swarms. □

Remark 7. *It is possible to extend $\mathcal{P}^{\mathcal{R}}$-swarm by considering four swarms (possibly needing to alter the form of the swarms). In two dimensions however, four is the limit of the amount of swarms, as in two dimensions this mobility model can be modelized as a planar complete graph in which the vertices represent swarms and the edges represent almost-touching boundaries of swarms. As is known, for all $n \geq 5$, the complete graph on n vertices is non-planar. Allowing for swarms in three dimensions solves this problem, and allows for any number of swarms.*

5.2.2.2 Grid swarm mobility model

The following mobility models are greatly inspired from the swarm mobility models presented, and are basically an adaptation so as to ensure some structure inside of swarms. The **grid swarm** mobility model for $\mathcal{C}^*$ (denoted $\mathcal{C}^*$-`gridswarm`):

1. Initialize agents in the swarm atomic block with some center c and some radius r.
2. Place a stationary agent at c (so in the messenger atomic block with destination c).
3. While the area of the swarm is not covered by stationary agents' combined communication range, add stationary agents at distance `comrange` to the left, right, top and bottom of already stationary agents (if not already present).
4. Allow passing swarm agents to replace stationary agents.

Lemma 21. *$\mathcal{C}^*$-`gridswarm` induces a temporal graph with property $\mathcal{C}^*$.*

Proof. The graph is always connected thanks to the stationary agents which induce a connected subgraph which covers the swarm's area, thus connecting to all the swarm's agents. This thus results in an always connected and spanning induced graph. The replacing of stationary agents by swarm agents is to make sure $\mathcal{C}^\cap$ is not satisfied through this mobility model. □

As somewhat of a combination between the two swarms with the messenger in $\mathcal{TC}^\mathcal{R}$-`swarm`, and $\mathcal{C}^*$-`gridswarm`, the next mobility model ensures $\mathcal{TC}^\mathcal{B}$. Give both swarms an connected grid of stationary agents, and don't allow for the messenger agent to indefinitely stay in one swarm.

Lemma 22. *$\mathcal{TC}^\mathcal{B}$-`gridswarm` induces a temporal graph with property $\mathcal{TC}^\mathcal{B}$.*

Proof. If each swarm in itself induces a temporal subgraph in $\mathcal{TC}^\mathcal{B}$, then forbidding the messenger to remain in a swarm indefinitely would have been sufficient for the whole mobility model to induce a graph in $\mathcal{TC}^\mathcal{B}$. However, due to our assumption that mobility is random inside swarms, each swarm may not induce a temporal subgraph in $\mathcal{TC}^\mathcal{B}$ (even though a probabilistic bound may exist). Adding the cover of connected stationary agents solves this issue, as now each swarm is in $\mathcal{TC}^\mathcal{B}$, say with time bound b. The overall time bound for the whole mobility model's induced temporal graph would then be at most $2(b + t)$, where t is the time it takes for the messenger to go from one boundary of a

swarm to the other swarm's boundary (this bound may not be tight, but still counts as a valid bound). □

One may allow the replacement of stationary agents by passing swarm agents to allow for more mobility, although this is not a necessary condition for $\mathcal{TC}^{\mathcal{B}}$.

5.2.2.3 Token mobility model

To address the remaining properties ($\mathcal{E}^{\mathcal{P}}$, $\mathcal{E}^{\mathcal{B}}$, and $\mathcal{E}^{\mathcal{R}}$), we present the following mobility models, in which a token is passed between interacting agents. Only the agent with the token is able to move. The **token** mobility model for $\mathcal{E}^{\mathcal{P}}$ (denoted $\mathcal{E}^{\mathcal{P}}$-`token`):

1. Initialize agents in rows and columns, in a grid-like formation, distanced by some distance $d >$ `comrange`. Denote this position as p_i for each agent.
2. Offset agents on the top row by $\frac{d-\texttt{comrange}}{2}$ to the left.
3. Offset agents on the leftmost column by $\frac{d-\texttt{comrange}}{2}$ to the top (except for the top left agent).
4. Offset other agents by $\frac{d-\texttt{comrange}}{2}$ to the left, or to the top, at random.
5. Denote each agent's position as p_o for each agent.
6. Give the top left agent, denoted a_0, a token.
7. Agents with the token move according to the roundabout atomic block, clockwise or counterclockwise (decided at random), at a constant speed (not necessarily the same speed as other agents), with center $c = p_i$, and radius $r = ||p_o - p_i||$.
8. Whenever agents communicate, the token is exchanged. One agent then loses the token and thus comes to a stop, while the other gains the token and thus starts moving.

Lemma 23. *$\mathcal{E}^{\mathcal{P}}$-`token` induces a temporal graph with property $\mathcal{E}^{\mathcal{P}}$.*

Proof. The token mobility model passes around a token between agents. Due to the careful positioning of the agents, all agents will at some point in time receive the token either from their top neighbor, or from their left neighbor (depending on p_o). One may observe that the token's owner follows a pattern which is repeated over time. Indeed, the second time that a_0 has the token and attains its position p_o, all agents' positions

are again p_o, and so the token is passed again through the MANET, in the exact same manner it was already passed. The time taken for the token to come back to a_0 at position p_o corresponds to the period at which edges appear in the communications graph. □

The token mobility model may easily be adapted for $\mathcal{E}^{\mathcal{B}}$ by allowing agents to change speed in the roundabout atomic block, instead of forcing them to keep the same individual speed. However, to keep a bound on edges' appearances, one must define a minimum speed. Let's refer to this modified token mobility model as $\mathcal{E}^{\mathcal{B}}$-`token`.

Lemma 24. *$\mathcal{E}^{\mathcal{B}}$-`token` induces a temporal graph with property $\mathcal{E}^{\mathcal{B}}$.*

Proof. The bound on the interactions reoccurring is the time it would take for a_0 to reach p_o again if all agents turn at minimum speed. Also, $\mathcal{E}^{\mathcal{P}}$ is not satisfied due to the random speed changes. □

Through a similar argument, modifying $\mathcal{E}^{\mathcal{B}}$-`token`, by allowing agents to randomly come to a stop, even while having the token, gives $\mathcal{E}^{\mathcal{R}}$-`token`.

Remark 8. *The communications graph created through the token mobility models is a temporal tree (the footprint, as well as the eventual footprint is a tree). Due to the random offset assigned to the agents, the created temporal tree is random as well.*

One may have some arguments against the token mobility model. First of, maybe it isn't mobile enough, in the sense that at any given point in time only one agent can move, while all others are at a stop. Also, most of the time, agents are at a stop, waiting for the token. Secondly, only a sparse amount of edges ($n-1$ edges, to be precise) appear in the communications graph. The next mobility model is very similar to the token mobility models, but allows for constant mobility of the agents after some time. We discuss and resolve the second argument later in Remark 9.

5.2.2.4 Roundabout mobility model

As somewhat of an extension of the token mobility model, the next mobility models aim to ensure the same properties while keeping agents mobile a majority of the time. The **roundabout** mobility model for $\mathcal{E}^{\mathcal{R}}$ (denoted $\mathcal{E}^{\mathcal{R}}$-`roundabout`):

1. Initialize agents as in the token mobility model, denoted as position p_i for every agent.

2. Offset the agents as in the token mobility model, denoted as position p_o for every agent.

3. The top left agent moves according to the roundabout atomic block, clockwise or counterclockwise (decided at random), at some speed, with center $c = p_i$, and radius $r = ||p_o - p_i||$.

4. When agents communicate, they move according to the roundabout atomic block, clockwise or counterclockwise (decided at random), at some fixed random speed, with center $c = p_i$, and radius $r = ||p_o - p_i||$ (if they do not already do so).

5. All agents may be put to a stop at random points in time, for a random amount of time.

Lemma 25. *$\mathcal{E}^{\mathcal{R}}$-`roundabout` induces a temporal graph with property $\mathcal{E}^{\mathcal{R}}$.*

Proof. Agents will communicate infinitely often with all of their adjacent neighbors, simply because there is a (possibly small) chance for agents to find themselves close enough to do so, due to the random stops and speed changes allowed, and due to the lifetime of the MANET being theoretically infinite. $\mathcal{E}^{\mathcal{B}}$ isn't satisfied though, since no bound may be put on the communications' reappearance, due to the agents being able to stop for any amount of time. □

Removing the last rule of $\mathcal{E}^{\mathcal{R}}$-`roundabout` results in $\mathcal{E}^{\mathcal{P}}$-`roundabout`. Both may even be adapted for $\mathcal{E}^{\mathcal{B}}$-`roundabout`, although some technical details arise in this case, depending on the constraints, which we will not cover in this document.

Remark 9. *The repeated trajectory of agents in the roundabout mobility models is necessary for our results, however the explicit roundabout shape of the trajectory is not. In other words, one may have considered other trajectories, such as a 45° tilted square shape, and the results would still hold. In fact, as a stronger statement, any repeated trajectory would do, as long as the agents induce a connected footprint (which is ensured in our model through the initial conditions). This may a priori induce graphs of arbitrary density, as opposed to our proposed mobility models inducing sparse graphs.*

5.2.3 Mobility constraints

Two mobility constraints have been implemented, namely `Racetrack` and the discrete model used for the robot in Figure 2.11, referred to here as `Gridwalk`. Since our mobility

models are composed entirely of three atomic blocks, only these atomic blocks had to be implemented for each constraint.

Readers using the software package to visualize mobility models may now switch between mobility constraints `Racetrack()` and `Gridwalk()`.

The main differences between the two are as follows.

- The Racetrack constraints allow for faster movement compared to grid walk, due to the possibility of attaining a greater velocity at each time step, whereas grid walk is stuck with a constant speed. To counterbalance this, one could consider for grid walk either a larger grid unit or an increased JBotSim internal clock speed. We choose the latter solution.

- In the Racetrack constraints, the roundabout atomic block was implemented in a square form as opposed to the circular form, for convenience. As discussed in Remark 9, this doesn't affect the induced properties.

- In the grid walk constraints, agents in the swarm atomic block seem almost reluctant to diffuse from the center area of the swarm. This is especially true when compared to swarm agents in the Racetrack constraints, which seem to prefer the outer sides of the swarm.

Remark 10. *Constant speed mobility constraints (such as grid walk) may at first not seem adaptable for our mobility models, since some rely on changes of speed. However, some of these constant speed mobility constraints may be able to simulate speed changes if one allows the agent a null speed option (a stop). Indeed, "lower" speeds may then be simulated by inserting some periodic stops in the trajectory of the agent. However, higher speeds may not.*

Other mobility constraints are possible, as long as the three atomic blocks may be implemented according to the constraints. Although JBotSim is by definition discrete, it is still possible to simulate continuous mobility constraints, such as Random Waypoint, presented in [23].

5.3 Usage examples

5.3.1 Testing environment

Users may use our mobility models to visualize and test their distributed algorithms. JBotSim proposes the option to extend the `Node` class, which represents the agents in the MANET (or the vertices in the communications graph). We give two examples of simple distributed algorithms (which are already present in the software package). The first is a naive broadcasting algorithm, known as flooding.

```
4  public class FloodNode extends Node {
5      @Override
6      public void onStart() {
7          super.onStart();
8          if (this.getID() == 0)
9              this.setColor(Color.GREEN);
10     }
11
12     @Override
13     public void onClock() {
14         if (this.getColor() != null)
15             this.sendAll(new Message());
16     }
17
18     @Override
19     public void onMessage(Message message) {
20         super.onMessage(message);
21         this.setColor(Color.GREEN);
22     }
23 }
```

To test this distributed algorithm with a mobility model, execute:

```
24 public static void main(String[] args) {
25     Class node = FloodNode.class;
26     new MobilityModel(new EP_token(), new Gridwalk(), node);
27 }
```

As can be observed, over time agents become green, meaning they received the message. As all mobility models induce a $\mathcal{TC}^{\mathcal{R}}$ graph, all agents (or nodes) will at some point in

time receive the message. Ensuring some bound on the completion of the algorithm may be achieved only in some particular classes however (such as $\mathcal{TC}^{\mathcal{B}}$).

As a second example, consider the following election algorithm, which only works correctly if $\mathcal{K}^{\mathcal{R}}$ is satisfied. The remaining green agent then represents the elected agent. Outside of the $\mathcal{K}^{\mathcal{R}}$ mobility models, $\mathcal{C}^*$-`gridswarm` ensures $\mathcal{K}^{\mathcal{R}}$ as well.

```
28  public class EliminationNode extends Node {
29      @Override
30      public void onStart() {
31          super.onStart();
32          this.setColor(Color.GREEN);
33      }
34
35      @Override
36      public void onLinkAdded(Link link) {
37          super.onLinkAdded(link);
38          Node n0 = link.endpoint(0);
39          Node n1 = link.endpoint(1);
40          if (n0.getColor().equals(Color.GREEN))
41              n1.setColor(Color.RED); //elimination
42      }
43  }
```

In general, our mobility models can serve as a testing environment for any distributed algorithms which depend on some temporal property. They can even serve so as to find such a property given the distributed algorithm. Also, all mobility models presented contain some form of randomness, which allows users to use our mobility models to construct random temporal graphs, which may induce/exclude some temporal property.

5.4 Conclusion

In this chapter, we explored an approach on how to control a MANET's mobile entities so as to induce some desired property in the induced temporal graph. We ended up creating a variety of mobility models which ensure and/or exclude most of the temporal properties present in the literature. This is done through a proposed framework allowing for the separation of the given mobility constraints and the desired temporal property, by using atomic mobility models for individual agents which depend solely on the mobility constraints, and which are then used so as to create a mobility model for the MANET

which depends solely on the temporal property. This work was published in the form of a Java package using the JBotSim library, and can be used as a testing environment for distributed algorithms.

Future work and possible extensions are discussed in Chapter 6.

Chapter 6

Conclusion

> *I guess I've been working so hard, I forgot what it's like to be hardly working.*
>
> — Michael Scott

Following are some concluding remarks from this document's chapters.

In Chapter 3, we established that sparse temporal spanners always exist in temporal cliques, proving constructively that one can find $O(n \log n)$ edges that suffice to preserve temporal connectivity. Our results hold for non-strict journeys with single or multiple labels on each edge, and strict journeys with single or multiple labels on each edge with the property that there is a subset of locally exclusive single labels. Our results give the first positive answer to the question of whether any class of dense graphs always has sparse temporal spanners.

To prove our results, we introduced several techniques (pivotability, delegation, dismountability and k-hop dismountability, forward and backward fireworks, partial delegation, and layered delegations), all of which are original and some of which might be of independent interest. Whether some of these techniques can be used for more general classes of graphs is an open question. Delegation and dismounting rely explicitly on the graph being complete; however, refined versions of these techniques like partial delegation might have wider applicability.

Next, in Chapter 4, we introduced a new version of TSP, with the aim of adding some sort of acceleration constraints to the well-known classical problem. This was done through the addition of a simple set of acceleration rules, inspired from the pen and pencil game Racetrack.

Several insights have been shown, some which prove the problem is distinct from the classical Euclidean TSP. We also gave some polynomial-time reductions from and to well-known problems, resulting in the problem being NP-complete in general. We also presented an algorithm resulting in a 2-optimal solution, through the adaptation of a TSP algorithm which uses an A* oracle for cost computation. Some experiments further pushed the VectorTSP as a stand-alone problem, distinct not only in theory, but thus also in practice, from classical TSP.

We finished in Chapter 5 with a software package of mobility models which induce (and/or exclude) several temporal properties on the underlying communications graph. These were implemented in Java using the JBotSim library, and use a framework allowing the separation of desired properties and constraints thanks to the manipulation of simple atomic blocks.

6.1 Open questions and future work

In this thesis, we've covered multiple results. However, several questions are still open concerning our presented work, and thus several subjects for future work remain. In this section, we list these subjects. For more background and details on some questions, the reader is referred to [25] and [27].

6.1.1 Temporal cliques admit sparse spanners

Experimental evidence shows temporal cliques may always admit a spanner of size $2n-3$ or $2n-4$ edges. The latter is in fact a lower bound originally from gossip theory (see *e.g.*, Facts F29 through F32 in [52]). In fact, in these experiments, many instances were found which are neither pivotable (see Section 3.2.3) nor k-hop dismountable (see Section 3.3), and yet admit a spanner of size $2n-3$ or $2n-4$. This suggests further investigation to understand the structure of simple temporal cliques. In particular:

Open question 1. *Do simple temporal cliques always admit $\Theta(n)$-sparse temporal spanners?*

Open question 2. *Do simple temporal cliques always admit temporal spanners with at most $2n-3$ edges?*

A first step towards answering these questions has been made in a probabilistic manner (following our work) in [29], where the authors consider an Erdős-Rényi random graph

$G_{n,p}$ with a uniformly random presence time on every edge. They show, among other thresholds, that at threshold $p = 4 \log n/n$, the graph asymptotically almost surely (*a.a.s.*) admits a spanner with $2n-4$ edges, based among others on the pivotability technique. As a corollary, random temporal cliques *a.a.s.* admit a spanner with $2n-4$ edges. One open question which still remains to be answered in a probabilistic manner, is the following.

Open question 3. *Are random simple temporal cliques a.a.s. (k-hop) dismountable?*

Finally, a natural question is whether a more general class than complete graphs may admit sparse spanners, and what is the role of density *per se*. The family of counter-examples from Axiotis and Fotakis [11] have asymptotic density 1/9, which leaves a significant gap between this family and that of complete graphs.

Open question 4. *Is there a larger class of dense graphs than complete graphs that always admit $o(n^2)$-sparse temporal spanners?*

6.1.2 VectorTSP: A Traveling Salesperson Problem with Racetrack-like acceleration constraints

Concerning VectorTSP, a first open question is if Fact 10 can be generalized for any maximum visiting speed.

Open question 5. *For cities placed on the vertices of a convex hull, does* VectorTSP *admit the clockwise or counterclockwise visit order as an optimal visit order if $\nu \neq 0$?*

Our example presented in Figure 4.5 seems to be adaptable for $\nu = 1$ (by simply removing unnecessary self loops), although proving it for any $\nu > 1$ may not be as simple.

Another question which remains open due to us considering a fixed maximum visiting speed $\nu = 0$, is the NP-hardness of the problem.

Open question 6. *Is* VectorTSP *NP-hard, even if $\nu \neq 0$?*

In particular, does the bounding of the maximum visiting speed even matter concerning the problem's hardness, *i.e.* is the problem NP-hard still if the visiting speed is not bounded?

Open question 7. *Is* VectorTSP *NP-hard, even if $\nu = \infty$?*

Intuitively, our proof of NP-hardness may be adapted to consider visiting speed $\nu = 1$, again by removing unnecessary self loops, although adapting it further to other visiting speeds, let alone no bounded visiting speed, does not seem as straightforward.

Another natural question is whether VectorTSP could be proved NP-hard through a direct reduction from ETSP, rather than from ExactCover.

Open question 8. *Does there exist a direct (natural) reduction from* ETSP *to* VTSP*?*

Another direction left open is the approximability of VectorTSP. Indeed, although EuclideanTSP is known to be approximable to a factor of 1.5, no such result was proven for VectorTSP.

Open question 9. *Is* VectorTSP *approximable, and if so, what is the approximation factor?*

6.1.3 Temporal properties induced by collective mobility

Concerning our work on mobility models inducing temporal properties, multiple options to extend the software package are possible. One might be interested in developing distributed mobility models, in which mobility models are executed by agents and have only access to information inside of their communication range. It is indeed more realistic to assume that agents with a short communication range should not be able to communicate with (or rather be controlled by) a centralized algorithm, which in principle isn't located in this communication range. Many of our proposed models were originally designed (as well as implemented) as distributed mobility models instead of centralized ones. However, not all properties may be satisfiable through a distributed mobility model.

Open question 10. *Which properties are (or are not) satisfiable through a distributed mobility model?*

If some property isn't satisfiable through a distributed mobility model with no global knowledge, one may look into what global knowledge does allow the agents to satisfy the property (such as the number of agents in the MANET, the dimensions of the topology, *etc.*).

Other future work may include implementing more mobility models, in particular ones for classes α-$\mathcal{TC}^{\mathcal{B}}$, and $\mathcal{C}^{\cap}$. Another option is implementing other mobility constraints. One may also be interested in allowing easy initialization or modification of the parameters of mobility models, so as to obtain mobility models which are more adaptable. Indeed, at the moment our mobility models execute on a fixed amount of agents, in a fixed area, starting in a fixed position and a fixed state or atomic block, *etc.*. Lastly, one may consider adding typical MANET missions such as delivery, exploration, *etc.* on top of ensuring temporal properties.

6.2 In the next few years

I think it's appropriate for me to state here as well what I would like to accomplish for myself in the not so distant future.
First of all, now that my defense has passed, I can proudly say I have applied, and have been accepted, for a postdoc position in which I will have the opportunity to continue working on temporal graph theory. There's one or two problems which I have noted down which may be of interest, in addition to my soon-to-be boss already having some ideas of his own for us to work on.
After this postdoc, I would probably apply for a second postdoc somewhere abroad for a year or two (or more?), so as to discover not only the world but different researchers and their corresponding mind sets and teams as well. The subject itself doesn't matter too much to me at the moment, since I'm interested in a wide variety of domains, and more than willing to learn about different subjects. I prefer to see a "Jack of all trades, master of none" as a potentially positive quality, especially if it turns out it's possible to in fact become a "Jack of all trades, master of **some**"!
I haven't thought about things much further than that. A permanent post in a permanent location and lab, is a decision which should be made carefully, and not set in stone too early. If at some point there's a possibility to apply for such a post, I will decide at that time. I'm curious to see where this journey will end.

Bibliography

[1] Eric Aaron, Danny Krizanc, and Elliot Meyerson. Dmvp: foremost waypoint coverage of time-varying graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 29–41. Springer, 2014.

(page 111)

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

(page 56)

[3] Eleni C. Akrida, Leszek Gasieniec, George B. Mertzios, and Paul G. Spirakis. The complexity of optimal design of temporally connected graphs. *Theory of Computing Systems*, 61(3):907–944, 2017.

(page 26, 51, 52, 53, 54, 55, 56)

[4] Karine Altisen, Stéphane Devismes, Anaïs Durand, Colette Johnen, and Franck Petit. Self-stabilizing systems in spite of high dynamics. In *International Conference on Distributed Computing and Networking 2021*, pages 156–165, 2021.

(page 10, 112)

[5] Lars Døvling Andersen. Hamilton circuits with many colours in properly edge-coloured complete graphs. *Mathematica scandinavica*, pages 5–14, 1989.

(page 30)

[6] David Applegate, Ribert Bixby, Vasek Chvatal, and William Cook. Concorde tsp solver, 2006. `http://www.math.uwaterloo.ca/tsp/concorde/`.

(page 105)

[7] Esther M Arkin and Refael Hassin. Approximation algorithms for the geometric covering salesman problem. *Discrete Applied Mathematics*, 55(3):197–218, 1994.

(page 84)

[8] Sanjeev Arora. Polynomial time approximation schemes for euclidean tsp and other geometric problems. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 2–11. IEEE, 1996.

(page 79)

[9] Sunil Arya, David M Mount, and Michiel Smid. Dynamic algorithms for geometric spanners of small diameter: Randomized solutions. *Computational Geometry*, 13(2):91–107, 1999.

(page 51)

[10] B. Awerbuch and S. Even. Efficient and reliable broadcast is achievable in an eventually connected network. In *Proceedings of 3rd Symposium on Principles of Distributed Computing (PODC)*, pages 278–281, 1984.

(page 26)

[11] Kyriakos Axiotis and Dimitris Fotakis. On the size and the approximability of minimum temporally connected subgraphs. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 149:1–149:14, 2016.

(page 8, 13, 26, 50, 51, 52, 54, 129)

[12] Michael A Bekos, Till Bruckdorfer, Henry Förster, Michael Kaufmann, Simon Poschenrieder, and Thomas Stüber. Algorithms and insights for racetrack. *Theoretical Computer Science*, 2018.

(page 48, 80, 100, 102, 104)

[13] Adi Botea, Bruno Bouzy, Michael Buro, Christian Bauckhage, and Dana Nau. Pathfinding in games. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

(page 42)

[14] Quentin Bramas, Stéphane Devismes, and Pascal Lafourcade. Infinite grid exploration by disoriented robots. In *International Colloquium on Structural Information and Communication Complexity*, pages 340–344. Springer, 2019.

(page 113)

[15] Bin-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(2):267–285, 2003.

(page 26)

[16] John Canny, Bruce Donald, John Reif, and Patrick Xavier. *On the complexity of kinodynamic planning.* IEEE, 1988.

(page 79)

[17] John Canny, Ashutosh Rege, and John Reif. An exact algorithm for kinodynamic planning in the plane. *Discrete & Computational Geometry*, 6(3):461–484, 1991.

(page 79)

[18] Arnaud Casteigts. *A Journey through Dynamic Networks (with Excursions).* Habilitation à diriger des recherches, Université de Bordeaux, June 2018.

(page 10, 14, 27, 29, 30, 110, 111)

[19] Arnaud Casteigts, Paola Flocchini, Emmanuel Godard, Nicola Santoro, and Masafumi Yamashita. On the expressivity of time-varying graphs. *Theoretical Computer Science*, 590:27–37, 2015.

(page 24)

[20] Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. Shortest, fastest, and foremost broadcast in dynamic networks. *International Journal of Foundations of Computer Science*, 26(4):499–522, 2015.

(page 26, 111)

[21] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

(page 10, 14, 23, 24, 27, 29, 110)

[22] Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. The computational complexity of finding temporal paths under waiting time constraints. *CoRR*, abs/1909.06437, 2019.

(page 26)

[23] Arnaud Casteigts and Rémi Laplace. Jbotsim, a tool for fast prototyping of distributed algorithms in dynamic networks. *In Proc of SIMUTools*, 2015.

(page 10, 110, 123)

[24] Arnaud Casteigts, Joseph Peters, and Jason Schoeters. Trouver des spanners peu denses dans les cliques temporelles. 2019.

(page 50)

[25] Arnaud Casteigts, Joseph G. Peters, and Jason Schoeters. Temporal cliques admit sparse spanners. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 134:1–134:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

(page 9, 13, 50, 128)

[26] Arnaud Casteigts, Joseph G Peters, and Jason Schoeters. Temporal cliques admit sparse spanners. *arXiv preprint arXiv:1810.00104*, 2019.

(page 50)

[27] Arnaud Casteigts, Mathieu Raffinot, and Jason Schoeters. Vectortsp: A traveling salesperson problem with racetrack-like acceleration constraints. In *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics*, pages 45–59. Springer, 2020.

(page 9, 14, 78, 128)

[28] Arnaud Casteigts, Mathieu Raffinot, and Jason Schoeters. Vectortsp: A traveling salesperson problem with racetrack-like acceleration constraints. *arXiv preprint arXiv:2006.03666*, 2020.

(page 78)

[29] Arnaud Casteigts, Michael Raskin, Malte Renken, and Viktor Zamaraev. Sharp thresholds in random simple temporal graphs. *arXiv preprint arXiv:2011.03738*, 2020.

(page 128)

[30] Paul Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of 2nd Symposium on Computational Geometry*, pages 169–177, 1986.

(page 51)

[31] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

(page 79)

[32] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

(page 37, 39)

[33] Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

(page 98)

[34] Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theoretical Computer Science*, 609:171–184, 2016.

(page 113)

[35] H.N. de Ridder et al. Information system on graph classes and their inclusions (isgci). 2001-2014. `https://www.graphclasses.org`.

(page 21)

[36] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Aaai*, volume 6, pages 942–947, 2006.

(page 42)

[37] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

(page 44)

[38] Dimos V Dimarogonas and Karl H Johansson. Bounded control of network connectivity in multi-agent systems. *IET control theory & applications*, 4(8):1330–1338, 2010.

(page 113)

[39] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 40(5):1048–1066, 1993.

(page 79)

[40] Michael Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proceedings of 26th ACM Symposium on Principles of Distributed Computing*, pages 185–194, 2007.

(page 51)

[41] Jessica Enright, Kitty Meeks, George B Mertzios, and Viktor Zamaraev. Deleting edges to restrict the size of an epidemic in temporal networks. *arXiv preprint arXiv:1805.06836*, 2018.

(page 26)

[42] Jeff Erickson. Ernie's 3d pancakes : "how hard is optimal racing?". 2009. http://3dpancakes.typepad.com/ernie/2009/06/how-hard-is-optimal-racing.html.

(page 84, 85)

[43] Ema Falomir, Serge Chaumette, and Gilles Guerrini. Mobility strategies based on virtual forces for swarms of autonomous uavs in constrained environments. *International Conference on Informaticsin Control, Automation and Robotics (ICINCO 2017)*, pages 221–229, 2017.

(page 113)

[44] Afonso Ferreira. Building a reference combinatorial model for manets. *IEEE network*, 18(5):24–29, 2004.

(page 23)

[45] M Gardner. Sim, chomp and race track-new games for intellect (and not for lady luck). *Scientific American*, 228(1):108–115, 1973.

(page 9, 13, 45, 78, 82)

[46] Michael R Garey, Ronald L Graham, and David S Johnson. Some np-complete geometric problems. *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 10–22, 1976.

(page 79)

[47] Cyril Gavoille. Notes de cours "techniques algorithmiques et programmation". https://dept-info.labri.fr/ gavoille/UE-TAP/cours.pdf.

(page 42)

[48] Carlos Gómez-Calzado, Arnaud Casteigts, Alberto Lafuente, and Mikel Larrea. A connectivity model for agreement in dynamic systems. In *European Conference on Parallel Processing*, pages 333–345. Springer, 2015.

(page 112)

[49] Lee-Ad Gottlieb and Liam Roditty. Improved algorithms for fully dynamic geometric spanners and geometric routing. In *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 591–600, 2008.

(page 51)

[50] Stephen Gould, Tom Kelly, Daniela Kühn, and Deryk Osthus. Almost all optimally coloured complete graphs contain a rainbow hamilton path. *arXiv preprint*

arXiv:2007.00395, 2020.

(page 30)

[51] Frank Harary and Gopal Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

(page 23)

[52] H.A. Harutyunyan, A.L. Liestman, J.G. Peters, and D. Richards. Broadcasting and gossiping in communication networks. In P. Zhang J.L. Gross, J. Yellen, editor, *Handbook of Graph Theory, Second Edition*, chapter 12.2, pages 1477–1494. CRC Press, 2013.

(page 128)

[53] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210, 1962.

(page 35, 79)

[54] Petter Holme and Jari Saramäki. Temporal networks as a modeling framework. pages 1–14. Springer, 2013.

(page 27)

[55] Markus Holzer and Pierre McKenzie. The computational complexity of racetrack. *International Conference on Fun with Algorithms*, pages 260–271, 2010.

(page 47, 80, 85)

[56] Arthur M Jaffe. The millennium grand challenge in mathematics. *Notices of the AMS*, 53(6), 2006.

(page 39)

[57] Roy Jonker and Ton Volgenant. Transforming asymmetric into symmetric traveling salesman problems: erratum. *Operations research letters*, 5(4):215–216, 1986.

(page 98)

[58] Paris-C Kanellakis and Christos H Papadimitriou. Local search for the asymmetric traveling salesman problem. *Operations Research*, 28(5):1086–1099, 1980.

(page 98)

[59] Richard M Karp. Reducibility among combinatorial problems. *Complexity of computer computations*, pages 85–103, 1972.

(page 38, 79)

[60] D. Kempe, J. Kleinberg, and A. Kumar. Connectivity and inference problems for temporal networks. In *32nd ACM Symposium on Theory of Computing (STOC)*, pages 504–513, 2000.

(page 8, 13, 50)

[61] David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. *Journal of Computer and System Sciences*, 64(4):820–842, 2002.

(page 26, 50, 52, 54)

[62] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. Distributed computation in dynamic networks. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 513–522, 2010.

(page 112)

[63] Ratnesh Kumar and Haomin Li. On asymmetric tsp: Transformation to symmetric tsp and performance bound. *Journal of Operations Research*, 6, 1996.

(page 98)

[64] Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Social Network Analysis and Mining*, 8(1):61, 2018.

(page 23)

[65] Jerome Le Ny, Emilio Frazzoli, and Eric Feron. The curvature-constrained traveling salesman problem for high point densities. In *2007 46th IEEE Conference on Decision and Control*, pages 5985–5990. IEEE, 2007.

(page 79, 80, 89)

[66] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

(page 37, 39)

[67] On line Encyclopedia of Integer Sequences. Sequence a027434. `https://oeis.org/A027434`.

(page 103)

[68] Nathan Michael, Michael M Zavlanos, Vijay Kumar, and George J Pappas. Maintaining connectivity in mobile robot networks. In *Experimental Robotics*, pages

117–126. Springer, 2009.

(page 113)

[69] Othon Michail and Paul G Spirakis. Elements of the theory of dynamic networks. *Communications of the ACM*, 61(2):72–72, 2018.

(page 27)

[70] Joseph SB Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric tsp, k-mst, and related problems. *SIAM Journal on computing*, 28(4):1298–1309, 1999.

(page 79)

[71] Hendrik Molter. *Classic graph problems made temporal–a parameterized complexity analysis*, volume 13. Universitätsverlag der TU Berlin, 2020.

(page 31)

[72] Giri Narasimhan and Michiel Smid. *Geometric Spanner Networks*. Cambridge Univ. Press, 2007.

(page 51)

[73] Charles E Noon and James C Bean. An efficient transformation of the generalized traveling salesman problem. *INFOR: Information Systems and Operational Research*, 31(1):39–44, 1993.

(page 97, 98)

[74] Ariel Orda and Raphael Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM (JACM)*, 37(3):607–625, 1990.

(page 26)

[75] Pekka Orponen and Heikki Mannila. On approximation preserving reductions: Complete problems and robust measures (revised version). *Department of Computer Science, University of Helsinki*, 1990.

(page 79)

[76] Christos Papadimitriou and Santosh Vempala†. On the approximability of the traveling salesman problem. *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, 26:101–120, 02 2006.

(page 79)

[77] Christos H Papadimitriou. The euclidean travelling salesman problem is np-complete. *Theoretical computer science*, 4(3):237–244, 1977.

(page 79, 92)

[78] David Peleg and Alejandro A Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

(page 51)

[79] Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

(page 85)

[80] Ram Ramanathan, Prithwish Basu, and Rajesh Krishnan. Towards a formalism for routing in challenged networks. In *Proceedings of the second ACM workshop on Challenged networks*, pages 3–10, 2007.

(page 112)

[81] Radhika Ranjan Roy. Mobile ad hoc networks. In *Handbook of Mobile Ad Hoc Networks for Mobility Models*, pages 3–22. Springer, 2011.

(page 113)

[82] Ketan Savla, Emilio Frazzoli, and Francesco Bullo. On the point-to-point and traveling salesperson problems for dubins' vehicle. In *proceedings of the American Control Conference*, volume 2, page 786, 2005.

(page 80)

[83] Jakob Schmid. Vectorrace - finding the fastest path through a two-dimensional track. *URL: http://schmid.dk/articles/vectorRace.pdf*, 2005.

(page 100)

[84] Mathilde Vernet. *Modèles et algorithmes pour les graphes dynamiques*. PhD thesis, Normandie, 2020.

(page 31)

[85] Mathilde Vernet, Maciej Drozdowski, Yoann Pigné, and Eric Sanlaville. A theoretical and experimental study of a new algorithm for minimum cost flow in dynamic graphs. *Discrete Applied Mathematics*, 2020.

(page 26)

[86] Tiphaine Viard, Matthieu Latapy, and Clémence Magnien. Computing maximal cliques in link streams. *Theoretical Computer Science*, 609:245–252, 2016.

(page 26)

[87] Richard M. Wilson. Decompositions of complete graphs into subgraphs isomorphic to a given graph. In *Proceedings of 5th British Combinatorial Conference*, pages 647–659. Congressus Numerantium XV, 1975.

(page 56)

[88] Junfei Xie, Yan Wan, Jae H Kim, Shengli Fu, and Kamesh Namuduri. A survey and analysis of mobility models for airborne networks. *IEEE Communications Surveys & Tutorials*, 16(3):1221–1238, 2013.

(page 113)

[89] Yuan Yuan and Yuxing Peng. Racetrack: An approximation algorithm for the mobile sink routing problem. In Ioanis Nikolaidis and Kui Wu, editors, *Ad-Hoc, Mobile and Wireless Networks*, pages 135–148, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

(page 81)

[90] Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. The complexity of finding small separators in temporal graphs. In *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 45:1–45:17, 2018.

(page 24, 26, 112)