Impact du changement du protocole de routage dans un réseau

Nina Pelagie Bekono

UNIVERSITÉ CLERMONT-AUVERGNE

École doctorale n° 70 : Sciences Pour l'Ingénieur (SPI)

THÈSE

présentée par

Nina Pelagie BEKONO

pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

Spécialité : Informatique

préparée dans le cadre du programme "mobilité internationale" du LABEX IMobS3 au sein du Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS, CNRS)

Impact du changement du protocole de routage dans un réseau

soutenue publiquement le 13 décembre 2018 devant le jury suivant :

M. Franck Rousseau, Maître de conférences, INP-Ensimag Grenoble,	Examinateur
Mme Dominique GAÏTI, Professeure, Université de Technologie de Troyes,	Rapporteur
M. Hervé RIVANO, Professeur, INSA Lyon,	Rapporteur
M. Alexandre GUITTON, Professeur, Université Clermont Auvergne,	Directeur
Mme Nancy EL RACHKIDY, Maître de conférences, UCA,	Co-encadrante

THESE

Abstract

Routing protocols in networks may change for many reasons : detection of a particular event, planned or unplanned change of topology, mobility of nodes, version obsolescence, etc. As these changes can not be simultaneously detected or taken into account by all nodes of the network, it is necessary to consider the case where some nodes use the initial routing protocol, while others have migrated to the new routing protocol. The work of this thesis deals with the problem of routing loops that may appear in this context, and which considerably degrade the performance of the network. We propose node scheduling solutions to control migration to avoid these loops.

First, we consider the context of static networks and centralized protocols with the particular case of changing metrics. We propose two centralized avoidance solutions : SCH-m (minor improvement of an existing heuristic), and ACH (new contribution), based on the identification of the routing loops in the strongly connected components contained in the union of the two routing protocols. We accelerate the migration of the network by a step-by-step merge operation of the different transitions produced. Second, we evolve towards the distributed protocols while preserving the static context of the network, and consider the particular case of the withdrawal or breakdown of a node. We also propose two solutions : RTH-d (minor improvement of an existing heuristic) and DLF (new contribution for loops of size 2) based on message exchange of nodes for both failure detection and for migration notification. Thirdly, we consider the context of nodes mobility, and study the performance of DLF- k (improved version of DLF which takes into account loops of size less than or equal to k, with $k \geq 2$) on two types of applications : applications with a single mobile node that is the destination, and applications with a group of mobile nodes.

Keywords : network, routing protocols, routing loops, node scheduling, migration, transition.

Resumé

Les protocoles de routage dans les réseaux peuvent être amenés à changer pour de nombreuses raisons : la détection d'un événement particulier, un changement de topologie planifié ou non, la mobilité des nœuds, l'obsolescence de version, etc. Ces changements ne pouvant être simultanément détectés ou pris en compte par tous les nœuds du réseau, il est nécessaire de considérer le cas où certains nœuds utilisent le protocole de routage initial, tandis que d'autres ont migré vers le nouveau protocole de routage. Les travaux de cette thèse portent sur le problème de boucles de routage susceptibles d'apparaître dans ce contexte, et qui dégradent considérablement les performances du réseau. Nous proposons des solutions d'ordonnancement des nœuds, dans le but de contrôler la migration afin d'éviter ces boucles.

Premièrement, nous considérons le contexte des réseaux statiques et des protocoles centralisés avec pour cas particulier le changement de métriques dans le réseau. Nous proposons deux solutions d'évitement des boucles centralisées : SCH-m (amélioration mineure d'un protocole existant), et ACH (nouvelle contribution), basées sur l'identification des boucles de routage dans les composantes connexes que contient l'union des deux protocoles de routage. Nous accélérons la migration du réseau par une opération de fusion étape par étape des différentes transitions produites. Deuxièmement, nous évoluons vers les protocoles distribués en conservant le contexte statique du réseau, et considérons le cas particulier du retrait ou de la panne d'un nœud. Nous proposons également deux solutions : RTH-d (amélioration mineure d'un protocole existant) et DLF (nouvelle contribution traitant les boucles de taille 2) basées sur un échange de messages entre les nœuds tant pour la détection de la panne que pour la notification de la migration. Troisièmement, nous considérons le contexte de mobilité des nœuds, et étudions les performances de DLF-k (version améliorée de DLF qui prend en compte les boucles de taille inférieures ou égales à k, avec $k \geq 2$) sur deux types d'applications : les applications avec un unique nœud mobile qui est la destination, et les applications avec un groupe de nœuds mobiles.

Mots clés : réseau, protocoles de routage, boucles de routage, ordonnancement de nœuds, migration, transition.

Table des matières

Liste des figures

Liste des tableaux

Introduction

Dans les réseaux informatiques, le protocole de routage sert à déterminer le chemin que suivent les paquets d'une source à une destination donnée. Pour ce faire, le protocole de routage utilise une métrique pouvant être le nombre de sauts, le coût, le délai, etc. Cette métrique sert à calculer le meilleur chemin jusqu'à la destination (cf annexe A.1).

La figure 1 montre un réseau de 6 nœuds. Si la source est le nœud 1, la destination est le nœud 6, et la métrique considérée est le coût des liens, alors le meilleur (plus court) chemin de 1 à 6 est : $1 - 2 - 5 - 6$.

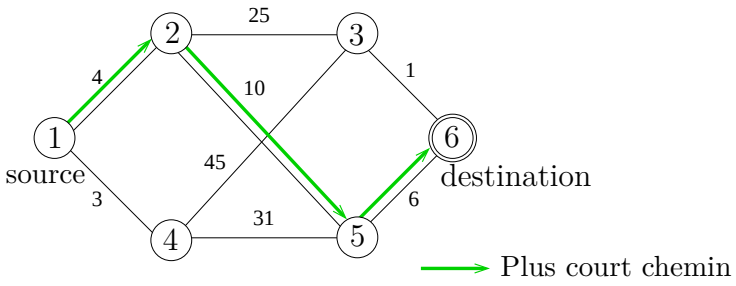

FIGURE 1 – Exemple de routage.

De nombreux travaux de recherche proposent et comparent des protocoles de routage pour tout type de réseaux : réseaux fixes et réseaux mobiles ou non [1, 2, 3, 4, 5, 6].

Dans ces réseaux, le protocole de routage peut être amené à changer pour de nombreuses raisons. En effet, une raison est la détection d'un événement particulier, qui peut amener les nœuds à adopter un comportement différent, se traduisant par un changement de protocole de routage [7, 8, 9, 10]. Une autre raison concerne les réseaux ayant une grande durée de vie, dans lesquels il est parfois nécessaire d'installer une nouvelle version d'un protocole de routage, pour améliorer ses performances ou sa sécurité [11]. Une autre raison est un changement important de métrique ou de topologie qui peut conduire à des routes très différentes.

Un premier exemple résumant ces changements est le cas d'un réseau de véhicules. En situation normale, les nœuds s'échangent des informations périodiquement sur leurs distances respectives ou sur les points d'intérêts locaux (proximité de stations service, de parkings avec des places disponibles, de restaurants, etc.). Lorsqu'un accident est détecté, le mode de fonctionnement des nœuds peut changer : ils doivent collaborer pour informer les nœuds voisins (cf annexe A.3) le plus rapidement possible, afin que les secours puissent être alertés et que les autres conducteurs à proximité puissent être notifiés de l'accident. Le protocole de routage utilisé en situation normale n'est pas nécessairement efficace en situation d'alarme, d'où la nécessité de le renforcer par un protocole assurant un délai minimum.

Un deuxième exemple de changement de protocole de routage est le cas d'un réseau de capteurs déployé dans une forêt pour une surveillance à travers une collecte

d'informations. En situation normale, les informations de collecte sont transmises périodiquement des sources vers la destination avec pour contraintes la garantie de livraison et l'économie d'énergie. Cependant, en cas de détection d'un feu de forêt, les nœuds doivent collaborer de manière efficace pour permettre à cette information d'arriver le plus rapidement possible à la destination. Dans ce cas, le protocole de routage appliqué est différent de celui appliqué en situation normale.

Un troisième exemple est la mise à jour d'un protocole de routage pour lequel un correctif vient d'être publié. Le protocole de routage doit être mis à jour, mais il n'est pas envisageable d'arrêter l'ensemble des nœuds du réseau, de leur appliquer individuellement le correctif, et de relancer les nœuds, car cela interromprait le fonctionnement du réseau pendant toute la procédure. Il est nécessaire que le correctif puisse être appliqué alors que le réseau est en fonctionnement, ce qui crée temporairement une situation dans laquelle certains nœuds fonctionnent avec le protocole initial tandis que d'autres fonctionnent avec le protocole corrigé. Il faut noter que le protocole corrigé peut conduire à des routes très différentes de celles du protocole initial.

Ces changements de protocoles de routage ne peuvent pas être effectués de manière immédiate, étant donné que les nœuds ne sont pas synchronisés. Il est donc important de considérer le cas où certains nœuds utilisent le protocole de routage initial, tandis que les autres nœuds utilisent le nouveau protocole de routage dû au déclenchement d'un événement. Cela peut malheureusement aboutir à l'apparition de boucles de routage lorsqu'aucun contrôle n'est fait sur l'ordre de migration des nœuds vers le nouveau protocole de routage, même si chaque protocole considéré individuellement est sans boucle.

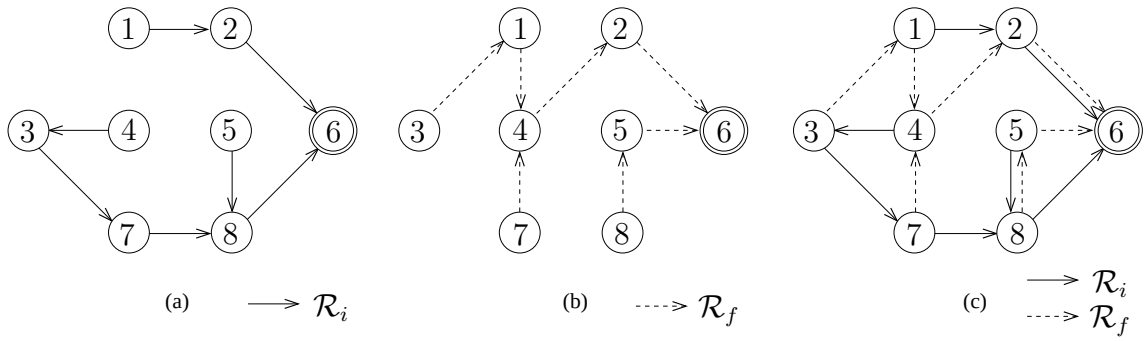

FIGURE 2 – Exemple d'un réseau de 8 nœuds avec pour destination le nœud 6. (a) Protocole de routage initial, (b) protocole de routage final, (c) phase transitoire.

Considérons la figure 2 qui présente l'exemple d'un réseau de 8 nœuds avec pour destination le nœud 6. La figure 2(a) présente le protocole initial noté $\mathcal{R}_i$, la figure 2(b) présente le protocole final noté $\mathcal{R}_f$, et la figure 2(c), quant-à elle, présente l'union des deux protocoles de routage, qui correspond à la phase transitoire durant la migration des nœuds. Durant cette phase transitoire, un nœud peut soit avoir déjà migré et route alors selon le protocole de routage final, soit continuer de router suivant le protocole initial. C'est pourquoi chaque nœud est représenté avec chacun de ses prochains sauts suivant chacun des deux protocoles.

La figure 3 montre trois exemples de cas de figure dans lesquels le réseau peut se trouver durant la phase de transition, si aucun ordre de migration n'est défini, avec pour chacun de ces cas la génération de boucles de routage. Dans la figure 3(a), si le nœud 8 est le premier à avoir migré, il route désormais ses paquets vers le nœud 5.

Cependant le nœud 5 n'ayant pas encore migré, il continue de router ses paquets vers le nœud 8 en attendant sa migration. D'où l'apparition de la boucle $(5, 8, 5)$ identifiée sur la figure. De même, dans la figure 3(b), on suppose que les nœuds $\{1, 2, 3\}$ migrent avant le reste des nœuds du réseau. Cependant, le nœud 4 a pour prochain saut suivant le protocole de routage initial le nœud 3, tandis que les nœuds 3 et 1 ont respectivement pour prochain saut suivant le protocole de routage final les nœuds 1 et 4. Cela aboutit à la boucle de routage $(1, 4, 3, 1)$. De même, dans la figure 3(c), les nœuds $\{1, 2, 7, 8\}$ sont les premiers à migrer. Le nœud 8 qui migre avant le nœud 5 crée la boucle $(5, 8, 5)$, et le nœud 7 qui migre avant les nœuds 3 et 4 génère la boucle $(4, 3, 7, 4)$.

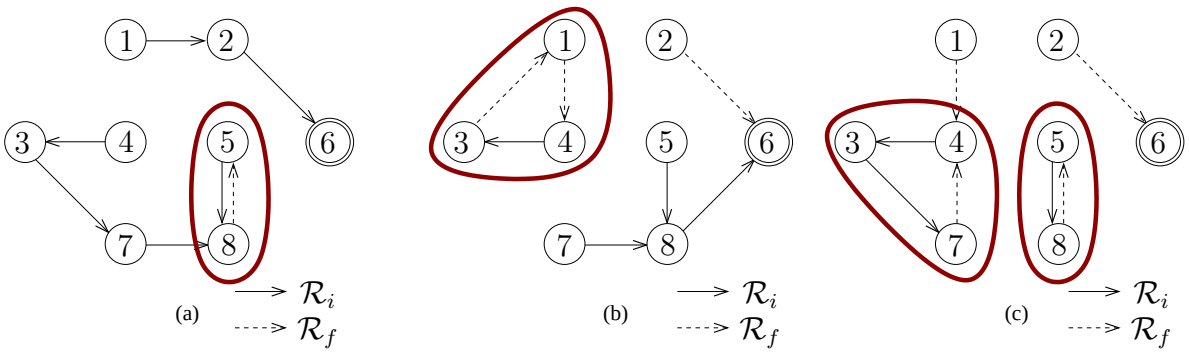

FIGURE 3 – Cas possibles de l'état du réseau durant la phase transitoire générant des boucles de routage. (a) Migration du nœud 8 et génération de la boucle $(5, 8, 5)$, (b) migration des nœuds $\{1, 2, 3\}$ et génération de la boucle $(1, 4, 3, 1)$, (c) migration des nœuds $\{1, 2, 7, 8\}$ et génération des boucles $(5, 8, 5)$ et $(4, 3, 7, 4)$.

Ces trois exemples montrent qu'une absence de définition d'un ordre de migration des nœuds dans un réseau durant un changement de protocole de routage peut entraîner des boucles de routage. Ces boucles de routage ont de lourdes conséquences sur le fonctionnement de réseau. En effet, l'apparition des boucles de routage, même temporaires, peut conduire aux problèmes suivants.

- La consommation de la bande passante : les boucles de routage maintiennent artificiellement les paquets dans le réseau, ce qui augmente la consommation de la bande passante.
- L'augmentation de la congestion : la congestion se produit lorsqu'il y a trop de paquets présents dans le réseau. Les boucles de routage vont conserver les paquets longtemps dans le réseau, et causer la retransmission des paquets considérés comme perdus, ce qui augmente la congestion. Cette congestion entraîne une accumulation des paquets en attente d'émission, qui aboutit à un ralentissement global du réseau.
- L'augmentation des délais : l'accroissement de paquets dans le réseau à cause des boucles de routage crée une augmentation des files d'attente et donc des délais.
- La perte des paquets : les paquets qui subissent les boucles de routage finissent par être détruits.

Quantification des boucles

Pour montrer la fréquence des boucles de routage lors du changement du protocole de routage, nous quantifions dans la suite ces boucles en fonction de la taille du réseau.

Pour ce faire, nous considérons deux types de topologies : les topologies aléatoires et les topologies réelles.

Pour les topologies aléatoires, nous avons généré des graphes connectés de taille variant de 10 à 250 nœuds, déployés aléatoirement sur une surface de dimension $100m$ x $100m$, où deux nœuds sont connectés si leur distance de séparation est inférieure à $20m$, seuil qui correspond à la portée généralement utilisée dans les réseaux de capteurs.

Pour les topologies réelles, nous considérons les topologies *Rocketfuel* [12] qui contiennent 6 jeux de données ayant respectivement pour taille 79, 87, 104, 138, 161, et 315 nœuds.

Nous considérons également trois scénarii de protocoles de routage tous basés sur le plus court chemin (cf annexe A.7), mais qui utilisent des métriques différentes. Dans le scénario 1, le protocole de routage initial $\mathcal{R}_i$ utilise la métrique *hopcount* (nombre de sauts, cf annexe A.11) et le protocole de routage final $\mathcal{R}_f$ utilise une métrique aléatoire (cf annexe A.12), choisie entre 1 et 100 pour chaque lien ($\mathcal{R}_f$ pouvant représenter un protocole basé sur le taux de perte). Dans le scénario 2, les deux protocoles $\mathcal{R}_i$ et $\mathcal{R}_f$ sont basés sur des métriques aléatoires indépendantes (telles que la bande passante pour $\mathcal{R}_i$ et le taux de perte pour $\mathcal{R}_f$). Dans le scénario 3, $\mathcal{R}_i$ est basé sur une métrique aléatoire choisie entre 1 et 100 pour chaque lien, et $\mathcal{R}_f$ utilise un poids corrélé pour chaque lien. Si $w \in [1;100]$ correspond au poids d'un lien pour $\mathcal{R}_i$, le poids de ce même lien pour $\mathcal{R}_f$ est choisi aléatoirement dans l'intervalle $[w-10; w+10]$. Cela correspond au cas où $\mathcal{R}_f$ est une version modifiée de $\mathcal{R}_i$, ou inclus un paramètre additionnel dans le calcul du poids des liens. Les simulations sont répétées 100 fois, et l'intervalle de confiance pour les courbes est de 95%.

Les figures 4 et 5 montrent l'évolution du nombre de boucles (sur une échelle logarithmique) par rapport à la taille du réseau, respectivement sur les topologies aléatoires et *Rocketfuel*.

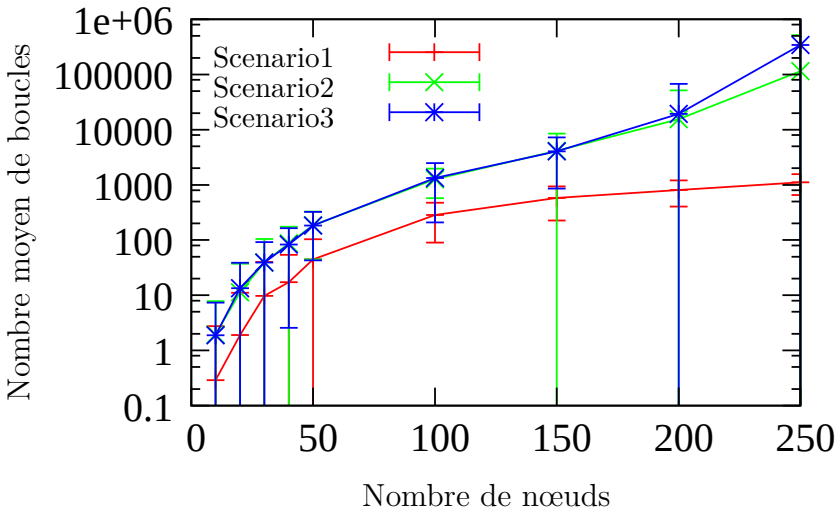

FIGURE 4 – Nombre moyen de boucles en fonction du nombre de nœuds dans le réseau, pour le cas des topologies aléatoires.

Ces deux figures montrent premièrement que quelle que soit la taille de réseau, les différents scénarii de changement de protocole de routage peuvent toujours générer des boucles de routage. Ensuite, elles montrent que le nombre de boucles de routage possibles croît avec la taille du réseau pour toutes les topologies et tous les scénarii. En effet, pour les topologies aléatoires, le scénario 1 a une moyenne de boucles possibles allant de 1 pour les réseaux de 10 nœuds à 1110 pour les réseaux de 250 nœuds.

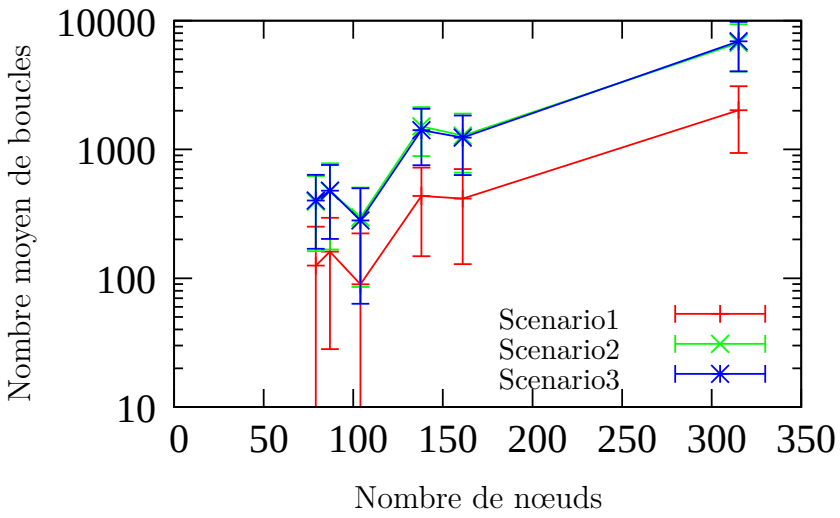

FIGURE 5 – Nombre moyen de boucles en fonction du nombre de nœuds dans le réseau, pour le cas des topologies *Rocketfuel*.

Le scénario 2 a une moyenne de boucles possibles allant de 2 pour les réseaux de 10 nœuds à plus de 110000 pour les réseaux de 250 nœuds. Le scénario 3 a une moyenne de boucles possibles allant de 2 pour les réseaux de 10 nœuds à plus de 340000 pour les réseaux de 250 nœuds. Pour les topologies *Rocketfuel*, le scénario 1 a une moyenne de boucles possibles allant de 125 pour les réseaux de 79 nœuds à plus de 2000 pour les réseaux de 315 nœuds. Le scénario 2 a une moyenne de boucles possibles allant de 392 pour les réseaux de 79 nœuds à plus de 6700 pour les réseaux de 315 nœuds. Le scénario 3 a une moyenne de boucles possibles allant de 402 pour les réseaux de 79 nœuds à plus de 6900 pour les réseaux de 315 nœuds. Plus les métriques sont indépendantes, moins les chemins calculés par les différents protocoles sont corrélés et plus la possibilité d'apparition des boucles de routage est augmentée. Nous constatons également que, pour le cas du scénario 3, même si les métriques sont fortement corrélées, pouvant ainsi conduire à un routage de paquets suivant des chemins similaires selon $\mathcal{R}_i$ et $\mathcal{R}_f$, le nombre de boucles de routage reste élevé sur des topologies de grande taille. Cela s'explique par le fait que sur ces topologies, malgré les poids des liens corrélés, les chemins sont quant à eux faiblement corrélés.

Des résultats similaires sont donnés dans [13] où les auteurs ont étudié avec le même paramétrage l'occurrence d'apparition des boucles.

Applications

Trois grandes classes d'applications impliquant les boucles de routage peuvent être définies, à savoir : les changements planifiés, les changements non planifiés, et la mobilité (planifiée et non planifiée). Nous présentons dans la suite chaque classe d'application.

Dans un réseau, le changement de protocole de routage peut parfois être planifié à l'avance. En effet, l'administrateur d'un réseau peut décider d'ajouter un ou plusieurs routeurs, de créer de nouvelles connexions entre routeurs existants, ou de supprimer certains routeurs du réseau (par exemple pour une opération de maintenance). Cela aboutit à des modifications de topologies qui entraînent le recalcul des routes vers la destination choisie, aboutissant ainsi à un nouveau protocole de routage. De même, le choix d'une nouvelle métrique au détriment de celle actuellement en fonction dans le réseau (choisir de considérer par exemple le délai à la place du taux de perte), entraîne

une modification du poids des liens entre les nœuds. Ce changement de métrique génère de nouvelles routes qui peuvent être très différentes de celles empruntées avant l'application de la modification. De même, la mise à jour du protocole de routage apportée dans un réseau de grande durée de vie par exemple est un changement planifié.

Cependant, dans un réseau, les boucles de routage peuvent également apparaître à la suite de changements non planifiés. En effet, la survenue d'une panne (panne d'un nœud ou d'un lien) dans un réseau entraîne le recalcul des routes de chaque source vers les destinations, recalcul aboutissant à de nouveaux chemins. De même, toutes les activités de surveillance (la surveillance environnementale telle que la surveillance des forêts, des rivières ou encore des volcans, la surveillance médicale pour des personnes à domicile, etc.) peuvent aboutir à un nouveau protocole de routage en cas de détection d'un événement particulier tel qu'un feu de forêt, une éruption volcanique, un vol dans un bâtiment intelligent, un patient en situation critique, etc.

Les boucles de routage peuvent également apparaître dans les réseaux mobiles où une partie des nœuds se déplace. Le fait que les nœuds soient mobiles entraîne une modification des chemins à emprunter pour atteindre les destinations. Nous distinguons la mobilité planifiée où les nœuds ont un itinéraire connu d'avance, et la mobilité non planifiée. Pour la mobilité planifiée, nous pouvons citer l'exemple du réseau de transport de la ville de Clermont-Ferrand *(t2c)*, ou encore l'exemple de la voiture intelligente sans conducteur, VIPA (Véhicule Individuel Public Autonome) [14], développée par le laboratoire d'excellence ImobS3 et ses partenaires. Pour la mobilité non planifiée, nous pouvons citer l'exemple des réseaux auto-organisés [15].

Solutions existantes

Le problème d'évitement des boucles de routage durant le changement de protocoles de routage est un problème étudié récemment. Dans la suite, nous faisons un survol des travaux existants, qui seront repris en détail ultérieurement.

Dans [1, 2], les nœuds sont synchronisés et les paquets sont routés suivant deux périodes p_1 et p_2 qui s'alternent. Durant la période p_1, tous les nœuds routent suivant le protocole de routage $\mathcal{R}_1$, et durant p_2, ils routent suivant le protocole de routage $\mathcal{R}_2$. les boucles peuvent apparaître si les nœuds du réseau se désynchronisent ou si la décision de router un paquet suivant $\mathcal{R}_1$ ou $\mathcal{R}_2$ est prise de manière locale par un des nœuds, pour profiter du temps libéré dans une période par exemple. Dans [1], les auteurs étudient les paires de protocoles et identifient trois catégories :

- les protocoles de routage compatibles, qui ne conduisent jamais à des boucles de routage lorsqu'ils sont utilisés ensemble,
- les protocoles de routage retardables, où les nœuds peuvent éviter les boucles en utilisant la connaissance des fonctions de distance des deux protocoles,
- les protocoles de routage combinés, où les fonctions de distance des deux protocoles ne sont pas connues ou sont difficiles à calculer localement par les nœuds.

Les deux dernières catégories (retardables et combinés) profitant de l'alternance entre $\mathcal{R}_1$ et $\mathcal{R}_2$ car elles requièrent que certains nœuds routent indéfiniment suivant un seul protocole. Dans [2], les auteurs définissent une approche probabiliste d'évitement des boucles où des nœuds susceptibles d'être impliqués dans une boucle de routage choisissent aléatoirement de router ou non un paquet. La migration vers $\mathcal{R}_2$ pour ces deux solutions n'est pas définitive.

Dans [9, 16, 17], les auteurs montrent que le changement de topologie planifié ou non peut conduire à des boucles de routage. Dans [9], les auteurs montrent que ces

boucles peuvent être évitées si les nœuds appliquent les mises à jour de routage suivant un ordre donné. Dans [16, 17], les auteurs montrent que les boucles qui apparaissent après des changements topologiques peuvent être évitées en appliquant une séquence de mises à jour de topologie. Dans [16], ils considèrent la modification d'un unique lien, tandis que dans [17], ils considèrent la modification des liens d'un unique nœud. Dans [16, 17], les auteurs cherchent à minimiser le nombre de mises à jour de décisions de routage qui garantissent la convergence d'un unique protocole de routage. Les trois propositions [9, 16, 17] ont pour caractéristique commune que le protocole de routage ne change pas contrairement à la topologie qui subit des modifications.

Dans [8, 10], les auteurs montrent que l'ordonnancement des mises à jour de routage conduit à un surcoût en termes de messages, ce qui augmente les délais. Ils proposent d'éviter les boucles en exploitant l'existence d'une table de routage par interface de sortie des routeurs. Les messages qui arrivent via des interfaces inattendues sont supprimés, car ceci traduit une différence de vue du réseau entre le nœud et ses voisins. Lorsque tous les nœuds ont la même vue du réseau, le protocole produit alors des routes sans boucles.

Objectifs

Nous considérons dans cette thèse que l'on passe d'un protocole initial à un protocole final de manière définitive. Organiser cette migration est une tâche complexe et critique.

L'objectif premier est d'éviter complètement les boucles de routage avec pour hypothèse que chacun des protocoles pris individuellement est sans boucle, tout en réduisant au maximum la durée transitoire.

L'objectif second est de proposer des algorithmes d'évitement de boucles adaptés à des situations particulières :

- lorsque le réseau est statique, mais la métrique des poids des liens est changée de manière planifiée ou non,
- lorsque le réseau est statique mais qu'il y a un retrait programmé de nœud ou de lien, ou une panne d'un nœud ou d'un lien qui survient dans le réseau,
- lorsque le réseau est mobile : faire l'étude de différents scénarii possibles, lorsque la destination est mobile, une partie des nœuds du réseau est mobile, ou encore la totalité des nœuds du réseau est mobile.

Plan de la thèse

Cette thèse est organisée en deux parties.

Dans la première partie, constituée des chapitres 1 et 2, nous faisons un état de l'art sur le routage dans les réseaux et les travaux existants traitant du problème de boucles de routage.

Dans le chapitre 1, nous faisons un tour d'horizon des protocoles de routage les plus connus de la littérature. Tout d'abord, nous nous intéressons aux protocoles de routage intérieur pour les réseaux filaires, que nous classons en deux groupes : les protocoles à vecteur de distance, et les protocoles à état de liens. Ensuite, nous nous intéressons aux protocoles de routage pour les réseaux ad-hoc classés également en deux groupes : les protocoles proactifs et les protocoles réactifs. Nous étudions à chaque

fois, le fonctionnement des protocoles, les possibilités de boucles, et les mécanismes de gestion des boucles de routage qu'ils implémentent en cas de modification de topologie.

Dans le chapitre 2, nous étudions les solutions proposées pour passer d'un protocole de routage initial à un protocole de routage final. Nous les classifions en deux catégories, à savoir : les solutions basées uniquement sur le routage final, et les solutions basées sur l'augmentation du poids des liens. Nous nous attelons à les décrire en détails.

Dans la deuxième partie, constituée des chapitres 3, 4, et 5, nous proposons des heuristiques au problème de boucles de routage en cas de changement de protocoles, en considérant différents contextes.

Dans le chapitre 3, nous considérons que l'environnement d'application est centralisé et statique, et nous nous intéressons aux changements planifiés comme type d'application. Nous montrons dans ce chapitre qu'en se concentrant sur les composantes fortement connexes induites par l'union des deux protocoles de routage, un ordre de migration des nœuds sans génération de boucles peut être défini. Nous proposons ainsi deux heuristiques centralisées : *Strongly connected Component Heuristic with merged steps* (SCH-m) et *Avoiding Cycle Heuristic* (ACH). Ces deux heuristiques construisent des étapes pendant lesquelles certains nœuds peuvent migrer ensemble sans générer de boucles transitoires de routage. L'objectif principal étant l'accélération de la migration des nœuds dans le réseau, SCH-m préconise la fusion des étapes des différentes transitions calculées par destination dans un réseau multi-destinations, tandis qu'ACH recherche la maximisation du nombre de nœuds dans les étapes en plus de la fusion des étapes.

Dans le chapitre 4, nous considérons que l'environnement d'application reste statique mais qu'il n'existe pas d'entité centrale chargée de calculer l'ordre de basculement des nœuds dans le réseau. Cet environnement est par conséquent distribué. Nous nous intéressons aux boucles de routage qui apparaissent lors de changements non planifiés, particulièrement lors de la survenue de pannes dans un réseau. Nous proposons tout d'abord l'heuristique distribuée *Routing Tree Heuristic-Distributed version* (RTH-d), qui est une version distribuée d'une heuristique se basant uniquement sur le protocole de routage final. Ensuite, nous proposons l'heuristique distribuée *Distributed Loop-Free heuristic* (DLF), qui définit un mécanisme de migration des nœuds basé sur l'hypothèse selon laquelle la panne d'un nœud dans un réseau génère des boucles de taille 2 uniquement.

Dans le chapitre 5, nous considérons que nous sommes dans un environnement dynamique car les nœuds se déplacent dans une zone géographique. Nous étudions dans un premier temps le problème de boucles de routage dans différentes applications mobiles. Nous proposons ensuite un protocole distribué DLF-k, une adaptation de DLF qui est une généralisation à des boucles de taille inférieure ou égale à un seuil k, $k \geq 2$ fixé. Nous considérons ensuite deux types d'applications de mobilité : les applications avec un unique nœud mobile (la destination), et les applications avec un groupe de nœuds mobiles. Nous quantifions enfin les boucles, et étudions les performances de DLF-k dans chacune de ces applications.

Nous concluons enfin ce travail de thèse en résumant nos différentes contributions, et en donnant les perspectives de nos travaux.

Première partie

Etat de l'art

Chapitre 1

Protocoles de routage existants

Sommaire

Introduction

Les protocoles de routage ont pour rôle d'établir et de maintenir des routes entre les nœuds qui souhaitent communiquer. Ces protocoles diffèrent selon qu'ils opèrent sur des topologies filaires et statiques, ou sur des topologies sans fil et sans infrastructure fixe (ad-hoc), amenées à évoluer. Ils ont pour fonctionnalités : la découverte des réseaux distants, l'actualisation des informations de routage, le choix du meilleur chemin vers les destinations, et la capacité à trouver un nouveau chemin si le chemin actuel n'est plus disponible.

Les protocoles de routage filaires peuvent être classés en deux grands groupes à savoir : les protocoles de routage intérieur à l'instar de OSPF *(Open Shortest Path First)* [18], et les protocoles de routage extérieur qui eux fonctionnent à une échelle plus grande, entre systèmes autonomes, dont BGP *(Border Gateway Protocol)* [19]. Tandis que les protocoles pour les réseaux ad-hoc se composent de deux principaux groupes à savoir : les protocoles proactifs à l'instar de OLSR (*Optimized Link State Routing Protocol*) [20], et les protocoles réactifs à l'instar de AODV (*Ad hoc On-Demand Distance Vector*) [21]. Dans ce chapitre, nous présentons tout d'abord quelques notions essentielles au routage. Puis, nous faisons d'une part un tour d'horizon des protocoles de routage intérieur les plus connus de la littérature, et d'autre part, un tour d'horizon des protocoles de routage ad-hoc les plus connus de la littérature.

1.1 Formalisation et notations

Dans cette section, nous présentons les notions liées au routage, et qui sont régulièrement employés dans la suite du manuscrit.

1.1.1 Routage

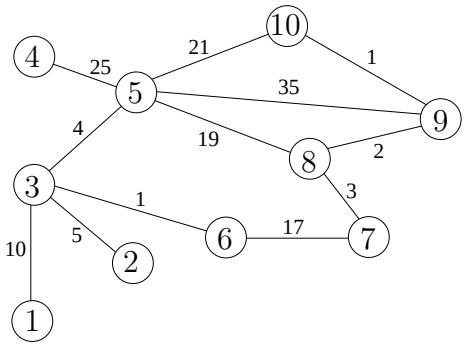

FIGURE 1.1 – Exemple de graphe.

Le routage est le mécanisme par lequel des chemins sont sélectionnés dans un réseau pour acheminer les données d'un expéditeur vers une destination donnée. Le choix du chemin repose sur une métrique de sélection (cf annexe A.6) qui peut être le nombre de sauts, la capacité des liens (bande passante), le délai, la charge, la fiabilité, ou encore le coût d'acheminement de l'information à la destination. Le routage calcule ainsi pour chaque nœud, le meilleur voisin pour que le paquet atteigne la destination. On parle alors de prochain saut selon le routage.

Soit un réseau représenté par un graphe $G = (V, E)$. Nous noterons de manière générale $\mathcal{R}^d$ le routage vers la destination d avec $d \in V$, et de manière spécifique

$\mathcal{R}^d(n)$, le prochain saut du nœud n vers la destination d avec $n, d \in V$. Formellement, nous avons :

$$\begin{cases} \forall n \neq d, \mathcal{R}^d(n) & \in & V \\ \mathcal{R}^d(d) & = & d \end{cases}$$

Considérons la figure 1.1 et supposons que la destination est le nœud 10.

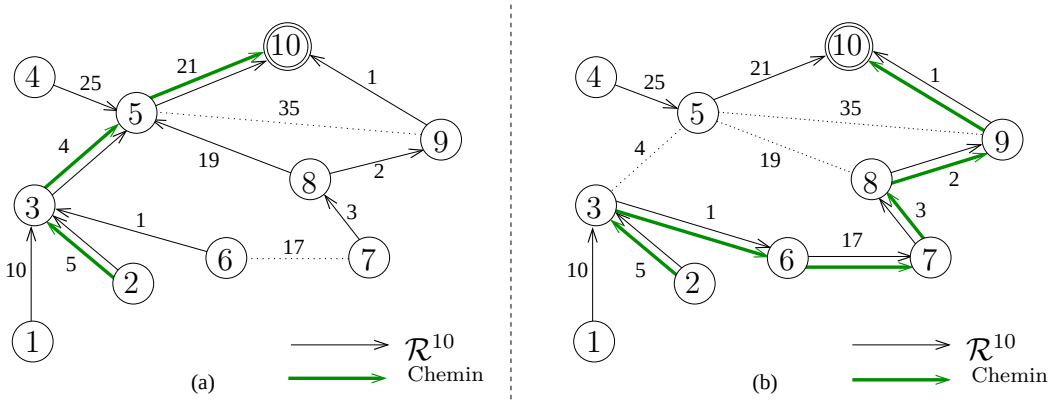

FIGURE 1.2 – Exemples de routage vers la destination 10 dans le réseau de la figure 1.1 suivant une métrique donnée : (a) le nombre de sauts, (b) le coût des liens.

La figure 1.2 montre l'exemple du calcul du prochain saut de chaque nœud lorsqu'on considère le nœud 10 comme destination en prenant deux métriques de sélection : le nombre de sauts *(hopcount)*, sur la figure 1.2(a), et le coût des liens, sur la figure 1.2(b). La figure 1.2 montre clairement que les chemins d'une source à une destination donnée varient selon la métrique choisie. En effet, si nous considérons le nœud 2 comme source, et le nombre de sauts comme métrique, alors le chemin vers la destination 10 est $2-3-5-10$ (figure 1.2(a)). Si la métrique considérée est le coût des liens, alors le chemin de 2 vers 10 est $2-3-6-7-8-9-10$ (figure 1.2(b)). Lorsqu'on considère le nombre de sauts comme métrique, nous constatons que le nœud 8 a deux prochains sauts possible vers la destination 10 (figure 1.2).

1.1.2 Acheminement

L'acheminement est le mécanisme par lequel les paquets sont conduits à destination, en fonction du routage. Tandis que le routage fournit le meilleur chemin d'une source à une destination après application d'un critère de sélection, l'acheminement se charge de conduire les paquets selon ce meilleur chemin obtenu.

1.1.3 Protocole de routage

Un protocole de routage est un algorithme qui fournit les mécanismes de routage à un routeur, lui permettant ainsi de partager des informations de routage avec les autres routeurs auxquels il est connecté, de calculer le meilleur itinéraire pour router les données, et de construire sa table de routage. Nous en étudions des exemples dans le chapitre 2, comme RIP *(Routing Information Protocol)* ou OSPF *(Open Shortest Path First)*.

Etant donné que nous traitons de la problématique de boucles de routage lorsque le protocole de routage change, nous noterons $\mathcal{R}_i^d$ le protocole de routage initial vers la destination d c'est-à-dire, le protocole avant que la migration ne soit effectuée, et

$\mathcal{R}_f^d$ le protocole de routage final vers la destination d, qui est celui après la migration. Ainsi $\mathcal{R}_i^d(n)$ (respectivement $\mathcal{R}_f^d(n)$) est le prochain saut du nœud n suivant le routage initial (respectivement suivant le routage final) vers la destination d .

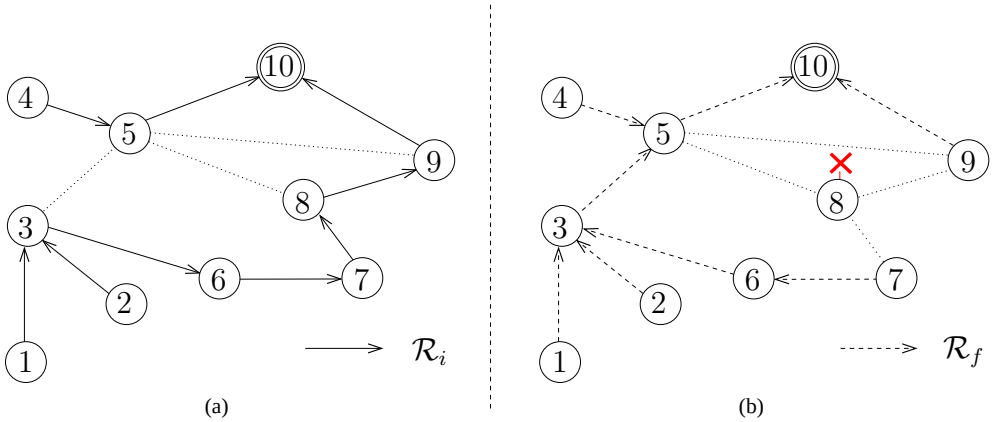

FIGURE 1.3 – Protocole de routage avant et après la survenue de la panne du nœud 8 dans le réseau de la figure 1.1 avec comme métrique le coût : (a) protocole de routage initial, (b) protocole de routage final.

La figure 1.3 montre l'exemple d'un protocole de routage initial et d'un protocole de routage final lorsqu'il y a la survenue de panne du nœud 8 dans le réseau de la figure 1.1. La figure 1.3(a) montre le protocole de routage initial qui est récapitulé par le tableau de la table 1.1. On y voit que les paquets provenant des nœuds $\{1, 2, 3, 6, 7\}$ passent à terme par le nœud 8 pour atteindre la destination 10. Lorsque le nœud 8 tombe en panne comme le montre la figure 1.3(b), ces nœuds changent d'itinéraire et leurs paquets passent désormais par le nœud 5 pour atteindre la destination 10. Le tableau 1.2 donne pour chaque nœud son prochain saut selon $\mathcal{R}_f$.

n	1	2	3	4	5	6	7	8	9	10
$\mathcal{R}_i^{10}(n)$	3	3	6	5	10	7	8	9	10	10

TABLE 1.1 – Routage initial vers la destination 10 du réseau de la figure 1.1 avant la survenue de la panne du nœud 8.

n	1	2	3	4	5	6	7	8	9	10
$\mathcal{R}_f^{10}(n)$	3	3	5	5	10	3	6	-	10	10

TABLE 1.2 – Routage final vers la destination 10 du réseau de la figure 1.1 après la survenue de la panne du nœud 8.

1.1.4 Validité

Un protocole de routage est dit valide si à partir d'un nœud quelconque du réseau et en suivant le prochain saut défini, on atteint toujours la destination. Formellement, soit le graphe $G = (V, E)$ représentant le réseau, $d \in V$ est la destination. $\mathcal{R}^d$ est valide ssi $\forall n \in V$, $\mathcal{R}^*(n) = d$ avec $\mathcal{R}^* : \mathcal{R}(\mathcal{R}(\mathcal{R}...(n)))$.

Le protocole de routage initial détaillé par le tableau 1.1 est un exemple de routage valide. Si par contre nous considérons le protocole de routage présenté dans le

tableau 1.3, ce dernier n'est pas valide car partant des nœuds $\{1, 2, 3, 6\}$, la destination 10 n'est jamais atteinte, à cause d'une boucle entre 3 et 6.

n	1	2	3	4	5	6	7	8	9	10
$\mathcal{R}_f^{10}(n)$	3	3	6	5	10	3	6	5	10	10

TABLE 1.3 – Exemple de protocole de routage non valide.

1.1.5 Routage *monopath/multipath*

Le routage *monopath* désigne un routage dans lequel un nœud ne peut avoir qu'un seul prochain saut, tandis que dans un routage *multipath*, un nœud peut avoir plusieurs prochains sauts (par exemple en cas d'égalité des coûts).

Considérons l'exemple de la figure 1.2(a). Avec pour métrique le nombre de sauts, le nœud 8 peut avoir comme prochain saut le nœud 5 ou le nœud 9. Dans le cas d'un routage *monopath*, l'un de ces deux nœuds sera choisi comme prochain saut, tandis que dans le cas d'un routage *multipath*, les deux seront les prochains sauts du nœud 8 vers la destination 10.

Le routage *multipath* a l'avantage de favoriser l'équilibrage de charges, et la tolérance aux pannes. Dans la suite du manuscrit, c'est le routage *monopath* qui est généralement utilisé. Mais l'extension au routage *multipath* est tout à fait possible.

1.1.6 Convergence

La convergence désigne l'état dans lequel tous les routeurs ont les mêmes informations topologiques du réseau dans lequel ils opèrent. L'état de convergence du réseau est atteint par collecte et échange d'informations.

1.2 Protocoles de routage intérieur pour les réseaux filaires

Les protocoles de routage intérieur pour les réseaux filaires, encore appelés protocoles internes, gèrent le routage au sein d'un système autonome (par exemple le réseau d'une entreprise, d'une organisation, etc.). Ils établissent des routes suivant un critère donné, le nombre minimal de sauts par exemple, entre les nœuds du réseau. En cas de modification de la topologie, comme la suppression d'un nœud par exemple, ils assurent la convergence du routage. La figure 1.4 distingue deux sous-groupes de protocoles pour l'établissement des routes : les protocoles à vecteur de distance, et les protocoles à état de liens.

1.2.1 Protocoles à vecteur de distance

Les protocoles de routage à vecteur de distance annoncent les routes au travers d'une liste de distances estimées à chaque destination. Cette distance peut représenter une métrique comme par exemple le nombre de sauts, le coût, la bande passante ou le délai. Les routeurs ne connaissent pas le chemin complet vers une destination, mais

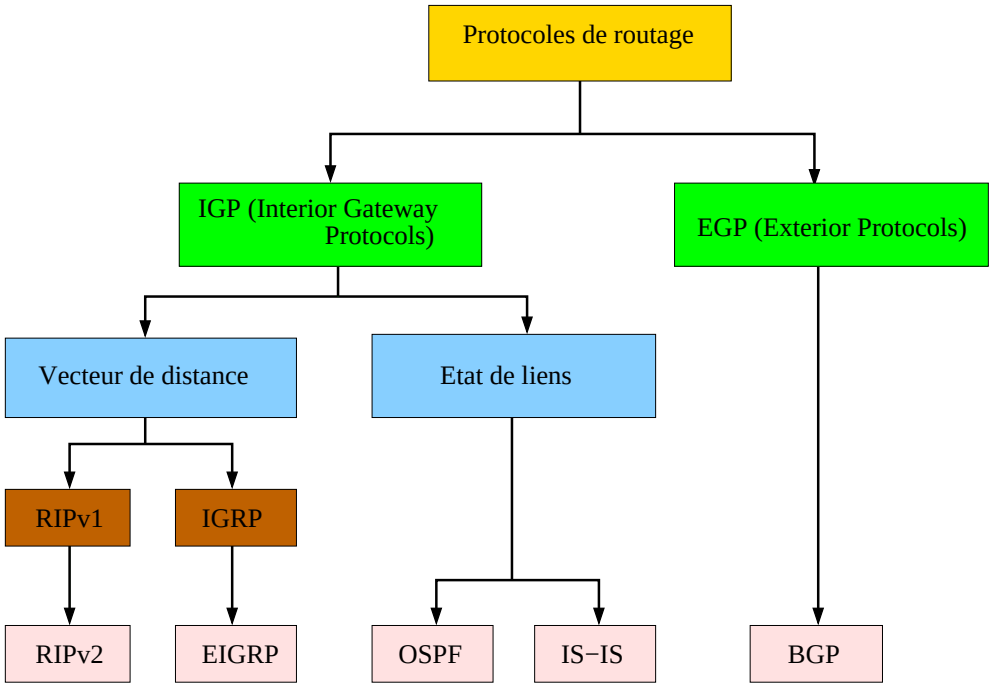

FIGURE 1.4 – Classification des protocoles de routage.

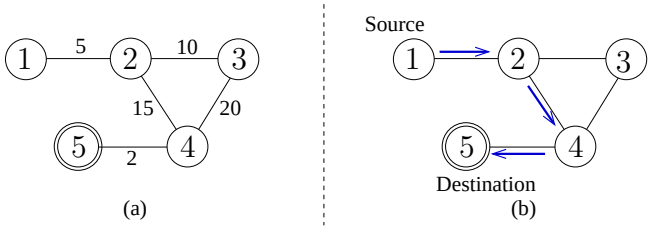

FIGURE 1.5 – Exemple de routage à vecteur de distance : (a) un graphe de 6 nœuds, (b) le routage sur ce graphe.

uniquement l'estimation de la distance d'éloignement de cette destination sur une interface donnée. Un vecteur de distance est un ensemble de triplets (destination, coût, prochain saut).

La figure 1.5 montre un protocole à vecteur de distance avec pour métrique le nombre de sauts. Le chemin $1-2-4-5$ est construit comme meilleur chemin du nœud source 1 au nœud destination 5. Les mises à jour des tables de routage sont effectuées périodiquement par voisinage : chaque routeur diffuse à ses voisins son vecteur de distance vers les destinations qu'il connaît. Lorsqu'un routeur voisin reçoit ce vecteur, il compare chaque entrée avec ses propres distances et met à jour sa propre table de routage si la distance reçue correspond à un plus court chemin ou si la destination reçue était encore inconnue.

Prenons l'exemple de la figure 1.5. La construction de la table de routage du nœud 1 est faite comme suit. Chaque nœud diffuse sa table de vecteurs. Le nœud 1 diffuse l'unique vecteur qu'il a : $(2, 1, 2)$ à son voisin 2. Le nœud 2 diffuse également la liste de ses vecteurs de distance à savoir : $\{(1, 1, 1),(3, 1, 3),(4, 1, 4)\}$ à ses trois voisins 1, 3, et 4. Lorsque le nœud 1 reçoit ce message de 2, n'ayant pas encore d'entrées pour les destinations 3 et 4, il ajoute une entrée pour chacune d'elle en incrémentant le nombre de sauts qui devient alors 2 : $(3, 2, 2)$ et $(4, 2, 2)$. Le nœud 2 reçoit un message de son voisin 4 qui diffuse les vecteurs $\{(2, 1, 2),(3, 1, 3), (5, 1, 5)\}$. Le nœud 2 ajoute ainsi l'entrée $(5, 2, 4)$ à sa liste et la diffuse à nouveau à 1. Le nœud 1 ne disposant pas encore d'une entrée pour la destination 5, ajoute cette dernière après réception du nouveau message en incrémentant le nombre de sauts : $(5, 3, 2)$. Le nœud 1 obtient en somme les différentes entrées suivantes : $\{(2, 1, 2), (3, 2, 2), (4, 2, 2), (5, 3, 2)\}$.

Les protocoles de routage à vecteur de distance nécessitent plus de bande passante que de puissance de calcul. C'est pourquoi le routage converge lentement mais est simple à implémenter. Ils sont particulièrement sensibles aux boucles de routage.

Techniques de gestion des boucles

Dû à une convergence lente qui vient du fait que les tables de routage sont mises à jour avec une grande périodicité, des routeurs voisins ne partageant plus la même topologie du réseau peuvent propager mutuellement des informations contradictoires et produire ainsi des boucles de routage. Les techniques suivantes servent à éviter ces boucles :

- **Définition d'une valeur maximale :** cette technique consiste à définir une valeur maximale de distance au delà de laquelle toute destination est considérée inaccessible. RIP que nous présenterons dans la suite définit par exemple une valeur maximale de 15 sauts.
- ***Split horizon* :** cette technique consiste à dire qu'une information de routage reçue sur une interface n'est jamais retransmise sur celle-ci. L'objectif est de ne pas retransmettre les informations de routage vers l'endroit dont elles proviennent, afin de réduire la prise en compte d'informations contradictoires.
- ***Poison reverse* :** c'est une technique d'empoisonnement des routes et une amélioration de la technique *split horizon*. Elle consiste à appliquer à une route non disponible une métrique maximale qui est le nombre de sauts maximum autorisé par le protocole utilisé plus un, et à diffuser prioritairement cette information.
- **Compteur de retenue *(holddown)* :** c'est une technique qui consiste à différer la mise à jour d'une table de routage dans le cas où, pour une destination devenue inaccessible, on recevrait une mise à jour indiquant le contraire mais avec une métrique moins bonne que celle initialement apprise.

— **Mises à jour déclenchées *(holddown timer)* :** C'est un mécanisme qui permet d'associer à une route empoisonnée un compteur *(Hold Down Timer)* qui va indiquer aux routeurs voisins de ne pas accepter des mises à jour de la part de la route empoisonnée tant que le compteur n'a pas expiré. A l'expiration du compteur, les routeurs recevront des messages de déclenchement leur indiquant qu'ils peuvent accepter des mises à jour de la part de la route ressuscitée et accepter de mettre à zéro le compteur *holddown*.

Nous présentons dans la suite les protocoles RIP, IGRP et EIGRP qui sont des protocoles à vecteur de distance.

RIP : *Routing Information Protocol*

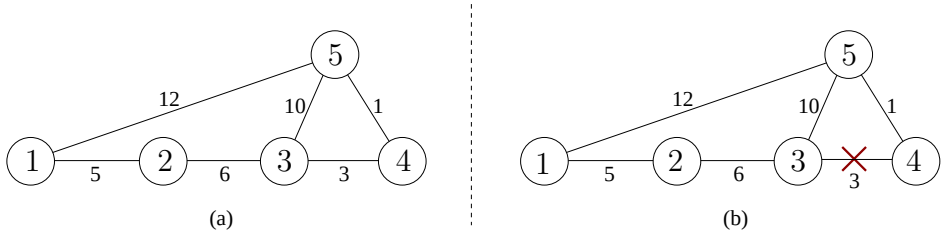

FIGURE 1.6 – Exemple de graphe pour illustrer le fonctionnement du routage à vecteur de distance : (a) un graphe de 6 nœuds, (b) suppression d'un lien.

RIP [22] est un protocole de routage s'appuyant sur l'algorithme de Bellman-Ford [23, 24] pour déterminer les routes de manière décentralisée. RIP fonctionne de la manière suivante : lors de l'initialisation d'un routeur puis périodiquement, celui-ci détermine l'adresse réseau de ses interfaces puis envoie sur chacune d'elles une demande d'informations des routeurs voisins. Lors de la réception d'une demande d'informations, un routeur envoie son vecteur de distances. Lorsqu'un routeur reçoit une réponse, il met à jour sa table. Quatre cas peuvent se présenter :

— Pour une nouvelle route, le routeur incrémente la distance vers la destination concernée, vérifie que celle-ci est strictement inférieure à 15 et diffuse immédiatement le vecteur de distance correspondant.
— Pour une route existante dont la distance reçue plus 1 est inférieure à la distance connue, la table est mise à jour. La nouvelle distance et, éventuellement, l'adresse du routeur voisin si elles sont différentes, sont intégrées dans la table.
— Pour une route existante, mais avec une distance supérieure ou égale à la distance connue, la table est mise à jour si la nouvelle distance est émise par le même routeur voisin que précédemment.
— Si le routeur reçoit une route dont la distance est supérieure à celle déjà connue d'un autre voisin, RIP l'ignore.

Considérons l'exemple de la figure 1.6, et la table des vecteurs de distance représentée sur le tableau 1.4. À la première étape, chaque nœud connaît uniquement comme destinations ses voisins et la distance qui les sépare. Ensuite, chaque nœud transmet à ses voisins sa table de voisinage. Considérons le nœud 1 qui reçoit ainsi des messages de 2 et de 3, et supposons qu'il traite en premier le message de 2. Il ajoute dans sa liste de vecteurs le vecteur $(3, 11, 2)$ puisqu'il n'a pas encore d'entrée pour la destination 3. Ensuite, il traite le message de 5 : il ignore la destination 1 puisqu'il s'agit de lui même. La destination 3 lui donne un coût de $12 + 10 = 22$ supérieur au coût 11 de l'entrée de

Nœud	Destination	Coût	Prochain saut
1	2	5	2
	5	12	5
2	1	5	1
	3	6	3
3	2	6	2
	4	3	4
	5	10	5
4	3	3	3
	5	1	5
5	1	12	1
	3	10	3
	4	1	4

TABLE 1.4 – Première étape du fonctionnement de RIP sur l'exemple de la figure 1.6(a)

Noeud	Destination	Coût	Prochain saut
1	2	5	2
	3	11	2
	4	13	5
	5	12	5
2	1	5	1
	3	6	3
	4	9	3
	5	10	3
3	1	11	2
	2	6	2
	4	3	4
	5	4	4
4	1	13	5
	2	9	3
	3	3	3
	5	1	5
5	1	12	1
	2	10	4
	3	4	4
	4	1	4

TABLE 1.5 – Le tableau après la convergence avec le protocole RIP sur l'exemple de la figure 1.6(a).

sa table associée à la destination 3 *via* 2, donc il ignore également cette information. Le nœud 5 lui fournit également la destination 4, et étant donné qu'il n'existe encore aucune information dans sa liste de vecteurs pour 5, il ajoute l'entrée $(4, 13, 5)$. Chaque nœud applique le même principe jusqu'à l'obtention de la table 1.5 qui montre l'état de la liste de vecteurs de chaque nœud lorsque tous les nœuds ont convergé.

Considérons la modification qui apparaît dans la figure 1.6(b) représentée par la perte du lien $(3, 4)$. Cette information est diffusée et chacun des nœuds va mettre à jour son vecteur de distances. Considérons le nœud 2 en guise d'exemple. Le nœud 3 lui diffuse une route vers la destination 4 et le nouveau coût de 2 vers 4 *via* 3 est de 17, coût supérieur au coût actuel 9. Le nœud 2 met donc à jour l'entrée associée à la destination 4 avec pour nouvelle entrée $(4, 17, 3)$. La liste finale de distances pour chaque nœud est dans le tableau 1.6.

Noeud	**Destination**	**Coût**	**Prochain saut**
1	2	5	2
	3	11	2
	4	13	5
	5	12	5
2	1	5	1
	3	6	3
	4	17	5
	5	16	3
3	1	11	2
	2	6	2
	4	11	5
	5	10	5
4	1	13	5
	2	17	5
	3	11	5
	5	1	5
5	1	12	1
	2	16	3
	3	10	3
	4	1	4

TABLE 1.6 – Le tableau après la convergence avec le processus RIP sur l'exemple de la figure 1.6(b). Les routes modifiées sont marquées par la couleur grise.

Une route est retirée d'une table RIP dans deux cas : (1) un routeur directement connecté devient inaccessible, (2) un routeur distant met plus de 30 secondes sans répondre et RIP considère qu'il est tombé en panne. Dans ce cas, une diffusion d'une distance infinie est faite et s'en suit la suppression de la route.

RIP implémente plusieurs mécanismes pour éviter les boucles de routage : la définition d'une valeur maximale du nombre de sauts à 15, le *split horizon*, le *poison reverse* et le *holddown timer*.

IGRP/EIGRP : *Interior Gateway Routing Protocol / Enhanced Interior Gateway Routing Protocol*

IGRP : IGRP *(Interior Gateway Routing Protocol)* [25] est un protocole à vecteur de distance qui peut prendre en compte plusieurs métriques de distance à savoir : la bande passante, le délai, la charge et la fiabilité. Il passe mieux à l'échelle que RIP car il prend en charge un nombre maximal de sauts de 255 (par défaut 100), et annonce les routes toutes les 90 secondes, utilisant ainsi moins de bande passante. Cependant, IGRP converge plus lentement que RIP.

En terme de gestion des boucles, IGRP implémente la définition d'une valeur maximale du nombre de sauts qu'il fixe à 100. Il implémente également le *split horizon*, le *poison reverse* et le *holddown timer*. IGRP, devenu obsolète, a laissé place à son successeur EIGRP.

EIGRP : EIGRP *(Enhanced Interior Gateway Routing Protocol)* [26] est un protocole à vecteur de distance proposé à la suite de IGRP. Il fonctionne comme suit : la découverte de voisinage se fait par échange de messages HELLO sur une adresse de *multicast* 224.0.0.10 qui aboutit à la création d'une table de voisinage. La découverte de route se fait par envoi de chaque routeur à son voisin de la totalité des routes pour lesquelles il a une interface active et configurée dans EIGRP. Puis, les routeurs s'envoient des messages HELLO de façon périodique. Un routeur constate que son voisin n'est plus disponible après 3 intervalles de messages HELLO non reçus. Il est à noter que les mises à jour de routage sont envoyées uniquement en cas de modification de la topologie (mises à jour déclenchées), ce qui permet à EIGRP d'utiliser moins de bande passante.

Le choix des routes qui vont être stockées dans la table de routage est basé sur l'algorithme DUAL *(Diffusing Update Algorithm)* [27] qui s'appuie sur les notions suivantes :

- **Local Distance (LD) :** Il s'agit du coût de l'arête pour atteindre un nœud voisin.
- **Advertised Distance (AD) :** Associé à un voisin du nœud concerné, il s'agit du coût du plus court chemin de ce voisin à la destination. Cette information est transmise aux autres routeurs *via* un message EIGRP UPDATE.
- **Successor :** Il s'agit du prochain nœud sur le chemin le plus court pour atteindre une destination donnée.
- **Feasible Distance (FD) :** C'est la métrique associée au plus court chemin de ce nœud vers la destination. Ainsi FD = LD + AD associée à son *successor*.
- **Feasible Successor (FS) :** Il s'agit d'un voisin qui dispose d'un chemin de secours sans boucle vers la même destination que le *successor* et pour lequel l'*Advertised Distance* est inférieure à la *Feasible Distance* du *successor*. Le chemin de secours *via* ce voisin va permettre au protocole EIGRP de réagir rapidement en cas de panne sur le chemin principal.
- **Feasibility Condition :** Il s'agit d'un concept utilisé pour savoir si un chemin est valide ou non. Il stipule que si la métrique de l'un des voisins d'un nœud vers une destination est plus petite que la métrique de ce nœud vers cette même destination, alors ce voisin est le plus proche de cette destination, et doit être sur un chemin sans boucle vers la destination.

L'algorithme DUAL utilise trois tables : la table de voisinage qui contient pour chaque routeur les informations sur tous ses voisins directement connectés, la table de la topologie qui contient les coûts de toutes les routes vers chacune des destinations, et

la table de routage qui est associée à la meilleure route (plus petit coût) vers chaque destination, correspondant ainsi aux *successors* de la table des topologies.

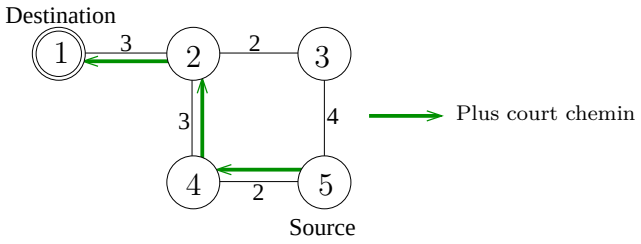

FIGURE 1.7 – Exemple d'un petit réseau de 5 nœuds pour EIGRP.

Considérons l'exemple de la figure 1.7 et supposons que 5 est le nœud source et 1 le nœud destination. DUAL calcule la route ainsi : tout d'abord, 5 demande à chacun de ses voisins son AD. 5 obtient $AD = 6$ *via* le nœud 4 et $AD = 5$ *via* le nœud 3 qui sont ses voisins. Ensuite, DUAL ajoute à chaque AD la valeur LD correspondante pour avoir sa FD, ce qui donne $FD = 6 + 2 = 8$ *via* 4 et $FD = 5 + 4 = 9$ *via* 3. DUAL marque ainsi 4 comme successor et 3 comme feasible successor de 5 pour la destination 1.

EIGRP utilise la *Feasibility Condition* pour s'assurer que seules les routes sans boucle sont sélectionnées car lorsqu'elle est vraie, aucune boucle ne peut se produire. Cependant, cette condition peut dans certains cas rejeter toutes les routes vers une destination, bien que certaines soient sans boucle. Dans ce cas, DUAL invoque un calcul de diffusion pour s'assurer que toutes les traces de l'itinéraire problématique sont éliminées du réseau. L'algorithme de Bellman-Ford est utilisé pour récupérer une nouvelle route.

Dans cette partie, nous avons présenté les protocoles à vecteur de distance. Il en ressort qu'avec ces protocoles, les routeurs n'ont pas une vision globale du réseau et construisent leur table de routage grâce à une diffusion des routes qui se fait de proche en proche et de façon périodique.

1.2.2 Protocoles à état de liens

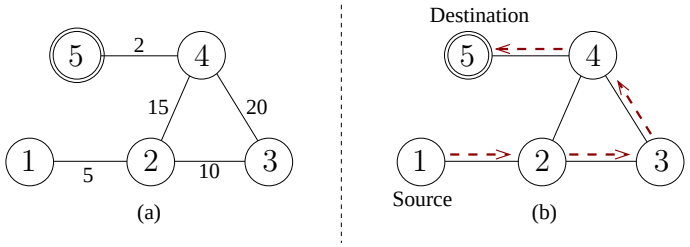

FIGURE 1.8 – Exemple de routage à état de liens : (a) un graphe de 5 nœuds, (b) le routage.

Les protocoles de routage à état de liens sont basés sur la connaissance complète de la topologie en chaque nœud. Chaque routeur découvre ses voisins à travers l'échange de messages HELLO, construit et transmet des paquets LSP *(Link State Packet)* qui contiennent des informations sur les voisins telles que l'identifiant du voisin, le type de liaison, et la bande passante. Les paquets LSP sont diffusés à tous les voisins qui stockent l'information et la transmettent jusqu'à ce que tous les routeurs aient la même information. Ainsi, les routeurs construisent tous une vision partagée de la topologie du réseau qui sera ensuite utilisée pour déterminer la meilleure route vers une destination.

La figure 1.8(a) montre un graphe avec 1 pour nœud source et 5 pour nœud destination. Un protocole à état de liens peut produire le chemin $1-2-3-4-5$ (figure 1.8(b)), comme celui permettant d'envoyer le maximum de paquets de 1 à 5, après calcul sur l'ensemble de la topologie.

Nous présentons dans la suite deux algorithmes phares de cette catégorie à savoir : OSPF et IS-IS.

OSPF : *Open Shortest Path First*

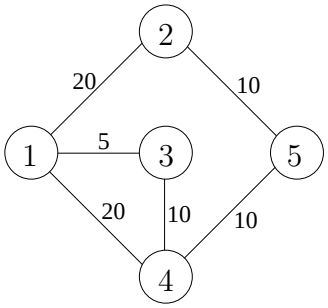

FIGURE 1.9 – Exemple d'un petit réseau de 5 nœuds pour OSPF.

OSPF [18] est défini par l'IETF [28]. C'est un protocole qui utilise l'algorithme de Dijkstra [29] pour calculer le plus court chemin d'un nœud à tous les autres. Le fonctionnement d'OSPF est le suivant :

- **La découverte des voisins :** chaque routeur découvre ses voisins immédiats par l'envoi de messages HELLO à intervalles réguliers. Une fois la relation d'adjacence avec ses voisins établie, le routeur communique la liste des réseaux auxquels il est directement rattaché à travers la propagation de messages LSA *(Link-State Advertisement)*. L'ensemble de ces LSA forme une base de données de l'état des liens appelée *Link-State Database* (LSDB).
- **Construction de la table de routage :** En se basant sur la LSBD et l'algorithme de Dijkstra, chaque routeur calcule un arbre des plus courts chemins (*Shortest Path Tree* ou SPT) vers les autres destinations. Cet arbre est ensuite utilisé pour construire la table de routage du routeur.
- **Le passage à l'échelle :** OSPF peut être utilisé sur de très petits réseaux comme sur de grands réseaux. Dans le cas de très grands réseaux, étant donné que chaque nœud a une vision globale du réseau, OSPF nécessiterait beaucoup de mémoire pour le stockage de la topologie du réseau par chaque nœud et plus de ressources de calcul pour l'application de l'algorithme de Dijkstra. Pour optimiser le passage à l'échelle, OSPF définit la notion de *zones* qui permettent une hiérarchisation des parties du réseau afin de diminuer la taille de la topologie à mémoriser sur chaque routeur. On distingue la zone 0 ou *backbone area* qui est la zone centrale connectée à toutes les autres. OSPF définit aussi les notions suivantes :
  - *Internal Router (IR)* qui représente un nœud interne à une unique zone et dont la fonction primordiale est de maintenir à jour sa base de données d'état des liens avec tous les réseaux de sa zone.
  - *Backbone Router (BR)* qui représente tout routeur ayant une interface dans la zone 0.
  - *Area Border Router (ABR)* qui connecte au moins deux zones différentes

dont la zone 0 et par conséquent, possède autant de LSBD que de zones couvertes.

— *Autonomous System Boundary Router (ASBR)* dont le rôle principal est l'échange d'informations entre un système autonome OSPF et d'autres systèmes autonomes.

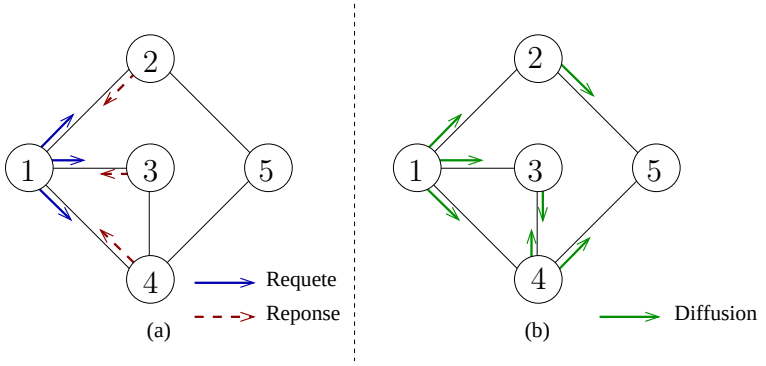

FIGURE 1.10 – (a) Échange de messages HELLO du nœud 1 à ses voisins, (b) Diffusion du message LSP provenant du nœud 1.

La figure 1.10(a) illustre un exemple d'un réseau de 5 nœuds. Le nœud 1 découvre ses voisins en diffusant un message HELLO représenté par des flèches pleines avec son identité et reçoit en retour un message HELLO représenté par des flèches en pointillés. Ensuite, tel que l'illustre la figure 1.10(b), une fois ses voisins identifiés, 1 construit un paquet LSP avec l'état de chacun des liens $\{(2, 20), (3, 5), (4, 20)\}$ qu'il diffuse à ses voisins. Ses voisins vont à leur tour le diffuser et le processus de partage du LSP de 1 s'arrête lorsque tous les nœuds l'ont reçu. Tous les autres nœuds font de même en construisant leur LSP et en les diffusant dans le réseau, ce qui permet à chacun de construire sa base de données LSDB. On obtient ainsi le tableau 1.7 pour 1 lui permettant ainsi d'avoir une vision globale de la topologie du réseau.

Noeud	**Voisin**	**Coût**
1	2	20
	3	5
	4	20
2	1	20
	5	10
3	1	5
	4	10
4	1	20
	3	10
	5	10
5	2	10
	4	10

TABLE 1.7 – LSBD du nœud 1 sur l'exemple de la figure 1.9.

A l'instar de 1, tous les nœuds vont utiliser cette base pour calculer la distance à chaque destination. Le tableau 1.8 montre la table de routage finale de 1.

Lorsqu'il y a un changement de topologie ou de métrique d'un lien, OSPF modifie les LSA associés et les diffuse dans le réseau. Puis, l'algorithme de Dijkstra est appliqué de

Source	Destination	Coût	Prochain saut
1	2	20	2
	3	5	3
	4	15	3
	5	25	3

TABLE 1.8 – Table de routage du nœud 1 sur l'exemple de la figure 1.9 après exécution d'OSPF.

nouveau pour avoir les nouvelles routes. Cependant, on peut voir apparaître des boucles transitoires dues au fait que les nœuds éloignés de la zone de modification continuent d'emprunter les mêmes routes qu'avant la réception de nouveaux LSA. Ce phénomène est amplifié par le fait que la génération de LSA est limitée dans le temps à 5 secondes. Ainsi, un réseau OSPF ne peut pas atteindre la convergence rapidement. Dans les nouvelles versions d'OSPF, la limitation d'annonce LSA a été réduite, permettant une convergence OSPF de l'ordre de la milliseconde, au détriment de la bande passante.

Un autre mécanisme de gestion des boucles semblable au processus de *Feasible Successors* de EIGRP, est implémenté depuis la version 2 d'OSPF. Il s'agit du mécanisme Loop-Free Alternate Fast Reroute (LFA-FRR) [30] décrit ci-après.

OSPF et LFA-FRR Loop-Free Alternate Fast Reroute (LFA-FRR) [30] a pour but de réduire la perte de paquets liée aux boucles transitoires de routage. Il implémente le calcul de prochains sauts fournissant une route de secours après une panne. Une fois que le réseau a convergé, c'est-à-dire que tous les nœuds ont reçu la notification du changement de topologie, OSPF exécute l'algorithme de Dijkstra. Si un voisin n'utilise pas le routeur actuel comme prochain saut pour une destination donnée, il peut alors être utilisé comme *feasible successor* ou nœud de secours. Les informations de *feasible successor* calculées par OSPF sont enregistrées dans la table de routage, où elles peuvent être utilisées immédiatement après détection d'une panne. LFA-FRR fonctionne de la manière suivante : soient src un nœud, v un de ses voisins, et $dest$ le nœud destination. v donne à src une alternative LFA vers $dest$ si et seulement si : $distance(v, dest) < distance(v, src) + distance(src, dest)$.

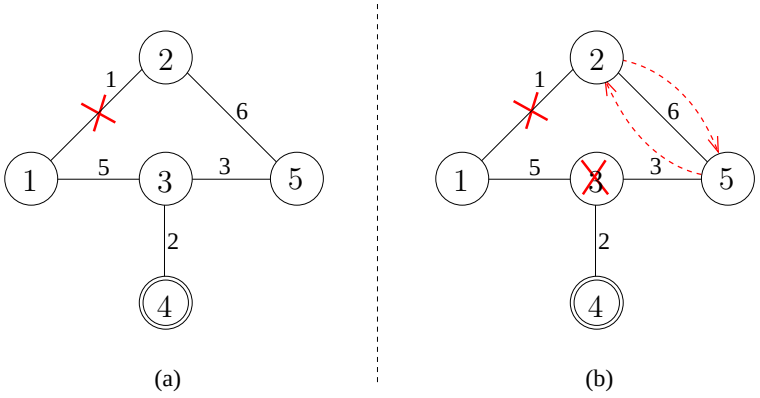

FIGURE 1.11 – Illustration de l'échec de LFA dans certains cas : (a) coupure du lien $(1, 2)$, LFA fonctionne correctement, (b) coupure du lien $(1, 2)$ et panne du nœud 3, LFA génère la boucle $(2, 5, 2)$.

Considérons la figure 1.9 et supposons que la source est le nœud 2 et la destination le nœud 3. Le prochain saut de 2 vers 3 est le nœud 1 après application de l'algorithme

Dijkstra. Le voisin 5 vérifie la condition $distance(5,3) < distance(5,2)+distance(2,3)$. 5 est par conséquent stocké comme le prochain saut de secours pour la destination 3 en cas de panne du lien $(1,2)$. Supposons à présent que le coût du lien $(5,4)$ vaut 30. Le nœud 1 est toujours prochain saut vers la destination 3. Le nœud 5 quant à lui ne respecte pas dans ce cas la condition LFA car $distance(5,3) \geq distance(5,2) + distance(2,3)$ et ne peut par conséquent être retenu. Par conséquent, l'existence d'un chemin alternatif en cas de panne dépend de la topologie du réseau et de la nature de la panne qui survient.

LFA-FRR ne marche que dans le cas de la panne d'un unique nœud, ou d'un unique lien. Considérons la figure 1.11. En cas de panne du lien $(1,2)$ (figure 1.11(a)), pour atteindre la destination 4, 2 envoie ses paquets au nœud 5 qui respecte la condition de LFA. Supposons que survienne ensuite la panne du nœud 3 (figure 1.11(b)). 2 et 5 après détection de la panne vont basculer à leur prochain saut de secours respectivement 5 et 2 et on a la génération de la boucle $(2,5,2)$ malgré l'utilisation des chemins alternatifs. Pour palier à ce problème, la solution proposée est la mise en place d'un chemin en aval *(downstream path)* vers la destination considérée. Un nœud v doit être le prochain saut sur le chemin en aval de *source* vers *destination*. Sur l'exemple de la figure 1.11, 5 est le prochain saut de 2 vers 4 sur le chemin en aval tandis que la réciproque est fausse. Ainsi en cas de pannes correlées comme celles du lien $(1,2)$ et du nœud 3, la boucle $(2,5,2)$ ne sera pas générée lors de l'utilisation de LFA-FRR.

Il est à noter que l'existence d'un nœud LFA ou d'un chemin en aval sans boucle dépend de la topologie du réseau. Les détails peuvent être trouvés dans [30].

IS-IS : *Intermediate System to Intermediate System*

Le protocole IS-IS *(Intermediate System to Intermediate System)* [31] est un protocole à état de liens dont le fonctionnement est similaire à celui de OSPF. En effet, les routeurs IS-IS maintiennent une vue topologique partagée. Afin de construire sa topologie, IS-IS utilise 3 types de messages : les messages HELLO permettant de construire les adjacences ; les messages LSP permettant d'échanger les informations sur l'état des liens et les messages SNP *(Sequence Number Packet)* permettant de confirmer la topologie. IS-IS découvre les voisins et forme des adjacences avec des messages HELLO transmis périodiquement.

IS-IS est aussi un protocole de routage hiérarchique permettant de définir plusieurs zones afin de réduire la taille des tables ainsi que le temps de convergence. On distingue ainsi le niveau 1 pour le routage interne à une zone avec pour caractéristiques la découverte du réseau et le routage, et le niveau 2 pour le routage extérieur à une zone avec pour caractéristiques l'agrégation des routes, l'apprentissage des adresses des autres routes, et la topologie pour le routage entre zones.

A l'instar de OSPF, IS-IS implémente le calcul de chemins de secours en cas de panne d'un lien ou d'un nœud grâce à *IS-IS Remote Loop-Free Alternate Fast Reroute (IS-IS LFA-FRR)*. La particularité avec IS-IS est que ce calcul se fait entre nœuds du même niveau entraînant ainsi une indisponibilité de chemins de secours pour la partie des nœuds non impliqués dans le processus.

IS-IS et *Sequence Number Packet* Le numéro de séquence des paquets permet à IS-IS d'assurer une synchronisation dans le partage des informations sur la topologie du réseau. On distingue deux types de paquets SNP :

— *Complete Sequence Number Packet* (CSNP) : Le but de ce type de paquets est

d'annoncer une liste complète des LSP contenus dans les LSDB propres aux expéditeurs. Ainsi, tout voisin qui reçoit ces paquets CSNP peut comparer sa LSDB existante au contenu du CSNP. Si des LSP ont des numéros de séquence plus élevés dans le CSNP, alors le voisin sait qu'il doit demander une mise à jour du LSP pour actualiser sa LSDB à la plus récente, ou recupérer un nouveau LSP.

— *Partial Sequence Number Packet* (PSNP) : Ils sont utilisés par les routeurs IS-IS pour demander des LSP mis à jour par des voisins et aussi des accusés de réception. Un seul PSNP peut demander plusieurs LSP.

1.3 Protocoles de routage pour les réseaux ad-hoc

Les réseaux sans fil ad-hoc sont des réseaux dans lesquels il n'y a pas d'infrastructure fixe. Dans ce contexte, le routage doit prendre en compte les contraintes de topologie dynamique, de taille du réseau, de bande passante limitée, de contraintes d'énergie des nœuds, de modestes capacités de calcul et de sauvegarde, etc. Les approches de routage pour ces réseaux sont plus complexes à mettre en œuvre comparées à celles utilisées dans les réseaux filaires classiques. Suivant la manière de créer et de maintenir les routes, une classification des protocoles de routage dans cette catégorie peut être faite en deux types : les proactifs et les réactifs.

1.3.1 Protocoles proactifs

Un protocole de routage est dit proactif si les routes vers toutes les destinations possibles sont créées et maintenues indépendamment des paquets de données qui circulent. Cette maintenance s'effectue même s'il n'y a pas de trafic circulant dans le réseau. L'avantage est la disponibilité immédiate des routes en cas de demande. Néanmoins, un changement de topologie qui conduit à un nombre élevé de modifications de chemins peut entraîner le gaspillage de la capacité du réseau. On distingue dans cette catégorie les protocoles DSDV et OLSR.

DSDV : *Dynamic Destination Sequenced Distance Vector*

DSDV [32] est un protocole proactif à vecteur de distance conçu spécialement pour les réseaux mobiles. Il est basé sur l'algorithme distribué de Bellman-Ford en ajoutant quelques améliorations. Chaque station mobile maintient une table de routage qui contient pour chaque destination connue une entrée contenant le prochain saut pour cette destination, le nombre de sauts, et un numéro de séquence pour garantir une convergence rapide. Chaque destination génère un numéro de séquence pair qu'elle envoie en même temps que sa table et qu'elle incrémente à chaque envoi d'un vecteur de données. Lorsqu'un voisin reçoit le vecteur de distances, il compare le numéro de séquence reçu avec celui de l'entrée correspondant à l'émetteur dans sa table de routage. L'entrée n'est mise à jour que si le numéro de séquence reçu est strictement supérieur à celui stocké. Si les deux numéros de séquence sont égaux, la route avec le coût minimum est choisie. Si un nœud détecte qu'un voisin devient inaccessible, il incrémente le numéro de séquence de l'entrée correspondante dans la table de routage pour rendre le numéro impair, signifiant que la route vers la destination est inaccessible. Le vecteur de distance est propagé et provoque des mises à jour par les voisins étant donné que le numéro de séquence a augmenté.

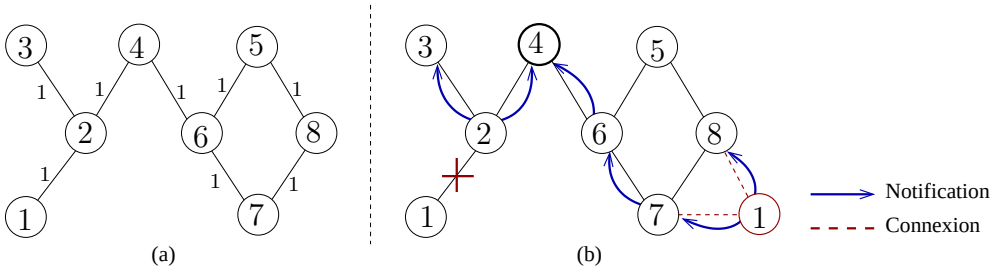

FIGURE 1.12 – Exemple d'un réseau de 8 nœuds pour DSDV : (a) graphe initial, (b) déplacement du nœud 1 et notification aux autres nœuds jusqu'au nœud 4 considéré.

Destination	Prochain saut	Nombre de sauts	numéro de sequence
1	2	2	$S052\_1$
2	2	1	$S256\_2$
3	2	2	$S748\_3$
4	4	0	$S602\_4$
5	6	2	$S378\_5$
6	6	1	$S554\_6$
7	6	2	$S136\_7$
8	6	3	$S494\_8$

TABLE 1.9 – Tableau de routage du nœud 4.

Considérons le graphe de la figure 1.12(a). Le tableau 1.9 représente la table de routage du nœud 4 avec une entrée pour chaque destination qui donne le prochain saut, le nombre de sauts nécessaires *(hopcount)* et le numéro de séquence de la destination. Supposons à présent que le nœud 1 décide de se déplacer et se connecte aux nœuds 7 et 8 comme l'illustre la figure 1.12(b). Le nœud 2 qui détecte la suppression du lien $(1, 2)$ met à jour sa table de routage en mettant à l'infini le nombre de sauts vers 1 et en incrémentant son numéro de séquence (qui devient $S053\_1$). Puis l'information est envoyée à ses voisins. Lorsque le nœud 1 se connecte aux nœuds 7 et 8, l'information est relayée au nœud 4 qui se retrouve avec deux routes vers la destination 1 : $(1, 2, \infty, S053\_1)$ et $(1, 6, 3, S162\_1)$. La route $(1, 6, 3, S162\_1)$ sera privilégiée car elle a un nombre de sauts plus petit que l'autre, et on obtient comme nouvelle table de routage du nœud 4, celle représentée sur la table 1.10.

Destination	Prochain saut	Nombre de sauts	numéro de séquence
1	6	3	$S162\_1$
2	2	1	$S366\_2$
3	2	2	$S858\_3$
4	4	0	$S712\_4$
5	6	2	$S488\_5$
6	6	1	$S664\_6$
7	6	2	$S246\_7$
8	6	3	$S604\_8$

TABLE 1.10 – Tableau de routage du nœud 4 après déplacement du nœud 1.

DSDV a l'avantage d'être simple, de réagir rapidement à la mobilité des nœuds et de limiter le problème de boucles grâce au numéro de séquence qui aide à faire la

distinction entre les anciennes et les nouvelles routes. Son inconvénient est le gaspillage des ressources dû à la maintenance des informations sur des routes.

OLSR : *Optimized Link State Routing Protocol*

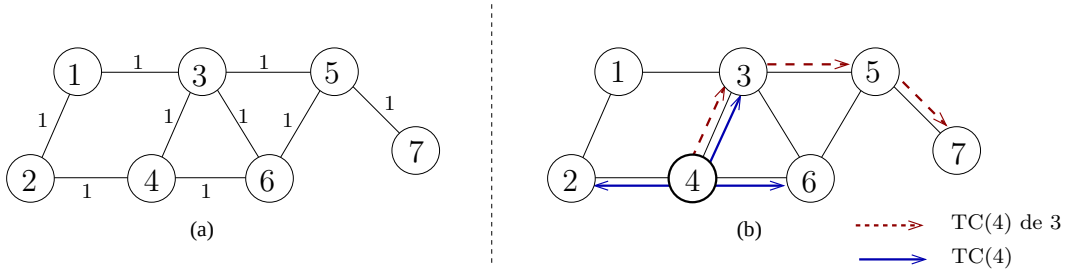

FIGURE 1.13 – Exemple d'un petit réseau de 7 nœuds pour OLSR : (a) graphe initial, (b) diffusion du message TC généré par le nœud 4.

OLSR [20] est un protocole proactif à état de liens qui vise à économiser les ressources lors des diffusions en réduisant le nombre de messages générés par l'inondation. En effet, contrairement à la diffusion faite par chaque nœud pour les protocoles de routage à état de liens, OLSR utilise le concept de relais multi-points (MPR pour *Multi Point Relay*). Un MPR est un sous-ensemble minimal des voisins d'un nœud choisi de façon à ce chaque voisin à deux sauts soit couvert par un voisin à un saut contenu dans ce sous-ensemble. Un nœud ne diffuse que les informations relatives aux liens avec les nœuds de son ensemble MPR, ce qui diminue la taille des messages de contrôle ; puis les messages ne sont transmis que de MPR en MPR, ce qui diminue également le nombre de messages de contrôle générés. Pour construire son ensemble MPR, un nœud envoie périodiquement des messages HELLO à ses voisins à un saut. A la réception des réponses, ce nœud est capable de déterminer ses voisins à deux sauts et ainsi, l'ensemble optimal MPR. Les nœuds n'appartenant pas au MPR d'un nœud donné traitent les paquets provenant de lui, mais ne les retransmettent pas.

A titre d'exemple, considérons le nœud 3 du graphe de la figure 1.13(a). Lorsque ses voisins 1, 4, 5, et 6 reçoivent un message du nœud 3, ils ne retransmettent le message qu'à leurs voisins autres que 3. A partir de la liste de voisins à un saut de chacun de ses voisins, 3 constitue son ensemble MPR en identifiant ceux qui ont pour voisins des nœuds n'apparaissant pas dans la liste de voisinage de 3. Dans ce cas, on a :

$$\begin{cases} \text{voisins}(1) = \{2\} \\ \text{voisins}(4) = \{2, 6\} \\ \text{voisins}(5) = \{6, 7\} \\ \text{voisins}(6) = \{4, 5\} \end{cases} \implies MPR(3) = \{4, 5\} \ .$$

A l'aide des nœuds 4 et 5, le nœud 3 peut atteindre les nœuds 2, 6 et 7 situés à deux sauts de lui.

OLSR distingue ensuite l'ensemble MS *(Multipoint Relay Selector Set)* d'un nœud qui contient tous les nœuds qui le choisissent comme MPR. Sur l'exemple, nous avons donc le nœud 3 qui appartient à la fois à MS(4) et MS(5). Chaque nœud du réseau envoie ensuite à chaque nœud contenu dans l'ensemble MPR, un message TC (*Topology Control*) contenant sa liste MS. Les informations qu'un nœud reçoit sont stockées dans sa table de topologie. Chaque entrée correspond à un lien entre un nœud et son MPR, et est conservée pendant une durée précise. Ensuite la table de routage est

construite et mise à jour lorsqu'un changement de topologie se produit par application de l'algorithme de Dijkstra qui calcule les plus courts chemins.

Reprenons l'exemple de la figure 1.13(a).

$$\begin{cases} \text{MPR}(1) = \{3\},\ \text{MPR}(2) = \{4\} \\ \text{MPR}(3) = \{4, 5\} \\ \text{MPR}(4) = \{3\},\ \text{MPR}(6) = \{3, 4, 5\} \\ \text{MPR}(5) = \{3\},\ \text{MPR}(7) = \{5\} \end{cases} \text{ et } \begin{cases} \text{MS}(1) = \emptyset,\ \text{MS}(2) = \emptyset \\ \text{MS}(3) = \{1, 4, 5, 6\} \\ \text{MS}(4) = \{2, 3, 6\},\ \text{MS}(6) = \emptyset \\ \text{MS}(5) = \{3, 6, 7\},\ \text{MS}(7) = \emptyset \end{cases}$$

Tous les nœuds ayant un ensemble MS non vide vont générer un message TC et l'envoyer à cet ensemble. Les MPR vont transmettre à nouveau ces messages jusqu'à ce que tout le réseau soit inondé. Prenons l'exemple du nœud 4 dont le MS est non vide. Le nœud 4 diffuse un message à son MS à savoir $\{2, 3, 6\}$. Le nœud 3, étant un MPR de 4, va à son tour retransmettre le message. Le nœud 5 fera de même en envoyant à 7 tel que l'illustre la figure 1.13(b). Lorsque les nœuds $3, 4$, et 5 ont généré les messages TC, chaque nœud du réseau a les informations sur l'état des liens pour pouvoir router vers toute destination. Pour le nœud 4, on obtient la table 1.11.

Destination	**Prochain saut**	**Nombre de sauts**
1	3	2
2	2	1
3	3	1
5	3 ou 6	2
6	6	1
7	3 ou 6	3

TABLE 1.11 – Table de routage du nœud 4 pour l'exemple de la figure 1.13.

OLSR a un fonctionnement en deux étapes :

— **Découverte des voisins :** Elle se fait par la découverte des voisins directs et à deux sauts, puis par le choix des MPR parmi les voisins directs. Pour ce faire, les nœuds s'échangent des messages HELLO.

— **Découverte de la topologie :** La topologie est découverte par la diffusion des informations sur le voisinage dans tout le réseau. Chaque MPR diffuse périodiquement un message TC *(Topology Control)* contenant la liste de ses voisins l'ayant choisi comme MPR. La construction de la table de la topologie se fait avec les informations contenues dans les messages TC et la construction de la table de routage se fait avec l'application de l'algorithme de recherche du plus court chemin dans un graphe.

OLSR offre des routes optimales en terme de nombre de sauts dans le réseau. Néanmoins, il souffre d'un problème de synchronisation des messages TC et des informations relatives aux routes contenues dans chacun des nœuds. Ces limites ont conduit à la conception d'une amélioration appelée BATMAN [33] que nous ne présenterons pas en détails.

L'approche d'OLSR pour annoncer seulement un sous-ensemble de liens augmente significativement la probabilité qu'un routeur n'ait pas de route et augmente la probabilité d'une boucle.

1.3.2 Protocoles réactifs

Les protocoles de routage réactifs créent et maintiennent les routes selon les demandes de trafic. Lorsque le réseau a besoin d'une route, une procédure de découverte de route est lancée. La méthode classique de recherche de route est basée sur le mécanisme d'apprentissage arrière *(backward learning)*. Cette méthode consiste à inonder le réseau avec une requête initiée par un nœud source. Lors de la réception de la requête, les nœuds intermédiaires stockent le chemin vers le nœud source. Une fois la station cible trouvée, elle répond par le chemin inverse ainsi établi.

L'inconvénient des protocoles réactifs par rapport aux protocoles proactifs est la réaction lente en cas de coupure d'une route en cours d'utilisation. Les mécanismes de maintenance sont lourds à gérer et créent des sursauts de trafic de contrôle à chaque tentative de recherche ou de réparation de route. Ces mécanismes entraînent un délai d'attente et une mise en file d'attente des paquets de données dus à l'absence temporaire de route. Ce type de routage est préconisé pour les réseaux de grande taille avec peu de destinations, car les nœuds n'ont pas à s'informer continuellement de la topologie.

Nous présentons dans la suite les protocoles AODV et DSR qui rentrent dans cette catégorie.

AODV : *Ad hoc On-Demand Distance Vector*

AODV [21] est un protocole réactif à vecteur de distance qui représente une amélioration du protocole DSDV. Son fonctionnement est le suivant : lorsqu'un nœud source veut établir une route vers un nœud destination pour lequel il ne possède pas encore de chemin, il diffuse un paquet RREQ (pour *Route REQuest*) à ses voisins. À la réception d'un paquet RREQ, un paquet RREP (pour *Route REPly*) est renvoyé à la source si le voisin connaît la destination. Dans le cas contraire, le voisin incrémente le nombre de sauts, et conserve les informations pour la construction du chemin inverse : l'identifiant du voisin qui a envoyé le paquet RREQ, l'adresse de la destination, l'adresse de la source, l'adresse de *broadcast*, le numéro de séquence du nœud source, et la durée de validité du chemin inverse. Puis, le voisin diffuse le paquet RREQ à son tour à toutes ses interfaces à l'exception de celle de réception du paquet RREQ. Un nœud recevant un paquet RREQ émet un paquet RREP s'il est la destination ou s'il possède une route vers la destination avec un numéro de séquence supérieur ou égal à celui du paquet RREQ, sinon il rediffuse le paquet RREQ. Un nœud intermédiaire avec AODV ne répond à la requête RREQ que si le numéro de séquence associé à la destination dont il dispose est supérieur ou égal à celui contenu dans la requête RREQ. Dans le cas, il diffuse le message à son tour. Les nœuds conservent chacun une trace des IP sources et des identifiants de diffusion des paquets RREQ. Dans le cas où ils reçoivent un paquet RREQ qu'ils ont déjà traité, ils le suppriment. Si le nœud source reçoit un paquet RREP, alors l'opération de découverte de route est terminée. Si ce n'est pas le cas, au bout d'un délai d'attente, le nœud source rediffuse le message RREQ et attend une certaine durée. En l'absence de réponse RREP, ce processus peut être répété jusqu'à RREQ_RETRIES fois. S'il n'y a toujours pas de réponse au bout des RREQ_ RETRIES + 1 tentatives, le processus de recherche de route est abandonné. Une nouvelle demande de route est initiée au bout d'un délai de 10 s.

Considérons la figure 1.14(a), supposons que le nœud 1 souhaite communiquer avec le nœud 7 et que la seule route actuellement connue vers la destination 7 est $5 - 7$. Comme l'illustrent les échanges de la figure 1.14(b), le nœud 1 envoie à ses voisins 2 et 3 un paquet RREQ qui contient son adresse, le numéro de séquence de 7, et le nombre

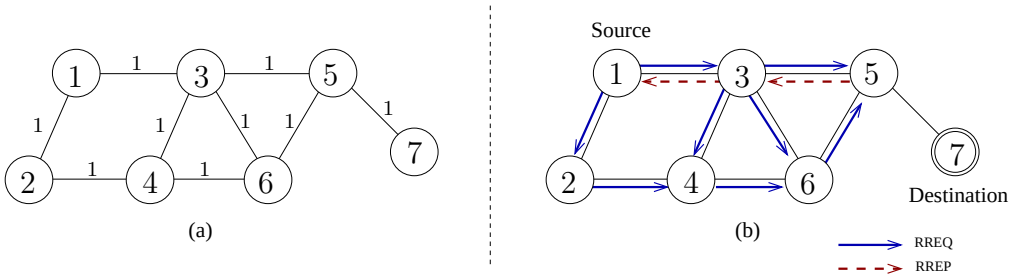

FIGURE 1.14 – Exemple d'établissement de route d'un nœud à un autre avec AODV.

de sauts *(hopcount)* qui vaut 0. À la réception du paquet, 3 et 2 créent un chemin inverse vers 1 : destination = 1, prochain saut = 1, *hopcount* = 1. N'ayant pas d'entrée pour 7, 3 diffuse à son tour le paquet RREQ à ses voisins 4, 5 et 6 (respectivement 2). Lorsque 5 reçoit le paquet RREQ de la part de 3, il crée à son tour un chemin inverse vers 1 : destination = 1, prochain saut = 3, *hopcount* = 2. Puis, étant donné qu'il dispose d'une route vers la destination 7, il vérifie les critères de numéro de séquence et la durée de validité puis crée un paquet RREP. Ce paquet contient : l'adresse et le numéro de séquence de 7, l'adresse et le numéro de séquence de 1, le *hopcount* qui vaut 1 et l'envoie à 3. 3 crée alors une entrée $(7, 5, 2)$ de chemin inverse pour la destination 7 et envoie le paquet RREP à 1. 1 reçoit le paquet RREP et ajoute l'entrée $(7, 3, 3)$.

En cas de coupure de liaison et de route, AODV utilise des messages d'erreur d'acheminement RERR (pour *Route ERRor*) pour notifier les nœuds concernés : si une coupure de liaison est détectée par un nœud, il invalide tous les itinéraires stockés dans sa table de routage ayant le lien brisé. Ensuite, il envoie un message RERR contenant le message que la destination est inaccessible, à tous les voisins directs utilisant cette route.

AODV a été considéré comme étant sans boucle grâce à l'utilisation des numéros de séquence. La preuve est la suivante [21]. Lorsqu'une boucle vers une destination d est générée, tous les nœuds x_i sur cette boucle ont une entrée de route pour la destination d avec le même numéro de séquence de destination. Les numéros de séquence de destination étant tous les mêmes, le prochain saut de chaque nœud x_i doit avoir été dérivé à partir du même RREP transmis par la destination d. Si l'on considère également la métrique m_i du nœud x_i pour atteindre la destination d et son prochain saut x_{i+1}, alors $m_i = m_{i+1} + 1$. Supposons l'existence d'une boucle $(x_1, x_2, ...x_n, x_1)$. On aurait ainsi $m_1 = m_n + (n - 1)$. Dans la boucle, x_1 étant prochain saut de x_n, on a $m_n = m_1 + 1$ et $n = 0$. Ce qui est contradictoire.

Cependant, Van Glabbeddk et al. [34] ont contesté cette affirmation en montrant qu'il existe des cas (rares dans la pratique), où le numéro de séquence ne garantit pas l'évitement des boucles. Ils réfutent l'idée selon laquelle les prochains sauts des nœuds ayant le même numéro de séquence de la destination doivent avoir été dérivés d'un même paquet RREP transmis par la destination d : certaines des informations peuvent provenir des messages RREQ, ou d'un message RREP transmis par un nœud intermédiaire qui a une route vers d. Pire encore, les nœuds sur la boucle peuvent avoir acquis leurs informations sur un itinéraire vers d à partir de différents messages RREP ou RREQ, qui ont tous le même numéro de séquence.

DSR : *Dynamic Source Routing*

DSR [35] est un protocole basé sur l'utilisation de la technique du routage calculé par la source. Dans cette technique, la source des paquets détermine la séquence complète

des nœuds à travers lesquels des paquets doivent être envoyés, ceci en stockant l'adresse de chaque routeur atteint lors de l'établissement de la route. La figure 1.15(a) illustre la construction de la route du nœud 1 vers la destination 8, où chaque nœud ajoute progressivement son adresse dans la liste. Puis, lorsque la destination 8 est atteinte, celle-ci répond par un message contenant la meilleure route. L'envoi de paquet se fait ainsi au premier nœud spécifié dans la route, qui est la source. Un nœud qui reçoit le paquet et qui est différent de la destination supprime son adresse de l'entête du paquet reçu, et transmet ce paquet au nœud suivant de la liste. Ce processus se répète jusqu'à ce que le paquet atteigne sa destination finale.

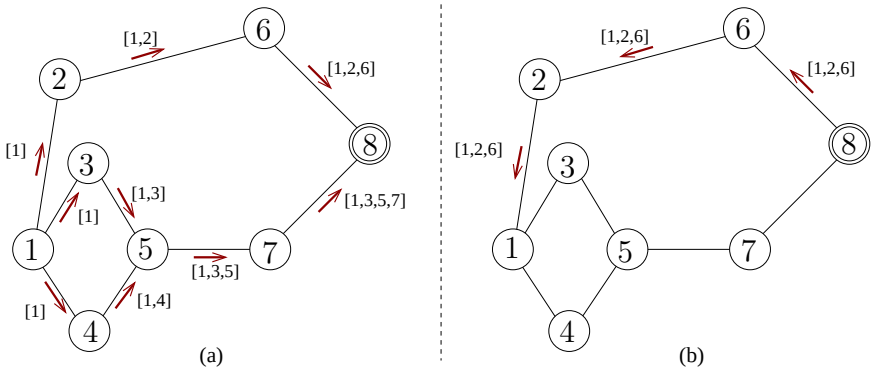

FIGURE 1.15 – Découverte de chemins avec DSR : (a) construction de l'enregistrement de routes, (b) retour de la route vers la destination 8.

Le protocole DSR n'intègre pas l'opération de découverte de routes à celle de la maintenance, comme le font les protocoles de routage conventionnels. DSR effectue sa maintenance comme suit : lorsqu'un nœud détecte un problème de transmission, un message d'erreur de route est envoyé à la source du paquet. Le message d'erreur contient l'adresse du nœud qui a détecté l'erreur et celle du nœud qui le suit dans le chemin. Lors de la réception de ce paquet d'erreur de route par l'hôte source, le nœud concerné par l'erreur est supprimé du chemin sauvegardé, et tous les chemins qui contiennent ce nœud sont tronqués à partir de ce point là. Par la suite, une nouvelle opération de découverte de routes vers la destination est initiée par la source.

Parmi les avantages du protocole DSR, le plus significatif est le fait que les nœuds de transit n'ont pas besoin de maintenir d'informations pour envoyer les paquets de données, puisque les paquets contiennent toutes les décisions de routage. En outre, dans DSR, il peut ne pas y avoir de boucle de routage, car le chemin de la source à la destination est indiquée explicitement par la source. En revanche, le chemin peut-être incorrect.

Conclusion

Il est important de pouvoir assurer la communication entre les nœuds d'un réseau. Ceci se fait par des protocoles de routage dont l'objectif est de calculer une route pour les paquets de données. Nous nous sommes attelés dans ce chapitre à faire un tour d'horizon de plusieurs protocoles proposés dans la littérature pour assurer le routage. Nous nous sommes concentrés sur les protocoles de routage intérieur et sur les protocoles de routage des réseaux sans fil ad-hoc.

Le bilan est le suivant.

— *Routage au sein des réseaux autonomes :* Les protocoles à vecteur de distance sont simples à mettre en œuvre mais ont généralement une convergence lente et donc une sensibilité accrue aux boucles de routage. Les protocoles à état de liens sont quant à eux caractérisés par un coût élevé en termes de capacité de calcul et de mémoire en cas de modification des topologies.
— *Routage au sein des réseaux mobiles ad hoc :* Nous avons noté dans ce cas le groupe des protocoles proactifs (DSDV et OLSR) qui calculent et maintiennent les routes de façon périodique, et les protocoles réactifs (DSR et AODV) qui ne le font qu'à la demande.
— *Gestion des boucles de routage :* Lorsque le problème de boucles de routage n'est pas géré par l'algorithme utilisé par le protocole pour le calcul des routes (à l'instar de l'algorithme DUAL pour EIGRP), le protocole implémente généralement d'autres mécanismes de gestion notamment la définition d'une valeur maximale pour éviter qu'un paquet ne circule indéfiniment dans le réseau, le *split horizon* qui ignore les messages de contrôle envoyés sur une interface différente de celle attendue, le *poison reverse* qui permet de rendre une route obsolète en définissant sa métrique au nombre maximal de sauts autorisé + 1, la mise en place de compteurs de retenue *holddown* qui définissent un délai d'attente avant une nouvelle validation d'informations sur un réseau tombé en panne, ou la mise en place de mises à jour déclenchées *holddown timer* qui rajoutent à la méthode *holddown* un compteur d'expiration à attendre.

Nous noterons qu'il existe d'autres classifications de protocoles dans les réseaux ad-hoc que nous n'avons pas présentées en détails, comme par exemple : les protocoles hybrides (ZRP [36], CBRP [37], etc) qui combinent les protocoles proactifs en local avec les protocoles réactifs en extérieur, les protocoles hiérarchiques (LEACH [38, 39], HSR [40], etc) qui sont basés sur une structure spécifique autour d'entités élues pour des rôles particuliers, ou encore les protocoles géographiques (GPSR [41], etc) qui utilisent les informations sur la position des nœuds mobiles.

Chapitre 2

Gestion des boucles transitoires

Sommaire

Introduction

Le protocole de routage peut être amené à changer pour s'adapter aux changements dans les applications comme par exemple : la détection d'un événement particulier (détection d'un feu de forêt, éruption d'un volcan actif) ou l'installation d'une nouvelle version de protocole apportant des améliorations sécuritaires ou de performances. Cependant, la migration d'un protocole de routage initial à un protocole de routage final dans un réseau est un problème complexe. En effet, si la migration n'est pas effectuée avec précaution, elle peut générer des boucles de routage qui peuvent dégrader considérablement les performances du réseau. Les conséquences sont *la consommation de la bande passante*, *l'augmentation des délais* due au fait que certains paquets sont routés dans des boucles de routage, et *la perte de paquets* car les paquets finissent par être détruits. Plusieurs heuristiques ont été proposées dans la littérature pour gérer ce problème de boucles de routage. Nous pouvons les classifier en deux groupes : les heuristiques basées sur le routage final et les heuristiques basées sur l'augmentation du poids des liens. Dans ce chapitre, nous décrivons en détails ces deux catégories.

2.1 Formalisation et notations

Les boucles de routage sont le thème principal des travaux de cette thèse. Dans cette section, nous présentons et définissons de manière formelle les éléments utilisés par les différentes solutions, à savoir : une boucle, une étape, une transition.

2.1.1 Boucle

Une boucle dans un réseau est un chemin (cf annexe A.5) de routage de deux routeurs ou plus ayant pour début et fin un même routeur, ce qui se traduit par une transmission de paquets qui n'atteint jamais la destination souhaitée.

Formellement, une boucle de routage dans un réseau représenté par un graphe $G = (V, E)$ (cf annexe A.2) est un cycle (cf annexe A.8) dans le graphe orienté $G' = (V, E')$ avec $E' = \{(u, v)\}$ où $u \in V$ et $v = \mathcal{R}(u)$.

Etant donné qu'une boucle correspond à un chemin ch de la forme $n_0 - n_1 - ... - n_l - n_0$ avec $n_i \in V$, $i \in [0, l]$, et $(n_i, n_{i+1}) \in E'$, nous noterons une boucle sous la forme $(n_0, n_1, ..., n_l, n_0)$.

2.1.2 Boucle transitoire

Une boucle transitoire est une boucle qui se produit lors de la phase de migration des nœuds d'un protocole de routage vers un autre protocole. Nous faisons généralement l'hypothèse que les deux protocoles de routage $\mathcal{R}_i$ et $\mathcal{R}_f$ sont valides. Ainsi, la boucle transitoire inclut nécessairement les deux protocoles de routage c'est-à-dire que l'état du réseau est sous la forme $G = (V, E')$ avec $E' = \{(u, v)\} \in E$ tel que :

— $v = \mathcal{R}_i(u)$ si u n'a pas migré et route encore suivant $\mathcal{R}_i$,
— $v = \mathcal{R}_f(u)$ si u a migré et route à présent suivant $\mathcal{R}_f$.

2.1.3 Etape

Une étape est un ensemble de nœuds $S = \{n_1, n_2, ..., n_k\}$ qui effectuent la migration d'un protocole de routage initial $\mathcal{R}_i$ vers un protocole de routage final $\mathcal{R}_f$ sans dépendance entre eux. Formellement, $\exists\ t$ tel que :

$$\forall n \in S, \begin{cases} \mathcal{R}(n, t-1) = \mathcal{R}_i(n), \\ \quad \mathcal{R}(n, t) = \mathcal{R}_i(n) \text{ ou } \mathcal{R}(n, t) = \mathcal{R}_f(n), \\ \mathcal{R}(n, t+1) = \mathcal{R}_f(n), \end{cases}$$

Pour chaque destination d, nous noterons l'étape : $S = \{n_1, n_2, ..., n_k\}_d$ pour signifier que les nœuds de S migrent ensemble vers la destination d.

Nous noterons l'étape : $S = \{n_1, n_2, ..., n_k\}_{d_1,d_2,...,d_m}$ avec $m \leq n$ pour signifier que les nœuds migrent pour les destinations $d_1, d_2, ..., d_m$.

Si certains nœuds de S migrent ensemble pour une partie des destinations, tandis que d'autres migrent ensemble vers une autre partie des destinations, nous utiliserons dans ce cas l'opérateur $\cup$ *(union)*. Si par exemple le nœud n_1 migre vers la destination d_m, les nœuds $n_2, .., n_{k-2}$ migrent vers les destinations d_1 et d_2, et les nœuds n_{k-1} et n_k migrent vers la destination d_{m-1}, alors nous noterons l'étape comme suit : $S = \{n_1\}_{d_m} \cup \{n_2, ..., n_{k-2}\}_{d_1,d_2} \cup \{n_{k-1}, n_k\}_{d_{m-1}}$.

La taille d'une étape correspond au nombre de nœuds distincts de cette étape, indépendamment du nombre de destinations.

2.1.4 Etape valide

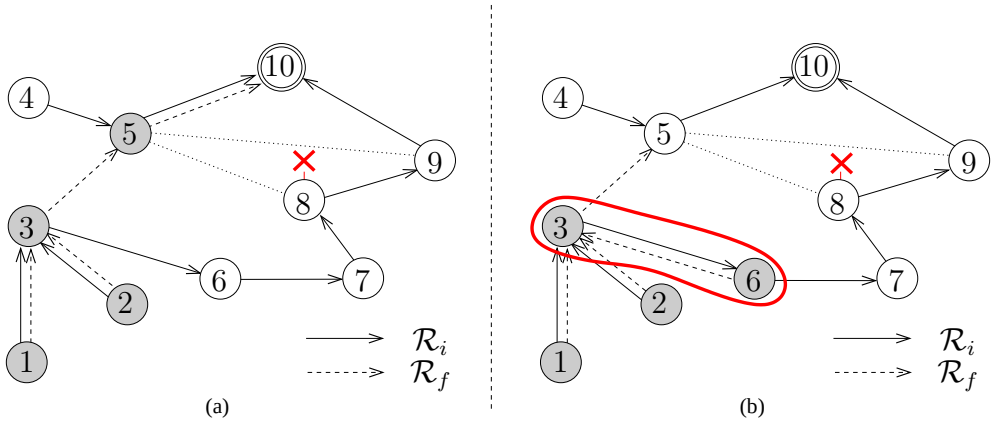

FIGURE 2.1 – Exemples d'étapes valide et non valide lors de la survenue de la panne du nœud 8 dans le réseau de la figure 1.3 : (a) $S = \{1, 2, 3, 5\}_{10}$ est valide, (b) $S = \{1, 2, 3, 6\}_{10}$ est non valide.

Une étape valide est une étape qui ne génère pas de boucles de routage quel que soit l'ordre réel de migration des nœuds au sein de l'étape.

Soient $G = (V, E)$ un réseau de n nœuds, et $S = \{n_1, n_2, ..., n_k\}$ une étape. Supposons que l'étape S se situe à un instant t. Tous les nœuds ayant migré avant l'instant t routent déjà suivant $\mathcal{R}_f$. Soit S' l'ensemble de ces nœuds. Supposons également $G' = (V, E')$ le graphe représentant l'état du réseau, n un nœud du réseau, et d la destination. E' est construit de la manière suivante :

— Si $n \in S'$ alors, $(n, \mathcal{R}_f^d(n)) \in E'$.
— Si $n \in V \setminus S'$ alors, $(n, \mathcal{R}_i^d(n)) \in E'$.

— Si $n \in S$ alors, $(n, \mathcal{R}_f^d(n)) \in E'$.
S est valide ssi il n'existe pas de cycle dans G'.

La figure 2.1 nous montre deux exemples d'étapes possibles quand le nœud 8 tombe en panne : le premier exemple, $S = \{1, 2, 3, 5\}_{10}$, est une étape valide car, comme le montre la figure 2.1(a), aucune boucle de routage ne peut être générée durant cette phase transitoire. Dans le deuxième exemple, si la première étape est $S = \{1, 2, 3, 6\}_{10}$, alors les nœuds 3 et 6 peuvent créer une boucle de routage si le nœud 3 migre avant le nœud 6 durant l'application de l'étape (figure 2.1(b)). Cette boucle n'apparaît pas dans tous les cas, et est amenée à disparaître quand tous les nœuds auront migré.

2.1.5 Transition

Une transition est un ordre de migration des nœuds. Elle se traduit par une séquence d'étapes $Tr = (S_1, S_2, ..., S_k)$ avec $1 \leq k \leq n$, telle que chaque nœud du réseau appartient à une étape par destination : $\forall n \in V$, $\forall d \in D$, $\exists j \in [1, k]$ tel que $n \in S_j$ pour d.

La taille d'une transition est donnée par son nombre d'étapes, k.

Considérons l'exemple de la figure 1.3. Un exemple de transition est $Tr = (\{10\}_{10}, \{5\}_{10}, \{9\}_{10}, \{4\}_{10}, \{8\}_{10}, \{7\}_{10}, \{6\}_{10}, \{3\}_{10}, \{2\}_{10}, \{1\}_{10})$ qui compte 10 étapes, chacune étant représentée par un nœud. Un autre exemple de transition est $Tr = (\{1, 2, 3, 5, 10\}_{10}, \{4, 6, 9\}_{10}, \{7, 8\}_{10})$ qui compte 3 étapes.

2.1.6 Transition valide

Une transition est dite valide si et seulement chacune de ses étapes est valide. Formellement, soient $G = (V, E)$ un réseau de n nœuds et $Tr = (S_1, S_2, ..., S_k)$ avec $1 \leq k \leq n$ la transition. Tr est valide ssi $\forall j \in [1, k], S_j$ est valide.

Considérons l'exemple du réseau de la figure 1.3 et supposons qu'il s'agit du retrait planifié du nœud 8 du réseau. La transition $Tr = (\{1, 2, 3, 5, 10\}_{10}, \{4, 6, 9\}_{10}, \{7, 8\}_{10})$ est un exemple de transition valide car chacune des trois étapes est valide. En effet, comme le montre la figure 2.2, l'application des trois étapes dans l'ordre ne génère aucune boucle de routage. La figure 2.2(a) montre le routage initial avant le début de la migration des nœuds. Puis, vient l'application de la première étape avec les nœuds $\{1, 2, 3, 5, 10\}$. Durant la phase migratoire, les nœuds peuvent router suivant l'un des deux protocoles $\mathcal{R}_i$ et $\mathcal{R}_f$. C'est pourquoi chacun des nœuds de l'étape indiqué en gris est représenté avec ses deux prochains sauts à la figure 2.2(b). A la fin de l'étape, les nœuds $\{1, 2, 3, 5, 10\}$ routent tous suivant $\mathcal{R}_f$ tandis que le reste du réseau continue à router suivant $\mathcal{R}_i$ comme le montre la figure 2.2(c). Puis, vient la deuxième étape avec les nœuds $\{4, 6, 9\}$. Comme pour la première étape, ils sont également représentés avec chacun de leurs prochains sauts durant la phase migratoire à la figure 2.2(d). Puis la fin de l'étape est représentée par la figure 2.2(e). Le processus est enfin repris pour la troisième et dernière étape formée des nœuds $\{7, 8\}$ dont la phase migratoire est représentée par la figure 2.2(f). La fin de cette étape marque la fin de la migration des nœuds car désormais tous les nœuds routent suivant $\mathcal{R}_f$ comme le montre la figure 2.2(g). Le nœud 8 peut à présent être retiré du réseau.

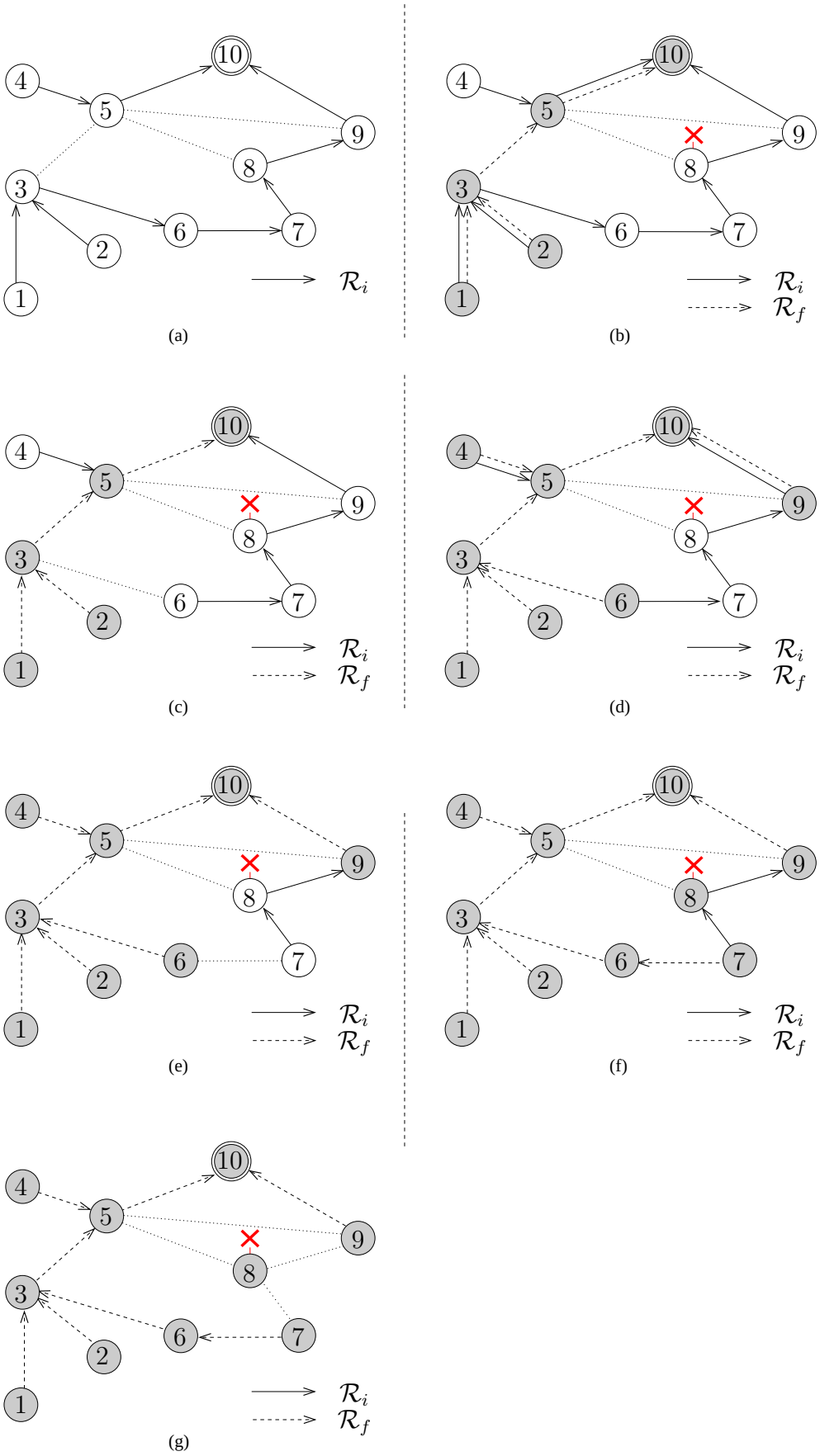

FIGURE 2.2 – Exemple d'une transition valide $Tr = (\{1,2,3,5,10\}_{10}, \{4,6,9\}_{10}, \{7,8\}_{10})$ pour le retrait planifié du nœud 8 dans le réseau de la figure 1.3 : (a) routage initial, (b) phase migratoire de l'étape 1, (c) fin de l'étape 1, (d) phase migratoire de l'étape 2, (e) fin de l'étape 2, (f) phase migratoire de l'étape 3, (g) fin de l'étape 3 et de la transition.

Un autre exemple de transition valide est $Tr = (\{10\}_{10}, \{5\}_{10}, \{9\}_{10}, \{4\}_{10}, \{8\}_{10}, \{7\}_{10}, \{6\}_{10}, \{3\}_{10}, \{2\}_{10}, \{1\}_{10})$.

Cependant, la transition $Tr = (\{10\}_{10}, \{5\}_{10}, \{9\}_{10}, \{4\}_{10}, \{8\}_{10}, \{7\}_{10}, \{3\}_{10}, \{6\}_{10}, \{2\}_{10}, \{1\}_{10})$ également de 10 étapes, est une transition non valide car l'étape 7 génère la boucle $(3, 6, 3)$.

2.2 Heuristiques basées sur le routage final

Dans cette partie, nous nous intéressons aux heuristiques qui basent leurs solutions sur l'ordre de migration dicté par l'ordre de succession des nœuds sur le routage final.

2.2.1 RTH : *Routing Tree Heuristic*

RTH [42] scinde les destinations en deux groupes : les destinations non problématiques pour lesquelles les migrations peuvent se faire ensemble sans causer des boucles de routage, et les destinations problématiques, qui doivent être gérées individuellement. Pour chaque destination, RTH identifie un ensemble de contraintes sur l'ordre de migration. Puis, RTH calcule un ordre de migration des nœuds qui satisfait l'ensemble de ces contraintes. Chaque contrainte par destination stipule qu'un nœud ne migre vers son prochain saut sur le routage final $\mathcal{R}_f$ que lorsque tous ses successeurs sur le chemin vers la destination ont déjà migré.

Pour chaque destination d, un ensemble S_d des nœuds n'apparaissant dans aucune boucle est calculé. S_d contient les nœuds n tels que $\mathcal{R}_i(n) = \mathcal{R}_f(n)$ et la destination d. Tous les nœuds dans S_d sont autorisés à migrer dès que possible. Tous les autres nœuds ayant leurs prochains sauts différents suivant $\mathcal{R}_i$ et $\mathcal{R}_f$ sont ajoutés dans un ensemble $\overline{V}_d$. Un ensemble C_d de contraintes est construit de telle sorte que pour chaque élément de P, qui est l'ensemble des chemins d'un nœud source (sans arc entrant) vers la destination d sur $\mathcal{R}_f$, une contrainte est générée pour la dernière paire de nœuds (u, v) avec $u \in \overline{V}_d$ et $u \notin S_d$. Ensuite un graphe orienté acyclique G_{C_d} est construit avec tous les nœuds et arcs de C_d, et un tri topologique est calculé sur G_{C_d}. Le tri topologique définit pour un graphe acyclique un ordre de visite des sommets tel qu'un sommet est toujours visité avant ses successeurs. En fonction de ce tri topologique, les nœuds effectuent la transition un par un.

Nous considérons dans la suite deux cas : le cas d'un réseau avec une seule destination et le cas d'un réseau multi-destinations.

La figure 2.3(a) représente un réseau de 10 nœuds avec un graphe pondéré (cf annexe A.4) par $\mathcal{R}_i$ et la figure 2.3(b) représente le même réseau mais avec la pondération de $\mathcal{R}_f$.

Exemple pour un réseau mono-destination

Considérons tout d'abord que la destination est le nœud 10. La figure 2.4(a) montre le routage initial $\mathcal{R}_i^{10}$, et la figure 2.4(b) montre le routage final $\mathcal{R}_f^{10}$.

— *Construction de l'ensemble* S_{10} : 10 étant la destination, ce nœud est automatiquement ajouté à S_{10}. Ensuite, le nœud 7 est ajouté à S_{10} car c'est le seul à avoir ses deux prochains sauts selon $\mathcal{R}_i$ et $\mathcal{R}_f$ identiques. Les nœuds de $S_{10} = \{7, 10\}$ peuvent immédiatement migrer.

— *Construction de l'ensemble* $\overline{V}_{10}$: $\overline{V}_{10} = V \setminus S_{10} = \{1, 2, 3, 4, 5, 6, 8, 9\}$.

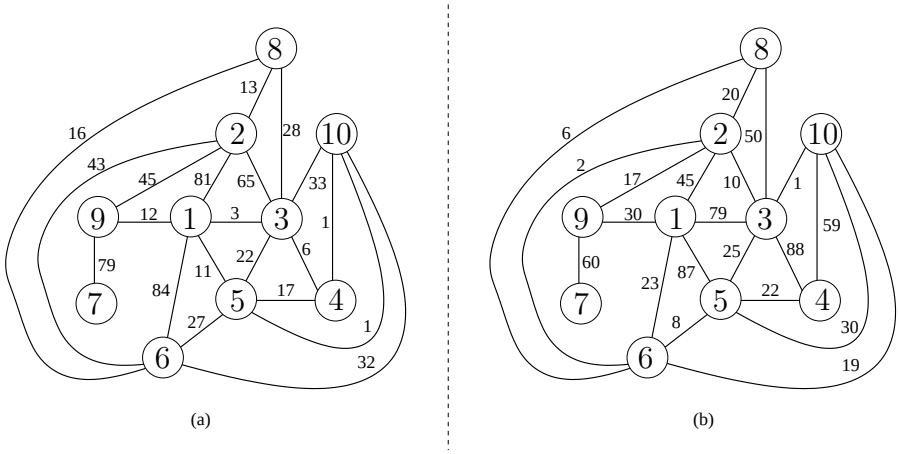

FIGURE 2.3 – Deux pondérations d'un même graphe : (a) la pondération correspondant à $\mathcal{R}_i$, (b) la pondération correspondant à $\mathcal{R}_f$.

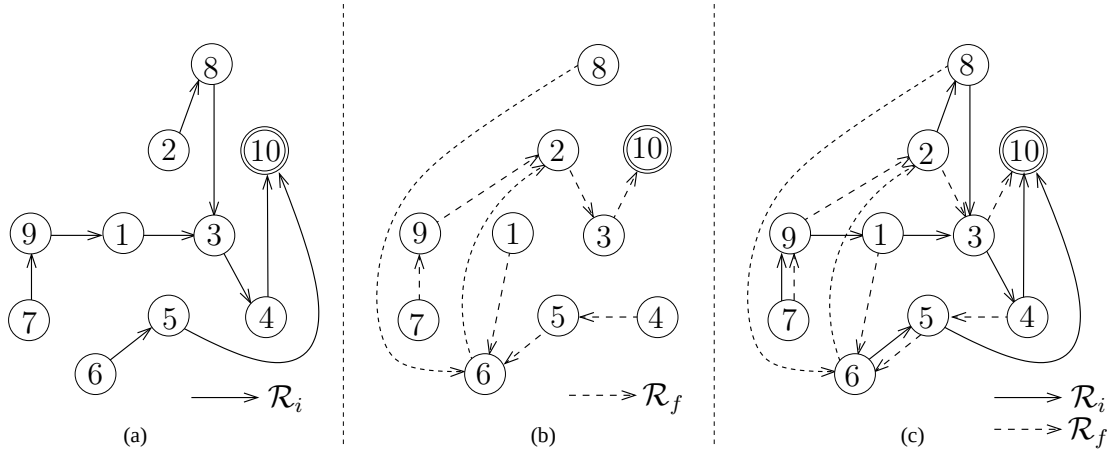

FIGURE 2.4 – (a) Le routage initial $\mathcal{R}_i$ vers la destination 10, (b) le routage final $\mathcal{R}_f$ vers la destination 10, (c) l'union des deux routages $\mathcal{R}_i$ et $\mathcal{R}_f$ vers la destination 10 durant la phase transitoire du changement de protocole qui génère les boucles $(2, 8, 6, 2)$, $(2, 3, 4, 5, 6, 2)$, $(2, 8, 3, 4, 5, 6, 2$, et $(5, 6, 5)$.

— *Construction de P l'ensemble des chemins des feuilles jusqu'à la destination* : tous les nœuds qui sont les feuilles de $\mathcal{R}_f$ sont $\{1, 4, 7, 8\}$.

Ainsi $P = \begin{cases} ch(1,10) = 1-6-2-3-10 \\ ch(4,10) = 4-5-6-2-3-10 \\ ch(7,10) = 7-9-2-3-10 \\ ch(8,10) = 8-6-2-3-10 \end{cases}$

— *Construction du graphe de contraintes* $G_{C_{10}}$: Une contrainte (u, v) stipule qu'un nœud v ne peut migrer avant son prochain saut u sur $\mathcal{R}_f$ avec $v \notin S_d$, et d la destination. RTH calcule donc l'ensemble C_{10} en construisant toutes les contraintes sur chaque chemin. Ainsi $ch(1, 10)$ donne les contraintes : $\{(6, 1), (2, 6), (3, 2)\}$. Par application du même principe sur chacun des 3 chemins restant, on obtient $C_{10} = \{(6, 1), (2, 6), (3, 2), (5, 4), (6, 5), (2, 9), (6, 8)\}$. Le graphe $G_{C_{10}} = (V, C_{10})$ est représenté sur la figure 2.5.

— *Calcul de la transition finale* : un tri topologique sur $G_{C_{10}}$ donne $(3, 2, 9, 6, 8, 5, 4, 1)$. L'ordre de migration de RTH pour la destination 10 est donc $Tr = (\{10\}, \{7\}, \{3\}, \{2\}, \{9\}, \{6\}, \{8\}, \{5\}, \{4\}, \{1\})$.

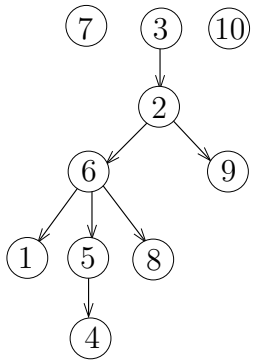

FIGURE 2.5 – Graphe des contraintes générées par RTH pour la destination 10.

Exemple pour un réseau multi-destinations

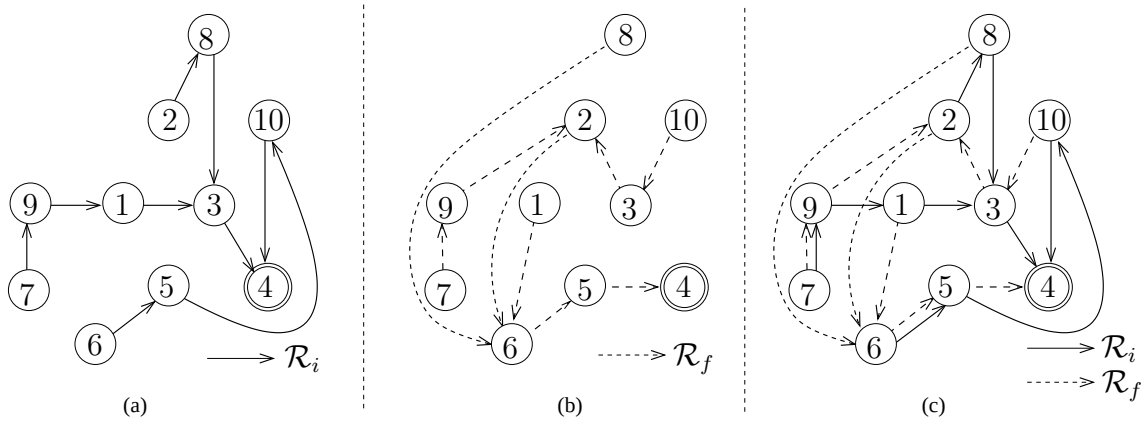

FIGURE 2.6 – (a) Le routage initial vers la destination 4, (b) le routage final vers la destination 4, (c) l'union des deux routages vers la destination 4 durant la phase transitoire du changement de protocole qui génère la boucle : $(2, 8, 3, 2)$.

Considérons à présent que les destinations sont les nœuds 4 (figure 2.6), 6 (figure 2.7) et 10 (figure 2.4). En appliquant le même procédé que précédemment pour chacune des destinations, on obtient le résultat suivant.

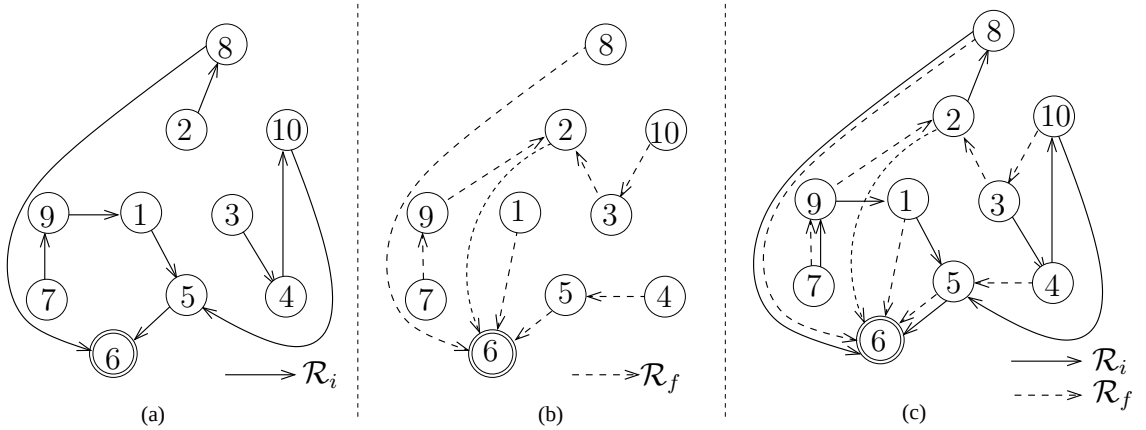

FIGURE 2.7 – (a) Le routage initial vers la destination 6, (b) le routage final vers la destination 6, (c) l'union des deux routages vers la destination 6 durant la phase transitoire du changement de protocole qui génère la boucle : $\{3, 4, 10\}$.

— Destination 4 : $Tr = (\{7\}_4, \{4\}_4, \{6\}_4, \{5\}_4, \{8\}_4, \{2\}_4, \{9\}_4, \{3\}_4, \{10\}_4, \{1\}_4)$.
— Destination 6 : $Tr = (\{7\}_6, \{4\}_6, \{8\}_6, \{6\}_6, \{5\}_6, \{2\}_6, \{9\}_6, \{3\}_6, \{10\}_6, \{1\}_6)$.
— Destination 10 : $Tr = (\{10\}_{10}, \{7\}_{10}, \{3\}_{10}, \{2\}_{10}, \{9\}_{10}, \{6\}_{10}, \{8\}_{10}, \{5\}_{10}, \{4\}_{10}, \{1\}_{10})$.

Donc, la transition finale est : $Tr = (\{7\}_4, \{4\}_4, \{6\}_4, \{5\}_4, \{8\}_4, \{2\}_4, \{9\}_4, \{3\}_4, \{10\}_4, \{1\}_4, \{7\}_6, \{4\}_6, \{8\}_6, \{6\}_6, \{5\}_6, \{2\}_6, \{9\}_6, \{3\}_6, \{10\}_6, \{1\}_6, \{10\}_{10}, \{7\}_{10}, \{3\}_{10}, \{2\}_{10}, \{9\}_{10}, \{6\}_{10}, \{8\}_{10}, \{5\}_{10}, \{4\}_{10}, \{1\}_{10})$.

Limites

Les principaux inconvénients de RTH sont les suivants. Premièrement, les auteurs n'ont pas spécifié l'algorithme d'identification des destinations problématiques dû au fait que leurs expérimentations n'en présentent qu'un faible taux [13]. Deuxièmement, la migration est faite nœud par nœud pour chaque destination non problématique, ce qui conduit à des transitions globalement longues.

Complexité

En termes de temps, RTH a une complexité de l'ordre de $\mathcal{O}(n^3) + \mathcal{O}(n + m)$ avec $m \leq 2n$ le nombre d'arcs. En effet, pour chaque destination, RTH construit le graphe des contraintes en $\mathcal{O}(n^2 + n)$. Puis, le tri topologique est appliqué et peut se faire en $\mathcal{O}(n + m)$.

En termes d'étapes, pour un réseau de n nœuds avec $d \in [1; n]$ destinations. S'il y a $t \in [1; d - 1]$ destinations problématiques, RTH produira une transition de $(t + 1).n$ étapes. De plus, lorsque n croît, t devient proche de d [13]. RTH a ainsi une complexité en étapes de l'ordre de $\mathcal{O}(n^2)$.

2.2.2 RTH-p : *Routing Tree Heuristic with parallel changes*

RTH-p [13] propose deux améliorations de RTH.

La première amélioration est un algorithme glouton pour identifier un grand nombre de destinations non problématiques. Le processus d'identification (algorithme 1) est le suivant. Initialement, l'ensemble des destinations problématiques T est initialisé à $\emptyset$.

Algorithme 1 : Détection des destinations problématiques.

Données : D un ensemble des destinations, V un ensemble de nœuds
Résultat : T un ensemble de destinations problématiques

```
1  T = ∅, E = ∅
2  G_C = (V, E)
3  pour d ∈ D faire
4  |   G_{C_d} = le graphe des contraintes pour la destination d
5  |   G_C ← G_C ∪ G_{C_d}
6  |   si G_C contient des boucles alors
7  |   |   ajouter d à T
8  |   |   retirer G_{C_d} de G_C
9  |   fin
10 fin
```

Dans la boucle principale qui considère chaque destination d individuellement, le graphe de contraintes G_{C_d} associé à la destination d est calculé. Puis les contraintes extraites C_d sont ajoutées (par union des graphes G_C et G_{C_d}) à l'ensemble des contraintes précédemment calculées pour les destinations non problématiques. Si le nouveau graphe G_C ne contient pas de boucles, alors d est considérée comme destination non problématique. Cependant, si le nouveau graphe de contraintes G_C contient au moins une boucle, la destination d est ajoutée à la liste des destinations problématiques T, et les contraintes générées à partir de celle-ci sont supprimées de C_d. Dans ce cas, les modifications apportées à G_C concernant d sont annulées. Ce processus est répété jusqu'à ce que toutes les destinations aient été traitées.

Lorsque toutes les destinations ont été traitées, le calcul de l'ordre de migration est d'abord fait pour les destinations non problématiques en utilisant G_C, puis il est fait pour les destinations problématiques en les considérant individuellement et en rajoutant chacun de leurs ordres de basculement.

La seconde amélioration permet des basculements parallèles dans RTH, afin d'obtenir une séquence d'étapes plutôt qu'une séquence de nœuds. Cette modification peut être effectuée efficacement à partir du graphe de contraintes G_C calculé pour toutes les destinations non problématiques (ainsi que pour chaque destination problématique individuellement). Cette amélioration consiste à changer la dernière étape de RTH, en remplaçant le tri topologique de G_C par l'algorithme 2. Dans cet algorithme, un nouvel ensemble U est ajouté à la séquence S d'étapes. À chaque itération, tous les nœuds sans contrainte, représentés par les nœuds sans arcs entrant dans G_C, sont ajoutés au nouvel ensemble U et tous les arcs sortant de ces nœuds sont retirés de G_C. Étant donné que G_C n'a pas de boucle (ce qui est inhérent à la construction de toutes les destinations non problématiques, et au fait que les destinations problématiques sont considérées indépendamment), le processus se termine avec tous les nœuds dans S.

Exemple pour un réseau mono-destination

Considérons l'exemple utilisé pour RTH avec pour destination le nœud 10 (figure 2.4). La figure 2.5 illustre le graphe de contraintes calculées. La migration se fait de la racine aux feuilles avec pour règle que les nœuds situés à une même profondeur peuvent être migrés dans une même étape sans causer de boucles de routage.

Algorithme 2 : Changements parallèles dans RTH.

Données : G_C un graphe de contraintes
Résultat : S une séquence d'étapes

$S = \emptyset$
tant que *certains nœuds ne sont pas dans* T **faire**
  $U = \emptyset$
  pour *tout nœud* $n \in G_C$ **faire**
    si $n \notin U$ *et* $n \notin S$ **alors**
      si n *n'a pas d'arcs entrant dans* G_C **alors**
        $U \leftarrow U \cup \{n\}$
      fin
    fin
  fin
  pour *tout nœud* $n \in U$ **faire**
    retirer de G_C tous les arcs sortant de n
  fin
  ajouter l'ensemble U à la séquence S
fin

Voici les séquences d'étapes après application de l'algorithme de calcul des étapes :

$$\begin{cases} S_1 = \{3, 7, 10\} \\ S_2 = \{2\} \\ S_3 = \{6, 9\} \\ S_4 = \{1, 5, 8\} \\ S_5 = \{4\} \end{cases}$$

RTH-p produit ainsi la transition $Tr = (\{3, 7, 10\}_{10}, \{2\}_{10}, \{6, 9\}_{10}, \{1, 5, 8\}_{10}, \{4\}_{10})$ qui a 5 étapes au lieu de 10 comparé à RTH. Comme tous les nœuds d'une même étape peuvent migrer indépendamment sans ordre, RTH-p est plus rapide que RTH.

Exemple pour un réseau multi-destinations

Pour le cas multi-destinations, nous considérons les destinations $\{4, 6, 10\}$ avec pour figures respectives, la figure 2.6, la figure 2.7, et enfin la figure 2.4. Nous rappelons que pour chaque destination d, RTH calcule S_d, l'ensemble des nœuds initialisé à d et complété par tous ceux qui ont leur prochain saut sur $\mathcal{R}_i$ et leur prochain saut sur $\mathcal{R}_f$ déjà inclus dans S_d ; $\overline{V}_d$ l'ensemble des nœuds dont les deux prochains sauts respectivement suivant $\mathcal{R}_i$ et suivant $\mathcal{R}_f$ sont différents ; C_d l'ensemble des contraintes de basculement des nœuds pour cette destination.

Pour notre cas exemple, nous avons :

$$\begin{cases} S_4 = \{4\} \\ \overline{V}_4 = \{1, 2, 3, 5, 8, 9, 10\} \\ C_4 = \{(2, 3), (2, 9), (3, 10), \\ (5, 1), (5, 2), (5, 8)\} \\ Tr = (\{7\}, \{4\}, \{6\}, \{5\}, \\ \{8\}, \{2\}, \{9\}, \{3\}, \{10\}, \{1\}) \end{cases} \quad \begin{cases} S_6 = \{1, 2, 5, 6, 7, 8\} \\ \overline{V}_6 = \{1, 2, 3, 4, 9, 10\} \\ C_6 = \{(2, 3), (3, 10)\} \\ Tr = (\{7\}, \{4\}, \{8\}, \{6\}, \\ \{5\}, \{2\}, \{9\}, \{3\}, \{10\}, \{1\}) \end{cases} \quad \begin{cases} S_{10} = \{7, 10\} \\ \overline{V}_{10} = \{1, 2, 3, 4, 5, \\ 6, 8, 9\} \\ C_{10} = \{(3, 2), \\ (2, 6), (2, 9), (6, 1), \\ (6, 5), (6, 8), (5, 4)\} \\ Tr = (\{10\}, \{7\}, \\ \{3\}, \{2\}, \{9\}, \{6\}, \\ \{8\}, \{5\}, \{4\}, \{1\}) \end{cases}$$

RTH-p commence par identifier les destinations problématiques.

Pour cela, il construit tout d'abord $G_{C_4} = (V, C_4)$, le graphe de contraintes générées pour la destination 4 avec V l'ensemble de tous les nœuds du réseau, illustré dans la figure 2.8(a). Ensuite, l'ensemble des contraintes générées C_6 pour la destination 6 est ajouté à ce graphe (figure 2.8(b)). Ces contraintes étant incluses dans les précédentes, le nouveau graphe est identique à l'ancien et ainsi aucune boucle n'est détectée. Les destinations 4 et 6 sont alors considérées comme non problématiques. Enfin, l'ensemble des contraintes générées pour la destination 10 est à son tour ajouté à ce graphe. Comme le montre la figure 2.8(c), une boucle est détectée entre le nœuds 2 et 3. RTH-p retire alors C_{10} et considère 10 comme une destination problématique. En conclusion, RTH-p traite les destinations $\{4, 6\}$ comme un groupe et la destination 10 toute seule.

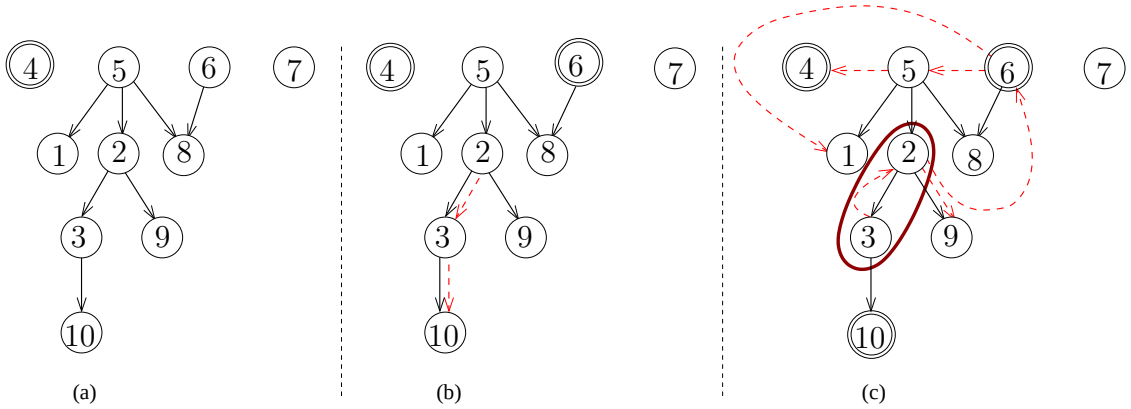

FIGURE 2.8 – (a) Graphe des contraintes générées pour la destination 4, (b) graphe des contraintes générées pour les destinations 4 et 6, (c) graphe des contraintes générées pour les destinations 4, 6, et 10.

RTH-p calcule ensuite l'ordre de migration des nœuds :

$$\begin{cases} S_1 = \{4,5,6,7\}_{4,6} \\ S_2 = \{1,2,8\}_{4,6} \\ S_3 = \{3,9\}_{4,6} \\ S_4 = \{10\}_{4,6} \end{cases} \text{ pour les destinations } \{4,6\} \text{ et}$$

$$\begin{cases} S_5 = \{3,7,10\}_{10} \\ S_6 = \{2\}_{10} \\ S_7 = \{6,9\}_{10} \\ S_8 = \{1,5,8\}_{10} \\ S_9 = \{4\}_{10} \end{cases} \text{ pour la destination 10.}$$

La transition finale est donc : $Tr = (\{4,5,6,7\}_{4,6}, \{1,2,8\}_{4,6}, \{3,9\}_{4,6}, \{10\}_{4,6}, \{3,7,10\}_{10}, \{2\}_{10}, \{6,9\}_{10}, \{1,5,8\}_{10}, \{4\}_{10})$ composée de 9 étapes.

Discussion : Il convient de relever que le résultat de RTH-p est lié au résultat de l'algorithme de détection des destinations problématiques. Or ce dernier dépend fortement de l'ordre de traitement des destinations. En effet, reprenons notre exemple avec l'ensemble des destinations sélectionnées à savoir $\{4, 6, 10\}$. Comme illustré précédemment dans la figure 2.8, si ces destinations sont traitées dans l'ordre $(4, 6, 10)$, la destination problématique est la destination 10. Considérons à présent l'ordre $(10, 6, 4)$. Les destinations problématiques sont alors 6 et 4. En effet, comme le montre la figure 2.9, lorsqu'on ajoute les contraintes générées par la destination 6 au graphe de contraintes

générées par la destination 10, la boucle $(3, 2, 3)$ est générée, ce qui classe 6 comme destination problématique. De même, ajouter les contraintes générées par la destination 4 au graphe des contraintes générées par la destination 10 génère les boucles : $(5, 2, 6, 5)$ et $(3, 2, 3)$, ce qui fait de la destination 4 également une destination problématique. RTH-p va ainsi calculer l'ordre de migration de la destination non problématique 10, puis répéter le processus pour la destination 6 et enfin pour la destination 4. La transition finale produite pour cet ordre de destinations est la suivante : $Tr = (\{3, 7, 10\}_{10}, \{2\}_{10}, \{6, 9\}_{10}, \{1, 5, 8\}_{10}, \{4\}_{10}, \{1, 2, 5, 6, 8, 4, 7\}_6, \{3, 9\}_6, \{10\}_6, \{5, 6, 4, 7\}_4, \{1, 2, 8\}_4, \{3, 9\}_4, \{10\}_4)$, composée de 12 étapes.

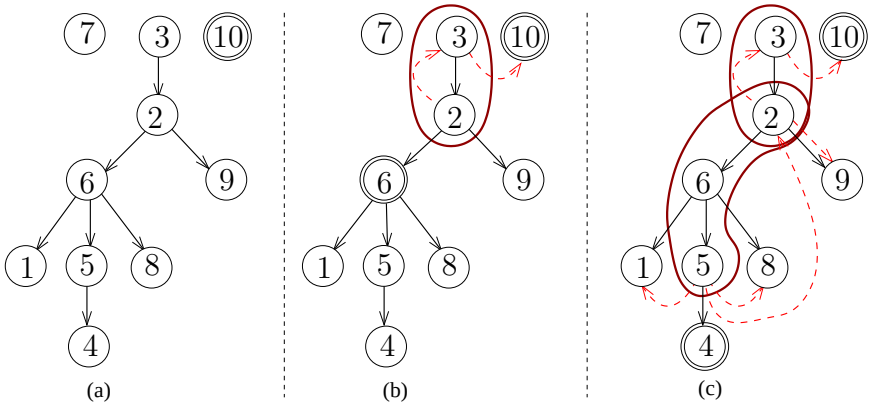

FIGURE 2.9 – (a) Graphe des contraintes générées pour la destination 10, (b) graphe des contraintes générées pour les destinations 10 et 6, (c) graphe des contraintes générées pour les destinations 10 et 4.

En somme, l'ordre $(4, 6, 10)$ produit une seule destination problématique et une transition de 9 étapes tandis que l'ordre $(10, 6, 4)$ produit deux destinations problématiques et une transition de 12 étapes. Cette transition est beaucoup plus longue car l'algorithme de détection n'a pas su regrouper les destinations non problématiques au mieux.

Limites

Malgré l'utilisation du parallélisme pour permettre la migration de plusieurs nœuds ensembles, les transitions produites par RTH-p restent relativement longues dû au fait que le nombre de destinations problématiques croît avec la taille du réseau.

Complexité

En termes de temps, pour un réseau de n nœuds et d destinations, RTH-p a une complexité de l'ordre de $\mathcal{O}(d.(n^2 + n) + n^2)$ avec $\mathcal{O}(d.(n^2 + n))$ pour la construction du graphe de contraintes et du parcours des chemins par destination, et $\mathcal{O}(n^2)$ pour la parallélisation.

En termes d'étapes, pour un réseau de n nœuds avec $d \in [1; n]$ destinations parmi lesquelles $t \in [1; d-1]$ destinations problématiques, RTH-p produit une transition d'au plus $t.n$ étapes. RTH-p a ainsi pour complexité $\mathcal{O}(n^2)$ étapes.

2.3 Heuristiques basées sur l'augmentation du poids des liens

Dans cette partie, nous nous intéressons aux événements de retrait programmé d'un nœud dans un réseau. Dans cette configuration, les poids des liens ne sont pas changés par l'événement. Néanmoins, une solution pour forcer le routage à éviter le nœud que l'on souhaite supprimer est d'augmenter le poids des liens autour de ce nœud, jusqu'à l'infini. En augmentant ces poids graduellement, les boucles peuvent être évitées. Nous étudions dans la suite deux algorithmes proposés dans la littérature pour le cas d'un retrait programmé d'un nœud : GBA et AGBA.

2.3.1 GBA : *Greedy Backward Algorithm*

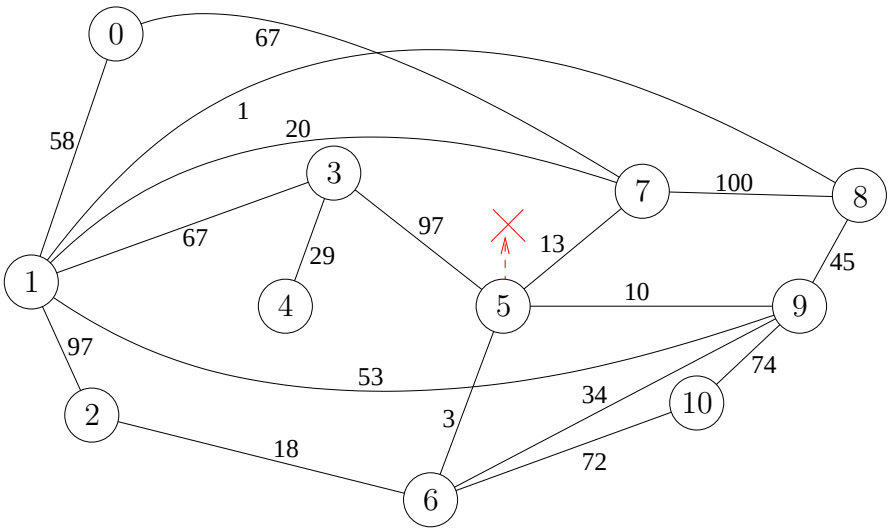

FIGURE 2.10 – Exemple du réseau Sprint avec le retrait programmé du nœud 5.

GBA [43] a été proposé dans le cadre de la gestion des mises à jour de routeurs pour le cas particulier de protocoles à état de liens.

Si l'on considère comme opération de mise à jour la suppression d'un nœud, celle-ci peut être vue comme une application d'un ensemble d'incréments aux poids dudit nœud avec ses voisins, de sorte que ce nœud ne soit inclus dans aucun plus court chemin vers une destination donnée. Le but de GBA est de calculer un ensemble minimal d'incréments à appliquer dans un tel cas. Pour ce faire, pour chaque destination, GBA extrait des contraintes associées à chacune des boucles transitoires susceptibles d'apparaître, boucles identifiées en faisant l'union des chemins de routage initial et final. Une contrainte extraite associée à une boucle est un couple $l = (\underline{l}, \bar{l})$ défini comme suit :

$$l = (\underline{l}, \bar{l}) : \begin{cases} \underline{l} := min_{\forall x \in L}(\Delta_d(x)) \\ \bar{l} := max_{\forall x \in L}(\Delta_d(x)) \\ |\underline{l}| = |\bar{l}| = k \end{cases}$$

avec L l'ensemble des nœuds formant la boucle, x un nœud appartenant à L, k le degré de x, d une destination, et $\Delta_d(x)$ le vecteur de poids à appliquer pour avoir les mêmes coûts de plus courts chemins suivant le protocole de routage initial $\mathcal{R}_i$ et suivant le protocole de routage final $\mathcal{R}_f$.

Calcul du vecteur de poids $\Delta_d(x)$: $\Delta_d(x)$ se définit comme suit :
$\forall x \in V, \Delta_d(x) = C'(x,d) - C(x,d)$ avec $C'(x,d)$ le plus court chemin de x à d suivant le routage $\mathcal{R}_f$, et $C(x,d)$ le plus court chemin de x à d suivant le routage initial $\mathcal{R}_i$.

Considérons un nœud x avec k voisins. $\forall i \in [1,k], \Delta_d^r(x)[i] = C'(x,d) - C(x,l_i,d)$ avec r le nœud à retirer, $C'(x,d)$ le coût du plus court chemin de x vers la destination d lorsque le nœud a été retiré, et $C(x,l_i,d)$ le coût (avant retrait du nœud) du chemin de x à la destination d en passant par le lien l_i entre le nœud r et son $i^{ème}$ voisin. $\Delta_d(x)$ représente la valeur à ajouter au poids du lien entre le nœud x et son $i^{ème}$ voisin pour que le coût du plus court chemin de x vers d suivant $\mathcal{R}_i$ et passant par ce $i^{ème}$ voisin soit égal à celui du plus court chemin de x vers d suivant $\mathcal{R}_f$. Cette valeur est le seuil du poids à appliquer pour chaque voisin pour que le nœud x passe par ce voisin pour atteindre la destination.

Pour l'illustrer, considérons le réseau de la figure 2.10 avec 5 le nœud à supprimer. Supposons que la destination choisie est le nœud 10 et le nœud inclus dans une boucle est le nœud 1 (la boucle $(1,7,1)$ par exemple). Le plus court chemin de 1 vers 10 avec le nœud 5 supprimé est $ch = 1-8-9-10$ dont le coût est $C'(1) = 120$.

Voisin v_i de $r=5$	$C(r,v_i,d)$	$C(1,(5,v_i),10)$	$\Delta_{10}^5(1)[i]$
3	284	317	120-317= -197
6	75	108	120-108= +12
7	153	186	120-186= -66
9	84	117	120-117= +3

TABLE 2.1 – Tableau de calcul des $\Delta_{10}(x)$ avec x, voisin du nœud 5 dans le graphe de la figure 2.11.

D'après le tableau 2.1, $\Delta_d(1) = [-197, +12, -66, +3]$ ce qui signifie que si on applique -197 au poids du lien $(5,3)$ (resp. $+12$ à $(5,6)$, -66 à $(5,7)$ et $+3$ à $(5,9)$), le coût du chemin vers 10 *via* tout voisin de 5 suivant le routage initial aura la même valeur que ce coût suivant le routage final $\mathcal{R}_f$. Ces valeurs nous donnent des seuils de migration. En effet, il suffit de trouver un vecteur positivement supérieur à $\Delta_d(x)$ pour faire basculer le nœud x sur $\mathcal{R}_f$. Nous rappelons que dans [43], un vecteur V_1 est dit positivement supérieur à un vecteur V_2 noté $V_1 >^+ V_2$ si :

$$\forall i \in [1,k] : \begin{cases} V_1[i] > V_2[i] \text{ si } V_2[i] \geq 0 \\ V_1[i] \geq 0 \text{ si } V_2[i] < 0 \end{cases}$$

Ainsi, le vecteur $(0,13,0,4)$, positivement supérieur à $\Delta_{10}(1)$, fait basculer le nœud 1 de $\mathcal{R}_i$ à $\mathcal{R}_f$ car par application de ce vecteur, les nouveaux coûts sont :

$$\begin{cases} C(1,(5,3),10) = 317 \\ C(1,(5,6),10) = 121 \\ C(1,(5,7),10) = 186 \\ C(1,(5,9),10) = 121 \end{cases} \text{ tous supérieurs à 120, coût suivant } \mathcal{R}_f.$$

Calcul de la séquence minimale : Lorsque la contrainte l associée à chaque boucle transitoire identifiée a été calculée, GBA produit alors une séquence d'incréments à appliquer pour que le retrait du nœud se fasse sans génération de ces potentielles boucles de routage identifiées. Pour ce faire, GBA définit la notion de satisfaction

d'une contrainte l associée à une boucle. Un vecteur v satisfait une contrainte $l = (\underline{l}, \bar{l})$ si $v >^+ \underline{l}$ et $\exists i, v[i] < \bar{l}[i]$. Pour produire la séquence d'incréments à appliquer, GBA calcule à chaque itération un vecteur v qui vérifie les deux propriétés suivantes.

— Pour toutes les contraintes encore non satisfaites à l'itération j, GBA calcule v tel que $\forall l = (\underline{l}, \bar{l}), v >^+ \underline{l}$,

— Il existe au moins une contrainte encore non satisfaite qui vérifie la condition $\exists i, v[i] = max(\underline{l}[i] + 1, 0)$.

GBA répète ce processus jusqu'à ce que toutes les contraintes soient satisfaites et la séquence constituée de tous les vecteurs produits à chaque itération est la séquence minimale à appliquer.

Calcul de la transition finale : Le résultat que produit GBA est une séquence de vecteurs de poids à appliquer aux liens entre le nœud à retirer et ses voisins. Pour déterminer dans quel ordre les nœuds migrent, il est nécessaire de transformer la séquence de poids en séquence de nœuds à faire migrer. En utilisant la propriété observée dans [43] selon laquelle les boucles générées par le retrait d'un nœud sont toutes de taille 2, nous avons mis en place une transformation qui suit les règles suivantes :

— Une itération correspond à une étape. Ainsi, les nœuds migrant à la même itération appartiennent à la même étape dans la transition finale.

— Lorsqu'une contrainte $l = (\underline{l}, \bar{l})$ est satisfaite à une itération par un vecteur v, le nœud de la boucle liée à cette contrainte et associé à $\underline{l}$ est migré car $v >^+ \underline{l}$. Le nœud de la boucle associé à $\bar{l}$ est quant à lui migré à l'itération suivante.

— Le processus est répété jusqu'à ce que toutes les contraintes soient satisfaites, ce qui résulte en la migration de tous les nœuds.

Exemple

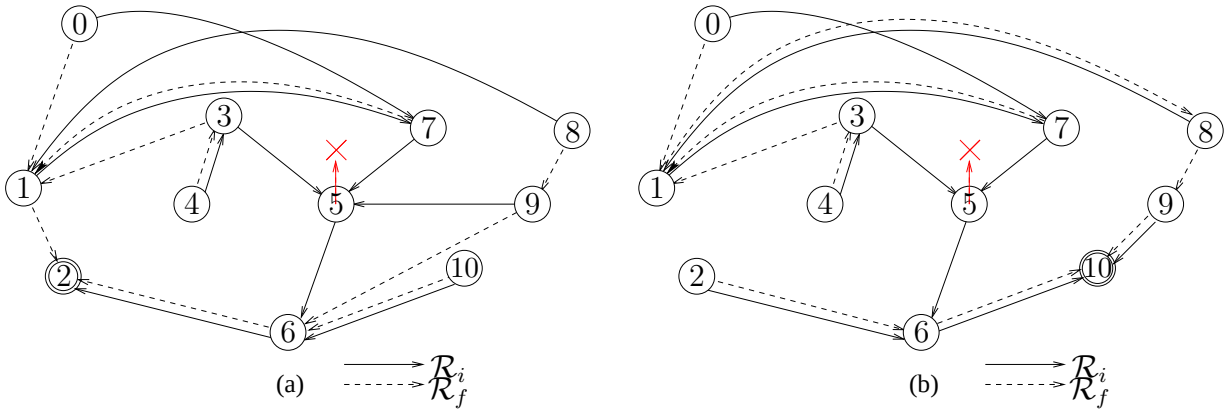

FIGURE 2.11 – (a) Graphe union du routage initial et du routage final vers la destination 2, (b) graphe union du routage initial et du routage final vers la destination 10.

Prenons l'exemple de la figure 2.10. L'ensemble des boucles pour les destinations 2 et 10 est : $\begin{cases} l_1 = \{1,7\}_2 \\ l_2 = \{1,7\}_{10} \\ l_3 = \{1,8\}_{10} \end{cases}$. Les contraintes calculées pour chacune des boucles sont les suivantes :

$$\begin{cases} c_1 = ([-197, 43, -66, 2], [-157, 83, -26, 42]) \\ c_2 = ([-197, 12, -66, 3], [-157, 52, -26, 43]) \\ c_3 = ([-199, 10, -68, 1], [-197, 12, -66, 3]) \end{cases}$$

GBA produit comme séquence d'incréments à appliquer $S = ([0, 11, 0, 2], [0, 44, 0, 4])$. Le premier vecteur de S $[0, 11, 0, 2]$, satisfait la contrainte c_3 car d'après [43] :

$$\begin{cases} [0, 11, 0, 2] >^{+} \underline{l_3}([-199, 10, -68, 1]) \\ \exists i, v[i] < \bar{l_3} \text{ car } 11 < 12 \end{cases}$$

Étant donné que $\underline{l_3}$ est associée au nœud 8 et $\bar{l_3}$ au nœud 1, le nœud 8 sera basculé en premier tandis que 1 continuera de router suivant $\mathcal{R}_i$. La boucle $(1, 8, 1)_{10}$ sera ainsi évitée. De même, le second vecteur de S $[0, 44, 0, 4]$ satisfait à la fois les contraintes c_1 et c_2. En effet, $[0, 44, 0, 4] >^{+} [-197, 43, -66, 2]$ $(\underline{l_1})$ avec $44 < 83$ $(\bar{l}[2])$ entraînant la satisfaction de la contrainte c_1, et $[0, 44, 0, 4] >^{+} [-197, 12, -66, 3]$ $(\underline{l_2})$ avec $4 < 43$ $(\bar{l}[4])$ entraînant la satisfaction de la contrainte c_2. Dans les deux cas, $\underline{l}$ est associée au nœud 1 et $\bar{l}$ est associée au nœud 7. Le nœud 1 sera ainsi basculé en premier pour les deux destinations 2 et 10, ensuite suivra le nœud 7. En somme, la transition correspondante est la suivante : Tr=$(\{2, 4, 5, 6, 10\}_{2,10} \cup \{8, 9\}_{10}, \{1\}_{2,10}, \{0, 3, 7\}_{2,10} \cup \{8, 9\}_{2})$. Les trois boucles sont évitées et le nœud 5 peut être supprimé sans provoquer de boucles transitoires.

Limites

La principale limite de GBA est que certaines séquences produites peuvent conduire à faire migrer des nœuds durant la phase transitoire vers des prochains sauts n'existant ni sur le routage initial $\mathcal{R}_i$, ni sur le routage final $\mathcal{R}_f$. Les auteurs ont qualifié ce problème de changements intermédiaires d'acheminement. Reprenons l'exemple précédent de la figure 2.11 pour les destinations 2 et 10 et supposons que GBA produit la séquence suivante : $S = ([0, 11, 0, 2], [0, 46, 0, 4])$ qui est une séquence valide. Cette dernière conduit à un changement intermédiaire de routage. En effet, le second vecteur d'incréments $[0, 46, 0, 4]$, en plus de casser la boucle $(1, 7, 1)$, conduit le nœud 5 à router vers le nœud 9 pour la destination 2, alors que 9 n'est ni le prochain saut de 5 suivant $\mathcal{R}_i$, ni suivant $\mathcal{R}_f$. Ces changements peuvent dans certains cas causer des boucles transitoires qui ne seront pas détectées en faisant l'union des routages initial et final. Ces boucles incluent toujours le nœud à supprimer.

Complexité

Pour un réseau de n nœuds avec r le nœud à supprimer, et E l'ensemble des liens du réseau, les auteurs scindent le calcul de la complexité de GBA en deux étapes.

1. La complexité de l'initialisation liée à la fusion des deux routages initial et final et l'extraction des contraintes : $\mathcal{O}(n.(n.\log(n)) + |E|)$.
2. Le nombre d'itérations nécessaires pour satisfaire toutes les contraintes est de l'ordre de $\mathcal{O}(n^2)$ dans le pire des cas. Dans la pratique, ce nombre est limité à une valeur fixe p. La complexité de la boucle principale de GBA est ainsi détaillée telle que suit :
   — le calcul du vecteur d'incréments en $\mathcal{O}(p.k.n)$ avec k le degré de r,
   — le calcul du prochain vecteur d'incréments à appliquer en $\mathcal{O}(min(p.n.|E|, n^2.|E|))$.

Ainsi, dans le pire des cas, GBA est en $\mathcal{O}(n^4)$ si le routeur à supprimer est connecté à tout le reste du réseau. En pratique, les auteurs réduisent cette complexité en fixant la taille maximale des séquences générées à $p \leq 5$, ce qui la ramène à $\mathcal{O}(n^3)$.

2.3.2 AGBA : *Adjusted Greedy Backward Algorithm*

AGBA [43] est une variation de GBA proposée pour palier au problème des changements intermédiaires d'acheminement. L'objectif d'AGBA est d'ajouter certaines contraintes pour que les nœuds soient obligés de router soit suivant $\mathcal{R}_i$, soit suivant $\mathcal{R}_f$. Pour ce faire, les auteurs ont tout d'abord défini la notion de *successeurs initiaux* qui représentent plusieurs prochains sauts du nœud qui tombe en panne sur ses plus courts chemins vers une destination donnée. Cet ensemble est noté S^*. Chaque poids du vecteur à appliquer aux liens autour du nœud à retirer doit permettre à S^* de rester inchangé, c'est-à-dire qu'aucun autre nœud ne doit devenir successeur initial par application d'un incrément durant la transition. Pour ce faire, les notions suivantes sont introduites.

— $v[x]$ est la composante d'un vecteur v associé au lien (r, x) où x est un nœud voisin du nœud à retirer r. Si l'on considère le vecteur $v = [0, 11, 0, 2] \in S$ produit par GBA pour l'exemple de la figure 2.10, 0 est associé au lien $(5, 3)$, 11 est associé au lien $(5, 6)$, 0 est associé au lien $(5, 7)$ et enfin 2 est associé au lien $(5, 9)$.

— $\text{Offset}_d^r(x) = C(r, x, d) - C(r, d)$ où $C(r, x, d)$ est le coût du plus court chemin sur $\mathcal{R}_i$ de r vers d passant par son voisin x, et $C(r, d)$ celui du plus court chemin de r vers d indépendamment du voisin choisi. L'offset définit l'attractivité d'un nœud qui est la valeur à ajouter au coût du lien entre le nœud et le routeur à retirer pour qu'il devienne successeur initial. Si $\text{Offset}_d^r(x) = 0$ alors x est déjà successeur initial de r. Pour notre cas de la figure 2.10, on a les résultats suivants :

$$\textit{Destination } 2 : \begin{cases} C(5,2) = 21 \\ \text{Offset}_2^5(3) = C(5,3,2) - C(5,2) = 261 - 21 = 240 \\ \text{Offset}_2^5(6) = C(5,6,2) - C(5,2) = 21 - 21 = 0 \\ \text{Offset}_2^5(7) = C(5,7,2) - C(5,2) = 130 - 21 = 109 \\ \text{Offset}_2^5(9) = C(5,9,2) - C(5,2) = 62 - 21 = 41 \end{cases}$$

$$\textit{Destination } 10 : \begin{cases} C(5,10) = 75 \\ \text{Offset}_{10}^5(3) = C(5,3,10) - C(5,10) = 272 - 75 = 197 \\ \text{Offset}_{10}^5(6) = C(5,6,10) - C(5,10) = 75 - 75 = 0 \\ \text{Offset}_{10}^5(7) = C(5,7,10) - C(5,10) = 219 - 75 = 144 \\ \text{Offset}_{10}^5(9) = C(5,9,10) - C(5,10) = 84 - 75 = 9 \end{cases}$$

Ainsi, le nœud 5 ne possède qu'un seul successeur initial, le nœud 6. L'idée étant de maintenir S^* inchangé, plus un nœud a son offset petit, moins la valeur du vecteur à associer sera importante. L'offset aide à contrôler le calcul des valeurs d'incréments de poids de manière à garantir que l'incrémentation des poids ne conduit pas à des prochains sauts non valides.

— Définition des CPC *(Change Prevention Conditions)*

$\begin{cases} v[s] = v[s^*] \\ v[x] > v[s^*] - \text{Offset}_d^r(x) \end{cases}$ pour $s \in S^*(d)$, $x \notin S^*(d)$, avec $S^*(d)$ l'ensemble des successeurs initiaux vers la destination d du nœud r à retirer. r' est successeur initial de r si le lien (r, r') appartient au routage initial. La valeur d'incrément pour tous les successeurs initiaux de r est la même et celle d'un voisin non successeur initial doit être supérieure à la différence entre son offset et l'incrément d'un successeur initial.

Une fois ces notions introduites, intéressons nous à la génération de la séquence d'incréments à appliquer. AGBA est divisé en deux parties.

1. L'identification de l'ensemble S^* des successeurs initiaux de r,
2. La mise à jour de la séquence d'incréments de poids produite par GBA de sorte que toutes les CPC triées soient satisfaites de la façon suivante :
$\begin{cases} v[s] = m_d \\ v[x] = m_d - \text{Offset[d][x]} + 1 \end{cases}$ où $s \in S^*(d), x \notin S^*(d)$ et $m_d = max_{\forall s \in S^*(d)}(v[s])$.

Exemple

Considérons les routages de la figure 2.11 vers les destinations 2 et 10 avec pour nœud à retirer le nœud 5. L'identification de l'ensemble des successeurs initiaux produit $S^* = \{6\}$ car $\text{Offset}_2^5(6) = 0$ et $\text{Offset}_{10}^5(6) = 0$. Considérons la séquence qui génère un changement intermédiaire de routage à savoir : $S = ([0, 11, 0, 2], [0, 46, 0, 4])$. AGBA traite chaque vecteur de la séquence :

1. Vecteur $[0, 11, 0, 2]$: le pivot (poids maximal) $m_d = 11$ est associé à l'unique successeur initial 6.

$$\textit{Destination } 2 : \begin{cases} v[3] = 11 - 240 < 0 => v[3] = 0 \\ v[6] = 11 \text{ car } 6 \in S^* \\ v[7] = 11 - 109 < 0 => v[7] = 0 \\ v[9] = 11 - 41 < 2 => v[9] = 2 \end{cases}$$

$$\textit{Destination } 10 : \begin{cases} v[3] = 11 - 197 < 0 => v[3] = 0 \\ v[6] = 11 \text{ car } 6 \in S^* \\ v[7] = 11 - 144 < 0 => v[7] = 0 \\ v[9] = 11 - 9 => v[9] = 3 \end{cases}$$

A partir du vecteur $[0, 11, 0, 2]$ de GBA, AGBA produit les vecteurs $[0, 11, 0, 2]$ et $[0, 11, 0, 3]$.

2. Vecteur $[0, 46, 0, 4]$: le pivot $m_d = 46$ est associé à l'unique successeur initial 6.

$$\textit{Destination } 2 : \begin{cases} v[3] = 46 - 240 < 0 => v[3] = 0 \\ v[6] = 46 \text{ car } 6 \in S^* \\ v[7] = 46 - 109 < 0 => v[7] = 0 \\ v[9] = 46 - 41 < 2 => v[9] = 6 \end{cases}$$

$$\textit{Destination } 10 : \begin{cases} v[3] = 46 - 197 < 0 => v[3] = 0 \\ v[6] = 46 \text{ car } 6 \in S^* \\ v[7] = 46 - 144 < 0 => v[7] = 0 \\ v[9] = 46 - 9 => v[9] = 38 \end{cases}$$

A partir du vecteur $[0, 46, 0, 4]$ de GBA, AGBA produit les vecteurs $[0, 46, 0, 6]$ et $[0, 46, 0, 38]$.

En somme pour la séquence initiale $([0, 11, 0, 2], [0, 46, 0, 4])$, AGBA produit la séquence $([0, 11, 0, 2], [0, 11, 0, 3], [0, 46, 0, 6], [0, 46, 0, 38])$. Cette séquence permet d'éviter les trois boucles l_1, l_2 et l_3 et empêche le nœud 5 de router temporairement vers le nœud 9.

Limites

AGBA résout à la fois le problème d'évitement des boucles de routage induits par le retrait d'un nœud et le problème des changements intermédiaires de routage causés par GBA. Cependant, les séquences qu'il génère sont beaucoup plus longues que celles générées par GBA, entraînant une migration plus lente.

Conclusion

Dans ce chapitre, nous nous sommes attelés à présenter en détails les heuristiques et algorithmes de la littérature qui traitent du problème des boucles de routage. Il en ressort que le changement de protocoles peut émaner soit d'une modification des poids des liens du réseau dû à un événement survenu dans la vie du réseau qui conduit à un recalcul des routes, soit le retrait d'un nœud ou d'un lien qui oblige également le recalcul des routes, avec pour nœuds les plus affectés ceux qui sont proches de l'opération de modification. Nous avons classé les différentes propositions de la littérature essentiellement en deux grands groupes.

— *Les heuristiques basées sur le routage final :* les heuristiques de cette catégorie construisent leurs solutions sur la base de l'ordre défini par le routage final. Nous avons présenté RTH [42] qui définit un ensemble de contraintes sur les nœuds de telle sorte qu'un nœud n'est autorisé à migrer que si tous ses successeurs vers la destination l'ont déjà fait. Puis, ces heuristiques calculent un tri topologique sur le graphe de contraintes obtenu et le résultat de ce tri topologique est l'ordre final de migration des nœuds. Sa particularité est le fait que la migration est faite nœud par nœud pour chaque destination, ce qui conduit à un grand nombre d'étapes. Nous avons également présenté RTH-p [13], une variation de RTH qui apporte deux améliorations principales : la proposition d'un algorithme de détection des destinations problématiques introduites dans RTH et une accélération de la transition en permettant une migration parallèle des nœuds sans causer de boucles de routage.

— *Les heuristiques basées sur l'augmentation des poids des liens :* les heuristiques de cette catégorie traitent du cas particulier du retrait des nœuds ou liens dans un réseau. Ces heuristiques assimilent ces opérations de retrait à une application d'une séquence de vecteurs d'incréments aux poids des nœuds ou liens impliqués. L'objectif est que ces liens soient très attractifs dans le cas de l'ajout, et non importants pour le routage car empruntés par aucun nœud lors du retrait. Nous avons ainsi présenté GBA [43], un algorithme glouton qui identifie toutes les boucles possibles en faisant l'union des routages initial et final, puis calcule des contraintes pour chaque nœud impliqué dans une boucle et enfin calcule la séquence minimale d'incréments à appliquer pour un retrait/ajout sans boucle. Nous avons également présenté AGBA [43], une variation de GBA qui, en plus de gérer le problème des boucles de routage durant la transition, traite aussi le problème de changements intermédiaires de routage causé dans certains cas par les séquences produites par GBA en imposant de nouvelles contraintes qui obligent les nœuds à router suivant $\mathcal{R}_i$ ou $\mathcal{R}_f$.

Deuxième partie

Contributions

Chapitre 3

Heuristiques dans un contexte statique centralisé

Sommaire

Introduction

Le changement d'un protocole de routage initial à un protocole de routage final peut conduire à des boucles de routage qui peuvent dégrader considérablement les performances du réseau. Les contributions de ce chapitre considèrent un contexte statique et centralisé. Une entité centrale qui a une connaissance de la totalité de la topologie du réseau, et des protocoles de routage avant changement ($\mathcal{R}_i$) et après changement ($\mathcal{R}_f$), est considérée.

Notre objectif est de proposer des heuristiques qui permettent de définir un ordre de migration des nœuds qui permet d'éviter la génération des boucles transitoires de routage et ce, en réduisant autant que possible le temps de migration.

Dans ce chapitre, nous présentons les deux heuristiques centralisées que nous avons proposées : *Strongly Connected Component Heuristic with merged steps* (SCH-m), et *Avoiding Cycle Heuristic* (ACH).

3.1 SCH-m : *Strongly Connected Component Heuristic with merged steps*

Dans cette partie, nous présentons notre première heuristique d'évitement des boucles de routage dans le contexte centralisé à savoir : *Strongly Connected Component Heuristic with merged steps* (SCH-m). SCH-m s'appuie sur l'heuristique existante SCH-p *(Strongly Connected component Heuristic with parallel changes)*. C'est pourquoi nous commencerons par détailler le fonctionnement de SCH-p.

3.1.1 SCH-p : *Strongly Connected component Heuristic with parallel changes*

SCH-p [13] est basée sur les composantes fortement connexes (C_i) dans un réseau. Si l'on connaît le routage initial $\mathcal{R}_i$ et le routage final $\mathcal{R}_f$ d'un réseau, alors les boucles transitoires possibles sont identifiables dans les composantes fortement connexes de l'union de ces deux routages. La figure 3.1(a) montre l'exemple d'un routage initial vers la destination 6 dans un réseau de 9 nœuds, la figure 3.1(b) montre l'exemple du routage final, et la figure 3.1(c) montre l'union des deux routages avec identification des composantes fortement connexes marquées en rouge : $C_1 = \{6\}$, $C_2 = \{2, 3\}$, et $C_3 = \{1, 4, 5, 7, 8, 9\}$. Cette union contient la boucle $\{(2, 3, 2)\}$ contenue dans C_2, et les boucles $\{(1, 4, 1),\ (5, 9, 8, 5),\ (5, 9, 8, 7, 5),\ (4, 5, 9, 8, 7, 4),\ (1, 5, 9, 8, 7, 4, 1)\}$ contenues dans C_3.

SCH-p considère les destinations séquentiellement et chacune individuellement. SCH-p construit tout d'abord l'union des protocoles de routage $\mathcal{R}_i$ et $\mathcal{R}_f$. Puis, SCH-p calcule chaque composante fortement connexe (cf annexe A.9) de cette union. Ensuite, SCH-p traite parallèlement toutes les composantes connexes calculées, en identifiant dans chacune d'elles les nœuds qui peuvent effectuer la migration ensemble sans causer de boucles. Ce traitement se fait de manière gloutonne. Si un nœud dans une composante ne peut être migré à l'itération i, il est de nouveau traité à l'itération $i + 1$. Tant qu'il existe un nœud non encore migré, le nombre d'itérations est incrémenté. Les nœuds identifiés comme pouvant migrer ensemble dans toutes les composantes connexes à la même itération forment une étape de la transition finale calculée par SCH-p. Cette identification est décrite par l'algorithme 3.

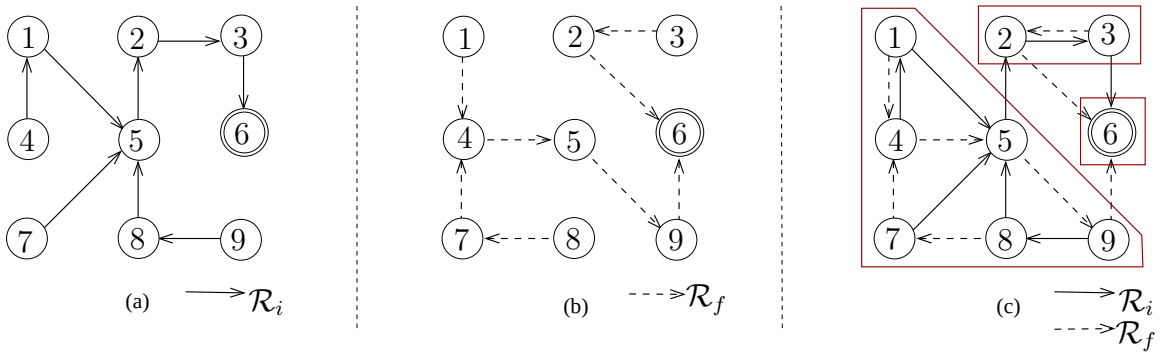

FIGURE 3.1 – (a) Le routage initial $\mathcal{R}_i$ vers la destination 6, (b) le routage final $\mathcal{R}_f$ vers la destination 6, (c) l'union des deux routages $\mathcal{R}_i$ et $\mathcal{R}_f$ vers la destination 6 : identification des composantes fortement connexes marquées en rouge à savoir $C_1 = \{6\}$, $C_2 = \{2, 3\}$ et $C_3 = \{1, 4, 5, 7, 8, 9\}$.

Dans l'algorithme 3, SCH-p traite séquentiellement, à chaque itération, chaque nœud n de la composante fortement connexe qui n'a pas encore changé de protocole de routage. Soient C_i la composante fortement connexe considérée, *dest* la destination, $C'_{i,j}$ l'ensemble des nœuds de C_i qui peuvent effectuer la migration ensemble sans causer de boucles de routage à l'itération j, et n le nœud en cours de traitement avec $n \notin C'_{i,j}$. SCH-p commence par construire un graphe G' ayant pour nœuds tous les nœuds du réseau. Puis, SCH-p ajoute les arcs suivant la règle suivante : $(m, \mathcal{R}_f(m))$ pour tout nœud m ayant migré avant l'itération j, à la fois $(m, \mathcal{R}_i(m))$ et $(m, \mathcal{R}_f(m))$ pour tout nœud m appartenant à l'union $C'_{i,j} \cup \{n\}$ avec j l'itération courante, et $(m, \mathcal{R}_i(m))$ pour tout autre nœud m restant excepté la destination *dest*. Une fois G' construit, SCH-p teste l'existence des boucles. Si G' n'a pas de boucle, alors le nœud n en cours de traitement est ajouté à $C'_{i,j}$. SCH-p reprend ce processus jusqu'à ce que tous les nœuds de C_i non encore migrés aient été traités.

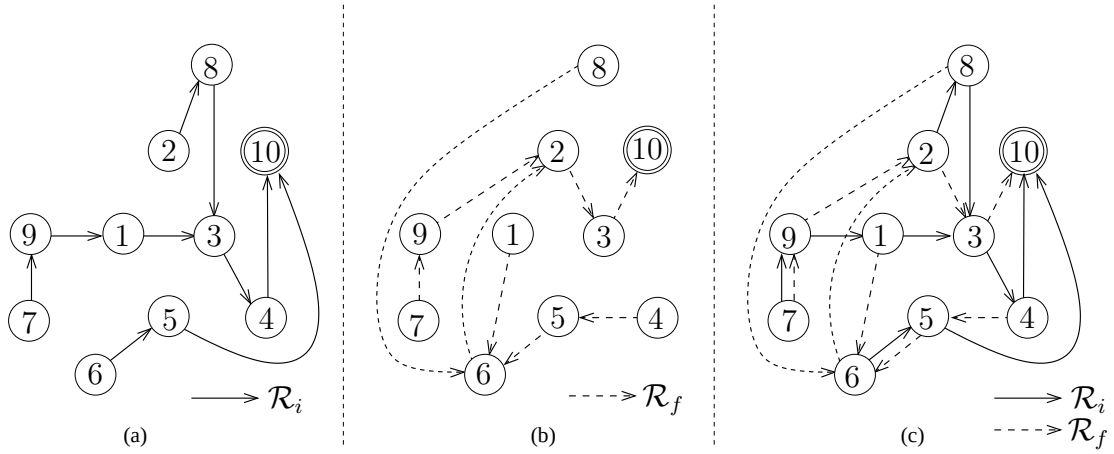

FIGURE 3.2 – (a) Le routage initial $\mathcal{R}_i$ vers la destination 10, (b) le routage final $\mathcal{R}_f$ vers la destination 10, (c) l'union des deux routages $\mathcal{R}_i$ et $\mathcal{R}_f$ vers la destination 10 durant la phase transitoire du changement de protocole qui génère les boucles : $(2, 8, 6, 2)$, $(2, 3, 4, 5, 6, 2)$, $(2, 8, 3, 4, 5, 6, 2)$, et $(5, 6, 5)$.

La figure 3.2 reprend l'exemple du routage initial (figure 3.2(a)), du routage final (figure 3.2(b)), et de l'union des deux routages (figure 3.2(c)) vers la destination 10 du réseau de la figure 2.3 présentée dans la partie 2.2. L'union des deux protocoles de routage aboutit aux quatre boucles suivantes : $(2, 8, 6, 2)$, $(2, 3, 4, 5, 6, 2)$, $(2, 8, 3, 4, 5, 6, 2)$,

Algorithme 3 : Identification des nœuds pouvant migrer ensemble dans une composante connexe par SCH-p.

Données : C_i une composante fortement connexe, *dest* une destination, E_liste la liste des étapes précédentes, V l'ensemble des nœuds.
Résultat : $C'_{i,j}$ un ensemble de nœuds de C_i

```
1  C'_{i,j} ← ∅
2  fin ← faux
3  tant que (fin = faux) faire
4      fin ← vrai
5      pour tout nœud n ∈ C_i faire
6          si n ∉ E_liste et n ∉ C'_{i,j} alors
7              Construire G' = (V', E')
8              V' ← V
9              E' ← ∅
10             pour tout nœud m ∈ E_liste faire
11                 ajouter (m, R_f(m)) à E'
12             fin
13             pour tout nœud m ∈ C'_{i,j} ∪ {n} faire
14                 ajouter (m, R_i(m)) à E'
15                 ajouter (m, R_f(m)) à E'
16             fin
17             pour les autres nœuds m restants excepté dest faire
18                 ajouter (m, R_i(m)) à E'
19             fin
20             si G' ne contient pas de boucle alors
21                 C'_{i,j} ← C'_{i,j} ∪ {n}
22                 fin ← faux
23             fin
24         fin
25     fin
26     retourner C'_{i,j}
27 fin
```

et $(5, 6, 5)$. La figure 3.3 présente dans les détails la procédure d'identification des nœuds dans une composante fortement connexe par SCH-p sur cet exemple. SCH-p calcule tout d'abord les composantes fortement connexes suivantes : $\{2, 3, 4, 5, 6, 8\}$, $\{1\}, \{7\}, \{9\}$, et $\{10\}$ identifiées dans la figure 3.3(a). Les nœuds $\{1\}, \{7\}, \{9\}, \{10\}$ appartiennent chacun à une composante fortement connexe de taille 1, et peuvent donc migrer ensemble. SCH-p les considère comme les premiers nœuds de la première étape de sa transition. C'est pourquoi dans la figure 3.3(b), ils sont grisés et illustrés chacun avec leur prochain saut suivant le routage initial $\mathcal{R}_i$, et leur prochain saut suivant le routage final $\mathcal{R}_f$ pour signifier le caractère transitoire de cette phase. Considérons que SCH-p débute son processus d'identification des nœuds qui migrent dans la composante fortement connexe $\{2, 3, 4, 5, 6, 8\}$ par le nœud 2. Le graphe construit en ajoutant le nœud 2 comme potentiel nœud effectuant la migration (grisage et ajout de son prochain saut suivant $\mathcal{R}_f$) ne contient aucune boucle de routage comme le montre la figure 3.3(c). Le nœud 2 est ajouté à l'étape contenant déjà l'ensemble des nœuds $\{1, 7, 9, 10\}$. SCH-p fait de même avec les nœuds 8, 3, et 4 qui n'engendrent pas non plus de boucles de routage par ajout de leur prochain saut suivant $\mathcal{R}_f$ comme le montre la figure 3.3(d). Lorsque SCH-p ajoute le nœud 5 à l'ensemble des nœuds qui peuvent migrer ensemble à savoir $\{1, 2, 3, 4, 7, 8, 9, 10\}$, la boucle $(5, 6, 5)$ est générée comme le montre la figure 3.3(e). Le nœud 5 n'est par conséquent pas ajouté à cet ensemble. SCH-p traite enfin pour cette composante fortement connexe le nœud 6. Son ajout génère la boucle $(6, 2, 8, 6)$ tel qu'illustré dans la figure 3.3(f). Il n'est pas ajouté non plus à l'étape. SCH-p achève le traitement de la composante fortement connexe $\{2, 3, 4, 5, 6, 8\}$, et migre dans cette étape les nœuds 2, 3, 4, et 8, ainsi que les nœuds $1, 7, 9, 10$.

A la seconde itération, SCH-p examine les nœuds non encore migrés. Les nœuds 5 et 6 ne pouvant être migrés ensemble dans une même étape, nous supposons que SCH-p examine en premier le nœud 6, puis le nœud 5. La figure 3.4 illustre ces différents traitements qui ne génèrent aucune boucle de routage.

SCH-p pour les réseaux multi-destinations

Reprenons l'exemple multi-destinations de la partie 2.2 avec pour destinations les nœuds 4 (figure 2.6), 6 (figure 2.7) et 10 (figure 2.4). SCH-p calcule la transition finale de chaque destination indépendamment des autres. Ainsi :

- Destination 4 : SCH-p identifie une composante fortement connexe de taille supérieure à 1 à savoir $\{2, 3, 5, 6, 8, 10\}$, et les composantes de taille 1. SCH-p produit donc la transition $Tr_4 = (\{1, 2, 4, 5, 6, 7, 8, 9\}_4, \{3\}_4)$ qui compte deux étapes.
- Destination 6 : SCH-p identifie la composante fortement connexe $\{3, 4, 10\}$ en plus de celles de taille 1, et produit donc la transition de deux étapes, $Tr_6 = (\{1, 2, 3, 4, 5, 6, 7, 8, 9\}_6, \{10\}_6)$.
- Destination 10 : SCH-p identifie la composante fortement connexe $\{2, 3, 4, 5, 6, 8\}$, et celles de taille 1. SCH-p produit donc la transition $Tr_{10} = (\{1, 2, 3, 4, 7, 8, 9, 10\}_{10}, \{6\}_{10}, \{5\}_{10})$ de trois étapes.

En somme, SCH-p pour cet exemple calcule la transition finale en concaténant toutes les transitions et produit : $Tr = (\{1, 2, 4, 5, 6, 7, 8, 9, 10\}_4, \{3\}_4, \{1, 2, 3, 4, 5, 6, 7, 8, 9\}_6, \{10\}_6, \{1, 2, 3, 4, 7, 8, 9, 10\}_{10}, \{6\}_{10}, \{5\}_{10})$ qui compte 7 étapes. Nous rappelons que pour le même exemple, l'heuristique RTH [42] produit une transition de 30 étapes, et l'heuristique RTH-p [13] produit une transition de 9 étapes.

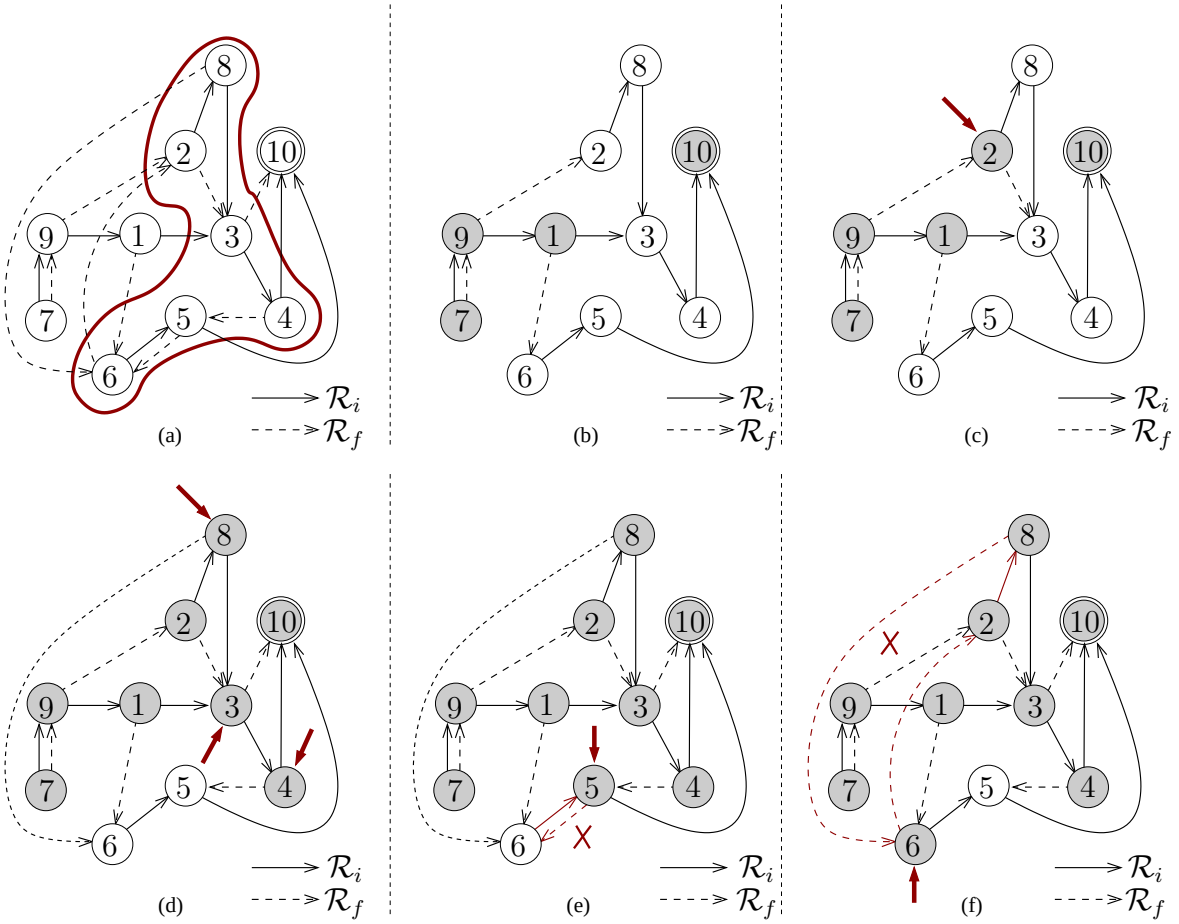

FIGURE 3.3 – Exemple d'identification des nœuds qui migrent ensemble par SCH-p dans une composante fortement connexe. (a) Identification de la composante fortement connexe $\{2, 3, 4, 5, 6, 8\}$ dans l'union des routages, (b) le routage des nœuds 1, 7, 9, et 10 en cours de migration et le routage des nœuds non encore traités, (c) traitement du nœud 2, (d) traitement des nœuds 3, 4, et 8, (e) traitement du nœud 5, (f) traitement du nœud 6.

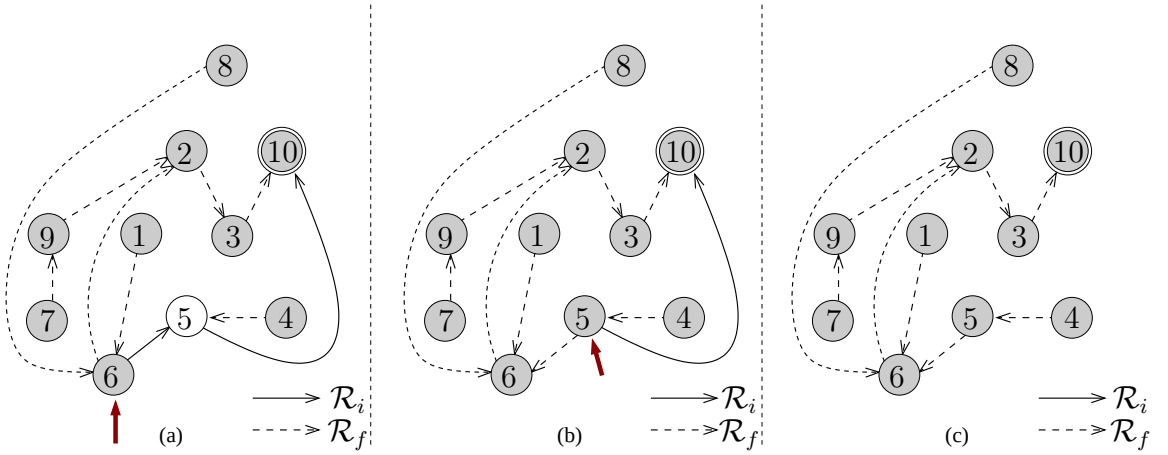

FIGURE 3.4 – (a) Traitement par SCH-p du nœud 6 à la seconde itération, (b) traitement par SCH-p du nœud 5 à la troisième itération, (c) routage lorsque tous les nœuds du réseau ont migré.

Discussion

Le nombre d'étapes de la transition dépend de l'ordre de traitement des nœuds par SCH-p lors de la phase d'identification des nœuds dans une composante connexe. Le choix d'un nœud peut exclure un ou plusieurs autres nœuds de la composante d'une migration à l'itération courante, rallongeant ainsi le nombre d'étapes. A contrario, le choix d'un autre nœud peut favoriser la formation d'une étape avec un grand nombre de nœuds pouvant migrer à l'itération courante, ce qui réduit le nombre d'étapes produites. Pour l'illustrer, reprenons l'exemple de la figure 3.3 et supposons que lors du traitement de la composante fortement connexe $\{2, 3, 4, 5, 6, 8\}$, après traitement du nœud 2, le nœud 6 est traité avant le nœud 8. Il est ajouté à l'ensemble des nœuds qui migrent à l'itération courante, de même que les nœuds 3 et 4, parce que ne générant aucune boucle de routage comme le montre la figure 3.5(d). Les figures 3.3(e) et 3.3(f) montrent que l'ajout du nœud 5 ou 8 génère respectivement les boucles $(5, 6, 5)$ et $(8, 6, 2, 8)$. SCH-p ne migre ainsi à cette itération que les nœuds 2, 3, 4, et 6 avec $1, 7, 9, 10$ au lieu de 2, 3, 4, 5, et 8 (ou 2, 3, 4, 6 et 8) avec $1, 7, 9, 10$ si le nœud 8 est traité après le nœud 2.

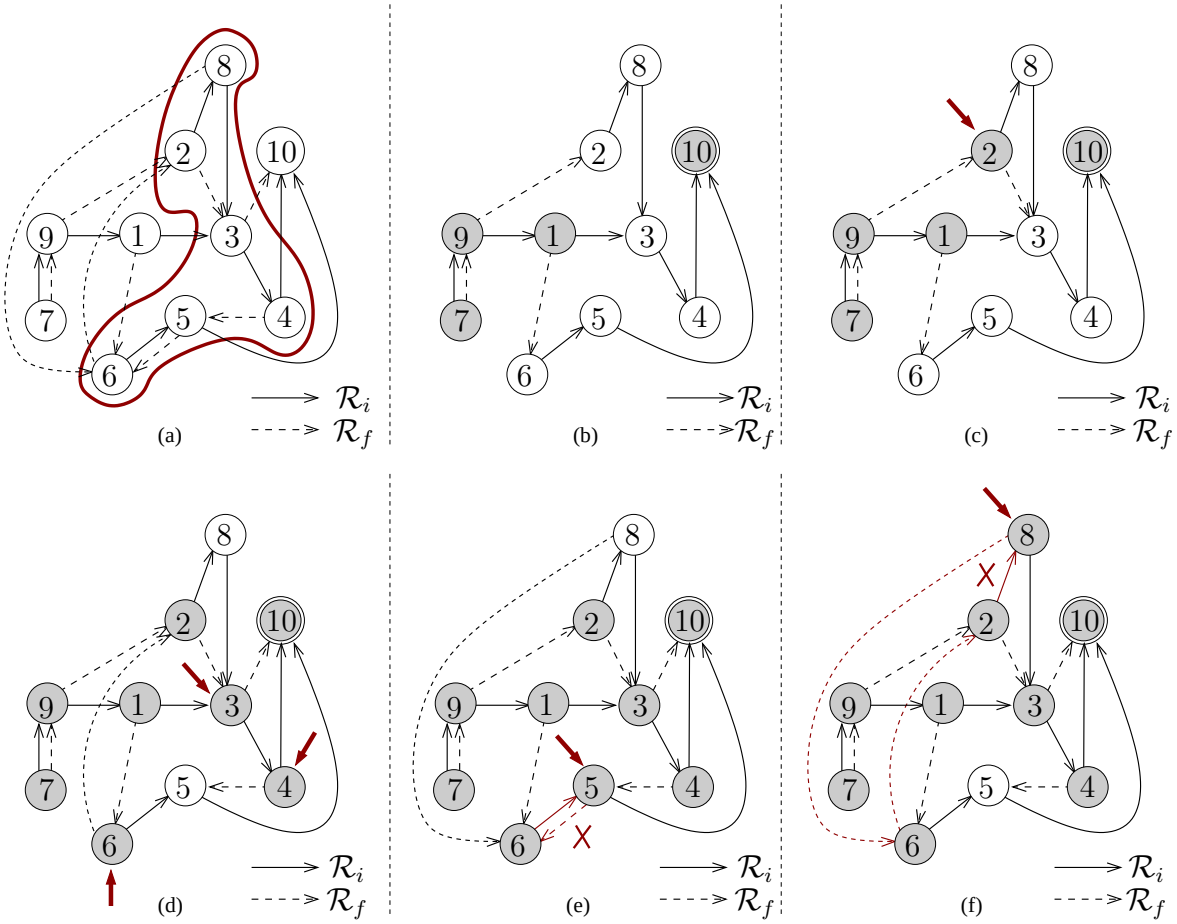

FIGURE 3.5 – Exemple d'identification des nœuds par SCH-p dans une composante fortement connexe. (a) Identification des composantes fortement connexes dans l'union des routages, (b) routage des nœuds $\{1, 7, 9, 10\}$ en cours de migration et routage initial des nœuds non encore traités, (c) traitement du nœud 2, (d) traitement des nœuds 3, 4, et 6, (e) traitement du nœud 5 et ajout impossible, (f) traitement du nœud 8 et ajout impossible.

À la seconde itération, les nœuds 5 et 8 continuent de router suivant $\mathcal{R}_i$ tandis que les nœuds $\{1, 2, 3, 4, 6, 7, 9, 10\}$ routent déjà suivant $\mathcal{R}_f$ (figure 3.6(a)). Lorsque SCH-p teste l'ajout du nœud 5, puis du nœud 8, on n'observe aucune génération de boucles

de routage comme le montre la figure 3.6(b). Les deux nœuds sont migrés ensemble et le processus d'identification est stoppé. Ainsi, en choisissant de traiter le nœud 6 avant le nœud 8, SCH-p produit la transition $Tr = (\{1, 2, 3, 4, 6, 7, 9, 10\}_{10}, \{5, 8\}_{10})$ de deux étapes, ce qui est un gain d'une étape par rapport à choisir de traiter le nœud 8 avant le nœud 6, qui produit la transition $Tr = (\{10, 8, 2, 4, 3, 1, 9, 7\}_{10}, \{6\}_{10}, \{5\}_{10})$ de trois étapes.

SCH-p n'a pas de politique de choix des nœuds car ils sont traités de manière arbitraire (par identifiant croissant).

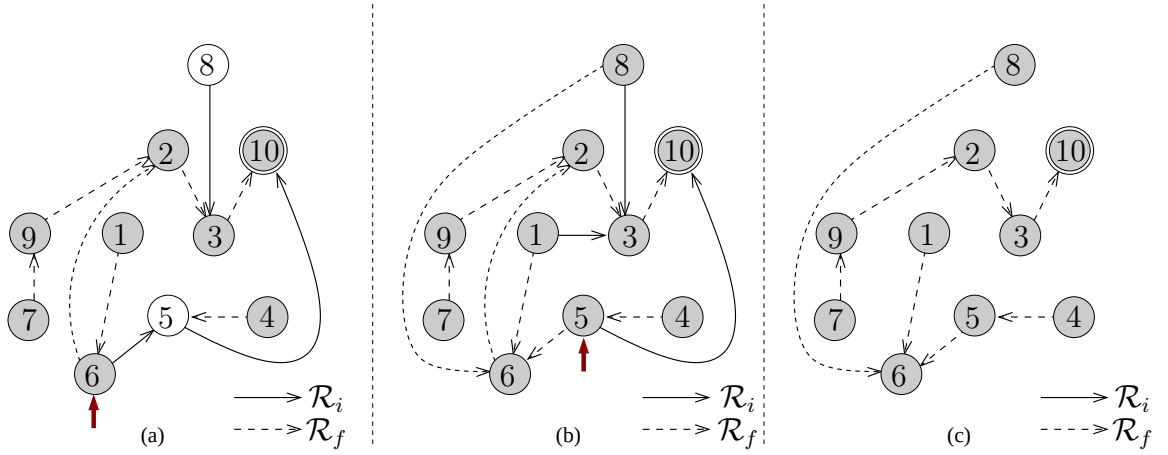

FIGURE 3.6 – (a) Traitement par SCH-p du nœud 6 à la seconde itération, (b) traitement par SCH-p du nœud 5 à la troisième itération, (c) routage lorsque tous les nœuds du réseau ont migré.

Limites

La principale limite de SCH-p est le traitement individuel des destinations. En effet, SCH-p concatène les transitions de chaque destination, ce qui peut générer un grand nombre d'étapes si le nombre de destinations est grand.

Complexité

Pour un réseau de n nœuds avec $d \in [1; n]$ destinations, SCH-p a une complexité de $\mathcal{O}(n^4)$. En effet les destinations sont traitées individuellement. Pour chaque destination, les composantes fortement connexes peuvent être calculées à l'aide de l'algorithme de Tarjan [44] en $\mathcal{O}(n+m)$ avec m le nombre d'arcs ($m \leq 2.n$). Pour chaque composante connexe, calculer $C'_{i,j}$ nécessite $|C_i|^2$ combinaisons de nœuds qui nécessitent chacune de construire un graphe G' et de déterminer s'il contient une boucle. Cela se fait en $\mathcal{O}(|C_i|)$. D'où une complexité de $\mathcal{O}(n^4)$.

3.1.2 Description de SCH-m

Nous avons proposé SCH-m [45] pour palier à la limite de SCH-p dans le cas multi-destinations. En effet, dans un réseau à multi-destinations, SCH-p calcule la transition de chaque destination individuellement, aboutissant à un nombre d'étapes qui dépend du nombre de destinations.

SCH-m propose de considérer et de traiter toutes ces destinations comme une seule destination, en appliquant une fusion des étapes pour construire la transition finale.

SCH-m calcule tout d'abord chaque transition produite par SCH-p pour chacune des destinations impliquées. Ensuite, SCH-m fait une opération de fusion étape par étape des différentes transitions calculées. Plus formellement, soit $etape(i, j)$ l'ensemble des nœuds de la $i^{ème}$ étape de la transition produite par SCH-p pour la destination j. La $i^{ème}$ étape de la transition de SCH-m est obtenue en fusionnant la $i^{ème}$ étape de chaque transition par destination : $etape(i) = \cup_{j=1}^{d} etape(i, j)$ où d est le nombre de destinations.

Reprenons l'exemple multi-destinations de la section 2.2 avec pour destinations les nœuds 4 (figure 2.6), 6 (figure 2.7) et 10 (figure 2.4). Les transitions produites pour chacune des destinations sont les suivantes :

— Destination 4 : $Tr_4 = (\{1, 2, 4, 5, 6, 7, 8, 9, 10\}_4, \{3\}_4)$,
— Destination 6 : $Tr_6 = (\{1, 2, 3, 4, 5, 6, 7, 8, 9\}_6, \{10\}_6)$,
— Destination 10 : $Tr_{10} = (\{1, 2, 3, 4, 7, 8, 9, 10\}_{10}, \{6\}_{10}, \{5\}_{10})$.

Pour les destinations 4 et 6, SCH-p produit des transitions de deux étapes et pour la destination 10, une transition de trois étapes. SCH-m produit ainsi une transition finale de trois étapes.

$$\begin{cases} etape(1) = \{1, 2, 4, 5, 6, 7, 8, 9, 10\}_4 \cup \{1, 2, 3, 4, 5, 6, 7, 8, 9\}_6 \cup \{1, 2, 3, 4, 7, 8, 9, 10\}_{10} \\ etape(2) = \{3\}_4 \cup \{10\}_6 \cup \{6\}_{10} \\ etape(3) = \{5\}_{10} \end{cases}$$

La transition finale est : $Tr = (\{1, 2, 4, 7, 8, 9\}_{4,6,10} \cup \{5, 6\}_{4,6} \cup \{3\}_{6,10} \cup \{10\}_{4,10}, \{3\}_4 \cup \{10\}_6 \cup \{6\}_{10}, \{5\}_{10})$.

3.2 ACH : *Avoiding Cycle Heuristic*

ACH [46] est notre deuxième heuristique d'évitement des boucles de routage centralisée basée sur les composantes fortement connexes comme les heuristiques SCH-p et SCH-m. Son but est d'accélérer le temps de migration en réduisant autant que possible le nombre des étapes générées. Pour ce faire, ACH cherche à maximiser le nombre de nœuds qui peuvent effectuer la transition à une même itération dans une composante fortement connexe, ce qui permet de réduire le nombre total d'étapes.

Nous présentons dans un premier temps le fonctionnement de ACH dans le cas d'une destination, puis dans un second temps son fonctionnement dans le cas multi-destinations.

3.2.1 ACH pour les réseaux mono-destination

ACH s'appuie sur SCH-m, mais remplace l'algorithme glouton de choix des nœuds à faire migrer dans les composantes connexes. Pour maximiser le nombre de nœuds pouvant effectuer la migration ensemble dans une même composante connexe, ACH introduit la notion de *casser un cycle*. En effet, un cycle dans l'union des deux routages conduit à une boucle de routage si et seulement si chacun des nœuds du cycle route à l'intérieur du cycle. Alors, la notion de *casser un cycle* pour ACH revient à trouver l'ensemble des nœuds qui routent suivant le protocole initial $\mathcal{R}_i$ en dehors du cycle, car de tels nœuds permettent d'éviter la boucle : tant qu'ils ne migrent pas, le cycle ne contient pas une boucle. Plus formellement, soient n un nœud et C un cycle. n *casse* C si $n \in C$, et $\mathcal{R}_i(n) \notin C$.

ACH construit tout d'abord l'union $\mathcal{R}_i \cup \mathcal{R}_f$ des deux protocoles de routage. Puis, ACH calcule toutes les composantes connexes qui en résultent. Pour chaque compo-

Algorithme 4 : Identification des nœuds pouvant migrer ensemble dans une composante connexe par ACH.

Données : SCC une composante fortement connexe, $\mathcal{R}_i$ le protocole de routage initial, $\mathcal{R}_f$ le protocole de routage final.
Résultat : liste des nœuds de SCC qui migrent ensemble.

```
S ← ∅
pour chaque cycle élémentaire C ∈ SCC faire
    pour chaque nœud n ∈ C faire
        si R_i^n ∉ C alors
            n casse C
        fin
    fin
    si C est cassé par un unique nœud n alors
        ajouter n à S
    fin
fin
répéter
    n ← le nœud de SCC\S qui casse le plus de cycles élémentaires non encore cassés
    ajouter n à S
jusqu'à chaque cycle C est cassé par un nœud de S;
retourner SCC\S
```

sante fortement connexe, ACH calcule tous les cycles élémentaires inclus dans cette dernière. Puis, ACH identifie le nœud à sélectionner pour effectuer la migration. Cette identification se fait en suivant les règles suivantes : (1) lorsqu'un cycle ne contient qu'un seul nœud qui le casse, alors ce nœud est par défaut sélectionné, (2) dans le cas où il existe plusieurs nœuds candidats pour casser encore les cycles, le nœud sélectionné est celui qui casse le plus grand nombre de cycles non cassés. Le nœud sélectionné est obligé de continuer de router suivant $\mathcal{R}_i$ tandis que les autres nœuds inclus dans le cycle peuvent migrer sans causer de boucles de routage. ACH répète ce processus d'identification tant qu'il existe un cycle élémentaire non cassé par un nœud. L'algorithme 4 décrit ce processus d'identification.

La figure 3.7 présente le processus d'identification des nœuds par ACH dans une petite composante fortement connexe de 4 nœuds : $\{1, 2, 3, 4\}$. La figure 3.7(a) montre que ACH identifie deux cycles à savoir $(1, 2, 3, 1)$ et $(2, 3, 4, 2)$. Comme le montre la figure 3.7(b), les potentiels candidats pour casser le cycle $(1, 2, 3, 1)$ sont les nœuds 2 et 3, tandis que les potentiels candidats pour casser le cycle $(2, 3, 4, 2)$ sont les nœuds 2 et 4. Le candidat 2 respecte la deuxième règle car il permet de casser les deux cycles à la fois. Il est élu vainqueur du processus de sélection comme l'illustre la figure 3.7(c). Ainsi le nœud 2 sera bloqué et continue de router suivant $\mathcal{R}_i$ tandis que les nœuds 1, 3, et 4 sont autorisés à migrer ensemble dans une même étape. La migration des nœuds de cette composante fortement connexe se fait donc en 2 étapes, ce qui correspond à la configuration optimale. En effet, pour cette composante fortement connexe, on pourrait avoir trois configurations possibles :

1. une migration en quatre étapes : chaque nœud migre dans une seule étape (comme avec l'heuristique RTH) suivant l'ordre 1, 3, 2 puis 4,
2. une migration en trois étapes : deux nœuds migrent ensemble dans une même

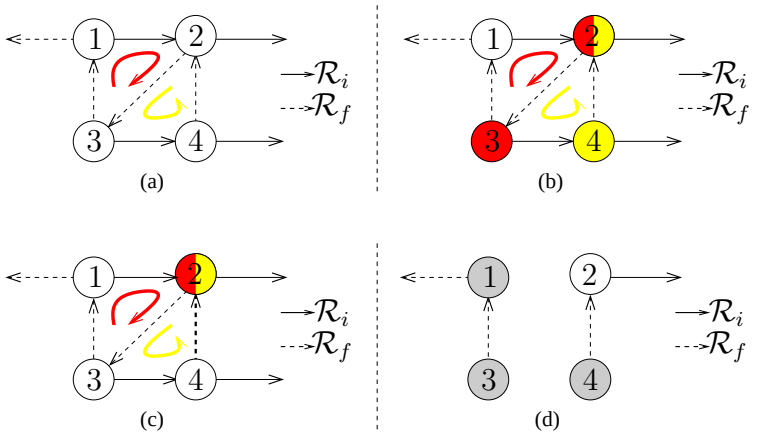

FIGURE 3.7 – Traitement d'une composante fortement connexe par ACH. (a) Identification des cycles élémentaires, (b) identification des candidats pour casser les cycles élémentaires, (c) sélection du candidat vainqueur, (d) le candidat vainqueur doit continuer de router suivant $\mathcal{R}_i$ tandis que les autres migrent et routent suivant $\mathcal{R}_f$.

étape et les deux autres chacun dans une étape : par exemple $\{1,3\},\{2\}$ puis $\{4\}$,

3. une migration en deux étapes : la seule configuration possible dans ce cas est $\{1,3,4\},\{2\}$, celle qu'a produit ACH.

Considérons à présent l'exemple complet de la section 2.2 qui traite de la destination 10 (figure 3.2). Dans l'union des routages, ACH trouve comme composante fortement connexe la composante $\{2,3,4,5,6,8\}$. Puis ACH identifie comme cycles élémentaires les cycles : $(3,4,5,6,2,3)$, $(3,4,5,6,2,8,3)$, $(2,8,6,2)$, et $(5,6,5)$. Pour chaque cycle, ACH identifie les potentiels candidats qui cassent ce cycle, comme suit :

$$\begin{cases} \text{cycle}= (3,4,5,6,2,3) \\ \text{candidats}= \{2,5\} \end{cases} \quad \begin{cases} \text{cycle}= (3,4,5,6,2,8,3) \\ \text{candidat}= \{5\} \end{cases}$$

$$\begin{cases} \text{cycle}= (2,8,6,2) \\ \text{candidats}= \{6,8\} \end{cases} \quad \begin{cases} \text{cycle}= (5,6,5) \\ \text{candidat}= \{5\} \end{cases}$$

En raison de la règle (1), ACH choisit prioritairement le nœud 5 qui est le seul candidat pour les cycles $(3,4,5,6,2,8,3)$ et $(5,6,5)$. Ce choix permet de casser en plus des deux cycles précédents, le cycle $(3,4,5,6,2,3)$. Le choix de 5 pour le cycle $(5,6,5)$ implique la migration du nœud 6, qui a pour conséquence que l'unique candidat restant pour le cycle $(2,8,6,2)$ est le nœud 8. ACH fait ainsi migrer dans la même étape les nœuds $\{2,3,4,6\}$, puis dans une seconde étape les nœuds $\{5,8\}$ qui peuvent effectuer la migration ensemble. La transition finale produite par ACH est la transition $Tr = (\{1,2,3,4,6,7,9,10\}_{10}, \{5,8\}_{10})$ de deux étapes, alors que SCH-p fait 3 étapes. Cette transition montre un gain de 33% en nombre d'étapes par rapport à celle produite par SCH-p.

3.2.2 ACH pour les réseaux multi-destinations

Dans un réseau à multi-destinations, ACH, comme SCH-p, calcule tout d'abord chaque transition pour chacune des destinations impliquées. Ensuite, ACH fait une opération de fusion étape par étape des différentes transitions calculées. En appliquant ACH sur l'exemple multi-destinations de la partie 2.2 avec pour destinations les nœuds 4 (figure 2.6), 6 (figure 2.7) et 10 (figure 2.4), nous obtenons les transitions suivantes :

— Destination 4 : $Tr_4 = (\{1,2,4,5,6,7,8,9,10\}_4, \{3\}_4)$.
— Destination 6 : $Tr_6 = (\{1,2,3,4,5,6,7,8,9\}_6, \{10\}_6)$.
— Destination 10 : $Tr_{10} = (\{1,2,3,4,6,7,9,10\}_{10}, \{5,8\}_{10})$.

Pour chacune des destinations, ACH produit une transition de deux étapes. Ainsi la transition finale aura deux étapes.

$$\begin{cases} etape(1) = \{1,2,4,5,6,7,8,9,10\}_4 \cup \{1,2,3,4,5,6,7,8,9\}_6 \cup \{1,2,3,4,6,7,9,10\}_{10} \\ etape(2) = \{3\}_4 \cup \{10\}_6 \cup \{5,8\}_{10} \end{cases}$$

La transition finale est $Tr = (\{1,2,4,6,7,9\}_{4,6,10} \cup \{5,8\}_{4,6} \cup \{10\}_{4,10} \cup \{3\}_{6,10}, \{3\}_4 \cup \{10\}_6 \cup \{5,8\}_{10})$.

Le tableau 3.1 donne une comparaison du nombre d'étapes générées pour cet exemple multi-destinations.

Heuristique	Nombre d'étapes	Gain de ACH
RTH [42]	30	93%
RTH-p [13]	9	77%
SCH-p [13]	7	71%
Contribution 1 (SCH-m) [46]	3	33%
Contribution 2 (ACH) [46]	2	-

TABLE 3.1 – Tableau comparatif du résultat des heuristiques RTH, RTH-p, SCH-p, SCH-m, et ACH sur l'exemple multi-destinations de la partie 2.2.

Complexité

Pour un réseau de n nœuds et d destinations, ACH a une complexité en temps au plus de $\mathcal{O}(c.n^3)$ avec c le nombre total de cycles identifiés. En effet, à l'image de SCH-m, ACH effectue pour chaque destination une décomposition en composantes fortement connexes. Puis dans chaque composante connexe, l'identification et le traitement de chaque cycle élémentaire nécessite une complexité de $\mathcal{O}(n^2)$ (cf algorithme 4). La fusion des étapes par destination se fait en $\mathcal{O}(n.d)$. D'où une complexité en temps de ACH de l'ordre de $\mathcal{O}(c.n^3)$.

En termes d'étapes, ACH a une complexité qui ne dépend ni de n, ni de d. La durée d'une transition produite par ACH est très petite grâce à l'opération de fusion des étapes.

Limites

La principale limite de ACH est le temps de calcul de la transition qui est beaucoup plus grand que celui des autres heuristiques RTH, RTH-p, SCH-p et SCH-m. Cela entraîne un problème de passage à l'échelle. En effet, le nombre de cycles élémentaires dans la fusion des routages peut croître de manière exponentielle avec la taille du réseau. Pour pouvoir sélectionner les meilleurs candidats pour la migration, ACH doit traiter chaque cycle individuellement, ce qui devient en pratique ingérable lorsqu'on a des millions de cycles générés. Cependant nous pensons qu'il est plus important que la transition en elle même soit rapide, car le calcul de la transition peut quant à lui être fait *off-line* puisqu'on est dans le cas statique.

3.3 Résultats

Dans cette partie, nous comparons les performances de nos heuristiques à celles des heuristiques de la littérature.

3.3.1 Métriques de performance

Nous avons retenu six principales métriques pour analyser les performances des heuristiques : le nombre de groupes de destinations compatibles, la durée de la transition en nombre d'étapes, le coût de la transition en messages de contrôle, la congestion, l'équilibrage et le temps de calcul.

Le nombre de groupes de destinations compatibles

Le nombre de groupes de destinations compatibles indique comment une heuristique gère un réseau à multi-destinations. Un groupe de destinations compatibles est un ensemble de destinations qui peuvent être traitées ensemble par une heuristique, sans générer des boucles transitoires de routage. Pour l'heuristique RTH-p, ce nombre est égal à $t + 1$ où t correspond au nombre de destinations problématiques calculées, et 1 correspond à l'ensemble de toutes les destinations non-problématiques qui peuvent être traitées ensemble sans l'apparition de boucles transitoires de routage. Pour l'heuristique SCH-p, ce nombre vaut d avec d le nombre de destinations, car chaque destination est traitée individuellement. Pour SCH-m, ce nombre vaut 1 car SCH-m traite toutes les destinations ensemble. Pour l'heuristique ACH, ce nombre vaut également 1 parce que ACH traite toutes les destinations ensemble. Cette métrique permet de mieux comprendre les résultats des différentes autres métriques.

La durée de la transition

La durée de la transition est notre principale métrique étant donné que l'objectif est non seulement de trouver un ordre de migration des nœuds sans génération de boucles de routage, mais aussi d'avoir des transitions aussi rapides que possible. Cette métrique est calculée comme le nombre total d'étapes nécessaires à une heuristique pour effectuer la migration. Nous supposons ici qu'une étape correspond à une unité de temps où tous les nœuds peuvent être configurés en parallèle et peuvent effectuer la migration dans un ordre arbitraire durant la même étape. La migration est donc plus rapide quand le nombre d'étapes est plus petit.

Le coût de la transition

Le coût de la transition est le nombre de messages de contrôle nécessaires pour effectuer la transition. Le contexte de travail étant un environnement centralisé, nous supposons qu'il existe une entité centrale chargée de notifier de manière individuelle à chaque nœud de migrer au début de chaque étape. Chaque notification nécessite un nombre de messages de contrôle qui est égal au nombre de nœuds intermédiaires sur le plus court chemin entre l'entité centrale et le nœud lui même. Cette entité centrale est sélectionnée ici à la position la plus centrale du réseau de manière à minimiser le nombre de messages.

Considérons la figure 2.3 de la partie 2.2 et supposons que le nœud 1 est l'entité centrale. Le tableau 3.2 présente les différentes distances du nœud 1 aux autres nœuds

du graphe. Supposons aussi que la transition $Tr = (\{1,2,4,6,7,9\}_{4,6,10} \cup \{5,8\}_{4,6} \cup \{10\}_{4,10} \cup \{3\}_{6,10}, \{3\}_4 \cup \{10\}_6 \cup \{5,8\}_{10})$ est la transition calculée par l'heuristique. Les nœuds $\{3,5,8,10\}$ sont basculés dans deux étapes en fonction de la destination, et les nœuds $\{1,2,4,6,7,9\}$ dans une seule étape. Les nœuds basculés dans n étapes requièrent d'être contactés n fois par le nœud central. C'est pourquoi le coût de cette transition est : distance$(1,1)$ + distance$(1,2)$ + 2 × distance$(1,3)$ + distance$(1,4)$ + 2 × distance$(1,5)$ + distance$(1,6)$ + distance$(1,7)$ + 2 × distance$(1,8)$ + distance$(1,9)$ + 2 × distance$(1,10) = 19$.

Nœud	1	2	3	4	5	6	7	8	9	10
Distance	0	1	1	2	1	1	2	2	1	2

TABLE 3.2 – Tableau des distances du nœud central 1 aux autres nœuds du graphe de la figure 2.3.

La congestion

La métrique congestion produite par une transition désigne l'augmentation possible du trafic lors de l'application de la transition calculée par une heuristique. Elle est approchée au moyen de la taille en nombre de nœuds de la plus grande étape de cette transition, car c'est cette étape qui nécessite le plus de paquets générés à une itération pour la configuration des nœuds. Pour la transition $Tr = (\{1,2,4,6,7,9\}_{4,6,10} \cup \{5,8\}_{4,6} \cup \{10\}_{4,10} \cup \{3\}_{6,10}, \{3\}_4 \cup \{10\}_6 \cup \{5,8\}_{10})$, elle correspond à une valeur de congestion de 10.

L'équilibrage

La métrique équilibrage définit la distribution des nœuds dans les étapes produites par une heuristique de manière à diminuer la congestion produite. Cette distribution se fait par test de déplacement d'une partie des nœuds d'une étape considérée trop longue par rapport à une moyenne du nombre de nœuds dans une étape définie en fonction de la taille du réseau, du nombre de destinations et du nombre d'étapes produites. Le test de déplacement est validé lorsque la nouvelle transition ne génère aucune boucle transitoire de routage. Pour la transition $Tr = (\{1,2,4,6,7,9\}_{4,6,10} \cup \{5,8\}_{4,6} \cup \{10\}_{4,10} \cup \{3\}_{6,10}, \{3\}_4 \cup \{10\}_6 \cup \{5,8\}_{10})$ avec une congestion de 10, une version plus équilibrée de cette transition est la suivante : $Tr' = (\{2,6,7\}_{4,6,10} \cup \{5,8\}_{4,6} \cup \{3\}_{6,10}, \{1,4,9,10\}_{4,6,10} \cup \{3\}_4 \cup \{5,8\}_{10})$ avec une congestion de 7. Ainsi on part d'une configuration en nombre de nœuds par étape de 10 et 5, à une configuration de 6 et 7.

Le temps de calcul

Le temps de calcul représente le temps système que met une heuristique à produire la transition finale à appliquer pour que les nœuds changent de protocole de routage sans générer de boucles de routage, sur l'ordinateur de test, qui a pour caractéristiques : un processeur Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, et une mémoire de 2 Go.

3.3.2 Environnement et paramètres de simulations

Les simulations ont été faites sur plusieurs types de graphes.

Pour les graphes connexes aléatoires, les tailles varient entre 10 et 200 nœuds (tableau 3.3). Les nœuds sont déployés uniformément de manière aléatoire dans un espace de dimension $100m \times 100m$. La portée des nœuds a été fixée à $20m$ qui est un paramètre standard dans les réseaux sans fil ad-hoc. Les simulations ont été faites d'autre part sur des topologies réelles nommées *ITZ* (tableau 3.4) collectées sur l'*Internet Topology Zoo* de l'université Adelaide [47], dont les tailles varient entre 9 et 278, et sur des topologies *Rocketfuel* [12] (tableau 3.5) dont les tailles varient entre 79 et 315.

Nous considérons que chaque nœud est une destination potentielle. Pour chaque destination d, nous avons généré le routage initial $\mathcal{R}_i^d$ en attribuant un poids aléatoire dans l'intervalle $[1; 100]$ à chaque lien, et en calculant le chemin le plus court de chaque nœud vers la destination d en utilisant ces poids. De la même manière, nous avons généré le routage final $\mathcal{R}_f^d$ en assignant un poids aléatoire à chaque lien compris dans l'intervalle $[1; 50]$, et en calculant le plus court chemin de chaque nœud vers la destination d. Les simulations sont moyennées sur 100 répétitions, et l'intervalle de confiance sur les courbes est de 95%. L'échelle logarithmique est utilisée pour l'axe des ordonnées dans certains cas.

Taille	Nombre moyen de liens	Diamètre
10	26	6
20	60	9
30	106	11
40	172	12
50	259	11
100	1037	9
150	2351	8
200	4181	8

TABLE 3.3 – Tableau pour les graphes aléatoires.

Nom	Taille	Nombre de liens	Diamètre
Nordu2005	9	18	3
Sprint	11	36	4
Oxford	20	52	7
Garr201005	55	138	7
Switch	74	180	20
Ion	125	292	25
TataNld	145	372	28
GtsCe	149	386	21
Cogentco	197	486	28
condensed_west_europe	278	788	18

TABLE 3.4 – Tableau des topologies *ITZ*.

3.3.3 Résultats sur des topologies aléatoires

Le nombre de groupes de destinations compatibles : La figure 3.8 montre le nombre moyen de groupes de destinations compatibles, pour des réseaux aléatoires de taille variable, et pour les heuristiques RTH-p, SCH-p, SCH-m et ACH. Pour ACH

Nom	Taille	Nombre de liens	Diamètre
Exodus	79	294	10
Ebone	87	322	11
Telstra	104	302	8
AboveNet	138	744	8
Tiscali	161	656	10
Sprint	315	1944	10

TABLE 3.5 – Tableau des topologies *Rocketfuel*.

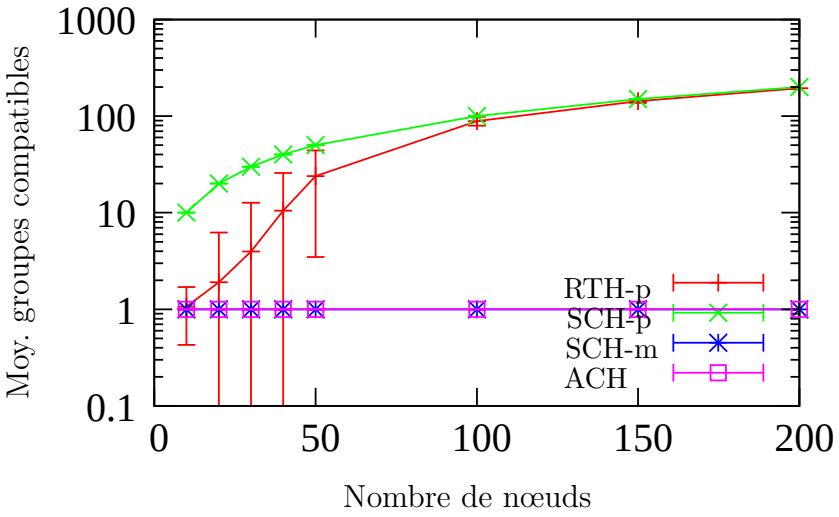

FIGURE 3.8 – Nombre de groupes de destinations compatibles en fonction du nombre de nœuds pour le cas des topologies aléatoires.

et SCH-m, le nombre de groupes de destinations compatibles est toujours égal à 1 parce que toutes les destinations sont traitées ensemble. Pour SCH-p et RTH-p, la figure 3.8 montre que le nombre de groupes de destinations compatibles croît avec la taille du réseau. Pour SCH-p, ce nombre est égal au nombre des destinations d (qui est quant à lui égal à la taille du réseau dans nos simulations), parce que SCH-p traite les destinations individuellement, les unes après les autres. Pour RTH-p, ce nombre est compris entre 1 et d. Pour les réseaux de taille inférieure ou égale à 100, nous remarquons que SCH-p est l'heuristique qui génère le plus de groupes de destinations compatibles. Ceci s'explique par le fait que chaque destination est considérée comme destination problématique, par conséquent, chaque destination représente à elle seule un groupe de destinations compatibles. Mais au delà de la taille 100 du réseau, le nombre de groupes de destinations compatibles des heuristiques SCH-p et RTH-p tend vers le nombre de destinations. Cela signifie que RTH-p génère pour ces tailles presque autant de destinations problématiques qu'il existe de destinations.

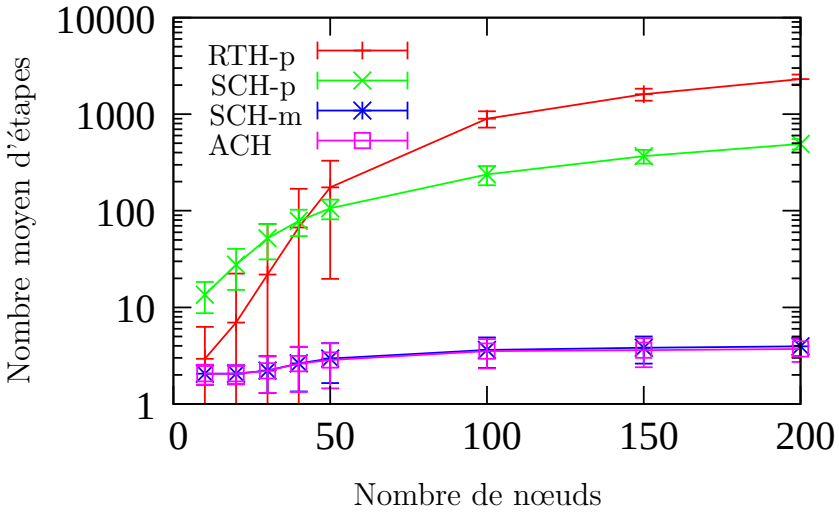

FIGURE 3.9 – Durée de la transition en nombre d'étapes, en fonction du nombre de nœuds pour le cas des topologies aléatoires.

La durée de la transition : La figure 3.9 montre la durée de la transition, calculée comme le nombre moyen d'étapes nécessaires pour effectuer la migration vers toutes les destinations, en fonction des différentes tailles de réseau. Tout d'abord, nous notons que le nombre d'étapes augmente considérablement avec la taille du réseau pour les deux heuristiques RTH-p et SCH-p. Pour RTH-p, cette augmentation est due à deux facteurs : le nombre croissant des groupes de destinations compatibles (figure 3.8), et le fait que le nombre d'étapes nécessaire pour une migration vers une destination avec RTH-p est égal à la profondeur de l'arbre de routage final $\mathcal{R}_f$. Pour SCH-p, les destinations étant traitées individuellement, le nombre d'étapes croît avec la taille du réseau parce que le nombre de destinations croît également. Ensuite, nous notons que pour de très petits réseaux (taille inférieure ou égale à 40), RTH-p est meilleur que SCH-p car produit moins d'étapes. Pour les tailles de réseau supérieures à 40, SCH-p est meilleure que RTH-p car produit moins d'étapes. Cependant, ACH et SCH-m sont capables de produire un petit nombre d'étapes, qui est à la fois peu dépendant du nombre de nœuds et du nombre de destinations. Ce nombre est égal au nombre maximal d'étapes pour toutes les destinations. ACH et SCH-m surpassent RTH-p avec un gain qui varie de 30% pour les réseaux de petite taille à 99% pour les réseaux de

grande taille. De même, ACH et SCH-m montrent un gain par rapport à SCH-p qui varie de 85% pour les réseaux de petite taille à 98% pour les réseaux de grande taille. Enfin, nous notons que ACH montre un léger gain par rapport à SCH-m qui varie de 3% à 6%.

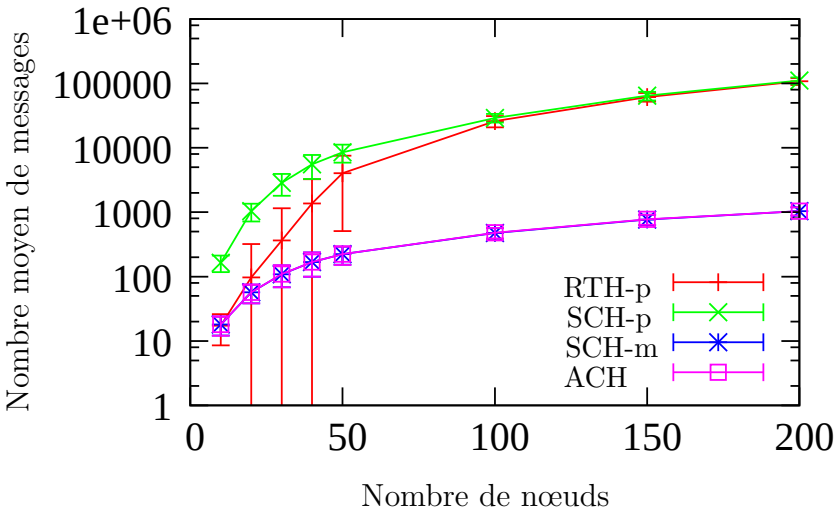

FIGURE 3.10 – Coût de la transition, en fonction du nombre de nœuds pour le cas des topologies aléatoires.

Le coût de la transition : La figure 3.10 montre le coût de la transition, calculée comme le nombre moyen de messages de contrôle nécessaires pour effectuer la migration vers toutes les destinations. Nous observons que pour toutes les heuristiques, le nombre de messages de contrôle augmente avec la taille du réseau. Ceci s'explique par le fait que, plus la taille augmente, plus le nombre de nœuds nécessitant d'être configurés augmente également. Nous observons ensuite que quelle soit la taille du réseau, ACH et SCH-m génèrent un faible coût en nombre de messages comparativement à RTH-p et SCH-p. Sur des réseaux de taille inférieure ou égale à 100, RTH-p est meilleure que SCH-p. Sur des réseaux de taille supérieure à 100, RTH-p et SCH-p produisent des résultats similaires. Ceci se justifie par l'évolution des groupes de destinations problématiques (figure 3.8) : plus ce nombre pour RTH-p se rapproche du nombre de destinations du réseau, plus le nombre de messages nécessaires pour la configuration des nœuds par RTH-p est similaire à celui de SCH-p. Pour ACH et SCH-m, le faible coût en nombre de messages vient du fait que les étapes sont fusionnées. En effet, lorsqu'un nœud apparaît plusieurs fois dans la même étape pour différentes destinations, l'entité centrale ne le notifie qu'une seule fois pour toutes ces destinations. La figure 3.10 montre qu'en terme de coût de la transition, ACH et SCH-m montrent un gain par rapport à RTH-p variant de 1% à 94% pour les réseaux de petite taille, et un gain de 99% pour les réseaux de grande taille. ACH et SCH-m montrent également un gain par rapport à SCH-p allant de 89% pour les réseaux de petite taille à 99% pour les réseaux de grande taille.

La congestion : La figure 3.11 montre la congestion générée par les différentes heuristiques, calculée comme le plus grand nombre de nœuds que peut contenir une étape d'une transition finale. La figure 3.11 montre que les heuristiques SCH-p, SCH-m, et ACH produisent des congestions beaucoup plus élevées que RTH-p. En effet, sur des réseaux de petites tailles, RTH-p montre un gain de 30% à 37% par rapport à ACH, SCH-m et SCH-p, tandis que sur les réseaux de grandes tailles, RTH-p montre un gain

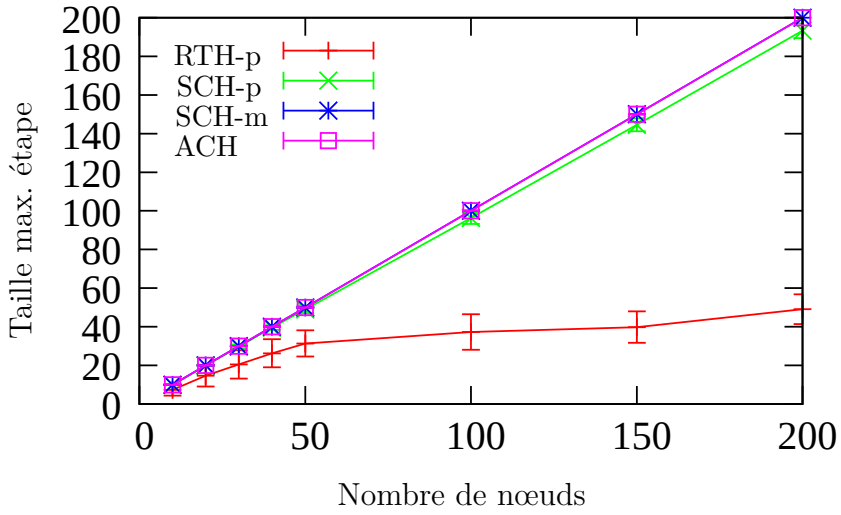

FIGURE 3.11 – Coût de la congestion en fonction du nombre de nœuds pour le cas des topologies aléatoires.

de 63% à 75% par rapport à ACH, SCH-m et SCH-p. Ceci se justifie par le fait qu'en cherchant à réduire au maximum le nombre d'étapes, les heuristiques SCH-m et ACH tendent à concentrer les nœuds dans les étapes. On note néanmoins un léger gain de SCH-p par rapport à ACH et SCH-m qui varie de 0.2% à 4%.

L'équilibrage : Les figures 3.12 et 3.13 montrent la congestion générée respectivement par les heuristiques RTH-p et SCH-p, et par les heuristiques SCH-m et ACH lorsque les transitions ont été équilibrées. La figure 3.12 montre que RTH-p en version équilibrée produit à peu près la même congestion que la version non équilibrée. Cela s'explique par le fait que RTH-p produit un grand nombre d'étapes, et plus le nombre d'étapes est élevé, mieux sont répartis les nœuds dans celles-ci. L'équilibrage n'y a donc pas un grand impact. La figure 3.12 montre également que la congestion est presque la même pour l'heuristique SCH-p car nous notons un gain qui va jusqu'à 7% de congestion en moins après équilibrage. Cependant, la figure 3.13 montre que l'équilibrage permet de diminuer la congestion des heuristiques ACH et SCH-m. En effet, nous notons un gain qui va jusqu'à 30% de congestion en moins pour ACH et un gain qui va jusqu'à 32% de congestion en moins pour SCH-m. RTH-p reste la meilleure heuristique pour les réseaux de taille supérieure à 50 avec un gain qui va jusqu'à 74% par rapport à SCH-m, ACH et SCH-p, exception faite pour les réseaux de taille inférieure ou égale à 50 pour lesquelles ACH et SCH-m montrent un gain de 13% par rapport à RTH-p.

Le temps de calcul : La figure 3.14 montre le temps d'exécution de chacune des heuristiques. Comme le montre la figure, le temps d'exécution de chacune des heuristiques augmente avec la taille du réseau. Ceci s'explique naturellement par le fait que plus la taille du réseau augmente, plus le nombre de nœuds devant être traités augmente. Nous observons ensuite que ACH est l'heuristique qui prend moins de temps d'exécution pour toutes les tailles de réseau inférieures ou égales à 150, et au delà RTH-p devient la plus rapide de toutes. En effet, ACH présente un gain allant de 94% à 99% par rapport à SCH-p et SCH-m, et un gain oscillant entre 20% et 69% par rapport à RTH-p sur les tailles entre 10 et 150. Par contre, pour la taille 200, RTH-p présente un gain de 77% par rapport à ACH, et un gain de 99% par rapport à SCH-p et SCH-m. La justification

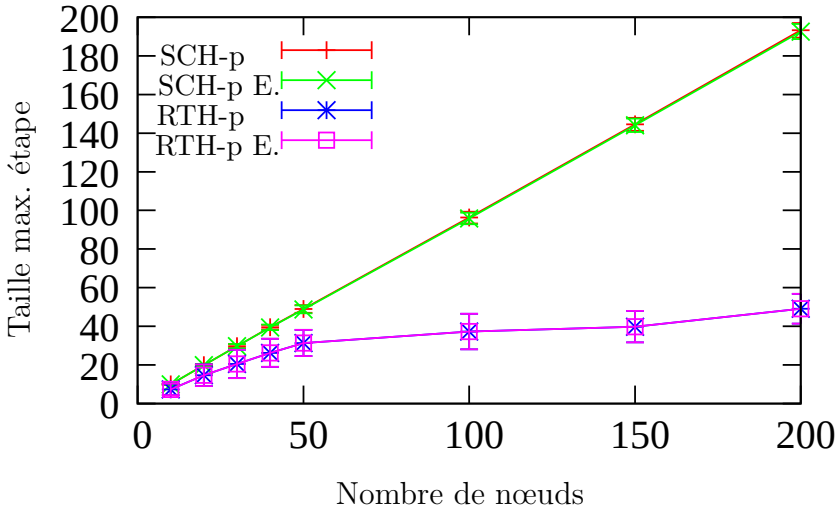

FIGURE 3.12 – Coût de la congestion avant et après équilibrage des étapes des heuristiques SCH-p et RTH-p en fonction du nombre de nœuds pour le cas des topologies aléatoires.

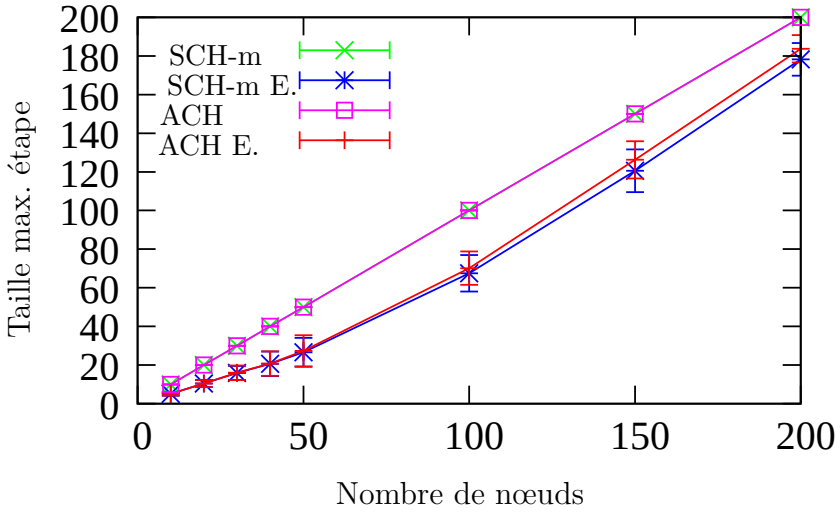

FIGURE 3.13 – Coût de la congestion avant et après équilibrage des étapes des heuristiques SCH-m et ACH en fonction du nombre de nœuds pour le cas des topologies aléatoires.

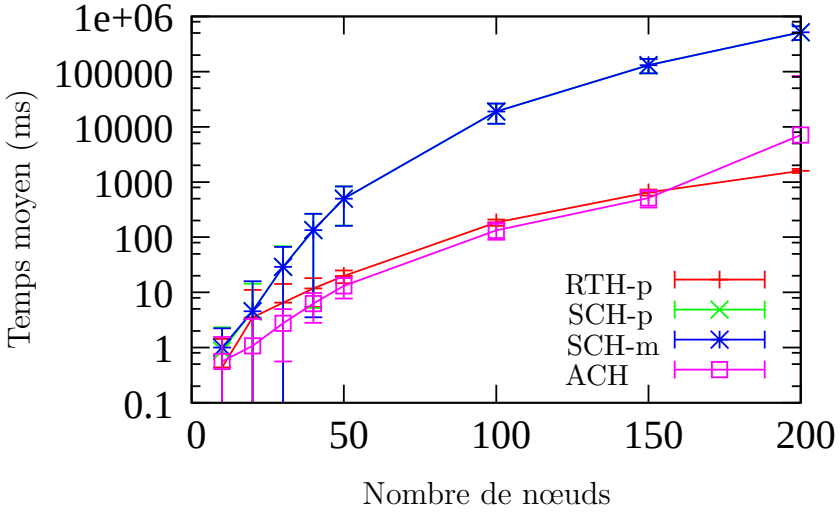

FIGURE 3.14 – Temps de calcul de la transition en fonction du nombre de nœuds pour le cas des topologies aléatoires.

est la suivante : ACH, à l'instar de SCH-p et SCH-m, est basée sur la décomposition en composantes fortement connexes de l'union des deux routages qui sont connus *a priori*. Dans le cas d'un réseau multi-destinations, SCH-m et ACH appliquent d'abord un traitement mono-destination pour chacune des destinations du réseau avant d'appliquer l'opération de fusion qui se fait en un temps constant. Le traitement mono-destination de SCH-m consiste à appliquer SCH-p, raison pour laquelle SCH-m et SCH-p ont des courbes de temps confondues sur la figure. Lorsqu'on s'intéresse au cas d'un réseau mono-destination, particulièrement au cas du traitement d'une composante fortement connexe, ACH traite des groupes de nœuds associés à des cycles tandis que SCH-p traite chaque nœud non encore migré et construit en fonction de celui un graphe sur lequel sera vérifié l'existence de boucles. Ainsi, le temps de traitement d'une composante fortement connexe par ACH est lié au nombre de cycles identifiés dans cette dernière, tandis que le temps de traitement de cette composante fortement connexe par SCH-p est lié au nombre de nœuds qu'elle contient. La figure 3.15 montre que pour la taille 200, on avoisine en moyenne quelques dizaines de milliers de boucles : pour pouvoir les traiter, ACH nécessite plus de temps. La figure 3.16 nous montre l'évolution de ce temps de calcul en fonction du nombre moyen de boucles générées.

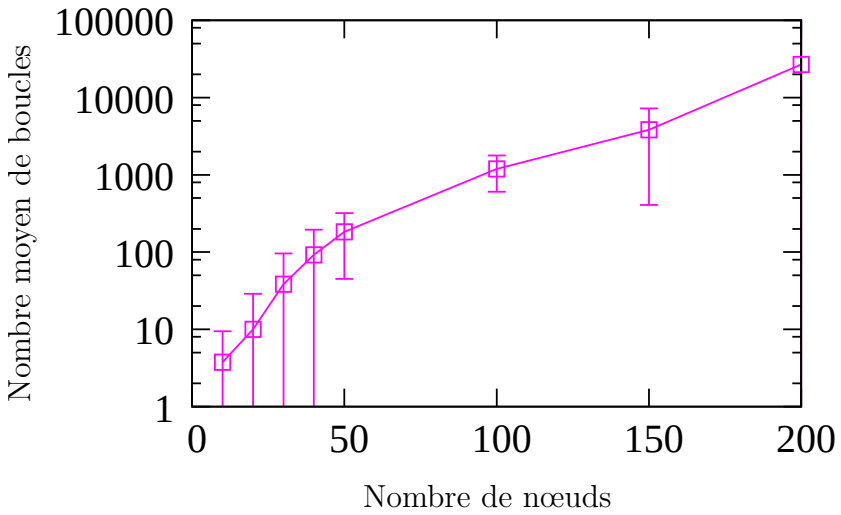

FIGURE 3.15 – Nombre moyen de boucles en fonction du nombre de nœuds pour le cas des topologies aléatoires.

3.3.4 Résultats sur des topologies *ITZ*

Le nombre de groupes de destinations compatibles : La figure 3.17 montre le nombre moyen de groupes de destinations compatibles pour chacune des heuristiques RTH-p, SCH-p, SCH-m et ACH dans le cas de topologies *ITZ*. La figure montre que ACH et SCH-m ont chacune un seul groupe de destinations compatibles. La raison reste le traitement collectif de toutes les destinations. Pour les heuristiques SCH-p et RTH-p, nous observons que le nombre moyen augmente avec la taille du réseau. En effet, SCH-p traite chaque destination individuellement, donc ce nombre est toujours égal à la taille du réseau. Le nombre de groupes de destinations de RTH-p se trouve dans l'intervalle $[1; d]$ avec d la taille du réseau. Nous observons une petite chute de ce nombre de la taille 145 à la taille 149 : cette petite chute correspond à la petite chute du nombre moyen de boucles générées entre ces deux tailles du réseau (cf figure 3.18). Nous remarquons que les courbes de RTH-p et SCH-p ne se confondent pas dans le cas des réseaux plus

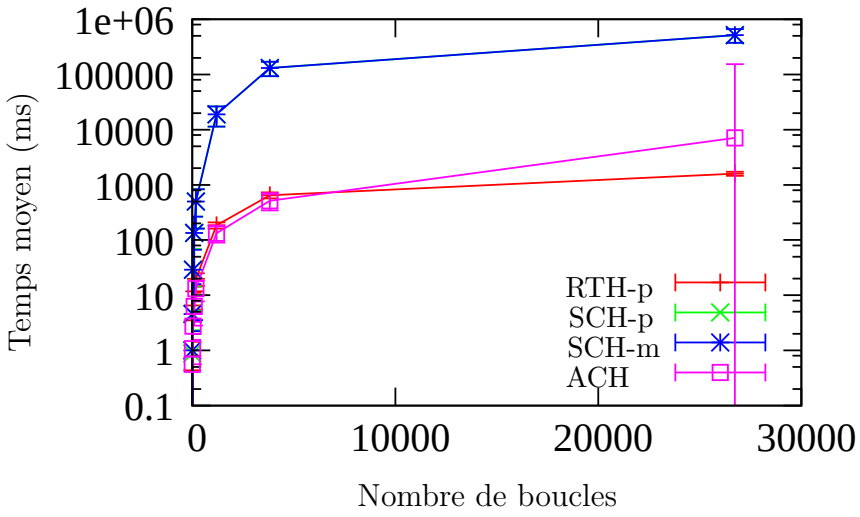

FIGURE 3.16 – Temps de calcul de la transition en fonction du nombre moyen de boucles pour le cas des topologies aléatoires.

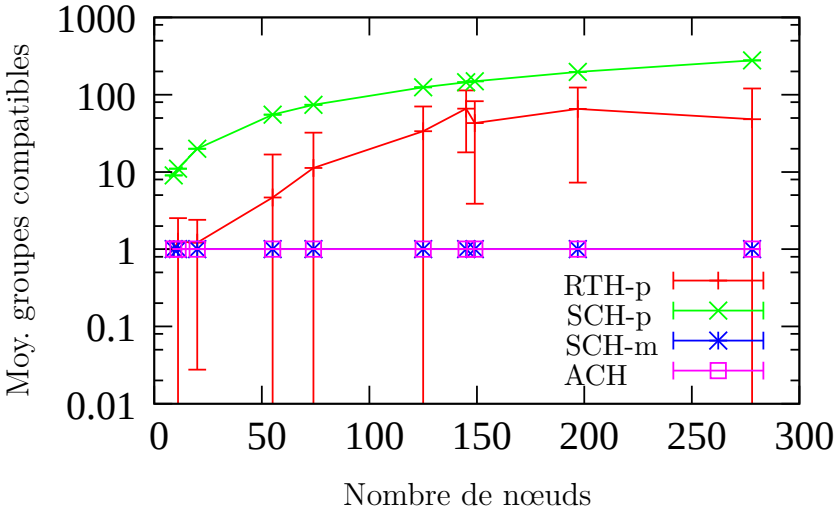

FIGURE 3.17 – Nombre de groupes de destinations compatibles en fonction du nombre de nœuds pour le cas des topologies *ITZ*.

grands comme c'était le cas des topologies aléatoires. Ceci s'explique par le fait que dans le cas des topologies aléatoires, le nombre moyen des boucles générées atteint le seuil de dizaines de milliers (figure 3.15), tandis que pour les topologies *ITZ*, le seuil atteint est celui de milliers de boucles générées (figure 3.18). En d'autres termes, les topologies réelles sont plus simples à faire migrer que les topologies aléatoires puisque le nombre de boucles est plus petit.

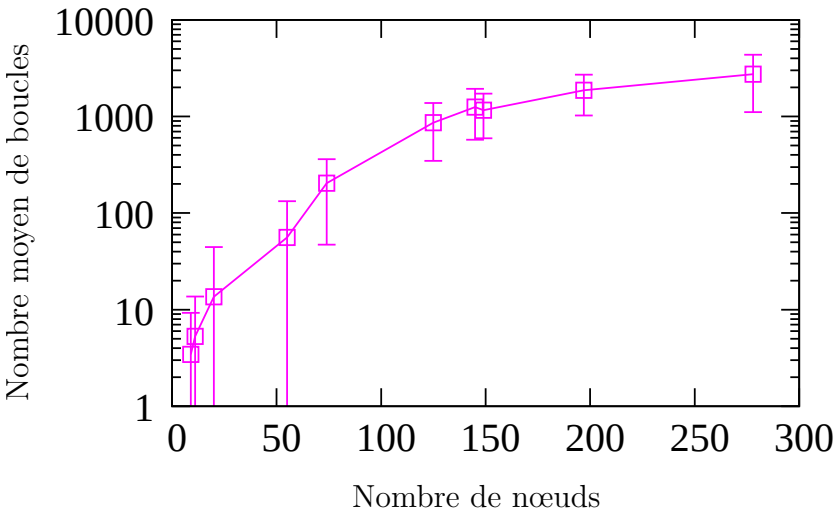

FIGURE 3.18 – Nombre moyen de boucles en fonction du nombre de nœuds pour le cas des topologies *ITZ*.

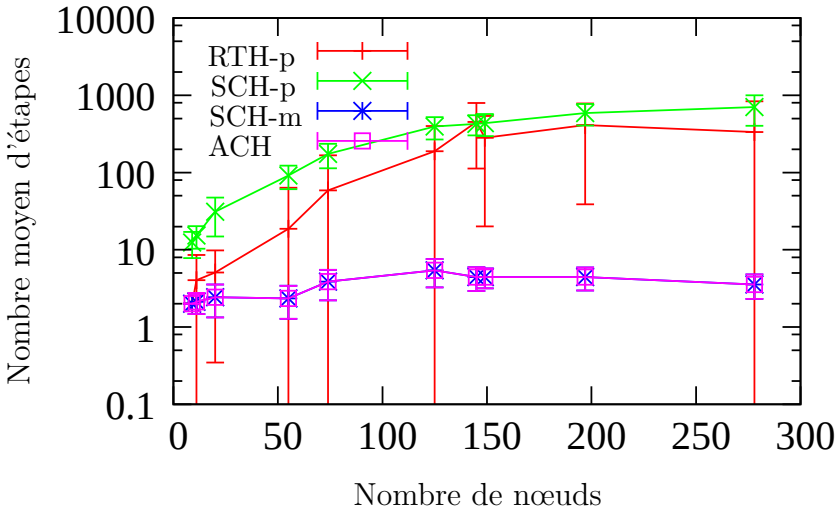

FIGURE 3.19 – Durée de la transition en nombre d'étapes, en fonction du nombre de nœuds pour le cas des topologies *ITZ*.

La durée de la transition : La figure 3.19 montre la durée de la transition calculée qui est le nombre d'étapes nécessaires pour effectuer la migration pour le cas des topologies *ITZ*. Nous faisons le même constat que dans les topologies aléatoires : le nombre moyen d'étapes pour ACH et SCH-m est un petit nombre, peu dépendant de la taille du réseau, tandis que ce nombre croît significativement avec la taille du réseau pour RTH-p et SCH-p. Cependant, on note ici une différence entre RTH-p et SCH-p : RTH-p produit moins d'étapes que SCH-p qui devient ainsi l'heuristique la

plus lente quelle que soit la taille du réseau. Ce cas se produit lorsque les métriques sont corrélées. En effet, les routages $\mathcal{R}_i^d$ et $\mathcal{R}_f^d$ pour chacune des destinations d sont choisis aléatoirement dans deux intervalles différents, mais l'analyse des tableaux 3.3 et 3.4 montre que le nombre de liens pour les topologies *ITZ* est inférieur à celui des topologies aléatoires pour des tailles similaires, offrant ainsi moins de choix possibles de routes en cas de modification de poids de topologies. Ainsi, ACH montre un gain allant jusqu'à 99% par rapport à RTH-p, et un gain allant de 83% à 99% par rapport à SCH-p.

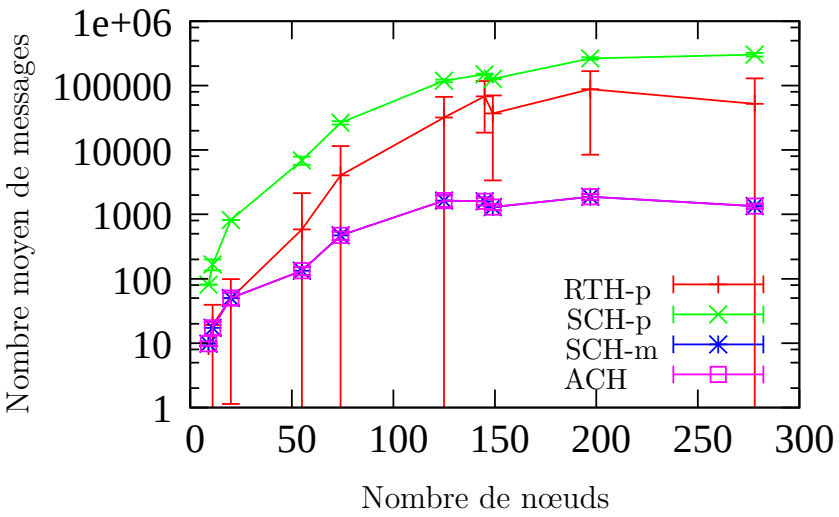

FIGURE 3.20 – Coût de la transition, en fonction du nombre de nœuds pour le cas des topologies *ITZ*.

Le coût de la transition : La figure 3.20 montre le coût, calculé en nombre de messages de contrôle, des différentes transitions produites par chacune des heuristiques dans le cas des topologies *ITZ*. Comme l'illustre la figure, chacune des courbes croît avec la taille du réseau. ACH et SCH-m sont les heuristiques produisant le moins de messages de contrôle. En effet, ACH et SCH-m montrent un gain allant jusqu'à 97% par rapport à RTH-p et un gain allant de 87% à 99% par rapport à SCH-p.

La congestion : La figure 3.21 montre la congestion générée par les transitions des différentes heuristiques RTH-p, SCH-p, SCH-m et ACH dans le cas de topologies *ITZ*. Cette congestion est calculée comme le nombre maximal de nœuds que peut contenir une étape. La figure nous illustre que pour ces topologies *ITZ*, la congestion augmente avec la taille du réseau pour toutes ces heuristiques. On note néanmoins que RTH-p, comme dans le cas de réseaux aléatoires, produit un peu moins de congestion que SCH-p, SCH-m et ACH. Comparativement à SCH-p, RTH-p montre un gain jusqu'à 16%, et comparativement à SCH-m et ACH, un gain jusqu'à 18%.

L'équilibrage : Les figures 3.22 et 3.23 montrent la congestion générée pour les topologies ITZ par les heuristiques RTH-p et SCH-p, et par les heuristiques SCH-m et ACH lorsque les transitions ont été équilibrées. La figure 3.22 montre que RTH-p et SCH-p en version équilibrée produisent sensiblement la même congestion que la version non équilibrée. Cependant, la figure 3.23 montre que l'équilibrage permet de diminuer la congestion des heuristiques ACH et SCH-m avec un gain qui va jusqu'à

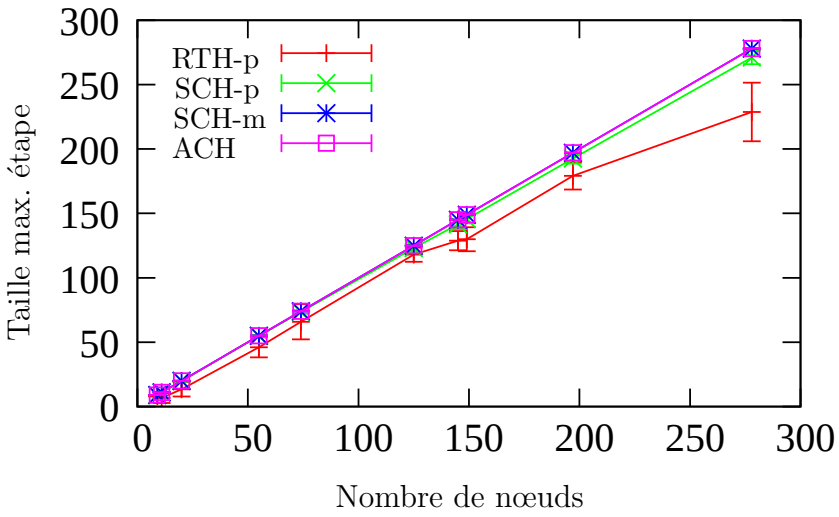

FIGURE 3.21 – Coût de la congestion, en fonction du nombre de nœuds pour le cas des topologies *ITZ*.

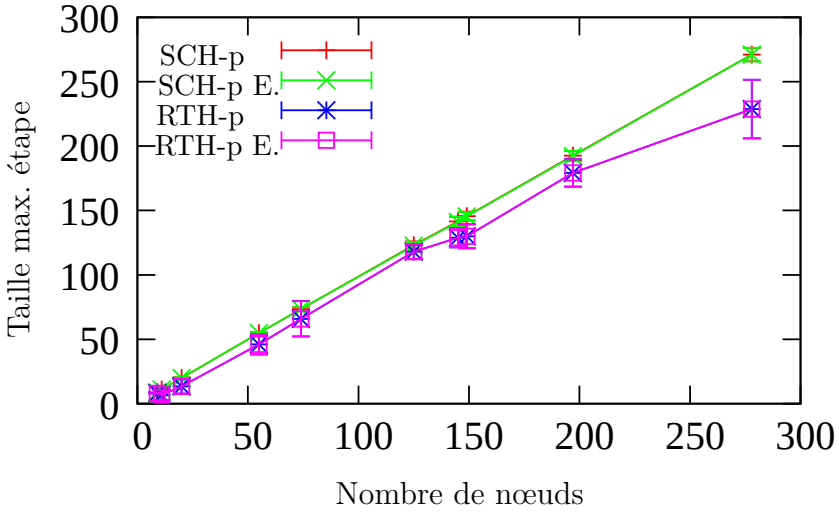

FIGURE 3.22 – Coût de la congestion avant et après équilibrage des étapes des heuristiques SCH-p et RTH-p, en fonction du nombre de nœuds pour le cas des topologies aléatoires.

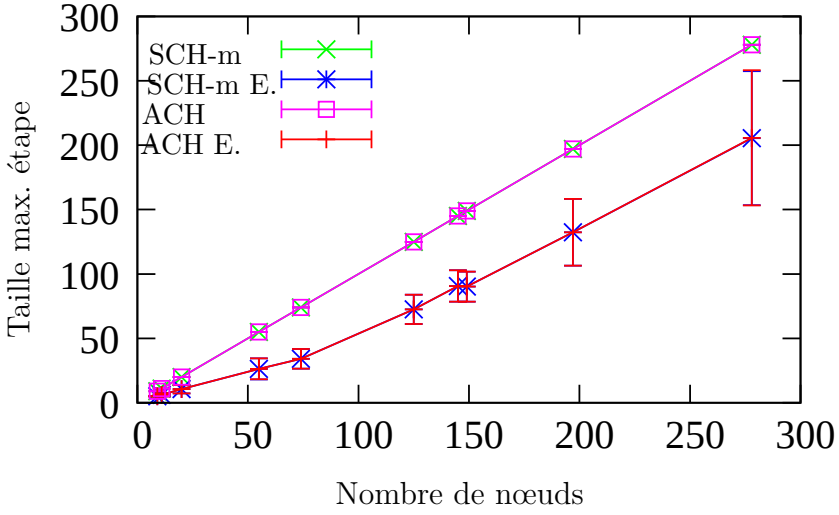

FIGURE 3.23 – Coût de la congestion avant et après équilibrage des étapes des heuristiques SCH-m et ACH, en fonction du nombre de nœuds pour le cas des topologies aléatoires.

26% de congestion en moins pour les deux. En equilibré, ACH et SCH-m sont les meilleures heuristiques pour toutes les tailles de réseau avec un gain allant jusqu'à 10% par rapport à RTH-p, et un gain allant jusqu'à 24% par rapport à SCH-p.

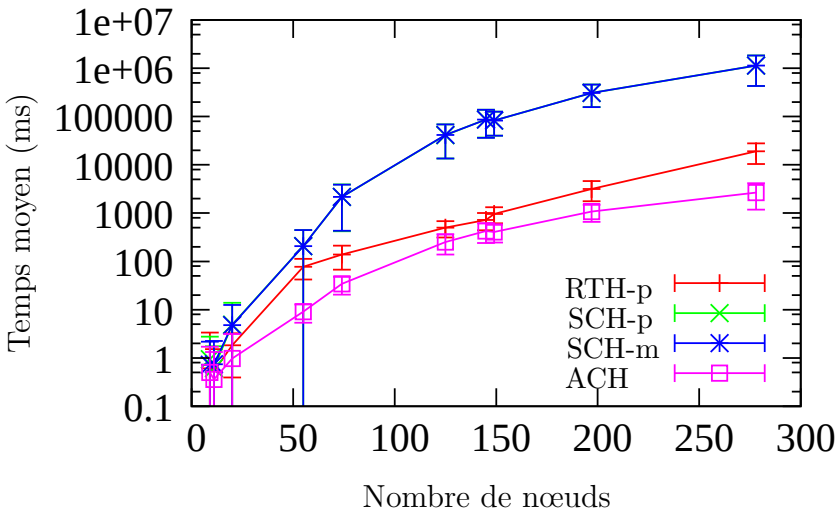

FIGURE 3.24 – Temps de calcul de la transition en fonction du nombre de nœuds pour le cas des topologies *ITZ*.

Le temps de calcul : La figure 3.24 montre le temps de calcul moyen des heuristiques RTH-p, SCH-p, SCH-m et ACH pour les topologies *ITZ*. Comme dans le cas des réseaux aléatoires, nous observons que les temps de calcul des différentes heuristiques évoluent avec le temps dû à l'augmentation du nombre de nœuds à traiter. Nous observons également que ACH est l'heuristique qui s'exécute le plus rapidement comparativement aux trois autres. En effet, ACH montre un gain de 16% à 86% comparativement à RTH-p, et un gain de 47% à 99% comparativement à SCH-p et SCH-m. Lorsque l'on observe la figure 3.25, on remarque que le nombre moyen de boucles générées n'avoisine pas le seuil de dizaines de milliers. Ce qui est similaire au résultat dans le cas des topologies aléatoires, à savoir qu'en dessous du seuil de 10.000 boucles, ACH s'exécute plus rapidement que les trois autres heuristiques RTH-p, SCH-p, et SCH-m.

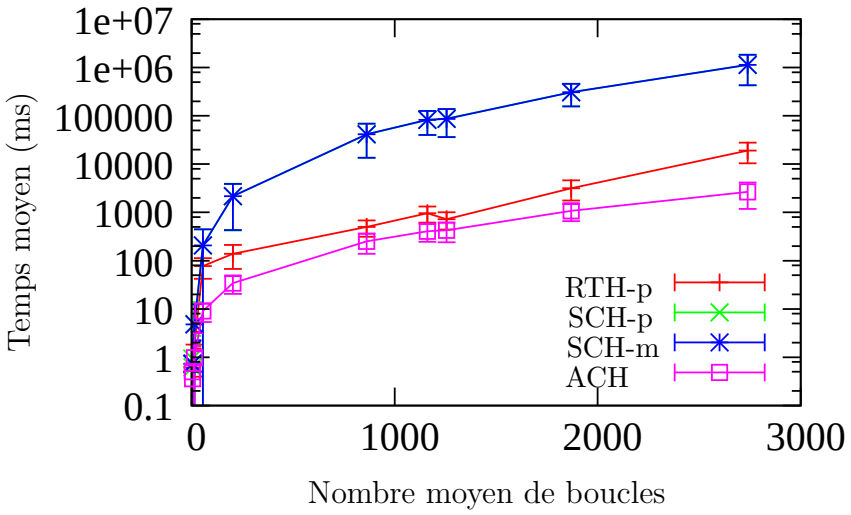

FIGURE 3.25 – Temps de calcul de la transition en fonction de la taille des boucles pour le cas des topologies *ITZ*.

3.3.5 Résultats sur des topologies *Rocketfuel*

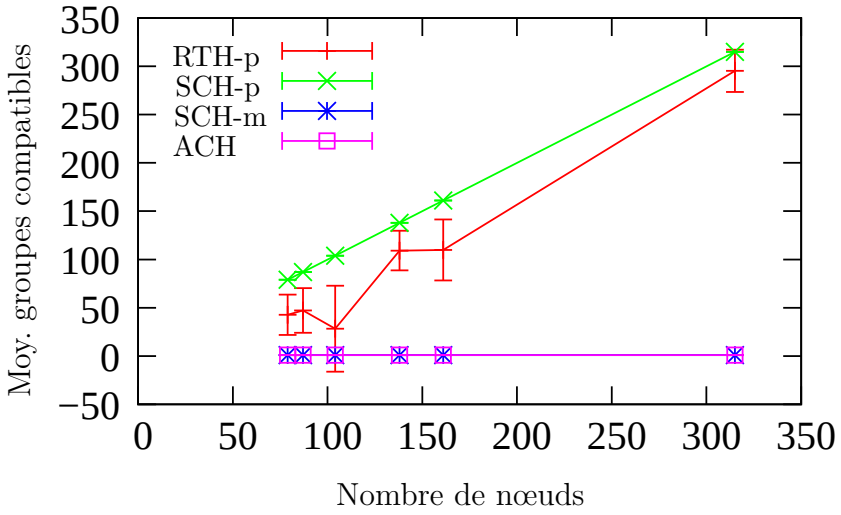

FIGURE 3.26 – Nombre de groupes de destinations compatibles en fonction du nombre de nœuds pour les topologies *Rocketfuel*.

Le nombre de groupes de destinations compatibles : La figure 3.26 montre le nombre moyen de groupes de destinations compatibles pour chacune des heuristiques RTH-p, SCH-p, SCH-m et ACH dans le cas de topologies *Rocketfuel*. Comme pour les deux cas présentés précédemment, ce nombre est toujours égal à 1 pour ACH et SCH-m dû au traitement collectif appliqué. Il est égal à d, nombre de destinations, pour SCH-p qui les traite de manière individuelle. Ce nombre est compris dans l'intervalle $[1; d]$ pour RTH-p qui traite les destinations non problématiques comme un seul groupe et les autres individuellement. On observe une petite chute de ce nombre entre les jeux de données *Ebone* et *Telstra* de tailles respectives 87 et 104 nœuds. Cette chute est liée à la baisse du nombre de boucles pour le jeu de données *Telstra* (figure 3.27).

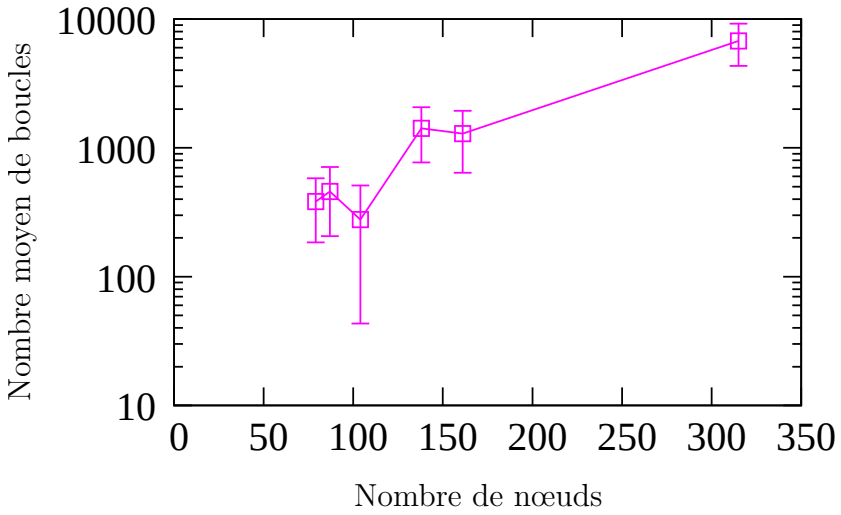

FIGURE 3.27 – Nombre moyen de boucles en fonction du nombre de nœuds pour le cas des topologies *Rocketfuel*.

La durée de la transition : La figure 3.28 montre la durée de la transition calculée en nombre d'étapes pour le cas des topologies *Rocketfuel*. Nous observons que ACH et

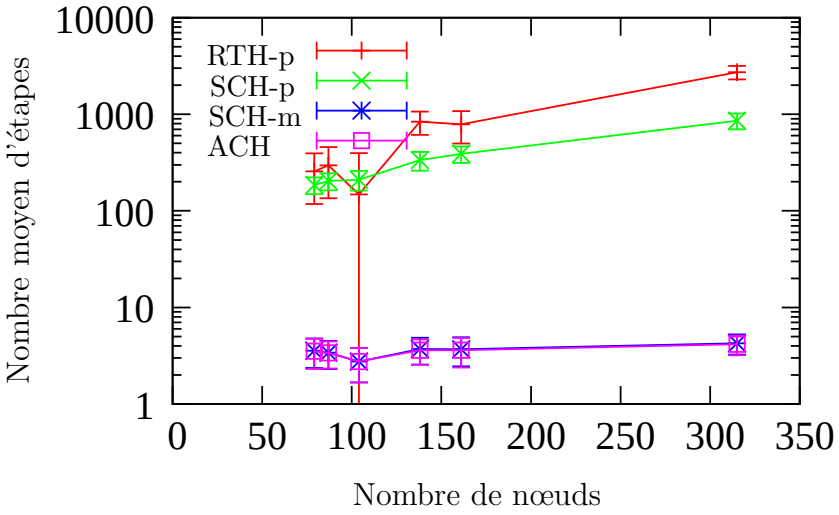

FIGURE 3.28 – Durée de la transition en nombre d'étapes, en fonction du nombre de nœuds pour les topologies *Rocketfuel*.

SCH-m produisent de petits nombres d'étapes, peu dépendants de la taille du réseau et du nombre de destinations, comme dans le cas des topologies aléatoires et des topologies *ITZ*. RTH-p et SCH-p produisent des durées de transition croissantes avec la taille du réseau. RTH-p est l'heuristique qui produit le plus d'étapes sauf dans le cas du jeu de données *Telstra* (104 nœuds) où SCH-p est celle qui en produit le plus dû à la diminution du nombre de destinations problématiques. ACH et SCH-m montrent un gain de 99% par rapport à SCH-p, et un gain de 98% par rapport à RTH-p.

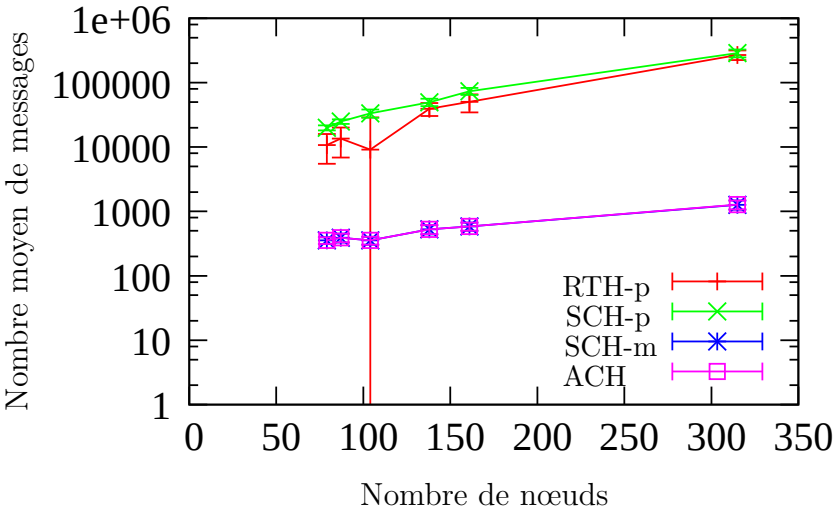

FIGURE 3.29 – Coût de la transition, en fonction du nombre de nœuds pour les topologies *Rocketfuel*.

Le coût de la transition : La figure 3.29 montre le coût des transitions générées par ACH, SCH-m, SCH-p et RTH-p, calculé en nombre de messages de contrôle nécessaires, dans le cas des topologies *Rocketfuel*. Nous y observons que le coût moyen de la transition croît avec la taille du réseau pour chaque heuristique. Dû à leur faible nombre d'étapes, ACH et SCH-m ne requièrent qu'un faible nombre de messages de contrôle pour configurer tous les nœuds. De même, SCH-p produit plus de messages de contrôle que les trois autres heuristiques sauf pour le jeu de données *Sprint* de 315

nœuds (où RTH-p produit à peu près le même nombre de messages de contrôle). ACH et SCH-m montrent un gain de 98% par rapport à RTH-p, et un gain de 99% par rapport à SCH-p.

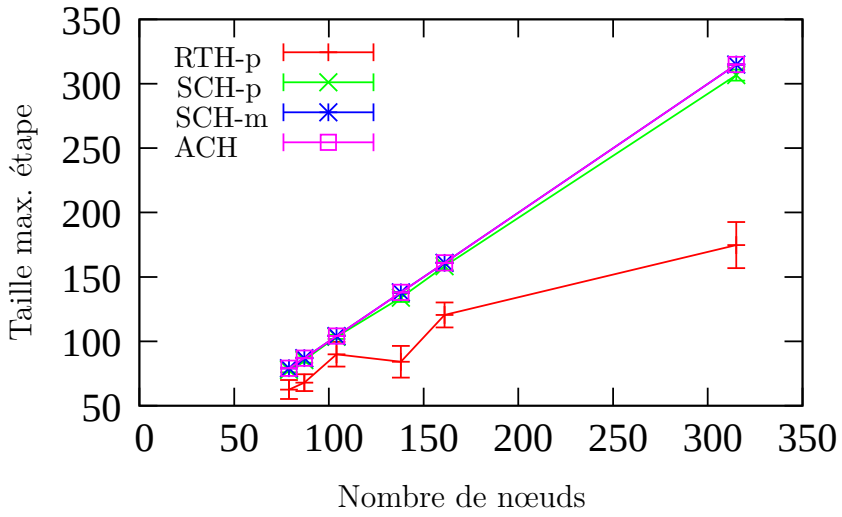

FIGURE 3.30 – Coût de la congestion, en fonction du nombre de nœuds pour les topologies *Rocketfuel*.

La congestion : La figure 3.30 montre la congestion produite par ACH, SCH-m, SCH-p et RTH-p dans le cas des topologies *Rocketfuel* calculée comme la taille de la plus grande étape. La congestion croît avec la taille du réseau pour chaque heuristique. Cependant comme dans les deux cas précédents, RTH-p produit beaucoup moins de congestion que les trois autres heuristiques. En effet, en cherchant à réduire le nombre d'étapes, les trois autres heuristiques concentrent autant que possible les nœuds dans les étapes. RTH-p montre un gain de 19% à 43% par rapport à SCH-p, et un gain de 25% à 45% par rapport à ACH et SCH-m.

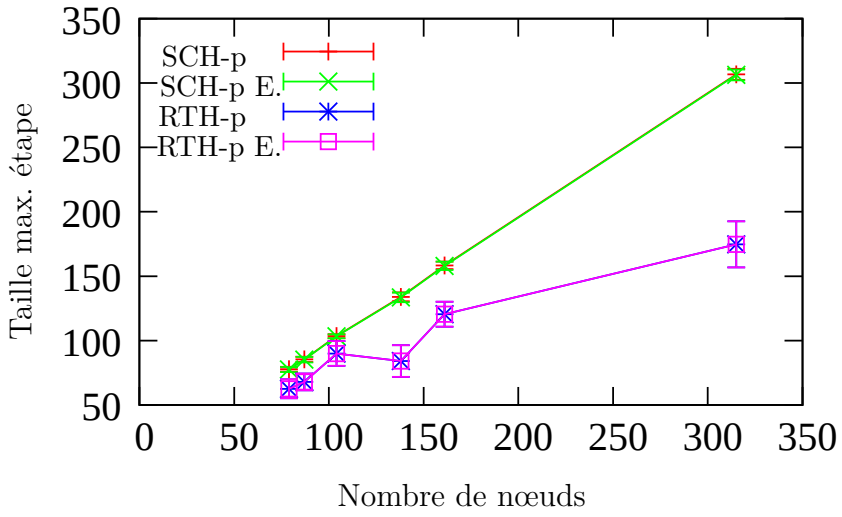

FIGURE 3.31 – Coût de la congestion avant et après équilibrage des étapes des heuristiques SCH-p et RTH-p, en fonction du nombre de nœuds pour le cas des topologies aléatoires.

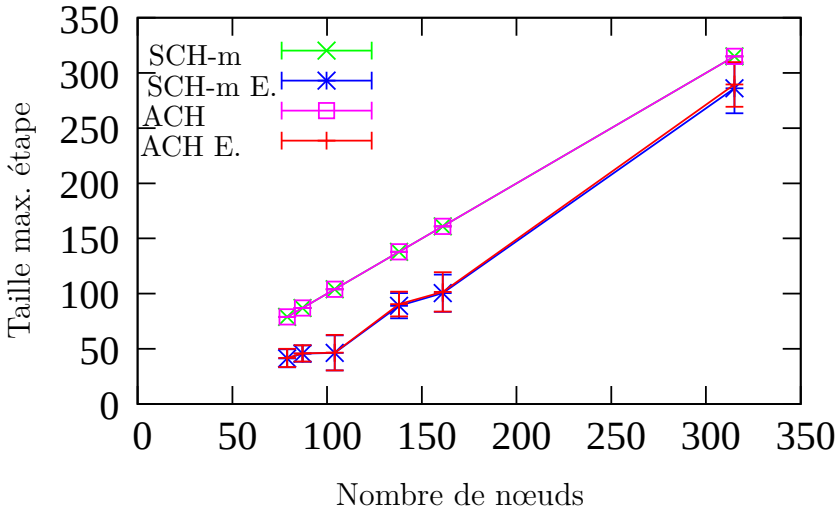

FIGURE 3.32 – Coût de la congestion avant et après équilibrage des étapes des heuristiques SCH-m et ACH, en fonction du nombre de nœuds pour le cas des topologies aléatoires.

L'équilibrage : Les figures 3.31 et 3.32 montrent la congestion générée respectivement par les heuristiques RTH-p et SCH-p, et par les heuristiques SCH-m et ACH lorsque les transitions ont été équilibrées dans le cas des topologies *Rocketfuel*.

La figure 3.31 montre que RTH-p et SCH-p en version équilibrée n'ont pas de différence en termes de congestion possible en cas de non équilibrage.

La figure 3.32 montre que l'équilibrage permet de diminuer la congestion des heuristiques ACH et SCH-m. En effet, nous notons un gain qui va jusqu'à 37% et 38% de congestion en moins respectivement pour ACH et SCH-m. Comme dans le cas de topologies ITZ, les heuristiques ACH et SCH-m en version equilibrée produisent moins de congestion que RTH-p (respectivement SCH-p) avec un gain qui va jusqu'à 16% (respectivement 36%).

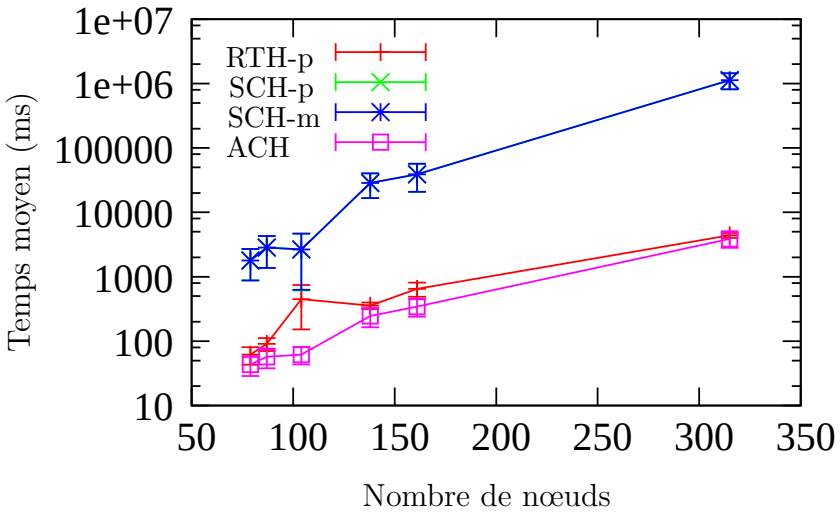

FIGURE 3.33 – Temps de calcul de la transition en fonction du nombre de nœuds pour les topologies *Rocketfuel*.

Le temps de calcul : La figure 3.33 présente le temps de calcul des transitions par les heuristiques ACH, SCH-m, SCH-p, et RTH-p pour le cas des topologies *Rocketfuel* en fonction de la taille du réseau. Nous observons que le temps de calcul associé à

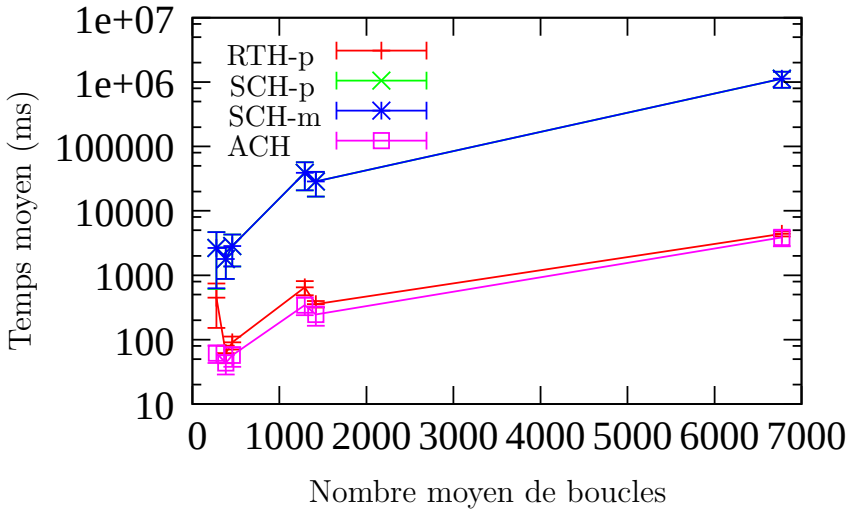

FIGURE 3.34 – Temps de calcul de la transition en fonction du nombre de boucles pour les topologies *Rocketfuel*.

chacune croît avec la taille du réseau. Nous notons également que ACH et RTH-p sont les plus rapides, ACH étant la meilleure de toutes. En effet, à l'exception du dernier jeu de données *Sprint* (315 nœuds) où ACH et RTH-p se confondent, ACH surpasse RTH-p avec un gain allant de 30% à 47%, et ACH surpasse SCH-m et SCH-p avec un gain de 96% à 99%. Les mêmes observations sont faites sur la figure 3.34 qui montre le temps d'exécution de chaque heuristique en fonction du nombre de boucles générées. La figure 3.34 confirme que SCH-m et SCH-p prennent plus de temps pour s'exécuter que les autres, et confirme également qu'en deçà du seuil de dizaines de milliers de boucles, ACH est l'heuristique la plus rapide.

Conclusion

Dans ce chapitre, nous avons présenté deux contributions d'approches d'ordonnancement des nœuds dans un contexte statique centralisé. La première contribution, SCH-m [45] est basée sur l'heuristique SCH-p [13] qui construit les étapes d'une transition en partitionnant d'une part l'union des deux routages en composantes fortement connexes, puis en fusionnant d'autre part dans une même étape tous les nœuds capables d'effectuer la migration ensemble sans causer de boucles de routage à la même étape. SCH-m propose la fusion étape par étape des différentes transitions produites par destination dans le cas d'un réseau multi-destinations, ceci dans le but d'accélérer la migration. La deuxième contribution est celle de l'heuristique ACH [46], qui propose de maximiser les nœuds qui effectuent la migration ensemble à une même itération à travers l'introduction de la notion de *casser un cycle*. Les résultats des simulations sur trois jeux de données ont permis de tirer les conclusions suivantes.

— En termes du nombre de groupes de destinations compatibles : SCH-m et ACH produisent un seul groupe parce que fusionnant les destinations, tandis que SCH-p produit un nombre de groupes égal au nombre de destinations, et RTH-p un nombre égal à $t + 1$ avec t le nombre de destinations problématiques calculées et 1 le groupe de destinations pouvant être traitées ensemble. Le nombre de groupes de destinations compatibles a un impact sur le nombre d'étapes, et par conséquent sur la transition finale car plus on a de groupes, plus est élevé le nombre d'étapes.

— En termes de durée de la transition : SCH-m et ACH produisent un petit nombre d'étapes, peu dépendant de la taille du réseau et du nombre de destinations. Le gain par rapport à SCH-p et RTH-p avoisine généralement 99%. Vient ensuite SCH-p qui est meilleure que RTH-p, sauf pour les topologies *ITZ* [47] où RTH-p a produit en moyenne moins d'étapes que SCH-p. Ceci se justifie par la très faible densité de ces topologies entraînant une corrélation dans les routages.
— En termes de coût de la transition : ACH et SCH-m surpassent toujours SCH-p et RTH-p avec un gain allant jusqu'à 99%. SCH-p est l'heuristique qui produit le plus de messages car elle traite les destinations les unes après les autres. RTH-p produit un coût qui dépend du nombre de groupes de destinations problématiques calculées. Plus ce nombre est élevé, plus RTH-p est gourmand en coût.
— En termes de congestion générée : SCH-p, SCH-m, et ACH sont moins bons que RTH-p car produisent plus de congestion. En effet, RTH-p montre un gain de 30% à 75% pour les topologies aléatoires, de 16% à 18% pour les topologies réelles, et de 19% à 45% pour les topologies *Rocketfuel*. Nous noterons que cette congestion élévée des heuristiques ACH et SCH-m ne change pas le fait qu'elles sont plus rapides que RTH-p et SCH-p. En effet, qui dit congestion dit délai. Or le délai réel est la somme du temps de calcul, et du temps nécessaire à la configuration d'une étape fois le nombre d'étapes produites. Étant donné que ACH et SCH-m produisent un faible nombre d'étapes peu dépendant de la taille du réseau, leur délai sera plus petit que celui de RTH-p et SCH-p. Nous avons néanmoins réfléchi à une métrique pour équilibrer les nœuds dans les étapes. Cette métrique a permis d'avoir un gain considérable en diminuant la congestion pour les heuristiques ACH et SCH-m et un léger gain pour l'heuristique SCH-p.
— En termes de temps de calcul : ACH est l'heuristique la plus rapide de toutes lorsque le nombre moyen de boucles générées est en deçà de 10.000 boucles ; au delà RTH-p devient la plus rapide. SCH-m et SCH-p requièrent toutes les deux toujours plus de temps que ACH et RTH-p.

Chapitre 4

Heuristiques dans un contexte statique distribué

Sommaire

Introduction

Les contributions de ce chapitre portent sur la migration d'un protocole de routage à un autre de manière distribuée dans un réseau statique. Les propositions faites dans le chapitre précédent considéraient qu'il existe une entité centrale dans le réseau qui connaît la topologie entière du réseau, connaît *a priori* les routages initial $\mathcal{R}_i$ et final $\mathcal{R}_f$ vers toutes les destinations considérées, et enfin calcule et diffuse la séquence d'étapes appelée transition. Dans ce chapitre, nous considérons que l'environnement est distribué (aucun nœud n'a une vue totale de la topologie du réseau) et nous nous intéressons particulièrement au cas du retrait planifié d'un routeur ou d'une panne dans un réseau. Ce changement peut conduire à des boucles de routage et dégrader considérablement les performances du réseau. Nous présentons deux heuristiques distribuées : *Routing Tree Heuristic-Distributed version* (RTH-d) et *Distributed Loop-Free heuristic* (DLF).

4.1 RTH-d : *Routing Tree Heuristic-distributed version*

RTH-d [45] est notre proposition de modification de l'heuristique RTH-p [13] qui lui permet d'être distribuée. Nous présentons d'abord RTH-d pour le cas d'un réseau mono-destination, puis pour le cas d'un réseau multi-destinations.

4.1.1 RTH-d pour un réseau mono-destination

RTH-d considère uniquement le protocole de routage final $\mathcal{R}_f$ vers la destination, et construit les étapes en faisant migrer dans une même étape les nœuds situés à la même profondeur. Les nœuds migrent par échange de messages. RTH-d traite en premier le nœud destination. Ce nœud migre, puis informe ses voisins à un saut par le biais d'un message. Ces derniers migrent à leur tour (les nœuds à un saut de la destination sont exceptionnellement ajoutés à la même étape que celle contenant le nœud destination). Ils informent eux aussi leurs voisins à un saut et une nouvelle étape est créée. Ce processus est répété jusqu'à ce qu'il ait atteint les feuilles, et ainsi tous les nœuds auront migré.

L'algorithme 5 présente les instructions exécutées par chaque nœud dans le réseau.

Algorithme 5 : RTH-d, version distribuée de RTH-p

Données : n est le nœud courant, $\mathcal{R}_f(n)$ est connu par le nœud n

1 **si** n *n'est pas la destination* **alors**
2 | attendre la notification de $\mathcal{R}_f(n)$
3 **fin**
4 n migre à $\mathcal{R}_f$
5 n notifie tous ses voisins

Considérons le graphe de la figure 4.1 qui montre un exemple de réseau avec le nœud 5 qui tombe en panne. Si nous considérons la destination 10, la figure 4.2 nous montre qu'une migration non contrôlée peut générer deux boucles à savoir : $(1, 7, 1)$ et $(1, 8, 1)$. Pour éviter ces boucles possibles, la figure 4.3 présente les détails du calcul de la transition finale par RTH-d pour cet exemple. Tout d'abord, comme précisé dans la description, RTH-d ne considère pas l'union des routages, mais uniquement le routage

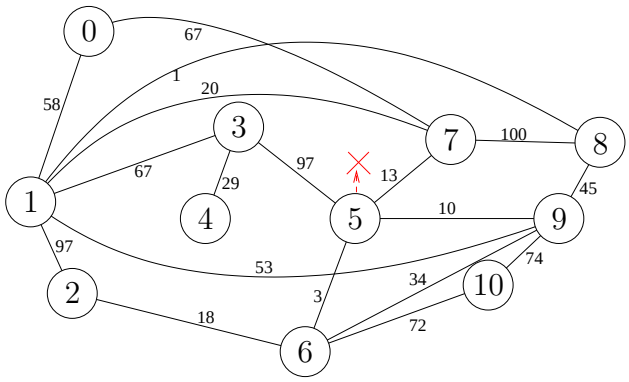

FIGURE 4.1 – Exemple du réseau américain *Sprint* avec la panne du nœud 5.

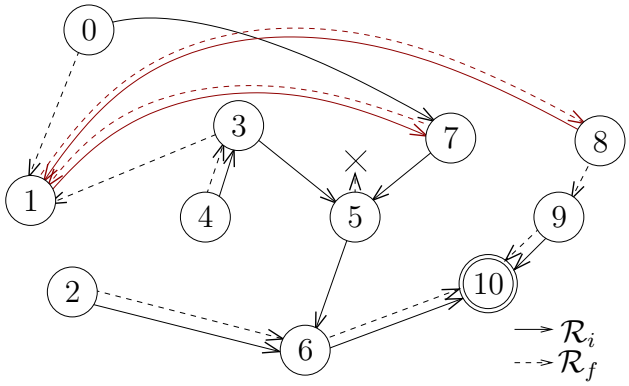

FIGURE 4.2 – Union des deux routages vers la destination 10 avant la panne du nœud 5 ($\mathcal{R}_i$) et après la panne du nœud 5 ($\mathcal{R}_f$), pour l'exemple de la figure 4.1.

final vers la destination. Ainsi, partant du routage initial illustré par la figure 4.3(a), RTH-d crée une étape avec le nœud destination 10. Le nœud 10 ayant migré envoie un message à ses voisins 6 et 9 qui peuvent à leur tour migrer. Ils sont ajoutés à l'étape en cours, ainsi que le nœud 5 qui tombe en panne (figure 4.3(b)). Un message est ensuite envoyé à chacun des nœuds de la profondeur suivante (pointés par des flèches rouges) à savoir 2 et 8 qui sont migrés dans une nouvelle étape (figure 4.3(c)). Vient ensuite le tour du nœud 1 qui est également migré dans une nouvelle étape (figure 4.3(d)). Ensuite, les nœuds de la profondeur suivante, à savoir : 0, 3 et 7, sont migrés ensemble (figure 4.3(e)). Enfin RTH-d traite le nœud 4 (figure 4.3(f)). La transition finale produite par RTH-d pour cet exemple est : $Tr = (\{5,6,9,10\}_{10}, \{2,8\}_{10}, \{1\}_{10}, \{0,3,7\}_{10}, \{4\}_{10})$.

4.1.2 RTH-d pour un réseau multi-destinations

RTH-d traite le cas multi-destinations de façon parallèle, tel que SCH-m et ACH. Pour construire sa transition finale, on peut considérer que RTH-d applique une opération de fusion des différentes transitions produites pour chaque destination individuellement. Nous rappelons que si l'on suppose que $etape(i,j)$ est l'ensemble des nœuds de la $i^{ème}$ étape de la transition produite par RTH-d pour la destination j, la $i^{ème}$ étape de la transition finale est obtenue en fusionnant la $i^{ème}$ étape de chaque transition par destination : $etape(i) = \cup_{j=1}^{d} etape(i,j)$ où d est le nombre de destinations.

Considérons les destinations 2 et 10 de notre exemple de réseau (figure 4.1) dont les routages initial et final sont présentés sur la figure 4.4. Les différentes transitions

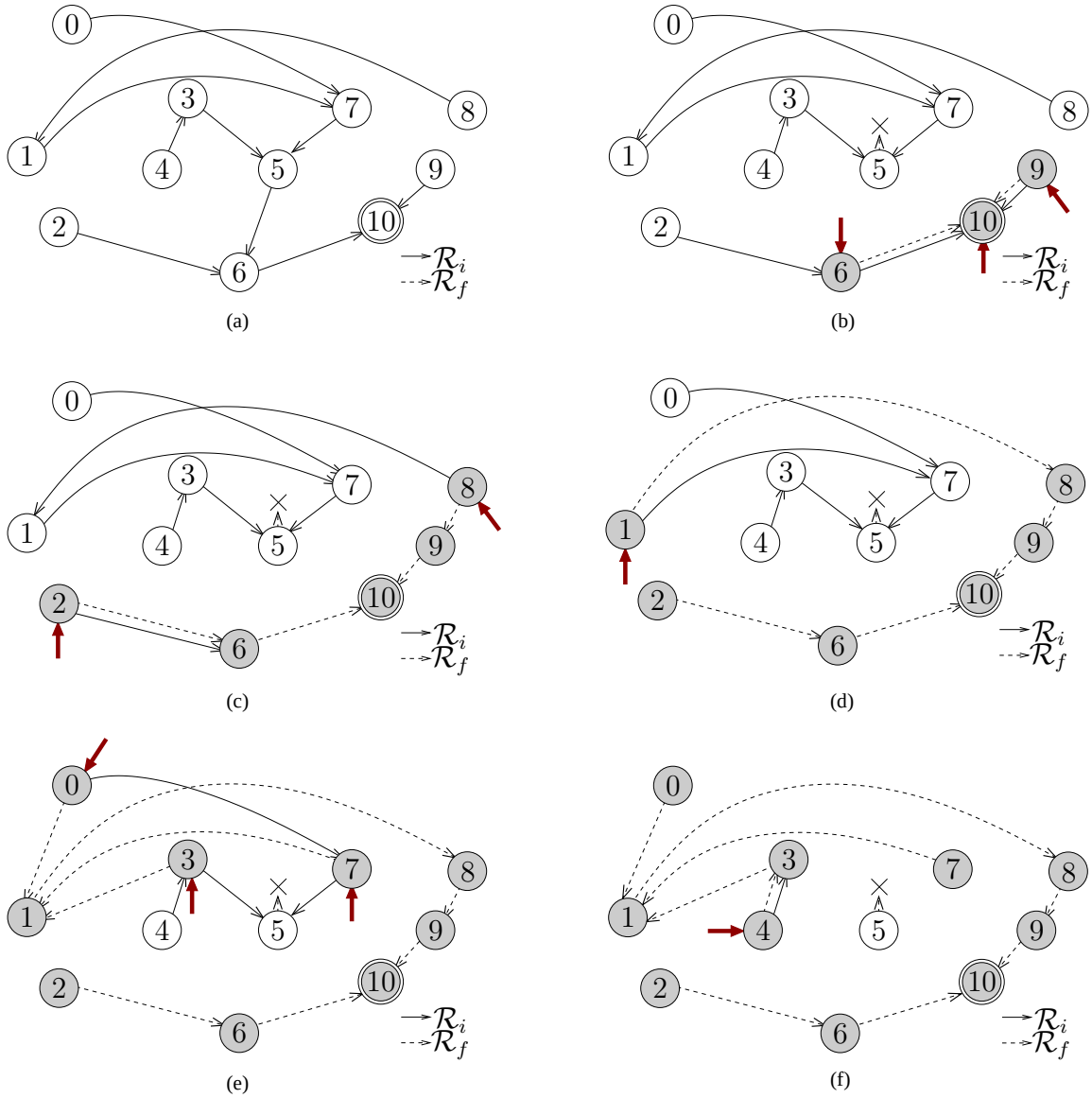

FIGURE 4.3 – Calcul de la transition sur l'exemple de la figure 4.1 par RTH-d. (a) Routage initial vers la destination 10 lorsque le nœud 5 tombe en panne, (b) migration des nœuds $\{5, 6, 9, 10\}$, (c) migration des nœuds $\{2, 8\}$, (d) migration du nœud $\{1\}$, (e) migration des nœuds $\{0, 3, 7\}$, (f) migration du nœud $\{4\}$.

que produit RTH-d sont les suivantes :

— destination 2 : $Tr = (\{1,2,5,6\}_2, \{0,3,7,9,10\}_2, \{4,8\}_2)$.

— destination 10 : $Tr = (\{5,6,9,10\}_{10}, \{2,8\}_{10}, \{1\}_{10}, \{0,3,7\}_{10}, \{4\}_{10})$.

$$\begin{cases} etape(1) = \{1,2,5,6\}_2 \cup \{5,6,9,10\}_{10} = \{5,6\}_{2,10} \cup \{1,2\}_2 \cup \{9,10\}_{10} \\ etape(2) = \{0,3,7,9,10\}_2 \cup \{2,8\}_{10} \\ etape(3) = \{4,8\}_2 \cup \{1\}_{10} \\ etape(4) = \{0,3,7\}_{10} \\ etape(5) = \{4\}_{10} \end{cases}$$

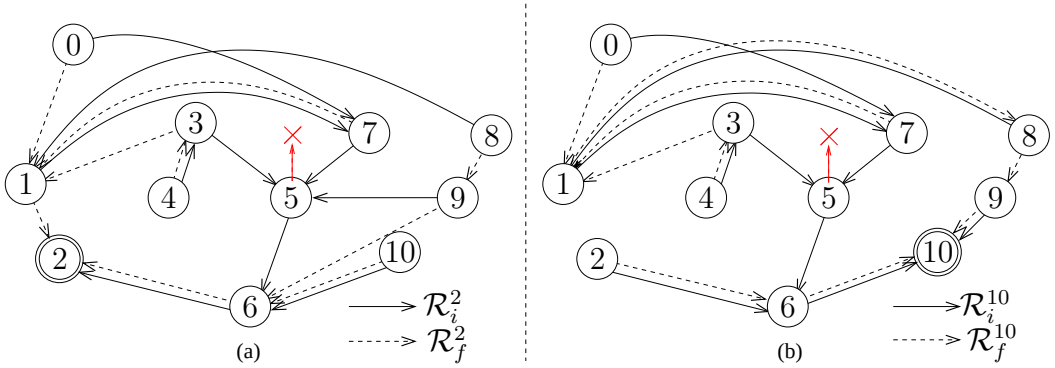

FIGURE 4.4 – Réseau *Sprint* de la figure 4.1 : (a) union des deux routages $\mathcal{R}_i^2$ et $\mathcal{R}_f^2$ vers la destination 2, (b) union des deux routages $\mathcal{R}_i^{10}$ et $\mathcal{R}_f^{10}$ vers la destination 10.

La transition finale produite par RTH-d est : $Tr = (\{5,6\}_{2,10} \cup \{1,2\}_2 \cup \{9,10\}_{10}, \{0,3,7,9,10\}_2 \cup \{2,8\}_{10}, \{4,8\}_2 \cup \{1\}_{10}, \{0,3,7\}_{10}, \{4\}_{10})$ qui compte 5 étapes.

4.1.3 Complexité

Pour un réseau de n nœuds et d destinations, la complexité en temps de RTH-d est de l'ordre $\mathcal{O}(n.d)$.

4.2 DLF : *Distributed Loop-Free heuristic*

DLF [45] est une heuristique distribuée que nous avons proposée pour l'évitement des boucles de routage dans le cas spécifique du retrait d'un nœud, à l'instar des algorithmes de la littérature GBA et AGBA [43]. Nous faisons dans un premier temps l'hypothèse que le prochain saut selon les routages $\mathcal{R}_i$ et $\mathcal{R}_f$ est connu d'avance par chaque nœud, et présentons DLF dans le cas d'un réseau mono-destination comme dans celui d'un réseau multi-destination. Ensuite, nous discutons dans un deuxième temps du cas plus réaliste où le routage final $\mathcal{R}_f$ n'est connu que par les nœuds qui ont été préalablement informés du changement topologique et présentons une discussion dans ce contexte.

4.2.1 DLF pour un réseau mono-destination

DLF exploite une propriété qui stipule que lorsqu'un nœud est retiré, les boucles de routage générées entre un protocole de routage sur la topologie initiale et le même protocole de routage sur la topologie finale sont toutes de taille 2 [43]. La justification

est donnée dans la partie 4.2.3. Cela implique que les boucles de routage possibles sont de la forme $(n, \mathcal{R}_i(n), \mathcal{R}_f(\mathcal{R}_i(n)), n)$ ou $(n, \mathcal{R}_f(n), \mathcal{R}_i(\mathcal{R}_f(n)), n)$. Identifier toutes les boucles commençant par un nœud n devient alors simple. La résolution d'une boucle dans ce contexte nécessite que parmi les deux nœuds impliqués dans la boucle, celui qui est le plus proche de la destination selon $\mathcal{R}_f$ effectue la migration avant l'autre. Formellement, si on suppose que (x, y, x) est la boucle à traiter et $\mathcal{R}_f(x) = y$, alors le nœud y est plus proche de la destination suivant $\mathcal{R}_f$ que le nœud x. Par conséquent, y migre à son prochain saut suivant $\mathcal{R}_f$ avant que x ne migre. Dans le cas contraire, c'est-à-dire $\mathcal{R}_f(x) \neq y$, x migre avant y.

Algorithme 6 : Distributed loop-free heuristic (DLF).

Données : n est le nœud courant, $\mathcal{R}_i$ et $\mathcal{R}_f$ sont connus *a priori* par tous les nœuds.

1 **si** $\mathcal{R}_i(\mathcal{R}_f(n)) = n$ **alors**
2 | n attend $\mathcal{R}_f(n)$
3 **sinon**
4 | n migre à $\mathcal{R}_f$
5 | **si** $\mathcal{R}_f(\mathcal{R}_i(n)) = n$ **alors**
6 | | n informe $\mathcal{R}_i(n)$ de sa migration
7 | **fin**
8 **fin**

L'algorithme 6 décrit l'heuristique DLF. Initialement, le nœud n envoie un message de contrôle *verificationBoucle1* à $\mathcal{R}_f(n)$ et attend la réponse.

— Si $\mathcal{R}_f(n)$ détecte que $\mathcal{R}_i(\mathcal{R}_f(n)) = n$ (ligne 1 de l'algorithme 6, et figure 4.5(a)), alors $\mathcal{R}_f(n)$ envoie un message *boucleDetectee1* à n. Le nœud n se met ainsi en attente de notification après migration de $\mathcal{R}_f(n)$ (ligne 2 de l'algorithme 6). Dans le cas contraire, il envoie un message *pasDeBoucle1* (figure 4.5(b)).

— Si $\mathcal{R}_i(\mathcal{R}_f(n)) \neq n$ ou si la boucle a été traitée, n migre automatiquement (ligne 4 de l'algorithme 6), car il est plus proche de la destination suivant $\mathcal{R}_f$. Le nœud n détermine ensuite si $\mathcal{R}_f(\mathcal{R}_i(n)) = n$ ou non en envoyant un message de contrôle *verificationBoucle2* à $\mathcal{R}_i(n)$ (ligne 5 de l'algorithme 6). Le nœud n attend en retour soit un message de détection de boucle *boucleDetectee2*, soit un message d'absence de boucle *pasDeBoucle2*.

  — Si le message *boucleDetectee2* est reçu (cas de la figure 4.5(c)), n informe $\mathcal{R}_i(n)$ à travers un message *boucleResolue* (ligne 6 de l'algorithme 6).

  — Si le message *pasDeBoucle2* est reçu n (cas de la figure 4.5(d)), aucune autre action n'est requise.

Le processus entier nécessite 4 messages de contrôle par nœud plus un message de contrôle additionnel pour chaque boucle.

Considérons l'exemple de la figure 4.1 où 5 est le nœud à retirer. Les nœuds migrent dans des étapes et une nouvelle étape est ici initiée lorsque tous les nœuds ont soit migré, soit sont en attente d'un autre nœud. La figure 4.2 nous montre qu'un mauvais ordonnancement des nœuds peut entraîner deux boucles de routage : $(1, 7, 1)$ et $(1, 8, 1)$. L'exécution de DLF par chacun des nœuds donne comme résultat :

— $\mathcal{R}_i(\mathcal{R}_f(1)) = \mathcal{R}_i(8) = 1$: le nœud 1 est bloqué et se met en attente du nœud 8 pour pouvoir effectuer la migration.

— $\mathcal{R}_i(\mathcal{R}_f(7)) = \mathcal{R}_i(1) = 7$: le nœud 7 est bloqué et attend le signal du nœud 1 pour migrer à son tour.

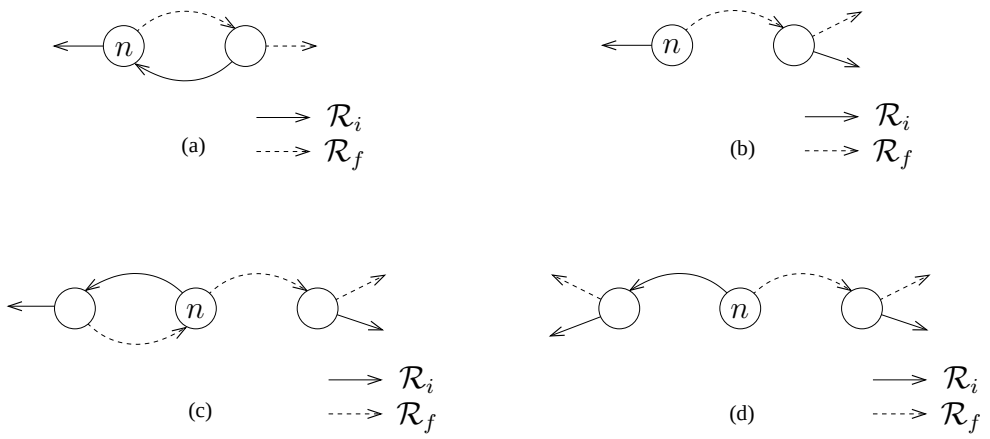

FIGURE 4.5 – Étude des cas pour un nœud n par DLF. (a) n est sur une boucle avec $\mathcal{R}_f(n)$, (b) n n'est pas sur une boucle avec $\mathcal{R}_f(n)$, (c) n sur une boucle avec $\mathcal{R}_i(n)$ mais pas avec $\mathcal{R}_f(n)$, (d) n n'est ni sur une boucle ni avec $\mathcal{R}_i(n)$ ni avec $\mathcal{R}_f(n)$.

— $\mathcal{R}_i(\mathcal{R}_f(8)) = \mathcal{R}_i(9) = 10 \neq 8$ et $\mathcal{R}_f(\mathcal{R}_i(8)) = \mathcal{R}_i(1) = 8$. Par conséquent, le nœud 8 migre immédiatement et informe le nœud 1 à travers le message *boucleResolue* qu'il peut à son tour migrer. Tous les autres nœuds migrent dans l'étape.

Ainsi la première étape de la transition finale est : $\{0, 2, 3, 4, 5, 6, 8, 9, 10\}$. Durant la seconde étape, 1 reçoit le message *boucleResolue* provenant de 8, avec le test $\mathcal{R}_f(\mathcal{R}_i(1)) = \mathcal{R}_f(7) = 1$. Il migre ainsi directement et informe le nœud 7 qu'il peut migrer à travers un autre message *boucleResolue*. Ainsi la seconde étape est $\{1\}$. Durant la troisième étape, 7 reçoit le message *boucleResolue* de 1, effectue le test $\mathcal{R}_f(\mathcal{R}_i(7)) = \mathcal{R}_f(5) = 5 \neq 7$, et migre directement sans besoin d'informer un autre nœud. La troisième étape est par conséquent égale à $\{7\}$. La transition finale est : $Tr = (\{0, 2, 3, 4, 5, 6, 8, 9, 10\}_{10}, \{1\}_{10}, \{7\}_{10})$.

Le nombre de messages de contrôle peut cependant être réduit car on peut faire abstraction des messages de contrôle *verificationBoucle2*, *boucleDetectee2* et *pasDeBoucle2*. En effet, lorsqu'une boucle est détectée par $\mathcal{R}_f(n)$ dû à la réception du message de contrôle *verificationBoucle1*, le nœud $\mathcal{R}_f(n)$ peut mémoriser qu'il doit informer le nœud n après sa propre migration. Ainsi, ce nouveau protocole ne nécessite que deux messages de contrôle *verificationBoucle1* et *boucleDetectee1*/*pasDeBoucle2* par nœud pour détecter une boucle, et un message de contrôle additionnel pour résoudre chaque boucle détectée.

4.2.2 DLF pour un réseau multi-destinations

En présence d'un réseau multi-destinations, DLF traite toutes les destinations en parallèle et procède de la même manière que RTH-d (à savoir, fusionner étape par étape les transitions obtenues pour chaque destination individuellement).

Reprenons l'exemple de la figure 4.4 qui montre les routages initial et final du réseau (figure 4.1) lorsque les destinations 2 et 10 sont considérées. DLF produit pour chacune des destinations la transition suivante :

— destination 2 : $Tr = (\{0, 1, 2, 3, 4, 5, 6, 8, 9, 10\}_2, \{7\}_2)$.
— destination 10 : $Tr = (\{0, 2, 3, 4, 5, 6, 8, 9, 10\}_{10}, \{1\}_{10}, \{7\}_{10})$.

La transition finale calculée par DLF est : $Tr = (\{0, 2, 3, 4, 5, 6, 8, 9, 10\}_{2,10} \cup$

$\{1\}_2, \{7\}_2 \cup \{1\}_{10}, \{7\}_{10})$ qui compte trois étapes.

4.2.3 Taille des boucles lors du retrait d'un nœud

D'après [48], les liens des topologies étant bidirectionnelles, toutes les boucles calculées sont de taille 2. Nous appuyons cela dans cette partie à l'aide d'un contre exemple.

Nous faisons l'hypothèse de liens symétriques. Le retrait d'un nœud r laisse deux configurations possibles pour deux autres nœuds n_j et n_k pris arbitrairement dans le réseau (avec $n_j \neq n_k$) :

1. Soit $r \in ch(n_j, n_k)$: le nœud à retirer appartient au plus court chemin entre n_j et n_k. Dans ce cas, aucune possibilité d'existence de boucle entre n_j et n_k.
2. Soit $r \notin ch(n_j, n_k)$: le nœud à retirer n'appartient pas au plus court chemin entre n_j et n_k.

Supposons que $r \notin ch(n_j, n_k)$ génère une boucle de routage $l = (n_j, n_{j+1}, .., n_k, n_j)$ de taille supérieure à 2 comme l'illustre la figure 4.6. Nous rappelons que chaque routage ($\mathcal{R}_i$ comme $\mathcal{R}_f$) pris individuellement est sans boucle. Par conséquent, l'existence de l implique nécessairement les deux protocoles de routage. Sans perte de généralité, $\mathcal{R}_f(n_k) = n_j$ et $\mathcal{R}_i(n_k) \neq n_j$. L'existence de l implique que le plus court chemin entre n_j et n_k suivant le protocole initial $\mathcal{R}_i$ est différent du plus court chemin entre n_j et n_k suivant le protocole final $\mathcal{R}_f$. Dans l'exemple de la figure 4.6, avant le retrait du nœud 4, il n'existe aucun nœud intermédiaire sur le plus court chemin entre les nœuds 5 et 7, tandis qu'après retrait du nœud 4, le plus court chemin entre les nœuds 7 et 5 passe par 8. Ceci est impossible car le poids des liens ne change pas avec le retrait d'un nœud. Par conséquent, le plus court chemin aussi reste inchangé entre ces deux nœuds. La figure 4.7 montre la configuration correcte des nœuds de l'exemple de la figure 4.6 après retrait du nœud 4 avec génération de la boucle $(5, 7, 5)$ de taille 2.

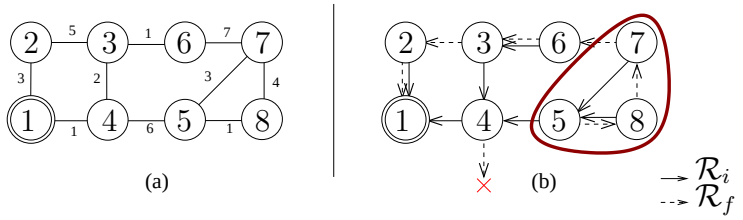

FIGURE 4.6 – Exemple de graphe avec panne d'un nœud. (a) Graphe initial, (b) union des routages avant et après retrait du nœud 4 générant une boucle incorrecte : $(5, 8, 7, 5)$ de taille 3.

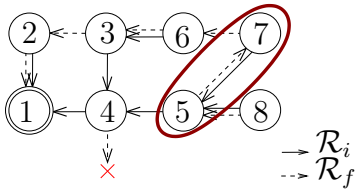

FIGURE 4.7 – Configuration correcte de l'exemple de la figure 4.6, et génération de la boucle $(5, 7, 5)$.

Nous notons que lorsque le poids des liens ne change pas dans le réseau, la taille des boucles générées par le retrait d'un nœud est toujours égale à 2. Ceci est également valable pour le retrait d'un nombre de nœuds supérieur à 1, comme l'illustre la

figure 4.8. Dans cet exemple, le retrait des nœuds 4 et 9 génère la boucle $(5, 8, 5)$.

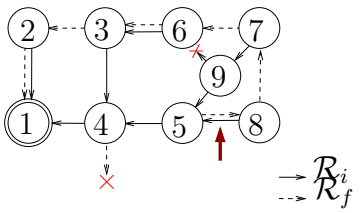

FIGURE 4.8 – Exemple d'un réseau de 9 nœuds avec retrait de deux nœuds.

4.2.4 Discussion sur la notification du changement

Nous avons précédemment fait l'hypothèse que chaque nœud connaissait à l'avance son prochain saut suivant le protocole initial $\mathcal{R}_i$, et son prochain saut suivant le protocole final $\mathcal{R}_f$. Nous discutons maintenant du retrait de cette hypothèse. Nous supposons donc qu'initialement, chaque nœud ne connaît que son prochain saut suivant $\mathcal{R}_i$. Lorsqu'une panne survient dans le réseau, les voisins du nœud qui tombe en panne sont les premiers à détecter la panne et se chargent d'informer le reste du réseau.

DLF peut être adapté de la manière suivante. Lorsqu'un nœud n détecte la panne de son voisin direct, il calcule son nouveau plus court chemin vers la destination et obtient son prochain saut noté $\mathcal{R}_f(n)$. n envoie ensuite à $\mathcal{R}_f(n)$ un message de contrôle *verificationBoucle1*. Si le nouveau prochain saut répond par un message de contrôle *pasDeBoucle1*, alors n migre immédiatement. Dans le cas contraire, n retarde sa migration jusqu'à la réception du message de contrôle *boucleResolue* de la part de $\mathcal{R}_f(n)$. Il bufferise tous les paquets reçus par la destination d. En parallèle, n informe ses voisins de la panne détectée. L'algorithme 7 décrit cette adaptation.

Algorithme 7 : Adaptation de DLF lorsque $\mathcal{R}_i$ uniquement connu *a priori*

Données : n est le nœud courant, $\mathcal{R}_i$ est connu *a priori* par tous les nœuds.

```
si n détecte la panne alors
    n informe ses voisins
    n calcule R_f(n)
    message_envoi(n,R_f(n),"verificationBoucle1")
    si (message_reception(n,R_f(n)) = "verificationBoucle1") alors
        n migre à R_f
    sinon
        répéter
            attendre()
        jusqu'à (message_reception(n,R_f(n)) = "boucleResolue");
        n migre à R_f
    fin
fin
```

Considérons le graphe de la figure 4.1 qui montre un exemple de réseau avec le nœud 5 qui tombe en panne. La figure 4.9(a) présente le routage avant la panne, lorsque la destination 10 est considérée. Nous représentons dans la suite les nœuds qui sont notifiés de la panne par la couleur grise. Lorsque le nœud 5 tombe en panne, ses voisins

$\{3, 6, 7, 9\}$ sont les premiers à être notifiés de la panne. Recalculant tous leur prochain saut suivant $\mathcal{R}_f$, seuls les nœuds $\{3, 6, 9\}$ sont autorisés à migrer comme l'illustre la figure 4.9(b), tandis que le nœud 7 continue de router suivant $\mathcal{R}_i$ parce qu'il reçoit le message de contrôle *boucleDetectee1* de la part de $\mathcal{R}_f(7) = 1$. De même, les nœuds $\{0, 1, 2, 4, 8, 10\}$ sont informés à leur tour de la panne par leurs voisins $\{3, 6, 7, 9\}$. Les nœuds $\{0, 2, 4, 8, 10\}$ migrent directement comme le montre la figure 4.9(c) tandis que le nœud 1 qui reçoit le message de contrôle *boucleDetectee1* de 8, son prochain saut suivant $\mathcal{R}_f$, continue de router suivant $\mathcal{R}_i$. Ainsi à cette étape, les nœuds 1 et 7 sont en attente du message de contrôle *boucleResolue*. Lorsque $\mathcal{R}_f(1) = 8$ migre, le nœud 8 informe le nœud 1 avec le message *boucleResolue*, et 1 migre à son tour comme le montre la figure 4.9(d). Le nœud 1 informe à son tour le nœud 7 qui migre aussi (figure 4.9(e)). La transition finale pour cette version de DLF est : $Tr = (\{3, 5, 6, 9\}, \{0, 2, 4, 8, 10\}, \{1\}, \{7\})$.

4.2.5 Complexité

Pour un réseau de n nœuds et d destinations, la complexité en temps de DLF est de l'ordre de $\mathcal{O}(1)$ car, tous les nœuds vérifient en parallèle s'ils ont le droit de migrer ou pas, par envoi de deux messages de contrôle et un message de notification en cas de boucle.

En termes d'étapes, DLF a une complexité très faible en étapes car ne dépend ni de n, ni de d.

4.2.6 Preuve de l'absence d'interblocage

Nous faisons une démonstration d'absence d'interblocage dans le cas de DLF par l'absurde. Les hypothèses posées sont les suivantes :

— le protocole de routage $\mathcal{R}_f$ est sans boucle de routage,
— aucune perte de messages durant la migration n'est possible,
— aucun nœud ne tombe en panne durant la migration,
— le temps de traitement pour chaque nœud est fini.

Nous rappelons qu'un interblocage correspond à la situation dans laquelle des nœuds sont en attente de notification d'autres nœuds, et ce de manière à former une boucle d'attente. Formellement, il y a interblocage entre les nœuds n_i avec $1 \leq i \leq n$ si $\exists\ m$ avec $2 \leq m \leq n$ tel que : n_1 attend n_2, n_2 attend n_3, ..., n_{m-1} attend n_m, et n_m attend n_1. Nous rappelons qu'avec DLF, un nœud n_i attend un nœud n_{i+1} si $\mathcal{R}_i(\mathcal{R}_f(n_i)) = n_i$ et $\mathcal{R}_f(n_i) = n_{i+1}$.

Considérons un réseau de n nœuds et supposons que DLF conduit à un interblocage entre les nœuds $\{n_1, n_2, ..., n_{m-1}, n_m\}$. Ceci se traduit par :

$$\begin{cases} n_1 \text{ attend } n_2 \Longrightarrow n_2 = \mathcal{R}_f(n_1) \\ n_2 \text{ attend } n_3 \Longrightarrow n_3 = \mathcal{R}_f(n_2) \\ ... \\ n_m \text{ attend } n_1 \Longrightarrow n_1 = \mathcal{R}_f(n_m) \end{cases}$$

Nous définissons ainsi A, l'ensemble des prochains sauts suivant $\mathcal{R}_f$:

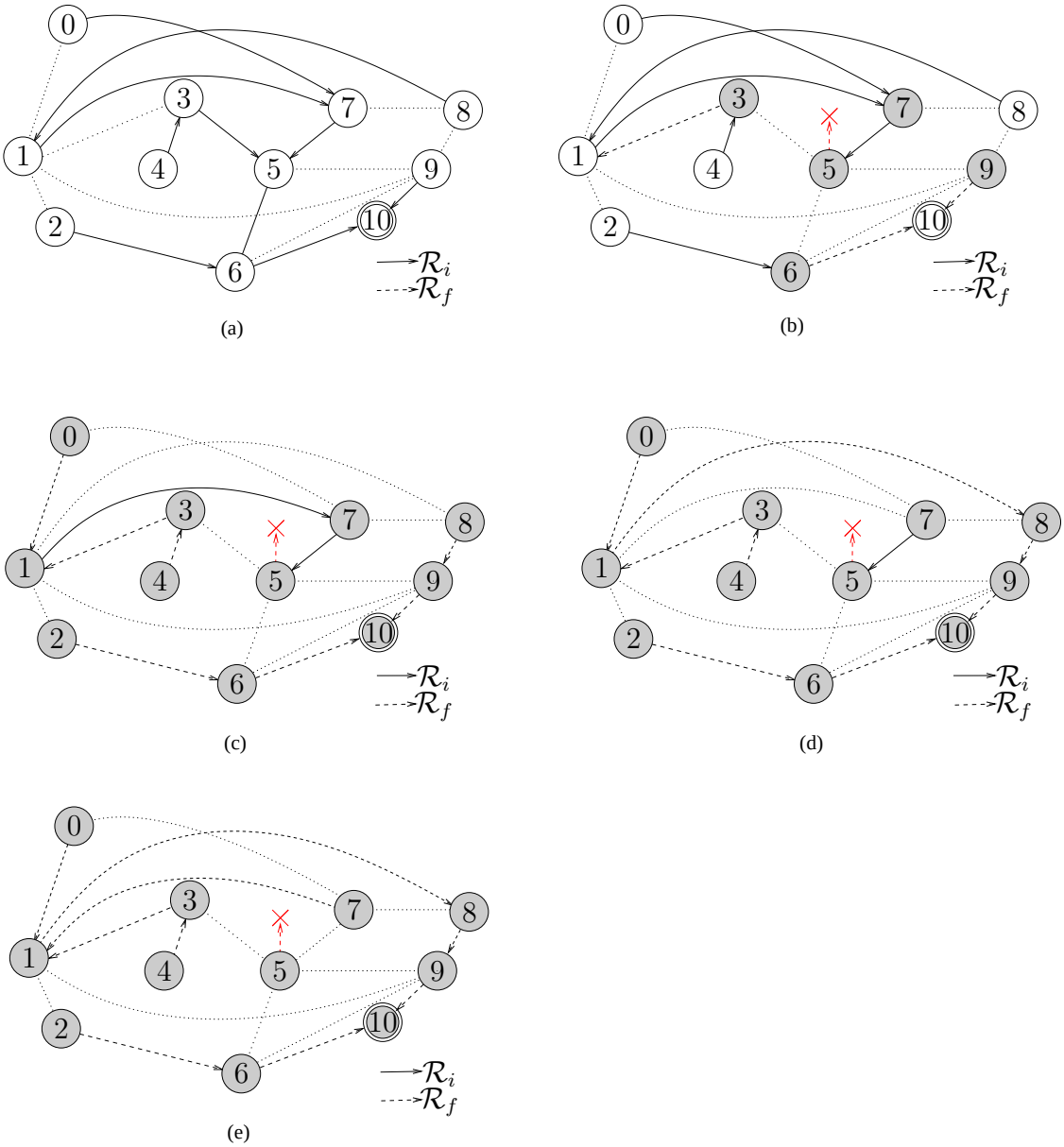

FIGURE 4.9 – Calcul de la transition sur l'exemple de la figure 4.1 par l'adaptation de DLF avec $\mathcal{R}_i$ uniquement connu à priori. (a) Routage initial vers la destination 10 avant la panne du nœud 5, (b) panne du nœud 5, détection de la panne par les nœuds $\{3, 6, 7, 9\}$, et migration des nœuds $\{3, 6, 9\}$, (c) notification des nœuds $\{0, 1, 2, 4, 8, 10\}$, et migration des nœuds $\{0, 2, 4, 8, 10\}$, (d) migration du nœud $\{1\}$, (e) migration du nœud $\{7\}$.

$$A : \begin{cases} n_2 = \mathcal{R}_f(n_1) \\ n_3 = \mathcal{R}_f(n_2) \\ ... \\ n_1 = \mathcal{R}_f(n_m) \end{cases}$$

L'ensemble A est décrit par la figure 4.10.

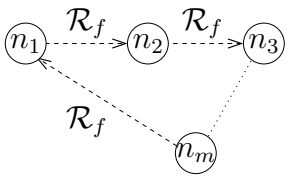

FIGURE 4.10 – Induction impossible d'un cas d'interblocage : boucle sur $\mathcal{R}_f$.

La figure 4.10 montre que l'ensemble A conduit à une boucle impliquant uniquement le protocole de routage final $\mathcal{R}_f$. Cela est impossible car, le protocole de routage $\mathcal{R}_f$ pris individuellement est supposé être sans boucle. Par conséquent, un cas d'interblocage ne peut survenir.

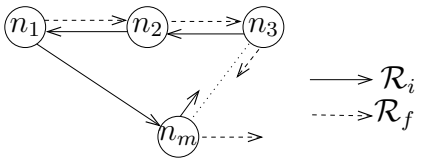

FIGURE 4.11 – Union des protocoles de routage $\mathcal{R}_i$ et $\mathcal{R}_f$ avec $\mathcal{R}_f$ sans boucle et $\mathcal{R}_i$ avec boucle.

Supposons à présent que le protocole de routage initial $\mathcal{R}_i$ conduit à une boucle, et que le protocole de routage final $\mathcal{R}_f$ est quant à lui sans boucle comme le montre la figure 4.11. Nous définissons dans ce cas l'ensemble B des prochains sauts suivant $\mathcal{R}_f$, et C l'ensemble des prochains sauts suivant $\mathcal{R}_i$:

$$B : \begin{cases} \mathcal{R}_i(n_2) = n_1 \\ \mathcal{R}_i(n_1) = n_m \\ ... \\ \mathcal{R}_i(n_3) = n_2 \end{cases} \text{ et } C : \begin{cases} \mathcal{R}_f(n_1) = n_2 \\ \mathcal{R}_f(n_2) = n_3 \\ ... \\ \mathcal{R}_f(n_m) \neq n_k, \forall k \in [1, n] \end{cases}$$

Cela traduit le fait que n_1 attend n_2, n_2 attend n_3,..., n_m n'attend aucun n_k où $1 \leq k \leq n$. Le nœud $\mathcal{R}_f(n_m)$ étant plus proche de la destination que n_m, migre puis débloque n_m. n_m migre à son tour et débloque n_{m-1} . Le processus est répété jusqu'à ce que n_1 migre à son tour. On ne sera jamais en présence d'un interblocage malgré le fait que $\mathcal{R}_i$ est un protocole avec boucle. Ce qui témoigne de la robustesse de DLF.

En somme, quel que soit l'état du protocole de routage initial $\mathcal{R}_i$, DLF ne sera jamais pris dans un interblocage si le protocole de routage final $\mathcal{R}_f$ est sans boucle.

4.3 Résultats

Dans cette partie, nous comparons les performances de nos heuristiques RTH-d et DLF à celles des heuristiques centralisées GBA, SCH-m et ACH.

4.3.1 Métriques de performance

Dans la suite, nous utilisons les métriques suivantes.

La durée de la transition en nombre d'étapes

La durée de la transition est la principale métrique. Elle est calculée comme le nombre d'étapes nécessaire pour effectuer la migration. Pour les heuristiques distribuées, lorsqu'un nœud n attend la notification d'un autre nœud $n\prime$, n migre dans l'étape qui suit l'étape de $n\prime$.

Le coût de la transition en nombre de messages de contrôle

Le coût de la transition est égal au nombre de messages nécessaires pour effectuer la transition. Nous rappelons que pour les heuristiques centralisées, nous supposons qu'il existe une entité centrale qui est chargée de notifier chaque nœud de sa migration vers une destination et ce, au début de chaque étape. Cette entité est choisie à la position la plus centrale dans le réseau pour minimiser le nombre de messages de contrôle nécessaires, et le coût prend en compte le nombre de sauts pour atteindre chaque nœud de l'étape. Pour les heuristiques distribuées, nous supposons que les deux protocoles de routage sont connus d'avance par chacun des nœuds. Le nombre de messages nécessaires pour notifier chaque nœud de sa migration est pris en compte pour RTH-d et DLF, avec en plus pour DLF le nombre de messages nécessaires pour identifier chacune des boucles.

Considérons que le nœud 10 est la destination traitée pour l'exemple de la figure 4.1. Supposons également que l'entité centrale pour les heuristiques centralisées est le nœud 10. Alors, le coût de la transition pour GBA, SCH-m, et ACH est égal à 29 (qui est la somme des nombres de sauts de tous les chemins des nœuds vers 10 suivant le protocole de routage initial $\mathcal{R}_i$). Quant aux heuristiques distribuées, RTH-d nécessite 9 messages de contrôle, soit un message par nœud notifiant sa migration, et DLF nécessite 20 messages de contrôle, soit 2 messages par nœud (hors destination) pour la notification et un message pour chacune des deux boucles.

La vitesse de correction d'une panne en nombre d'étapes

La vitesse de correction d'une panne est égale au nombre d'étapes nécessaire à une transition pour que tous les nœuds routent soit suivant le protocole de routage final $\mathcal{R}_f$, soit suivant le protocole de routage initial $\mathcal{R}_i$ avec dans ce cas un chemin n'empruntant pas le routeur qui tombe en panne. Cette métrique traduit le temps nécessaire pour une transition sans perte potentielle de paquets. En effet, une perte se produit quand un nœud tombe en panne et qu'un autre nœud continue de router vers ce dernier. La dernière étape où cela se produit permet d'évaluer le temps de correction nécessaire de la panne.

Le temps de calcul centralisé ou distribué en secondes

Le temps de calcul représente le temps système que met une heuristique à produire la transition finale à appliquer pour que les nœuds changent de protocole de routage sans générer de boucles de routage, sur l'ordinateur de test qui a pour caractéristiques : un processeur Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, et une mémoire de 2 Go.

Le temps de calcul centralisé est récupéré pour toutes les heuristiques, y compris nos heuristiques distribuées implémentées pour ce faire de manière centralisée.

Le temps distribué correspond quant à lui au temps de calcul des heuristiques distribuées, obtenu à l'aide d'une implémentation avec des *threads* (cf annexe A.10).

4.3.2 Environnement et paramètres de simulation

Les simulations ont été faites sur des graphes aléatoires et réels. Pour les graphes aléatoires, les tailles varient entre 10 et 300 nœuds (tableau 4.1). Les nœuds sont déployés uniformément de manière aléatoire dans un espace de dimension $100m \times 100m$. La portée des nœuds a été fixée à $20m$. Pour les graphes réels, nous avons considéré les deux groupes utilisés dans le chapitre 3 à savoir : les topologies *ITZ* [47] (tableau 3.4) collectées dans l'*internet Topology Zoo* de l'université d'Adelaide [47], dont les tailles varient entre 9 et 278, et les topologies *Rocketfuel* [12] (tableau 3.5) dont les tailles varient entre 79 et 315. Les topologies *ITZ* ont été sélectionnées pour leur taille de manière à couvrir l'intervalle entre 0 et 300.

Nous considérons que chaque nœud est une destination potentielle. Pour chaque destination d, nous avons généré le routage initial $\mathcal{R}_i^d$ en attribuant un poids aléatoire dans l'intervalle $[1; 100]$ à chaque lien, et en calculant le chemin le plus court de chaque nœud vers la destination d selon ces poids. De la même manière, nous avons généré le routage final $\mathcal{R}_f^d$ en calculant le plus court chemin de chaque nœud vers la destination d dans la topologie où un nœud arbitraire tombe en panne. Les simulations sont répétées 100 fois, et l'intervalle de confiance pour les courbes est de 95%. L'échelle logarithmique est utilisée pour l'axe des ordonnées dans certains cas.

Taille	Nombre moyen de liens	Diamètre
10	25	6
20	59	9
30	105	11
40	168	12
50	257	11
100	1039	9
150	2354	8
200	4169	8
250	6511	8
300	9419	8

TABLE 4.1 – Tableau des données des topologies aléatoires.

4.3.3 Résultats sur les topologies aléatoires

La durée de la transition : La figure 4.12 montre la durée de la transition calculée comme le nombre moyen d'étapes nécessaires pour que les nœuds migrent du protocole initial $\mathcal{R}_i$ au protocole final $\mathcal{R}_f$ pour toutes les destinations. Nous constatons que l'heuristique RTH-d est celle qui produit le plus d'étapes, et ce nombre croît avec la taille du réseau. Ceci s'explique par le fait qu'avec RTH-d, la durée de la transition est égale à la profondeur de l'arbre selon $\mathcal{R}_f$. Plus la topologie du réseau est grande, plus RTH-d construit d'étapes aboutissant à de longues transitions. Nous constatons ensuite que comparativement à l'heuristique distribuée DLF et les heuristiques centralisées

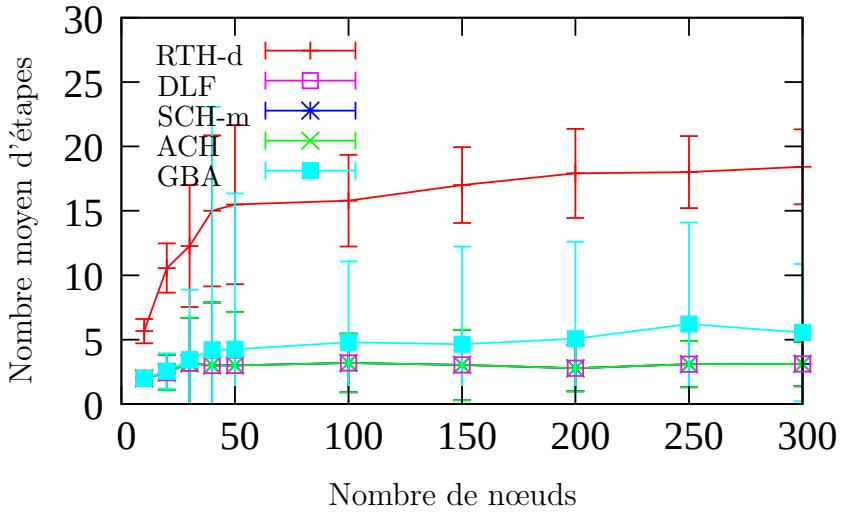

FIGURE 4.12 – Durée de la transition en nombre d'étapes, en fonction du nombre de nœuds, pour le cas des topologies aléatoires.

SCH-m et ACH, GBA est l'heuristique qui produit les transitions les plus longues. En effet, pour GBA, le calcul des étapes dépend du nombre de boucles de routage. Si le nombre de boucles est élevé, GBA ne peut pas calculer la transition optimale dû à la construction gloutonne des étapes. L'allure de la courbe produite par GBA suit ainsi celle de la courbe du nombre moyen de boucles en fonction de la taille de réseau, présentée à la figure 4.13. Nous constatons également que ACH et SCH-m produisent des transitions avec un très petit nombre d'étapes quelle que soit la taille du réseau. Ceci est lié au fait que ACH et SCH-m reposent sur le concept des composantes fortement connexes, et ces dernières permettent de traiter efficacement des boucles de petite taille comme c'est le cas dans notre contexte. De la même manière, la figure 4.12 nous montre enfin que DLF produit de très petits nombres d'étapes pour chaque taille de réseau. Ceci s'explique par le fait que les boucles de petite taille peuvent être traitées très efficacement de manière indépendante. DLF montre un gain allant jusqu'à 50% par à rapport à GBA et un gain allant jusqu'à 83% par rapport à RTH-d.

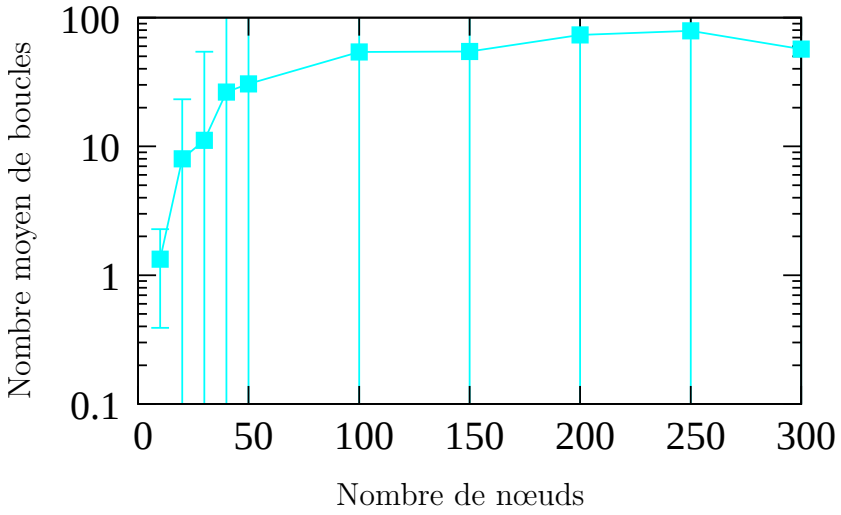

FIGURE 4.13 – Nombre moyen de boucles en fonction du nombre de nœuds, pour le cas des topologies aléatoires.

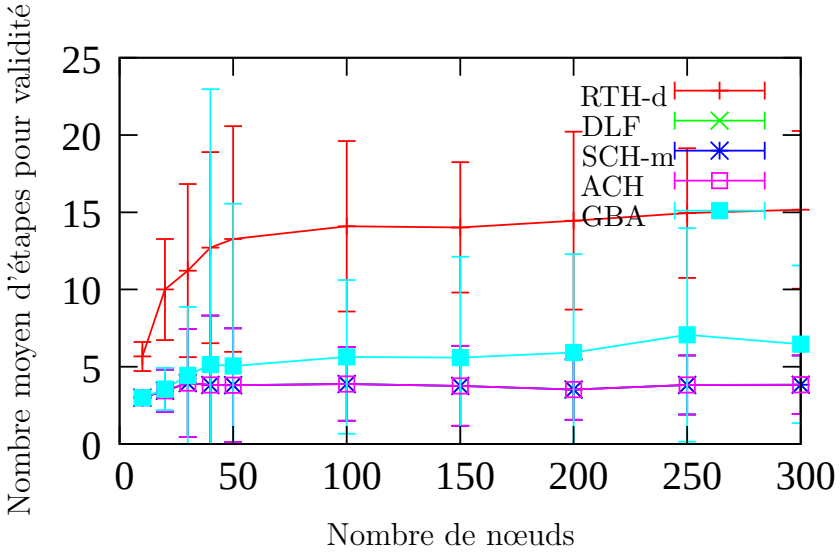

FIGURE 4.14 – Nombre moyen d'étapes nécessaires pour qu'il n'y ait aucune perte de paquets, en fonction du nombre de nœuds, pour le cas des topologies aléatoires.

La vitesse de correction d'une panne : La figure 4.14 montre la vitesse de correction d'une panne dans le cas des topologies aléatoires. Nous constatons que chacune des différentes heuristiques suit le même comportement que sur la figure 4.12 qui donne la durée de la transition en nombre moyen d'étapes nécessaires. Nous remarquons que pour les quatre heuristiques ACH, SCH-m,GBA et DLF, cette vitesse est sensiblement égale à la durée de la transition car le nombre moyen d'étapes nécessaires à la correction de la panne est égal au nombre moyen d'étapes (cf figure 4.12) plus un. Nous constatons cependant une légère différence avec l'heuristique RTH-d (cf figure 4.12). En effet, pour chaque taille de réseau, il faut en moyenne trois étapes en moins par rapport à la durée de la transition pour corriger la panne pour le cas de RTH-d. Ceci s'explique par le fait que RTH-d est une solution simple mais non optimale.

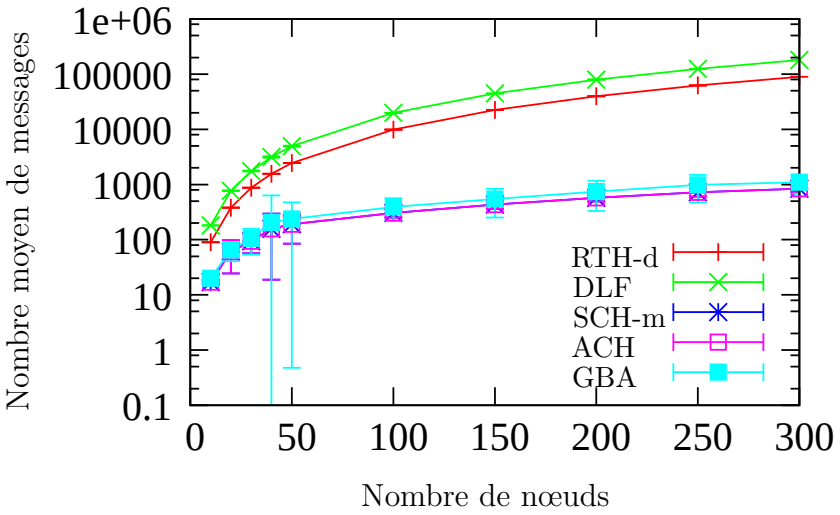

FIGURE 4.15 – Coût de la transition, en fonction du nombre de nœuds, pour le cas des topologies aléatoires.

Le coût de la transition : La figure 4.15 montre le coût de la transition sur des topologies aléatoires, calculé comme le nombre moyen de messages de contrôle nécessaires

pour effectuer la transition pour toutes les destinations considérées. Nous constatons que quelle que soit l'heuristique, le coût de la transition augmente avec la taille du réseau. Les heuristiques centralisées ACH, SCH-m et GBA produisent moins de messages de contrôle que nos heuristiques distribuées DLF et RTH-d. Le nombre de messages de contrôle pour ces heuristiques représente le coût dû à la notification des étapes par l'entité centrale. Nous notons un gain de 24% de ACH et SCH-m par rapport à GBA, et un gain de 99% par rapport à RTH-d et à DLF. L'entité centrale fusionne les notifications pour toutes les destinations, alors que RTH-d et DLF émettent des messages pour les destinations indépendamment. Pour nos heuristiques distribuées, RTH-d produit un nombre de messages de contrôle égal à $n-1$ par destination avec n la taille du réseau. Ce nombre correspond au nombre de messages nécessaires pour que tous les nœuds du réseau soient informés par inondation suivant le protocole de routage final partant de la destination. La figure 4.15 nous montre également que DLF produit plus de messages que RTH-d. En effet, lorsque RTH-d nécessite un message pour notifier un nœud de sa migration pour une destination, DLF en nécessite deux avec en plus un message pour chaque boucle identifiée. C'est pourquoi RTH-d montre un gain de 57% par rapport à DLF.

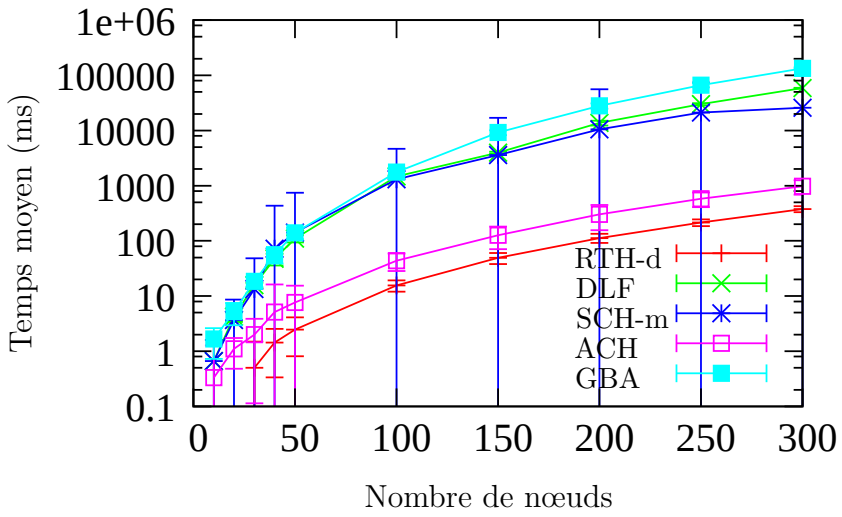

FIGURE 4.16 – Temps de calcul de la transition en fonction du nombre de nœuds, pour le cas des topologies aléatoires.

Le temps de calcul centralisé : La figure 4.16 montre le temps moyen d'exécution de chaque heuristique sur la machine de la simulation. Nous constatons tout d'abord que pour chacune d'elle, ce temps croît avec la taille du réseau. Cela s'explique simplement par le fait que plus la taille augmente, plus le nombre de nœuds devant être traités augmente. Nous constatons ensuite que les heuristiques RTH-d et ACH sont plus rapides que les heuristiques SCH-m, DLF et GBA avec RTH-d la plus rapide de toutes. En effet, RTH-d montre un gain de 61% par rapport à ACH, RTH-d et ACH montrent un gain respectivement de 98% et 96% par rapport à SCH-m (sensiblement le même gain par rapport à DLF sauf pour les grandes topologies où il vaut 99%), et un gain de 99% par rapport à GBA. La rapidité de RTH-d s'explique par le fait que les nœuds migrent par inondation du réseau partant de la destination et par profondeur suivant aussi le protocole final $\mathcal{R}_f$.

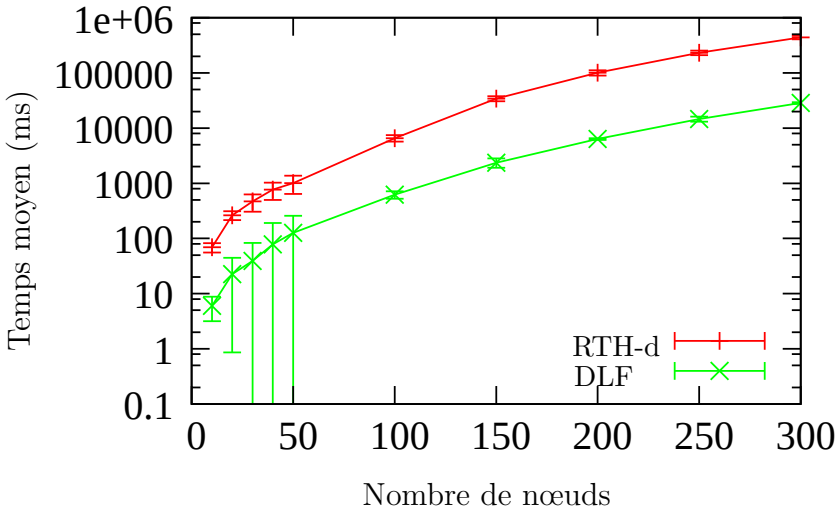

FIGURE 4.17 – Temps de calcul de la transition récupéré à travers une implémentation utilisant les *threads*, en fonction du nombre de nœuds, pour le cas des topologies aléatoires.

Le temps de calcul distribué : La figure 4.17 montre le temps de calcul des heuristiques distribuées DLF et RTH-d sur les topologies aléatoires lorsque une simulation est faite à l'aide de *threads*. La figure montre que les temps d'exécution augmentent avec la taille du réseau pour les deux heuristiques. La figure montre également que DLF s'exécute plus rapidement que RTH-d quelle que soit la taille du réseau avec un gain allant jusqu'à 93%. La rapidité de DLF s'explique par le fait que chaque nœud n effectue individuellement son test et migre directement sauf en cas d'attente de notification de son prochain saut $\mathcal{R}_f(n)$ avec lequel il est impliqué dans une boucle de routage. Tandis que RTH-d migre les nœuds par profondeur en partant de la destination.

4.3.4 Résultats sur les topologies *ITZ*

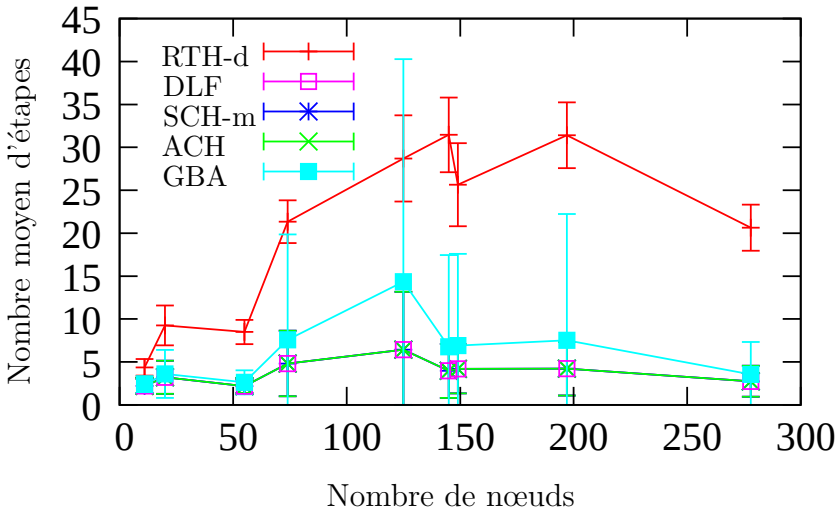

FIGURE 4.18 – Durée de la transition en nombre d'étapes, en fonction du nombre de nœuds, pour le cas des topologies *ITZ*.

La durée de la transition : La figure 4.18 montre la durée de la transition dans le cas des topologies *ITZ*. La figure montre que chacune des heuristiques a une allure non

lisse sur l'ensemble des tailles de réseau. Ceci est dû d'une part, à la nature des données traitées qui sont des topologies réalistes et d'autre part, au nombre très variable de boucles de routages identifiées , qui est présenté sur la figure 4.19. Cette dernière montre que, le nombre moyen de boucles ne croît pas de manière constante. La figure 4.18 nous montre également que RTH-d produit toujours le plus grand nombre d'étapes quelle que soit la taille du réseau. A l'opposé, DLF produit de très faibles nombres de boucles à l'image des heuristiques centralisées ACH et SCH-m. En résumé, DLF montre un gain de 55% par rapport à GBA, et un gain de 87% par à rapport à RTH-d.

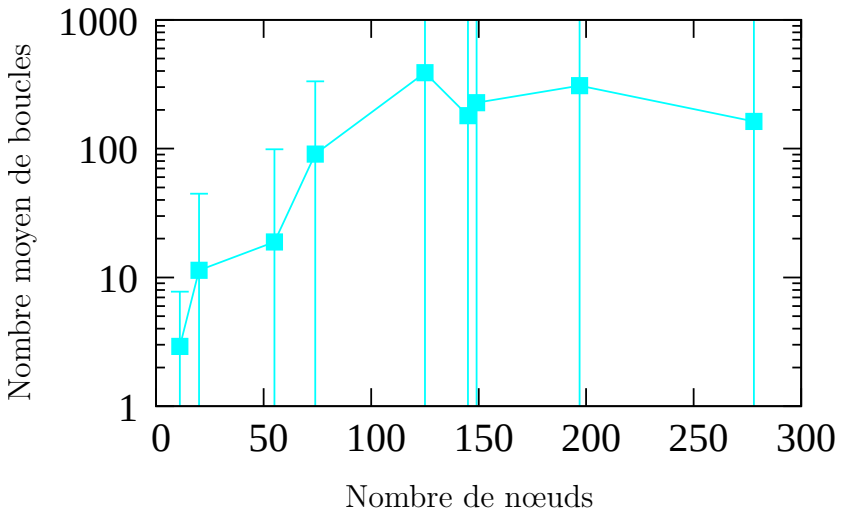

FIGURE 4.19 – Nombre moyen de boucles en fonction du nombre de nœuds, pour le cas des topologies *ITZ*.

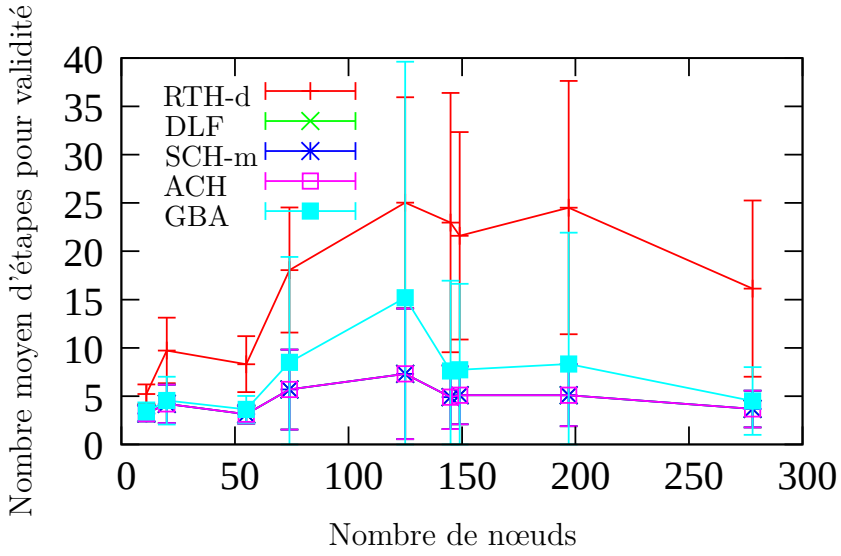

FIGURE 4.20 – Nombre moyen d'étapes nécessaires pour qu'il n'y ait aucune perte de paquets, en fonction du nombre de nœuds, pour le cas des topologies *ITZ*.

La vitesse de correction d'une panne : La figure 4.20 présente la vitesse de correction de la panne pour chaque taille de réseau dans le cas des topologies *ITZ*. Nous constatons, comme pour le cas aléatoire, que chaque heuristique suit le même comportement que sur la figure 4.18 de durée de transition calculée en nombre d'étapes

nécessaires avec une légère différence dans le cas du jeu de données *TataNld* de 145 nœuds pour RTH-d. Nous constatons de même que pour les quatre heuristiques DLF, ACH, SCH-m et GBA, le nombre d'étapes nécessaires pour qu'aucun nœud ne route suivant le nœud en panne est égal au nombre d'étapes nécessaires pour effectuer la transition plus un. Ceci traduit le cas où tous les nœuds ont migré vers le routage final signifiant que les nœuds corrigeant la panne sont les derniers à migrer. Par contre, l'heuristique RTH-d montre une différence allant jusqu'à cinq étapes en moins (particulièrement pour les grandes topologies) par rapport au nombre d'étapes requis pour effectuer la migration. Ceci est lié au fait que RTH-d n'est pas optimale. En somme, DLF montre un gain qui va jusqu'à 52% par rapport à GBA et un gain qui va jusqu'à 79% par rapport à RTH-d.

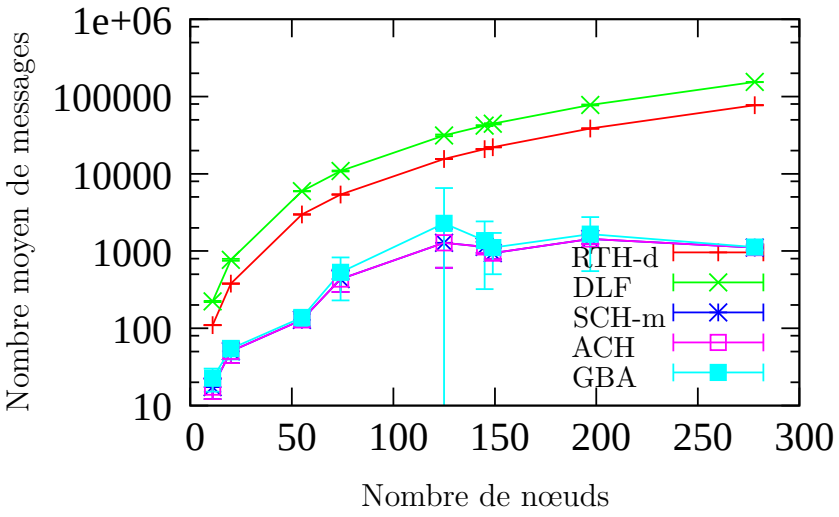

FIGURE 4.21 – Coût de la transition, en fonction du nombre de nœuds, pour le cas des topologies *ITZ*.

Le coût de la transition : La figure 4.21 montre le coût de la transition pour chaque heuristique dans le cas des topologies *ITZ*. Le coût croît avec la taille du réseau pour toutes les heuristiques. Les heuristiques distribuées DLF et RTH-d produisent plus de messages de contrôle que les heuristiques centralisées ACH, SCH-m et GBA. Nous rappelons que les heuristiques centralisées élisent le nœud le plus central (pour diminuer le coût) dans le réseau comme celui devant notifier chaque nœud de sa migration. Etant donné que le nombre de boucles calculées est plus petit que le nombre de messages pour chaque nœud et pour chaque destination, DLF nécessite environ deux fois plus de messages de contrôle que RTH-d. ACH et SCH-m montrent un gain qui va jusqu'à 44% par rapport à GBA, et toutes les trois montrent un gain qui va jusqu'à 94% par rapport à RTH-d et 96% par rapport à DLF. RTH-d quant à elle montre un gain de 50% par rapport à DLF.

Le temps de calcul centralisé : La figure 4.22 montre le temps de calcul des heuristiques dans le cas des topologies *ITZ*. Comme dans le cas des topologies aléatoires, les heuristiques distribuées RTH-d et ACH s'exécutent plus rapidement que les heuristiques centralisées DLF, GBA et SCH-m (classées dans l'ordre croissant de rapidité). RTH-d est l'heuristique la plus rapide de toutes. En effet, RTH-d montre un gain allant jusqu'à 63% par rapport ACH, et un gain allant jusqu'à 99% par rapport à DLF, GBA et SCH-m.

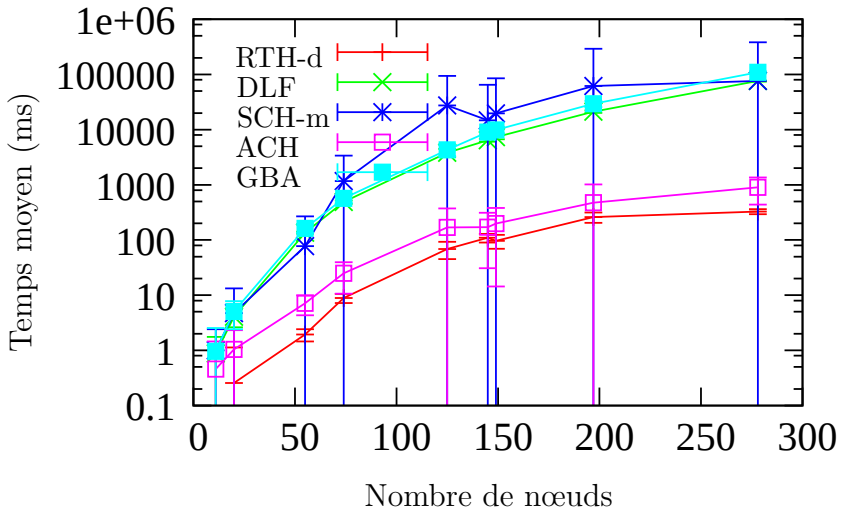

FIGURE 4.22 – Temps de calcul de la transition en fonction du nombre de nœuds, pour le cas des topologies *ITZ*.

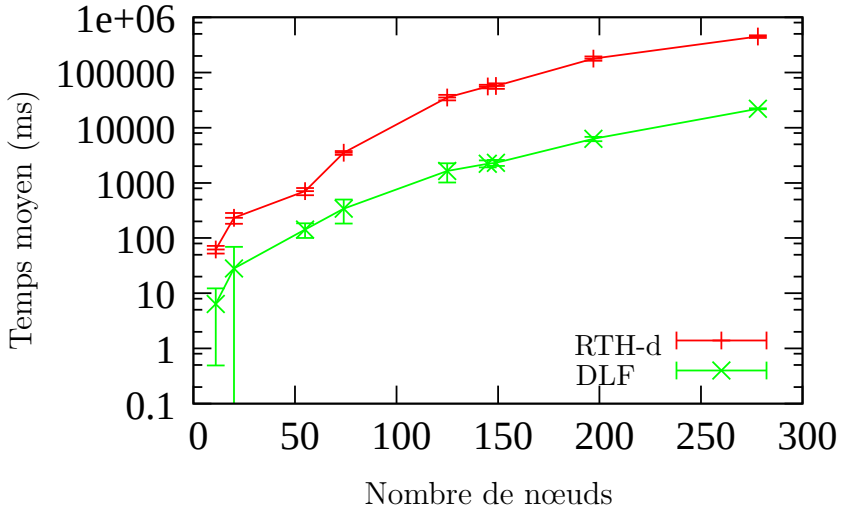

FIGURE 4.23 – Temps de calcul de la transition obtenu à travers une implémentation utilisant les *threads*, en fonction du nombre de nœuds, des heuristiques distribuées RTH-d et DLF, pour le cas des topologies ITZ.

Le temps de calcul distribué : La figure 4.23 montre le temps de calcul des heuristiques distribuées DLF et RTH-d pour les topologies ITZ, lorsque la simulation distribuée est faite à l'aide des *threads*. Comme dans le cas des topologies aléatoires, le temps de calcul croît avec la taille du réseau pour chacune des deux heuristiques, et DLF s'exécute plus rapidement que RTH-d. En effet, DLF montre un gain allant jusqu'à 95% par rapport à RTH-d.

4.3.5 Résultats sur les topologies *Rocketfuel*

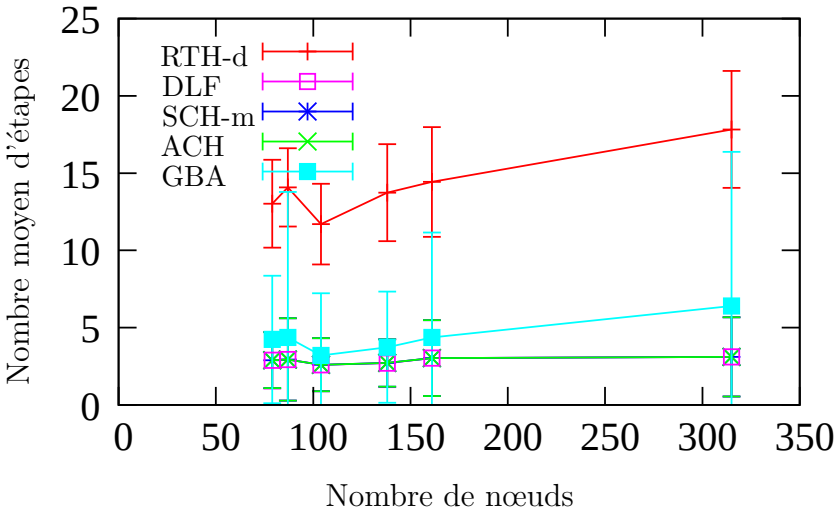

FIGURE 4.24 – Durée de la transition en nombre d'étapes, en fonction du nombre de nœuds pour le cas des topologies *Rocketfuel*.

La durée de la transition : La figure 4.24 montre la durée de la transition pour le cas des topologies *Rocketfuel*. Nous remarquons que DLF (comme ACH et SCH-m), produit un très faible nombre d'étapes, quelle que soit la taille du réseau. A l'opposé, RTH-d produit pour chaque taille de réseau le plus grand nombre d'étapes, parce que le nombre d'étapes que produit RTH-d dépend de la profondeur du réseau suivant le protocole de routage final $\mathcal{R}_f$. La courbe RTH-d sur la figure 4.24 permet d'affirmer que cette profondeur croît avec la taille du réseau, sauf pour le réseau *Telstra* de 104 nœuds pour lequel on note une chute. Nous remarquons que pour ces topologies, GBA produit un faible nombre d'étapes (autour de 5) avec un nombre d'étapes légèrement supérieur à 5 pour le réseau *Sprint* 315 de nœuds. (La figure 4.25 nous montre que le seuil de 100 boucles n'est atteint que pour ce réseau.) En somme, DLF montre un gain jusqu'à 50% par rapport à GBA, et un gain de 82% par rapport à RTH-d.

La vitesse de correction d'une panne : La figure 4.26 montre la vitesse de correction de la panne pour le cas des topologies *Rocketfuel*. Elle confirme le comportement observé dans les autres jeux de données à savoir : la vitesse de correction de panne pour chaque heuristique suit le comportement de la durée de la transition présentée à la figure 4.24. Les heuristiques ACH, SCH-m, GBA et DLF corrigent la panne avec les derniers nœuds qui migrent car le nombre d'étapes nécessaires pour qu'aucun nœud ne route suivant le nœud en panne est égal au nombre d'étapes nécessaires pour effectuer la transition plus un. L'heuristique RTH-d corrige la panne en 3 étapes de moins que la durée de la transition.

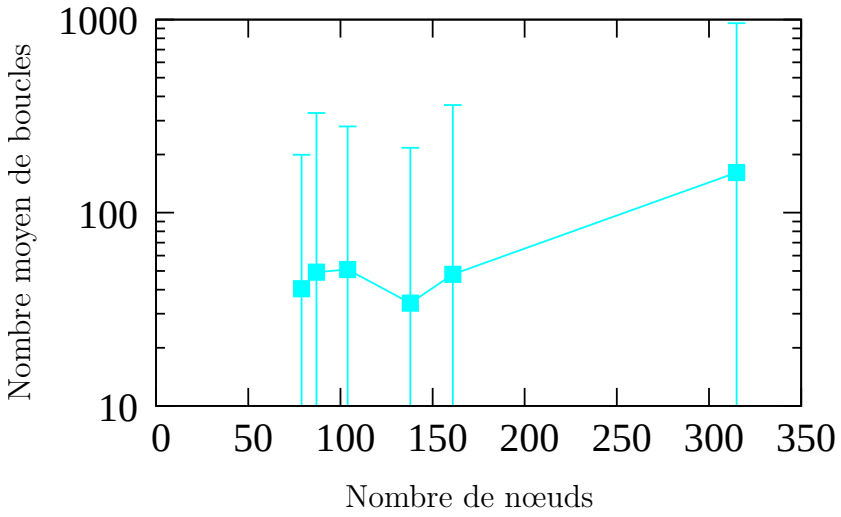

FIGURE 4.25 – Nombre moyen de boucles en fonction du nombre de nœuds, pour le cas des topologies *Rocketfuel*.

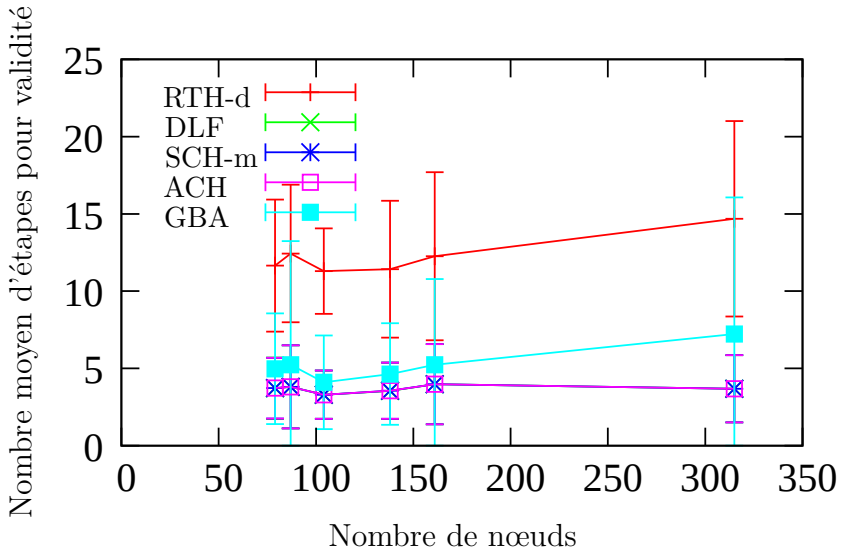

FIGURE 4.26 – Nombre moyen d'étapes nécessaires pour qu'il n y'ait aucune perte de paquets, en fonction du nombre de nœuds, pour le cas des topologies *Rocketfuel*.

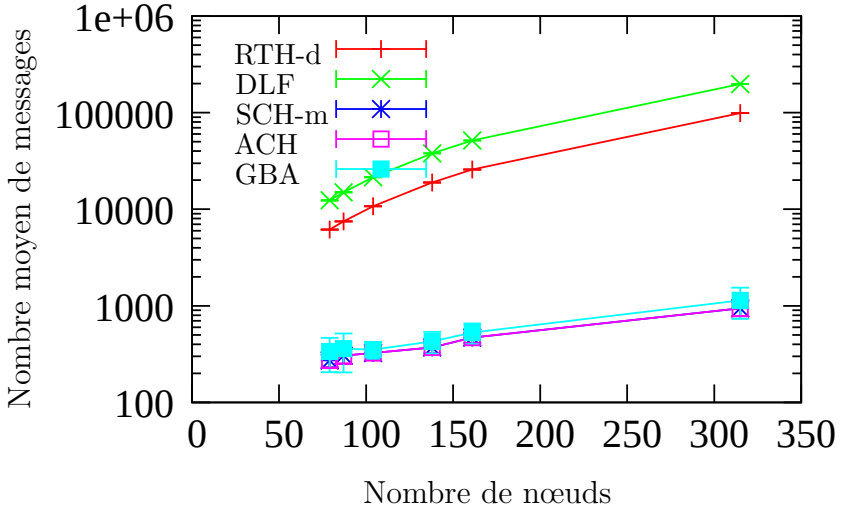

FIGURE 4.27 – Coût de la transition, en fonction du nombre de nœuds, pour le cas des topologies *Rocketfuel*.

Le coût de la transition : La figure 4.27 montre le coût de la transition dans le cas des topologies *Rocketfuel*. Nous remarquons que tous les coûts augmentent avec la taille du réseau. Nous remarquons également que toutes les heuristiques nécessitent un nombre limité de messages de contrôle pour effectuer la transition pour chaque réseau. Néanmoins, une séparation nette existe entre le coût des heuristiques centralisées et celui des heuristiques distribuées. En effet, DLF et RTH-d produisent beaucoup plus de messages de contrôle que ACH, SCH-m et GBA. Le faible nombre de messages de contrôle de ACH, SCH-m et GBA est lié au faible nombre d'étapes générées pour chaque réseau (figure 4.24). Par opposition, le coût de DLF et RTH-d dépend surtout du nombre de destinations dans le réseau. Ainsi, ACH et SCH-m montrent un léger gain de 17% par rapport à GBA, un gain de 92% par rapport à RTH-d, et un gain de 99% par rapport à DLF. RTH-d montre un gain de 93% par rapport à DLF.

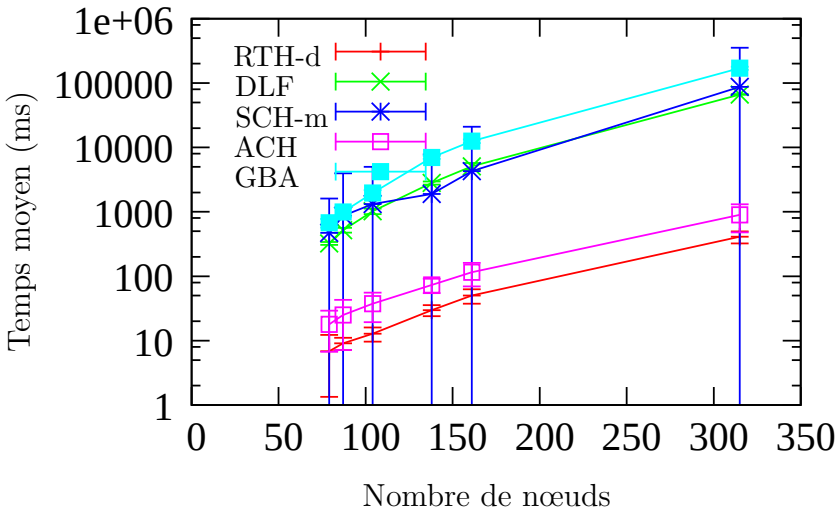

FIGURE 4.28 – Temps de calcul de la transition en fonction du nombre de nœuds, pour le cas des topologies *Rocketfuel*.

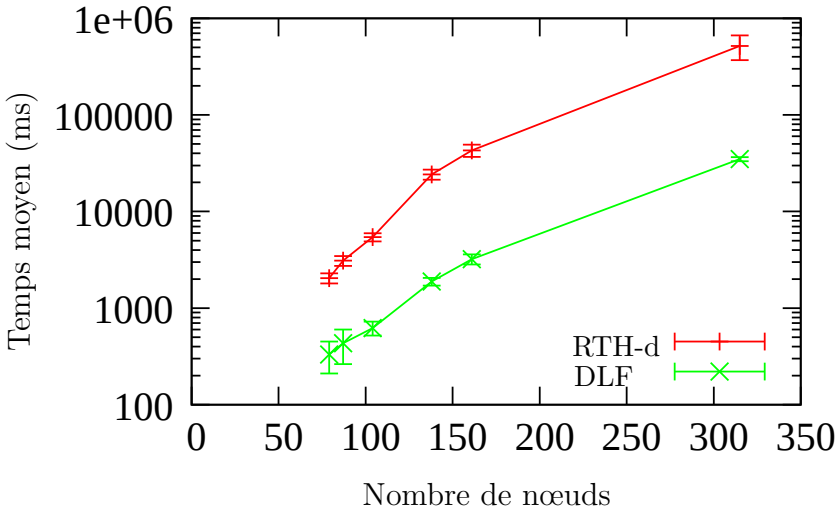

FIGURE 4.29 – Temps de calcul de la transition obtenu à travers une implémentation utilisant les *threads*, en fonction du nombre de nœuds des heuristiques distribuées DLF et RTH-d, pour le cas des topologies *Rocketfuel*.

Le temps de calcul centralisé : La figure 4.28 montre le temps d'exécution des différentes heuristiques sur les topologies *Rocketfuel*. Le temps augmente avec la taille du réseau quelle que soit l'heuristique. Cependant, la figure nous montre que RTH-d et ACH sont plus rapides que les heuristiques SCH-m, DLF et GBA. RTH-d montre un gain de 54% par rapport à ACH, et un gain de 99% par rapport à SCH-m, DLF et GBA.

Le temps de calcul distribué : La figure 4.29 montre le temps de calcul des heuristiques DLF et RTH-d pour les topologies *Rocketfuel* avec une simulation distribuée à l'aide des *threads*. Comme pour les topologies aléatoires et ITZ, la figure 4.29 montre que le temps augmente avec la taille du réseau, et que DLF s'exécute plus rapidement que RTH-d avec un gain allant jusqu'à 93%.

Conclusion

Dans ce chapitre, nous avons présenté deux approches distribuées d'ordonnancement des nœuds dans un contexte statique, pour le cas d'une panne ou du retrait d'un nœud.

La première approche, RTH-d [45], construit les différentes étapes d'une transition en ne considérant que le protocole de routage final $\mathcal{R}_f$ et en faisant migrer les nœuds par profondeur au travers d'un échange de messages : partant de la destination, un nœud n après migration informe par envoi d'un message les nœuds pour lesquels il est prochain saut sur $\mathcal{R}_f$. Ces derniers migrent à leur tour, puis répètent le processus.

La deuxième approche, DLF [45], exploite la propriété de l'article [43] qui stipule que le retrait d'un nœud dans un réseau génère des boucles de taille deux uniquement. DLF propose qu'après la détection d'une panne d'un nœud, un nœud n ne migre que s'il n'est pas dans une boucle avec son prochain saut suivant le protocole de routage final $\mathcal{R}_f$, ceci grâce à un test fait par envoi de messages. Si n est sur une telle boucle avec $\mathcal{R}_f(n)$, alors n reste bloqué jusqu'à la réception d'un message de notification de migration de $\mathcal{R}_f(n)$.

Les différentes simulations ont permis de tirer les conclusions suivantes :

- En termes du nombre de boucles : le retrait ou la panne d'un nœud du réseau génère un nombre de boucles de routage qui dépend beaucoup de la topologie. Nous avons constaté de grandes disparités selon les topologies : pour les topologies aléatoires dont le nombre maximal de liens est 9419, le nombre maximal de boucles avoisine 80, pour les topologies *Rocketfuel* dont le nombre maximal de liens est 1944, le nombre maximal de boucles de routage est de 150, et pour les topologies *ITZ* dont le nombre maximal de liens est 788, le nombre maximal de boucles est de 400.
- En termes de durée de transition : DLF fonctionne aussi bien que les meilleures heuristiques centralisées ACH et SCH-m qui produisent un nombre d'étapes peu dépendant de la taille du réseau et du nombre de destinations. RTH-d, malgré sa simplicité de mise en œuvre, produit le plus d'étapes pour chaque taille de réseau. DLF montre un gain allant jusqu'à 55% par rapport à GBA, et un gain allant jusqu'à 99% par rapport à RTH-d.
- En termes de coût de la transition : toutes les heuristiques fonctionnent avec un nombre limité de messages de contrôle. Néanmoins, une séparation assez nette est identifiée entre les heuristiques centralisées et les heuristiques distribuées :

DLF et RTH-d produisent plus de messages que ACH, SCH-m et GBA. ACH et SCH-m montrent un gain allant jusqu'à 44% par rapport à GBA, et un gain supérieur à 90% par rapport à DLF et RTH-d.

— En termes de temps de calcul centralisé : toutes les heuristiques ont un temps de calcul qui croît avec la taille du réseau. Néanmoins, quel que soit le jeu de données, RTH-d et ACH sont toujours plus rapides que SCH-m, DLF et GBA. RTH-d montre un gain allant jusqu'à 63% par rapport à ACH et un gain allant jusqu'à 99% par rapport à SCH-m, DLF et GBA.

— En termes de temps de calcul décentralisé : obtenu à travers une implémentation utilisant des *threads*, le temps de calcul des heuristiques distribuées DLF et RTH-d croît avec la taille du réseau. Cependant, DLF est toujours plus rapide que RTH-d avec un gain allant jusqu'à 95%. Cela s'explique par le fait qu'avec DLF, les nœuds migrent directement sauf en cas d'attente de notification du prochain saut suivant $\mathcal{R}_f$, tandis qu'avec RTH-d, les nœuds migrent par profondeur.

— En termes de vitesse de correction de la panne : elle correspond en général au nombre d'étapes pour effectuer la transition plus un pour les heuristiques DLF, ACH, SCH-m, et GBA, signifiant dans ce cas que la panne se corrige avec les derniers nœuds qui migrent. La correction de la panne nécessite des étapes en moins (jusqu'à 5) comparativement au nombre d'étapes nécessaires pour la migration pour le cas de RTH-d, signifiant ainsi que la panne est corrigée bien avant la fin de la migration de tous les nœuds. La durée de la migration pour RTH-d est donc artificiellement plus longue.

Chapitre 5

Heuristiques dans un contexte dynamique

Sommaire

Introduction

Les contributions de ce chapitre portent sur la migration d'un protocole de routage initial vers un protocole de routage final dans un contexte mobile et distribué. Nous considérons que les nœuds du réseau sont déployés dans une surface géographique définie *a priori*. Nous présentons dans un premier temps deux types d'applications à savoir : la mobilité d'un unique nœud dans le réseau, et la mobilité d'un groupe de nœuds du réseau. Dans un second temps, nous présentons le protocole DLF-k, une version améliorée de DLF qui traite des boucles de taille inférieure ou égale à k, pour $k \geq 2$. Dans un troisième temps, nous étudions la quantification des boucles de routage générées pour chacune de nos différentes applications, ainsi que l'adaptation de DLF-k dans chacun des cas.

5.1 Applications

Dans les chapitres précédents, nous avons considéré des réseaux statiques où les nœuds pouvaient être retirés ou non, mais ne changeaient pas de position initiale. Cependant, la mobilité est une notion importante dans de nombreuses applications de nos jours avec par exemple les réseaux de véhicules [49, 50], les utilisateurs mobiles qui se connectent à de réseaux sans fil à leur portée [51], ou encore les surveillances ou des livraisons faites par des drones. Nous nous sommes ainsi intéressés à cette caractéristique de réseau, et présentons dans cette partie deux cas de mobilité sur lesquels nous nous sommes concentrés à savoir : la mobilité d'un unique nœud dans le réseau, et la mobilité d'un groupe de nœuds du réseau (le groupe pouvant être le réseau complet).

5.1.1 Mobilitité d'un unique nœud

Nous considérons ici qu'il existe dans le réseau un unique nœud qui change de position entre un instant t_i et un instant t_{i+1}, tandis que tous les autres nœuds du réseau conservent la même position.

Nous avons l'exemple de drone collectant des données sur des capteurs statiques dans de nombreux domaines [52, 53, 54, 55] comme le domaine agricole, le trafic portuaire, la surveillance militaire, des contrôles à accès difficile, etc. Nous supposons que le drone est ici l'unique nœud mobile.

Nous avons également l'exemple d'un policier disposant d'un terminal, qui se déplace dans un espace de parking de voitures, et qui est chargé de donner des contraventions aux utilisateurs ayant dépassé la limite de stationnement autorisée. Sous chaque place de voiture, se trouve un capteur qui récupère l'heure d'arrivée du véhicule.

Nous supposons dans tout le chapitre que les nœuds sont représentés dans un espace $2D$. Un nœud mobile se déplace d'une position à une autre suivant une distance et un angle. Formellement, soit un graphe $G = (V, E)$ qui représente un réseau de n nœuds, $n_i \in V$ un nœud du réseau de coordonnées initiales (x_i, y_i), $dist \in \mathbb{N}^+$ une distance, $\theta \in [0, 2\pi]$ un angle de déplacement. Déplacer le nœud n_i d'une distance $dist$ de sa position initiale (x_i, y_i) vers une nouvelle position (x'_i, y'_i) revient à calculer les nouvelles coordonnées (x'_i, y'_i) de telle sorte que :

$$\begin{cases} x'_i = x_i + dist.\cos\theta \\ y'_i = y_i + dist.\sin\theta \end{cases}$$

Le nœud peut suivre un itinéraire aléatoire, ou un itinéraire défini d'avance pour sa collecte de données. Quel que soit le modèle de mobilité retenu, le protocole de routage ou les chemins peuvent être amenés à changer. En effet, le déplacement d'un nœud mobile peut conduire à de nouveaux chemins vers la destination dû au fait que le voisinage du nœud mobile évolue en fonction des critères de connectivité retenus, et tous les nœuds ne sont pas *a priori* informés instantanément du déplacement. Cela peut alors aboutir à des boucles de routage. Nous illustrons cela avec un exemple détaillé.

Nœud	**Abscisse** (x)	**Ordonnée** (y)
1	97	95
2	68	77
3	49	82
4	86	86
5	74	97
6	58	68
7	73	83
8	98	91
9	55	91
10	36	96

TABLE 5.1 – Tableau des nœuds avec leurs coordonnées pour le réseau de la figure 5.1(a)

Considérons l'exemple d'un réseau de 10 nœuds (figure 5.1(a)) déployés dans une zone $100m \times 100m$ où chaque coordonnée de nœud est comprise dans l'intervalle $[0, 100]$ tel que détaillé dans le tableau 5.1. Soit u le nœud mobile qui se déplace ici de manière aléatoire. Nous supposons qu'à chaque nouvelle position, u se connecte aux différents nœuds situés à sa portée (ici fixée à $20m$). Soit u le nœud mobile et destination des paquets dans le réseau, qui part de la position initiale $u_i = (31, 92)$, à la position finale $u_f = (71, 99)$ située $50m$ plus loin tel qu'illustré dans la figure 5.1(b). Une première capture du réseau est faite lorsque u s'est déplacé de $5m$ vers $u_1 = (35, 89)$. Puis une deuxième capture après un déplacement de $15m$ par rapport à la position initiale vers $u_2 = (48, 69)$, une troisième autre capture après $30m$ de déplacement par rapport à la position initiale vers $u_3 = (60, 54)$, et enfin une dernière à la position finale u_f.

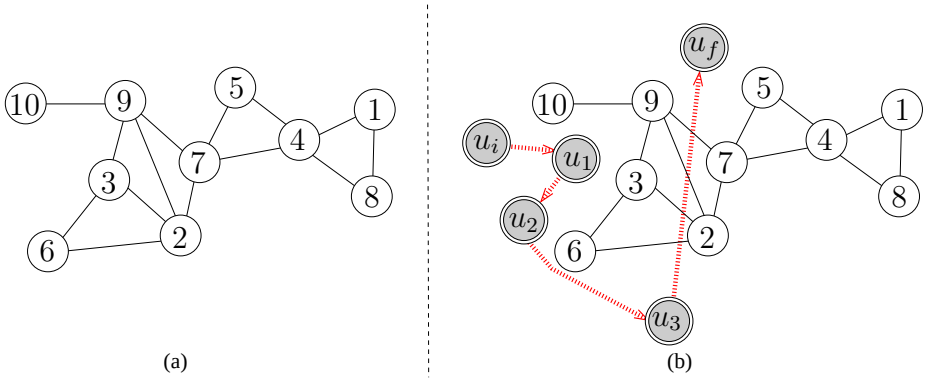

FIGURE 5.1 – Exemple d'un réseau de 10 nœuds avec mobilité d'un utilisateur. (a) Réseau initial, (b) réseau avec différentes positions du nœud mobile

A la position initiale u_i, u avec pour portée $20m$ est connecté au nœud 10 comme le montre la figure 5.2 (a). La figure 5.2 (b) quant à elle montre le protocole de routage

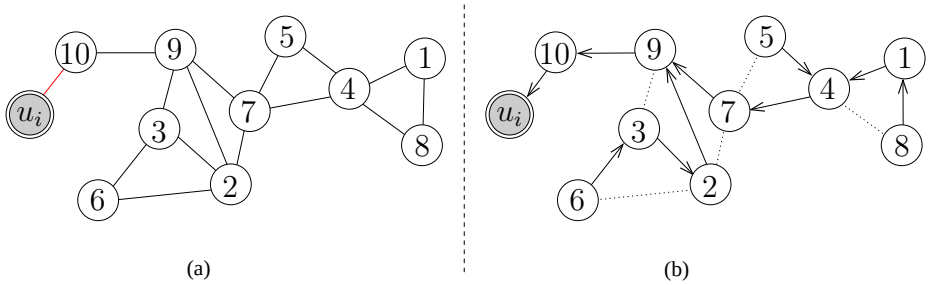

FIGURE 5.2 – Exemple d'un utilisateur mobile en position initiale u_i. (a) Réseau avec la position initiale, (b) routage initial.

initial avant tout déplacement de u. Ce protocole est considéré comme le protocole initial $\mathcal{R}_i$ pour tous les déplacements vers les positions u_1, u_2, u_3, et u_f.

Ensuite, l'utilisateur u se déplace de $5m$ vers la position u_1. A cette position, il se connecte aux nœuds 3, 9 et 10 qui deviennent ses voisins comme le montre la figure 5.3(a). Ce déplacement est susceptible de générer la boucle de routage $(2, 3, 2)$ car comme le montre la figure 5.3(b), les nœuds 2, 3, et 9 se voient changer de prochains sauts vers u_1 passant respectivement de 9, 2, et 10 à 3, u_1, et u_1.

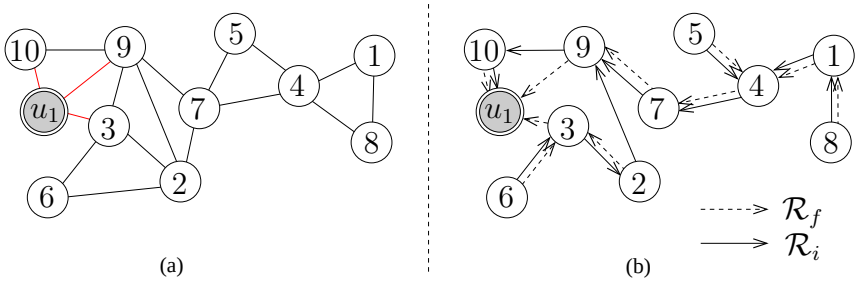

FIGURE 5.3 – Exemple du déplacement de l'utilisateur mobile de $5m$ vers la position u_1, avec génération de la boucle $(2, 3, 2)$.

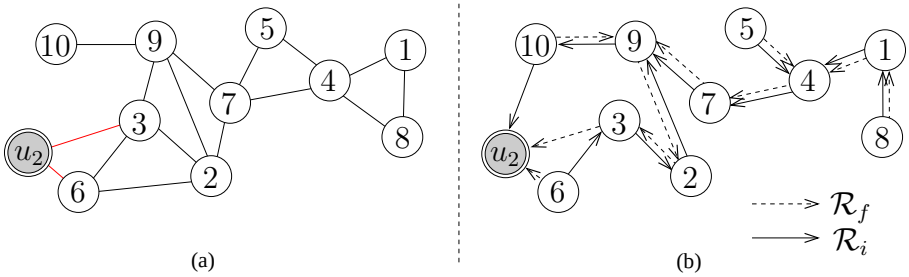

FIGURE 5.4 – Exemple du déplacement de l'utilisateur mobile de $15m$ vers la position u_2, avec génération des boucles $(2, 3, 2)$, $(2, 9, 2)$, et $(9, 10, 9)$.

De même, le déplacement de $15m$ de u vers la position u_2 le conduit à avoir pour voisins les nœuds 3 et 6 (figure 5.4(a)). A cette position, avec le protocole initial $\mathcal{R}_i$ de la figure 5.2, les boucles $(2, 3, 2)$, $(2, 9, 2)$, et $(9, 10, 9)$ sont susceptibles d'être générées comme le montre la figure 5.4(b) car, les nœuds 2, 3, 9 et 10 changent de prochain saut pour arriver à u_2 avec respectivement 3, u_2, 2, et 9 au lieu de 9, 2, 10, et u_i à la position initiale de u.

De même, le déplacement de $30m$ de u vers u_3 l'amène à n'avoir que le nœud 6 comme voisin (figure 5.5(a)). Ce déplacement est susceptible de générer les boucles

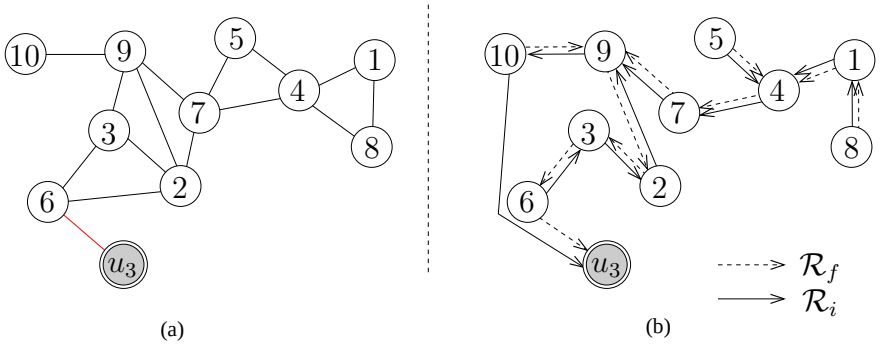

FIGURE 5.5 – Exemple du déplacement de l'utilisateur mobile de $30m$ vers la position u_3, avec génération des boucles $(2, 3, 2)$, $(2, 9, 2)$, $(9, 10, 9)$, et $(3, 6, 3)$.

$(2, 3, 2)$, $(2, 9, 2)$, $(9, 10, 9)$, $(3, 6, 3)$ comme le montre la figure 5.5(b) car comme dans les cas précédents, certains nœuds changent de prochain saut. En effet, les nœuds 2, 3, 6, 9 et 10 passent respectivement des nœuds 9, 2, 3, 10 et u_i comme prochains sauts aux nœuds 3, 6, u_3, 2 et 9.

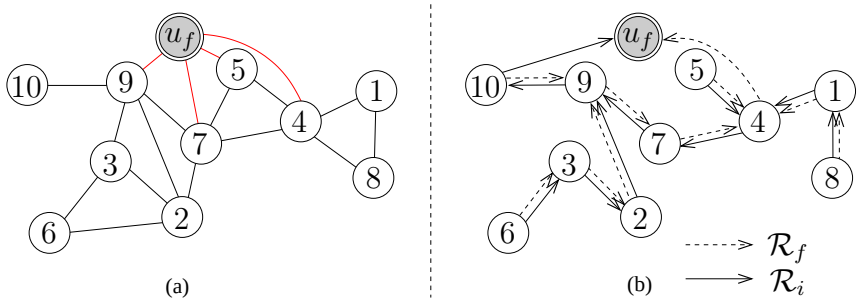

FIGURE 5.6 – Exemple du déplacement de l'utilisateur mobile de $50m$ vers la position u_f, avec génération des boucles $(7, 9, 7)$, $(9, 10, 9)$, et $(4, 7, 4)$.

De même, le déplacement de $50m$ de u vers u_f le conduit à avoir comme nouveaux voisins les nœuds $\{4, 5, 7, 9\}$ tel que l'illustre la figure 5.6. Les boucles $(7, 9, 7)$, $(9, 10, 9)$, $(4, 7, 4)$ sont susceptibles d'apparaître comme le montre la figure 5.6(b), les nœuds 4, 7, 9 et 10 passent respectivement de 7, 9, 10, et u_i à u_f, 4, 7, et 9 comme prochains sauts.

5.1.2 Mobilité d'un groupe de nœuds

Nous considérons ici qu'une partie voire la totalité du réseau est mobile. Nous avons à ce titre l'exemple des *MANET (Mobile Ad hoc Network)* [56] qui sont des ensembles de nœuds mobiles dotés de capacité de communication sans fil sans contrôle d'un réseau central. Dans ces réseaux, tous les nœuds sont mobiles et peuvent être connectés dynamiquement de manière arbitraire. Comme la portée de la transmission sans fil de chaque hôte est limitée, un hôte doit demander l'aide de ses hôtes voisins pour transférer les paquets vers une destination située hors de sa portée de transmission. Ainsi, tous les nœuds de ces réseaux se comportent comme des routeurs et participent à la découverte des routes et à la maintenance des itinéraires vers les autres nœuds du réseau [57].

Nous avons le cas particulier des *VANET (Vehicular Ad-hoc Network)* [58] encore appelés réseaux véhiculaires, qui sont des réseaux de communication sans fil autonomes et auto-organisés, dans lesquels les nœuds sont très mobiles et coopèrent en tant que

serveurs et/ou clients pour l'échange et le partage d'information [59]. De nombreuses applications ont été proposées pour les réseaux véhiculaires [60, 61], et peuvent être classifiées ainsi [62] :

— Applications de sécurité [63, 64, 65, 66] dont le but est d'améliorer la sécurité des passagers sur les routes en avisant de toute situation dangereuse : les conducteurs ont ici une connaissance de l'état de la route et des véhicules voisins à travers l'échange périodique ou non de messages informatifs. On a ainsi les services d'avertissement de collisions ou d'accident, d'avertissement sur les conditions de la route, etc.

— Applications de gestion de trafic [67, 68] : elles visent à réduire les embouteillages, et les risques d'accident, ou améliorer les conditions de circulation. Nous avons les services de surveillance du trafic, d'ordonnancement des feux de signalisation, etc.

— Applications de confort [69, 70, 71] ou de divertissement dont l'objectif est de rendre les voyages plus agréables en permettant aux passagers de communiquer soit avec d'autres véhicules, soit avec des stations fixes comme les hôtes internet. Des exemples d'applications sont la gestion de parkings ou les jeux/discussions distribués.

L'une des caractéristiques majeures des *VANET* est la grande dynamicité de la topologie due au mouvement des véhicules. Cette mobilité est susceptible de générer des boucles de routage. En effet, pour une destination donnée, certains nœuds peuvent être amenés à emprunter de nouvelles routes, et durant le temps de convergence, d'autres nœuds peuvent continuer à utiliser des anciennes routes qui ne sont plus valides.

Considérons l'exemple précédent (figure 5.1(a)) d'un réseau de 10 nœuds. Supposons qu'une partie des nœuds se déplace de manière aléatoire. Soient 3, 5, 6, 7, 9, et 10 ces nœuds mobiles. La figure 5.7(a) montre le réseau initial avant tout déplacement de nœuds, et la figure 5.7(b) après déplacement des nœuds.

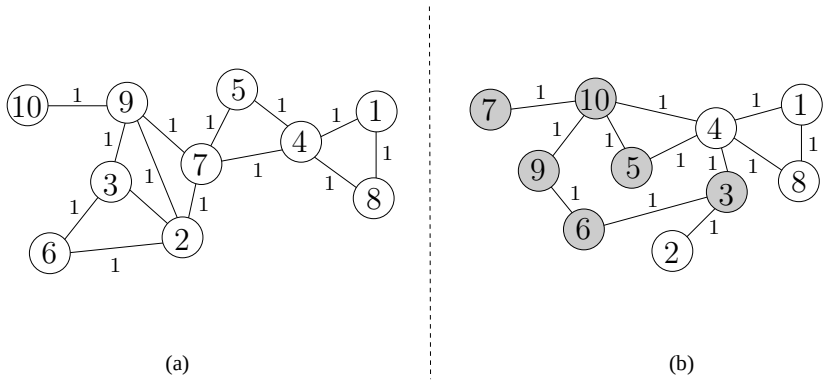

FIGURE 5.7 – Exemple du déplacement d'un groupe de nœuds : (a) réseau original, (b) réseau après déplacement de manière aléatoire des nœuds 3, 5, 6, 7, 9, et 10.

Soit 4 le nœud destination. La figure 5.8 montre l'union des deux routages avant et après déplacement des nœuds. Cette figure permet de voir que durant la convergence, des boucles de routage peuvent apparaître à savoir : $(2, 3, 2)$, $(9, 10, 9)$, $(7, 10, 9, 7)$.

Nous constatons ainsi que la mobilité d'un ou de plusieurs nœuds dans un réseau peut générer des boucles de routage, car les routages $\mathcal{R}_i$ et $\mathcal{R}_f$ peuvent ne pas être valides à chaque instant.

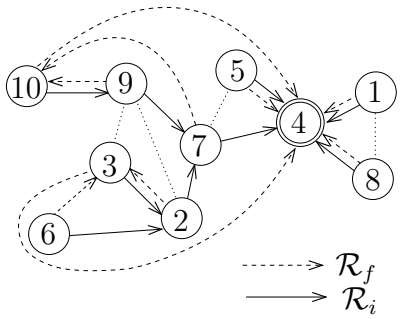

FIGURE 5.8 – Union des routages avant et après déplacement de manière aléatoire des nœuds 3, 5, 6, 7, 9, et 10, et génération des boucles $(2, 3, 2)$, $(9, 10, 9)$, $(7, 10, 9, 7)$.

5.2 Protocole DLF-k : *Distributed Loop-Free heuristic for loop of size k*

Dans cette section, nous présentons le protocole DLF-k conçu pour traiter en distribué non seulement les boucles de taille 2 comme c'est le cas pour DLF [45], mais également toutes les boucles de taille inférieure ou égale à k, avec $k \geq 2$. Nous discutons dans un premier temps de l'adaptation de DLF aux boucles de taille 3. Puis dans un second temps, nous détaillons DLF-k.

5.2.1 Discussion de l'adaptation de DLF au traitement des boucles de taille 3

Dans cette partie, nous discutons de l'adaptation de DLF pour le traitement de boucle de taille $k = 3$.

DLF a été conçu pour le retrait des nœuds, ce qui ne génère que des boucles de taille 2. Dû à la mobilité et au changement frequent des positions des noeuds, des boucles de plus grande taille peuvent apparaître. Nous étudions tout d'abord pour tous les cas possibles de boucles de taille 3, la possibilité qu'elles soient résolues par DLF dans sa version actuelle (appelée DLF-2).

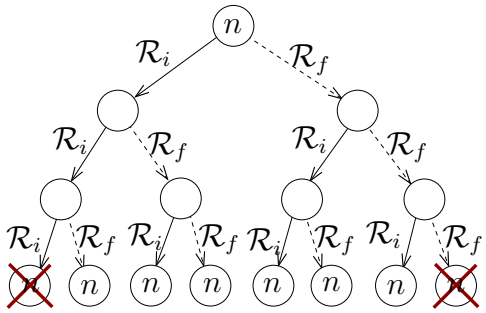

FIGURE 5.9 – Différents cas possibles de boucles de routage de taille 3.

La figure 5.9 montre les 6 cas possibles de boucles de taille de 3 à savoir :
$\mathcal{R}_i(\mathcal{R}_i(\mathcal{R}_i(n)))=n$, $\mathcal{R}_i(\mathcal{R}_i(\mathcal{R}_f(n)))=n$, $\mathcal{R}_i(\mathcal{R}_f(\mathcal{R}_i(n)))=n$, $\mathcal{R}_i(\mathcal{R}_f(\mathcal{R}_f(n)))=n$,
$\mathcal{R}_f(\mathcal{R}_i(\mathcal{R}_i(n)))=n$, $\mathcal{R}_f(\mathcal{R}_i(\mathcal{R}_f(n)))=n$, $\mathcal{R}_f(\mathcal{R}_f(\mathcal{R}_i(n)))=n$, (8) $\mathcal{R}_f(\mathcal{R}_f(\mathcal{R}_f(n)))=n$.

Étant donné qu'on fait l'hypothèse que chaque routage pris indépendamment est sans boucle, les cas 1 et 8 ne peuvent survenir. Cela réduit le nombre de cas à étudier à 6.

— Cas $\mathcal{R}_i(\mathcal{R}_i(\mathcal{R}_f(n)))=n$: ce cas est présenté dans la figure 5.10 (Cas 1). Dans ce cas, pour que cette boucle soit évitée, il faut que n continue à router suivant $\mathcal{R}_i$ tant que $n' = \mathcal{R}_f(n)$ n'a pas migré. Cependant, DLF autorise le nœud n à migrer car $\mathcal{R}_i(\mathcal{R}_f(n)) \neq n$. Par conséquent, DLF ne corrige pas ce cas de boucle dans sa version actuelle.
— Cas $\mathcal{R}_i(\mathcal{R}_f(\mathcal{R}_i(n)))=n$: ce cas est présenté dans la figure 5.10 (Cas 2). Dans ce cas, DLF peut autoriser ou interdire la migration du nœud n selon $\mathcal{R}_f(n)$. Cependant, étant en distribué, tous les nœuds exécutent simultanément DLF. Lorsque $\mathcal{R}_i(n)$ fait le test $\mathcal{R}_i(\mathcal{R}_f(n)) = n$ de DLF, ce dernier est faux (figure 5.10 (Cas 2)) et DLF l'autorise ainsi à migrer. Ce qui peut aboutir à la boucle $(n, \mathcal{R}_i(n), \mathcal{R}_f(\mathcal{R}_i(n)), n)$. DLF par conséquent ne corrige pas ce cas.
— Cas $\mathcal{R}_f(\mathcal{R}_i(\mathcal{R}_i(n)))=n$: ce cas est présenté dans la figure 5.10 (Cas 3). Dans ce cas également, la boucle $(n, \mathcal{R}_i(n), \mathcal{R}_i(\mathcal{R}_i(n)), n)$ ne peut être évitée par DLF que s'il empêche $\mathcal{R}_i(\mathcal{R}_i(n))$ de migrer. Cependant, si nous notons n'' le nœud $\mathcal{R}_i(\mathcal{R}_i(n))$, nous avons $\mathcal{R}_i(\mathcal{R}_f(n'')) \neq n''$. DLF autorise ainsi n'' à migrer. Par conséquent, DLF ne permet pas d'éviter cette boucle.
— Cas $\mathcal{R}_i(\mathcal{R}_f(\mathcal{R}_f(n)))=n$: ce cas est présenté dans la figure 5.11 (Cas 4 (a)). Dans ce cas, les nœuds n et $\mathcal{R}_f(n)$ routent suivant $\mathcal{R}_f$ (figure 5.11 (Cas 4 (a)). Si le nœud $\mathcal{R}_f(n)$ a pour prochain saut suivant $\mathcal{R}_i$ le nœud n (figure 5.11 (Cas 4 (b)) où $\mathcal{R}_i(\mathcal{R}_f(n)) = n$, alors, DLF n'autorise pas n à migrer et évite ainsi la boucle $(n, \mathcal{R}_f(n), \mathcal{R}_f(\mathcal{R}_f(n)), n)$. Sinon, DLF autorise le nœud n à migrer et la boucle n'est par conséquent pas évitée.
— Cas $\mathcal{R}_f(\mathcal{R}_i(\mathcal{R}_f(n)))=n$: ce cas est présenté dans la figure 5.11 (Cas 5 (a)). Dans ce cas, comme le montre la figure 5.11 (Cas 5 (a)), les nœuds n et $\mathcal{R}_i(\mathcal{R}_f(n))$ routent suivant $\mathcal{R}_f$. Si le nœud n a pour prochain saut suivant $\mathcal{R}_i$ le nœud $n'' = \mathcal{R}_i(\mathcal{R}_f(n))$ comme le montre la figure 5.11 (Cas 5 (b)), alors ce dernier ne sera pas autorisé à migrer avant n par DLF car $\mathcal{R}_i(\mathcal{R}_f(n'')) = n''$. Par conséquent la boucle $(n, \mathcal{R}_f(n), \mathcal{R}_i(\mathcal{R}_f(n)), n)$, dans ce cas, pourra être évitée par DLF. Sinon, DLF autorise le nœud n à migrer et la boucle n'est par conséquent pas évitée.
— Cas $\mathcal{R}_f(\mathcal{R}_f(\mathcal{R}_i(n)))=n$: ce cas est présenté dans la figure 5.11 (Cas 6 (a)). Dans ce cas, ce sont les nœuds $\mathcal{R}_i(n)$ et $\mathcal{R}_f(\mathcal{R}_i(n))$ qui routent suivant $\mathcal{R}_f$, comme le montre la figure 5.11 (Cas 6 (a)). Si le nœud $n'' = \mathcal{R}_f(\mathcal{R}_i(n))$ a pour prochain saut suivant $\mathcal{R}_i$ le nœud $n' = \mathcal{R}_i(n)$ comme le montre la figure 5.11 (Cas 6 (b)), alors nous avons $\mathcal{R}_i(\mathcal{R}_f(n')) = n'$. DLF empêche ainsi n' de migrer avant n''. Ce qui permet à DLF d'éviter aussi la boucle $(n, \mathcal{R}_i(n), \mathcal{R}_f(\mathcal{R}_i(n)), n)$. Sinon, DLF autorise le nœud n à migrer et la boucle n'est par conséquent pas évitée.

Cas non corrigés : $\begin{cases} \mathcal{R}_i(\mathcal{R}_i(\mathcal{R}_f(n))) = n \\ \mathcal{R}_i(\mathcal{R}_f(\mathcal{R}_i(n))) = n \\ \mathcal{R}_f(\mathcal{R}_i(\mathcal{R}_i(n))) = n \end{cases}$

Cas avec certaines configurations permettant la correction : $\begin{cases} \mathcal{R}_i(\mathcal{R}_f(\mathcal{R}_f(n))) = n \\ \mathcal{R}_f(\mathcal{R}_i(\mathcal{R}_f(n))) = n \\ \mathcal{R}_f(\mathcal{R}_f(\mathcal{R}_i(n))) = n \end{cases}$

Nous constatons que pour les boucles de taille 3 qui n'ont qu'un seul nœud qui route suivant $\mathcal{R}_f$, DLF ne peut pas éviter la boucle. Par contre, tous les cas ayant deux nœuds routant suivant $\mathcal{R}_f$ admettent une possibilité d'évitement par DLF. En effet, supposons n, n' et n'' les 3 nœuds impliqués dans une boucle de taille 3. Si n et n' sont par exemple les deux nœuds de la boucle qui routent suivant $\mathcal{R}_f$ et qu'en plus, on a $\mathcal{R}_f(n) = n'$ et $\mathcal{R}_i(n') = n$, alors DLF en corrigeant la boucle de taille 2 (n, n', n), corrige également la boucle (n, n', n'', n) car, DLF empêche n de migrer avant n'. Pour

tous les autres cas, DLF n'est pas en mesure d'éviter la boucle.

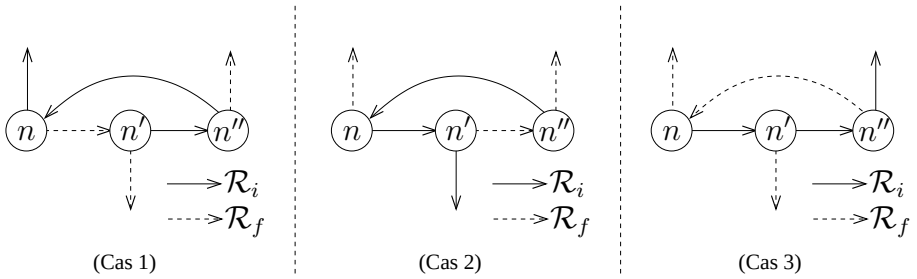

FIGURE 5.10 – Boucles de taille 3 non corrigées par DLF.

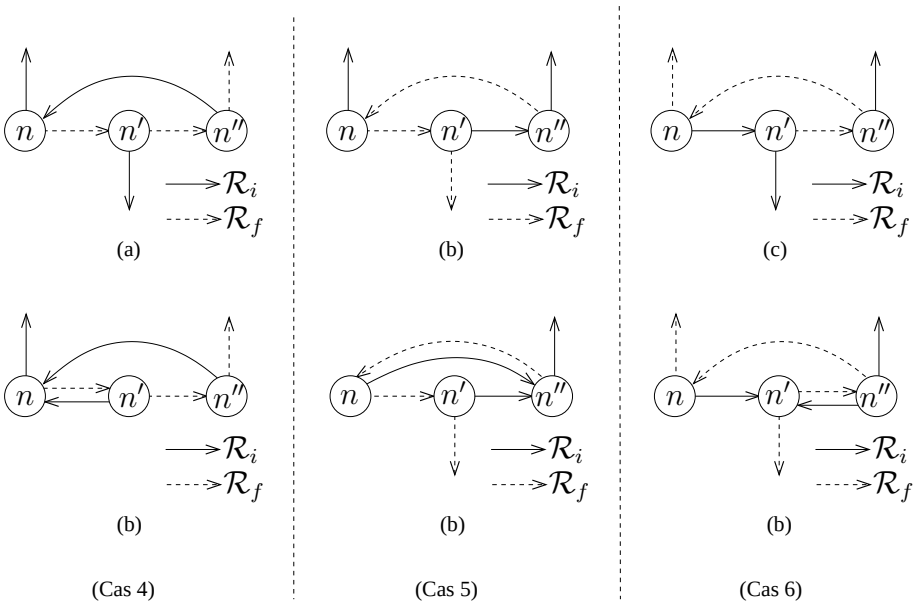

FIGURE 5.11 – Boucles de taille 3 corrigées par DLF dans certains cas.

5.2.2 DLF-k

Nous proposons une version modifiée de DLF qui prend en compte toutes les tailles de boucle. Elle repose sur l'idée qu'un nœud n n'est autorisé à migrer que s'il n'est pas inclus dans une boucle avec $\mathcal{R}_f(n)$. Dans le cas contraire, n doit attendre la notification de $\mathcal{R}_f(n)$ pour pouvoir migrer. Lorsqu'un nœud migre, il informe celui dont il est le prochain saut sur $\mathcal{R}_f$. Pour ce faire, il utilise l'information selon laquelle ce nœud est l'un de ses voisins, et envoie un message à chacun d'eux. Ainsi le nœud bloqué pourra se reconnaître et migrer à son tour. Ce processus est décrit par l'algorithme 8.

L'algorithme 9 décrit, quant à lui, le processus d'identification d'appartenance à une même boucle entre un nœud et son prochain saut suivant $\mathcal{R}_f$. Chacun des nœuds ne connaissant *a priori* que ses deux prochains sauts (suivant $\mathcal{R}_i$ et suivant $\mathcal{R}_f$), la vérification d'appartenance à une même boucle de routage se fait par échange de messages (algorithme 9). Pour ce faire, nous construisons de manière récursive l'arbre binaire ayant pour racine le nœud $\mathcal{R}_f(n)$ associé au nœud n. Dans cet arbre, chaque nœud x a deux fils : l'un selon $\mathcal{R}_i$, l'autre selon $\mathcal{R}_f$. Nous définissons un paramètre *ancetres* qui conserve le chemin partant de la racine de l'arbre $\mathcal{R}_f(n)$ au nœud courant x, et qui

sert à stopper la recherche sur une branche de cet arbre. Le nœud n est propagé dans chaque échange de message. Les cas de figure d'arrêt sont les suivants :

- $n = d$: le nœud propagé est la destination. Il s'agit du cas exception dans lequel la destination est autorisée à migrer directement.
- $x = n$: le nœud courant x est égal au nœud n propagé. Dans ce cas, on a la confirmation d'appartenance de n et $\mathcal{R}_f(n)$ à la même boucle.
- $x \in ancetres$ ou $x = d$: le nœud courant x a déjà été visité, ou se trouve être la destination. Dans ce cas, pas de boucle possible entre n et $\mathcal{R}_f(n)$ sur cette branche.
- $|ancetres| = k$: la taille maximale k des boucles a été atteinte, on conclut à une non existence de boucle de taille inférieure ou égale à k entre n et $\mathcal{R}_f(n)$ sur cette branche.

Dans le cas où aucun de ces cas de figure ne se présente, nous ne pouvons conclure et continuons ainsi la recherche avec les fils du nœud courant. Si les deux fils retournent $faux$ (lignes 11 et 12 de l'algorithme 9), alors le nœud courant x retourne également $faux$.

Algorithme 8 : Distributed loop-free heuristic for loop of size of k (DLF-k).

Données : n est le nœud courant, $\mathcal{R}_i$ et $\mathcal{R}_f$ sont connus *a priori* par tous les nœuds, d la destination.

```
1 P ← ∅
2 si inclus-dans-boucle(n,R_f(n), P, d, k) alors
3 |  n attend R_f(n)
4 fin
5 n migre à R_f
6 n informe ses voisins
```

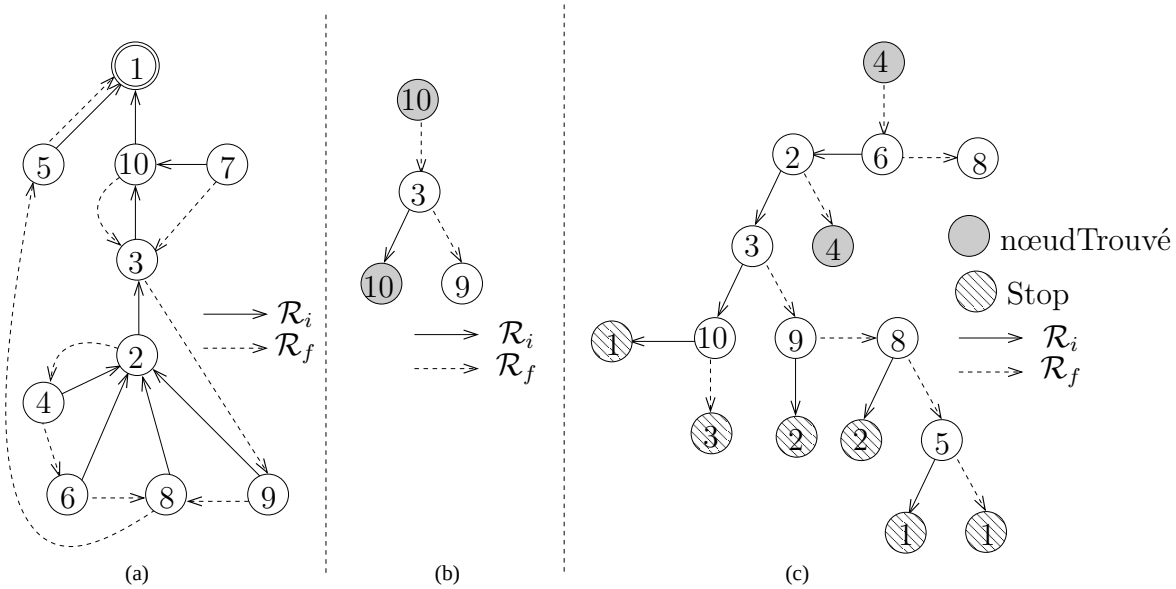

FIGURE 5.12 – Exemple pour DLF-k, avec $k = \infty$. (a) Union des routages $\mathcal{R}_i$ et $\mathcal{R}_f$ d'un réseau de 10 nœuds vers la destination 1 générant les boucles : $(2, 3, 9, 2)$, $(2, 3, 9, 8, 2)$, $(2, 4, 2)$, $(2, 4, 6, 2)$, $(2, 4, 6, 8, 2)$, $(3, 10, 3)$. (b) Recherche de l'appartenance de 10 et $\mathcal{R}_f(10)$ à une boucle avec $k = \infty$. (c) Recherche de l'appartenance de 4 et $\mathcal{R}_f(4)$ à une boucle avec $k = \infty$.

Algorithme 9 : inclus-dans-boucle.

Données : n le nœud propagé, x le nœud courant, *ancetres* le chemin de $\mathcal{R}_f(n)$ à x dans $\mathcal{R}_i \cup \mathcal{R}_f$, d la destination, k la taille maximale de boucle
Résultat : un booléen

```
si (n = d) alors
    retourner faux
fin
si (x = n) alors
    retourner vrai
fin
si ((x = d) ou (x ∈ ancetres) ou (|ancetres| = k)) alors
    retourner faux
fin
ajouter x à ancetres
si (inclus-dans-boucle(n, R_i(x), ancetres, d, k)=faux) alors
    si (inclus-dans-boucle(n, R_f(x), ancetres, d, k)=faux) alors
        supprimer x de ancetres
        retourner faux
    fin
fin
retourner vrai ;
```

Considérons l'exemple de la figure 5.12, d'un réseau de 10 nœuds dont l'union des routages vers la destination 1 génère les boucles $(2, 3, 9, 2)$, $(2, 3, 9, 8, 2)$, $(2, 4, 2)$, $(2, 4, 6, 2)$, $(2, 4, 6, 8, 2)$, $(3, 10, 3)$ (figure 5.12(a)). Nous rappelons que pour pouvoir construire la transition finale, chaque nœud applique l'algorithme 8 en parallèle. Pour les deux exemples détaillés de la figure 5.12, nous précisons que l'arrêt sur une branche est marqué par la couleur rouge sur le nœud courant, et la couleur grise en cas d'identification du nœud propagé dans l'arbre.

La figure 5.12(b) présente le cas du nœud 10. La construction de l'arbre débute sur $\mathcal{R}_f(10) = 3$ avec le paramètre *ancetres* vide. Le nœud 3 n'est ni destination, ni le nœud propagé, ni inclus dans *ancetres*, il ne respecte aucune condition d'arrêt, et est donc ajouté à la variable *ancetres*. Puis un appel récursif est fait sur son fils gauche, qui est le nœud $\mathcal{R}_i(3) = 10$. Comme le montre la figure, la condition que le nœud courant égal au nœud propagé est vérifiée. La recherche sur cette branche est ainsi stoppée avec pour valeur de retour *vrai* car le nœud propagé est retrouvé. La valeur de retour étant *vrai*, le nœud 10 reste bloqué en attente de notification du nœud 3, ce qui corrige la boucle.

De même, la figure 5.12(c) présente les détails de construction de l'arbre pour le nœud 4. La construction débute avec $\mathcal{R}_f(4) = 6$. Aucune condition d'arrêt n'étant vérifiée, la construction et l'exploration se poursuivent avec ses deux fils 2 et 8. A partir du nœud 2, la recherche est stoppée sur les branches $6-2-3-10-1$, $6-2-3-9-8-5$ car le nœud courant est la destination qui est 1. La recherche est également stoppée pour les branches $6-2-3-10-3$, $6-2-3-9-2$, $6-2-3-9-8-2$ car le nœud courant est inclus dans le paramètre *ancetres*. La valeur de retour du sous-arbre ayant pour racine $\mathcal{R}_i(2)$ étant $faux$, 2 poursuit ainsi sa recherche à $\mathcal{R}_f(2)$. La recherche est stoppée à ce niveau car $\mathcal{R}_f(2) = 4$ est le nœud propagé, d'où la valeur de retour *vrai*. Par conséquent, 4 reste bloqué en attente de notification de 6.

Pour les nœuds restants non illustrés ici par un schéma, nous avons :

— nœud 1 : C'est le nœud destination. Dans l'appel de l'algorithme 9, le test est stoppé à la ligne 1. 1 migre ainsi dans la première étape.
— nœud 2 : Le nœud appelle l'algorithme 9 avec le nœud 4, son prochain saut suivant $\mathcal{R}_f$. L'arbre est ainsi construit et exploré à partir de 4. Il n'est ni égal à la destination, ni égal au nœud propagé 2, la recherche se poursuit avec ses fils 2 (selon $\mathcal{R}_i$)) et 6 (selon $\mathcal{R}_f$). L'appel récursif sur $\mathcal{R}_i(4)$ est stoppé car $\mathcal{R}_i(4) = 2$. Par conséquent, *vrai* est renvoyé et 2 est mis en attente de la notification de 4.
— nœud 3 : l'exploration débute sur le nœud $\mathcal{R}_f(3) = 9$. Le nœud 9 n'est ni égal à la destination 1, ni égal au nœud propagé sur cette branche qui est 3. La recherche se poursuit ainsi avec ses deux fils 2 et 8. La recherche sur la branche avec le fils 2 retourne *vrai* car dans la variable *ancetres* qui vaut ici $[9, 2, 3]$, apparaît le nœud propagé 3. Cela traduit l'appartenance des nœuds 3 et 9 à une même boucle. Par conséquent 3 est bloqué en attente de 9.
— nœud 5 : dans ce cas, $\mathcal{R}_f(5) = 1$ le nœud destination. La recherche est stoppée avec pour valeur de retour $faux$. Le nœud 5 est ainsi autorisé à migrer.
— nœud 6 (resp. 9) : est bloqué par le nœud $\mathcal{R}_f(6) = 8$ car à partir de lui, la branche $[8, 2, 4, 6]$ est construite (resp. la branche $[8, 2, 3, 9]$).
— nœud 7 : $\mathcal{R}_f(7) = 3$. Pour le cas du nœud 7, toutes les branches à partir de 3 sont explorées et le message $faux$ est remonté. N'appartenant pas à la même boucle que 3, le nœud 7 est autorisé à migrer dans la première étape.
— nœud 8 : l'arbre a pour racine 5 qui n'est ni destination, ni nœud propagé. La recherche se poursuit avec ses deux fils qui ont tous pour valeur 1. 1 étant la destination, la recherche est stoppée avec un $faux$. Ainsi le nœud 8 est autorisé à migrer.

La première étape est donc : $\{1, 5, 7, 8\}$. Le nœud 8 ayant migré, débloque les nœuds 6 et 9 qui forment ainsi la deuxième étape $\{6, 9\}$. Les nœuds 6 et 9 débloquent à leur tour respectivement les nœuds 4 et 3. La troisième étape est donc $\{3, 4\}$. Les nœuds 3 et 4 débloquent à leur tour respectivement les nœuds 10 et 2. La transition valide finale est ainsi : $Tr = (\{1, 5, 7, 8\}, \{6, 9\}, \{3, 4\}, \{2, 10\})$.

5.2.3 Complexité

Pour un réseau de n nœuds et d destinations, DLF-k a une complexité en temps de l'ordre de $\mathcal{O}(k \times 2^k)$.

5.3 Resultats

Dans cette partie, nous étudions les performances de DLF-k à travers différents scénarii.

5.3.1 Métriques de performance

Pour évaluer les performances de DLF-k, nous utilisons les métriques suivantes.

La quantification des boucles

La quantification des boucles se fait à travers le nombre moyen de boucles pouvant être générées durant le temps de convergence des nœuds.

Le pourcentage des boucles traitées

Le pourcentage des boucles corrigées par DLF-k pour un certain $k \geq 2$ donné est la métrique qui donne le nombre de toutes les boucles de taille inférieure à k divisée par le nombre total de boucles générées. Considérons l'exemple de la figure 5.8. Les boucles susceptibles d'être générées sont : $(2, 3, 2)$, $(9, 10, 9)$, $(7, 10, 9, 7)$. Ainsi DLF-2 traitera $2/3 = 67\%$ des boucles soit $(2, 3, 2)$, $(9, 10, 9)$, et DLF-3 traitera $3/3 = 100\%$ des boucles soient $(2, 3, 2)$, $(9, 10, 9)$, et $(7, 10, 9, 7)$.

La durée de la transition en nombre d'étapes

La durée de la transition est le nombre d'étapes nécessaires pour effectuer la transition. Etant donné que nous sommes dans un environnement distribué, tous les nœuds migrent directement, sauf ceux en attente de notification de leur prochain saut.

Le coût de la transition en nombre de messages de contrôle

Le coût de la transition est le nombre de messages de contrôle nécessaires pour effectuer la migration. Soient $n, d, k, l \in \mathbb{N}^+$ respectivement la taille du réseau, le nombre de destinations traitées, la taille maximale des boucles traitées, et l le nombre de boucles identifiées. DLF-k nécessite un nombre de messages de contrôle inférieur ou égal à $n \times d \times (2^k - 2) + l$, car chaque nœud vérifie pour chaque destination s'il appartient à une même boucle que son prochain saut suivant le protocole final à travers l'algorithme 9, et reste en attente de notification en cas d'appartenance à une même boucle.

Le temps de calcul distribué en secondes

Le temps de calcul distribué correspond au temps moyen d'exécution lorsque les simulations sont faites à l'aide de *threads* (cf annexe A.10).

5.3.2 Paramètres et environnement de simulation

Les simulations ont été faites sur des graphes de taille 100 déployés uniformément de manière aléatoire dans un espace de dimensions $100m \times 100m$ avec une portée de $20m$. Les distances de déplacement considérées sont $1m$, $5m$, $10m$, $15m$, $20m$, $30m$, $50m$, et $70m$. Le déplacement d'un nœud de coordonnées (x, y) se fait suivant le protocole aléatoire décrit dans la section 5.1.1 à savoir : un angle de déplacement θ est choisi arbitrairement entre 0 et 2π. Puis les nouvelles coordonnées (x', y') sont calculées en fonction de la distance de déplacement $dist$ et de θ, avec $x' = x + dist.\cos\theta$ et $y' = y + dist.\sin\theta$.

Deux catégories de jeux de données ont été générées pour la simulation :

— la première : un nœud est considéré comme destination, et est l'unique nœud mobile du réseau. Il se connecte à tous les nœuds accessibles à sa portée. Le protocole de routage initial $\mathcal{R}_i$ est généré en calculant le chemin le plus court de chaque nœud vers la destination avant mobilité, tandis que le protocole de routage final $\mathcal{R}_f$ est généré en calculant le chemin le plus court de chaque nœud vers la destination après déplacement des nœuds. Trois scénarii de génération de poids de liens ont été retenus. Le scénario 1 consiste à affecter aux liens des poids aléatoires compris entre 1 et 100 *(random)*. Dans le scénario 2, le poids

d'un lien correspond à la distance euclidienne qui sépare les nœuds extrémités de ce lien *(euclidian)*. Dans le scénario 3, tous les liens ont pour poids 1 *(hopcount)*.

— la deuxième : pour chaque graphe, 4 pourcentages de nœuds sont sélectionnés pour être ceux qui se déplacent : 25%, 50%, 75% et 100%. Les deux scénarii retenus pour la génération des poids des liens sont le scénario 1 *(random)* et le scénario 3 *(hopcount)*.

Les simulations sont répétées 100 fois, et l'intervalle de confiance pour les courbes est de 95%. Nous avons testé DLF-2, DLF-3 et DLF-4.

5.3.3 Résultats de DLF-k sur la mobilité d'un nœud

Les résultats décrits dans cette partie portent sur la mobilité d'un unique nœud.

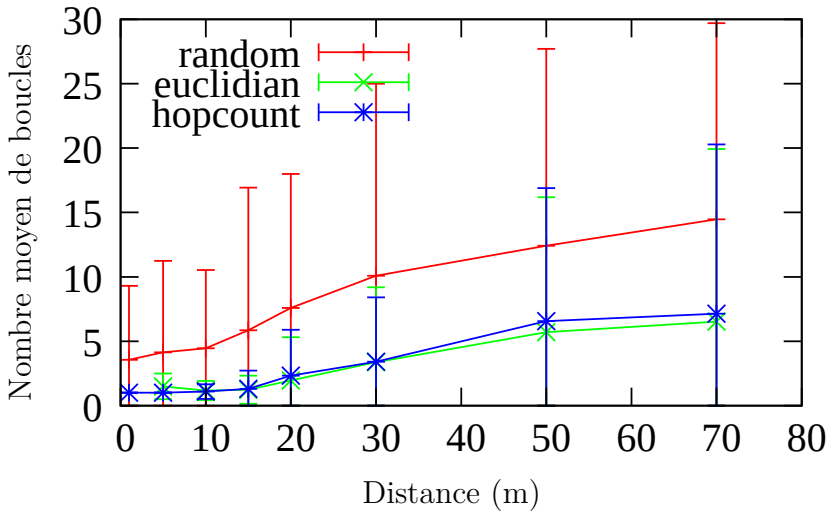

FIGURE 5.13 – Nombre moyen de boucles en fonction de la distance de déplacement pour les scénarii : *random*, *euclidian*, *hopcount* lorsqu'un seul nœud est mobile.

Quantification des boucles : La figure 5.13 montre le nombre moyen de boucles de routage générées pour chaque scénario en fonction de la distance de déplacement. Cette figure montre que ce nombre est en général faible (valeur moyenne maximale de 15, avec 30 comme valeur maximale). Cependant, quel que soit le scénario, ce nombre semble croître avec la distance de déplacement, particulièrement à partir de $10m$ pour les scénarii *euclidian* et *hopcount*. La figure 5.13 montre également que le scénario *random* produit plus de boucles en moyenne que les scénarii *euclidian* et *hopcount* qui semblent se confondre, avec néanmoins un léger avantage pour *euclidian* pour les distances $50m$ et $70m$. Le scénario *euclidian* montre un gain qui va jusqu'à 8% par rapport au scénario *hopcount*, et un gain qui va jusqu'à 55% par rapport au scénario *random*.

Pourcentage de boucles traitées : Les figures 5.14, 5.15, et 5.16 montrent le pourcentage de boucles corrigées par DLF-2, DLF-3 et DLF-4 respectivement pour le scénario *hopcount*, le scénario *euclidian* et le scénario *random*.

La figure 5.14 montre que DLF-2, DLF-3, et DLF-4, ont de très bonnes performances suivant la distance pour le scénario *hopcount*. De $1m$ à $20m$ de déplacement, 100% des boucles sont traitées. Ce pourcentage chute légèrement à partir de $30m$ de déplacement

de la destination pour atteindre 97%. Cela s'explique par le fait que jusqu'à $20m$ de déplacement, la taille maximale des boucles générées est de 2. A partir de $30m$, cette taille maximum monte jusqu'à 6 pour atteindre 10 pour $70m$ de déplacement comme le montre la figure 5.17. La figure 5.14 permet également de conclure que la majorité des boucles générées pour le scénario *hopcount* sont de taille 2 car DLF-2 et DLF-4 ont sensiblement le même pourcentage de boucles traitées.

La figure 5.15 montre également que DLF-2, DLF-3 et DLF-4 ont également de très bonnes performances en termes de pourcentage de boucles traitées quelle que soit la distance de déplacement pour le scénario *euclidian*. Cependant, nous notons que DLF-4 est meilleure que DLF-2 et DLF-3 avec un pourcentage presque toujours égal à 100% tandis que DLF-2 et DLF-3 qui semblent se confondre ont une performance qui décroit légèrement à partir de $15m$ de déplacement et va jusqu'à 97%. Cela donne l'information selon laquelle la majorité des boucles de routage générées sont de taille 2 avec certaines pouvant aller jusqu'à 8 (figure 5.17).

La figure 5.16 montre que DLF-2, DLF-3 et DLF-4 ont également de très bonnes performances en termes de pourcentage de boucles corrigées pour le scénario *random*. En effet, ce pourcentage est toujours compris entre 99.5% et 100%. Cela s'explique par la taille maximale des boucles générées qui oscille entre 2 et 6 comme le montre la figure 5.17.

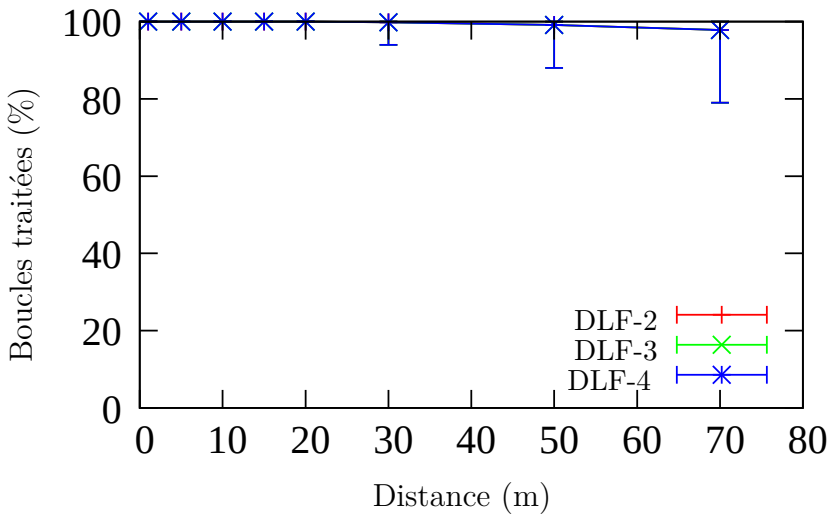

FIGURE 5.14 – Pourcentage de boucles corrigées par DLF-2, DLF-3, et DLF-4 en fonction de la distance de déplacement des nœuds, pour le cas du scénario *hopcount* lorsqu'un seul nœud est mobile.

Durée de la transition : Les figures 5.18, 5.19, et 5.20 montrent la durée de la transition produite par DLF-2, DLF-3, et DLF-4 respectivement pour les scénarii *hopcount*, *euclidian*, et *random* en termes de nombre d'étapes nécessaires pour que tous les nœuds convergent. Ces trois figures montrent premièrement que, quel que soit le scénario ou la taille de k $(2, 3,$ ou $4)$, le nombre d'étapes nécessaires est relativement faible (nombre maximal atteint 6, cf figure 5.20). Cela traduit une convergence rapide des nœuds du réseau. Ensuite, nous constatons que ce nombre croît avec la distance de déplacement des nœuds. Cela s'explique par le fait que le nombre de boucles générées croît également avec la distance de déplacement. Or, plus on a de boucles, plus on a besoin d'étapes pour effectuer une migration sans boucles. Enfin, nous notons que DLF-2, DLF-3 et DLF-4 produisent le même nombre d'étapes pour les scénarii

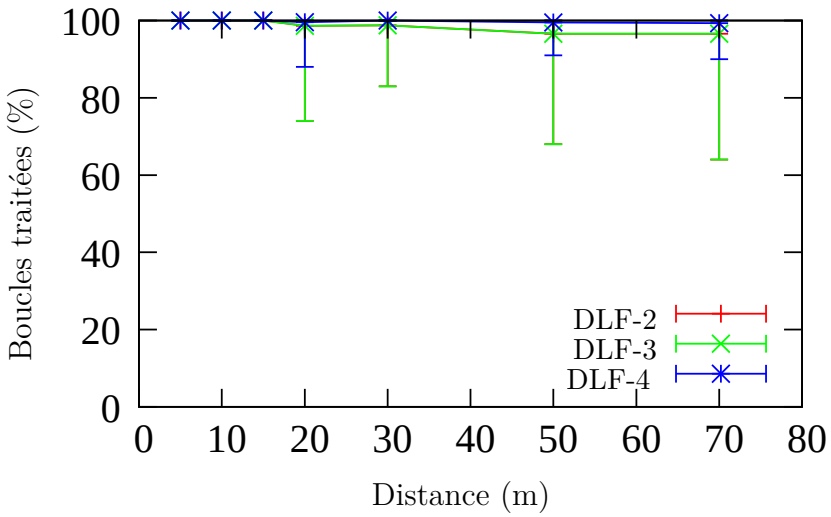

FIGURE 5.15 – Pourcentage de boucles corrigées par DLF-2, DLF-3, et DLF-4 en fonction de la distance de déplacement des nœuds, pour le cas du scénario *euclidian* lorsqu'un seul nœud est mobile.

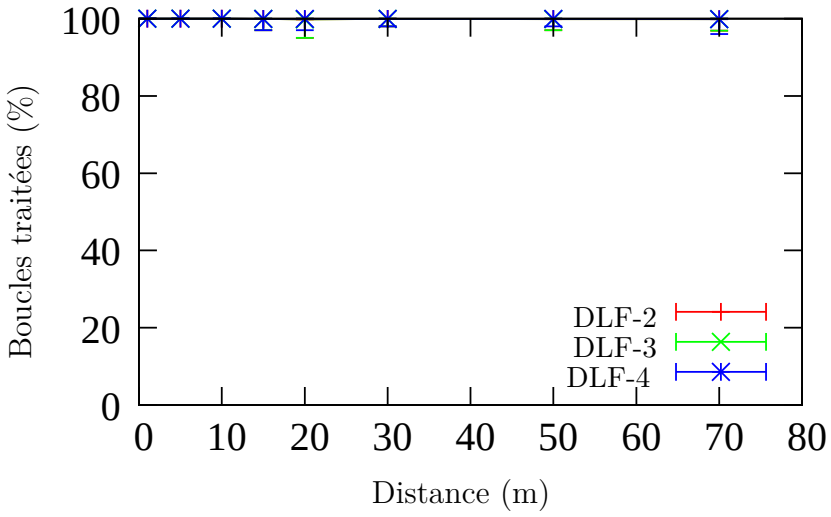

FIGURE 5.16 – Pourcentage de boucles corrigées par DLF-2, DLF-3, et DLF-4 en fonction de la distance de déplacement des nœuds, pour le cas du scénario *random* lorsqu'un seul nœud est mobile.

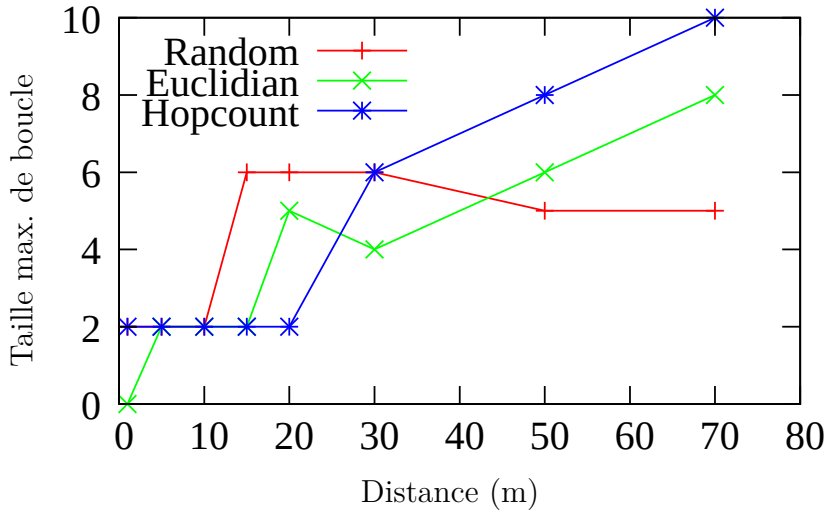

FIGURE 5.17 – Plus grande taille de boucle obtenue pour chacun des scénarii *hopcount*, *euclidian*, et *random* lorsqu'un seul nœud est mobile.

hopcount et *random* tandis DLF-4 nécessitent en moyenne légèrement plus d'étapes que DLF-2 et DLF-3 pour le scénario *euclidian*. Cela s'explique par le fait que DLF-4, pour ce scénario, traite plus de boucles que DLF-2 et DLF-3 (cf figure 5.15) tandis que pour les scénarii *hopcount* et *random*, DLF-2, DLF-3, et DLF-4 traitent sensiblement le même pourcentage de boucles (cf. figure 5.14 et figure 5.16).

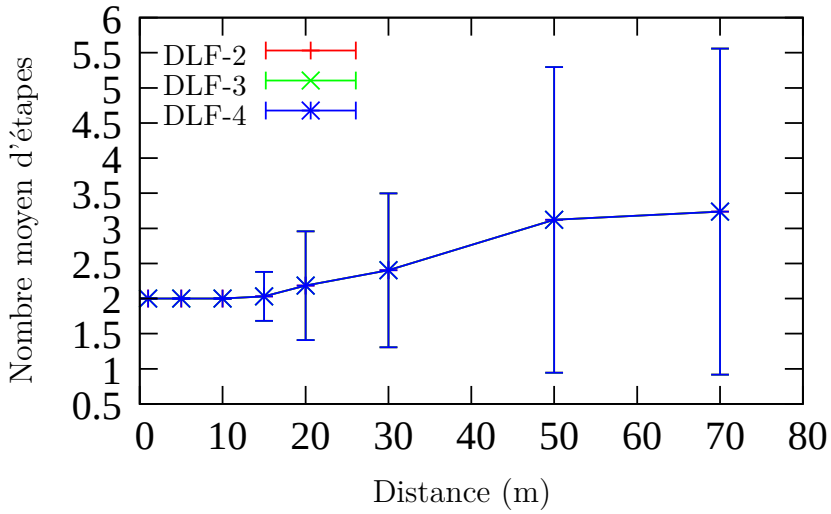

FIGURE 5.18 – Durée de la transition en nombre d'étapes produite par DLF-2, DLF-3, et DLF-4 en fonction de la distance de déplacement des nœuds, pour le cas du scénario *hopcount* lorsqu'un seul nœud est mobile.

Coût de la transition : Les figures 5.21, 5.22, et 5.23 montrent le coût de la transition en termes de nombre de messages de contrôle nécessaires à DLF-2, DLF-3, et DLF-4 pour chacun des scénarii *hopcount*, *euclidian*, *random* respectivement. Les trois figures semblent être identiques indépendamment du scénario et de la distance de déplacement du nœud mobile. Cela s'explique par le faible nombre de boucles identifiées dans chacun des scénarii, qui rend le coût de la transition fortement lié à valeur de k. Ainsi, nous observons sur chacune de ces courbes que DLF-2 nécessite environ 2 fois moins de messages de contrôle que DLF-3 qui à son tour, nécessite également 2 fois

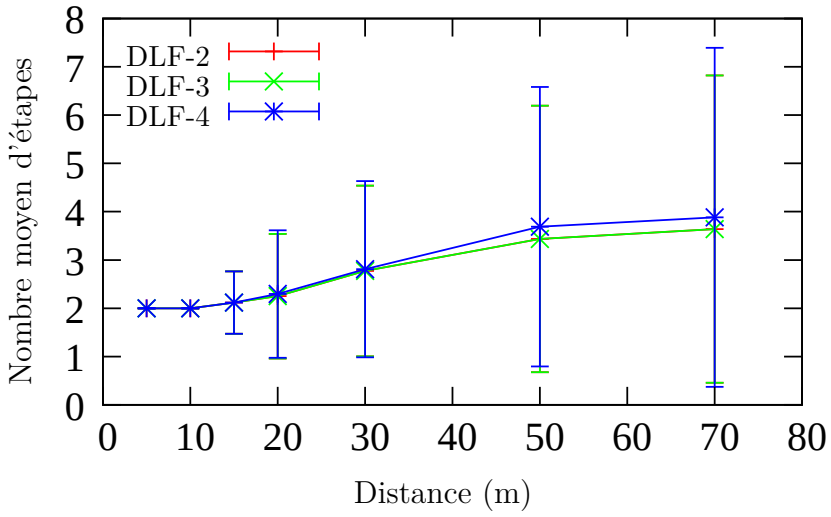

FIGURE 5.19 – Durée de la transition en nombre d'étapes produite par DLF-2, DLF-3, et DLF-4 en fonction de la distance de déplacement des nœuds, pour le cas du scénario *euclidian* lorsqu'un seul nœud est mobile.

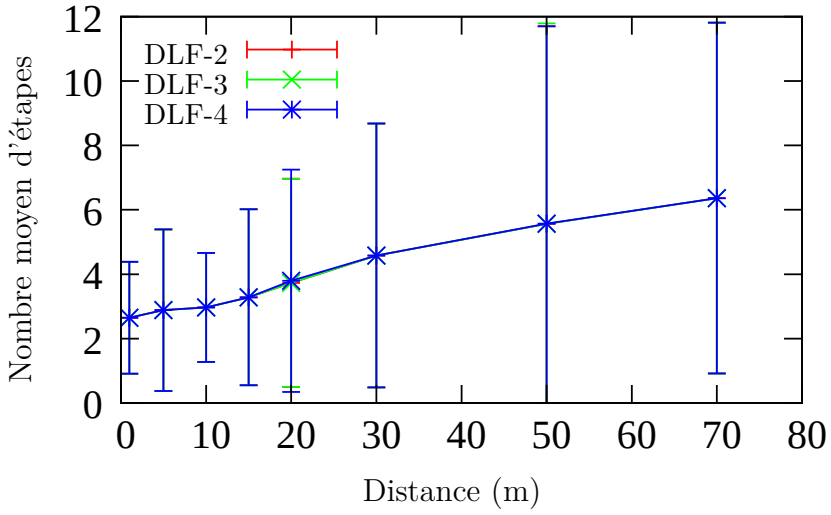

FIGURE 5.20 – Durée de la transition en nombre d'étapes produite par DLF-2, DLF-3, et DLF-4 en fonction de la distance de déplacement des nœuds, pour le cas du scénario *random* lorsqu'un seul nœud est mobile.

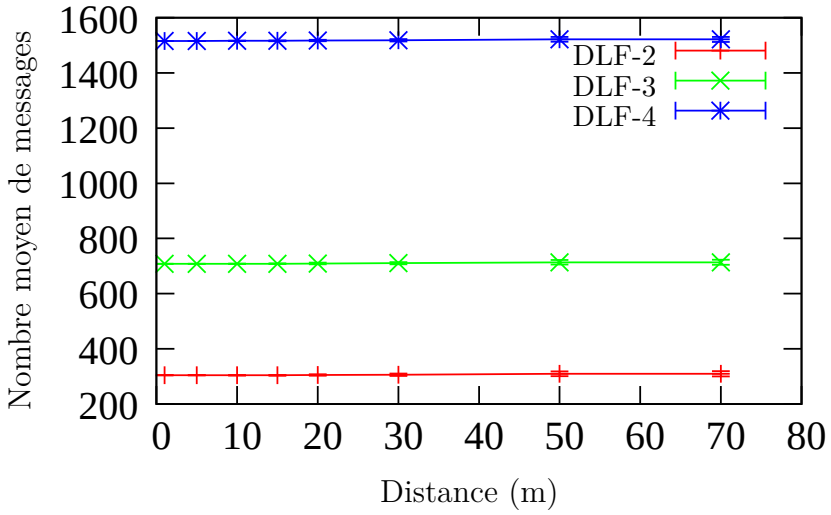

FIGURE 5.21 – Coût de la transition produite respectivement par DLF-2, DLF-3, et DLF-4, en fonction de la distance de déplacement des nœuds, pour le cas du scénario *hopcount* lorsqu'un seul nœud est mobile.

moins de messages de contrôle que DLF-4. Ainsi, plus la valeur de k est grande, plus coûteux est DLF-k.

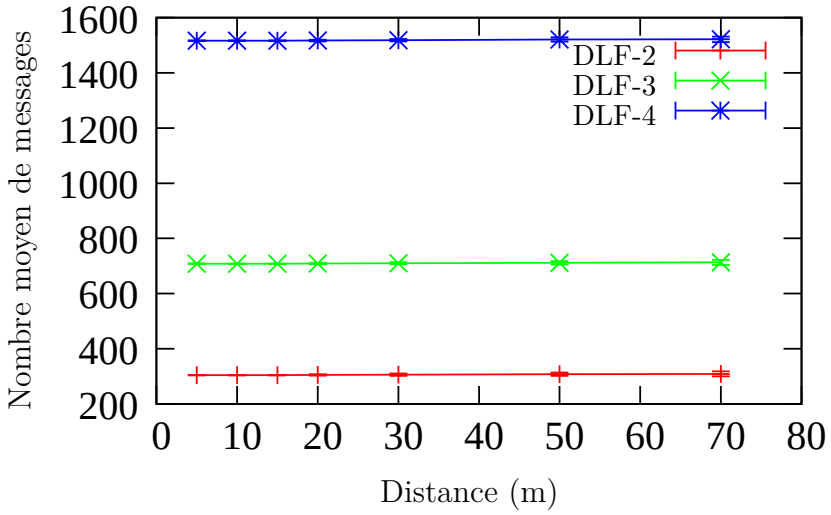

FIGURE 5.22 – Coût de la transition produite respectivement par DLF-2, DLF-3, et DLF-4, en fonction de la distance de déplacement des nœuds, pour le cas du scénario *euclidian* lorsqu'un seul nœud est mobile.

Temps d'exécution : Les figures 5.24, 5.25, et 5.20 montrent le temps d'exécution de DLF-2, DLF-3, et DLF-4 respectivement pour les scénarii *hopcount*, *euclidian*, et *random*. Nous notons premièrement que pour chacun de ces scénarii, DLF-2 est plus rapide que DLF-3 qui à son tour est plus rapide que DLF-4. Cela s'explique aisément par le fait que plus petite est la valeur de k, plus tôt sera stoppée la recherche décrite par l'algorithme 9. Nous notons ensuite que pour les trois valeurs de k, la distance de déplacement n'impacte pas véritablement le temps d'exécution. Cependant, nous notons quelques variations pour chaque valeur de k en fonction du scénario. Respectivement pour les scénarii *hopcount*, *euclidian*, *random* avec $k = 2$, le temps d'exécution avoisine 13, 15, et 18ms, avec $k = 3$, le temps avoisine 20, 22, et 27ms, et avec $k = 4$,

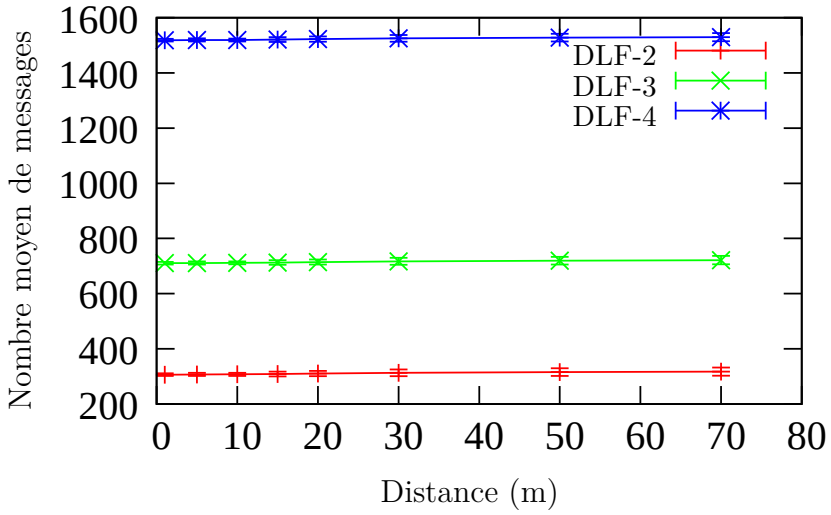

FIGURE 5.23 – Coût de la transition produite respectivement par DLF-2, DLF-3, et DLF-4, en fonction de la distance de déplacement des nœuds, pour le cas du scénario *random* lorsqu'un seul nœud est mobile.

le temps d'exécution avoisine 27, 33, et 37.5ms. Pour chaque taille de k, le temps croît légèrement. Cela est dû au temps que requièrent l'identification des boucles et la notification des nœuds bloqués par des boucles. Il est à noter qu'un petit nombre de nœuds en attente de notification bloqués par des boucles en série peuvent nécessiter plus de temps qu'un plus grand nombre de nœuds bloqués par des boucles pouvant être résolues en parallèle.

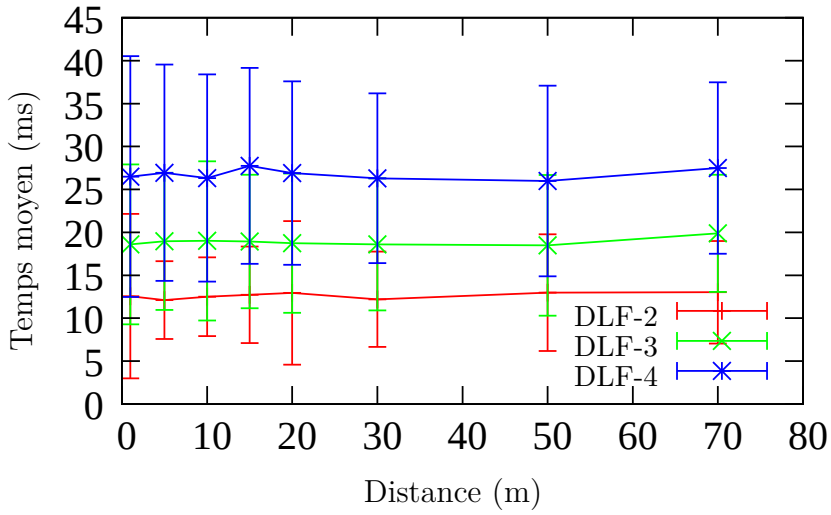

FIGURE 5.24 – Temps de calcul de chacune des transitions calculé respectivement par DLF-2, DLF-3, et DLF-4, en fonction de la distance de déplacement des nœuds pour le cas du scénario *hopcount* lorsqu'un seul nœud est mobile.

5.3.4 Résultats de DLF-k sur la mobilité d'un groupe de nœuds

Les résultats décrits dans cette partie portent sur la mobilité d'un groupe de nœuds.

Quantification des boucles : Les figures 5.27 et 5.28 montrent le nombre moyen de boucles identifiées pour les scénarii respectifs *hopcount* et *random*, et pour chacun des

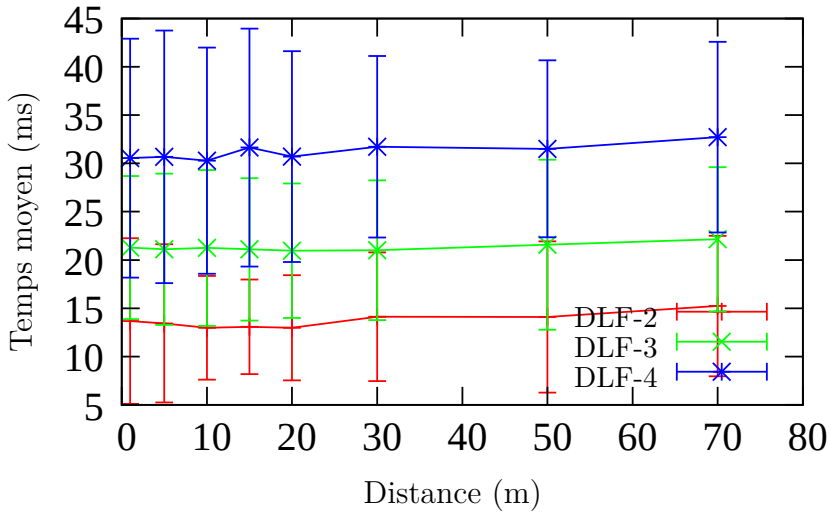

FIGURE 5.25 – Temps de calcul de chacune des transitions calculé respectivement par DLF-2, DLF-3, et DLF-4, en fonction de la distance de déplacement des nœuds pour le cas du scénario *euclidian* lorsqu'un seul nœud est mobile.

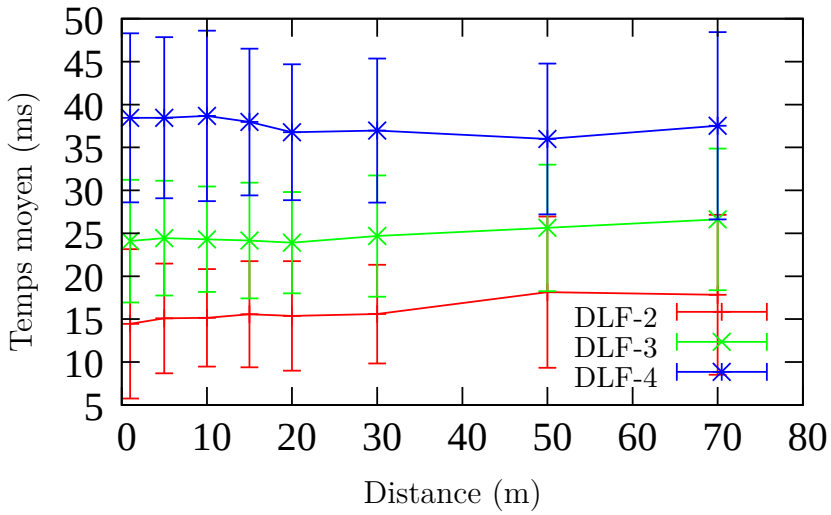

FIGURE 5.26 – Temps de calcul de chacune des transitions calculé respectivement par DLF-2, DLF-3, et DLF-4, en fonction de la distance de déplacement des nœuds pour le cas du scénario *random* lorsqu'un seul nœud est mobile.

pourcentages de nœuds retenus. La figure 5.27 montre qu'avec la métrique *hopcount*, le nombre de boucles est de l'ordre du millier, tandis que la figure 5.28 montre que la métrique *random* impacte considérablement le nombre de boucles en cas de mobilité d'un groupe de nœuds car, il est de l'ordre de centaines de milliers. Nous notons cependant des similitudes à savoir : ce nombre croît avec la distance de déplacement et le pourcentage des nœuds mobiles. En effet, plus la distance de déplacement est incrémentée, plus le nombre de boucles s'accroît. De même, bouger 100% des nœuds génère jusqu'à 100 fois plus de boucles que bouger 25% des nœuds, pour une même distance.

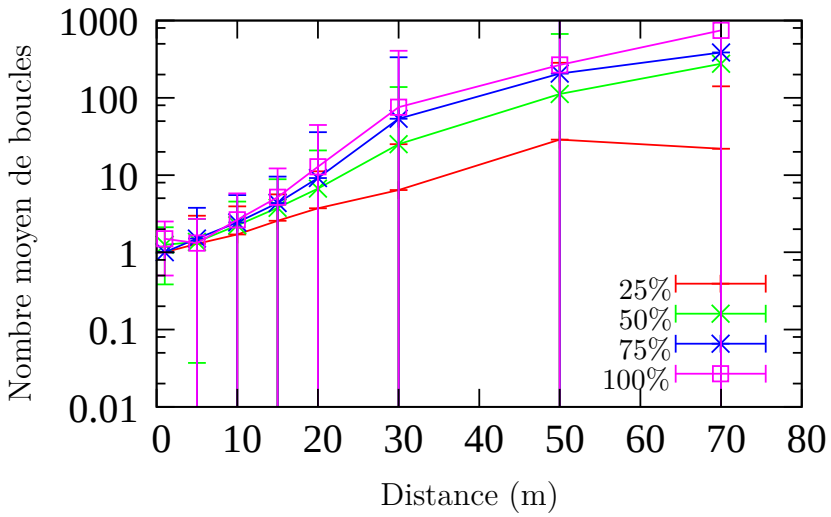

FIGURE 5.27 – Nombre moyen de boucles en fonction de la distance de déplacement, pour le scénario *hopcount* et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

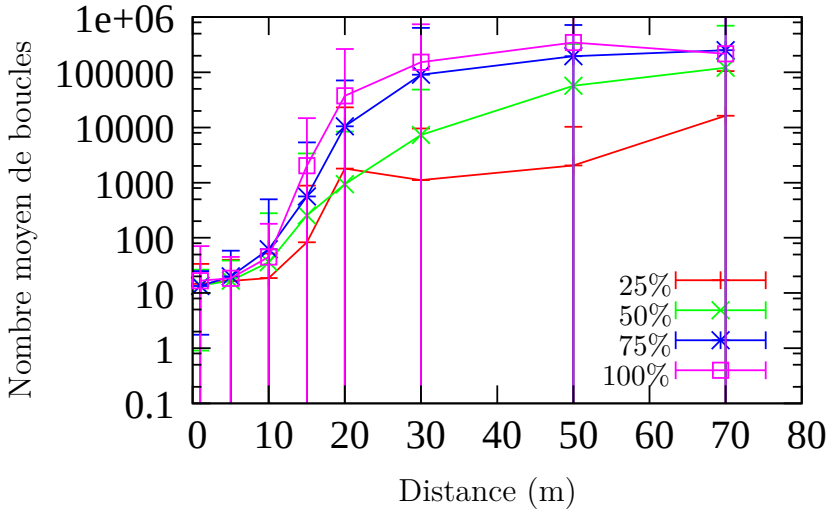

FIGURE 5.28 – Nombre de boucles moyen en fonction de la distance de déplacement, pour le scénario *random* et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

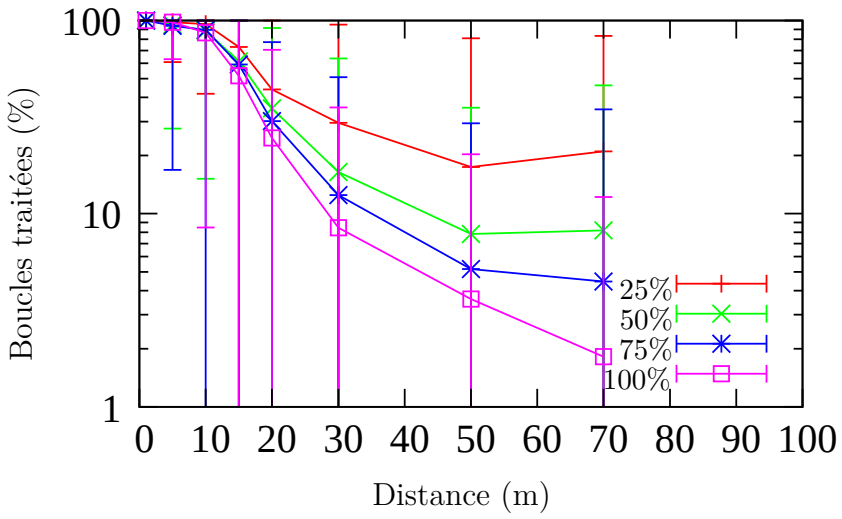

FIGURE 5.29 – Pourcentage de boucles corrigées par DLF-2 en fonction de la distance de déplacement des nœuds, pour le scénario *hopcount*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

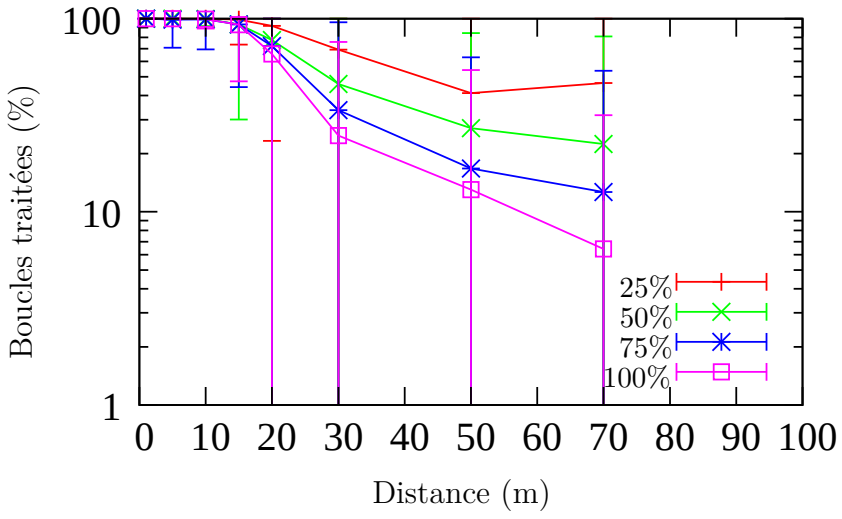

FIGURE 5.30 – Pourcentage de boucles corrigées par DLF-4 en fonction de la distance de déplacement des nœuds, pour le scénario *hopcount*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

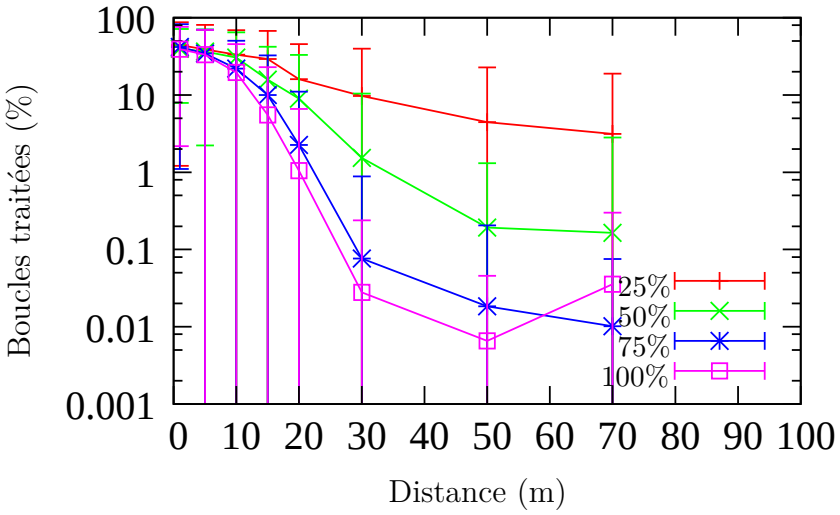

FIGURE 5.31 – Pourcentage de boucles corrigées par DLF-2 en fonction de la distance de déplacement des nœuds, pour le scénario *random*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

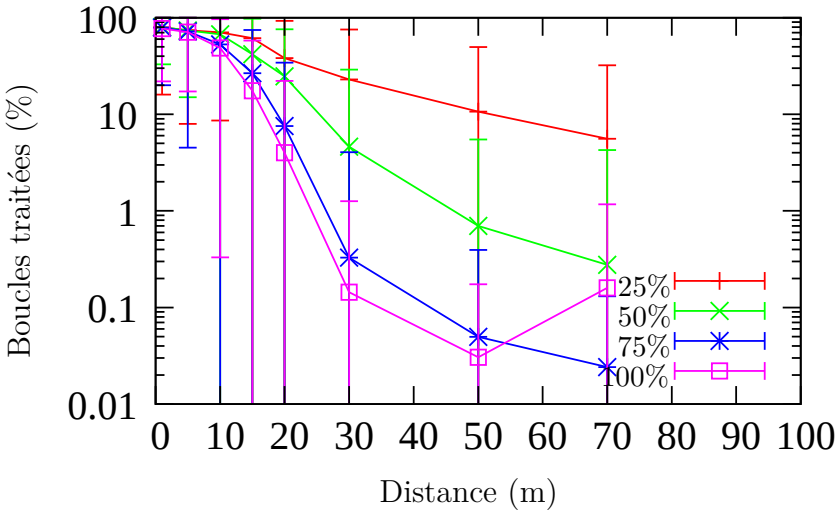

FIGURE 5.32 – Pourcentage de boucles corrigées par DLF-4 en fonction de la distance de déplacement des nœuds, pour le scénario *random*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

Pourcentage de boucles traitées : Les figures 5.29 et 5.30 montrent les pourcentages de boucles corrigées respectivement par DLF-2 et DLF-4 dans le cas du scénario *hopcount*. Nous notons que pour une faible distance de déplacement (inférieure ou égale à $10m$), DLF-2 et DLF-4 corrigent 100% des boucles. Au delà de ce seuil, le taux de boucles corrigées décroit avec la distance. En effet, la figure 5.29 montre que DLF-2 passe de 100% des boucles corrigées à respectivement 21%, 8%, 4%, et 2% pour 25%, 50%, 75%, et 100% de nœuds mobiles, tandis que la figure 5.30 montre que DLF-4 passe de 100% de boucles traitées à respectivement 46%, 22%, 13%, et 6% pour 25%, 50%, 75%, et 100% de nœuds mobiles. Le pourcentage de boucles corrigées est inversement proportionnel ici à la taille du groupe de nœuds mobiles.

De même, les figures 5.31 et 5.32 montrent les pourcentages de boucles corrigées respectivement par DLF-2 et DLF-4 dans le cas du scénario *random*. Comme dans le cas du scénario *hopcount*, les pourcentages des boucles corrigées sont inversement

proportionnels aux pourcentages de nœuds mobiles, autant pour DLF-2 que pour DLF-4. En effet, la figure 5.31 montre que DLF-2 passe de 44% à 3% pour 25% des nœuds, de 40% à 0.2% pour 50% des nœuds mobiles, de 42% à 0.02% pour 75% des nœuds, de 39% à 0.006% avec une légère remontée vers 0.03% à 70m pour 100% des nœuds. Tandis que la figure 5.32 montre que DLF-4 passe de 79% à 6% pour 25% des nœuds, de 80% à 0.3% pour 50% des nœuds mobiles, de 79% à 0.02% pour 75% des nœuds, et de 77% à 0.03% avec un léger pic à 1.17% pour 70%.

Quel que soit le scénario (*hopcount* ou *random*), le pourcentage de boucles corrigées est très faible. Cela donne l'information que la majorité des boucles générées dans le cas de mobilité de groupes de nœuds est de taille supérieure à 4. Les figures 5.33 et 5.34 renforcent cette conclusion car les tailles de boucles pour le scénario *hopcount* vont jusqu'à 38 et pour le scénario *random* jusqu'à 68. Un k fixé à 2 ou 4 est ainsi trop faible.

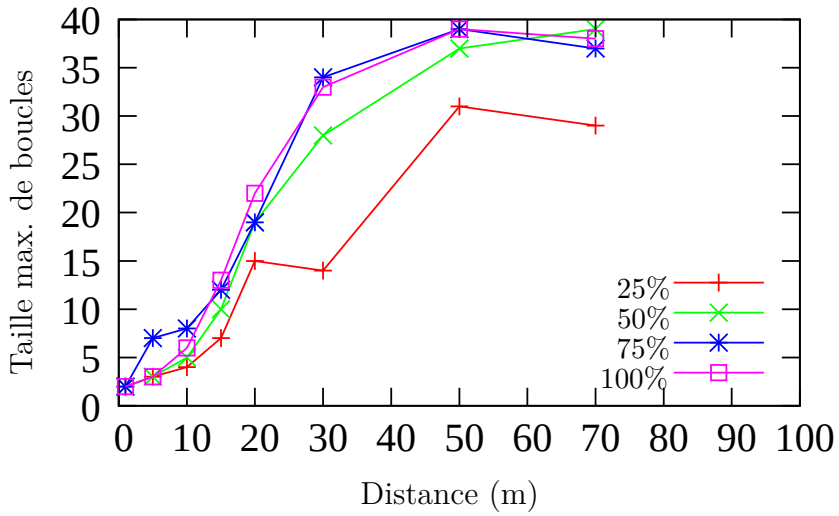

FIGURE 5.33 – Taille maximale des boucles en fonction de la distance de déplacement des nœuds, pour le scénario *hopcount*.

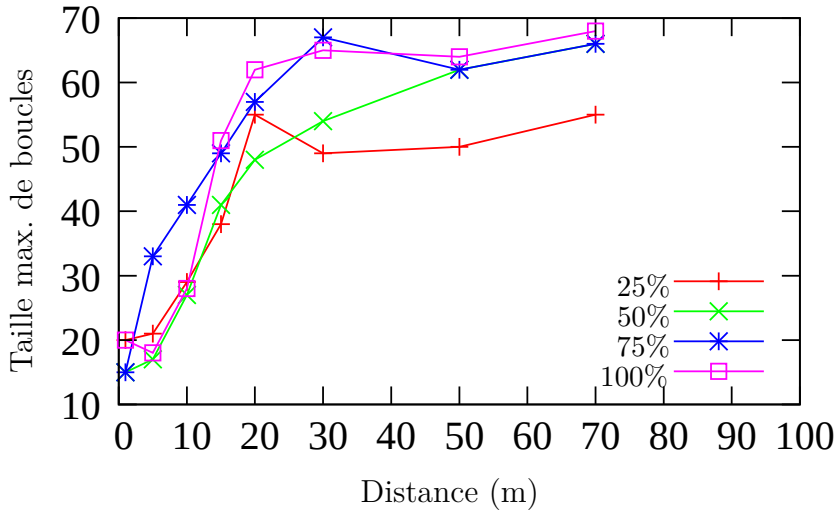

FIGURE 5.34 – Taille maximale des boucles en fonction de la distance de déplacement des nœuds, pour le scénario *random*.

Durée de la transition : Les figures 5.35 et 5.36 montrent la durée des transitions produites par DLF-2 et DLF-4 respectivement pour le scénario *hopcount*. La figure 5.35 montre que DLF-2 produit des transitions de deux étapes en moyenne. Elle montre également qu'au delà de $20m$ de déplacement pour 100% des nœuds mobiles et qu'au delà de $50m$ de déplacement pour 75% des nœuds mobiles, DLF-2 n'a pas été capable de calculer une transition valide pour toutes les répétitions, car le nombre d'étapes est égal à 0. La figure 5.36 quant-à elle montre que quel que soit le pourcentage de noeuds mobiles ou la distance de déplacement, DLF-4 arrive toujours à produire une transition valide pour au moins une répétition car le nombre moyen d'étapes est toujours différent de 0. Cette figure montre également que le nombre d'étapes nécessaires est relativement faible (4 en moyenne) et que plus le pourcentage de nœuds mobiles est élevé, ce nombre est légèrement plus grand également sauf exception de la distance $50m$ où le nombre d'étapes nécessaires pour 75% est au-dessus du nombre d'étapes nécessaires pour 100%.

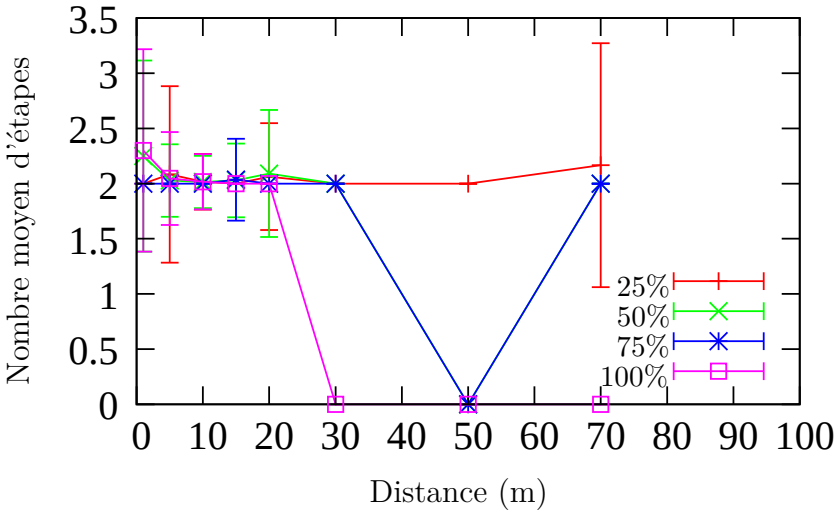

FIGURE 5.35 – Durée de la transition en nombre d'étapes produite par DLF-2 en fonction de la distance de déplacement des nœuds, pour le scénario *hopcount*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

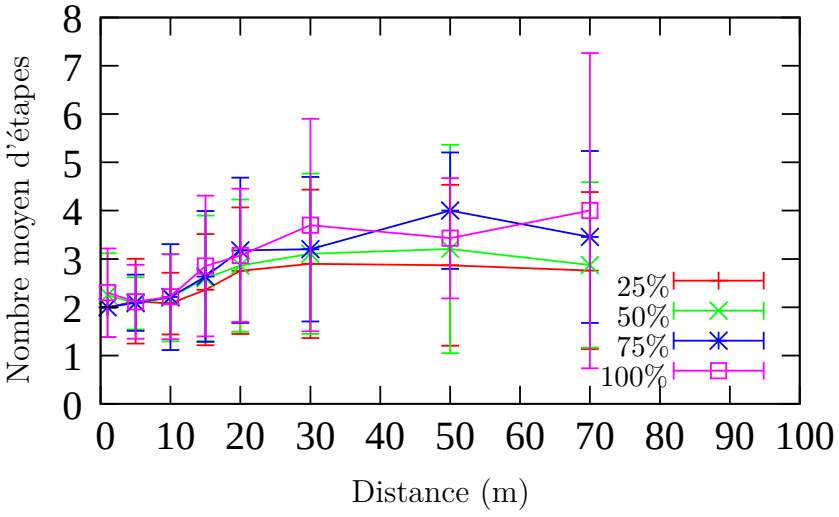

FIGURE 5.36 – Durée de la transition en nombre d'étapes produite par DLF-4 en fonction de la distance de déplacement des nœuds, pour le scénario *hopcount*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

Les figures 5.37 et 5.38 montrent quant-à elles la durée de la transition en nombre d'étapes respectivement pour DLF-2 et DLF-4 dans le cas du scénario *random*. La figure 5.37 montre que DLF-2 ne réussit en général pas à calculer une transition valide quel que soit le pourcentage de nœuds mobiles, et lorsqu'elle réussit, elle a besoin de 2 étapes en moyenne pour faire converger les nœuds. La figure 5.38 montre que pour 25% de nœuds mobiles, DLF-4 calcule une transition qui nécessite généralement autour de 4 étapes ; pour 50% des nœuds, ce nombre va jusqu'à 5 étapes à $20m$, mais au delà il chute pour n'avoir aucune transition valide à $70m$; pour 75% de nœuds mobiles, ce nombre est également autour de 4, et au delà de $20m$ aucune transition valide n'est calculée ; pour 100% de nœuds mobiles, ce nombre va jusqu'à 6 étapes en moyenne à l'exception de $30m$ et $50m$ de déplacement où aucune transition valide n'est calculée.

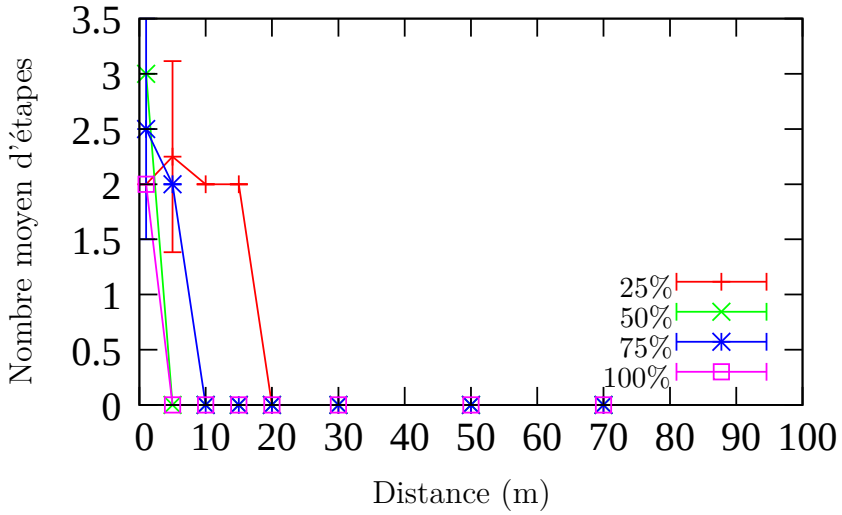

FIGURE 5.37 – Durée de la transition en nombre d'étapes produite par DLF-2 en fonction de la distance de déplacement des nœuds, pour le scénario *random*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

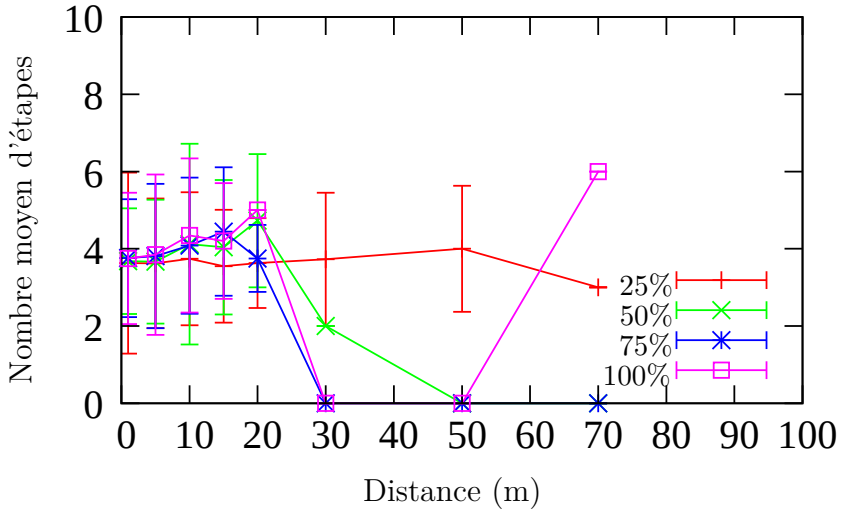

FIGURE 5.38 – Durée de la transition en nombre d'étapes produite par DLF-4 en fonction de la distance de déplacement des nœuds, pour le scénario *random*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

Coût de la transition : Les figures 5.39 et 5.40 montrent respectivement le coût de la transition en nombre de messages de contrôle de DLF-2 et DLF-4 pour le scénario *hopcount*. La figure 5.39 montre que le coût pour DLF-2, lorsqu'une transition valide est calculée, est autour de 300 messages, indépendamment du pourcentage de nœuds mobiles. La figure 5.40 montre que DLF-4 nécessite beaucoup plus de messages avec une moyenne autour de 1500 messages de contrôle nécessaires, avec de légères différences en fonction du pourcentage de nœuds mobiles : les pourcentages les plus élevés nécessitent plus de messages.

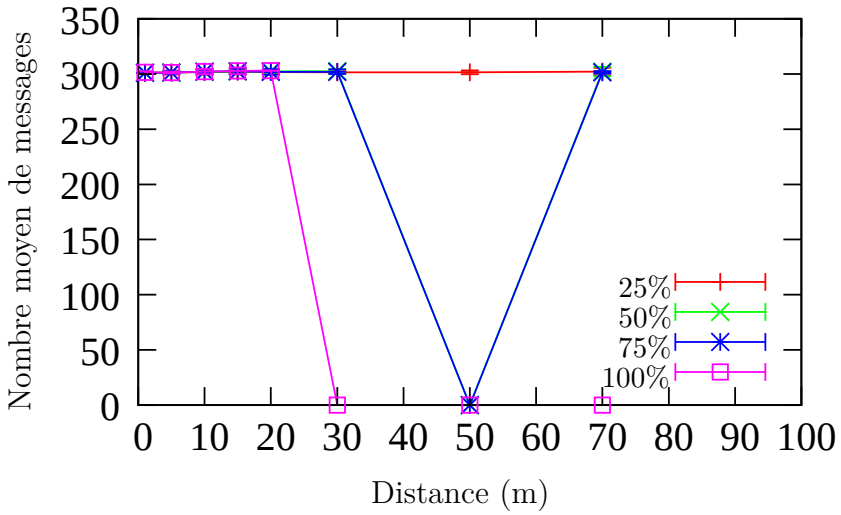

FIGURE 5.39 – Coût de la transition produite par DLF-2 en fonction de la distance de déplacement des nœuds, pour le scénario *hopcount*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

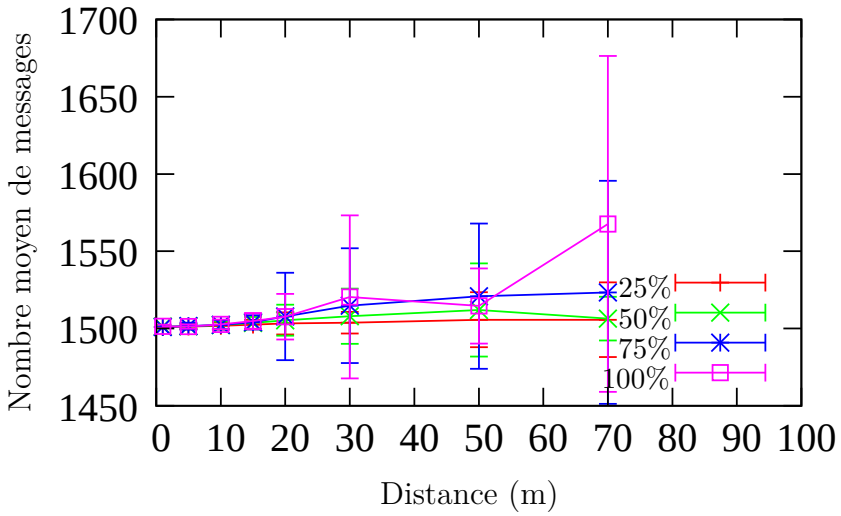

FIGURE 5.40 – Coût de la transition produite par DLF-4 en fonction de la distance de déplacement des nœuds, pour le scénario *hopcount*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

De même, les figures 5.41 et 5.42 montrent respectivement le coût de la transition en nombre de messages de contrôle de DLF-2 et DLF-4 pour le scénario *random*. Comme c'est le cas pour le scénario *hopcount*, la figure 5.41 montre que lorsque DLF-2 calcule une transition valide, elle nécessite environ 300 messages. Cependant au delà de $20m$,

ce nombre est égal à 0 dû à l'absence de transitions valides. La figure 5.42 quant à elle montre que DLF-4 nécessite en moyenne un nombre de messages autour de 1600 messages avec une exception pour 100% des nœuds mobiles à $70m$ qui va jusqu'à 9673 messages en moyenne.

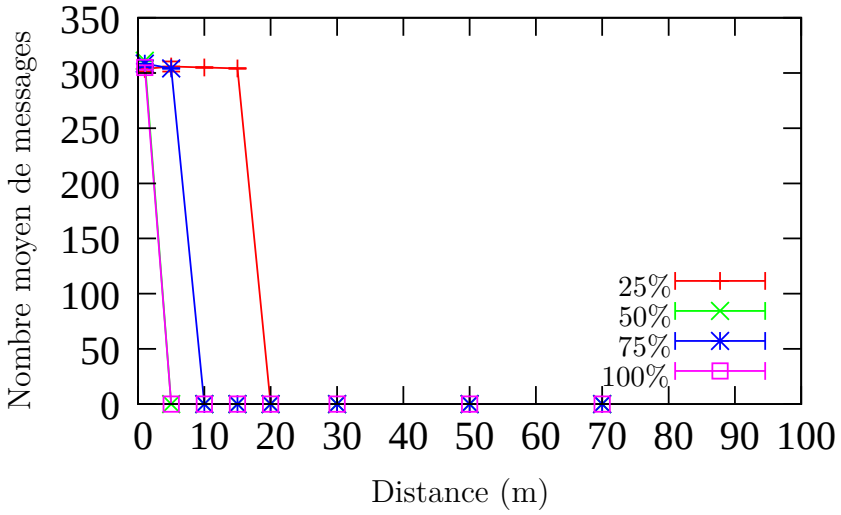

FIGURE 5.41 – Coût de la transition produite par DLF-2 en fonction de la distance de déplacement des nœuds, pour le scénario *random*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

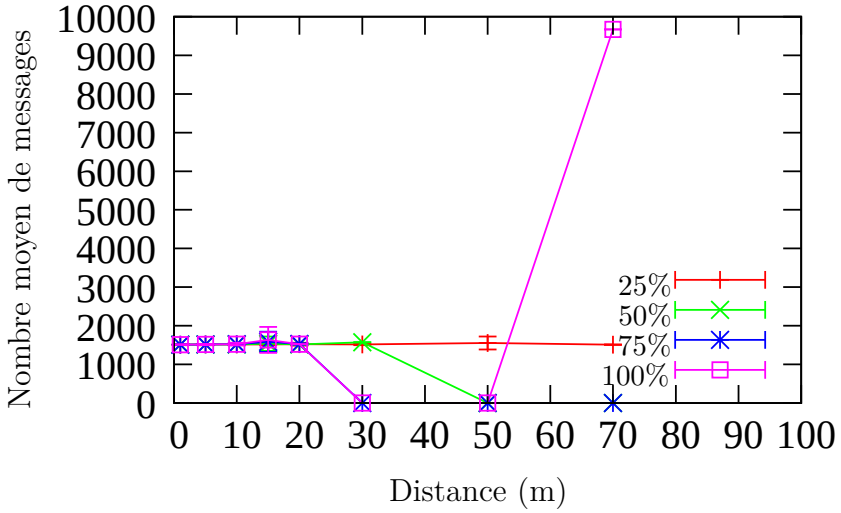

FIGURE 5.42 – Coût de la transition produite par DLF-4 en fonction de la distance de déplacement des nœuds, pour le scénario *random*, et en considérant 4 pourcentages de nœuds déplacés : 25%, 50%, 75%, 100%.

Temps d'exécution : Les figures 5.43 et 5.44 montrent respectivement la comparaison des temps d'exécution à l'aide des *threads* de DLF-2 et DLF-4 en fonction de la distance de déplacement, respectivement pour *hopcount* et *random*. Les temps d'exécution étant sensiblement les mêmes pour tous les pourcentages de nœuds mobiles, nous avons choisi de présenter ici les résultats lorsque la totalité des nœuds bougent.

Ces deux figures montrent des similitudes. Premièrement les temps d'exécution sont relativement faibles. Deuxièmement, DLF-2 met moins de temps que DLF-4 quelle que soit la distance. Cela est logique puisque DLF-2 stoppe sa recherche d'appartenance

à des boucles plus tôt que DLF-4. Troisièmement, la distance semble ne pas avoir un grand impact dans les deux scénarii sauf à $70m$ pour DLF-2, et au delà de $30m$ pour DLF-4.

Cependant, DLF-2 et DLF-4 semblent prendre plus de temps pour le scénario *random* que pour le scénario *hopcount*. En effet, pour le scénario *hopcount*, DLF-2 prend entre $12ms$ et $16ms$ tandis que pour le scénario *random*, DLF-2 prend de $18ms$ à $22ms$. De même, DLF-4 pour le scénario *hopcount* prend de $24ms$ à $33ms$ tandis que pour le scénario *random*, DLF-4 prend de $37ms$ à $43ms$. Cela s'explique par le fait que, avec le scénario *random*, on a plus de boucles générées.

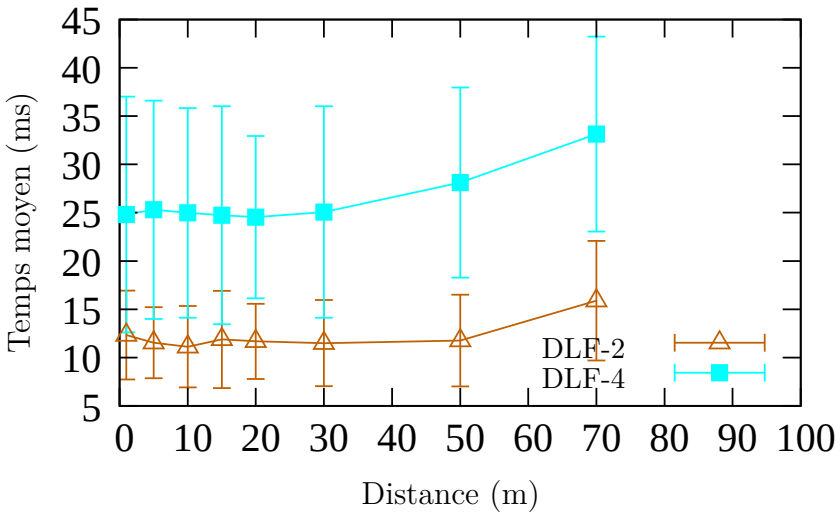

FIGURE 5.43 – Temps de calcul de chacune des transitions calculé respectivement DLF-2, et DLF-4, en fonction de la distance de déplacement des nœuds pour le cas du scénario *hopcount* lorsque 100% des nœuds bougent.

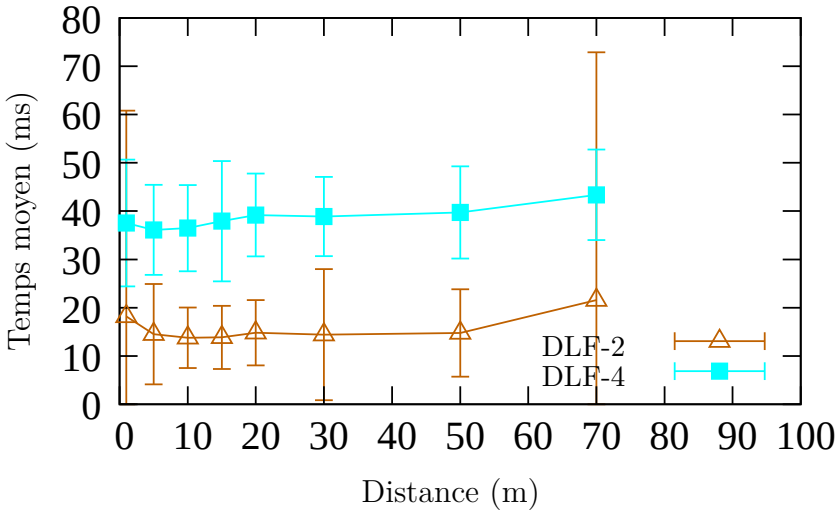

FIGURE 5.44 – Temps de calcul de chacune des transitions calculé respectivement DLF-2, et DLF-4, en fonction de la distance de déplacement des nœuds pour le cas du scénario *random* lorsque 100% des nœuds bougent.

Conclusion

Dans ce chapitre, nous nous sommes intéressés au cas de réseaux dynamiques, et avons proposé un protocole distribué DLF-k qui traite toutes les boucles de taille inférieure ou égale à k, avec $k \geq 2$. Puis, nous avons étudié les performances de DLF-k avec $k \in \{2, 3, 4\}$ pour deux types d'applications mobiles à savoir : les applications avec un unique noeud mobile qui est la destination, et les applications avec un groupe de nœuds mobiles. Trois métriques de poids des liens ont été considérés à savoir : la métrique *hopcount*, la métrique euclidienne, et la métrique aléatoire.

Les différentes simulations ont permis de tirer les conclusions suivantes :

— En termes de quantification des boucles : les applications avec un seul nœud mobile génèrent un faible nombre de boucles (de l'ordre de dizaines) comparativement aux applications avec un groupe de nœuds mobiles, où le nombre de boucles est de l'ordre de milliers, voire de centaines de milliers en fonction de la métrique des liens et du pourcentage de nœuds mobiles. Les deux types d'applications montrent néanmoins des similitudes à savoir que le nombre de boucles croît avec la distance de déplacement des nœuds, et la métrique aléatoire produit toujours plus de boucles que la métrique *hopcount* (resp. euclidienne).

— En termes de pourcentage de boucles traitées : dans les deux types d'applications, plus la valeur de k est élevée, plus le pourcentage de boucles corrigées est élevé. Cependant, DLF-2, DLF-3, et DLF-4 ont de très bonnes performances (97% à 100%) pour le cas des applications avec un unique nœud mobile, contrairement aux applications avec un groupe de nœuds mobiles où les pourcentages sont rapidement très faibles avec la distance de déplacement (n'atteignant à peine 0.01% de boucles corrigées). Cela traduit qu'un k fixé à 4 est adapté aux applications d'un unique nœud mobile, mais inadapté aux applications de plusieurs nœuds mobiles car trop faible par rapport à la taille des boucles générées.

— En termes de durée de la transition : dans les deux cas d'application (mobilité d'un nœud et mobilité d'un groupe de nœuds), le nombre d'étapes nécessaires pour effectuer la migration est faible (respectivement 6 et 4 maximum). Nous notons néanmoins que plus k est grand et le pourcentage de nœuds mobiles est élevé, plus DLF-k peut nécessiter des étapes supplémentaires.

— En termes de coût de la transition : DLF-k nécessite un nombre de messages de contrôle qui dépend de la taille de k et du nombre de boucles générées. Pour les applications avec un seul nœud mobile, le nombre de boucles étant relativement faible, ce coût dépend grandement de la valeur de k et peu de la métrique du poids des liens, tandis que pour les applications avec plusieurs nœuds mobiles, ce coût dépend de k, du pourcentage de nœuds mobiles, de la distance de déplacement des nœuds, et de la métrique des liens. Cela est lié à la quantification des boucles qui dépendent de ces paramètres. Ainsi, DLF-4 est plus coûteux que DLF-3, qui à son tour est plus coûteux que DLF-2.

— En termes de temps d'exécution : obtenu à l'aide d'une simulation par le biais des *threads*, il est relativement faible quel que soit le type d'applications et n'est pas grandement impacté par la distance de déplacement des nœuds. Cependant, plus le k est grand, plus le temps d'exécution est élevé.

En somme DLF-k est adapté aux applications dynamiques avec uniquement une destination mobile, mais pas aux applications avec un groupe de nœuds mobiles. Dans le premier cas, une petite valeur de k suffit pour corriger la totalité des boucles, tandis

que dans le second cas, il faudrait fixer une grande valeur de k pour corriger toutes les boucles, ce qui en pratique n'est pas adapté car générerait un coût important.

Conclusion et perspectives

Le routage dans un réseau permet de déterminer les routes que doivent suivre les paquets pour atteindre les destinations. Il peut être calculé de manière statique lorsqu'on fixe ces routes *a priori*, ou de manière dynamique lorsque les routes sont calculées périodiquement, en fonction de l'état du réseau. Dans le cas dynamique, un protocole de routage définit les mécanismes à appliquer pour avoir les chemins des sources aux destinations. Cependant, de nombreuses raisons telles que la détection d'événements particuliers (feu de forêt, éruption volcanique, accident de circulation, etc), les mises à jour sécuritaires de protocole, ou encore l'ajout ou le retrait d'un nœud du réseau, peuvent amener à faire évoluer le réseau vers un autre protocole de routage. Ce nouveau protocole de routage peut produire des chemins complètement différents de ceux calculés par le protocole initial. Ces différences dans le routage conduisent au fait que la migration des nœuds d'un protocole de routage initial $\mathcal{R}_i$ vers un protocole de routage $\mathcal{R}_f$ devient alors une tâche complexe et critique.

Quels que soient le type et la raison du changement, un administrateur réseau ne peut ni se permettre d'éteindre tous les nœuds du réseau, de configurer le nouveau protocole sur chacun d'eux, puis de tous les redémarrer (le changement ne doit pas perturber le fonctionnement actuel du réseau), ni envisager que tous les nœuds détectent au même moment la survenue d'un événement particulier (les nœuds sont généralement déployés sur de grandes surfaces géographiques). Or, l'ordre de migration des nœuds lorsqu'il n'est pas contrôlé, est susceptible de mettre le réseau dans un état inconsistant au travers de la génération de boucles de routage. Ces boucles de routage se traduisent par le fait que certains nœuds routent selon $\mathcal{R}_f$ tandis que d'autres continuent de router suivant $\mathcal{R}_i$ avant la convergence complète du réseau. Ces boucles dégradent considérablement les performances du réseau avec pour conséquences une plus grande consommation de la bande passante, l'augmentation des délais d'attente, l'augmentation de la congestion, la perte de paquets, et l'augmentation de la consommation d'énergie.

Cette thèse avait pour objectif principal l'évitement complet de boucles de routage durant la phase migratoire, en considérant plusieurs contextes de génération des boucles de routage.

Contributions

Les contributions de cette thèse suivent les trois contextes d'apparition de boucles de routage étudiés durant la thèse : le contexte statique centralisé où le réseau est statique et le routage est centralisé, le contexte statique distribué où le réseau reste statique et le protocole de routage est distribué, et le contexte de mobilité où une partie ou la totalité des nœuds est mobile.

Contexte statique centralisé

Dans le contexte statique centralisé, nous avons considéré qu'il existe un nœud central qui se charge de contacter chaque nœud individuel du réseau pour lui notifier son changement de protocole. Cette notification a lieu après le calcul de l'ordre de migration correcte des nœuds. Pour ce faire, nous avons supposé un changement de métriques des liens dans le réseau et avons proposé deux heuristiques : *Strongly Connected Component Heuristic with merged steps* (SCH-m) et *Avoiding Cycle Heuristic* (ACH).

La première heuristique (SCH-m [45]) est une amélioration mineure de l'heuristique SCH-p proposée dans [13] avec de bonnes performances. Elle accélère la transition produite en introduisant la notion de fusion d'étapes dans le cas d'un réseau multi-destinations. En effet, au lieu que la transition finale soit une application séquentielle de chaque transition calculée par destination, SCH-m propose d'avoir une transition pour toutes les destinations qui a pour nombre total d'étapes le nombre maximum d'étapes qu'une transition peut avoir pour une destination. Cela réduit considérablement le nombre d'étapes comparativement au traitement séquentiel. Cette fusion permet également au nœud central de ne contacter qu'une seule fois un nœud par étape pour toutes les destinations considérées, ce qui diminue également grandement le nombre de messages de contrôle nécessaires à la configuration des nœuds.

La deuxième heuristique (ACH [46]), identifie une boucle de routage à un cycle dans une composante connexe. ACH introduit la notion de casser un cycle, qui correspond à identifier les nœuds du cycle capables d'empêcher la survenue d'une boucle suivant cette règle : un nœud empêche une boucle de se former s'il route suivant le protocole de routage initial en dehors du cycle. Cette opération vise à maximiser à chaque étape le nombre de nœuds dans une étape. Pour le cas de réseau multi-destinations, ACH applique le même principe de fusion que SCH-m.

Ces deux heuristiques ont été comparées aux heuristiques les plus rapides de la littérature à savoir RTH-p et SCH-p. Les résultats de simulations ont montré qu'à l'inverse de RTH-p et SCH-p, SCH-m et ACH n'ont pas de notions de destinations problématiques étant donné que l'opération de fusion aboutit au traitement des destinations comme étant un seul groupe. Ces résultats montrent également que SCH-m et ACH accélèrent la migration en produisant un petit nombre d'étapes qui ne dépend ni de la taille du réseau, ni du nombre de destinations traitées. Les résultats montrent également que SCH-m et ACH diminuent le nombre de messages de contrôle avec des gains allant jusqu'à 99% par rapport à RTH-p et SCH-p. Nous nous sommes aussi intéressés au temps de calcul de chacune de ces heuristiques. Le temps de calcul de ACH dépend fortement du nombre de cycles identifiés dans l'union des deux protocoles de routage : plus il y en a, plus lente sera ACH. Quant à SCH-m, le temps de calcul est beaucoup plus long car dépend fortement du nombre de destinations traitées.

Nous nous sommes également intéressés à la congestion que pouvait générer la configuration des nœuds avec les transitions calculées par nos heuristiques. Étant donné que SCH-m et ACH utilisent l'opération de fusion pour diminuer à la fois le nombre d'étapes et le nombre de messages de contrôle nécessaires, la conséquence est un grand nombre de nœuds dans les étapes. Cela justifie que les deux heuristiques SCH-m et ACH sont susceptibles de faire plus de congestion que RTH-p et SCH-p. Nous avons proposé pour diminuer cette congestion une opération d'équilibrage des nœuds dans les étapes. Le gain en diminution de congestion atteint environ 30%.

Contexte statique distribué

Dans le contexte statique distribué, nous avons considéré que les nœuds collaborent entre eux pour savoir s'ils ont le droit de migrer ou pas. Nous avons considéré les cas particuliers du retrait et de la survenue de panne d'un nœud dans un réseau. Nous avons proposé une version distribuée de l'heuristique RTH-p [13] que nous avons appelé *Routing Tree Heuristic-Distributed version* (RTH-d), et nous avons également proposé l'heuristique *Distributed Loop-Free heuristic* (DLF).

L'heuristique RTH-d [45] fait migrer les nœuds par profondeur dans l'arbre correspondant au protocole de routage final $\mathcal{R}_f$ avec pour racine la destination considérée. Partant de la racine, lorsqu'un nœud migre, il informe ses voisins situés à la profondeur suivante qui migrent ainsi à leur tour. Les nœuds situés à la même profondeur appartiennent tous à la même étape, et, par conséquent effectuent la migration ensemble.

L'heuristique DLF [45] quant à elle exploite la propriété démontrée dans [43] selon laquelle, si les métriques restent inchangées dans un réseau, la panne d'un nœud génère uniquement des boucles de taille 2. Ainsi, un nœud n'est autorisé à migrer avec DLF que s'il n'est pas dans une boucle avec son prochain saut suivant le protocole de routage final. Nous avons supposé dans un premier temps que chaque nœud connaît *a priori* ses deux prochains sauts (suivant $\mathcal{R}_i$ et $\mathcal{R}_f$). Lorsque l'information de panne est reçue, chaque nœud vérifie par échange de messages s'il est dans une même boucle que son prochain saut suivant $\mathcal{R}_f$. Si tel est le cas, il reste bloqué jusqu'à la notification de ce dernier qui l'autorise à migrer à son tour. Dans le cas contraire, il migre directement. Nous avons supposé dans un second temps que chaque nœud connaît *a priori* uniquement son prochain saut suivant $\mathcal{R}_i$. Lorsqu'un nœud détecte la panne, il recalcule tout d'abord son nouveau prochain saut. Puis, il teste aussi par échange de messages la possibilité de migrer directement. Dans le cas d'impossibilité de migrer directement, il attend une notification l'autorisant à migrer.

Des simulations ont été faites sur des topologies réelles et aléatoires en comparant RTH-d et DLF aux heuristiques et algorithmes centralisés SCH-m, ACH [46], et GBA [43]. Les résultats ont montré qu'en termes de durée de la transition, DLF produit un nombre d'étapes aussi faible que ACH et SCH-m. Cela n'est pas le cas de RTH-d qui, malgré sa simplicité de mise en œuvre, produit toujours plus d'étapes que toutes les autres heuristiques. Le gain de DLF par rapport à RTH-d est de 99%. En termes de coût de la transition calculé en nombre de messages de contrôle nécessaires à la configuration des nœuds, RTH-d et DLF nécessitent plus de messages que ACH, SCH-m et GBA. Nous avons également introduit la métrique de vitesse de correction qui nous permet de voir pour chacune des heuristiques testées, le nombre d'étapes nécessaires pour corriger les boucles. Pour les heuristiques DLF, ACH, SCH-m, et GBA, la panne se corrige avec les derniers nœuds qui migrent tandis que pour RTH-d, la panne est corrigée bien avant la fin de la migration de tous les nœuds. Nous avons enfin implémenté nos heuristiques distribuées à l'aide des *threads* pour comparer leur temps d'exécution. Les simulations ont montré que DLF est beaucoup plus rapide que RTH-d avec un gain allant jusqu'à 95%. Cela s'explique par le fait qu'un nœud avec RTH-d est obligé d'attendre la notification de son prochain saut suivant $\mathcal{R}_f$, tandis qu'un nœud avec DLF n'est bloqué que s'il est dans une boucle avec celui-ci.

Nous avons également montré de façon formelle qu'il n'y a aucun risque d'interblocage pour DLF en faisant les hypothèses que chaque protocole de routage est sans boucle.

Contexte de mobilité

Dans le contexte de mobilité, nous avons considéré que le réseau est dynamique avec un ou plusieurs nœuds qui se déplacent suivant une distance donnée dans une zone géographique délimitée. Nous avons également considéré que les nœuds collaborent de manière distribuée pour savoir s'ils ont le droit de migrer ou pas. Nous avons ainsi proposé un protocole distribué DLF-k, une version évoluée du protocole DLF [45] qui corrige toutes les boucles de taille inférieure ou égale à k avec $k \geq 2$.

Le protocole DLF-k garde le même principe que DLF, celui de n'autoriser un nœud à migrer que s'il n'est pas dans une boucle avec son prochain suivant $\mathcal{R}_f$. DLF-k ne se limite pas cependant aux boucles de taille 2 comme c'est le cas pour DLF, mais étend la recherche aux boucles de taille inférieure ou égale k. La vérification d'un nœud n se fait de manière récursive sur un arbre binaire dont la racine est $\mathcal{R}_f(n)$, et les fils des nœuds de l'arbre, leurs deux prochains sauts suivant respectivement le protocole initial $\mathcal{R}_i$ et le protocole final $\mathcal{R}_f$. Cette vérification est stoppée lorsque soit on est en présence d'une boucle (visite d'un nœud déjà visité), soit on atteint la taille k. Nous avons considéré deux types d'applications : les applications avec un unique nœud mobile, et les applications avec un groupe de nœuds mobiles.

Les simulations avaient pour but dans un premier temps, de quantifier les boucles générées, puis dans un second temps d'évaluer les performances de DLF-k pour chacun de ces deux types d'applications.

Le nombre de boucles dépend du type d'applications, de la distance de déplacement des nœuds, et de la métrique des liens appliquée. Les applications avec un unique nœud mobile génèrent un faible nombre de boucles de routage (de l'ordre de dizaines), tandis que les applications avec un groupe de nœuds mobiles génèrent un grand nombre de boucles de routage (de l'ordre de milliers voire centaines de milliers). Cependant pour les deux types d'applications, le nombre de boucles augmente avec la distance de déplacement, et la métrique : la métrique aléatoire par exemple impacte fortement ce nombre comparativement à la métrique *hopcount*.

En termes de performances, DLF-k produit des transitions de courte durée (avec un faible nombre d'étapes en général), est rapide en temps d'exécution, devient très coûteux en termes de messages de contrôle nécessaires (croît de manière exponentielle avec la valeur de k). Cependant, DLF-k est adapté aux applications avec un unique nœud mobile, car les simulations ont montré qu'une valeur de $k = 4$ dans ce cas permettait de corriger un pourcentage de boucles de routage compris entre 97% et 100%. Pour les applications avec un groupe de nœuds mobiles, DLF-k n'est pas adapté car, les boucles générées sont de grandes tailles, et il faudrait mettre k à l'infini pour être sûr de tout corriger, ce qui nécessite un énorme coût.

Perspectives

A l'issue de la présentation de ces travaux, nous tenons à présenter des points de recherche qui nous semblent intéressants à étudier dans l'avenir.

Durant cette thèse, nous avons implémenté à l'aide du langage de programmation Java [72] toutes les fonctions nécessaires pour faire les simulations et tester nos heuristiques. L'une des perspectives est d'utiliser un simulateur réseau comme NS-3 [73], pour tester nos heuristiques. On pourrait ainsi récupérer par exemple le temps de migration des nœuds non pas en nombre d'étapes, mais en secondes.

Une autre perspective est de cibler des protocoles en particulier, tels que OSPF ou RIP, d'étudier leurs spécificités, et d'optimiser nos heuristiques ACH et DLF en fonction de ces spécificités (voire de proposer de nouvelles heuristiques). En effet, durant cette thèse, nous avons supposé que nous avions un protocole de routage initial quelconque, et un protocole de routage final quelconque. D'où l'intérêt de cibler des protocoles de routage en particulier.

Dans le cas du contexte centralisé, nous avons proposé un algorithme d'équilibrage des nœuds dans les étapes, qui permet de diminuer la congestion qui peut être générée lors de la configuration des transitions produites par les heuristiques SCH-m et ACH. En effet, ces heuristiques cherchent à accélérer le plus possible les transitions produites en réduisant au maximum le nombre d'étapes nécessaires, ce qui aboutit à une concentration des nœuds dans les premières étapes. L'algorithme d'équilibrage des nœuds redistribue les nœuds dans les étapes calculées. Cette redistribution se fait de manière gloutonne par sélection des nœuds de manière arbitraire, puis en vérifiant l'existence de boucles ou non sur un graphe intermédiaire construit, et enfin en répétant le processus jusqu'à obtention d'une transition équilibrée en termes de nombre de nœuds par étape. Il serait intéressant d'optimiser cet algorithme en affûtant les critères de sélection et de migration des nœuds de leurs étapes d'origine à de nouvelles étapes.

Dans le contexte dynamique distribué, les résultats des simulations faites ont montré que DLF-k est adapté lorsque les boucles générées sont de faible taille. Plus la valeur de k est élevée, moins DLF-k est utilisable en pratique, et notamment dans le cas de mobilité d'un groupe de nœuds. Il serait intéressant de repenser complètement un autre algorithme distribué pour ce cas précis. Les pistes de solution sont les suivantes :

- Soit de conserver l'idée de base selon laquelle un nœud ne peut migrer que s'il n'est inclus dans une même boucle avec son prochain suivant le protocole de routage final, et définir un algorithme de vérification de boucles moins onéreux,
- Soit de changer complètement d'idée de base. Une piste à explorer est par exemple de définir un protocole hiérarchique à l'instar du protocole LEACH *(Low-Energy Adaptive Clustering Hierarchy)* [38, 39] qui partitionne le réseau en zones et clusters de façon distribuée. Ensuite, les boucles de routage pourraient être traitées de manière locale à l'intérieur des clusters, avec DLF-k par exemple. Enfin, un traitement inter-cluster pourrait être appliqué pour toute boucle ayant des nœuds appartenant à des clusters différents. On pourrait ensuite étudier et comparer le lien avec d'autres protocoles tels que BGP *(Border Gateway Protocol)* [19].

Dans le contexte dynamique distribué également, il serait intéressant d'étudier

d'autres modèles de mobilité comme par exemple le cas de mobilité prévisible avec un itinéraire connu à l'avance, ou un itinéraire cyclique.

Les autres perspectives consistent à apporter des réponses à des questionnements, ou des aspects à prendre en compte :

- L'implémentation de nos heuristiques sur de vrais routeurs tels que Linksys de Cisco [74] ferait-elle naître de nouvelles problématiques (mémoire, temps de calcul, etc.) ?
- La prise en compte d'autres aspects tels que la sécurité, la localisation des nœuds, la QoS, etc, change-t-elle nos alogorithmes ?
- Comment proposer des algorithmes qui prennent en compte le *cross-layering* dans des réseaux hétérogènes par exemple ?
- Peut-on obtenir des résultats optimaux ?

Bibliographie

[1] N. El Rachkidy, A. Guitton, and M. Misson, "Avoiding routing loops in a multi-stack WSN," *JCM (Journal of Communications)*, vol. 8, no. 3, pp. 751–761, 2013. 1, 6

[2] ——, "Improving routing performance when several routing protocols are used sequentially in a WSN," in *ICC (IEEE International Conference on Communications)*, 2013, pp. 1408–1413. 1, 6

[3] J. Steffan and M. Voss, "Security mechanisms for multi-purpose sensor networks," in *GI Jahrestagung*, vol. 2, 2005, pp. 158–160. 1

[4] D. Manjunath and S. V. Gopaliah, "A multi-purpose wireless sensor network for residential layouts," in *BWCCA (International Conference on Broadband and Wireless Computing, Communication and Applications)*, 2007, pp. 49–58. 1

[5] P. Javier del Cid, "Optimizing resource use in multi-purpose WSNs," in *PerCom Workshops (Pervasive Computing Workshops)*, 2011, pp. 395–396. 1

[6] L. Sarakis, T. Zahariadis, H.-C. Leligou, and M. Dohler, "A framework for service provisioning in virtual sensor networks," *EURASIP Journal on Wireless Communications and Networking*, no. 1, p. 135, 2012. 1

[7] P. Trakadas, T. Zahariadis, S. Voliotis, P. Karkazis, T. Velivassaki, and L. Sarakis, "Routing metric selection and design for multi-purpose WSN," in *IWSSIP (International Conference on Systems, Signals and Image Processing)*, 2014, pp. 199–202. 1

[8] Z. Zhong, R. Keralapura, S. Nelakuditi, Y. Yu, J. Wang, C. N. Chuah, and S. Lee, "Avoiding transient loops through interface-specific forwarding," in *IEEE/ACM IWQoS (International Symposium on Quality of Service)*, ser. Lecture Notes in Computer Science, vol. 3552. Springer, 2005, pp. 219–232. 1, 7

[9] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Transactions on Networking*, vol. 15, no. 6, pp. 1280–1292, Dec 2007. 1, 6, 7

[10] S. Nelakuditi, Z. Zhong, J. Wang, R. Keralapura, and C. N. Chuah, "Mitigating transient loops through interface-specific forwarding," *Computer Networks*, vol. 52, no. 3, pp. 593–609, 2008. 1, 7

[11] K. Butler, T. R. Farley, P. McDaniel, and J. Rexford, "A survey of BGP security issues and solutions," *Proceedings of the IEEE*, vol. 98, no. 1, pp. 100–122, 2010. 1

[12] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring ISP topologies with Rocketfuel," *IEEE/ACM Trans. Netw.*, vol. 12, no. 1, pp. 2–16, Feb. 2004. [Online]. Available : http://dx.doi.org/10.1109/TNET.2003.822655 4, 71, 102

[13] N. El Rachkidy and A. Guitton, "Changing the routing protocol without transient loops," *Computer Communications*, 2016. 5, 43, 54, 58, 61, 68, 87, 90, 148, 149

[14] P. Faure, "L'automobile : un enjeu considérable pour la france, tant en ce qui concerne les entreprises que l'intérêt public," in *Annales des Mines-Réalités industrielles*, no. 2. ESKA, 2014, pp. 3–10. 6

[15] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, pp. 59–69, 1982. 6

[16] F. Clad, P. Merindol, J.-J. Pansiot, P. Francois, and O. Bonaventure, "Graceful convergence in link-state IP networks : A lightweight algorithm ensuring minimal operational impact," *IEEE/ACM Transactions on Networking*, vol. 22, no. 1, pp. 300–312, 2013. 6, 7

[17] F. Clad, P. Merindol, S. Vissicchio, J.-J. Pansiot, and P. François, "Graceful router updates for link-state protocols," in *IEEE ICNP (International Conference on Network Protocols)*, 2013. 6, 7

[18] J. Moy, "OSPF version 2," Internet RFC 2328, April 1998. 12, 23

[19] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)." *RFC*, vol. 4271, pp. 1–104, January 2006. [Online]. Available : http://dblp.uni-trier.de/db/journals/rfc/rfc4200-4299.html 12, 151

[20] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," RFC 3626 (Experimental), Internet Engineering Task Force, October 2003. [Online]. Available : http://www.ietf.org/rfc/rfc3626.txt 12, 29

[21] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc on-demand distance vector (AODV) routing," IETF, Request For Comments 3561, July 2003. 12, 31, 32

[22] C. Hendrik, "Routing Internet Protocol (RIP)," IETF, The Internet Society 1058, June 1988. 18

[23] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, pp. 87–90, 1958. [Online]. Available : https://doi.org/10.1090/qam/102435 18, VI

[24] J. Ford and R. Lester, "Network flow theory," The RAND Corporation, Santa Monica, California, Paper P-923, August 1956. [Online]. Available : http://www.rand.org/cgi-bin/Abstracts/ordi/getabbydoc.pl?doc=P-923 18

[25] L. Bosack, "Method and apparatus for routing communications among computer networks," http ://www.freepatentsonline.com/5088032.html, February 1992. 21

[26] D. Savage, J. Ng, S. Moore, D. Slice, P. Paluch, and R. White, "Enhanced Interior Gateway Routing Protocol (EIGRP)," RFC 7868, may 2016. [Online]. Available : https://rfc-editor.org/rfc/rfc7868.txt 21

[27] J. J. Garcia-Lunes-Aceves, "Loop-free routing using diffusing computations," *IEEE/ACM Trans. Netw.*, vol. 1, no. 1, pp. 130–141, 1993. 21

[28] S. Bradner, "The Internet engineering task force," 1999. 23

[29] E. W. Dijkstra, "A note on two problems in connexion with graphs." *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. [Online]. Available : http://gdzdoc.sub.uni-goettingen.de/sub/digbib/loader?did=D196313 23, VI

[30] A. Atlas and A. Zinin, "Basic Specification for IP Fast Reroute : Loop-Free Alternates," RFC 5286 (Proposed Standard), Internet Engineering Task Force, September 2008. [Online]. Available : http://www.ietf.org/rfc/rfc5286.txt 25, 26

[31] D. Oran, "OSI IS-IS Intra-domain Routing Protocol," RFC 1142 (Informational), Internet Engineering Task Force, February 1990. [Online]. Available : http://www.ietf.org/rfc/rfc1142.txt 26

[32] C. E. Perkins and P. Bhagwat, "Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers," in *Proceedings of the conference on Communications architectures, protocols and applications*, ser. SIGCOMM '94, no. 4. New York, NY, USA : ACM, Oct. 1994, pp. 234–244. [Online]. Available : http://dx.doi.org/10.1145/190314.190336 27

[33] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich, "Better approach to mobile ad-hoc networking," *IETF*, 2008. 30

[34] R. J. van Glabbeek, P. Höfner, W. L. Tan, and M. Portmann, "Sequence numbers do not guarantee loop freedom : AODV can yield routing loops." in *MSWiM*, B. Landfeldt, M. Aguilar-Igartua, R. Prakash, and C. Li, Eds. ACM, 2013, pp. 91–100. [Online]. Available : http://dblp.uni-trier.de/db/conf/mswim/mswim2013.htmlGlabbeekHTP13 32

[35] D. Johnson, D. Maltz, and Y.-C. Hu, "The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)," Internet Draft, Work in Progress, April 2003. 32

[36] N. Beijar, "Zone Routing Protocol (ZRP)," *Networking Laboratory, Helsinki University of Technology, Finland*, vol. 9, pp. 1–12, 2002. 34

[37] M. Jiang, "Cluster Based Routing Protocol (CBRP)," *IETF Internet-draft*, 1999. 34

[38] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *System sciences, 2000. Proceedings of the 33rd annual Hawaii international conference on.* IEEE, 2000, pp. 10–pp. 34, 151

[39] D. Culler, D. Estrin, and M. Srivastava, "Guest editors' introduction : Overview of sensor networks," *Computer*, vol. 37, no. 8, pp. 41–49, 2004. 34, 151

[40] G. Pei, M. Gerla, X. Hong, and C.-C. Chiang, "A wireless hierarchical routing protocol with group mobility," in *Wireless Communications and Networking Conference, 1999. WCNC. 1999 IEEE*, vol. 3. IEEE, 1999, pp. 1538–1542. 34

[41] B. Karp and H.-T. Kung, "GPSR : Greedy perimeter stateless routing for wireless networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking.* ACM, 2000, pp. 243–254. 34

[42] L. Vanbever, S. Vissichio, C. Pelsser, P. Francois, and O. Bonaventure, "Lossless migrations of link-state IGPs," *IEEE/ACM Transactions on Networking*, vol. 20, no. 6, pp. 1842–1855, 2012. 40, 54, 61, 68

[43] F. Clad, S. Vissicchio, P. Mérindol, P. François, and J. Pansiot, "Computing minimal update sequences for graceful router-wide reconfigurations," *IEEE/ACM Trans. Netw.*, vol. 23, no. 5, pp. 1373–1386, 2015. [Online]. Available : http://dx.doi.org/10.1109/TNET.2014.2332101 48, 49, 50, 51, 52, 54, 93, 113, 149

[44] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972. 64

[45] N. Bekono, N. El Rachkidy, and A. Guitton, "Distributed fast loop-free transition of routing protocols," *Vehicular Technology Conference*, 2017. 64, 87, 90, 93, 113, 121, 148, 149, 150

[46] ——, "Fast loop-free transition of routing protocols," *Vehicular Technology Conference*, 2016. 65, 68, 87, 148, 149

[47] Internet Topology Zoo. Last updated on 18th of July 2012. [Online]. Available : http://www.topology-zoo.org/dataset.html 71, 88, 102

[48] F. Clad, "Disruption-free routing convergence : computing minimal link-state update sequences," Ph.D. dissertation, Strasbourg, 2014. 96

[49] J. J. Blum, A. Eskandarian, and L. J. Hoffman, "Challenges of intervehicle ad hoc networks," *IEEE transactions on intelligent transportation systems*, vol. 5, no. 4, pp. 347–351, 2004. 116

[50] J. Blum, A. Eskandarian, and L. Hoffman, "Mobility management in ivc networks," in *Intelligent Vehicles Symposium, 2003. Proceedings. IEEE.* IEEE, 2003, pp. 150–155. 116

[51] P. Mühlethaler and O. Salvatori, *802.11 et les réseaux sans fil.* Eyrolles Paris, 2002. 116

[52] H. Mabrouk, F. Ben Jemaa, D. Feurer, and S. Massuel, "Utilisation des drones dans le domaine agricole en Tunisie (cas de la gestion technique d'un verger de pêcher)," in *Séminaire international « Drones et moyens légers aéroportés pour les applications géospatiales en recherche : État des lieux et perspectives »*, ENIT, Tunis, Tunisia, Novembre 2015. 116

[53] C. Zhang and J. Kovacs, "The application of small unmanned aerial systems for precision agriculture : a review," *Precision Agric*, 2012. 116

[54] M. Fournier, "La surveillance maritime en méditerranée," vol. 2010-1, pp. 147–154, 08 2010. 116

[55] R. Jobard, *Les drones : La nouvelle révolution*, ser. Serial makers. Eyrolles, 2014. [Online]. Available : https://books.google.fr/books?id=gQGOBAAAQBAJ 116

[56] D. B. Johnson, "Routing in ad hoc networks of mobile hosts," in *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, ser. WMCSA '94. Washington, DC, USA : IEEE Computer Society, 1994, pp. 158–163. [Online]. Available : http://dx.doi.org/10.1109/WMCSA.1994.33 119

[57] P. Ranjan and K. K. Ahirwar, "Comparative Study of VANET and MANET Routing Protocols," in *Proc. of the International Conference on Advanced Computing and Communication Technologies (ACCT 2011)*, 2011, pp. 517–523. 119

[58] S. Zeadally, R. Hunt, Y.-S. Chen, A. Irwin, and A. Hassan, "Vehicular ad hoc networks (VANETS) : Status, results, and challenges," vol. 50, pp. 1–25, 08 2010. 119

[59] B. Paul, M. Ibrahim, and M. A. N. Bikas, "VANET routing protocols : Pros and cons," *International Journal of Computer Applications*, vol. 20, no. 3, pp. 28–34, April 2011. 120

[60] Y. Khaled, M. Tsukada, J. Santa, J. Choi, and T. Ernst, "A usage oriented analysis of vehicular networks : from technologies to applications," *JCM*, vol. 4, pp. 357–368, 2009. 120

[61] F. Bai, T. Elbatt, G. Hollan, H. Krishnan, and V. Sadekar, "Towards characterizing and classifying communication-based automotive applications from a wireless networking perspective," in *Proceedings of IEEE Workshop on Automotive Networking and Applications (AutoNet).* San Francisco, CA, USA, 2006, pp. 1–25. 120

[62] K. Ait Ali, "Modelling and performance study in VANET networks," Theses, Université de Technologie de Belfort-Montbeliard, Nov. 2012. [Online]. Available : https://tel.archives-ouvertes.fr/tel-00827552 120

[63] R. Sengupta, S. Rezaei, S. E. Shladover, D. Cody, S. Dickey, and H. Krishnan, "Cooperative collision warning systems : Concept definition and experimental implementation," *Journal of Intelligent Transportation Systems*, vol. 11, no. 3, pp. 143–155, 2007. [Online]. Available : https://doi.org/10.1080/15472450701410452 120

[64] J. K. Kim, R. Sharman, H. R. Rao, and S. Upadhyaya, "Efficiency of critical incident management systems : Instrument development and validation," *Decision Support Systems*, vol. 44, no. 1, pp. 235 – 250, 2007. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S016792360700067X 120

[65] M. Guo, M. H. Ammar, and E. W. Zegura, "V3 : A vehicle-to-vehicle live video streaming architecture," *Pervasive and Mobile Computing*, vol. 1, no. 4, pp. 404 – 424, 2005, special Issue on PerCom 2005. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S1574119205000477 120

[66] A. Qureshi, J. Carlisle, and J. Guttag, "Tavarua : Video streaming with wwan striping," in *Proceedings of the 14th ACM International Conference on Multimedia*, ser. MM '06. New York, NY, USA : ACM, 2006, pp. 327–336. [Online]. Available : http://doi.acm.org/10.1145/1180639.1180714 120

[67] J. Ploeg, N. van de Wouw, and H. Nijmeijer, "Lp string stability of cascaded systems : Application to vehicle platooning," *IEEE Transactions on Control Systems Technology*, vol. 22, no. 2, pp. 786–793, March 2014. 120

[68] R. Lenain, B. Thuilot, C. Cariou, and P. Martinet, "High accuracy path tracking for vehicles in presence of sliding : Application to farm vehicle automatic guidance for agricultural tasks," *Auton. Robots*, vol. 21, no. 1, pp. 79–97, Aug. 2006. [Online]. Available : http://dx.doi.org/10.1007/s10514-006-7806-4 120

[69] F. Caicedo, "Real-time parking information management to reduce search time, vehicle displacement and emissions," *Transportation Research Part D : Transport and Environment*, vol. 15, no. 4, pp. 228 – 234, 2010. [Online]. Available : http://www.sciencedirect.com/science/article/pii/S1361920910000179 120

[70] J. Kurose and K. Ross, "Peer-to-peer applications," *Computer Networking. A Top-Down Approach, IV Edition, Addison Wesley*, 2007. 120

[71] J. Fletcher and S. Tobias, *Computer games and instruction.* IAP, 2011. 120

[72] J. Gosling, B. Joy, and G. Steele, *The Java language specification.* Addison-Wesley Professional, 2000. 151

[73] G. F. Riley and T. R. Henderson, "The NS-3 Network Simulator." in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Günes, and J. Gross, Eds. Springer, 2010, pp. 15–34. 151

[74] Y. Gagnon, *Linksys E3000 de Cisco [Test].* Blogue de Geek, 2011. 152

[75] A. S. Tanenbaum, *Modern operating system.* Pearson Education, Inc, 2009. VII

Troisième partie

Annexes

Annexe A

Définitions

Dans cette partie, nous donnons quelques définitions complémentaires des termes utilisés dans le manuscrit.

A.1 Réseau

Un réseau est un ensemble d'éléments reliés par des liens qui donnent l'information sur la nature de la relation entre ces éléments : un lien social, un lien temporel, un lien économique, un lien géographique, etc. Pour le cas d'un réseau informatique, il représente un ensemble d'équipements interconnectés (possiblement par des connexions) échangeant des informations numériques. Quelle que soit la nature du réseau, il est généralement modélisé à l'aide d'un graphe (cf annexe A.2).

A.2 Graphe

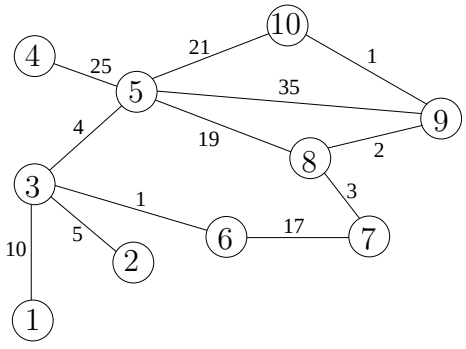

FIGURE A.1 – Exemple de graphe.

Un graphe non orienté (respectivement orienté) est un ensemble de nœuds reliés par des arêtes (respectivement arcs), et qui est généralement symbolisé par le couple $G = (V, E)$ avec V pour *Vertices* l'ensemble des nœuds, et E pour *Edges* l'ensemble de paires (respectivement couples) d'éléments de V.

Nous avons l'exemple d'un réseau de capteurs qui peut être représenté par un graphe où les capteurs sont les nœuds, et les connexions sans fil possibles entre capteurs forment l'ensemble des liens.

Un graphe peut-être valué si les liens entre nœuds ont une pondération associée, ou non dans le cas contraire. L'exemple de la figure A.1 est un exemple de graphe valué

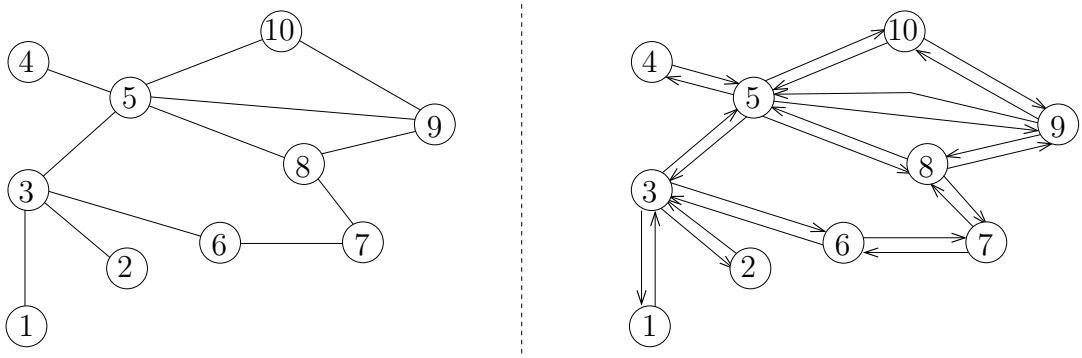

FIGURE A.2 – Exemples de graphe : (a) graphe non orienté, (b) graphe orienté.

de 10 nœuds non orienté. Formellement, un graphe orienté valué est appelé un réseau, mais dans notre cas, nous appelons réseau tout graphe valué (y compris non orienté).

Tout graphe non orienté peut être transformé en graphe orienté en transformant chaque arête $\{u, v\}$ en deux arcs symétriques (u, v) et (v, u). A titre d'exemple, la figure A.2(b) illustre la version orientée du graphe de la figure A.2(a).

A.3 Voisin

Etant donné un sommet u d'un graphe $G = (V, E)$, tout sommet $v \in V$ tel que $(u, v) \in E$ est appelé voisin de u.

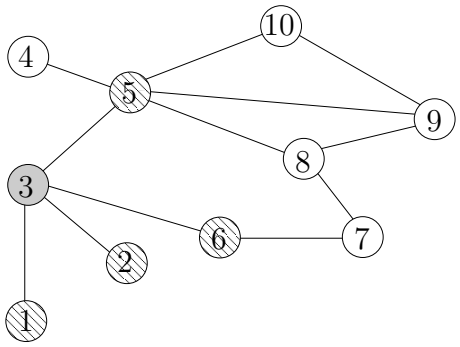

FIGURE A.3 – Exemple de voisinage : le nœud 3 dans le réseau de la figure A.1 et ses voisins $\{1, 2, 5, 6\}$.

La figure A.3 montre l'exemple du nœud 3 dans le réseau de la figure A.1 qui a pour voisins les nœuds $\{1, 2, 5, 6\}$.

A.4 Pondération

La pondération ou métrique de lien, w, est une fonction coût qui associe à une arête/arc un réel. Plus formellement, soit $G = (V, E)$ un graphe, $e \in V : w(e) = \gamma$, avec $\gamma \in \mathbb{R}$. Si G est un réseau de routeurs, $w(e)$ peut représenter l'état du lien e, son coût, sa latence, son débit, etc.

Dans l'exemple de la figure A.1, si nous considérons l'arête $e = (4, 5)$, alors $w(e) = 25$.

La pondération des liens est un élément clé dans le routage, car elle a un impact significatif dans le calcul des routes des sources vers les destinations.

A.5 Chemin

Un chemin ch (aussi appelé route) dans un graphe $G = (V, E)$ est une suite d'arêtes (arcs) consécutifs reliant une origine dite source s à une extrémité dite destination d. Soient n_0 et n_k respectivement l'origine et l'extrémité : $ch(n_0, n_k) = (n_0, n_1, ..., n_{k-1}, n_k)$ avec $n_0 = s$, $n_k = d$, et $\forall i \in [0, k-1], (n_i, n_{i+1}) \in E$.

Nous noterons tout chemin $ch = (n_0, n_1, ..., n_{k-1}, n_k)$ sous la forme $n_0 - n_1 - ... - n_{k-1} - n_k$.

Considérons l'exemple de la figure A.1. Plusieurs chemins possibles existent entre le nœud 7 et le nœud 10 :

$$\begin{cases} \text{ch(7, 10)=7-8-5-10,} \\ \text{ch(7, 10)=7-8-9-10,} \\ \text{ch(7, 10)=7-8-9-5-10,} \\ \text{ch(7, 10)=7-6-3-5-10.} \end{cases}$$

A.6 Métrique d'un chemin

La métrique d'un chemin ch est une valeur $w(ch)$ affectée à ce chemin, et qui identifie le coût associé à l'utilisation de ce dernier. Le chemin peut par exemple être évalué en termes de débit, de nombre de sauts ou de délai. $w(ch)$ est calculée comme la somme des poids des liens qui forment le chemin ch. Formellement, soit un graphe $G = (V, E)$, et $ch = (n_0, ..., n_k)$ un chemin de G : $w(ch) = \sum_{i=0}^{k-1} w(n_i, n_{i+1})$ avec $\forall i \in [0; k]$ $n_i \in V$, et $\forall i \in [0; k-1]$ $(n_i, n_{i+1}) \in ch$.

Considérons l'exemple de la figure A.1 avec pour nœud source 7 et pour nœud destination 10. Soient $ch_1 = 7-8-5-10$, $ch_2 = 7-8-9-10$, $ch_3 = 7-8-9-5-10$, et $ch_4 = 7-6-3-5-10$, les 4 chemins possibles de 7 à 10. Nous avons :

$$\begin{cases} ch_1 = w((7,8)) + w((8,5)) + w((5,10)) = 43, \\ ch_2 = w((7,8)) + w((8,9)) + w((9,10)) = 6, \\ ch_3 = w((7,8)) + w((8,9)) + w((9,5)) + w((5,10)) = 61, \\ ch_4 = w((7,6)) + w((6,3)) + w((3,5)) + w((5,10)) = 43. \end{cases}$$

Généralement, plus la métrique est faible , meilleur est le chemin en comparaison à d'autres chemins.

A.7 Plus court chemin

Calculer un plus court chemin dans un graphe revient à calculer un chemin qui minimise le coût. Soient $G = (V, E)$ le graphe qui représente le réseau, $n_0, n_k \in V$ respectivement la source et la destination. Le plus court chemin ch^* entre n_0 et n_k vérifie l'équation : $w(ch^*) = \min(w(ch(n_0, n_k)))$.

Considérons l'exemple de la figure A.1 avec pour nœud source 7 et pour nœud destination 10. Etant donné que les coûts des chemins du nœud 7 au nœud 10 sont

par ordre croissant 6, 43, 43, 61, le chemin le plus court est celui de coût 6, c'est-à-dire $ch_2 : 7 - 8 - 9 - 10$.

Les algorithmes de calcul de plus court chemin les plus connus dans un graphe sont :

— L'algorithme de Dijkstra [29], qui ne s'applique que lorsque les poids des liens sont positifs, et donne la meilleure distance d'un nœud source vers tous les autres nœuds.

— L'algorithme de Bellman-Ford [23] qui, contrairement à l'algorithme de Dijkstra, autorise la présence de certains arcs de poids négatif.

A.8 Cycle

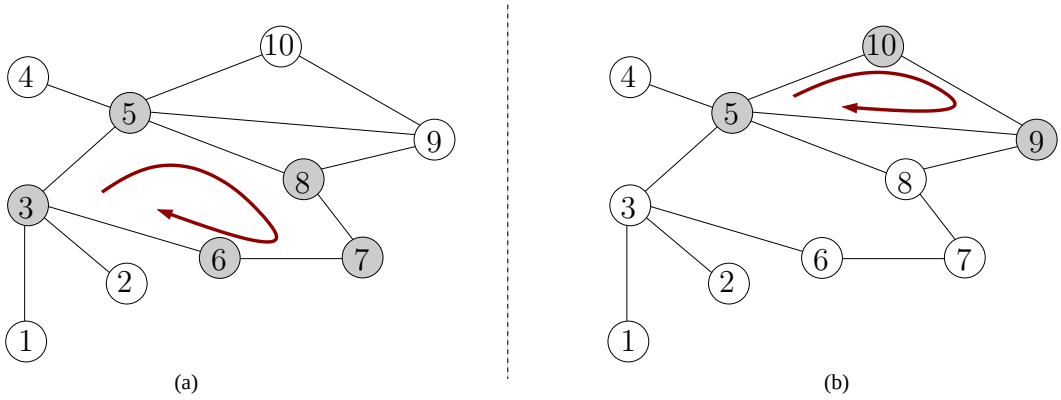

FIGURE A.4 – Exemple de cycles : (a) cycle $(3, 5, 8, 7, 6, 3)$, (b) cycle $(5, 10, 9, 5)$.

Un cycle dans un graphe est une suite d'arêtes consécutives dont les deux nœuds extrémités sont identiques. Un cycle de taille n est un cycle impliquant n nœuds. Soit un graphe $G = (V, E)$. G contient un cycle si $\exists\ ch(n_0, n_0) = (n_0, n_1, ..., n_k, n_0)$ avec $n_i \in V$, $i \in [0, k]$, $n_0 \neq n_1$ et $(n_i, n_{i+1}) \in E$.

La figure A.4 présente plusieurs cycles. Nous montrons comme exemples dans la figure A.4(a) le cycle $(3, 5, 8, 7, 6, 3)$, et dans la figure A.4(b) le cycle $(5, 10, 9, 5)$.

A.9 Composante connexe

Soit $G = (V, E)$ un graphe. Une composante connexe est un sous-graphe G' de G qui est connexe, c'est-à-dire qu'il existe toujours un chemin entre deux nœuds quelconques de G', et maximal c'est-à-dire qu'il n'est pas inclus dans une composante connexe plus grande. Formellement, $G' = (V', E')$ est une composante connexe de G (avec $V' \subseteq V, E' \subseteq E$) si et seulement si $\forall x, y \in V', \exists\ ch(x, y)$, et $\nexists G'' \neq G'$ connexe tel que $G' \subseteq G''$.

L'acronyme *SCC (Strongly Connected Component)* pour désigner une composante fortement connexe, est le terme employé dans le cas d'un graphe orienté.

La figure A.5 montre deux exemples de composantes connexes obtenus à partir du graphe de la figure A.1. Lorsqu'on le considère tel quel, c'est-à-dire non orienté, on obtient une seule composante connexe : $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, qui est constituée de tous les nœuds tel que le montre la figure A.5(a). On parle ici de graphe connexe. Lorsqu'on considère une version orientée de ce graphe, on obtient un autre exemple de composante fortement connexe : $\{3, 5, 6, 7, 8, 9, 10\}$, tel que le montre la figure A.5(b).

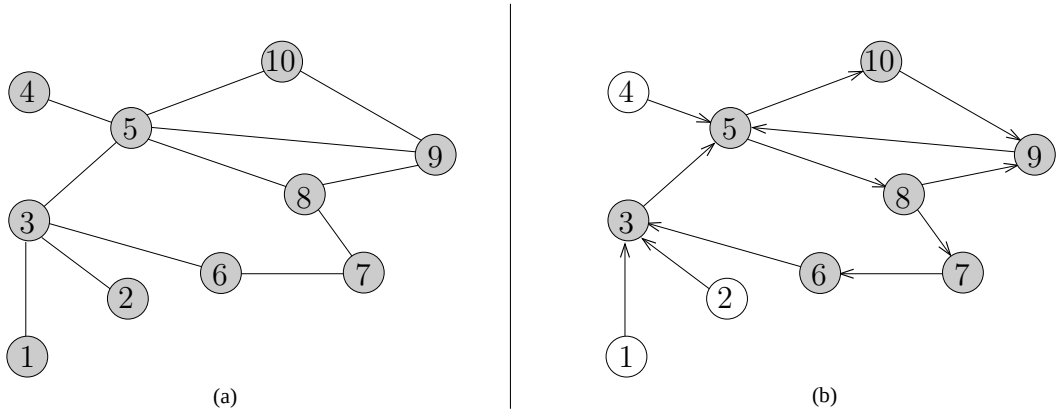

FIGURE A.5 – Exemple de composante connexe : (a) graphe non orienté, et compososante connexe $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, (b) graphe orienté, et composante fortement connexe $\{3, 5, 6, 7, 8, 9, 10\}$.

En effet, quelle que soit la paire de nœuds que l'on choisit, il existe toujours un chemin pour relier les deux nœuds. De plus, si l'on ajoute l'un des trois nœuds $1, 2$, ou 4, la composante $\{3, 5, 6, 7, 8, 9, 10\}$ n'est plus connexe car par exemple il n'existerait pas de chemin entre les nœuds 6 et 1. $\{3, 5, 6, 7, 8, 9, 10\}$ est par conséquent une composante (fortement) connexe.

A.10 *Thread*

Un *thread*, encore appelé processus léger, est une unité d'exécution autonome rattachée à un processus (programme qui s'exécute), chargée d'exécuter une partie du processus et qui comprend un compteur ordinal, des registres et une pile. Un processus en général induit une certaine lourdeur : le changement de contexte, pas ou peu de partage de mémoire, et une exécution qui dépend de l'ordonnanceur. La programmation par *thread* permet d'intégrer dans la programmation des applications les bénéfices de la programmation parallèle. Un *thread* peut exécuter des taches en parallèle avec d'autres *threads*. Avoir plusieurs *threads* en parallèle dans un processus est analogue à faire fonctionner plusieurs processus en parallèle sur un seul ordinateur [75].

A.11 Métrique *hopcount*

La métrique *hopcount* ou au nombre de sauts, correspond au nombre de nœuds intermédiaires dans un réseau par lesquels les paquets doivent passer d'une la source vers une destination. Un *hopcount* de n signifie que n nœuds intermédiaires séparent l'hôte source de l'hôte de la destination.

A.12 Métrique aléatoire

La métrique aléatoire consiste à choisir une valeur aléatoire de poids entre deux nœuds d'un réseau.

Annexe B

Lexique des abbréviations et sigles utilisés

ACH *Avoiding Cycle Heuristic*
ABR *Area Border Router*
AD *Advertised Distance*
AGBA *Adjusted Greedy Backward Algorithm*
AODV *Ad hoc On-Demand Distance Vector*
ASBR *Autonomous System Boundary Router*
BATMAN *Better Approach to Mobile Adhoc Networking*
BGP *Border Gateway Protocol*
BR *Backbone Router*
CBRP *Cluster Based Routing Protocol*
CNRS Centre National de la Recherche Scientifique
CPC *Change Prevention Conditions*
CSNP *Complete Sequence Number Packet*
DLF *Distributed Loop-Free heuristic*
DLF-*k* *Distributed Loop-Free heuristic for loop of size k*
DSDV *Dynamic Destination Sequenced Distance Vector*
DSR *Dynamic Source Routing*
DUAL *Diffusing Update Algorithm*
EGP *Exterior Protocols*
EIGRP *Enhanced Interior Gateway Routing Protocol*
FD *Feasible Distance*
FS *Feasible Successor*
GBA *Greedy Backward Algorithm*
GPSR *Greedy Perimeter Stateless Routing*
HSR *Hierarchical State Routing*
IETF *Internet Engineering Task Force*
IGP *Interior Gateway Protocols*
IGRP *Interior Gateway Routing Protocol*
IMobS3 *Innovative Mobility : Smart and Sustainable Solutions*
IP *Internet Protocol*
IR *Internal Router*
IS-IS *Intermediate System to Intermediate System*
ITZ *Internet Topology Zoo*
LABEX Laboratoire d'Excellence

LEACH *Low-Energy Adaptive Clustering Hierarchy*
LFA-FRR *Loop-Free Alternate Fast Reroute*
LIMOS Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes
LSA *Link-State Advertisement*
LSDB *Link-State Database*
LSP *Link State Packet*
MANET *Mobile Ad hoc Network*
MPR *Multi Point Relay*
MS *Multipoint Relay Selector Set*
NS-3 *Network Simulator version 3*
OLSR *Optimized Link State Routing Protocol*
OSPF *Open Shortest Path First*
PSNP *Partial Sequence Number Packet*
Qos *Quality of Service*
RIP *Routing Information Protocol*
RERR *Route ERRor*
RREP *Route REPly*
RREQ *Route REQuest*
RTH *Routing Tree Heuristic*
RTH-d *Routing Tree Heuristic-distributed version*
RTH-p *Routing Tree Heuristic with parallel changes*
SCC *Strongly Connected Component*
SCH-m *Strongly Connected Component Heuristic with merged steps*
SCH-p *Strongly Connected component Heuristic with parallel changes*
SNP *Sequence Number Packet*
SPT *Shortest Path Tree*
VANET *Vehicular Ad-hoc Network*
VIPA Véhicule Individuel Public Autonome
ZRP *Zone Routing Protocol*