Complexité en espace de l'exploration de graphes

David Ilcinkas

Orsay
N° d'ordre : 8334

THÈSE

présentée pour obtenir le grade de
Docteur en Sciences de l'Université Paris XI, Orsay
Spécialité : Informatique

—

Complexité en espace de l'exploration de graphes

David Ilcinkas

—

Soutenue le 7 juillet 2006 devant le jury composé de :

Directeur de thèse :	Pierre Fraigniaud
Rapporteurs :	Paola Flocchini et Jacques Mazoyer
Examinateurs :	Michel Habib et Joffroy Beauquier

Remerciements

La réussite de cette thèse et le plaisir que j'ai eu à travailler pendant ces trois années est le fait de nombreuses personnes et particulièrement de Pierre Fraigniaud. Durant toute cette période, Pierre a su prendre de son temps et de son énergie pour m'apprendre tous les aspects du métier de chercheur. Je veux réellement le remercier pour toute l'écoute, la disponibilité et la pédagogie dont il a fait preuve. J'ai beaucoup appris à ses côtés et c'est un réel plaisir de travailler avec lui.

Ma recherche a été rythmée tout au long de ces trois ans par de nombreuses collaborations, toutes très riches humainement et scientifiquement. Dans le désordre, merci à Reuven Cohen, Cyril Gavoille, Amos Korman, Euripides Markou, Andrzej Pelc, David Peleg, Sergio Rajsbaum, Sébastien Tixeuil et plus généralement à tous les membres des communautés AlgoTel et SIROCCO.

Travailler au LRI a été un grand plaisir. Je voudrais particulièrement remercier toutes les personnes avec qui j'ai partagé mon bureau, Moaiz, Taoufik, Wadie, Nacho, Lehuy, Frédéric, Philippe et Nicolas pour leur bonne humeur quotidienne. Merci également à tous les autres thésards de l'équipe et plus généralement aux autres membres de l'équipe GraFComm et de l'équipe Parallélisme. Un grand merci à toute l'équipe administrative, et en particulier à Martine Lelièvre, que j'ai beaucoup sollicitée.

Je remercie très chaleureusement Paola Flocchini et Jacques Mazoyer d'avoir accepté la lourde tâche de rapporteur. Je remercie également Joffroy Beauquier et Michel Habib d'avoir bien voulu faire partie de mon jury de soutenance.

La rédaction de ce mémoire a été une étape importante de mon doctorat. Merci à Pierre et à ma mère pour le lourd travail de relecture qu'ils ont tous deux effectué. Merci également aux jeunes docteurs Frédéric Dangngoc, Nathalie Mitton, Laurence Pilard et Arnaud Rivoira pour leurs nombreux conseils, ainsi qu'au un peu moins jeune docteur Sébastien Tixeuil.

Mes différentes prestations orales et écrites ont beaucoup gagné en qualité grâce aux remarques critiques de mes collègues lors des répétitions d'exposés et des relectures d'articles. Merci donc à mes coauteurs et aux thésards de l'équipe. En particulier, merci beaucoup à Pierre Fraigniaud et à Nicolas Nisse pour leur aide précieuse.

Une thèse n'est pleinement réussi que si le pot l'est aussi. A ce titre, je tiens à dire bravo à ma mère qui a préparé, cuisiné, organisé et orchestré mon pot de thèse de façon remarquable. Ce n'était pas une tâche facile et je veux encore une fois la remercier pour tous ses efforts. A ces remerciements, j'aimerais associer ma soeur, mon père, Geneviève et ses parents, qui l'ont épaulée.

Je veux remercier mes collègues, ma famille et mes amis pour leur soutien et leur affection tout au long de cette thèse. Parmi ceux que je n'ai pas encore cité, merci au groupe Ad2g, merci à Frédéric, Christophe, Martin et globalement à tous mes amis de Supélec, merci à mes amis monocyclistes, et merci à Laurent, Léonard et Christophe.

Et pour finir, merci pour tout à ma chère et tendre...

Table des matières

Introduction

Cette thèse traite de l'exploration de graphes et plus particulièrement des questions d'espace mémoire qui y sont liés. Le problème de l'exploration de graphes est le suivant : une entité mobile se déplace de sommet en sommet le long des arêtes d'un graphe dont elle ignore la structure et a pour tâche de visiter au moins une fois tous les sommets du graphe. Nous nous intéressons principalement à la quantité de mémoire que doit posséder une entité mobile pour pouvoir explorer tous les graphes ou au moins certaines familles de graphes.

Les fondements du problème

Les graphes peuvent être utilisés pour modéliser une grande variété d'environnements physiques tels que les réseaux de tuyauterie, les galeries souterraines, les réseaux routiers, etc. Le problème de l'exploration de graphes est donc important en *robotique* où les robots doivent souvent accomplir leur mission dans des environnements inconnus. Les graphes permettent également de modéliser des environnements plus abstraits comme les réseaux informatiques, la toile, etc. Les entités mobiles dans ce cas sont des *agents logiciels* mobiles. Explorer un réseau peut alors être utilisé en maintenance : l'agent mobile visite périodiquement tous les noeuds ou tous les liens du réseau pour vérifier leur bon fonctionnement.

Le problème de l'exploration de graphes possède également des fondements plus théoriques, en particulier lorsque la mémoire de l'entité mobile est limitée. Classiquement, l'entité mobile est modélisée par un automate dont on mesure la quantité de mémoire en comptant le nombre d'états. Les familles de graphes explorables par automate fini sont utilisées pour prouver certaines équivalences entre des types de *logiques* et des classes de langages. Grossièrement, la classe des langages décrit par la logique du premier ordre à laquelle on ajoute un opérateur de fermeture transitive pour les formules d'arité au plus k est équivalente à la classe des langages reconnaissables par un automate à k têtes pouvant également utiliser quelques marqueurs de sommets.

L'exploration de graphes par des automates de mémoire limitée est enfin centrale en *théorie de la complexité*. Ce problème est en effet intimement lié au problème de décision suivant, appelé USTCON : étant donné un graphe non orienté G et deux sommets s et t de ce graphe, existe-t-il un chemin dans G allant de s à t ? Les chercheurs ont longtemps essayé de trouver un algorithme déterministe résolvant ce problème en espace logarithmique. S'il

existe un automate ayant peu de mémoire capable d'explorer tout graphe, quel que soit le sommet de départ, alors on peut utiliser cet automate pour résoudre le problème USTCON. Cette remarque a motivé un grand nombre de travaux sur le problème de l'exploration de graphes et en particulier sur sa complexité en espace.

Quelques faits marquants de la théorie de l'exploration

Le problème de l'exploration a d'abord été étudié dans le cas restreint des labyrinthes, qui sont des sous-graphes de la grille de dimension 2. Dans les labyrinthes, l'automate dispose d'une boussole qui lui indique dans quelles directions vont les arêtes incidentes à sa position courante. Budach [Bud78] a montré qu'aucun automate fini n'est capable d'explorer tous les labyrinthes. Par conséquent, aucun automate fini n'est capable d'explorer tous les graphes. Cependant, l'exploration des labyrinthes et celle des graphes arbitraires n'ont pas la même difficulté. En effet, Blum et Kozen [BK78] ont montré que deux automates finis coopérant lorsqu'ils se rencontrent peuvent explorer tous les labyrinthes. Ce n'est pas le cas pour les graphes arbitraires car aucune équipe finie d'automates finis ne peut explorer tous les graphes [Rol80]. Le résultat reste vrai même si les automates communiquent en permanence et même s'ils peuvent se rejoindre par téléportation [CR80].

Les nombreux travaux motivés par le problème USTCON et sa résolution en espace logarithmique ont introduit deux types de suites : les UTS [AKL$^+$79] et les UXS [Kou02]. Ces suites sont des séquences de directions absolues (UTS) ou relatives (UXS) définissant un parcours dans le graphe devant visiter tous les sommets. De nombreuses bornes inférieures et supérieures ont été démontrées sur la mémoire nécessaire à la construction de ces suites. Finalement, Reingold [Rei05] a présenté un algorithme déterministe résolvant USTCON en espace logarithmique. Une conséquence de sa preuve est l'existence d'une UXS et d'un automate de mémoire logarithmique pour l'exploration de graphes. Une borne inférieure correspondante avait été montré par Rollik dans [Rol80].

Contributions de la thèse

Dans cette thèse, nous poursuivons l'étude de la complexité en espace de l'exploration de graphes. Nous considérons deux aspects différents du problème. Dans la première partie, nous étudions l'exploration de graphe par un automate n'ayant aucune information sur le graphe exploré. Dans la seconde partie, l'automate est aidé de diverses manières dans sa tâche par un oracle. Ces deux parties sont précédées d'un chapitre préliminaire (chapitre 1). Ce chapitre commence par développer plus en détail les trois motivations du problème, à savoir : la robotique et les agents logiciels, la logique, et la théorie de la complexité. Il définit ensuite formellement le problème de l'exploration de graphes, ainsi que les modèles des entités mobiles devant réaliser l'exploration. La troisième et dernière partie du chapitre consiste en un état de l'art sur le problème de l'exploration et les problèmes liés à celui-ci.

La première partie de la thèse s'intéresse à la quantité de mémoire nécessaire et suffisante à un automate pour explorer une famille donnée de graphes. Dans cette partie, l'automate ne possède aucune information sur le graphe à explorer.

Le chapitre 2 porte sur l'exploration des graphes non orientés. Nous prouvons que $\Theta(D \log d)$ bits de mémoire sont nécessaires et suffisants à un automate pour explorer tous les graphes de diamètre au plus D et de degré maximum borné par d. Par ailleurs, nous améliorons la meilleure borne inférieure connue dans le cas d'une équipe finie d'automates finis. Ce résultat nous permet de montrer que la borne inférieure $\Omega(\log n)$ bits pour l'exploration des graphes d'au plus n sommets prouvée par Rollik dans le cas d'un automate simple est aussi valable pour un automate disposant d'un caillou. (Un caillou est un marqueur de sommets que l'automate peut déposer et reprendre à sa guise lors de ses déplacements dans le graphe.)

Le chapitre 3 étudie l'exploration (avec arrêt) dans les graphes orientés. Nous montrons que $\Omega(n \log d)$ bits de mémoire sont nécessaires pour réaliser l'exploration avec arrêt dans les graphes d'au plus n sommets et de degré sortant au plus d. Ce résultat reste valable même si l'automate dispose d'un nombre linéaire de cailloux. Nous décrivons ensuite un algorithme d'exploration avec arrêt permettant d'obtenir une borne supérieure seulement distante d'un facteur logarithmique de notre borne inférieure dans les graphes de degré sortant borné. Cet algorithme utilise un seul caillou et s'exécute en temps exponentiel, ce qui est inhérent car Bender et al. [BFR$^+$02] ont montré que $\Omega(\log \log n)$ cailloux sont nécessaires pour rendre possible l'exploration des graphes orientés anonymes en temps polynomial. Nous décrivons finalement un autre algorithme, utilisant $O(\log \log n)$ cailloux, s'exécutant en temps polynomial et utilisant $O(n^2 d \log(nd))$ bits de mémoire, soit seulement n fois plus que notre premier algorithme. Notre deuxième algorithme est en fait une version fortement remaniée de l'algorithme présenté dans [BFR$^+$02].

Le troisième et dernier chapitre de la première partie (Chapitre 4) s'intéresse à l'impact de la quantité de mémoire des automates sur leur capacité d'exploration. En particulier, est-ce que toute augmentation du nombre d'états permet d'explorer strictement plus de graphes ? Nous donnons une réponse partielle à cette question en montrant qu'une augmentation polynomiale du nombre d'états permet d'obtenir un gain strict en terme de capacité d'exploration. Nous prouvons également que, pour les automates ayant peu d'états, l'ajout d'un seul état résulte en la capacité d'explorer plus de graphes.

La deuxième partie de la thèse s'intéresse à différents moyens d'aider un automate à explorer tous les graphes. En particulier, nous introduisons une nouvelle méthode d'étude pour les problèmes de calculs distribués et les problèmes utilisant des agents mobiles. La connaissance qu'ont les nœuds ou les agents mobiles sur la topologie du réseau a souvent des conséquences sur l'existence ou l'efficacité de solutions à ces problèmes. Dans la littérature, cette connaissance porte sur des informations particulières telles que le nombre de nœuds, le diamètre du graphe, etc. Nous introduisons la notion d'*oracle* pour modéliser la connaissance donnée aux nœuds ou aux agents d'une manière plus générale. Notre approche permet ainsi de répondre à des questions *quantitatives* sur les connaissances nécessaires à la résolution d'un problème, indépendamment du type de cette connaissance. Nous illustrons cette nouvelle approche par le problème de l'exploration de graphes,

à travers les trois chapitres de cette deuxième partie.

Dans le chapitre 5, nous considérons le problème de l'exploration d'arbres en se concentrant sur le temps d'exploration, et non plus sur la mémoire de l'automate. Nous définissons le rapport compétitif d'un algorithme comme le rapport entre le temps d'exécution de l'algorithme et la longueur du plus court chemin visitant tous les sommets, maximisé sur tous les arbres, toutes les orientations locales et tous les sommets de départ. Dessmark et Pelc [DP04] ont montré qu'un parcours en profondeur d'abord (DFS) a un rapport compétitif de 2 et qu'aucun algorithme ne peut le battre sans disposer a priori d'informations sur l'arbre. Par contre, dans le cas où l'algorithme dispose d'une carte non étiquetée de l'arbre exploré, il est possible de faire mieux que le DFS. En utilisant la notion d'oracle, nous prouvons que le seuil sur la quantité d'informations sur la topologie que doit posséder un algorithme pour pouvoir obtenir un rapport compétitif strictement plus petit que 2 est approximativement $\log \log D$ bits, où D est le diamètre de l'arbre.

Dans le chapitre 6, nous étudions quelle quantité d'informations l'oracle doit donner aux sommet des graphes pour qu'il existe un automate fini capable d'explorer tous les graphes ainsi étiquetés. Dans ce cadre, l'oracle est vu comme un schéma d'étiquetage. Nous montrons que 2 bits (en fait 3 valeurs) par sommet sont suffisants pour atteindre cet objectif. Nous prouvons également qu'un seul bit (deux valeurs) suffit si on se restreint aux graphes de degré borné. Rappelons qu'une seule valeur d'étiquette est impossible puisqu'il n'existe aucun automate fini explorant tous les graphes.

Le chapitre 7 étudie l'exploration périodique (tous les sommets doivent être visités périodiquement) lorsque l'orientation locale est choisie pour aider l'automate. L'orientation locale d'un graphe est l'ensemble des numéros de port qui permettent à l'automate de distinguer les différentes arêtes incidentes au sommet sur lequel il se trouve. Habituellement, un automate est dit explorer un graphe s'il réussit à visiter tous les sommets du graphe indépendamment du sommet de départ et de l'orientation locale. Dans ce chapitre, nous montrons qu'il existe un algorithme choisissant l'orientation locale de telle manière qu'il existe un automate fini capable d'explorer tous les graphes munis d'une telle orientation locale. Ce résultat était déjà connu mais nous montrons également que l'exploration ainsi réalisée peut être rapide (au plus $4n$ traversées d'arêtes dans les graphes de taille n).

Enfin, nous terminons ce document par un chapitre de conclusion donnant quelques perspectives de recherche.

Chapitre 1

Préliminaires

Ce chapitre comporte trois sections qui permettent de fixer le contexte des résultats énoncés dans ce mémoire. La section 1.1 décrit les trois grandes motivations du problème étudié, à savoir l'exploration de graphes. La section 1.2 définit en détail tout le formalisme utilisé dans ce document : les notations utilisées, les modèles de graphes, les modèles d'entités mobiles et les définitions des différents types d'exploration. Enfin, la section 1.3 présente un vaste état de l'art des problèmes liés à l'exploration.

1.1 Motivations

Dans cette section, nous présentons les trois princpales motivations de l'exploration de graphes, d'abord la motivation pratique avec la robotique et les agents logiciels, puis deux motivations issues de l'informatique théorique : la logique et la théorie de la complexité.

1.1.1 Robotique et agents logiciels

La motivation la plus naturelle de l'exploration de graphes par agents mobiles est bien entendu la robotique. On s'intéresse ici au cas où le robot est une entité physique mobile et automatisée, c'est-à-dire une machine dont les décisions proviennent d'un programme embarqué et non d'ordres donnés par des humains, par exemple à l'aide d'une télécommande.

Un des principaux problèmes que rencontrent les roboticiens est la *localisation*. Le robot doit être capable de se repérer dans son environnement afin de remplir sa mission (cartographie, secours, accueil, etc.). Deux principales techniques sont utilisées. La première est l'*odométrie*, qui consiste à mesurer le déplacement du robot grâce à des capteurs internes. On peut citer, par exemple, les accéléromètres et les capteurs permettant de mesurer la rotation des roues. La seconde est l'utilisation de *repères* choisis dans l'environnement et détectés grâce à divers capteurs, généralement optiques.

Ces deux méthodes ont chacune leurs défauts. L'odométrie fonctionne mal lorsque le sol est très irrégulier, les erreurs pouvant de plus s'accumuler au cours du temps. Les capteurs externes peuvent avoir des difficultés à percevoir l'environnement, notamment

à cause de certaines propriétés de la surface des obstacles. Une solution pour essayer de compenser ces problèmes consiste à utiliser deux robots. Les deux robots disposent de moyens fiables pour évaluer leur position respective (cibles et capteurs spécifiques aux cibles) et se déplacent à tour de rôle, permettant d'avoir en permanence une connaissance relativement fiable de leur position absolue (cf [RDM01, Rek03]).

Une fois les problèmes de localisation à peu près réglés, la problématique se réduit à l'exploration d'environnements inconnus sous l'hypothèse de capteurs parfaits. De nombreux travaux étudient le cas de pièces en dimension 2, modélisées par des polygones simples. Plusieurs types d'obstacles sont considérés : aucun obstacle [HIKK01, Kle94], obstacles carrés [PY91] ou rectangulaires [AKS02, BBFY94], ou polygones simples [DKP98] (pour plus de détails, se reporter à [Isl01, RKSI93]. Il est possible de modéliser l'environnement de façon plus synthétique par des graphes tout en conservant un intérêt pour les roboticiens. Voir notamment à ce propos un article de Rekleitis, Dudek et Milios [RDM00] dans lequel les auteurs utilisent la modélisation de l'environnement par des graphes pour concevoir des algorithmes d'exploration et de cartographie qui sont utilisés par leurs robots physiques dans leur laboratoire.

Une autre motivation naturelle du problème de l'exploration est le domaine des réseaux informatiques, et particulièrement des agents logiciels mobiles. Les réseaux d'ordinateurs sont classiquement modélisés par des graphes, éventuellement orientés. Des agents logiciels peuvent être utilisés dans ces réseaux, notamment pour la maintenance. En effet l'exploration périodique du réseau s'avère utile pour la maintenance de celui-ci. Chaque nœud et chaque lien du réseau sont ainsi vérifiés périodiquement. Les agents logiciels sont aussi utilisés pour assurer des services comme la recherche d'informations. L'utilisation d'un agent logiciel explorant le réseau permet d'éviter par exemple une inondation, coûteuse en bande passante. Une liste détaillée des utilisations d'agents logiciels mobiles peut être trouvée dans [PK98].

Pour finir, notons que dans le cas des robots physiques comme dans le cas des agents logiciels, la mémoire est une ressource généralement limitée. Les robots physiques sont souvent de petites entités d'autonomie limitée. L'encombrement est souvent un critère qu'il est important de minimiser. Il est donc préférable de ne pas utiliser de puces spécifiques pour la mémoire mais d'utiliser les ressouces en mémoire du microcontrôleur qui commande le robot. Des raisons de coût ou de consommation énergétique expliquent aussi les contraintes de conception qui peuvent amener à une minimisation de la mémoire. Concernant les agents logiciels, ceux-ci sont hébergés et exécutés sur les ordinateurs hôtes lors de leur déplacement. Il est important que la quantité de mémoire soit faible pour ne pas gêner le fonctionnement de l'ordinateur. Par ailleurs, les réseaux informatiques peuvent être de très grande taille, ce qui rend le stockage complet par les agents de la carte du réseau difficilement envisageable.

1.1.2 Lien avec la logique

Dans cette section, nous allons décrire succinctement en quoi l'exploration de graphes est liée à certaines classes de logique.

Etant donné un alphabet fini Σ, on note Σ^* l'ensemble des mots de taille finie écrits avec des lettres de Σ (incluant le mot vide). Tout sous-ensemble de Σ^* est appelé un langage. Un résultat classique de la théorie des langages formels montre qu'une certaine classe, la classe des langages *réguliers*, peut être caractérisée de deux manières équivalentes. D'une part, la classe des langages réguliers est exactement la classe des langages que l'on peut décrire par des formules de la logique du second ordre monadique. D'autre part, la classe des langages réguliers est exactement la classe des langages reconnaissables par automates finis (déterministes ou non).

Il est possible d'étendre ces notions de langages sur les mots aux langages sur les arbres. Dans ce cas, chaque lettre de l'alphabet a un rang, son *arité*. A partir de cet alphabet avec rangs, on définit des arbres enracinés dont chaque sommet est étiqueté par une lettre de l'alphabet. Plusieurs sommets peuvent être étiquetés par la même lettre. Ils ont alors exactement le même nombre de fils : l'arité de la lettre. De la même façon que pour les mots, la classe des langages réguliers (pour les arbres) peut être caractérisée, de façon équivalente, d'une part par la logique du second ordre monadique, d'autre part par les automates finis de type *bottom-up* (déterministes ou non). Un automate d'arbres bottom-up parcourt les arbres des feuilles jusqu'aux racines. Un tel automate fonctionne donc en parallèle sur toutes les feuilles et remonte dans l'arbre. On peut définir un nouveau type d'automate qui agit de manière séquentielle sur l'arbre en se déplaçant de sommet en sommet le long des arêtes. Ces automates sont moins puissants que les automates bottom-up (ils reconnaisssent moins de langages).

Des automates plus puissants, mais toujours séquentiels, ont donc été introduits pour permettre la caractérisation de plus grandes classes de langage. En particulier, on définit $\mathbf{dPW}^k\mathbf{A}$, pour "deterministic pebble walking automaton with k heads", comme la classe des langages reconnaissables par les automates finis déterministes à k têtes pouvant utiliser des *cailloux* de façon imbriquée. Un caillou est un marqueur que l'automate (une de ses têtes) peut déposer et reprendre sur les sommets. L'utilisation de cailloux dans ce contexte est différente de leur utilisation dans le cadre de l'exploration de graphes, que nous verrons plus tard. Ici les cailloux sont gérés comme une file, c'est-à-dire que l'automate ne peut reprendre un caillou déposé sur un sommet que s'il s'agit du dernier qu'il a déposé (utilisation imbriquée). Par ailleurs, l'automate est autorisé à reprendre un caillou posé sur un sommet de l'arbre même si aucune tête n'est située sur ce sommet. On appelle **dPWA** l'union des classes $\mathbf{dPW}^k\mathbf{A}$, pour $k \geq 1$. Pour les mots, comme pour les arbres, la classe **dPWA** coïncide avec la classe $\mathbf{DSPACE}(\log n)$ des langages reconnaissables par une machine de Turing déterministe utilisant un espace logarithmique.

En logique, il existe un opérateur de fermeture transitive déterministe qui s'applique à un certain type de formules, les prédicats fonctionnels. Soit $\overline{u}$ et $\overline{v}$ des k-uplets de variables, correspondant à des k-uplets de sommets. Etant donné une formule $\phi(\overline{u}, \overline{v})$, la fermeture transitive appliquée à ϕ donne une formule ϕ^*. Informellement, $\phi^*(\overline{x}, \overline{y})$ signifie que l'on peut effectuer une série de sauts des sommets $\overline{x}$ aux sommets $\overline{y}$ de telle façon que chaque paire $(\overline{x'}, \overline{y'})$ de k-uplets consécutifs connectés par un saut vérifie $\phi(\overline{x'}, \overline{y'})$. On dit que ϕ est d'arité k. La classe des langages décrits par la logique du premier ordre augmentée de la fermeture transitive déterministe des formules d'arité au plus k est notée $\mathbf{FO{+}dTC}^k$,

pour "first order logic with k-ary deterministic transitive closure". L'union de ces classes pour $k \geq 1$ est notée **FO+dTC**. Cette classe coïncide avec la classe **DSPACE**$(\log n)$ et est donc égale à la classe **dPWA**. Ce résultat a été prouvé par Immerman [Imm87].

Engelfriet et Hoogeboom [EH06] ont montré que, pour tout $k \geq 1$, $\mathbf{dPW}^k\mathbf{A} = \mathbf{FO{+}dTC}^k$. Ce résultat est valable pour les arbres et donc pour les mots. En fait, ce résultat est extensible aux graphes sous certaines conditions. Les auteurs ont d'abord étendu les définitions de langages d'arbres aux langages de graphes. Ils ont procédé de même avec la logique et les automates. Ensuite, ils ont prouvé que pour toute famille de graphes, et pour $k \geq 1$, $\mathbf{dPW}^k\mathbf{A} \subseteq \mathbf{FO{+}dTC}^k$. L'inclusion inverse n'est prouvée que pour les familles de graphes explorables. Dans [EH06], une famille de graphes est dite explorable s'il existe un automate fini déterministe, muni de cailloux imbriqués, capable d'explorer les graphes de la famille. Plus précisément, pour tout graphe de la famille et pour tout sommet de départ dans le graphe, l'automate doit passer au moins une fois par chaque sommet puis s'arrêter. Pour les familles explorables, on a donc encore $\mathbf{dPW}^k\mathbf{A} = \mathbf{FO{+}dTC}^k$. Engelfriet et Hoogeboom introduisent également la notion de famille k-explorables pour les familles dont les graphes peuvent être explorés par un automate fini déterministe à k têtes utilisant des cailloux imbriqués. Alors, pour tout $k \geq 1$, et pour toute famille k-explorable, on a $\mathbf{dPW}^k\mathbf{A} = \mathbf{FO{+}dTC}^k$. Le problème de l'exploration de graphes par automates finis est donc central dans l'étude des classes de langages de graphes. Notons que tous ces résultats ont aussi été montrés pour les classes équivalentes en non déterministe.

1.1.3 Théorie de la complexité

La théorie de la complexité utilise un modèle assez simple pour modéliser un ordinateur exécutant un algorithme : la machine de Turing. Il existe principalement deux variantes. D'une part, la machine de Turing déterministe pour laquelle l'exécution est complètement déterminée lorsque l'entrée et le programme sont fixés. D'autre part, la machine de Turing non déterministe pour laquelle plusieurs exécutions sont possibles : à partir d'une configuration donnée (description complète de l'état courant) plusieurs configurations sont éventuellement atteignables par une action élémentaire de la machine de Turing.

Pour un problème donné, il est intéressant de trouver un algorithme efficace, c'est-à-dire qui s'exécute à la fois en un temps raisonnable et en utilisant peu de mémoire. On peut classer les problèmes suivant leurs performances par rapport à ces deux paramètres : le temps et la mémoire. **P**, resp. **NP**, est la classe des problèmes qu'une machine de Turing déterministe, resp. non déterministe, peut résoudre en temps polynomial. On a bien sûr $\mathbf{P} \subseteq \mathbf{NP}$. Sans doute l'un des plus grands défis de l'informatique théorique est de déterminer si cette inclusion est stricte ou non. Concernant l'espace, des classes équivalentes peuvent être définies. **L**, resp. **NL**, est la classe des problèmes qu'une machine de Turing déterministe, resp. non déterministe, peut résoudre en espace logarithmique. De même, on a $\mathbf{L} \subseteq \mathbf{NL}$, et savoir si ces deux classes sont égales ou non est également un problème ouvert. En fait, on a $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP}$ et aucune de ces inclusions n'est

connue comme étant stricte ou non.

Le problème de l'exploration est liée à deux problèmes classiques : STCON (pour "s,t-connectivity") et USTCON (pour "undirected s,t-connectivity"). Le problème STCON, resp. USTCON, est le problème de décision suivant. Étant donné un graphe orienté, resp. non orienté, G et deux sommets s et t, existe-t-il un chemin (orienté) de s à t dans G ? Le problème STCON est complet pour la classe **NL**, c'est-à-dire que ce problème est dans **NL** et qu'il est au moins aussi difficile que tout problème de **NL**. De la même façon, le problème USTCON est complet dans la classe **SL**. Cette classe, introduite pour USTCON par Lewis et Papadimitriou [LP82], est la classe des problèmes qu'une machine de Turing *symétrique* non déterministe peut résoudre en espace logarithmique. Une machine de Turing est symétrique si la relation "une configuration mène à une autre" est symétrique. Autrement dit, s'il est possible de passer de la configuration C_1 à la configuration C_2 par une action élémentaire de la machine, alors on peut aussi passer de la configuration C_2 à la configuration C_1 par une action élémentaire.

On a $\mathbf{L} \subseteq \mathbf{SL} \subseteq \mathbf{NL}$. Comme USTCON est complet pour **SL**, il suffit de trouver un algorithme déterministe pour USTCON de complexité logarithmique en espace pour prouver $\mathbf{SL} = \mathbf{L}$. Supposons qu'il existe un automate fini déterministe capable d'explorer tous les graphes. Considérons une instance (G, s, t) du problème USTCON. Alors il est possible de simuler cet automate universel par une machine de Turing déterministe en mémorisant la position de l'automate avec $\log n$ bits, où n est le nombre de sommets de G. Il existe un chemin de s à t si et seulement si l'automate universel, partant de s, explore t. Ainsi, s'il existe une machine exploratrice déterministe de taille constante capable d'explorer tous les graphes, alors on prouve $\mathbf{SL} = \mathbf{L}$. Malheureusement, une des plus puissantes machines de ce type, le "Jumping Automaton for Graphs" (JAG), n'est pas capable d'explorer tous les graphes. Ce résultat est dû aux inventeurs du JAG, Cook et Rackoff [CR80].

Il est cependant possible de fournir une mémoire logarithmique à ces machines exploratrices sans invalider l'argument permettant d'en déduire $\mathbf{SL} = \mathbf{L}$. Cette remarque a donné lieu en particulier à l'étude des suites universelles de numéros d'arêtes, appelées UTS et UXS (pour, respectivement, "Universal Traversal Sequence" et "Universal eXploration Sequence"). Le but est de construire, si possible en espace logarithmique, une suite de directions relatives (UXS) ou absolues (UTS) permettant d'explorer tous les graphes réguliers de taille et de degré fixés. Ces suites ont été utilisées dans [Rei05] en conjonction avec d'autres outils de la théorie des graphes pour construire un algorithme déterministe en espace logarithmique pour USTCON, prouvant finalement $\mathbf{SL} = \mathbf{L}$.

1.2 Définitions et modèles

Dans cette section, nous décrivons le formalisme qui sera utilisé dans le document. Nous commençons par fixer quelques notations concernant les graphes, puis nous décrivons les différents types d'étiquettes qu'auront les sommets et/ou les arêtes des graphes explorés. Nous définissons ensuite une longue série de modèles d'entités mobiles voués à l'exploration de graphe. Enfin, nous terminons la section par la définition des différents types d'exploration de graphes.

1.2.1 Formalisme

Dans cette thèse, nous utilisons le formalisme classique des graphes orientés et non orientés. Nous ne précisons ici que certaines notations ou définitions qui pourraient prêter à confusion.

Soit G un graphe non orienté. On notera $V(G)$, resp. $E(G)$, l'ensemble des sommets, resp. l'ensemble des arêtes, du graphe G. Soit G' un graphe orienté. On notera $V(G')$, resp. $A(G')$, l'ensemble des sommets, resp. l'ensemble des arcs, du graphe G'. Dans les deux cas, n désignera le nombre de sommets et m désignera le nombre d'arêtes ou d'arcs.

Soit G un graphe non orienté. Soient u et v deux de ses sommets. La distance de u à v dans G, notée $\text{dist}_G(u, v)$ ou plus simplement $\text{dist}(u, v)$, est la longueur du plus court chemin de u à v. L'excentricité de u, notée $\text{exc}(u)$, est la plus grande distance $\text{dist}(u, v)$, où v parcourt l'ensemble des sommets du graphe. Enfin, le diamètre du graphe, noté D, est la plus grande des excentricités, pour tous les sommets du graphe.

Un chemin ou un cycle est élémentaire s'il ne contient pas deux fois le même sommet. Un chemin est fermé si ses deux extrémités correspondent au même sommet.

Étant donné un sommet u d'un graphe non orienté, on note $\deg(u)$ son degré. On utilisera souvent la notation d_u voire seulement d lorsqu'il n'y aura pas d'ambiguïté. Pour un sommet u d'un graphe

orienté, on note $\deg^+(u)$ son degré sortant et $\deg^-(u)$ son degré entrant. Dans la plupart des cas, on ne s'intéressera qu'au degré sortant, et on le notera alors d_u voire seulement d lorsqu'il n'y aura pas d'ambiguïté. Si (u, v) est un arc, on dit que u est un voisin entrant de v, et que v est un voisin sortant de u.

On notera $\mathbb{Z}_d$ l'anneau $\mathbb{Z}/d\mathbb{Z}$, c'est-à-dire l'ensemble $\{0, \ldots, d-1\}$ muni de l'addition et de la multiplication modulo d.

1.2.2 Etiquetage des sommets et des arêtes

Nous considérons le problème de l'exploration de graphes dans lequel une entité mobile doit explorer tous les sommets d'un graphe arbitraire. Cette entité mobile doit donc pouvoir différencier les arêtes incidentes au sommet courant. Dans le cas contraire, l'exploration d'une étoile à trois branches serait impossible. En effet, après avoir exploré deux branches et en revenant sur le sommet central, l'entité mobile ne pourrait pas distinguer l'arête déjà explorée de l'arête menant à la dernière feuille.

Soit G un graphe. Fixons u un quelconque de ses sommets. Toute arête e incidente à ce sommet est identifié par un numéro de port, noté $\text{port}_u(e)$. Les numéros de port des arêtes incidentes au sommet u sont deux à deux distincts et appartiennent à l'ensemble $\{0, \ldots, \deg(u) - 1\}$. Il n'existe a priori aucune relation entre les deux numéros de port $\text{port}_u(e)$ et $\text{port}_v(e)$ d'une arête $e = \{u, v\}$.

Définition 1 (Orientation locale)
On appelle orientation locale*, l'ensemble des numéros de ports d'un graphe.*

Définition 2 (Graphe à arêtes colorées)
On appelle graphe à arêtes colorées *un graphe dont l'orientation locale vérifie la propriété suivante : pour toute arête $e = \{u, v\}$, on a* $\text{port}_u(e) = \text{port}_v(e)$.

Notons que tout graphe n'admet pas nécessairement une orientation locale telle que le graphe soit à arêtes colorées (cf. le théorème de Vizing sur l'arête-coloriabilité). Par ailleurs, tout graphe dont le degré maximal est atteint en un unique sommet n'admet aucune telle orientation locale.

Définition 3 (Graphe d-homogène)
On appelle graphe d-homogène *un graphe d-régulier à arêtes colorés.*

On définit l'orientation locale d'un graphe orienté de façon similaire. La différence est que seuls les arcs sortants sont dotés de numéros de port. Plus précisément, pour un arc $e = (u, v)$, le numéro de port en u, noté $\text{port}_u(e)$ est compris entre 0 et $\deg^+(u) - 1$. L'arc e n'a pas de numéro de port en v.

Définition 4 (Graphe anonyme)
Un graphe anonyme *est un graphe dont les sommets ne portent aucune marque distinctive, c'est-à-dire aucun identifiant ni aucune étiquette.*

Dans le chapitre 6, les sommets du graphe disposeront de petites étiquettes, éventuellement de taille constante. Ces étiquettes, ou couleurs, serviront à guider l'exploration d'un automate fini.

Définition 5 (Graphe coloré)
Un graphe coloré *est un graphe dont les sommets sont marqués d'une étiquette, ou couleur. Deux sommets différents (éventuellement voisins) peuvent avoir la même couleur.*

Enfin, les sommets peuvent parfois être identifiés de manière unique. Si le robot est capable de percevoir ces identifiants, nous utilisons le modèle suivant.

Définition 6 (Graphe étiqueté)
Un graphe étiqueté *est un graphe dont les sommets sont identifiés de manière univoque par une étiquette. Par exemple, ces identifiants peuvent être les nombres de 1 à n, où n est le nombre de sommets du graphe.*

1.2.3 Modèles de robot

Il existe une multitude de modèles différents d'entités mobiles évoluant dans un graphe. Certains sont équivalents, d'autres sont incomparables et enfin certains ont des pouvoirs d'expressivité strictement supérieurs à d'autres.

Nous ne considérons ici que des entités mobiles déterministes et synchrones. A un instant t, toutes les entités mobiles sont sur des sommets du graphe. Les entités mobiles agissent pendant l'étape t, c'est-à-dire entre l'instant t et l'instant $t + 1$. On ne considère dans un premier temps que les graphes non orientés anonymes.

Nous commençons la zoologie des entités mobiles par l'automate de Moore fini.

Définition 7 (Automate de Moore)
Un automate de Moore fini *est un quintuplet* $(\mathcal{S}, s_{init}, \mathcal{F}, \delta, \lambda)$.
- $\mathcal{S}$ *est l'ensemble fini des états ;*
- $s_{init} \in \mathcal{S}$ *est l'état initial ;*
- $\mathcal{F} \subseteq \mathcal{S}$ *est l'ensemble (éventuellement vide) des états finaux ;*
- $\delta : \mathcal{S} \times (\mathbb{N} \cup \{\perp\}) \times \mathbb{N} \to \mathcal{S}$ *est la fonction de transition ;*
- $\lambda : \mathcal{S} \to \mathbb{N} \cup \{\perp\}$ *est la fonction de sortie.*

L'automate démarre dans l'état s_{init} en un sommet donné. Lorsque l'automate est sur le sommet u dans un état non final s, il calcule $p = \lambda(s)$. Si $p = \perp$, l'automate reste en u. Sinon, il quitte u par le port p. Lorsque l'automate arrive sur un sommet v par une arête e dans l'état s, il passe de l'état s à l'état $s' = \delta(s, \text{port}_v(e), \deg(v))$. Si l'automate ne s'est pas déplacé à l'étape précédente, le nouvel état s' à l'étape courante est calculé par la formule $s' = \delta(s, \perp, \deg(v))$.

L'automate de Moore est assez facile à manipuler et sera beaucoup utilisé dans la suite. Malheureusement, le nombre d'états de cet automate est toujours au moins égal au degré maximal du graphe à explorer. En effet, le port utilisé pour quitter un sommet ne dépend que de l'état. Dans une étoile de degré d, l'automate doit pouvoir sortir par chacune des d arêtes pour réussir l'exploration et doit donc avoir au moins d états différents.

Cette contrainte a pour conséquence que la capacité mémoire nécessaire à une tâche n'est pas toujours bien capturée par ce type d'automate. Dans les arbres par exemple, il existe un algorithme d'exploration très simple permettant d'effectuer un parcours en profondeur d'abord (DFS) perpétuel : si l'entité mobile arrive en un sommet u par le port i, alors elle quitte u par le port $i + 1$ (modulo $\deg(u)$). On aimerait pouvoir dire que l'entité mobile n'a pas besoin de mémoire. On utilise alors un automate de Mealy fini.

Définition 8 (Automate de Mealy)
Un automate de Mealy fini *est un quadruplet* $(\mathcal{S}, s_{init}, \mathcal{F}, \delta)$.
- $\mathcal{S}$ *est l'ensemble fini des états ;*
- $s_{init} \in \mathcal{S}$ *est l'état initial ;*
- $\mathcal{F} \subseteq \mathcal{S}$ *est l'ensemble (éventuellement vide) des états finaux ;*
- $\delta : \mathcal{S} \times (\mathbb{N} \cup \{\perp\}) \times \mathbb{N} \to \mathcal{S} \times (\mathbb{N} \cup \{\perp\})$ *est la fonction de transition/sortie.*

Dans la suite, les termes "fonction de transition" et "fonction de sortie" désigneront tous deux δ.

L'automate démarre dans l'état s_{init} sur un sommet donné. Supposons que l'automate est sur le sommet u dans l'état non final s. Si l'automate s'est déplacé à l'étape précédente, soit i le port par lequel il est arrivé. Sinon, on pose $i = \perp$. L'automate calcule $(j, s') = \delta(s, i, \deg(u))$ et passe dans l'état s'. Si $j = \perp$, il reste en u, sinon il quitte u par le port j.

Avec cette définition, il existe donc un automate de Mealy sans mémoire, c'est-à-dire avec un seul état, capable d'explorer (sans s'arrêter) tous les arbres.

Un automate $\mathcal{A}'$ simule un automate $\mathcal{A}$ dans un graphe G si $\mathcal{A}'$ effectue les mêmes traversées d'arêtes que $\mathcal{A}$ dans G, les deux automates partant du même sommet.

Proposition 1.1 *Pour tout automate de Moore à K états, il existe un automate de Mealy à $K+1$ états qui le simule.*

Preuve. Soit $\mathcal{A} = (\mathcal{S}, s_{init}, \mathcal{F}, \delta, \lambda)$ un automate de Moore fini. On suppose que l'état initial s_{init} n'est pas un état final. On pose $\mathcal{S}' = \mathcal{S} \cup \{s'_{init}\}$ où $s'_{init} \notin \mathcal{S}$ est un nouvel état. Soit $\mathcal{A}'$ l'automate de Mealy $(\mathcal{S}', s'_{init}, \mathcal{F}, \delta')$, avec δ' défini de la manière suivante :

$$\begin{cases} \delta'(s'_{init}, \perp, d) = (s_{init}, \lambda(s_{init})) & \forall\, d \in \mathbb{N} \\ \delta'(s, i, d) = (\delta(s, i, d), \lambda(\delta(s, i, d))) & \forall\, (s, i, d) \in \mathcal{S} \times (\mathbb{N} \cup \{\perp\}) \times \mathbb{N} \end{cases}$$

Le nouvel état s'_{init} sert à simuler correctement la première étape en faisant partir le nouvel automate $\mathcal{A}'$ dans le bon état par la bonne arête. Ensuite, la simulation est très simple. $\mathcal{A}'$ calcule par une seule fonction δ' ce que $\mathcal{A}$ calcule avec deux fonctions (δ puis λ). En conséquence, l'automate de Mealy $\mathcal{A}'$ simule exactement le comportement de l'automate de Moore $\mathcal{A}$. □

La simulation inverse est également possible mais l'augmentation du nombre d'états est plus significative.

Proposition 1.2 *Pour tout automate de Mealy à K états, il existe un automate de Moore à $(d+1)K+1$ états qui le simule dans tous les graphes de degré maximum au plus d, et avec exactement un pas de retard.*

Preuve. Fixons $d > 0$. Soit $\mathcal{A} = (\mathcal{S}, s_{init}, \mathcal{F}, \delta)$ un automate de Mealy fini. On pose $\mathcal{S}' = \{s'_{init}\} \cup (\mathcal{S} \times (\mathbb{Z}_d \cup \{\perp\}))$ et $\mathcal{F}' = \mathcal{F} \times (\mathbb{Z}_d \cup \{\perp\})$. Soit $\mathcal{A}'$ l'automate de Moore $(\mathcal{S}', s'_{init}, \mathcal{F}', \delta', \lambda')$. δ' est défini de la manière suivante :

$$\begin{cases} \delta'(s'_{init}, \perp, d) = \delta(s_{init}, \perp, d) & \forall\, d \in \mathbb{N} \\ \delta'((s, j), i, d) = \delta(s, i, d) & \forall\, ((s, j), i, d) \in (\mathcal{S}' \setminus \{s'_{init}\}) \times (\mathbb{N} \cup \{\perp\}) \times \mathbb{N} \end{cases}$$

et λ' comme suit :

$$\begin{cases} \lambda'(s'_{init}) = \perp \\ \lambda'((s, j)) = j & \forall\, (s, j) \in \mathcal{S}' \setminus \{s'_{init}\} \end{cases}$$

Le nouvel état s'_{init} sert à simuler correctement la première étape, mais prend un pas de retard. En effet, l'automate de Mealy commence par exécuter sa fonction de transition/sortie δ et peut donc effectuer sa première traversée d'arête dans un état différent de l'état initial. L'automate de Moore $\mathcal{A}'$ débute en restant immobile et, à l'étape suivante, change d'état et se déplace comme le fait l'automate $\mathcal{A}$ lors de la première étape. Ensuite, $\mathcal{A}'$ mémorise dans son état le port de sortie que sa fonction λ' doit donner pour correspondre au mouvement de $\mathcal{A}$. En conséquence, l'automate de Moore $\mathcal{A}'$ simule exactement le comportement de l'automate de Mealy $\mathcal{A}$, avec exactement une étape de retard. □

Nous avons introduit dans les deux définitions précédentes (Mealy et Moore) la possibilité pour les automates de s'arrêter en entrant dans un état final. Nous utiliserons cette propriété pour l'exploration avec arrêt (cf. Définition 20 pour plus de détails) qui nécessite que le ou les automates s'arrêtent après que tous les sommets soient explorés.

En fait, un automate tel que défini pour l'instant ne peut pas résoudre l'exploration avec arrêt dans les cycles. En effet dans les graphes anonymes, les sommets de même degré ne peuvent être distingués par l'automate. En conséquence, l'automate, s'il veut s'arrêter, doit le faire après un nombre arbitraire mais fixé c de traversées d'arêtes. L'automate échoue donc dans tout cycle d'au moins $c + 1$ sommets.

La façon la plus courante d'augmenter légèrement la puissance d'une entité mobile pour rendre possible l'exploration avec arrêt est de doter l'entité mobile de cailloux.

Définition 9 (Automate avec cailloux)
Un caillou *est un marqueur qui peut être déposé sur un sommet, détecté par l'automate, et éventuellement repris. Les cailloux sont indistinguables. Un* automate avec cailloux *est un automate fini de Moore ou de Mealy qui dispose d'un nombre fini de cailloux.*

L'automate débute l'exploration en possession de tous les cailloux. La fonction de transition est modifiée pour prendre en compte en entrée la présence ou non d'un caillou sur le sommet courant. La fonction de sortie est modifiée pour spécifier une action concernant les cailloux. Cette action consiste, au choix, "à ramasser le caillou", "à déposer un caillou", "à ne pas toucher aux cailloux". On veillera à ce que l'automate n'essaie pas de déposer un caillou si un autre caillou est déjà présent[1] ou si sa réserve est épuisée. De la même façon, ramasser un caillou est impossible si aucun caillou n'est présent sur le sommet.

Nous avons introduit dans les définitions de l'automate de Moore et de l'automate de Mealy la possibilité de rester immobile (grâce au symbole $\perp$). Cette possibilité est bien sûr inutile dans le cas d'un automate seul mais s'avère utile lorsque plusieurs automates sont présents simultanément dans le graphe. Nous définissons maintenant plusieurs modèles d'équipes d'automates, par ordre de puissance croissante (un type d'équipe peut en simuler un autre). Dans tous ces modèles, les automates partent du même sommet au même instant dans leur état initial respectif.

Définition 10 (Automates non coopératifs)
Une équipe finie d'automates finis non coopératifs *est un ensemble fini d'automates finis inconscients de l'existence des autres automates.*

Définition 11 (Automates localement coopératifs)
Une équipe finie d'automates finis localement coopératifs *est un ensemble fini d'automates finis qui communiquent lorsqu'ils se rencontrent en un sommet.*

Plus précisément, les différents automates de l'équipe ont chacun un identifiant unique et chaque automate connaît le nombre de ses coéquipiers et leur identifiant. La fonction de transition d'un automate $\mathcal{A}$ de l'équipe est modifiée pour prendre en compte la présence ou non des autres automates sur le même sommet. Pour tout automate $\mathcal{A}'$ présent sur le même sommet que $\mathcal{A}$, la fonction de transition de $\mathcal{A}$ dépend également de l'état de $\mathcal{A}'$ ainsi que du port d'origine de $\mathcal{A}'$ ($\perp$ si $\mathcal{A}'$ ne s'est pas déplacé à l'étape précédente).

[1] Cette contrainte est parfois omise dans certains papiers.

Les différents modèles définis jusqu'ici peuvent être classés par ordre croissant de puissance : automate fini, automate fini avec q cailloux, équipe finie de $q+1$ automates finis localement coopératifs. En effet chaque modèle peut être simulé par le précédent. Nous définissons maintenant deux modèles encore plus puissants.

Définition 12 (Automates globalement coopératifs)
Une équipe finie d'automates finis globalement coopératifs *est un ensemble fini d'automates finis qui communiquent en permanence (même s'ils ne sont pas situés sur le même sommet).*

Plus précisément, les différents automates de l'équipe ont chacun un identifiant unique et chaque automate connaît le nombre de ses coéquipiers et leur identifiant. La fonction de transition d'un automate $\mathcal{A}$ de l'équipe est modifiée pour prendre en compte la présence ou non des autres automates sur le même sommet. L'automate sait également toutes les rencontres qui ont lieu sur d'autres sommets entre les autres automates. Enfin, pour tout automate $\mathcal{A}$, la fonction de transition de $\mathcal{A}$ dépend également de l'état de tout autre automate $\mathcal{A}'$ ainsi que du port d'origine de $\mathcal{A}'$ ($\perp$ si $\mathcal{A}'$ ne s'est pas déplacé à l'étape précédente).

Dans ce modèle, tous les automates savent tout sur tous les autres automates. Ce modèle est donc équivalent au modèle suivant, dans lequel le contrôle est centralisé.

Définition 13 (Automate multi-têtes)
Un automate fini multi-têtes *est un automate fini contrôlant plusieurs têtes se déplaçant sur le graphe. L'automate (i.e., la fonction de transition) connaît, à une étape donnée, toutes les rencontres que font les têtes entre elles.*

Rappelons qu'une des principales motivations de ces modèles pour l'exploration de graphes est la résolution du problème USTCON en espace logarithmique. Ces modèles sont en fait simulés par une machine de Turing qui stocke l'état du ou des automates (en espace constant) ainsi que la position du ou des automates/têtes ($\log n$ bits par position donc $O(\log n)$ en tout). Puisque les positions des automates ne sont en fait que des pointeurs, il est possible de copier la valeur d'un pointeur dans un autre. Cela revient à téléporter un automate à la position d'un autre automate, d'où la définition suivante :

Définition 14 (JAG)
Un JAG *(Jumping Automaton for Graphs) est un automate fini multi-têtes possédant la capacité supplémentaire que toute tête peut se téléporter sur la position d'une autre tête.*

Ce modèle a été introduit par Cook et Rackoff [CR80]. Dans le même papier, les auteurs ont montré qu'aucun JAG n'est capable d'explorer tous les graphes. Il semble donc impossible de trouver une machine finie raisonnable capable d'explorer tous les graphes. Ainsi, pour résoudre USTCON en espace logarithmique grâce à l'exploration, il faut considérer des machines ayant une mémoire non constante. La première catégorie de "machines" (UTS et UXS) est un peu spéciale car la séquence d'instruction ne dépend pas du graphe mais est déterminée à l'avance.

Définition 15 (UTS)
Soit n et $d < n$ deux entiers strictement positifs tels que dn est pair. Une UTS *(universal traversal sequence) pour l'ensemble $\mathcal{G}_{n,d}$ des graphes d-réguliers à n sommets est une suite finie d'entiers $p_1, \ldots, p_k$ de $\mathbb{Z}_d^k$ qui explore tous les graphes de $\mathcal{G}_{n,d}$ au sens suivant. Une UTS est exécutée par une entité mobile dans un graphe de $\mathcal{G}_{n,d}$: à l'étape i, l'entité mobile quitte le sommet courant par le port p_i, $i = 1, \ldots, k$*

De manière similaire, nous définissons l'UXS, introduite par Koucký [Kou02].

Définition 16 (UXS)
Soit n et $d < n$ deux entiers strictement positifs tels que dn est pair. Une UXS *(universal exploration sequence) pour l'ensemble $\mathcal{G}_{n,d}$ des graphes d-réguliers à n sommets est une suite finie d'entiers $p_1, \ldots, p_k$ de $\mathbb{Z}_d^k$ qui explore tous les graphes de $\mathcal{G}_{n,d}$ au sens suivant. Une UXS est exécutée par une entité mobile dans un graphe de $\mathcal{G}_{n,d}$: à l'étape 1, l'entité mobile quitte le sommet courant par le port p_1. Considérons une étape $i = 2, \ldots, k$ et soit q_{i-1} le port par lequel l'entité mobile est entré sur le sommet courant. Alors l'entité mobile quitte ce sommet par le port $q_{i-1} + p_i$ (calculé dans $\mathbb{Z}_d$, donc modulo d).*

Intuitivement, l'UTS est une suite de directions absolues et l'UXS une suite de directions relatives. Deux types d'UTS et d'UXS sont recherchés : des suites courtes et des suites constructibles en espace logarithmique. Ces dernières sont bien sûr utiles dans l'objectif de prouver l'appartenance de USTCON à la classe **L**. Reingold [Rei05] a d'ailleurs prouvé **SL** = **L** en prouvant l'existence d'UXS constructibles en espace logarithmique.

Il est possible de définir un automate ayant une mémoire non bornée en prenant un ensemble d'états infinis (ou un nombre d'états dépendant de la taille du graphe). Cependant, ce modèle n'est pas complètement satisfaisant, en particulier vis-à-vis de la motivation de trouver un algorithme résolvant USTCON en espace logarithmique. En effet, il est très facile de trouver un automate ayant un nombre polynomial d'états, c'est-à-dire ayant une mémoire logarithmique, capable d'explorer tous les graphes de degré maximum d d'une taille donnée n. Prenons une UTS $p_1, \ldots, p_{n^6}$ de longueur polynomiale n^6 pour l'ensemble $\mathcal{G}_{n,d}$ des graphes d-réguliers à n sommets (son existence est prouvée, entre autres, par [AKL+79]). Considérons l'automate de Moore suivant $\mathcal{A} = (\{1, \ldots, n^6 + 1\}, 1, \{n^6 + 1\}, \delta, \lambda)$, avec λ et δ définis comme suit :

$$\begin{cases} \lambda(s) = p_s & \forall\, s \in \{1, \ldots, n^6\} \\ \delta(s, i, d) = s + 1 & \forall\, (s, i) \in \{1, \ldots, n^6\} \times \mathbb{Z}_d \end{cases}$$

Cet automate se contente d'exécuter l'UTS puis de s'arrêter. Il possède bien un nombre polynomial d'états et donc une mémoire logarithmique. Cependant si l'UTS choisie n'est pas constructible en espace logarithmique, cet automate ne peut pas être simulé par une machine de Turing en espace logarithmique. Cela provient du fait que la fonction λ est très difficile à calculer. Nous définissons donc le modèle suivant :

Définition 17 (Robot)
On appelle robot *une machine de Turing mobile.*

Le fonctionnement d'un robot est identique à celui d'un automate sauf que la fonction de transition et la fonction de sortie sont calculables par une machine de Turing. En pratique, tout algorithme d'exploration raisonnable entre dans le cadre de ce modèle.

Remarque. Tous les modèles définis ci-dessus concernent uniquement les graphes anonymes non orientés mais peuvent s'appliquer facilement aux graphes orientés ou non anonymes moyennant de légères modifications. Dans le cas des graphes orientés, la fonction de transition des automates n'a pas accès au numéro de port d'entrée (car il n'est pas défini). Elle ne dépend pas non plus du degré entrant mais uniquement du degré sortant du sommet courant. Dans le cas des graphes colorés ou étiquetés, la fonction de transition des automates dépend également de la couleur ou de l'étiquette du sommet courant.

1.2.4 Les différents types d'exploration

Nous définissons ici les différents types d'exploration. Nous commençons par la définition la moins contrainte de l'exploration.

Définition 18 (Exploration perpétuelle)
Chaque sommet du graphe doit être visité au moins une fois par au moins une entité mobile. Les entités mobiles n'ont pas à s'arrêter. Le temps d'exploration d'une exploration perpétuelle est le nombre d'étapes entre le début de l'exploration et le premier instant où tous les sommets ont été visités.

Quitte à ce que les entités mobiles parcourent le graphe indéfiniment, il peut être intéressant que chaque sommet du graphe soit visité périodiquement.

Définition 19 (Exploration périodique)
Chaque sommet du graphe doit être visité infiniment souvent. Le critère pertinent ici est la longueur de la période entre deux visites successives d'un même sommet.

Nous définissons également trois types d'exploration pour lesquels les entités mobiles doivent s'arrêter après avoir accompli la tâche désirée. Le temps d'exécution considéré dans ces cas est le nombre d'étapes entre le démarrage et le premier instant où toutes les entités mobiles sont dans un état final.

Définition 20 (Exploration avec arrêt)
Chaque sommet du graphe doit être visité au moins une fois par au moins une entité mobile. Les entités mobiles doivent ensuite s'arrêter (pas nécessairement immédiatement après que le dernier sommet inconnu ait été visité).

Définition 21 (Exploration avec retour)
L'exploration avec retour est une exploration avec arrêt avec une contrainte supplémentaire : les entités mobiles doivent s'arrêter à leur position de départ.

Définition 22 (Cartographie)
Les entités mobiles doivent s'arrêter et fournir en sortie une carte du graphe, c'est-à-dire une copie isomorphe du graphe, munie de tous les numéros de port. Le ou les sommets courants au moment de l'arrêt doivent également être indiqués sur la carte.

Pour un problème donné et une classe donnée de graphes, la ou les entités mobiles doivent satisfaire les contraintes du problème pour tous les graphes de la classe, quels que soient le sommet de départ et l'orientation locale du graphe. Ceci conduit à la définition suivante :

Définition 23 (Piège)
On appelle piège *pour un ensemble d'entités mobiles une paire* (G, u)*, constituée d'un graphe* G *munie d'une orientation locale et d'un sommet* u *du graphe, telle qu'il existe un sommet qu'aucune entité mobile n'explore lorsque celles-ci démarrent dans* G *sur le sommet* u*.*

1.3 Etat de l'art

Dans cette dernière section du chapitre 1, nous présentons un état de l'art du domaine. Nous commençons par les résultats de possibilité et surtout d'impossiblité de l'exploration par automates finis. Au vu de ces résultats, nous passons en revue les différentes études de la mémoire nécessaire et suffisante à l'exploration. Ces études concernent principalement les UTS et les UXS. Nous décrivons ensuite les travaux précédents concernant l'exploration des graphes étiquetés puis anonymes. Enfin, nous présentons quelques résultats sur l'exploration de graphes tolérante aux fautes.

1.3.1 Capacité d'exploration des automates finis

Cas des labyrinthes.

L'exploration a d'abord été étudiée dans le cadre naturel des labyrinthes, au sens du célèbre labyrinthe de Dédale décrit dans la mythologie grecque. Le problème de l'exploration des labyrinthes par automates finis a été introduit en 1951 par Shannon [Sha51]. Vingt ans plus tard, Döpp [Döp71a, Döp71b] définit formellement la notion de labyrinthe. Un labyrinthe est une grille infinie à deux dimensions de laquelle on retire un ensemble L non nécessairement fini de sommets. Le labyrinthe est dit fini si L est fini, co-fini si L couvre l'ensemble de la grille sauf un nombre fini de sommets, et infini sinon. Le but pour l'automate est de s'éloigner arbitrairement loin de son point de départ dans les cas fini et infini, ou d'explorer tous les sommets dans le cas co-fini (la définition des labyrinthes impose certaines contraintes de connexité pour que ces problèmes aient un sens).

Döpp soupçonne dans ces mêmes papiers qu'il n'existe pas d'automate fini capable de sortir de tout labyrinthe fini. Cette conjecture est prouvée par Budach [Bud78] qui utilise une très longue preuve d'une centaine de pages (l'auteur prouve aussi le résultat

pour les labyrinthes co-finis). Des preuves un peu plus simples ont ensuite été publiées en russe [Kil91, KPU85] mais je n'ai pas pu consulter ces publications. Finalement, Kilibarda [Kil93] donne une preuve plus courte de ce résultat en 1993. Müller [Mül79] renforce le résultat en prouvant que pour tout automate fini, il existe un labyrinthe fini avec seulement trois composantes connexes retirées de la grille dont l'automate n'arrive pas à sortir. Blum et Kozen [BK78] prouvent que deux automates finis localement coopératifs peuvent explorer les labyrinthes finis et co-finis. Ils montrent le même résultat avec un automate fini muni de deux cailloux. Hoffmann [Hof81] clôt le problème en démontrant qu'un seul caillou est insuffisant. Enfin, pour toute équipe finie d'automates finis non coopératifs, il existe un piège sous la forme d'un labyrinthe fini (cf [ABR79]).

Concernant les labyrinthes infinis, un premier résultat de Blum et Sakoda [BS77] montre que sept cailloux sont suffisants. Ce résultat est amélioré à cinq cailloux par Szepietowski [Sze82]. Ce dernier résultat est prouvé optimal par [Hof85]. Finalement, Kilibarda [Kil93] exhibe d'autres caractérisations en variant le nombre d'automates et de cailloux. Enfin, on pourra se reporter à [BH67] et [DM02] pour plus d'informations sur le comportement des automates finis dans les grilles infinies (ces papiers traitent de la reconnaissance de motifs).

Cas des graphes arbitraires.

En 1967, Rabin [Rab67] a proposé la conjecture suivante lors d'un séminaire à Berkeley : aucun automate fini équipé d'un nombre fini de cailloux ne peut explorer tous les graphes. Les premiers travaux pour prouver cette conjecture se concentrent sur l'automate fini sans caillou. En 1971, Müller [Mül71] donne quelques arguments formels en ce sens. En 1977, Coy [Coy77] fournit une preuve plus complète mais dont certaines parties restent floues. La première preuve formelle est généralement attribuée à Budach [Bud78] qui montre qu'aucun automate fini ne peut explorer tous les labyrinthes co-finis et donc tous les graphes. Blum et Kozen [BK78] ont donné une preuve plus simple s'appliquant aux graphes et ont ensuite démontré que deux automates, puis trois automates localement coopératifs, ne pouvaient pas explorer tous les graphes. Le résultat pour quatre automates a été démontré l'année suivante par Kozen [Koz79].

La conjecture de Rabin a finalement été démontrée en 1980 par Rollik [Rol80]. En fait, il démontre un résultat plus fort, à savoir qu'aucune équipe finie d'automates finis localement coopératifs ne peut explorer tous les graphes, même en se restreignant aux graphes planaires de degré maximal 3. En 1987, Hemmerling [Hem87] démontre que pour tout $k \geq 2$, une équipe de k automates globalement coopératifs est strictement plus puissante qu'une équipe de k automates localement coopératifs. Malheureusement, les automates finis multi-têtes ne sont pas assez puissants pour l'exploration puisque même un JAG ne peut explorer tous les graphes [CR80]. Ce résultat met un terme à l'espoir qu'il existe une machine finie raisonnable capable d'explorer tous les graphes.

Un tour d'horizon des résultats sur l'exploration de graphes et de labyrinthes peut être trouvé dans [Hem89] et [Del97].

Ajout de tableaux blancs.

Pour aider l'automate fini à effectuer la difficile tâche de l'exploration, on peut doter chaque sommet d'un tableau blanc. Le tableau blanc est une zone de mémoire sur laquelle l'automate peut lire, écrire et effacer des messages. Ces tableaux blancs ont généralement une taille de $O(\log n)$ bits, où n est la taille du graphe. Cette zone de mémoire additionnelle aide considérablement l'exploration. En effet, il existe un automate fini explorant tous les graphes orientés anonymes munis de tableaux blancs, cf. [AG94]. L'automate réalise l'exploration avec arrêt en $O(mn + n^2 \log n)$ étapes, où m est le nombre d'arcs. Les auteurs prouvent également une borne inférieure de $\Omega(mn)$ étapes. En 2004, Bourdonov [Bou04b] améliore la borne supérieure en proposant un algorithme s'exécutant en $O(mn + n^2 \log\log n)$ étapes. L'auteur propose même dans [Bou04a] un algorithme en $O(mn+n^2 \log^* n)$ étapes, mais il ne s'applique que pour un sous-ensemble des orientations locales possibles.

L'algorithme consistant à quitter un sommet u par le port p modulo $d^+(u)$ lors de la p-ième visite, explore bien entièrement le graphe. Cependant Afek et Gafni [AG94] prouvent l'existence d'une famille de graphes orientés pour laquelle l'exploration prend un temps exponentiel. Par contre, dans les graphes eulériens, cet algorithme traverse perpétuellement un cycle eulérien, après une période de stabilisation [BEGT02]. Les auteurs prouvent en fait que, en utilisant les tableaux blancs, l'exploration de graphes orientés eulériens est plus facile que l'exploration de graphes orientés arbitraires. En effet, il existe un automate fini explorant les graphes orientés eulériens avec arrêt en au plus $3m$ étapes. Ce résultat s'étend aux graphes anonymes non orientés. En effet, on peut voir un graphe non orienté à m arêtes comme un graphe orienté symétrique à $2m$ arcs. Cela donne une borne supérieure de $6m$ étapes. En fait, un simple DFS permet d'effectuer l'exploration avec arrêt par automate fini en $2m$ étapes (les tableaux blancs permettent d'indiquer si le sommet a déjà été visité et, si c'est le cas, l'arête qui mène à la racine de l'arbre DFS).

Notons pour finir que les tableaux blancs peuvent aussi servir à la communication entre agents. Dans [FGKP04], les auteurs s'intéressent à l'exploration collaborative des arbres et étudient l'impact sur les performances de l'absence de tableaux blancs. Dans [DFNS05], les tableaux blancs servent à élire un leader parmi les agents. Celui-ci coordonne ensuite l'exploration et la cartographie du graphe.

1.3.2 UTS et UXS

On note $U(d, n)$, resp. $X(d, n)$, la longueur minimale d'une UTS, resp. UXS, pour les graphes d-réguliers à n sommets.

Bornes non constructives.

La plupart des bornes supérieures sur $U(d, n)$ proviennent de preuves purement existentielles d'UTS. Généralement, ces preuves sont basées sur des arguments probabilistes

à partir de l'étude des marches aléatoires. La première borne supérieure polynomiale apparaît dans [AKL+79]. Les auteurs prouvent qu'une marche aléatoire de longueur $2dn^2$ explore tous les sommets d'un graphe d-régulier à n sommets avec une probabilité au moins $1/2$. Si la longueur est augmentée à $\Theta(d^2n^3 \log n)$, alors on obtient le résultat avec forte probabilité. En fait, la probabilité est suffisamment grande pour pouvoir en déduire l'existence d'une UTS de longueur $\Theta(d^2n^3 \log n)$. Cette borne est améliorée d'un facteur d par [KLNS89] par une étude des marches aléatoires dédiée aux graphes réguliers. Dans le cas des graphes denses, le même type d'étude [CRR+97] donne une borne supérieure $O(n^3 \log n)$. Pour le cas particulier des cliques, Bar-Noy et al. [BNBK+89] avaient prouvé auparavant $O(n^3 \log^2 n)$, amélioré ensuite en $O(n^3 \log n)$ par Alon, Azar et Ravid [AAR90].

A ce jour, aucune borne inférieure ne coïncide avec ces bornes supérieures. Bar-Noy et al. [BNBK+89] ont montré une borne inférieure générale de $\Omega(n \log n)$ sur $U(d, n)$. Les auteurs ont fourni une meilleure borne, $\Omega(n \log^2 n / \log \log n)$ pour les cliques. Cette borne est améliorée dans [AAR90] à $\Omega(n^2)$. Borodin, Ruzzo et Tompa [BRT92] prouvent $U(d, n) = \Omega(d^2n^2 + dn^2 \log(n/d))$ pour $3 \leq d \leq n/3 - 2$. Ce résultat est amélioré par Tompa et Coppersmith [Tom92] qui prouvent $U(d, n) = \Omega(d^{0,71}n^{2,29})$ à partir d'un résultat pour les cycles $U(2, n) = \Omega(n^{1,29})$. Une nouvelle amélioration de Buss et Tompa [BT95] donne $U(2, n) = \Omega(n^{1,43})$ et $U(d, n) = \Omega(d^{0,57}n^{2,43})$.

Bornes supérieures constructives.

Construire explicitement des UTS est une tâche très difficile, même pour les cas a priori simples comme les cycles. Bridgland [Bri87], et indépendamment Bar-Noy et al. [BNBK+89] sont les premiers à concevoir des UTS. Celles-ci sont de longueur $n^{O(\log n)}$, ne sont valables que pour les cycles et peuvent être construites en espace logarithmique par rapport à leur longueur. Le même résultat est obtenu pour les cliques par Karloff, Paturi et Simon [KPS88]. Istrail [Ist88] est le premier à contruire une UTS pour les cycles de taille polynomiale et en utilisant uniquement un espace logarithmique. Koucký [Kou03a] obtient plus tard une borne $O(n^{4,03})$ qui améliore la borne $O(n^{4,76})$ d'Istrail. Hoory et Wigderson [HW93] obtiennent également des UTS de taille polynomiale constructibles en espace logarithmique. Cependant celles-ci se limitent au cas des expanders de degré constant munis d'orientations locales ayant certaines propriétés spécifiques.

Peu de bornes constructives existent pour le cas général. La plupart proviennent du développement des générateurs de bits pseudo-aléatoires (cf. [Sak96]). Le premier résultat [BNS92] donne des UTS de longueur $2^{2^{O(\sqrt{\log n})}}$. Ce résultat est drastiquement amélioré par Nisan [Nis92] qui construit des UTS de longueur $n^{O(\log n)}$. Ce résultat est ensuite approfondi dans [INW94]. Finalement, Reingold [Rei05] construit en espace logarithmique des UTS pour tous les graphes munis d'orientations locales ayant certaines propriétés de symétrie. L'existence d'UTS constructibles en espace logarithmique pour les graphes arbitraires reste un problème ouvert.

La construction d'UXS semble beaucoup plus facile. En effet, dans son papier introduisant les UXS, Koucký [Kou02] exhibe des UXS extrêmement simples pour les cycles, les cliques et les arbres. Il décrit aussi une famille d'UXS constructibles en espace logarith-

mique pour les expanders et plus généralement pour les graphes de diamètre au plus logarithmique. Dans un autre papier [Kou03b], Koucký prouve que les résultats valables pour les UTS le sont souvent aussi pour les UXS. Notamment, il montre $X(d,n) \leq U(3,dn)$. Finalement, Reingold [Rei05] décrit comment obtenir une UXS constructible en espace logarithmique.

Résolution compacte de USTCON.

La quête d'UXS et d'UTS constructibles en espace logarithmique avait pour but de prouver USTCON $\in \mathbf{L}$. Comme nous l'avons déjà dit, ce résultat a finalement été prouvé par Reingold [Rei05] en 2005. Précédant celui-ci, de nombreux papiers ont essayé de cerner plus précisément la classe **SL**. Savitch [Sav70] a montré dès 1970 que $\mathbf{SL} \subseteq \mathbf{NL} \subseteq \mathbf{L}^2$, où $\mathbf{L}^\alpha$ désigne l'ensemble des langages reconnaissables par une machine de Turing déterministe utilisant $O(\log^\alpha n)$ bits de mémoire pour les entrées de taille n. Le résultat d'Aleliunas et al. [AKL+79] sur les marches aléatoires prouve l'existence d'un algorithme probabiliste de type Monte-Carlo pour USTCON. En terme de classes de complexité, ce résultat peut se formuler comme suit : $\mathbf{SL} \subseteq \mathbf{RL}$. On a en fait $\mathbf{SL} \subseteq (\mathbf{RL} \cup \mathbf{co\text{-}RL}) = \mathbf{ZPL}$. En effet, un algorithme de type Las Vegas est décrit dans [BCD+89a, BCD+89b]. Ce résultat peut aussi se retrouver indirectement à partir de $\mathbf{SL} = \mathbf{co\text{-}SL}$ démontré dans [NTS95].

La première grande amélioration en terme d'espace nécessaire pour USTCON apparaît dans [NSW92] où les auteurs prouvent $\mathbf{SL} \subseteq \mathbf{L}^{3/2}$. Ce résultat est généralisé par Saks et Zhou [SZ99]. S'inspirant de ces deux papiers, Armoni, Ta-Shma, Wigderson et Zhou [ATSWZ00] montrent un résultat plus puissant : $\mathbf{SL} \subseteq \mathbf{L}^{4/3}$. Finalement, indépendamment de Reingold, Trifonov [Tri05] prouve que USTCON peut être résolu de manière déterministe en utilisant $O(\log n \log\log n)$ bits d'espace mémoire. Pour compléter ce panorama, citons deux papiers, [BBR+99] et [CCvM06], présentant des compromis temps-espace pour USTCON (entre autres).

1.3.3 Exploration de graphes étiquetés

Cartographie des graphes orientés.

La *déficience* d'un graphe orienté est le nombre minimal d'arcs à rajouter pour rendre le graphe eulérien. Cette notion a été introduite en 1988 par Kutten [Kut88]. Les graphes orientés eulériens sont assez faciles à explorer rapidement. En fait, plusieurs travaux [AH00, DP99, FT05, Kwe97] ont montré que la difficulté à explorer un graphe orienté croît avec la déficience du graphe. Considérons le problème de la cartographie de graphes orientés étiquetés par un robot. La mesure d'efficacité d'un algorithme est le rapport entre le temps d'exécution de l'algorithme sur un graphe G et le temps de cartographie optimal en connaissant le graphe, maximisé pour tous les graphes orientés et tous les sommets de départ. Le temps de cartographie optimal en connaissant le graphe correspond au problème polynomial du "facteur chinois" (Chinese Postman Problem), qui consiste à trouver le chemin orienté de taille minimale traversant toutes les arêtes (cf. [EJ73]).

Deng et Papadimitriou [DP99] ont obtenu de nombreux résultats sur ce problème. Pour les graphes de déficience 0, c'est-à-dire les graphes eulériens, le rapport compétitif optimal est 2. Pour les graphes de déficience 1, le rapport compétitif optimal est compris entre 3 et 4. Pour les graphes de déficience inconnue et arbitraire d, les auteurs ont montré une borne inférieure de d et une borne supérieure de $d^{O(d)}$. Cette borne supérieure est améliorée à $d^{O(\log d)}$ par Albers et Henzinger [AH00]. Dans ce même papier, les auteurs complètent l'étude des algorithmes simples (glouton, DFS, BFS) menée par Kwek [Kwe97] sur ce problème. Finalement, en 2005, Fleischer and Trippen [FT05] montrent que le rapport compétitif optimal est au plus polynomial en montrant une borne supérieure de $O(d^8)$ pour ce rapport.

Exploration avec contraintes.

L'exploration de graphes a aussi été étudiée en rajoutant des contraintes visant à la rendre plus réaliste. Le robot est souvent autonome en énergie et a donc une quantité d'énergie limitée. Betke, Rivest et Singh [BRS95] ont introduit la contrainte suivante : le robot a un réservoir d'essence limité et doit retourner périodiquement faire le plein au sommet de départ. Plus précisément, étant donné un graphe étiqueté non orienté G et un sommet u de ce graphe, le robot doit cartographier G sans jamais effectuer plus de $B = (2 + \alpha)r$ traversées d'arêtes entre deux visites du sommet de départ u, où r désigne l'excentricité de u dans G, et où α est une constante strictement positive. Le robot ne connaît au départ que l'excentricité r et la limite B. Dans ce même papier, les auteurs décrivent un algorithme optimal pour les grilles avec obstacles rectangulaires, s'exécutant en $\Theta(m)$ étapes (m est le nombre d'arêtes du graphe). Awerbuch et les mêmes auteurs [ABRS99] ont présenté un algorithme valable pour les graphes arbitraires mais utilisant $O(m+n^{1+o(1)})$ traversées d'arêtes dans le pire des cas. Ce résultat a ensuite été amélioré par Awerbuch et Kobourov [AK98] pour obtenir $O(m + n \log^2 n)$ traversées d'arêtes, toujours dans les graphes arbitraires. Finalement, Duncan, Kobourov et Kumar [DKK01] ont clos le problème en prouvant un algorithme optimal utilisant $\Theta(m)$ traversées d'arêtes.

1.3.4 Exploration de graphes anonymes

Graphes non orientés.

Nous considérons maintenant la cartographie des graphes anonymes munis d'une orientation locale relative par un robot à mémoire non bornée et muni de cailloux indistinguables (au moins un). Une orientation locale relative est une orientation locale pour laquelle le robot ne perçoit pas le numéro de port en lui-même mais n'a conscience que de l'ordre cyclique des arêtes incidentes à un sommet. Ce problème peut être résolu en temps $\Theta(mn)$ dans les graphes de n sommets et m arêtes [DJMW91]. Une accélération est possible en utilisant plusieurs cailloux mais l'algorithme proposé reste en $O(mn)$. Ce problème est relié à celui de la *vérification* de cartes dans lequel le robot doit vérifier que la carte qui lui est fournie est bien la carte du graphe [DMM01, DJMW97]. Deng et Mirzaian [DM96] définissent le rapport compétitif d'un algorithme de cartographie

comme le rapport entre le temps d'exécution de l'algorithme et le temps d'exécution du meilleur algorithme de vérification de carte. Dans le cas d'un unique caillou, les auteurs prouvent que l'algorithme présenté dans [DJMW91] est d'une certaine manière optimal. Plus précisément, cet algorithme a un rapport compétitif de $O(n)$ et ce rapport est optimal pour une certaine classe d'algorithmes à laquelle il appartient ("relaxed depth-one strategies"). Les auteurs conjecturent que cette borne est également optimale pour les algorithmes sans restriction. La seule borne inférieure connue sur le rapport compétitif reste cependant $\Omega(\log n)$.

Graphes orientés.

Les graphes orientés anonymes ont peu été étudiés. Un premier papier de Bender et Slonim [BS94] décrit un algorithme probabiliste d'exploration avec arrêt par deux robots globalement coopératifs. L'espérance du temps d'exécution est $O(d^2n^5)$ étapes pour les graphes à n sommets de degré extérieur maximal d. Les auteurs prouvent qu'un seul robot est insuffisant puisqu'il n'existe pas d'algorithme d'exploration perpétuelle, même probabiliste, s'exécutant en temps polynomial dans tous les graphes orientés. En fait, $\Omega(\log\log n)$ cailloux sont nécessaires pour cette tâche [BFR$^+$02]. Les auteurs de ce dernier papier montre que cette borne est optimale en proposant un algorithme déterministe de cartographie permettant à un robot muni de $O(\log\log n)$ cailloux de cartographier le graphe en temps polynomial.

Arbres.

Diks, Fraigniaud, Kranakis et Pelc [DFKP04] ont étudié la quantité de mémoire nécessaire à l'exploration d'arbres anonymes. Alors que l'exploration perpétuelle par un automate de Mealy ne nécessite aucune mémoire, les auteurs montrent que $\Omega(\log\log\log n)$ bits de mémoire sont nécessaires pour effectuer l'exploration avec arrêt. L'exploration avec retour nécessite $\Omega(\log n)$ bits de mémoire et peut être réalisé par un robot ayant $O(\log^2 n)$ bits de mémoire.

1.3.5 Exploration de graphes avec fautes

Arbres avec fautes.

Markou et Pelc [MP06] ont considéré le problème de l'exploration des arbres dont certaines arêtes sont fautives. Un robot, connaissant l'arbre mais ne connaissant pas l'emplacement des fautes, doit explorer la composante connexe sans faute dans laquelle il se trouve. Le coût de l'exploration est le nombre de traversées d'arêtes. Pour un arbre donné et un sommet de départ donné, le surcoût d'un algorithme d'exploration est le maximum (sur toutes les configurations de fautes) du rapport entre son coût et le coût d'un algorithme optimal qui connaît l'emplacement des fautes. Les auteurs décrivent un algorithme pour les lignes ayant le meilleur surcoût possible. Pour les arbres, ils proposent un algorithme ayant un surcoût au plus 9/8 fois plus grand que le meilleur surcoût possible.

Recherche de trou noir.

Un trou noir est un nœud *hostile* qui détruit tous les agents mobiles qui le visitent, et qui ne laisse pas de trace observable de la destruction (cf. [CS02]). La recherche de trou noir est la tâche consistant à localiser les trous noirs d'un réseau par l'exploration de celui-ci par un ensemble d'agents mobiles. Un agent mobile au moins doit survivre, et tous les survivants doivent connaître l'emplacement du trou noir. Les travaux sur ce sujet cherchent à minimiser deux paramètres : d'abord le nombre d'agents, et ensuite le temps d'exécution. Le problème est étudié en environnements synchrone ou asynchrone.

En environnement (partiellement) synchrone, on suppose qu'il existe une borne supérieure sur le temps de traversée d'une arête. Formellement, une traversée d'arête par un agent mobile (ainsi que les calculs associés) est supposée prendre une unité de temps. Le but est de trouver le trou noir (ou de prouver qu'il n'y en a pas) en un temps minimal. Grâce à la propriété de synchronisme, deux agents coopérant localement suffisent à accomplir cette tâche. Etant donné un graphe, un sommet de départ et un ensemble S de sommets sûrs (n'hébergeant pas le trou noir), le problème de trouver la stratégie la plus rapide est **NP**-difficile [CKMP06]. Les auteurs décrivent un algorithme d'approximation de rapport 9,3. Ce rapport est amélioré à 6 par Klasing, Markou, Radzik et Sarracco [KMRS05a]. Ces derniers prouvent aussi que le problème n'est pas approximable avec un rapport inférieur à $\frac{389}{388}$ à moins que **P**=**NP**. Dans le cas où l'ensemble S de sommets a priori sûrs est réduit au sommet de départ, le problème reste **NP**-difficile [KMRS05b], et **APX**-difficile [KMRS05a]. Dans le premier des deux papiers, les auteurs décrivent un algorithme d'approximation de rapport 7/2 et prouvent que tout algorithme basé sur un arbre couvrant a un rapport d'approximation d'au moins 3/2. Enfin, le problème est étudié dans les arbres [CKMP05]. Pour certaines classes d'arbres, il existe un algorithme exact en temps linéaire. Pour les arbres arbitraires, il existe un algoritme d'approximation de rapport 5/3.

En environnement asynchrone, le temps de traversée d'une arête est fini mais non borné. Il n'est donc pas envisageable d'attendre le retour d'un agent se sacrifiant pour tester une arête. Le problème est alors plus difficile. Les agents doivent savoir s'il existe ou non un trou noir, que le trou noir est unique s'il en existe, et le graphe doit être 2-sommet-connexe. Les sommets du graphe sont équipés de tableaux blancs. Le nombre de sommets du graphe doit être connu par les agents. Généralement, ceux-ci ont la connaissance de toute la carte du graphe. Dans ce cas, deux agents sont nécessaires et suffisants. Pour résoudre le problème de la recherche du trou noir dans les graphes arbitraires, $\Theta(n \log n)$ traversées d'arêtes sont nécessaires et suffisantes [DFPS02]. Ce résultat reste vrai pour la classe des anneaux [DFPS01]. Il est cependant possible de résoudre le problème plus rapidement dans certaines classes de graphes. En particulier, dans les hypercubes, les graphes papillon, les tores et les grilles (sous certaines conditions, cf. [DFK$^+$06b]), le problème peut être résolu en $\Theta(n)$ traversées d'arêtes. La même performance est obtenue dans les graphes de petit diamètre puisque $O(n + D \log D)$ traversées d'arêtes sont suffisantes dans les graphes arbitraires de diamètre D [DFS04]. Enfin, si les agents n'ont pas de carte du graphe mais ne connaissent que sa taille n et son degré maximal d, alors $d+1$ agents sont nécessaires et suffisants et le coût en traversées d'arêtes est alors $\Theta(n^2)$ [DFPS02].

Première partie

Exploration sans assistance

Contenu de la partie

Cette partie porte sur l'étude de la quantité de mémoire nécessaire et suffisante que doit posséder un automate pour réaliser l'exploration d'une famille donnée de graphes. Nous supposons ici que l'automate ne possède aucune information a priori sur le graphe à explorer. De plus, un graphe est dit exploré par l'automate si et seulement si il visite tous les sommets *quelle que soit* l'orientation locale du graphe et *quelle que soit* le sommet de départ.

Le chapitre 2 s'intéresse à la quantité de mémoire que doit posséder un automate pour réaliser l'exploration (perpétuelle ou avec arrêt) de deux familles : la famille des graphes dont le nombre de sommets est borné par un certain entier n, et la famille des graphes de diamètre au plus D et de degré maximum au plus d, où D et d sont deux entiers donnés.

Le chapitre 3 étudie l'espace mémoire nécessaire à l'exploration des graphes orientés. Une borne inférieure sur cette quantité de mémoire est prouvée au début du chapitre. Ensuite, deux algorithmes d'exploration avec arrêt sont proposés. Le premier utilise un seul caillou mais s'exécute en temps exponentiel dans le pire des cas. Le second utilise $O(\log \log n)$ cailloux mais s'exécute en temps polynomial. L'espace utilisé par ces deux algorithmes est analysé et comparé à la borne inférieure.

Le chapitre 4 considère le nombre d'états dont a besoin un automate pour explorer un graphe régulier donné. Il est ainsi possible de définir la classe $\mathcal{G}_K$ des graphes réguliers explorables par un automate à K états. Clairement $\mathcal{G}_K \subseteq \mathcal{G}_{K+1}$. Nous conjecturons que la suite $\mathcal{G}_K$ est strictement croissante, c'est-à-dire $\mathcal{G}_K \neq \mathcal{G}_{K+1}$ pour tout $K > 0$. Nous donnons différents arguments en faveur de cette conjecture. En particulier, une conséquence de Rollik [Rol80] et de notre amélioration du chapitre 2 est qu'un saut exponentiel de K à $K^{O(K)}$ permet d'obtenir une inclusion stricte. Nous réduisons ce saut à une quantité polynomiale.

Chapitre 2

Exploration des graphes non orientés

Dans ce chapitre, nous étudions la complexité en espace de l'exploration perpétuelle et de l'exploration avec arrêt des graphes non orientés anonymes. Dans la sous-section 1.3.1, nous avons vu qu'aucun JAG ne pouvait explorer tous les graphes. Puisqu'aucune machine finie raisonnable ne semble capable d'explorer tous les graphes, il est naturel de se demander quelle quantité de mémoire un robot doit posséder pour accomplir l'exploration dans une famille donnée de graphes. Nous prouvons ici diverses bornes inférieures et supérieures sur la quantité de mémoire nécessaire et suffisante pour l'exploration de graphes par différents types de machines.

2.1 Introduction

La méthode la plus classique pour obtenir une borne inférieure sur la mémoire consiste à construire un piège le plus petit possible pour un robot de taille donnée. Le premier piège pour un automate fini est celui de Budach [Bud78]. Malheureusement celui-ci est de trop grande taille pour en déduire une borne inférieure intéressante. Rollik [Rol80] construit un piège beaucoup plus petit. Pour un automate fini à K états, il construit un piège planaire de degré maximum 3 ayant au plus $2K$ sommets. Par conséquent, un automate explorant tous les graphes d'au plus n sommets doit posséder au moins $n/2$ états et donc avoir au moins $\Omega(\log n)$ bits de mémoire. Rollik construit également un piège pour une équipe de q automates non coopératifs ayant au plus K états chacun. Le piège est de taille $K^{O(q)}$. A partir de ce piège, l'auteur construit un piège pour une équipe de q automates localement coopératifs ayant au plus K états chacun. Ce piège a un nombre de sommets en $\tilde{O}(K^{K^{\cdot^{\cdot^{K}}}})$, avec $2q+1$ niveaux d'exponentielle, où la notation $\tilde{O}$ cache des facteurs logarithmiques. A partir de la construction de Rollik, il est possible d'obtenir un piège de taille $K^{O(K)}$ pour un automate muni d'un caillou. Ce piège donne une borne $\Omega(\log\log n)$ pour l'exploration avec arrêt par un automate muni d'un caillou.

Les bornes supérieures sur la mémoire proviennent de l'étude des algorithmes pour USTCON. La meilleure borne supérieure, en fonction du nombre de sommets du graphe, est celle donnée par Reingold [Rei05]. Il prouve en effet l'existence d'un robot ayant

$O(\log n)$ bits de mémoire capable d'explorer les graphes d'au plus n sommets. Le fait que USTCON soit complet dans la classe **L** ne permet pas d'exhiber de bornes inférieures sur la mémoire d'un robot réalisant l'exploration. En effet, la simulation du déplacement d'un robot dans un graphe par une machine de Turing utilise déjà $\log n$ bits de mémoire pour stocker la position courante du robot, même si celui-ci a une mémoire constante.

Dans ce chapitre, nous présentons une nouvelle méthode de construction de pièges pour un ou plusieurs automates. Cette méthode utilise la notion d'automate réduit qui nous permet de nous concentrer sur la structure intrinsèque d'un automate et son comportement dans les graphes d-homogènes (Section 2.2). Grâce à l'étude d'un automate réduit, nous construisons l'ébauche d'un piège pour l'automate correspondant (Section 2.3). Ensuite, nous complétons cette ébauche pour construire des pièges pour cet automate. En particulier, étant donné un automate à K états et pour tout $d \geq 3$, nous construisons un piège de degré maximum exactement d et ayant au plus $K + d + 2$ sommets (Sous-section 2.4.1). Nous retrouvons ainsi la borne inférieure de $\Omega(\log n)$ bits obtenue par Rollik. Nous construisons également un piège ayant un faible diamètre. Ce piège nous permet d'obtenir une borne inférieure de $\Omega(D \log d)$ bits de mémoire pour l'exploration des graphes de diamètre au plus D et de degré maximum au plus d (Sous-section 2.4.2). Nous prouvons ensuite que cette borne est optimale en notant qu'un algorithme de parcours en profondeur d'abord, de profondeur croissante, utilise $O(D \log d)$ bits de mémoire (Sous-section 2.4.3).

Nous construisons ensuite, pour toute équipe de q automates non coopératifs à K états, un piège de taille $O(qK)$ (Section 2.5). En utilisant ce piège de taille réduite dans la construction de Rollik [Rol80], nous obtenons un piège pour une équipe de q automates localement coopératifs à K états ayant au plus $\tilde{O}(K^{K^{\cdot^{\cdot^{K}}}})$ sommets, cette fois-ci avec "seulement" $q+1$ niveaux d'exponentielle. A partir du piège pour une équipe d'automates non coopératifs, nous construisons un piège pour un automate à K états muni d'un caillou. Ce piège a au plus $O(K^3)$ sommets, ce qui permet d'obtenir une borne inférieure de $\Omega(\log n)$ bits pour l'exploration avec arrêt (Sous-section 2.6.1). Enfin, nous décrivons un algorithme d'exploration permettant à un robot muni d'un caillou d'explorer tous les graphes avec arrêt. Dans les graphes de diamètre D et de degré maximum d, ce robot utilise au plus $O(D \log d)$ bits de mémoire (Sous-section 2.6.2).

Notons que toutes nos bornes inférieures ne concernent pas seulement les robots mais aussi les automates, c'est-à-dire des machines n'ayant pas forcément une fonction de transition "raisonnable" (cf. la discussion autour de la définition 17). Notons également que la plupart des pièges présentés dans ce chapitre sont planaires et de degré maximum constant, ce qui renforce nos bornes inférieures.

Les résultats de ce chapitre sont issus de deux collaborations. La première, avec Pierre Fraigniaud, Guy Peer, Andrzej Pelc et David Peleg, a donné lieu à deux publications, l'une en conférence [FIP+04], l'autre dans un journal [FIP+05]. La seconde collaboration, avec Pierre Fraigniaud, Sergio Rajsbaum et Sébastien Tixeuil, a également donné lieu à deux publications : [FIRT05] puis [FIRT06].

2.2 Automates réduits

Toutes nos bornes inférieures s'appliquent aux automates de Moore. Dans cette section, nous définissons la notion d'automate *réduit*, qui correspond grossièrement à la forme épurée d'un automate de Moore se déplaçant dans des graphes homogènes.

2.2.1 Remarques préliminaires

Nous considérons ici le comportement d'un automate fini $\mathcal{A} = (\mathcal{S}, s_{init}, \mathcal{F}, \delta, \lambda)$ dans un graphe d-homogène. Supposons que l'automate est dans un état s sur un sommet v. L'automate quitte le sommet par le port $\lambda(s)$. Comme le graphe est d-homogène, le numéro de port d'entrée est aussi $\lambda(s)$. De plus, le degré du sommet atteint est nécessairement d. L'automate transite donc dans l'état $\delta(s, \lambda(s), d)$. Nous définissons la fonction de transition simplifiée f_d de la façon suivante :

$$\forall s \in \mathcal{S} \quad f_d(s) = \delta(s, \lambda(s), d)$$

Comme le comportement de l'automate ne dépend pas du graphe d-homogène sur lequel il se déplace, un automate ayant des états finaux, s'il s'arrête, le fait après un nombre i toujours identique d'étapes. Tout graphe de $i+1$ sommets, pour tout sommet de départ, est donc un piège pour l'automate. Nous supposons par conséquent que l'automate n'atteint pas ces états finaux dans les graphes d-homogènes. En résumé, nous pouvons décrire l'automate sous une forme plus simple, appelée d-simplifiée : l'automate $(\mathcal{S}, s_{init}, \mathcal{F}, \delta, \lambda)$ devient l'automate $(\mathcal{S}, s_{init}, f_d, \lambda)$.

Les automates n'ont aucun moyen de s'orienter dans les graphes homogènes. Nous allons nous en servir pour construire des pièges. Ceux-ci seront construits à partir de sous-graphes que les automates ne pourront pas quitter. Ces sous-graphes paraîtront homogènes aux automates. Ensuite, ces sous-graphes seront complétés, pas nécessairement en un graphe homogène.

Plus précisément, le sous-graphe est un graphe contenant des demi-arêtes. Une arête peut en effet être considérée comme deux demi-arêtes incidentes. Si une demi-arête n'est pas incidente à une autre demi-arête, on dit que c'est une demi-arête pendante.

Définition 24 (Graphe partiel homogène)
Un graphe partiel d-homogène *est un graphe possédant éventuellement des demi-arêtes pendantes et vérifiant les propriétés suivantes :*

- *Tous les sommets ont le même degré d, où le degré d'un sommet est ici le nombre de demi-arêtes, pendantes ou non, incidentes au sommet.*
- *Toutes les demi-arêtes incidentes à un sommet quelconque du graphe possèdent chacune un numéro de port. Ces numéros de port sont deux à deux distincts et pris dans l'ensemble $\mathbb{Z}_d$.*
- *Si deux demi-arêtes forment une arête, alors elles ont le même numéro de port.*

Les sous-graphes de piège que nous considérons, appelés squelettes de piège, sont en fait des graphes partiels homogènes vérifiant les propriétés suivantes :

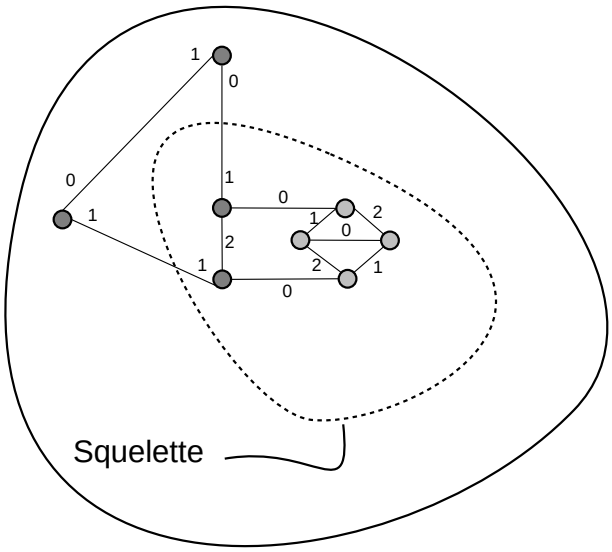

FIG. 2.1 – Un piège et son squelette

Définition 25 (Squelette de piège)
Un squelette de piège *pour une équipe d'automates est une paire* (G, u)*, où* G *est un graphe partiel homogène contenant au moins une demi-arête pendante et* u *un sommet de* G*, telle que, si tous les automates débutent en* u*, chacun dans leur état initial, alors aucun automate ne cherche à prendre une demi-arête pendante.*

Les squelettes de piège sont ensuite complétés en des graphes pour former des pièges. Un exemple est donné en figure 2.1. Les squelettes de piège sont construits à partir des graphes de transition des automates. La suite est consacrée à leur étude.

Considérons un automate fini dans sa forme d-simplifiée $\mathcal{A} = (\mathcal{S}, s_{init}, f_d, \lambda)$. Les fonctions de transition f_d et de sortie λ définissent un graphe orienté $G(\mathcal{A}) = (\mathcal{S}, F)$ de la façon suivante. L'ensemble des sommets est l'ensemble des états de l'automate. Pour chaque sommet s de $G(\mathcal{A})$ il existe exactement un arc sortant de s. Son extrémité débouche sur le sommet $f_d(s)$. Cet arc est étiqueté $\lambda(s)$. Notons que la donnée du graphe G et du sommet initial s_{init} détermine complètement le comportement de l'automate $\mathcal{A}$ dans les graphes d-homogènes. Nous supposons dans le reste du chapitre que tout état $s \in \mathcal{S}$ de $\mathcal{A}$ est accessible depuis s_{init}. En effet, les états non accessibles n'affectent pas le comportement de l'automate dans les graphes d-homogènes et peuvent donc être ignorés.

Comme chaque sommet de $G(\mathcal{A})$ a un degré sortant exactement égal à 1, $G(\mathcal{A})$ consiste en un chemin orienté élémentaire, éventuellement de longueur nulle, partant de s_{init} et se terminant en un sommet s_1, suivi d'un cycle orienté élémentaire partant et se terminant en s_1. Cela provient aussi du fait que l'on a supposé l'automate fini et ne possédant pas d'états non accessibles dans les graphes d-homogènes. Les étiquettes des arcs du chemin définissent un *mot de chemin* W_0 d'alphabet $\mathbb{Z}_d$. Les étiquettes des arcs du cycle définissent un *mot de cycle* W, également d'alphabet $\mathbb{Z}_d$. On a $|W_0| \geq 0$, $|W| \geq 1$ et clairement $|W_0 W| = |\mathcal{S}|$. L'*empreinte* de $\mathcal{A}$ est $emp(\mathcal{A}) = W_0 W^*$. Quand $\mathcal{A}$ est placé sur

un sommet d'un graphe d-homogène G dans son état initial s_{init}, $emp(\mathcal{A})$ est la suite des étiquettes des arêtes traversées par $\mathcal{A}$.

Un automate $\mathcal{A}$ est dit *irréductible* si $G(\mathcal{A})$ vérifie deux propriétés : (i) pour toute paire d'arcs consécutifs (distincts) $((s, s_1), (s_1, s_2))$, on a $\lambda(s) \neq \lambda(s_1)$, c'est-à-dire les étiquettes des deux arcs sont différentes, et (ii) pour toute paire d'arcs ayant la même destination, les deux arcs n'ont pas la même étiquette. Nous montrons ici comment obtenir un automate irréductible $\mathcal{A}'$ à partir d'un automate $\mathcal{A}$. Les comportements de $\mathcal{A}$ et de $\mathcal{A}'$ dans le graphe ne seront pas exactement identiques mais seront liés dans le sens où la région du graphe traversée par $\mathcal{A}$ ne peut pas être beaucoup plus grande que celle traversée par $\mathcal{A}'$.

Soit $\bar{G}(\mathcal{A})$ le graphe non orienté correspondant à $G(\mathcal{A})$. Grossièrement, nous voulons que l'automate $\mathcal{A}$ soit irréductible pour que $\bar{G}(\mathcal{A})$ puisse être transformé en un graphe partiel homogène qui soit un squelette de piège pour $\mathcal{A}$. Pour ce faire, $\bar{G}(\mathcal{A})$ doit être à arêtes colorées. Pour obtenir un automate irréductible $\mathcal{A}'$ à partir de $\mathcal{A}$, nous réalisons une série d'étapes de réduction qui modifient la fonction de transition ainsi que les états accessibles. Quand $\mathcal{A}$ et $\mathcal{A}'$ sont placés sur le même sommet du graphe, le chemin parcouru par $\mathcal{A}'$ est contenu dans le chemin parcouru par $\mathcal{A}$. En fait, seuls quelques chemins fermés sont parcourus par $\mathcal{A}$ et pas par $\mathcal{A}'$. Nous définissons maintenant formellement les réductions.

Une *étape de réduction* est l'opération consistant à transformer un automate $\mathcal{A} = (\mathcal{S}, s_{init}, f, \lambda)$ en un autre automate $\mathcal{A}' = (\mathcal{S}', s'_{init}, f', \lambda')$, $\mathcal{S}' \subseteq \mathcal{S}$, où l'une des propriétés énoncées plus haut, (i) ou (ii), est assurée pour deux arcs. Nous définissons deux types de réductions, correspondant aux deux propriétés. L'idée est de répéter des réductions de type i jusqu'à ce que ce ne soit plus possible, et alors l'automate vérifie la propriété (i). Ensuite, si la propriété (ii) n'est pas vérifiée, nous effectuons une seule réduction de type ii pour assurer la propriété (ii). Seules les réductions de type i modifient le chemin parcouru par l'automate.

2.2.2 Réduction de type i

Une étape de réduction est *applicable* si $G(\mathcal{A})$ a deux arcs consécutifs distincts (s, s_1) et (s_1, s_2) avec $\lambda(s) = \lambda(s_1)$. L'idée de la réduction est illustrée par la figure 2.2. Dans (a), il y a un segment de $G(\mathcal{A})$ avec deux arcs consécutifs étiquetés 1, et dans (b) il y a le segment correspondant de $G(\mathcal{A}')$ après la réduction. Dans cet exemple, s a seulement un voisin entrant, t, et donc s devient inaccessible. Ceci est l'idée de base de la réduction de type i, mais dans la définition formelle plus bas, nous considérerons plusieurs cas, dépendant du degré entrant de s et de l'emplacement de l'état initial.

Les propriétés de la réduction de type i dont nous avons besoin sont illustrées par la figure 2.2(c) et (d), où le chemin parcouru par $\mathcal{A}$ et $\mathcal{A}'$ respectivement est indiqué par des flèches en pointillé. Si $\mathcal{A}$ est sur le sommet w de G dans l'état t, il se déplace sur le sommet v dans l'état s, puis va en v', transite dans l'état s_1, et revient en v dans l'état s_2 (puisque $\lambda(s) = \lambda(s_1) = i$, où i est l'étiquette de $\{v, v'\}$; dans la figure, $i = 1$). Il est facile de vérifier qu'une réduction de type i élimine l'aller-retour v, v', v du chemin parcouru par le robot dans le graphe, et ne modifie pas autrement le chemin. Autrement dit, si le

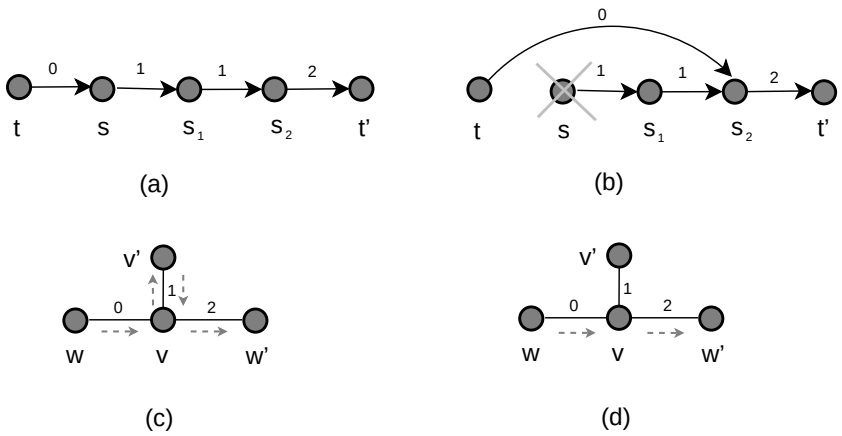

FIG. 2.2 – Une réduction de type i

chemin arrive en v depuis w et ensuite continue en w' après avoir fait l'aller-retour v, v', v, alors, après la réduction de type i, l'automate va de w en v et ensuite directement en w'. En conséquence, avant l'étape de réduction, l'automate explore un sommet à distance au plus 1 des sommets explorés par l'automate après la réduction.

Formellement, une *réduction de type i* transforme $\mathcal{A}$ en $\mathcal{A}'$ par les modifications suivantes de la fonction de transition f et de l'état initial s_{init} ($f' = f$ et $s'_{init} = s_{init}$ à moins qu'un changement soit spécifié). Nous considérons quatre cas :

Cas $s = s_2$: Dans ce cas, le cycle est de longueur 2 avec les mêmes étiquettes sur les deux arcs. Supposons, sans perte de généralité, que s n'a pas de voisins entrants autre que s_1 (il est impossible qu'à la fois s et s_1 aient deux voisins entrants). Soit $f'(s_1) = s_1$. Si $s = s_{init}$, alors $s'_{init} = s_1$.

Autrement, si $s \neq s_2$, il est possible que s ait 0, 1 ou 2 voisins entrants.

Cas $s \neq s_2$, s a 0 voisin entrant : Dans ce cas, $s = s_{init}$. Soit $s'_{init} = s_2$.

Cas $s \neq s_2$, s a 1 voisin entrant : Soit t le voisin entrant de s ($t \neq s_1$). On pose $f'(t) = s_2$. Si $s = s_{init}$, alors $s'_{init} = s_2$.

Cas $s \neq s_2$, s a 2 voisins entrants : Supposons que les deux voisins entrants de s sont t_1 et t_2. On a $s \neq s_{init}$. Alors $f'(t_1) = s_2$ et $f'(t_2) = s_2$.

Une fois ces modifications effectuées, $\mathcal{A}'$ est obtenu par suppression de tout état non accessible. Notons que pour chacun des quatre cas précédents, au moins un état inaccessible est retiré, à savoir s. Donc au plus $K - 1$ réductions de type i sont possibles en partant d'un automate à K états.

Lemme 2.1 *Soit $\mathcal{A}'$ le robot obtenu à partir de $\mathcal{A} = (\mathcal{S}, s_{init}, f, \lambda)$ par application d'une réduction de type i sur les arcs (s, s_1) et (s_1, s_2) avec $\lambda(s) = \lambda(s_1)$. Alors*

1. *Le sommet s et l'arc (s, s_1) n'apparaissent plus dans $\mathcal{A}'$.*
2. *Si $\mathcal{A}$ et $\mathcal{A}'$ débutent sur le même sommet d'un graphe dans le même état t_0 appartenant à leur cycle, alors la prochaine fois que $\mathcal{A}$ et $\mathcal{A}'$ retourneront dans l'état t_0,*

ils seront situés sur le même sommet v et $\mathcal{A}$ aura traversé au plus une arête de plus que $\mathcal{A}'$.

Preuve. La première partie du lemme est vraie car l'état s devient inaccessible dans $\mathcal{A}'$. Nous prouvons maintenant la deuxième partie du lemme. Nous considérons donc que $\mathcal{A}$ et $\mathcal{A}'$ sont tous deux démarrés sur un sommet x_0 dans un état t_0 qui appartient à leur cycle.

Supposons qu'une réduction de type i est appliquée à $G(\mathcal{A})$ sur les arcs (s, s_1) et (s_1, s_2) avec $\lambda(s) = \lambda(s_1)$, pour obtenir $\mathcal{A}'$. Nous traitons le cas où s n'a qu'un voisin entrant t et $s \neq s_{init}$. Les autres cas se traitent de manière similaire. Dans le cas étudié, $\mathcal{A}'$ est égal à $\mathcal{A}$ excepté que $f'(t) = s_2$ (et donc s devient inaccessible).

Considérons la suite de configurations de l'automate $\mathcal{A}$ lorsqu'il est démarré sur le sommet x_0 dans l'état t_0,

$$(x_0, t_0) \to (x_1, t_1) \to \cdots$$

Alors la suite de configurations de $\mathcal{A}'$ est la même, excepté qu'à chaque fois que $\mathcal{A}$ entre dans l'état t, disons à la i-ème étape

$$\cdots \to (x_i, t_i) \to (x_{i+1}, t_{i+1}) \to (x_{i+2}, t_{i+2}) \to \cdots$$

où $t_i = t$, et donc $t_{i+3} = s_2$ avec $x_{i+1} = x_{i+3}$ (puisque $\lambda(t_{i+1}) = \lambda(t_{i+2})$), alors la suite de configurations de $\mathcal{A}'$ est

$$\cdots \to (x_i, t_i) \to (x_{i+3}, t_{i+3}) \to \cdots$$

En conséquence, le parcours d'origine dans le graphe

$$x_0, x_1, \ldots, x_i, x_{i+1}, x_{i+2}, x_{i+3}, \ldots$$

devient

$$x_0, x_1, \ldots, x_i, x_{i+3}, \ldots$$

et le cycle $x_{i+1}, x_{i+2}, x_{i+3}$ ($x_{i+1} = x_{i+3}$) parcourant l'arête $\{x_{i+1}, x_{i+2}\}$ dans les deux sens est éliminé du parcours. □

2.2.3 Réduction de type ii

Lorsque plus aucune réduction de type i n'est applicable dans $G(\mathcal{A})$, une réduction de type ii peut être utilisée. Une réduction de type ii est *applicable* si $G(\mathcal{A})$ a deux états s et s_1 ayant le même voisin sortant t et tels que $\lambda(s) = \lambda(s_1)$. Se référer à la figure 2.3 où $\lambda(s) = \lambda(s_1) = 1$; $G(\mathcal{A})$ est présenté dans la partie (a), et $G(\mathcal{A}')$, après la réduction donc, dans la partie (b). Une *réduction de type ii* transforme $\mathcal{A}$ en $\mathcal{A}'$ par les modifications suivantes de f et de l'état initial s_{init} ($f' = f$ et $s'_{init} = s_{init}$ à moins qu'un changement soit spécifié). Exactement un des deux états s et s_1 doit être dans le cycle de $G(\mathcal{A})$, par exemple s_1. Il y a donc un chemin orienté de t à s_1. Ce chemin est de longueur au moins 1,

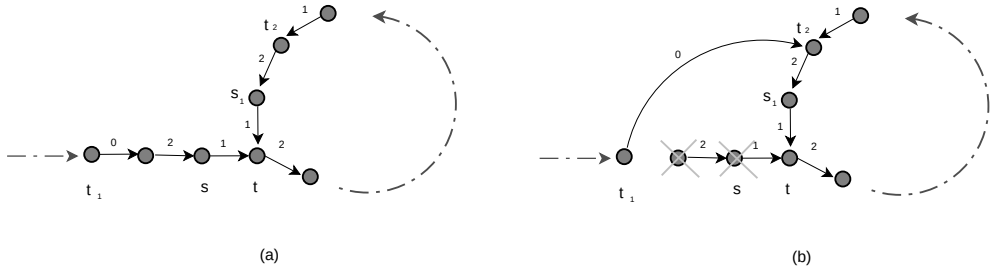

FIG. 2.3 – Une réduction de type ii

parce que sinon $t = s_1$ et il y aurait une boucle de t à lui-même étiqueté $\lambda(s)$. De ce fait, une réduction de type i serait applicable. Rappelons que $emp(A) = W_0W^*$. Soit W' le plus long préfixe commun de W_0 et W^*. $|W'| > 0$ par l'hypothèse d'applicabilité de la réduction de type ii. Soit t_2 le sommet où W' commence dans le cycle de $G(\mathcal{A})$. Dans la figure 2.3, on a $W' = 21$. Nous considérons deux cas. Dans les deux cas, $\mathcal{A}'$ est obtenu à partir de $\mathcal{A}$ par les modifications suivantes, et en supprimant les états inaccessibles :

Cas $|W_0| > |W'|$ **:** C'est-à-dire W' est un préfixe strict de W_0 ; soit t_1 le voisin entrant du sommet t'_1 où W' commence dans le chemin élémentaire de $G(\mathcal{A})$. Alors $\lambda(t'_1) = \lambda(t_2)$ est la première lettre de W'. On pose $f'(t_1) = t_2$.

Cas $|W_0| = |W'|$ **:** On change l'état initial $s'_{init} = t_2$.

Le lemme suivant est immédiat.

Lemme 2.2 *Soit $\mathcal{A}' = (\mathcal{S}', s'_{init}, f', \lambda')$ l'automate obtenu à partir de $\mathcal{A} = (\mathcal{S}, s_{init}, f, \lambda)$ par application d'une réduction de type ii sur les arcs (s,t) et (s_1,t), avec $\lambda(s) = \lambda(s_1)$, et s_1 dans le cycle de $G(A)$. Alors*

1. *Le sommet s et l'arc (s,t) n'apparaissent plus dans $\mathcal{A}'$. De plus, une réduction de type ii n'est pas applicable dans $G(\mathcal{A}')$.*
2. *Si $\mathcal{A}$ et $\mathcal{A}'$ débutent sur le même sommet d'un graphe, ils effectuent tous les deux le même parcours.*
3. *Si aucune réduction de type i n'est applicable sur $G(\mathcal{A})$, alors aucune n'est applicable sur $G(\mathcal{A}')$.*

2.2.4 Propriétés de la réduction

En utilisant les lemmes 2.1 et 2.2, il est facile de montrer le lemme suivant, résumant la procédure pour obtenir un automate irréductible.

Lemme 2.3 *Soit $\mathcal{A}' = (\mathcal{S}', s'_{init}, f', \lambda')$ l'automate obtenu à partir de $\mathcal{A} = (\mathcal{S}, s_{init}, f, \lambda)$ par application de la plus longue suite possible de réductions de type i, suivie par une (éventuelle) réduction de type ii. Soit k le nombre de réductions de cette suite.*

1. *$\mathcal{A}'$ est irreductible.*

2. $|\mathcal{S}'| + k \leq |\mathcal{S}|$.
3. *Si $\mathcal{A}$ et $\mathcal{A}'$ débutent sur le même sommet d'un graphe dans le même état s appartenant à leur cycle, alors la prochaine fois que $\mathcal{A}$ et $\mathcal{A}'$ retourneront dans l'état s, ils seront situés sur le même sommet v et $\mathcal{A}$ aura traversé au plus k arêtes de plus que $\mathcal{A}'$.*

Preuve. La première partie du lemme est une conséquence du lemme 2.1(1) et du lemme 2.2(1,3) : s'il existe deux arcs violant la propriété (i), alors une réduction de type i peut être effectuée, et au moins un état est retiré dans le processus. Par ailleurs, si tous les arcs vérifient la propriété (i) mais qu'il existe deux arcs violant la propriété (ii), alors une réduction de type ii éliminera ce problème, sans créer d'arcs violant la propriété (i).

La deuxième partie du lemme est vraie car chaque réduction de type i et de type ii élimine au moins un état, comme il a été observé dans le lemme 2.1(1) et le lemme 2.2(1).

La troisième partie du lemme vient du lemme 2.1(2) et du lemme 2.2(2), par récurrence sur k. □

2.3 Squelettes de piège

Dans cette section, nous considérons un automate et une version irréductible de celui-ci. Nous montrons d'abord comment construire un squelette de piège pour l'automate réduit, et ensuite comment l'étendre en squelette de piège pour l'automate originel.

2.3.1 Cas d'un automate irréductible

Soit $\widehat{\mathcal{A}} = (\mathcal{S}, s_{init}, f, \lambda)$ un automate irréductible d'empreinte $emp(\widehat{\mathcal{A}}) = W_0W^*$, avec $|W_0W| = K$. Rappelons que son graphe $G(\widehat{\mathcal{A}})$ de transitions d'états dans les graphes d-homogènes consiste en un chemin orienté partant de l'état initial s_{init}, suivi d'un cycle orienté. Donc, le graphe non orienté correspondant, $\bar{G}(\widehat{\mathcal{A}})$, consiste en un chemin P connecté à un cycle C. Si C est de longueur au moins 3, nous posons $\bar{G}_1(\widehat{\mathcal{A}}) = \bar{G}(\widehat{\mathcal{A}})$. Si C est de longueur plus petite que 3, nous modifions un peu $\bar{G}(\widehat{\mathcal{A}})$ pour obtenir $\bar{G}_1(\widehat{\mathcal{A}})$ de la façon suivante :

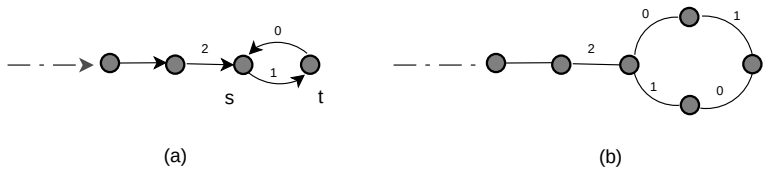

FIG. 2.4 – Elimination des arêtes multiples

– Supposons que le cycle orienté de $G(\widehat{\mathcal{A}})$ est de longueur 2, composé des états s et t (la figure 2.4(a) illustre ce cas avec $W = 10$). Alors le cycle non orienté de $\bar{G}_1(\widehat{\mathcal{A}})$

a 4 arêtes, étiquetées WW, en ajoutant deux nouveaux sommets, comme indiqué dans la figure 2.4(b). Le chemin reste P, comme dans $\bar{G}(\widehat{\mathcal{A}})$.

– Supposons que le cycle orienté de $G(\widehat{\mathcal{A}})$ est de longueur 1, composé de l'état s (la figure 2.4(a) illustre ce cas avec $W = 1$). Alors le cycle non orienté de $\bar{G}_1(\widehat{\mathcal{A}})$ aura 4 arêtes, étiquetées $abab$, où a est égal à l'unique lettre de W, et où b est différent de a et de la dernière lettre (si elle existe) de W_0, comme indiqué dans la figure 2.5(b), où $abab = 1010$. Le chemin reste P, comme dans $\bar{G}(\widehat{\mathcal{A}})$.

Notons que $\bar{G}_1(\widehat{\mathcal{A}})$ a au plus un sommet de degré 3, le sommet où le chemin et le cycle sont liés. Ce graphe n'est donc pas homogène. Nous complétons $\bar{G}_1(\widehat{\mathcal{A}})$ par des demi-arêtes pendantes pour que chaque sommet soit de degré exactement d. Soit $\mathrm{sp}_d(\widehat{\mathcal{A}})$ le graphe partiel d-homogène obtenu ("sp" pour "squelette de piège"). Soit x_0 le premier sommet de P. Clairement, avec x_0 comme point de départ, $\mathrm{sp}_d(\widehat{\mathcal{A}})$ est un squelette de piège pour $\widehat{\mathcal{A}}$. Ce graphe a au plus 3 sommets de plus que $\bar{G}(\widehat{\mathcal{A}})$. Nous avons le lemme suivant.

Lemme 2.4 *Pour tout automate $\widehat{\mathcal{A}} = (\mathcal{S}, s_{init}, f, \lambda)$, le graphe $\mathrm{sp}_d(\widehat{\mathcal{A}})$ est un graphe partiel d-homogène, avec au plus $|\mathcal{S}| + 3$ sommets, tel qu'il en existe une réprésentation plane dans laquelle toutes les demi-arêtes pendantes appartiennent à une même face. De plus $(\mathrm{sp}_d(\widehat{\mathcal{A}}), x_0)$ est un squelette de piège pour $\widehat{\mathcal{A}}$.*

FIG. 2.5 – Elimination d'une boucle

2.3.2 Cas d'un automate quelconque

Considérons un automate quelconque $\mathcal{A}$. Soit $\widehat{\mathcal{A}}$ un automate irréductible obtenu à partir de $\mathcal{A}$. Considérons son squelette de piège $\mathrm{sp}_d(\widehat{\mathcal{A}})$. D'après le lemme 2.4, le graphe $\mathrm{sp}_d(\widehat{\mathcal{A}})$ est un graphe partiel d-homogène et un squelette de piège pour $\widehat{\mathcal{A}}$. Ainsi, $\widehat{\mathcal{A}}$ peut être placé sur le premier sommet x_0 du chemin P de $\mathrm{sp}_d(\widehat{\mathcal{A}})$ dans son état initial, et il n'essaiera jamais de traverser une demi-arête pendante du graphe.

Maintenant, plaçons $\mathcal{A}$ en x_0 dans son état initial. Chaque fois que $\mathcal{A}$ veut prendre une demi-arête pendante, ayant un certain numéro de port i, nous complétons cette demi-arête en une arête complète, d'étiquette i, dont la nouvelle extrémité est un nouveau sommet v. Nous ajoutons $d - 1$ demi-arêtes pendantes à v pour qu'il ait le degré d. D'après le lemme 2.1, les chemins parcourus par $\mathcal{A}$, et pas par $\widehat{\mathcal{A}}$, sont des allers-retours le long d'arêtes où $\mathcal{A}$ est dans des états éliminés par les réductions de type i. Les arêtes ajoutées à $\mathrm{sp}_d(\widehat{\mathcal{A}})$ forment donc des arbres. Comme les sommets ajoutés correspondent aux états éliminés, nous obtenons un graphe ayant au plus $K + 3$ sommets, où K est le

nombre d'états de $\mathcal{A}$. Maintenant, $\mathcal{A}$ n'essaie jamais de prendre une demi-arête pendante du nouveau graphe, que nous appelons $\mathrm{sp}_d(\mathcal{A})$.

Lemme 2.5 *Pour tout automate $\mathcal{A} = (\mathcal{S}, s_{init}, f, \lambda)$, le graphe $\mathrm{sp}_d(\mathcal{A})$ est un graphe partiel d-homogène, avec au plus $|\mathcal{S}| + 3$ sommets, tel qu'il en existe une réprésentation plane dans laquelle toutes les demi-arêtes pendantes appartiennent à une même face. De plus $(\mathrm{sp}_d(\mathcal{A}), x_0)$ est un squelette de piège pour $\mathcal{A}$.*

2.4 Piège pour un automate fini

Dans cette section, nous utilisons les squelettes de piège construits précédemment pour obtenir deux pièges. Nous construisons ces deux pièges en veillant à minimiser le nombre de sommets pour le premier, et le diamètre pour le second.

Le premier piège donne une borne inférieure de $\Omega(\log n)$ bits pour l'exploration des graphes de taille au plus n. Cette borne est optimale car l'algorithme de Reingold [Rei05] pour USTCON a pour conséquence l'existence d'un robot de $O(\log n)$ bits capable d'explorer tous les graphes d'au plus n sommets.

Le second piège donne une borne inférieure de $\Omega(D \log d)$ bits, en fonction du degré d et du diamètre D. Nous montrons l'optimalité de cette borne en décrivant un algorithme d'exploration utilisant cette quantité de mémoire dans les graphes de diamètre D et de degré d.

2.4.1 Piège de petite taille

Une fois un squelette de piège obtenu pour un automate $\mathcal{A}$, il n'est pas difficile de construire un piège pour $\mathcal{A}$ en complétant les demi-arêtes pendantes. Notons que le graphe obtenu n'est pas nécessairement d-homogène. Cependant, l'automate ne visite que des sommets de degré d et ne traverse que des arêtes dont les deux numéros de port sont identiques.

Nous partons du graphe $\mathrm{sp}_d(\mathcal{A})$. Tant qu'il existe deux demi-arêtes pendantes incidentes à des sommets différents u et u' tels que l'arête $\{u, u'\}$ n'existe pas, nous relions ces deux demi-arêtes pour former une arête. Le graphe partiel obtenu possède au plus d sommets ayant encore des demi-arêtes pendantes. Chacun de ces sommets possède au plus $d - 1$ demi-arêtes pendantes. Nous ajoutons au plus $d - 1$ sommets pour compléter ces demi-arêtes en arêtes. Plus précisément, chaque sommet ajouté est relié par une arête de numéros de port appropriés à tous les sommets possédant encore une demi-arête pendante. Les sommets ajoutés ne sont donc pas nécessairement de degré d. Enfin, si aucun sommet n'a été ajouté au cours du processus, nous insérons un sommet, qui ne sera pas visité, au milieu d'une des arêtes formées au cours de l'opération (il en existe nécessairement une car il y a au moins une demi-arête pendante dans $\mathrm{sp}_d(\mathcal{A})$). Soit G le graphe obtenu.

Par construction, le graphe obtenu est un graphe simple. De plus, comme il a été construit à partir d'un squelette de piège $(\mathrm{sp}_d(\mathcal{A}), x_0)$, nous avons le théorème suivant :

Théorème 2.1 *Pour tout automate $\mathcal{A}$ de K états, et pour tout $d \geq 3$, il existe un piège d'au plus $K + d + 2$ sommets et de degré maximum exactement égal à d.*

Par contraposé de l'énoncé, nous obtenons le corollaire suivant, déjà montré par Rollik dans [Rol80] :

Corollaire 2.1 *Un automate explorant tous les graphes simples de taille au plus n a besoin d'au moins $\Omega(\log n)$ bits de mémoire.*

2.4.2 Piège planaire de petit diamètre

Le résultat suivant relie le nombre d'états d'un automate avec le diamètre des graphes qu'il peut explorer.

Théorème 2.2 *Pour tout automate à K états et pour tout $d \geq 3$, il existe un graphe simple planaire de degré maximum d, de diamètre au plus $4\lceil \log_{d-1}(K+3) \rceil + 2$ et ayant $O(K)$ sommets, que l'automate ne peut explorer à partir d'un de ses sommets.*

Preuve. Fixons $d \geq 3$ et $\mathcal{A}$ un automate. Soit K son nombre d'états. Nous partons du squelette de piège $sp_d(\mathcal{A})$ défini dans la sous-section 2.3.2. Le graphe partiel $sp_d(\mathcal{A})$ a au plus $K+3$ sommets et, pour chaque sommet, au plus $d-1$ demi-arêtes pendantes. Nous considérons l'arbre d-régulier T de hauteur $h = \lceil \log_{d-1}(K+3) \rceil$. L'arbre B a $d(d-1)^{h-1}$ feuilles, avec $K+3 \leq d(d-1)^{h-1} < d(K+3)$. Pour tout sommet de $sp_d(\mathcal{A})$ possédant des demi-arêtes pendantes, nous relions ces demi-arêtes à une même feuille de T. A chaque sommet de $sp_d(\mathcal{A})$ correspond une feuille différente. Pour éviter les arêtes multiples, nous insérons un sommet au milieu de chacune d'elles. Les numéros de port qui n'étaient pas encore définis le sont arbitrairement, puisque l'automate ne les verra jamais. D'après le lemme 2.5, il existe une représentation plane du graphe telle que toutes les demi-arêtes pendantes appartiennent à une même face. Le graphe T étant un arbre, il existe un couplage entre les sommets de $sp_d(\mathcal{A})$ et les feuilles de l'arbre tel que le graphe obtenu G soit planaire. Le nombre de sommets de G est au plus $O(dK)$. Cependant il suffit de réduire l'arbre pour qu'il ait $K+3$ feuilles et $O(K)$ sommets en tout pour que G ait lui aussi $O(K)$ sommets.

Nous calculons maintenant une borne supérieure sur le diamètre de G. Le graphe $sp_d(\mathcal{A})$ est construit à partir du graphe $\bar{G}_1(\widehat{\mathcal{A}})$. Rappelons que ce graphe consiste en un cycle de longueur au moins 3 connecté éventuellement à un chemin élémentaire. Le graphe $sp_d(\mathcal{A})$ est obtenu en ajoutant des arbres enracinés en des voisins des sommets de $\bar{G}_1(\widehat{\mathcal{A}})$. En résumé, $sp_d(\mathcal{A})$ est donc composé d'un cycle, auquel sont attachés des arbres, le tout complété par des demi-arêtes pendantes pour former un graphe partiel d-homogène. Dans $\bar{G}_1(\widehat{\mathcal{A}})$, tous les sommets sont de degré au plus 2 sauf éventuellement un, qui est de degré 3 (le sommet à la jonction du chemin et du cycle). Puisque $d \geq 3$, tout sommet u de $sp_d(\mathcal{A})$ est incident à une demi-arête pendante ou est à distance au plus 1 d'un sommet u' d'un des arbres. Supposons qu'il n'existe pas de demi-arête pendante à distance inférieure à h de u'. Alors à distance au plus h du sommet u', il y a au moins $(d-1)^h + 1$ sommets.

Comme $(d-1)^h \geq K+3$, il existe un sommet u'' à distance au plus $h-1$ de u' qui n'a pas d voisins et qui donc est incident à une demi-arête pendante. En conséquence, u'' est connecté à l'arbre T. Donc tout sommet de $sp_d(\mathcal{A})$ est à distance au plus $h+1$ d'un sommet de T. Le diamètre de T étant $2h$, on obtient que le diamètre de G est au plus $(h+1)+2h+(h+1) = 4h+2$, ce qui conclut la preuve. □

Remarque 2.1 *Nous avons utilisé l'arbre d-régulier T dans la preuve du théorème 2.2, d'abord pour sa simplicité, mais surtout pour obtenir un graphe planaire. Cependant, il est possible d'utiliser d'autres graphes de diamètre plus petit que T. Par exemple, pour certaines valeurs de d, il est possible d'utiliser le graphe $(d-1)$-régulier non orienté de de Bruijn. Dans ce cas cependant, le piège obtenu ne serait plus planaire.*

Comme conséquence directe du théorème 2.2, nous avons :

Corollaire 2.2 *Un automate explorant tous les graphes planaires de diamètre D et de degré maximum d a au moins $\Omega(D \log d)$ bits de mémoire.*

En conséquence, le mieux que puisse faire un automate de k bits de mémoire est d'explorer tous les graphes de diamètre au plus D et de degré maximum au plus d. Nous prouvons maintenant que ce but peut être atteint.

2.4.3 Borne supérieure correspondante

Dans cette sous-section, nous présentons un algorithme, appelé `DFS-Incrémental`, qui permet à un robot d'explorer tous les graphes de diamètre et de degré maximum suffisamment petits. L'algorithme est décrit dans la figure 2.6. Grossièrement, l'exploration est assurée en utilisant une suite de parcours en profondeur d'abord (DFS) de profondeur bornée, en augmentant graduellement la borne. Ces DFS sont effectués à partir de la position de départ u_0 du robot. Le robot garde en mémoire la suite de numéros de port permettant d'aller de sa position courante à la racine u_0 dans l'arbre de DFS. A la phase i, $i \geq 1$, le robot effectue un DFS de profondeur bornée par i. Dans le cas où nous nous donnons un robot $\mathcal{R}$ de k bits de mémoire, nous utilisons la variante k-`DFS-Incrémental`, qui est `DFS-Incrémental` dans lequel le robot vérifie en permanence la taille de mémoire allouée à cet instant. Si cette taille dépasse k bits, le robot s'arrête.

Théorème 2.3 *L'algorithme* `DFS-Incrémental` *autorise un robot à explorer tous les graphes. De plus, l'algorithme k-*`DFS-Incrémental` *explore tous les graphes de diamètre au plus D et de degré maximum au plus d, pourvu que $k \geq \alpha D \log d$, pour une certaine constante strictement positive α.*

Preuve. Démarrons le robot $\mathcal{R}$ du sommet u_0 dans le graphe G. Après que $\mathcal{R}$ ait effectué un DFS de profondeur p, il a visité tous les sommets à distance au plus p de u_0. Soit $p = D+1$ où D est le diamètre de G. Après la $p$ième phase de l'algorithme `DFS-Incrémental`, toutes les arêtes ont été traversées, et donc l'exploration est terminée. Si $k \geq \alpha D \log d$,

```
Entrées
 - d est le degré du sommet courant
 - provenance est le numéro de port menant à la position précédente du robot

Etat initial
 - pile := pile_vide()
 - borne := 1
 - profondeur := 1
 - descente := true
 - port_sortie := 1

Prochain port à prendre (dépend seulement de l'état) /* fonction lambda */
  port:=port_sortie

Fonction de transition /* fonction delta */
  si descente=vrai
    empiler(pile,provenance)
  finsi
  si est_vide(pile) et provenance=d
    borne:=borne+1
    profondeur:=profondeur+1
    descente:=vrai
    port_sortie:=1 ; terminer
  finsi
  si (provenance+1)=tête(pile) ou profondeur=borne
    profondeur:=profondeur-1
    descente:=faux
    port_sortie:=dépiler(pile) ; terminer
  sinon
    profondeur:=profondeur+1
    descente:=vrai
    port_sortie:=provenance+1 ; terminer
  finsi
```

FIG. 2.6 – Algorithme `DFS-Incrémental`

alors une pile de $D+1$ éléments de $\log d$ bits, et un nombre constant de variables (également de taille logarithmique), peuvent être stockés dans la mémoire du robot, pour $\alpha = O(1)$ suffisamment grand. Donc, quand $p = D+1$, l'exploration est effectuée, sans utiliser plus de k bits. En conséquence, tout graphe de diamètre au plus D et de degré maximum au plus d peut être exploré. □

Comme conséquence directe du théorème 2.3 et du corollaire 2.2, nous avons :

Corollaire 2.3 *Un robot explorant tous les graphes de diamètre au plus D et de degré maximum au plus d a besoin d'exactement $\Theta(D \log d)$ bits de mémoire.*

Notons enfin que l'algorithme `DFS-Incrémental` utilise une mémoire infinie pour certains graphes de taille bornée. Néanmoins, ce phénomène ne peut être évité par aucun algorithme d'exploration. En effet, de façon surprenante, n'importe quel automate infini explorant tous les graphes doit utiliser une quantité infinie de mémoire pour explorer certains graphes finis. En particulier, pour $d \geq 0$, notons $\mathcal{G}_d$ l'ensemble des graphes d-homogènes (cet ensemble est non vide car il contient, par exemple, l'hypercube Q_d). Nous avons le résultat suivant :

Théorème 2.4 *Pour tout automate (infini) $\mathcal{R}$ explorant tous les graphes, et pour tout $G \in \mathcal{G}_d$, $\mathcal{R}$ utilise une infinité d'états lorsqu'il explore G.*

Preuve. Soit $\mathcal{R}$ un automate explorant tous les graphes et soit $G \in \mathcal{G}_d$. Il découle du théorème 2.1 que $\mathcal{R}$ est un automate *infini* $(\mathcal{S}, s_{init}, \mathcal{F}, \delta, \lambda)$ ($\mathcal{S}$ est infini). Supposons, pour obtenir une contradiction, que $\mathcal{R}$ utilise K états de $\mathcal{S}$ lorsqu'il est exécuté dans G, en partant d'un certain sommet u_0. Soit $\mathcal{R}'$ l'automate obtenu en restreignant $\mathcal{R}$ à ces K états de $\mathcal{S}$. Plus précisément, $\mathcal{R}' = (\mathcal{S}', s_{init}, \mathcal{F} \cap \mathcal{S}', \delta', \lambda')$ où $\mathcal{S}'$ est l'ensemble des K états utilisés par $\mathcal{R}$ lorsqu'il explore G en partant de u_0, λ' est λ restreint à $\mathcal{S}'$, et δ' est δ restreint à $\mathcal{S}' \times (\mathbb{N} \cup \{\perp\}) \times \mathbb{N}$. Soit $\mathcal{G}_d(\mathcal{R}')$ l'ensemble des paires (H, v_0), où $H = (V, E)$ est un graphe et $v_0 \in V$, telles que, partant de v_0 dans H, $\mathcal{R}'$ visite uniquement des sommets de degré d et traverse uniquement des arêtes ayant leur deux numéros de port identiques. Soit (H, v_0) le piège pour $\mathcal{R}'$ construit dans la preuve du théorème 2.1. D'après notre construction, nous avons $(H, v_0) \in \mathcal{G}_d(\mathcal{R}')$. De plus, comme $G \in \mathcal{G}_d$, nous avons aussi $(G, u_0) \in \mathcal{G}_d(\mathcal{R}')$. Soit $(s_i)_{i \geq 0}$ la suite des états de $\mathcal{R}'$ lorsqu'il explore G en partant de u_0. Par construction de $\mathcal{R}'$, $(s_i)_{i \geq 0}$ est aussi la suite des états de $\mathcal{R}$ lorsque celui-ci explore G en partant de u_0. En fait, nous avons $\{s_i, i \geq 0\} = \mathcal{S}'$, et $s_{i+1} = \delta'(s_i, \lambda'(s_i), d) = \delta(s_i, \lambda(s_i), d)$. Par conséquent, la suite $(s_i)_{i \geq 0}$ est indépendante de l'instance (graphe, sommet de départ) $\in \mathcal{G}_d(\mathcal{R}')$, et indépendante de quel automate $\mathcal{R}$ ou $\mathcal{R}'$ explore cette instance. En particulier, la suite $(s_i)_{i \geq 0}$ est la même pour $\mathcal{R}$ et pour $\mathcal{R}'$ dans (H, v_0). Les suites de sommets visités par $\mathcal{R}$ et $\mathcal{R}'$ lorsqu'ils explorent H en partant de v_0 sont donc identiques. Comme (H, v_0) est un piège pour $\mathcal{R}'$, cette dernière observation est en contradiction avec le fait que $\mathcal{R}$ explore tous les graphes, incluant H. Donc $\mathcal{R}$ utilise un nombre infini d'états pour explorer G. □

2.5 Piège pour une équipe d'automates

Désormais, et jusqu'à la fin du chapitre, nous nous restreignons au cas $d = 3$. Outre le gain en clarté, cette restriction nous permet de construire des pièges planaires.

Dans cette section, nous nous intéressons à l'exploration de graphes par une équipe finie d'automates finis non coopératifs. Le résultat principal de cette section est la construction d'un piège de taille $O(qK)$ pour toute équipe de q automates non coopératifs de K états chacun maximum. Ce résultat est énoncé dans le théorème 2.5. Pour le prouver, nous avons besoin de deux lemmes auxiliaires. Le premier (Lemme 2.6) montre comment transformer un graphe homogène planaire quelconque en un squelette de piège. Le second (Lemme 2.7) montre comment étendre ce squelette de piège en un piège homogène planaire. Ces résultats sont utilisés à chaque étape de la récurrence de la preuve du théorème 2.5.

2.5.1 Transformation d'un graphe en un squelette de piège

Supposons qu'un automate irréductible $\widehat{\mathcal{A}} = (\mathcal{S}, s_{init}, f, \lambda)$ est placé en un sommet x_0 d'un graphe homogène planaire G dans son état initial s_{init}. Nous voulons créer un squelette de piège pour $\widehat{\mathcal{A}}$ en étendant G au niveau d'une arête $\{v, v'\}$. De plus, l'extension doit être de taille $O(K)$. Voir Figure 2.7. L'extension s'opère en coupant $\{v, v'\}$ pour créer

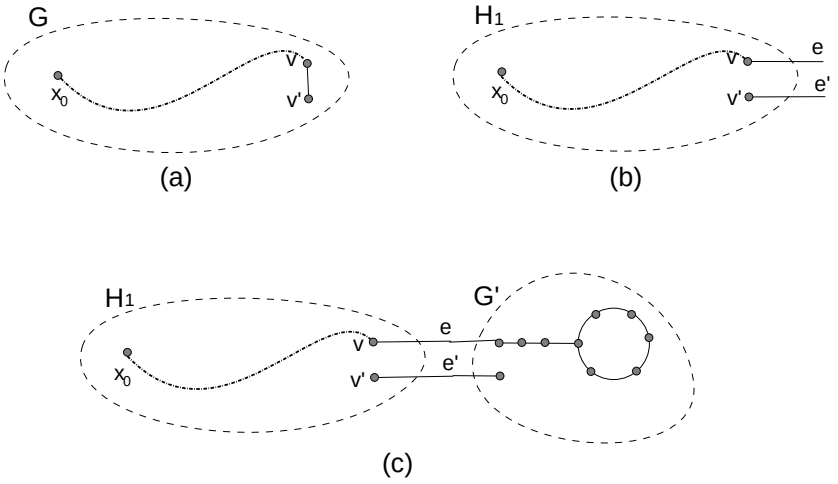

FIG. 2.7 – Etendre un graphe à partir d'une arête en un squelette de piège

deux arêtes pendantes (cf. Figure 2.7(b)). Le sommet v, resp. v', est incident à la demi-arête pendante e, resp. e'. Soit H_1 le graphe résultant. Un graphe partiel G' de $O(K)$ sommets est attaché à e et à e' (cf Figure 2.7(c)) de telle façon que le graphe résultant H soit un squelette de piège pour $\widehat{\mathcal{A}}$. Si l'automate $\widehat{\mathcal{A}}$ ne traverse jamais l'arête $\{v, v'\}$ lorsqu'il démarre de x_0 dans G, alors on pose $H = H_1$. Autrement, l'extension ajoutée à G est assez simple : soit on ajoute un chemin connecté à un cycle (basés sur $\bar{G}(\widehat{\mathcal{A}})$) comme illustré dans la figure 2.7(c), soit on connecte e et e' par un chemin (étiqueté de façon appropriée).

Plus précisément, considérons l'empreinte de $\widehat{\mathcal{A}}$, $emp(\widehat{\mathcal{A}}) = W_0W^*$, $|W_0W| = K$, où p_i est la i-ème lettre de $emp(\widehat{\mathcal{A}})$. Considérons la suite de configurations de $\widehat{\mathcal{A}}$

$$(x_0, s_0) \to (x_1, s_1) \to \dots$$

où $s_0 = s_{init}$ est l'état initial de $\widehat{\mathcal{A}}$. Soit $p_{i+1} = \lambda(s_i)$. Supposons que $\widehat{\mathcal{A}}$ traverse $\{v, v'\}$ pour la première fois à l'étape i, $i \geq 1$, c'est-à-dire en passant de la configuration (x_{i-1}, s_{i-1}) à la configuration (x_i, s_i). Supposons, sans perte de généralité, qu'il la traverse à ce moment de v vers v'. Donc $x_{i-1} = v$ et $x_i = v'$. p_i est l'étiquette de $\{v, v'\}$. En d'autres termes, si nous coupons l'arête pour obtenir deux demi-arêtes pendantes e et e', alors $\widehat{\mathcal{A}}$ traverse e.

Nous considérons deux cas, dépendant du moment où $\widehat{\mathcal{A}}$ traverse e, dans un état du chemin ou dans un état du cycle de $G(\widehat{\mathcal{A}})$.

Cas 1. Supposons que $\widehat{\mathcal{A}}$ traverse e à l'étape $i \leq |W_0|$. Donc p_i appartient à W_0. Dans ce cas, nous utilisons le graphe non orienté de $\widehat{\mathcal{A}}$, $\bar{G}(\widehat{\mathcal{A}})$, et nous construisons la version sans arête multiple ni boucle, $\bar{G}_1(\widehat{\mathcal{A}})$, comme dans la sous-section 2.3.1 en ajoutant au plus trois sommets. Nous collons la partie de $\bar{G}_1(\widehat{\mathcal{A}})$ qui commence après p_i à e. Plus précisément, nous connectons à e le chemin de longueur $|W_0| - i$ dont l'extrémité est notée w. Les arêtes de ce chemin sont étiquetées $p_{i+1}, \dots, p_{|W_0|}$. En w, nous ajoutons le cycle de $\bar{G}_1(\widehat{\mathcal{A}})$. Parmi les sommets ajoutés, seul w est de degré 3. Nous complétons le graphe partiel par des demi-arêtes pendantes pour que chaque sommet soit de degré 3. Soit H le graphe obtenu. Notons que nous avons ajouté au plus $K+2$ nouveaux sommets.

Cas 2. Si $\widehat{\mathcal{A}}$ traverse e à l'étape $i > |W_0|$, alors il transite dans l'état s du cycle de $G(\widehat{\mathcal{A}})$ après avoir traversé e. Supposons que c'est le j-ième état du cycle (rappelons que le cycle est supposé commencer dans le dernier état du chemin de $G(\widehat{\mathcal{A}})$). Cela veut dire que, après avoir traversé e, $\widehat{\mathcal{A}}$ traversera les arêtes étiquetées $p_{|W_0|+j}, p_{|W_0|+j+1}, \dots$

Soit x le sommet de H_1 atteint par $\widehat{\mathcal{A}}$ après $|W_0|$ étapes, soit W^{-1} la suite W écrite en sens inverse, et soit $\widehat{\mathcal{A}}^{-1}$ l'automate traversant les arêtes étiquetées $(W^{-1})^*$. Donc, quand $\widehat{\mathcal{A}}^{-1}$ démarre en x, $\widehat{\mathcal{A}}^{-1}$ procède comme $\widehat{\mathcal{A}}$ lorsque ce dernier atteint x après $|W_0|$ étapes, mais à l'envers. Soit $\widehat{\mathcal{A}}^*$ l'automate traversant les arêtes étiquetés W^*, c'est-à-dire l'automate dérivé de $\widehat{\mathcal{A}}$ par la suppression des états et des transitions impliqués dans W_0.

Assertion 2.1 *En démarrant en x, $\widehat{\mathcal{A}}^{-1}$ finit par traverser l'une des demi-arêtes pendantes en v ou en v'.*

Preuve. Supposons par contradiction que $\widehat{\mathcal{A}}^{-1}$ ne traverse aucune des demi-arêtes e et e'. Soit s l'état dans lequel démarre $\widehat{\mathcal{A}}^{-1}$ en x. Nous montrons d'abord que $\widehat{\mathcal{A}}^{-1}$ finit par retourner en x dans le même état s. Pour ce faire, considérons la suite de configurations commençant par (x, s)

$$(x_0, s_0) \to (x_1, s_1) \to \cdots$$

où $(x, s) = (x_0, s_0)$. La suite contient nécessairement deux configurations égales, disons $(x_i, s_i) = (x_{i+k}, s_{i+k})$, pour un $i \geq 0$ et un $k \geq 1$, car H_1 et $\widehat{\mathcal{A}}^{-1}$ sont tous deux finis. Supposons que k est le plus petit possible. Si $i = 0$, le lemme est prouvé, donc supposons

$i > 0$. Nous allons prouver que $(x_{i-1}, s_{i-1}) = (x_{i+k-1}, s_{i+k-1})$, ce qui implique $(x_0, s_0) = (x_k, s_k)$.

Notons que $\widehat{\mathcal{A}}^{-1}$ se déplace de x_{i-1} à x_i le long d'une arête étiquetée $\lambda(s_{i-1})$. Quand $\widehat{\mathcal{A}}^{-1}$ retourne finalement dans la même configuration (x_{i+k}, s_{i+k}), l'état s_{i+k-1} est égal à s_{i-1} (tous les états considérés sont dans le cycle de $G(\widehat{\mathcal{A}}^{-1})$ car l'état s est lui-même dans ce cycle). On a donc $\lambda(s_{i-1}) = \lambda(s_{i+k-1})$. Il s'ensuit que l'arête $\{x_{i+k-1}, x_{i+k}\}$ a aussi l'étiquette $\lambda(s_{i-1})$. Finalement $x_{i+k-1} = x_{i-1}$ puisque $x_{i+k} = x_i$ et que G est à arêtes colorées.

Donc $\widehat{\mathcal{A}}^{-1}$ finit par retourner en x dans le même état, et donc le parcours dans H_1 est un parcours fermé. Ce chemin parcouru à l'envers est exactement ce que $\widehat{\mathcal{A}}^*$ parcourt depuis x. Donc $\widehat{\mathcal{A}}^*$ ne traverse aucune des deux demi-arêtes e et e'. En conséquence, $\widehat{\mathcal{A}}$ non plus ne les traverse pas, une contradiction. □

D'après l'assertion 2.1 nous pouvons considérer l'état atteint par $\widehat{\mathcal{A}}^{-1}$ après qu'il ait traversé une des demi-arêtes pendantes. Supposons que c'est le k-ième état du cycle de $G(\widehat{\mathcal{A}})$. Nous considérons deux sous-cas, selon que $\widehat{\mathcal{A}}^{-1}$ traverse la même demi-arête que $\widehat{\mathcal{A}}$ ou non.

Cas 2.1. L'automate $\widehat{\mathcal{A}}^{-1}$ traverse la demi-arête e, c'est-à-dire la même que $\widehat{\mathcal{A}}$. Cela implique que la k-ième étiquette de W est égale à la $(j-1)$-ième, qui est aussi l'étiquette de e. Nous considérons la section du cycle de $G(\widehat{\mathcal{A}})$ du j-ième au k-ième état. Les arêtes extrémales de cette section ont l'étiquette de e. Nous considérons maintenant le mot suivant : $W' = W(j-1)W(j)W(j+1)\dots W(k-1)W(k)W(k+1)\dots W(j-1)W(j)W(j+1)\dots W(k-1)W(k)W(k+1)\dots W(j-1)W(j)W(j+1)\dots W(k-1)W(k)$. (Notons que $W(j-1) = W(k)$ et $|W'| \geq 2 \times |W| + 2$). Les deux robots $\widehat{\mathcal{A}}$ et $\widehat{\mathcal{A}}^{-1}$ ne peuvent pas suivre indéfiniment le même chemin après avoir traversé e : autrement, cela signifierait qu'en les déplaçant tous deux à l'envers, ils suivraient aussi indéfiniment le même chemin, ce qui est impossible car les deux robots ont pris des chemins différents au sommet x dans le passé. De plus, les deux robots doivent se séparer après au plus $|W|$ étapes, et puisque $|W'| \geq 2 \cdot |W| + 2$, ils doivent se séparer après au moins une étape et au plus $|W| - 1$ étapes. Maintenant, si les deux automates se séparent l'un de l'autre à un certain moment après avoir traversé l'arête e, considérons le plus petit entier l tel que $W(j+l) \neq W(k-1-l)$, c'est-à-dire l'endroit le plus proche de e où les deux automates se sont séparés. Puisque $W(j-1) = W(k)$, on a $l \geq 1$. Par définition de l, nous avons $W(j+l-1) = W(k-l)$. Puisque les automates considérés sont réduits, nous avons aussi $W(j+l-1) \neq W(j+l)$. Toujours par définition de l, on obtient $W(j+l) \neq W(k-1-l)$. Finalement, parce que nous considérons des automates réduits et que nous avons $W(j+l-1) = W(k-l)$, on obtient $W(j+l-1) \neq W(k-1-l)$. En résumé, cela signifie que $W(j+l-1)$, $W(j+l)$, et $W(k-1-l)$ sont disjoints deux à deux. Nous sommes maintenant prêts à construire le graphe suivant : à partir de e, nous disposons un chemin élémentaire qui se termine en $W(j+l-1)$ au sommet w, et de ce dernier sommet un cycle W'' va de $W(j+l)$ à $W(k-l-1)$. Puisque $W(j+l) \neq W(k-1-l)$ (voir ci-dessus), on a $|W''| > 2$. Nous ajoutons à w un cycle de longueur $|W''|$ étiqueté W'', commençant et se terminant en w, de telle façon qu'une fois que $\widehat{\mathcal{A}}$ et $\widehat{\mathcal{A}}^{-1}$ ont atteint w, chacun traverse ce cycle dans une

direction opposée, et revient en w dans l'état approprié pour continuer le long du chemin en direction de e. Parmi les sommets ajoutés, seul w est de degré 3. Nous complétons le graphe partiel par des demi-arêtes pendantes pour que chaque sommet soit de degré 3. Soit H le graphe obtenu. Notons que nous avons ajouté au plus $2K+1$ nouveaux sommets.

Cas 2.2. L'automate $\widehat{\mathcal{A}}^{-1}$ traverse la demi-arête e', c'est-à-dire pas la même que $\widehat{\mathcal{A}}$. Supposons que $\widehat{\mathcal{A}}^{-1}$ est dans l'état s quand il est en v'. Nous considérons de nouveau la section du cycle de $G(\widehat{\mathcal{A}})$ du j-ième état au k-ième état (si la section est de longueur 1, nous l'étendons avec W pour s'assurer que la section ait au moins un sommet interne). Nous connectons e et e' par un chemin avec les étiquettes de cette section. Donc, quand $\widehat{\mathcal{A}}$ traverse la demi-arête e, il suit le chemin nouvellement rajouté, et va en v' dans l'état approprié, s, pour continuer le long du chemin de $\widehat{\mathcal{A}}^{-1}$ mais dans l'autre sens, et finalement retourner en x. Nous complétons le graphe partiel par des demi-arêtes pendantes pour que chaque sommet soit de degré 3. Soit H le graphe obtenu. Notons que nous avons ajouté au plus K nouveaux sommets.

Lemme 2.6 *Le graphe H est un graphe partiel homogène tel qu'il en existe une représentation plane dans laquelle les demi-arêtes pendantes appartiennent à une même face. De plus, H est un squelette de piège pour $\widehat{\mathcal{A}}$ commençant en x_0, avec au plus $2K+1$ sommets de plus que G.*

Preuve. Il découle directement de la construction et du fait que G est homogène et planaire que H un graphe partiel homogène tel qu'il en existe une représentation plane dans laquelle les demi-arêtes pendantes appartiennent à une même face. Le nombre de sommets de H est compté dans les trois cas précédents et n'excède pas $2K+1$.

La preuve du cas 1 se déroule comme suit, les autres cas sont similaires. Supposons que $\widehat{\mathcal{A}}$ traverse e à l'étape $i \leq |W_0|$. Dans ce cas, $\widehat{\mathcal{A}}$ ne traverse pas l'arête e' de H. Observons que $\widehat{\mathcal{A}}$ est piégé dans la portion de $\bar{G}_1(\widehat{\mathcal{A}})$ ajoutée à e. Cela vient du fait que $\widehat{\mathcal{A}}$ est dans la configuration (x_{i-1}, s_{i-1}) avant de traverser e, et dans la configuration (x_i, s_i) après l'avoir traversée. A ce moment, il est au début de la portion de $\bar{G}_1(\widehat{\mathcal{A}})$ ajoutée. Il continuera donc à parcourir les arêtes de cette portion sans chercher à prendre les demi-arêtes pendantes, comme dans le lemme 2.4. □

2.5.2 Du squelette au piège

Considérons un automate quelconque $\mathcal{A}$ et un graphe 3-homogène planaire G. Soit x_0 un sommet de G. Soit $\widehat{\mathcal{A}}$ un automate irréductible obtenu à partir de $\mathcal{A}$. Dans cette sous-section, nous montrons comment obtenir un piège planaire 3-homogène pour $\mathcal{A}$ à partir du squelette de piège H pour $\widehat{\mathcal{A}}$ obtenu à partir de G par la construction de la sous-section précédente (cf. Lemme 2.6). Nous avons d'abord besoin de deux résultats techniques.

Assertion 2.2 *Soit H un graphe partiel 3-homogène. Pour $\ell = 0, 1, 2$, soit* parité(ℓ) *la parité du nombre de demi-arêtes pendantes étiquetées ℓ. Pour tout $\ell, \ell' \in \{0, 1, 2\}$,* parité$(\ell)$ = parité(ℓ').

Preuve. Une arête de H étiquetée ℓ peut être considérée comme deux demi-arêtes non pendantes étiquetées ℓ. Pour $\ell \in \{0, 1, 2\}$, soit t_ℓ le nombre total de demi-arêtes de H étiquetées ℓ, et p_ℓ, resp. np_ℓ, le nombre de demi-arêtes de H pendantes, resp. non pendantes, étiquetées ℓ. Tous les sommets de H sont exactement de degré 3 et sont incidents à une demi-arête de chaque étiquette. D'où $t_0 = t_1 = t_2 = n$, où n est le nombre de sommets du graphe. Dans H, si une demi-arête n'est pas pendante, elle forme une arête complète avec une autre demi-arête non pendante de même étiquette. Donc tous les np_ℓ sont pairs. Puisque $t_\ell = p_\ell + np_\ell$, t_ℓ et p_ℓ ont la même parité et donc tous les p_ℓ ont la même parité. □

Assertion 2.3 *Soit H un graphe partiel 3-homogène tel qu'il en existe une représentation plane dans laquelle les demi-arêtes pendantes appartiennent à une même face. Soit p le nombre de demi-arêtes pendantes de H. Alors on peut étendre H en un graphe homogène planaire en ajoutant au plus $2p$ sommets.*

Preuve. Nous montrons le résultat par récurrence sur p. Le résultat est trivialement vrai pour $p = 0$. Supposons le résultat vrai pour tout $i < p$, avec $p \geq 1$. Considérons un graphe partiel H vérifiant les hypothèses de l'assertion et ayant exactement p demi-arêtes pendantes. Si H est réduit à un sommet et trois demi-arêtes pendantes, le sommet peut être complété en le graphe complet à 4 sommets et le résultat est vrai. Sinon, pour des raisons de parité liées à l'assertion 2.2, il existe au moins deux sommets possédant une ou plusieurs demi-arêtes pendantes. Choisissons deux demi-arêtes pendantes e et e' incidentes à deux sommets distincts v et v', respectivement, telles que, si on les reliait, le graphe partiel obtenu aurait encore une représentation plane dans laquelle les demi-arêtes pendantes restantes appartiennent à une même face.

Supposons d'abord que e et e' ont des numéros de port différents ℓ et ℓ', respectivement. On ajoute alors un nouveau sommet u. La demi-arête e, resp. e', est prolongée en une arête d'étiquette ℓ, resp. ℓ', reliant v, resp. v', au nouveau sommet u. Une demi-arête pendante de numéro de port $\ell'' \notin \{\ell, \ell'\}$ est ajoutée à u pour que ce dernier soit de degré 3. Clairement, le graphe partiel obtenu H' est homogène et possède une représentation plane dans laquelle ses $p - 1$ demi-arêtes pendantes appartiennent à la même face. Par application de l'hypothèse de récurrence, H' peut être étendu en un graphe homogène planaire en ajoutant au plus $2(p - 1)$ sommets. Donc H peut être étendu en un graphe homogène planaire en ajoutant au plus $2(p - 1) + 1 \leq 2p$ sommets.

Supposons maintenant que e et e' ont le même numéro de port ℓ. S'il n'existe pas encore d'arête $\{v, v'\}$, alors nous formons une arête d'étiquette ℓ avec les deux demi-arêtes e et e'. Sinon, pour éviter les arêtes multiples, nous relions e et e' par le gadget à quatre sommets décrit dans la figure 2.8, dans laquelle $\ell = 0$. Dans les deux cas, le graphe partiel obtenu H' est clairement homogène et possède une représentation plane dans laquelle ses $p - 2$ demi-arêtes pendantes appartiennent à la même face. Par application de l'hypothèse de récurrence, H' peut être étendu en un graphe homogène planaire en ajoutant au plus $2(p-2)$ sommets. Donc H peut être étendu en un graphe homogène planaire en ajoutant au plus $2(p - 2) + 4 = 2p$ sommets, ce qui conclut la preuve de l'assertion. □

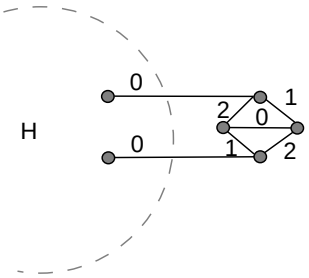

FIG. 2.8 – Le gadget reliant deux demi-arêtes

Revenons au squelette de piège H construit dans la sous-section précédente. Seuls les sommets ajoutés à G et les deux sommets v et v' de G sont éventuellement incidents à des demi-arêtes pendantes. H étant connexe, chacun de ces sommets a au plus deux demi-arêtes pendantes incidentes. En fait, notre façon de construire H implique qu'il y a au plus une demi-arête pendante incidente à chaque sommet. H a donc au plus $(2K+1)+2 = 2K+3$ demi-arêtes pendantes. D'après l'assertion 2.3, H peut être étendu en un graphe homogène planaire H^{hom} ayant au plus $4K+6$ sommets de plus que H, et donc au plus $6K+7$ sommets de plus que G.

La paire (H^{hom}, x_0) est un piège pour $\widehat{\mathcal{A}}$ mais pas nécessairement pour $\mathcal{A}$. Pour construire un piège pour $\mathcal{A}$, nous ajoutons une "tour" à H^{hom}. Plus précisément, soit $\{w, w'\}$ une arête de H^{hom} mais pas de H. L'automate réduit $\widehat{\mathcal{A}}$ ne traverse jamais cette arête lorsqu'il démarre de x_0 dans H^{hom}. Coupons l'arête pour produire deux demi-arêtes pendantes f et f'. Ajoutons une "tour" de hauteur $K+1$ connectée à f et f', et un gadget fermant la tour, comme l'illustre la figure 2.9. Les deux sommets internes du gadget au sommet de la tour sont notés v_1 et v'_1. Ajoutons des étiquettes à la tour et au gadget pour

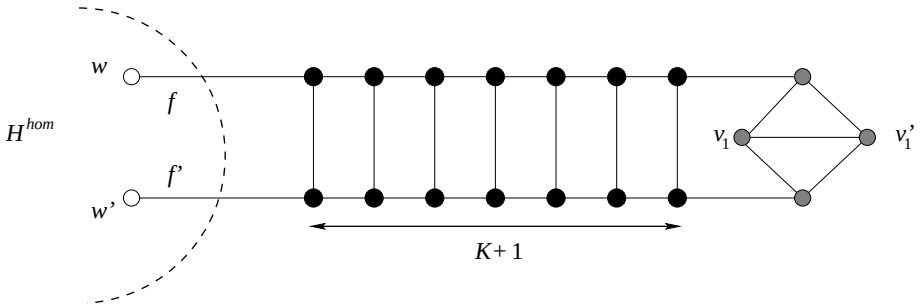

FIG. 2.9 – La tour surmontée du gadget

rendre le graphe à arêtes colorées. Soit $H(G, \mathcal{A})$ le graphe obtenu. $H(G, \mathcal{A})$ est homogène.

Lemme 2.7 *Soit G un graphe planaire 3-homogène quelconque. Soit x_0 un sommet et $\{v, v'\}$ une arête de G. Considérons un automate arbitraire $\mathcal{A}$ ayant K états. Alors il est possible d'étendre G en l'ouvrant au niveau de l'arête $\{v, v'\}$ pour obtenir un graphe homogène planaire $H(G, \mathcal{A})$ tel que ce graphe a au plus $8K+13$ sommets de plus que G et tel que $(H(G, \mathcal{A}), x_0)$ est un piège pour $\mathcal{A}$.*

Preuve. Considérons le graphe $H(G, \mathcal{A})$ construit précédemment à partir de H^{hom} par l'ajout d'une tour en lieu et place de l'arête $\{w, w'\}$. L'automate réduit $\widehat{\mathcal{A}}$ ne traverse jamais l'arête $\{w, w'\}$, et donc n'entre pas dans la tour. D'après le lemme 2.3, la trajectoire de $\mathcal{A}$ n'est jamais à une distance plus grande que K de la trajectoire de $\widehat{\mathcal{A}}$. Puisque la tour est de hauteur $K + 1$, $\mathcal{A}$ n'atteint jamais le sommet de la tour. En conséquence, $\mathcal{A}$ n'explore jamais les sommets v_1 et v'_1, et $(H(G, \mathcal{A}), x_0)$ est un piège pour $\mathcal{A}$.

Il reste à compter le nombre de sommets de $H(G, \mathcal{A})$. Le graphe H^{hom} a au plus $6K+7$ sommets de plus que G. La tour a $2K + 6$ sommets, et donc $H(G, \mathcal{A})$ a au plus $8K + 13$ sommets. □

2.5.3 Piège pour une équipe d'automates non coopératifs

Grâce au lemme précédent, nous sommes en mesure de prouver le résultat principal de cette section.

Théorème 2.5 *Pour toute équipe $\mathcal{T}$ de q automates non coopératifs à K états, il existe un graphe 3-homogène planaire G et deux paires de sommets voisins $\{u, u'\}$ et $\{v, v'\}$ tels que (1) l'arête $\{u, u'\}$ est étiquetée 0, (2) en commençant en u ou en u', tout automate de $\mathcal{T}$ échoue à explorer les sommets v et v', et (3) G a $O(qK)$ sommets.*

Preuve. La preuve est par récurrence sur $q \geq 0$. La base correspond à $q = 0$. Le graphe correspondant est explicité par la figure 2.10.

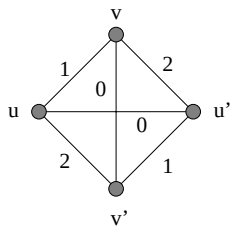

FIG. 2.10 – Base de la récurrence

Pour l'hypothèse de récurrence, supposons que le théorème 2.5 est vrai pour q. Montrons qu'il est vrai pour $q + 1$. Soit $\mathcal{T}$ une équipe de $q + 1$ automates non coopératifs à K états. Soit $\mathcal{A}$ un automate de l'équipe. Par hypothèse de récurrence, soit G_q un graphe 3-homogène de n sommets (où n est $16qK + O(q)$) ayant deux paires de sommets voisins $\{u, u'\}$ et $\{v, v'\}$ avec l'arête $\{u, u'\}$ étiquetée 0, et tel que, démarrant en u ou en u', tout robot de $\mathcal{T} \setminus \{\mathcal{A}\}$ échoue à explorer les sommets v et v'. Nous construisons maintenant un graphe G_{q+1} qui vérifie le théorème 2.5 pour $\mathcal{T}$.

Considérons le graphe $G' = H(G, \mathcal{A})$ décrit dans le lemme 2.7 en considérant u comme le sommet de départ et $\{v, v'\}$ comme l'arête de G à ouvrir pour étendre le graphe. Soient v_1 et v'_1 les deux sommets internes du gadget au sommet de la tour de G'. Nous définissons maintenant G_{q+1} comme le graphe $H(G', \mathcal{A})$ en considérant u' comme le sommet de départ et $\{v_1, v'_1\}$ comme l'arête de G à ouvrir pour étendre le graphe. Soient v_2 et v'_2 les deux

sommets internes du gadget au sommet de la tour de G'. Nous affirmons que G_{q+1} et les deux paires de voisins $\{u, u'\}$ et $\{v_2, v_2'\}$ vérifient les propriétés du théorème 2.5 pour l'équipe $\mathcal{T}$.

Par hypothèse de récurrence et d'après le lemme 2.7, G_{q+1} est un graphe 3-homogène planaire. Tout automate de $\mathcal{T}$ échoue à traverser v_2 et v_2' en démarrant de u ou u'. En effet, par hypothèse de récurrence, un robot de $\mathcal{T}\setminus\{A\}$ démarrant en u ou en u' n'explore jamais v et v' dans G_q et donc ne traverse pas non plus les sommets rajoutés pour construire G_{q+1}, en particulier v_2 et v_2'. En démarrant de u, l'automate $\mathcal{A}$ n'arrive pas à explorer v_1 et v_1' dans G'. Cette arête étant celle "ouverte" pour construire G_{q+1} à partir de H_2, $\mathcal{A}$ n'atteint ni v_2 ni v_2' dans G_{q+1}. Finalement, par construction de G_{q+1} à partir de G', $\mathcal{A}$ n'explore pas non plus ces deux sommets en démarrant de u'.

Pour finir la preuve, il reste à calculer la taille de G_{q+1}. D'après le lemme 2.7, nous avons $|G'| \leq |G_q| + 8K + 13$ et $|G_{q+1}| \leq |G'| + 8K + 13$. En résumé, nous avons $|G_{q+1}| \leq |G| + 16K + 26$ et donc, par hypothèse de récurrence $|G_{q+1}| \leq 16(q+1)K + O(q)$, ce qui conclut la preuve du théorème. □

Simplement en réécrivant le théorème 2.5, nous dérivons une borne sur la taille du plus petit piège pour une équipe de q automates non coopératifs à K états, améliorant le résultat de Rollik [Rol80] :

Corollaire 2.4 *Pour toute équipe de q automates non coopératifs à au plus K états, il existe un piège 3-homogène planaire de taille $O(qK)$.*

En utilisant notre piège dans la construction de Rollik [Rol80] pour les équipes d'automates localement coopératifs, nous obtenons :

Corollaire 2.5 *Pour toute équipe de q automates localement coopératifs ayant au plus K états, il existe un piège de taille $\tilde{O}(K^{K^{\cdot^{\cdot^{K}}}})$, avec $q+1$ niveaux d'exponentielle.*

Rappelons que la construction de Rollik engendre $2q+1$ niveaux d'exponentielle.

2.6 Exploration avec arrêt

Dans cette section, nous considérons le problème de l'exploration avec arrêt. Comme nous l'avons déjà noté dans le paragraphe introduisant la définition 9, un robot doit être muni de cailloux pour explorer tous les graphes avec arrêt. Il est connu qu'aucun automate fini muni d'un nombre fini de cailloux ne peut explorer tous les graphes [Rol80]. D'un autre côté, un robot de mémoire non bornée peut explorer tous les graphes avec arrêt, en utilisant un caillou [DJMW91]. Nous bornons ici la taille de la mémoire du robot nécessaire et suffisante pour l'exploration avec arrêt.

2.6.1 Piège pour un automate muni d'un caillou

Le théorème suivant montre qu'un automate muni d'un caillou réalisant l'exploration avec arrêt dans tous les graphes d'au plus n sommets a besoin de $\Omega(n^{1/3})$ états ou, de façon équivalente, $\Omega(\log n)$ bits de mémoire. Ce résultat reste valable pour l'exploration perpétuelle.

Théorème 2.6 *Pour tout automate à K états muni d'un caillou, il existe un piège planaire de degré maximum 3 ayant au plus $O(K^3)$ sommets.*

Preuve. Soit $\mathcal{A}$ un robot à K états muni d'un caillou. Nous construisons un piège de taille $O(K^3)$ pour $\mathcal{A}$. Pour ce faire, nous considérons la restriction de $\mathcal{A}$ aux transitions telles que, en entrée, l'automate ne détient pas le caillou et ne voit pas le caillou sur le sommet courant. Ceci définit un automate de Moore sans caillou, comme les automates étudiés jusqu'ici dans ce chapitre. Pour chaque état s de cet automate, nous considérons l'automate $\mathcal{A}_s$, en tout point identique à $\mathcal{A}$, sauf en ce qui concerne l'état initial, qui est s pour l'automate $\mathcal{A}_s$. Soit $\mathcal{T} = \{\mathcal{A}_s\}$ l'ensemble de ces automates. Donc $|\mathcal{T}| \leq K$.

Soit G un graphe vérifiant le théorème 2.5 pour l'ensemble $\mathcal{T}$. Retirons les arêtes $\{u, u'\}$ et $\{v, v'\}$ de G. Considérons deux copies du graphe résultant, avec les quatre sommets de degré 2 indicés par le numéro de la copie, 1 et 2. Ces sommets sont reconnectés de la façon suivante. Soit ℓ l'étiquette de l'arête supprimée $\{v, v'\}$. Créons deux arêtes $\{v_1, v'_2\}$ et $\{v'_1, v_2\}$ d'étiquette ℓ. Le graphe obtenu est noté G_1 (voir Figure 2.11).

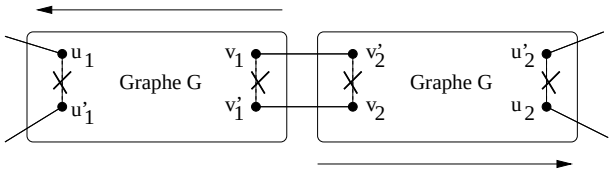

FIG. 2.11 – Le graphe G_1

Considérons un arbre ternaire infini modifié comme suit. Chaque sommet est remplacé par un cycle de 6 sommets. Les arêtes des cycles sont étiquetées alternativement 1 et 2. Ensuite, les arêtes de l'arbre infini sont remplacées par deux arêtes "parallèles" étiquetées 0, comme indiqué sur la figure 2.12. Le graphe obtenu est noté T.

Les deux graphes G_1 et T sont composés entre eux en remplaçant chaque paire $\{\{x, y\}, \{x', y'\}\}$ d'arêtes parallèles dans T par une copie de G_1. Plus précisément, x, y, x', y' sont connectés respectivement aux sommets u_1, u'_2, u'_1, u_2 de G_1. Ces nouvelles arêtes sont étiquetées 0. Le graphe obtenu est noté G_2. Une "méta-arête" de G_2 est définie comme une copie de G_1 remplaçant un couple d'arêtes parallèles de T.

Par définition de $\mathcal{T}$ et de G, l'automate $\mathcal{A}$ est incapable de traverser une méta-arête de G_2 sans l'aide du caillou[1]. Nous modifions maintenant G_2 pour obtenir un graphe G_3 que l'automate $\mathcal{A}$ n'arrive pas à explorer, même avec le caillou. G_3 contient $O(K)$ cycles

[1]Puisque les arêtes $\{u, u'\}$ de G sont "ouvertes", la preuve nécessite de considérer la dernière fois que l'automate est dans un sommet u ou u'.

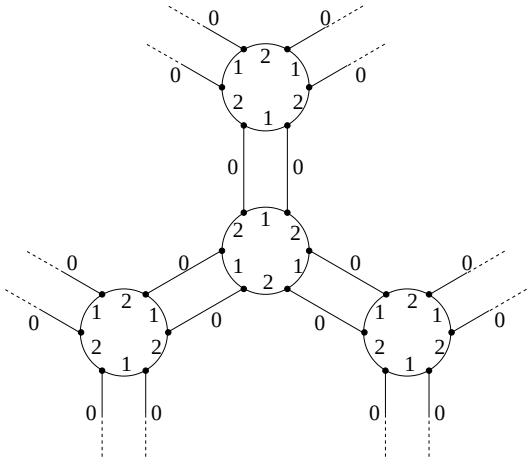

FIG. 2.12 – L'arbre infini modifié T

à 6 sommets de T, et donc a au plus $O(K^3)$ sommets. La transformation de G_2 à G_3 est technique et très similaire à la construction utilisée dans [Rol80]. Par conséquent, nous esquissons seulement la construction de G_3, en éludant les détails techniques. Puisque l'automate ne peut aller d'un cycle de 6 sommets de G_2 à un autre sans utiliser le caillou, nous définissons les étapes *clés* comme les étapes pendant lesquelles l'automate quitte pour la dernière fois un cycle à 6 sommets avec le caillou avant d'entrer dans la méta-arête et d'arriver dans un autre cycle à 6 sommets avec le caillou. Comme son nombre d'états est fini, $\mathcal{A}$ finit par être deux fois dans le même état lors de ces étapes clés, en deux sommets w et w'. En utilisant la même technique que dans [Rol80], nous identifions les sommets w et w'. Ceci donne le graphe G_3 ayant les propriétés désirées. A savoir, G_3 est constitué de $O(K)$ cycles à 6 sommets, et donc de $O(K)$ méta-arêtes. Chaque méta-arête est une copie de G_1. Comme G_1 a $O(K^2)$ sommets d'après le théorème 2.5, G_3 a $O(K^3)$ sommets. □

Corollaire 2.6 *Un automate muni d'un caillou a besoin de* $\Omega(\log n)$ *bits de mémoire pour réaliser l'exploration perpétuelle ou avec arrêt de tous les graphes de taille* n*, même planaires et de degré maximum* 3*.*

2.6.2 Algorithme d'exploration avec arrêt

Théorème 2.7 *Il existe un algorithme permettant à un robot muni d'un caillou de réaliser l'exploration de tous les graphes avec arrêt. De plus, cet algorithme utilise au plus* $O(D \log d)$ *bits de mémoire dans les graphes de diamètre* D *et de degré maximum* d*.*

Preuve. Nous décrivons un algorithme appelé `DFS-avec-arrêt`, qui permet à un robot d'explorer tous les graphes avec arrêt. L'exploration est effectuée par un parcours en profondeur d'abord (DFS) du graphe. Soit u_0 la position initiale du robot. Le caillou est déposé sur u_0 et y restera jusqu'à la fin de l'exploration. Celle-ci procède en une suite de

phases. Lors de la phase $i \geq 1$, le robot effectue un DFS de profondeur i. A tout moment lors de chaque phase, le robot garde en mémoire la suite de numéros de port menant de la position courante à la racine u_0 de l'arbre DFS. Cette suite occupe $O(i \log d)$ bits de mémoire pour la phase i, dans un graphe de degré maximum d.

Au début de la phase i, le robot fixe la variable *stop* à $vrai$. Le robot traverse les arêtes incidentes à un sommet u dans l'ordre croissant de leur numéro de port en u. Quand le robot quitte le sommet courant u et arrive dans un certain sommet v, il procède comme suit. Si le caillou est en v, alors il revient en arrière en u. Autrement, si la profondeur courante est inférieure ou égale à $i-1$, le robot continue son DFS. Si la profondeur courante du DFS est égale à i, alors le robot contrôle si v a déjà été visité ou non lors d'une phase précédente. Pour ce faire, le robot effectue un DFS auxiliaire de profondeur $i-1$ depuis v. De nouveau, $O(i \log \Delta)$ bits de mémoire sont utilisés pour stocker la suite de numéros de port menant à v dans l'arbre du DFS auxiliaire. Si le robot trouve le caillou lors de l'exécution du DFS auxiliaire, alors v est à distance au plus $i-1$ de u_0, et donc a déjà été visité lors d'une phase précédente. Si le robot ne trouve pas le caillou lors de l'exécution du DFS auxiliaire, alors le sommet v est à distance exactement i de u_0 et le robot positionne la variable *stop* à $faux$. Après avoir terminé le DFS de la phase i, le robot s'arrête si et seulement si $stop = vrai$. Sinon, il continue l'exploration en commençant la phase $i+1$.

Clairement, le robot s'arrête après la phase $D+1$ dans un graphe de diamètre D. Les besoins en mémoire de cet algorithme d'exploration sont dominés par le stockage des suites de numéros de port correspondant aux deux chemins, l'un pour le DFS principal, l'autre pour le DFS auxiliaire. Ces chemins sont de longueur au plus $D+1$ dans la famille des graphes de diamètre au plus D, et donc contribuent pour au plus $O(D \log d)$ bits quand le degré maximum du graphe est au plus d. □

2.7 Perspectives

D'une part, nous avons prouvé que l'exploration avec arrêt en utilisant un caillou nécessite $\Omega(\log n)$ bits pour la famille des graphes d'au plus n sommets. D'autre part, nous avons décrit un algorithme d'exploration avec arrêt utilisant un robot de $O(D \log \Delta)$ bits de mémoire pour tous les graphes de diamètre au plus D et de degré maximum au plus d. Déterminer la complexité exacte de l'exploration avec arrêt reste un problème ouvert.

Chapitre 3

Exploration des graphes orientés

Dans ce chapitre, nous étudions la complexité en espace de l'exploration des graphes orientés anonymes. Plus précisément, nous nous intéressons à la quantité de mémoire nécessaire et suffisante que doit avoir un robot, muni d'un ou plusieurs cailloux, pour explorer les graphes orientés d'au plus n sommets et de degré sortant maximum au plus d avec arrêt. Dans cette étude, nous considérons aussi les graphes non simples, c'est-à-dire comportant des boucles ou des arcs multiples. Dans ce cas, il est ainsi possible que d soit plus grand que n.

3.1 Introduction

Explorer des graphes orientés est beaucoup plus difficile qu'explorer des graphes non orientés. En effet, après la première traversée d'arc, le robot est déjà perdu dans le graphe dans le sens où il ne connaît a priori aucun moyen simple de revenir au sommet de départ. Si en plus le graphe orienté est anonyme et régulier (en terme de degré sortant), alors l'exploration des graphes orientés ne peut pas toujours se faire rapidement. En effet, l'exploration des graphes orientés anonymes ne peut se faire en temps polynomial que si le robot dispose de $\Omega(\log \log n)$ cailloux [BFR$^+$02]. Ce nombre de cailloux est en fait optimal : les auteurs décrivent un algorithme de cartographie s'exécutant en temps polynomial, pour un robot muni de $O(\log \log n)$ cailloux. Par contre, si le robot connaît une borne supérieure polynomiale sur le nombre de sommets du graphe, alors il peut effectuer la cartographie du graphe avec un seul caillou.

L'exploration des graphes orientés est coûteuse en temps mais aussi en espace. Cook et Rackoff [CR80] ont montré qu'un JAG a besoin d'une mémoire de $\Omega(\log^2 n / \log \log n)$ bits pour réaliser l'exploration des graphes orientés.

Dans ce chapitre, nous montrons une borne inférieure de $\Omega(n \log d)$ bits de mémoire pour l'exploration par un robot des graphes orientés de n sommets et de degré sortant d (Section 3.2). Cette borne inférieure reste vraie même si le robot dispose d'un nombre linéaire de cailloux. De la même façon que dans le chapitre précédent, notre borne inférieure ne concerne pas seulement les robots mais aussi les automates, c'est-à-dire des machines n'ayant pas forcément des fonctions de transition/sortie calculables par

une machine de Turing.

Nous présentons ensuite deux algorithmes pour l'exploration avec arrêt dans les graphes orientés. Notre premier algorithme (Section 3.3) requiert $O(nd(\log n + \log d))$ bits de mémoire et utilise un caillou. La performance de cet algorithme diffère seulement d'un facteur logarithmique de notre borne inférieure dans les graphes de degré sortant maximum constant. Son temps d'exécution est cependant exponentiel. Nous décrivons un autre algorithme (Section 3.4), qui effectue l'exploration avec arrêt en temps polynomial, mais nécessite $O(n^2 d(\log n + \log d))$ bits de mémoire et utilise $O(\log \log n)$ cailloux. Cet algorithme est une variante de l'algorithme dans [BFR+02]. Cette variante est conçue dans le but de minimiser la quantité de mémoire utilisée, ce qui nécessite de réécrire entièrement la structure de l'algorithme originel présenté dans [BFR+02]. Notons que nos deux algorithmes réalisent en fait la cartographie.

Les résultats de ce chapitre ont fait l'objet d'une publication [FI04], en collaboration avec Pierre Fraigniaud.

3.2 Borne inférieure

Nous prouvons tout d'abord une borne inférieure sur la taille de la mémoire d'un robot explorateur. La preuve utilise le graphe orienté appelé *verrou combinatoire* (voir, par exemple, [Moo56]) défini comme suit.

Définition 26 (Verrou combinatoire)
Le verrou combinatoire $L_{d,n}$ *est un graphe orienté de taille* $n \geq 1$ *dont tous les sommets ont le même degré sortant* d*. Les* n *sommets* $u_0, u_1, \ldots, u_{n-1}$ *sont connectés comme suit. Pour tout* $i < n-1$*, le sommet* u_i *a un arc sortant pointant vers* u_{i+1}*, et* $d-1$ *arcs pointant vers* u_0*. Les* d *arcs sortants du sommet* u_{n-1} *pointent vers* u_0*.*

La figure 3.1 donne un exemple d'un verrou combinatoire.

Théorème 3.1 *L'exploration perpétuelle, et donc l'exploration avec arrêt, dans les graphes orientés à* n *sommets et de degré sortant maximum* $d > 2$ *ne peut pas être accomplie par un robot ayant moins de* $\Omega(n \log d)$ *bits de mémoire, même s'il lui est fourni jusqu'à* n *cailloux. Pour* $d = 2$*, ce résultat reste vrai pour un maximum de* $n/4$ *cailloux.*

Preuve. Fixons d et n et considérons un robot capable d'explorer tous les graphes orientés à n sommets et de degré sortant maximum d. Cela inclut le verrou combinatoire $L_{d,n}$ muni de toutes ses orientations locales possibles. Supposons que le robot dispose de k cailloux, $k \geq 1$. Une *traversée complète* du robot dans $L_{d,n}$ est un parcours du robot le long du chemin $u_0, u_1, \ldots, u_{n-1}$. Pour toute orientation locale, plaçons le robot en u_0 et considérons l'état du robot au départ de sa première traversée complète. Puisque les n sommets u_i, $i = 0, \ldots, n-1$, paraissent identiques au robot, à la présence près d'un caillou, la capacité d'effectuer une traversée complète est déterminée par l'état du robot juste avant de quitter le sommet u_0, et par la position des k cailloux. Il existe d^{n-1} façons

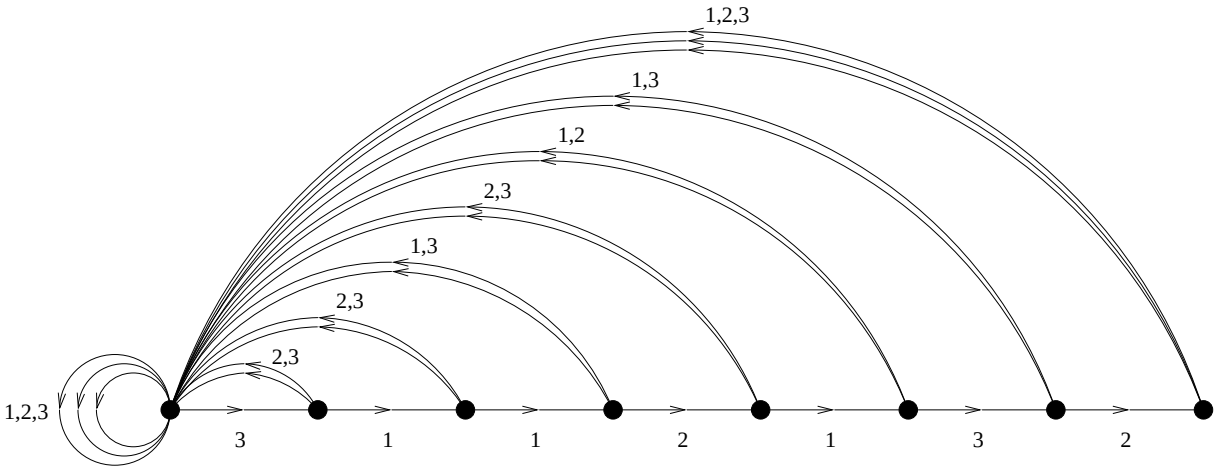

FIG. 3.1 – Un verrou combinatoire $L_{3,8}$ muni d'une orientation locale

d'étiqueter les arcs (u_i, u_{i+1}), $i = 0, \ldots, n-2$, et $p = \sum_{i=0}^{k} \binom{n}{i}$ positions possibles pour les cailloux. Par conséquent, le robot doit pouvoir être dans au moins d^{n-1}/p différents états en u_0. Il doit donc avoir au moins $\lceil (n-1)\log d - \log p \rceil$ bits de mémoire. Puisque $p \leq 2^n$, le théorème est vrai pour $d > 2$. Pour $d = 2$, nous utilisons le lemme 1.6 de [Lei92] affirmant que $\binom{a}{b} \leq (\frac{ae}{b})^b$ pour $0 < b < a$, où $\ln e = 1$. Puisque $k \leq n/2$, nous avons $p \leq k(\frac{ne}{k})^k$, et donc $\log p \leq \log k + k\log(\frac{ne}{k})$. Comme $k \leq n/4$, on a $\log p < \log n + \frac{\log(4e)}{4} n$. D'où $\log p < \log n + \alpha n$ avec $\alpha < 1$, ce qui conclut la preuve. □

Notons que nos algorithmes d'exploration utilisent beaucoup moins que $O(n)$ cailloux. L'un utilise un seul caillou, et l'autre en utilise $O(\log \log n)$.

3.3 Exploration avec mémoire compacte

Dans la suite, nous décrivons un algorithme d'exploration avec arrêt, appelé `Teste-toutes-les-cartes`, dont l'utilisation mémoire correspond à la borne du théorème 3.1 à un facteur logarithmique près pour les graphes orientés de degré sortant constant. L'algorithme possède les propriétés suivantes :

Théorème 3.2 *L'algorithme* `Teste-toutes-les-cartes` *accomplit la cartographie, et donc l'exploration avec arrêt, dans tous les graphes orientés, par un robot utilisant un caillou et dont la mémoire n'excède pas* $O(nd(\log n + \log d))$ *bits dans les graphes orientés à* n *sommets de degré sortant maximum* d.

Remarque 3.1 *Notons que cet algorithme utilise une quantité de mémoire optimale pour la cartographie car* $O(nd(\log n + \log d))$ *bits correspondent à la quantité de mémoire nécessaire pour stocker la carte d'un graphe orienté à* n *sommets de degré sortant maximum* d, *où* d *peut être plus grand que* n.

3.3.1 Algorithme `Teste-toutes-les-cartes`

Le robot considère successivement toutes les valeurs de n, en commençant à $n = 1$. Pour un n fixé, le robot teste toutes les cartes possibles des graphes orientés (munis d'une orientation locale) de taille n. Tester une carte signifie vérifier si la carte est isomorphe au graphe, en conservant l'orientation locale. Le robot s'arrête quand il trouve une carte correspondant au graphe. Notons que le robot explore complètement le graphe quand il teste une carte correcte. En effet, un arc inexploré pourrait mener à un nouveau sommet. Dans notre algorithme, le test d'une carte G est effectué par la procédure `Teste-Carte`, qui sera détaillée dans la prochaine sous-section.

Nous considérons des graphes orientés pouvant avoir des boucles et des arcs multiples. Comme le degré sortant peut être arbitrairement grand (éventuellement plus grand que le nombre de sommets), il existe une infinité de graphes orientés ayant un nombre donné de sommets. Pour résoudre ce problème, le robot mémorise le degré sortant maximum vu jusqu'alors lors de l'exploration. Seules les cartes ayant cette valeur d comme degré sortant maximum sont considérées par le robot. Si, lors de l'exploration, le robot visite un sommet ayant un degré sortant plus grand, alors il essaie toutes les cartes possibles de même nombre de sommets mais ayant ce nouveau degré sortant maximum.

Lemme 3.1 *Supposons que la procédure* `Teste-Carte` *est correcte, c'est-à-dire que la procédure est capable de décider si oui ou non la carte est isomorphe au graphe orienté exploré G. Alors l'algorithme* `Teste-toutes-les-cartes` *finira par tester une carte correcte de G et donc terminera avec succès.*

Preuve. Nous prouvons ce lemme par récurrence sur le nombre de sommets de G. Si G a seulement un sommet, alors le robot connaît trivialement le degré sortant maximum du graphe et donc teste la bonne carte. Supposons que le résultat est vrai pour tous les graphes orientés de taille au plus $n-1$ et supposons que G a n sommets. Si aucune carte de G n'est testée, cela signifie que le degré sortant maximum a été sous-estimé lors de l'étude des cartes de graphes à n sommets (il ne peut pas être sur-estimé). Soit t l'instant à partir duquel le robot commence l'étude des cartes de graphes à $n+1$ sommets. Il existe un sommet u de G de degré sortant d n'ayant jamais été visité lors des tests précédant l'instant t.

Soit u_0 le sommet où démarre l'exploration. Soit G' le graphe induit par les sommets de G explorés avant l'instant t. Nous construisons un graphe orienté G'' à partir de G' en ajoutant des arcs additionnels de la façon suivante. Pour tout arc dans G d'un sommet v de G' vers un sommet w' hors de G', nous plaçons un arc dans G'' partant du même sommet v, ayant le même numéro de port, et pointant vers le sommet de départ u_0. Puisque u n'est pas dans G' et que G est fortement connexe, il existe au moins un arc additionnel dans G''. (Remarquons que G' et G'' ont le même ensemble de sommets.)

Soit v un sommet quelconque de G''. Dans G, il existe un chemin orienté P_v de v vers u_0 car G est fortement connexe. Si le chemin P_v n'est pas inclus dans G', alors soit $e = (w_1, w_2)$ le premier arc du chemin qui n'est pas dans G'. Nécessairement, w_1 est un sommet de G' mais pas w_2. Il existe donc un arc additionnel dans G'' de w_1 vers u_0. Au

final, que P_v soit inclus dans G' ou non, il existe dans G'' un chemin orienté de v vers u_0. D'autre part, dans G', et donc aussi dans G'', il existe un chemin orienté de u_0 vers n'importe quel autre sommet (le chemin suivi par le robot par exemple). Par conséquent G'' est fortement connexe.

La taille n' de G'' est strictement plus petite que n car le sommet u n'est pas dans G''. Par hypothèse de récurrence, si le robot démarre en u_0 dans G'', il finira par tester une carte correcte de G'' et explorera tous les arcs additionnels. Puisque tout sommet de G'' a exactement le même degré sortant dans G'' et dans G, les graphes orientés G'' et G paraissent parfaitement identiques au robot jusqu'à ce qu'un arc absent de G' soit exploré. S'il est dans G'', le robot explore un arc additionnel. Donc de la même façon dans G, le robot visite un arc sortant de G' vers un sommet w. Cela signifie que w est en fait exploré quand la taille $n' < n$ est considéré, contredisant la définition de G'. Cela conclut la preuve du lemme. □

3.3.2 Procédure `Teste-Carte`

Dans la suite, nous utiliserons quelques notations de [BFR+02]. Etant donné deux graphes orientés G et X, et deux sommets u et x de G et X respectivement, nous notons $(G, u) \cong (X, x)$ s'il existe un isomorphisme f entre G et X, préservant les numéros de port, et tel que $f(u) = x$. Nous disons que (G, u) est *cohérent* avec (X, x) s'il existe un sous-graphe X' de X tel que $(G, u) \cong (X', x)$.

Pour une carte donnée $G = (V, E)$, avec $V = \{v_1, \ldots, v_n\}$, le robot procède comme suit. Soit x la position courante du robot dans le graphe orienté inconnu X. Le robot choisit le sommet $v_1 \in V$ et teste s'il est sur le sommet v_1 de G, c'est-à-dire si $(G, v_1) \cong (X, x)$. Si le test réussit, alors l'exploration s'arrête. Autrement, le robot choisit le sommet $v_2 \in V$, et teste $(G, v_2) \cong (X, x)$. Ce processus est poursuivi jusqu'à ce qu'un test réussisse ou que tous les sommets de G aient été considérés.

Remarquons que lors d'un test pour décider si, pour un certain i, on a $(G, v_i) \cong (X, x)$ ou non, le robot se déplace dans le graphe X. Donc, puisque la procédure peut échouer pour v_i, il n'existe aucune garantie que le robot est maintenant de nouveau sur le sommet x de X quand il démarre le test $(G, v_{i+1}) \cong (X, x)$. Par conséquent, le robot utilise un tableau `position`, de taille n, tel que `position`$[i]$ est l'indice j du sommet $v_j \in V$ où le robot se trouverait maintenant *si* la position originelle x du robot vérifiait $x = v_i$.

Ce que le robot fait pour tester $(G, v_i) \cong (X, x)$ dépend de s'il est en possession du caillou ou non.

- Si le robot n'a pas le caillou, il exécute la procédure `Trouve-Caillou`(G, v_j) où $j =$ `position`$[i]$. Si la procédure échoue, cela signifie que $(G, v_i) \ncong (X, x)$. Si la procédure réussit, le robot utilise le caillou tout juste retrouvé pour exécuter la procédure `Contrôle-Isomorphie` avec l'entrée (G, v_k) où $k =$ `position`$[i]$.
- Si le robot a le caillou, il exécute la procédure `Contrôle-Isomorphie` avec l'entrée (G, v_k) où $k =$ `position`$[i]$.

Dans les deux cas, on a $(G, v_i) \cong (X, x)$ si et seulement si la procédure `Contrôle-Isomorphie` renvoie un succès.

Nous décrivons maintenant la procédure `Trouve-Caillou`. La procédure `Contrôle-Isomorphie` est décrite dans la prochaine sous-section.

Procédure `Trouve-Caillou`(G, u) **:**

Le robot calcule un chemin orienté P (non nécessairement simple) dans la carte G, commençant en u, et visitant tous les sommets $\{v_1, \ldots, v_n\}$ de G. P est calculé à la volée. Par exemple, P est constitué d'un chemin P_0 de u à v_1, suivi d'une suite de chemins P_i de v_i à v_{i+1}, $i = 1, \ldots, n-1$. Les chemins P_i sont calculés les uns après les autres. Si le robot ne trouve pas le caillou, alors forcément $(G, v_i) \ncong (X, x)$ et la procédure retourne `échec`. Autrement, la procédure retourne `succès`.

3.3.3 Procédure `Contrôle-Isomorphie`

Avant de décrire la procédure `Contrôle-Isomorphie`, nous présentons une autre procédure, appelée `Contrôle-Cohérence`, vérifiant si une carte est cohérente avec le graphe exploré. Cette procédure utilise un caillou pour s'exécuter.

Procédure `Contrôle-Cohérence`(G, u) **:**

Etant donné un graphe orienté $G = (V, E)$ et un sommet u de G, la procédure `Contrôle-Cohérence` vérifie si (G, u) est cohérent avec (X, x), où x est la position courante du robot dans le graphe orienté inconnu X. La procédure emprunte à [BFR+02] la technique de marquage des sommets d'un cycle. Cependant cette technique est implémentée sans l'utilisation d'une grande structure de donnée. Plus précisément, le robot assigne des numéros, de 1 à m, à tous les $m \leq nd$ arcs de la carte G, avec la condition supplémentaire que l'arc 1 sorte de u, et que l'arc m entre dans u (de tels arcs existent car G est fortement connexe). On a donc $E = \{e_1, \ldots, e_m\}$. Pour tout $i \in \{1, \ldots, m-1\}$, le robot calcule un plus court chemin P_i dans la carte G partant de la destination de l'arc e_i et arrivant à l'origine de l'arc e_{i+1}. Lors de la procédure `Contrôle-Cohérence`, les chemins P_i sont calculés à la volée, et au plus un chemin est conservé en mémoire par le robot à un instant donné. Soit C le chemin fermé suivant commençant et finissant en u : $C = e_1, P_1, \ldots, e_{m-1}, P_{m-1}, e_m$. Ce chemin sera traversé plusieurs fois durant l'exécution de la procédure `Contrôle-Cohérence`. C est donc recalculé plusieurs fois par le robot. Quand le chemin P_i est calculé, le robot oublie le chemin P_{i-1}.

La procédure `Contrôle-Cohérence` est constituée d'au plus n phases, une pour chaque sommet de G. La procédure présume que le robot détient le caillou. (C'est toujours le cas dans notre algorithme `Teste-toutes-les-cartes`.) Pour chaque phase, il y a un nouveau sommet *considéré*. Lors de la phase $i \geq 1$, le robot quitte u avec le caillou et suit les arcs de C jusqu'à ce qu'il visite un sommet v de G qui n'a pas encore été *considéré* lors des $i-1$ premières phases. Ce sommet est marqué *considéré* sur la carte G, et le caillou y est déposé. Le premier sommet considéré est le sommet u à la phase 1. Le robot continue son trajet guidé par C jusqu'à ce qu'il soit à la fin de C, de retour en u d'après la carte. Alors le robot traverse C de nouveau. Durant ce trajet, il contrôle la propriété suivante :

Propriété $\mathcal{P}$: le caillou est sur le sommet courant x si et seulement si x est le sommet considéré v, d'après la carte G.

Si la propriété $\mathcal{P}$ n'est pas vérifiée pour un certain sommet de C, le robot termine son trajet le long de C et la procédure retourne `échec`. Cette situation est possible dans deux cas : le caillou n'est pas là où il devrait être, ou le caillou est là où il ne devrait pas être. Si la propriété $\mathcal{P}$ est vérifiée pour tout sommet de C, alors le robot suit C une nouvelle fois pour ramener le caillou en u. S'il reste encore un autre sommet à considérer, la phase suivante procède avec ce sommet. Autrement, c'est-à-dire si tous les sommets de G ont été considérés et qu'aucun problème n'est survenu, la procédure retourne `succès`.

Lemme 3.2 *Etant donné un robot sur un sommet x d'un graphe orienté inconnu X, la procédure* `Contrôle-Cohérence` *retourne* `succès` *pour (G, u) si et seulement si (G, u) est cohérent avec (X, x).*

Preuve. Si (G, u) est cohérent avec (X, x), c'est-à-dire si $(G, u) \cong (X', x)$ pour un sous-graphe X' de X, la procédure `Contrôle-Cohérence` retourne `succès` pour (G, u) car le parcours dans G et dans X' sera exactement le même, à un isomorphisme près.

Nous prouvons que, si la procédure `Contrôle-Cohérence` retourne `succès` pour (G, u), alors (G, u) est cohérent avec (X, x). Donc supposons que la procédure `Contrôle-Cohérence` retourne `succès` pour (G, u). Nous notons d'abord que le cycle C dans G définit un cycle dans X car, après son premier parcours le long de C, le robot trouve le caillou, qui avait été déposé au début de C, au sommet x. Nous pouvons donc définir $x(t)$ et $u(t)$ comme les sommets dans X et dans G, respectivement, visités après t étapes le long du chemin fermé C. Soient t et t' deux entiers compris entre 0 et $|C|$, inclus. Considérons la phase où le sommet $u(t)$ est considéré. Le robot trouve le caillou en $x(t)$ car la procédure `Contrôle-Cohérence` retourne `succès`. De plus, $u(t') = u(t)$ si et seulement si le robot trouve le caillou en $u(t')$, et il l'y trouve si et seulement si $x(t') = x(t)$. Par conséquent,

$$\forall t, t' \in \{0, \ldots, |C|\}, x(t) = x(t') \Leftrightarrow u(t) = u(t').$$

Nous pouvons maintenant définir la fonction f, qui fait correspondre à tout sommet $v = u(t)$ dans G le sommet $f(v) = x(t)$ dans X. Soient $v = u(t)$ et $v' = u(t')$ deux sommets de G. On a

$$f(v) = f(v') \Leftrightarrow x(t) = x(t') \Leftrightarrow u(t) = u(t') \Leftrightarrow v = v'.$$

Donc f est injective. Nous prouvons maintenant que si $e = (v, v')$ est un arc de G de numéro de port p, alors $f(e) = (f(v), f(v'))$ est un arc dans X, également étiqueté p. Il existe t tel que $v = u(t)$ et $v' = u(t + 1)$ car C contient tous les arcs de G. Nous avons $x(t) = f(v)$ et $x(t + 1) = f(v')$. Donc il existe un arc $(f(v), f(v')) = (x(t), x(t + 1))$ avec la même étiquette que $(u(t), u(t + 1)) = e$. D'où (G, u) est cohérent avec (X, x), ce qui conclut la preuve. □

Nous pouvons maintenant décrire la procédure `Contrôle-Isomorphie`.

Procédure `Contrôle-Isomorphie`(G, u) **:**

Etant donné la carte d'un graphe orienté $G = (V, E)$, avec n sommets et de degré sortant maximum d, et étant donné un sommet u de G, la procédure `Contrôle-Isomorphie` vérifie si le robot est à cet instant sur le sommet u de G, c'est-à-dire si $(G, u) \cong (X, x)$, où x est la position courante du robot dans le graphe orienté inconnu X.

Le robot exécute d'abord la procédure `Contrôle-Cohérence`(G, u). Si elle retourne `échec`, alors la procédure `Contrôle-Isomorphie` retourne également `échec`. Autrement, soit C le chemin fermé utilisé dans la procédure `Contrôle-Cohérence`. Le robot suit une dernière fois le chemin C pour vérifier si il y a bien égalité entre le degré de chaque sommet dans la carte G, et le degré du sommet correspondant dans le graphe exploré X. Si c'est le cas, le robot renvoie `succès`. Sinon, il renvoie `échec`.

Lemme 3.3 *Etant donné un robot sur un sommet x d'un graphe orienté X, la procédure* `Contrôle-Isomorphie` *retourne* `succès` *pour (G, u) si et seulement si $(G, u) \cong (X, x)$.*

Preuve. Si $(G, u) \cong (X, x)$, la procédure `Contrôle-Cohérence` puis la procédure `Contrôle-Isomorphie` retournent `succès` pour (G, u) puisque le trajet dans G et dans X est exactement le même, à un isomorphisme près.

Nous prouvons que, si la procédure `Contrôle-Isomorphie` retourne `succès` pour (G, u), alors on a $(G, u) \cong (X, x)$. D'après le lemme 3.2, (G, u) est cohérent avec (X, x), c'est-à-dire il existe un sous-graphe X' de X tel que $(G, u) \cong (X', x)$.

Le dernier parcours dans la procédure `Contrôle-Isomorphie` assure que pour tout sommet v dans G, les degrés sortants de v et de $f(v)$ sont les mêmes. Par conséquent, X a le même nombre de sommets et d'arcs que G et donc que X'. D'où $(G, u) \cong (X, x)$, ce qui conclut la preuve. □

3.3.4 Preuve du théorème 3.2.

Nous prouvons d'abord que l'algorithme `Teste-toutes-les-cartes` est correct. Pour un graphe orienté G donné, la procédure `Teste-Carte` teste $(G, v) \cong (X, x)$ pour tous les $v \in V(G)$ grâce au tableau `position`. D'après le lemme 3.3 et la justesse de la procédure `Trouve-Caillou`, chaque test $(G, v) \cong (X, x)$ est exécuté de façon correcte et donc la procédure `Teste-Carte` est correcte. D'après le lemme 3.1, il s'ensuit que l'algorithme `Teste-toutes-les-cartes` est correct.

Nous prouvons que l'algorithme `Teste-toutes-les-cartes` peut être implémenté de telle manière qu'il n'utilise pas plus de $O(nd \log n)$ bits de mémoire dans les graphes orientés à n sommets de degré sortant maximum d. Il est facile de lister tous les graphes orientés munis d'une orientation locale et ayant n sommets et un degré sortant maximum d en utilisant une zone de mémoire de $O(nd(\log n + \log d))$ bits. Puisque le cycle $C = e_1, P_1, e_2, P_2, \ldots, e_{m-1}, P_{m-1}, e_m$ visitant tous les arcs d'une carte donnée est calculé à la volée, et puisque tout chemin P_i peut être encodé par une suite d'au plus n numéros de port, nous obtenons que la procédure `Contrôle-Cohérence` nécessite au plus $O(n \log d)$ bits de mémoire pour le stockage de C. Les mêmes arguments s'appliquent dans

la procédure `Trouve-Caillou` concernant le stockage de P. Le robot n'utilise donc pas plus de $O(nd(\log n + \log d))$ bits de mémoire au total.

Ceci conclut la preuve du théorème 3.2.

3.4 Exploration en temps polynomial

L'algorithme présenté dans la section précédente est compact mais effectue l'exploration avec arrêt en un temps exponentiel dans le pire des cas (rappelons que le temps est compté comme le nombre de traversées d'arcs). Dans cette section, nous décrivons une variante de l'algorithme `Explore-and-Map` presenté dans [BFR+02]. Bien que polynomial en temps, `Explore-and-Map` est coûteux en espace : une analyse rapide montre qu'il requiert une mémoire de $O(n^5 d \log n)$ bits. Notre variante, appelée `Compacted-Explore-and-Map`, offre de bien meilleures performances :

Théorème 3.3 *Pour tout graphe orienté à n sommets de degré sortant maximum d, l'algorithme* `Compacted-Explore-and-Map` *accomplit la cartographie en temps polynomial, avec un robot utilisant $O(\log \log n)$ cailloux, et une mémoire de taille $O(n^2 d(\log n + \log d))$ bits.*

Remarque 3.2 *L'algorithme* `Compacted-Explore-and-Map`*, de même que l'algorithme* `Explore-and-Map`*, utilise $O(\log \log n)$ cailloux. Bender et al. [BFR+02] ont montré que $\Omega(\log \log n)$ cailloux étaient nécessaires pour l'exploration en temps polynomial.*

Remarque 3.3 *La pierre d'angle de l'algorithme* `Explore-and-Map` *est la construction de n cartes indépendantes, une pour chaque sommet. Par conséquent, toute variante de* `Explore-and-Map` *nécessite un robot avec $\Omega(n^2 d(\log n + \log d))$ bits de mémoire.*

Nous commençons la description de notre algorithme en donnant quelques définitions et en décrivant plusieurs procédures. Ces procédures utilisent une borne supérieure $\hat{n}$ sur le nombre de sommets du graphe exploré et utilisent un caillou. Tout ceci conduit à la description d'un algorithme, appelé `CEM-Borne-Sup` (pour `Compacted-Explore-and-Map -Borne-Sup`), qui utilise un caillou et prend $\hat{n}$ en entrée. Nous expliquerons à la fin de la section comment utiliser cet algorithme quand aucune borne supérieure n'est connue. La connaissance d'une borne supérieure est utilisée en conjonction avec l'observation suivante :

Fait 3.1 *Soit $f : V \to V$ une fonction quelconque déterministe. Pour tout $v \in V$, la suite $v, f(v), f^2(v), \ldots$ devient périodique après au plus $|V|$ répétitions.*

Le robot utilise le caillou pour s'informer sur son environnement. Si le caillou est déposé sans précaution particulière sur un sommet, le robot peut ne pas être capable de le retrouver, du fait que le graphe est dirigé. Par conséquent, avant d'exécuter une procédure, le robot entre dans un cycle C en répétant $\hat{n} \geq n$ fois le chemin (non nécessairement

élémentaire) utilisé par la procédure. Donc, si le robot perd le caillou, il peut continuer son parcours le long du cycle C pour le retrouver.

Nous donnons maintenant quelques idées permettant de facilité la compréhension par le lecteur des procédures décrites dans les sous-sections suivantes.

Une carte partielle, qui est une vue depuis un certain sommet u du graphe exploré G, est une carte d'un sous-graphe de G contenant u. L'algorithme maintient en mémoire plusieurs cartes partielles, qui sont autant de vues depuis différents sommets de G. Le but de l'algorithme est d'améliorer ces cartes partielles jusqu'à ce qu'une carte complète de G soit obtenue. Pour améliorer une carte partielle M, le robot utilise la procédure `MàJ-Carte`, pour `Mise-à-Jour-Carte`. Cette procédure nécessite un chemin qui parcourt toutes les arcs référencés dans M plus un nouvel arc.

Cependant, le robot doit d'abord connaître approximativement sa position dans le graphe pour savoir quelle carte partielle il doit mettre à jour. La procédure `Vérifie`(M) permet au robot de vérifier si une carte partielle M est une vue depuis le sommet courant. Cependant, ce test peut échouer pour toutes les cartes partielles stockées. Pour pallier à ce problème, le robot ne teste pas ces cartes partielles dans un ordre arbitraire. Les cartes partielles sont stockées dans une structure de données spéciale, appelée arbre étiqueté, et le robot utilise cette structure de données pour décider dans quel ordre vérifier les cartes partielles. Cette suite d'appels à la procédure `Vérifie` est implémentée par la procédure `Classifie`.

Après avoir exécuté la procédure `Classifie`, le robot a déterminé quelle carte partielle M correspond au sommet courant u de G. La procédure `Nouvelle-Arête`(M) explore alors tous les arcs référencés dans M plus un arc non cartographié. Malheureusement, puisque le robot ne sait pas où mène ce dernier arc, il peut être perdu dans le graphe après avoir traversé cet arc. Le robot applique donc la procédure `Classifie` une nouvelle fois. Après cette exécution, le robot a de nouveau déterminé quelle carte partielle M' correspond au sommet courant u' de G, et il applique la procédure `Nouvelle-Arête`(M'). Cependant, il peut n'y avoir aucun lien entre u et u'. Pour résoudre ce problème, le robot exécute $\hat{n}$ fois les procédures `Classifie` et `Nouvelle-Arête` pour entrer dans un cycle. Ce cycle est ensuite traité par la procédure `MàJ-Carte` pour améliorer une des cartes partielles impliquées dans le cycle.

3.4.1 Cartes partielles

Les algorithmes `CEM-Borne-Sup` et `Compacted-Explore-and-Map` utilisent des cartes partielles pour stocker les informations récoltées à propos du graphe orienté exploré. Dans [BFR+02], l'algorithme `Explore-and-Map` utilise simplement des graphes orientés. Dans notre cas, nous avons besoin d'avoir un ordre sur les arcs. Premièrement, cela nous permet de spécifier les différentes procédures plus précisément que dans [BFR+02]. Deuxièmement, le stockage de toutes les cartes partielles sera réduit d'un facteur n grâce à cet ordre.

Une *carte partielle* M est un couple (N, L), où N est un entier positif et L est une liste d'éléments distincts dans $\{1, \ldots, N\}^2 \times \mathbb{N}$. Une carte partielle définit un graphe orienté

$g(M)$ de la façon suivante. L'ensemble des sommets de $g(M)$ est $\{1, \ldots, N\}$. Un élément $((p, q), \ell)$ de L décrit un arc (p, q) de numéro de port ℓ. Toute carte partielle considérée dans ce chapitre induira un graphe orienté $g(M)$ fortement connexe. Le sommet 1 de $g(M)$ est noté $v(M)$.

Soit u un sommet quelconque de G. Nous disons que M est une *vue depuis* u (dans G) si $(g(M), v(M))$ est cohérent avec (G, u). Intuitivement, cela signifie que M est une carte d'un sous-graphe G' de G contenant u. Un arc dans G' est dit *référencé* dans M.

Une carte partielle $M = (N, L)$ est un *préfixe* d'une autre carte partielle $M' = (N', L')$ si $N \leq N'$ et la liste L est un préfixe de la liste L'. Cela implique que $g(M)$ est un sous-graphe de $g(M')$.

Nous définissons la *carte partielle minimale* comme l'unique carte partielle avec un sommet et aucun arc, c'est-à-dire $(1, \epsilon)$, où ϵ est la liste vide. Clairement la carte partielle minimale est un préfixe de toute carte partielle.

Nous définissons maintenant le *chemin fermé* $cp(M)$ et la suite de numéros de port $ts(M)$ de M. Si $g(M)$ n'a aucun arc, alors $cp(M)$ et $ts(M)$ sont vides. Autrement, soit $L = ((e_1, p_1), \ldots, (e_q, p_q))$ la liste ordonnée des arcs de M. Rappelons que $g(M)$ est fortement connexe. Pour tout $i \in \{1, \ldots, q-1\}$, le robot calcule un plus court chemin P_i dans le graphe $g(M)$ partant de la destination de l'arc e_i vers l'origine de l'arc e_{i+1}. Soit P_0 un plus court chemin dans $g(M)$ partant de $v(M)$ et arrivant à l'origine de l'arc e_1. Soit P_q un plus court chemin dans $g(M)$ partant de la destination de l'arc e_q et arrivant au sommet $v(M)$. Alors $cp(M)$ est le chemin fermé suivant, commençant et finissant en $v(M)$:

$$cp(M) = P_0 e_1 P_1 \ldots e_{q-1} P_{q-1} e_q P_q.$$

Nous définissons $ts(M)$ comme la suite des numéros de port de $cp(M)$.

Finalement, étant donné une carte partielle M, un chemin fermé P de G est *compatible avec* M si

- P commence et termine en un sommet u tel que M est une vue depuis u ;
- P visite tous les sommets et tous les arcs référencés dans M avant d'éventuellement visiter un arc non référencé dans M.

Un chemin compatible avec M sera utilisé dans la procédure `MàJ-Carte` pour améliorer la carte partielle M.

Lemme 3.4 *Soit $M = (N, L)$ une carte partielle. Supposons que M est une vue depuis un certain sommet u de G. On a :*

1. *M peut être mémorisée en utilisant $O(nd(\log n + \log d))$ bits.*
2. *La longueur de $cp(M)$ (et donc celle de $ts(M)$) est au plus $O(n^2 d)$;*
3. *Le robot peut suivre $ts(M)$ en utilisant seulement $O(n \log d)$ bits de mémoire (sans prendre en compte la taille de la carte partielle elle-même).*

Preuve.

1. Puisque M est une vue depuis un sommet de G, $g(M)$ est un sous-graphe de G. Donc $g(M)$ a au plus n sommets, au plus nd arcs, et les numéros de ports sont

inférieurs à d. Cela implique que $N \leq n$ et $|L| \leq nd$. Par conséquent, une carte partielle peut être encodée en utilisant $O(\log n) + nd \cdot O(\log n + \log d)$ bits, i.e., $O(nd(\log n + \log d))$ bits.

2. La longueur de chaque plus court chemin P_i dans $cp(M)$ peut être bornée par n. Comme au plus $nd + 1$ tels chemins sont combinés dans $cp(M)$, on a $|cp(M)| = |ts(M)| \leq O(n^2 d)$.
3. Lorsque le robot emprunte la suite de numéros de port $ts(M)$, les chemins P_i sont calculés à la volée, et au plus un chemin est stocké à la fois dans la mémoire du robot. Donc $ts(M)$ peut être calculé à la volée en utilisant $O(\log nd)$ bits pour stocker le numéro i du chemin courant P_i, et $O(n \log d)$ bits pour stocker le chemin lui-même (car la longueur d'un chemin P_i est plus petite que n).

□

3.4.2 Vérification et mise à jour des cartes partielles

Procédure `Vérifie`(M).

Nous décrivons la procédure `Vérifie`M qui contrôle si une carte partielle M est une vue depuis un sommet v dans G, et retourne soit 0 (échec) soit 1 (succès). La procédure présume que le robot détient le caillou. Elle consiste en deux parties : `Init-Vérifie` et `Effectif-Vérifie`. La première partie est utilisée par le robot pour entrer dans un cycle, dans le but de ne pas perdre le caillou. La seconde partie est principalement un appel à la procédure `Contrôle-Cohérence`, défini dans la section 3.3. Les déplacements additionnels sont effectués par le robot pour assurer que celui-ci termine la deuxième partie à l'endroit où il l'a commencée, et en possession du caillou. En effet, ce n'est pas nécessairement le cas lorsque seule la procédure `Contrôle-Cohérence` est appelée.

- `Init-Vérifie`(M) consiste à suivre $\hat{n}$ fois la suite de numéros de port $ts(M)$.
- `Effectif-Vérifie`(M) consiste d'abord à déposer le caillou sur le sommet courant v, et à parcourir une fois $ts(M)$. Soit w le sommet courant après ce parcours. Si le caillou n'est pas en w, alors le robot parcourt plusieurs fois $ts(M)$ jusqu'à ce qu'il retrouve le caillou. Le robot s'arrête et la procédure `Vérifie` retourne 0. Si le caillou est en w, cela signifie que $w = v$. Le robot exécute alors la procédure `Contrôle-Cohérence`$(g(M), v(M))$, en utilisant $cp(M)$ comme chemin fermé C (cf. la description de la procédure `Contrôle-Cohérence` dans la sous-section 3.3.3). Si `Contrôle-Cohérence` retourne `succès`, alors la procédure `Vérifie` renvoie 1. Si `Contrôle-Cohérence` retourne `échec`, le robot peut avoir perdu le caillou. Dans ce cas, le robot parcourt une nouvelle fois $ts(M)$ pour récupérer le caillou. La procédure `Vérifie` renvoie alors 0.

Remarquons que la procédure `Vérifie`(M) ne fait rien et renvoie toujours 1 quand M est la carte partielle minimale. Dans ce cas, $ts(M)$ est en effet réduit à la suite vide.

Lemme 3.5 *Soit M une carte partielle. Soit u un sommet quelconque du graphe. Soit v le sommet sur lequel le robot se trouve après avoir terminé l'exécution de* `Init-Vérifie`(M) *depuis le sommet u. On a :*

1. *La sous-procédure* `Init-Vérifie`(M) *n'a pas besoin du caillou.*
2. *La sous-procédure* `Effectif-Vérifie`(M) *visite uniquement des sommets et des arcs déjà visités par la sous-procédure* `Init-Vérifie`(M).
3. *La sous-procédure* `Effectif-Vérifie`(M) *commence et termine en v avec le caillou.*
4. *La procédure* `Vérifie`(M) *renvoie 1 si et seulement si M est une vue depuis v dans G.*
5. *La procédure* `Vérifie`(M) *s'exécute en temps $O(\hat{n}n^2d)$ et utilise $O(\log \hat{n} + n \log d)$ bits de mémoire.*

Preuve.

1. Immédiat d'après la description de `Init-Vérifie`.
2. D'après le fait 3.1 et comme $\hat{n} \geq n$, nous savons que le robot entre dans un cycle composé d'un ou plusieurs $ts(M)$ durant la sous-procédure `Init-Vérifie`. Par conséquent, parcourir une ou plusieurs fois $ts(M)$, comme c'est le cas dans la sous-procédure `Effectif-Vérifie`(M), maintient le robot dans C, c'est-à-dire dans une partie de G déjà visitée pendant `Init-Vérifie`(M).
3. Au début de `Effectif-Vérifie`(M), le robot dépose le caillou et parcourt une fois $ts(M)$. S'il ne trouve pas le caillou en w, alors parcourir de nouveau $ts(M)$ finira par le ramener en v et au caillou, comme nous l'avons noté dans la preuve du point précédent. Si le robot trouve le caillou en $w = v$, alors $ts(M)$ décrit un chemin fermé dans G commençant et terminant en v. Puisque le robot effectue uniquement des traversées complètes de $ts(M)$ pendant et après l'exécution de la procédure `Contrôle-Cohérence`, le robot s'arrête en v. De plus, il détient le caillou.
4. Si M est une vue depuis v, alors le robot trouve le caillou après son premier parcours le long de $ts(M)$. Donc la procédure `Vérifie`(M) renvoie 1 si et seulement si M est une vue depuis v dans G car la procédure `Contrôle-Cohérence` retourne `succès` si et seulement si $(g(M), v(M))$ est cohérent avec (G, v).
5. La sous-procédure `Effectif-Vérifie`(M) parcourt $ts(M)$ au plus $O(n)$ fois (cf. la description de la procédure `Contrôle-Cohérence` dans la sous-section 3.3.3). Donc la procédure `Vérifie` parcourt $ts(M)$ au plus $\hat{n} + O(n)$ fois. D'après le lemme 3.4, la longueur de $ts(M)$ est bornée par $O(n^2d)$. Donc la procédure `Vérifie` s'exécute en temps $O(\hat{n}n^2d)$.

   D'après le point 3 du lemme 3.4, suivre $ts(M)$ requiert au plus $O(n \log d)$ bits. La sous-procédure `Init-Vérifie` utilise un compteur jusqu'à $\hat{n}$ et la quantité de mémoire nécessaire pour suivre $cp(M)$, donc au plus $O(\log \hat{n} + n \log d)$ bits. La procédure `Contrôle-Cohérence`, et donc la sous-procédure `Effectif-Vérifie`, utilise $O(\log n)$ bits pour se souvenir du sommet considéré, plus $O(n \log d)$ bits pour suivre $cp(M)$. En résumé, la procédure `Vérifie` utilise $O(\log \hat{n} + n \log d)$ bits.

La procédure `Vérifie` permet au robot de contrôler si une carte partielle est une vue depuis un sommet. Nous présentons maintenant une procédure, appelée `MàJ-Carte`, permettant de mettre à jour une carte partielle, avec l'aide d'un caillou. En effet, cette procédure suppose que le robot détient le caillou quand elle est appelée.

Procédure `MàJ-Carte`(M, P).

Soit $M = (N, L)$ une carte partielle. Soit P un chemin fermé partant d'un certain sommet u, et compatible avec M. Supposons que le robot est capable de visiter P et de détecter la fin de P sans l'aide d'un caillou. Notons que cela ne signifie pas que le robot détecte systématiquement quand il est en u, mais seulement qu'il est capable de détecter quand le chemin P est complètement exploré. De plus, le robot ne sait pas a priori où la partie de P référencée dans M s'arrête dans P. Partant de u avec le caillou, le robot exécute `MàJ-Carte`(M, P) pour créer une carte partielle M' telle que , intuitivement, M' est la carte du sous-graphe de G induit par P.

La procédure `MàJ-Carte` utilise la même technique que la procédure `Contrôle-Cohérence`. Précisément, la procédure `MàJ-Carte`(M, P) procède comme suit. Le robot parcourt P et simule ses déplacements sur $g(M)$. Si le robot atteint la fin de P sans visiter un arc qui n'est pas dans $g(M)$, alors la procédure retourne $M' = M$. Sinon, soit $e = (v, v')$ le premier arc de P qui n'est pas référencé dans M. Soit ℓ le numéro de port de l'arc e. Le sommet v de G correspond au sommet p de $g(M)$. Le robot dépose le caillou en v' et revient en u en finissant l'exploration de P. Le robot parcourt ensuite de nouveau P tout en suivant ses déplacements sur la carte $g(M)$. Si le robot voit le caillou quand il visite un sommet q de $g(M)$, il met à jour M en ajoutant $((p, N+1), \ell)$ à la fin de L. Si le robot voit le caillou pour la première fois en un sommet n'ayant pas de correspondance dans $g(M)$, il met à jour M en ajoutant $((p, N+1), \ell)$ à la fin de L, et incrémente N de 1. Dans les deux cas, le robot reprend le caillou et termine l'exploration de P. Le robot continue de cette façon avec les arcs non identifiés successifs jusqu'à ce qu'aucun arc de P ne reste non identifié. La procédure `MàJ-Carte`(M, P) retourne la carte partielle obtenue M'.

Lemme 3.6 *Soit M une carte partielle et soit P un chemin fermé partant d'un sommet u, et compatible avec M. Supposons que le robot, en possession du caillou, exécute la procédure* `MàJ-Carte`(M, P) *depuis u. Soit M' le résultat obtenu. On a :*

1. *Le robot termine la procédure en u avec le caillou.*
2. *M' est une carte partielle bien définie. M est un préfixe de M'. Soit H le sous-graphe de G induit par P. On a $(g(M'), v(M')) \cong (H, u)$.*
3. *La procédure* `MàJ-Carte`(M, P) *s'exécute en temps $O(|P|nd)$ et utilise $O(nd(\log n + \log d))$ bits de mémoire plus la mémoire nécessaire pour suivre P.*

Preuve.

1. Puisque P est un chemin fermé commençant et terminant en u et puisque le robot effectue uniquement des traversées complètes de P, alors le robot termine la

procédure en u. De plus, d'après la description de la procédure, le robot est en possession du caillou.

2. Immédiat, par construction de M'.
3. En deux traversées de P, la procédure `MàJ-Carte` identifie un nouvel arc de G. Puisque G a au plus nd arcs, la procédure traverse P au plus $2nd$ fois. Elle utilise uniquement la mémoire nécessaire pour suivre P et la mémoire pour stocker la carte partielle M'.

□

Dans la procédure `MàJ-Carte`, le chemin P est exploré un nombre constant de fois pour chaque arc. Dans [BFR+02], la procédure correspondante, appelée `Compress`, explore le chemin un nombre constant de fois pour chaque nouveau sommet, donc moins souvent. Le problème est que la technique plus rapide de [BFR+02] nécessite de maintenir en mémoire un tableau dont la taille est la longueur du chemin. Comme nous le verrons plus tard, le chemin utilisé dans la procédure `MàJ-Carte` est très long (environ $\hat{n}n^5d^2$). Nous avons donc dû procéder différemment.

Dans le but d'obtenir une carte partielle M' strictement plus précise que M, au moins un arc non référencé dans M doit être présent dans P. Ceci est assuré grâce à la procédure suivante.

Procédure `Nouvelle-Arête`(M)**.**

Etant donné une carte partielle M qui ne représente pas le graphe en entier, le but de la procédure `Nouvelle-Arête`(M) est de visiter tous les sommets et tous les arcs référencés dans M, et de trouver un arc non référencé dans M (mais partant d'un sommet de M). Plus précisément, le robot commence au sommet v avec une carte partielle M qui est une vue depuis v. Le robot parcourt $cp(M)$ une fois complètement, et ensuite encore une fois jusqu'à trouver un sommet ayant un degré sortant différent dans G et dans M (un tel sommet existe car M n'est pas une carte exhaustive de G). Le robot prend finalement l'un des arcs sortants non explorés du sommet courant.

3.4.3 Stockage des cartes partielles

Pendant qu'il exécute l'algorithme, le robot utilise un arbre étiqueté comme structure de données pour stocker une partie des informations acquises jusque-là sur le graphe exploré. Toutes les cartes partielles calculées sont stockées dans des sommets de cet arbre. L'arbre est utilisé par le robot dans la procédure `Classifie` en vue d'apprendre sa position approximative. L'arbre étiqueté et la procédure `Classifie` sont similaires à l'arbre et à la procédure `cop` utilisés dans [BFR+02], excepté en ce qui concerne la mise à jour de l'arbre.

Le robot utilise un arbre binaire enraciné étiqueté T de hauteur au moins 1, pour lequel chaque sommet interne (donc incluant la racine) a exactement deux fils. L'étiquette d'un sommet v de T est notée $l(v)$. La racine est étiquetée par le mot vide. Soit v un sommet

interne de T. L'un de ses deux fils a pour étiquette $l(v)\circ 0$ et l'autre $l(v)\circ 1$, où $\circ$ représente la concaténation de chaînes de caractères. Par conséquent, l'étiquette de tout sommet est un mot fini sur l'alphabet $\{0,1\}$. Deux sommets différents ont des étiquettes différentes, et un sommet v est un ancêtre d'un sommet v' si et seulement si $l(v)$ est un préfixe de $l(v')$.

Toutes les cartes partielles calculées jusqu'à l'instant courant sont stockées dans l'arbre. Plus précisément, une carte partielle $M(v)$ est stockée dans chaque sommet interne v de l'arbre. Nous verrons plus tard que certaines cartes partielles sont des préfixes d'autres et que ceci nous permettra d'encoder l'arbre efficacement.

Nous nous concentrons maintenant sur la procédure `Classifie`. Intuitivement, cette procédure permet au robot d'apprendre approximativement sa position. Après avoir exécuté la procédure, le robot détermine une carte partielle qui est une vue depuis le sommet où il se trouve.

Procédure `Classifie`().

La procédure se déroule en phases, chaque phase correspondant à un sommet dans l'arbre T. La procédure part de la racine et termine lorsqu'elle atteint une feuille. Nous décrivons la phase correspondant à un sommet v de l'arbre. Si v est une feuille, le robot ne fait rien et la procédure retourne $l(v)$. Si v est un sommet interne, le robot exécute `Vérifie`$(M(v))$, qui retourne un bit b. Alors le robot procède avec la phase suivante, correspondant au sommet de l'arbre étiqueté $l(v)\circ b$.

Lemme 3.7 *Supposons que la hauteur courante de l'arbre T est h et que le nombre de sommets de T est polynomial en n et d. La procédure* `Classifie`() *s'exécute en temps $O(h\hat{n}n^2d)$ et utilise $O(\log\hat{n}+n\log d)$ bits de mémoire.*

Preuve. La complexité en temps de la procédure `Classifie` est bornée par h fois le temps d'exécution de la procédure `Vérifie`, c'est-à-dire $O(h\hat{n}n^2d)$ étapes, d'après le point 5 du lemme 3.5. Concernant la mémoire, $O(\log\hat{n}+n\log d)$ bits sont utilisés pour tous les appels à la procédure `Vérifie` puisque les différents appels sont séquentiels et donc la mémoire peut être réutilisée. La procédure `Classifie` a aussi besoin de mémoriser un pointeur vers le sommet courant de l'arbre. Ce pointeur utilise un espace logarithmique puisque l'arbre a un nombre polynomial de sommets. □

Supposons que le robot applique la procédure `Classifie`() en un sommet u de G. La procédure termine en un sommet u' du graphe et retourne un mot w. Soit v le père dans T de la feuille étiquetée w. Supposons que w termine par un 1. Cela signifie que $M(v)$ est une vue depuis le sommet courant u'. En un certain sens, la procédure `Classifie`() permet au robot d'obtenir de l'information sur l'endroit où il se trouve.

Procédure `Extension`().

Une fois que le robot connaît approximativement sa position dans le graphe grâce à la procédure `Classifie`, il essaie d'améliorer cette connaissance partielle en explorant au

moins un arc inconnu, grâce à la procédure `Nouvelle-Arête`. Par conséquent, un appel à la procédure `Classifie` sera toujours suivi d'un appel à la procédure `Nouvelle-Arête`. Un tel couple d'appels correspond à la procédure `Extension`.

Plus précisément, nous définissons la procédure `Extension`() comme suit. D'abord le robot exécute la procédure `Classifie`(). Soit w la valeur obtenue en retour, et soit v le père dans l'arbre du sommet étiqueté w. Le robot poursuit en exécutant la procédure `Nouvelle-Arête`($M(v)$). Finalement, la procédure `Extension`() retourne w.

Sous-chemins complets.

Dans l'algorithme, le robot utilise le chemin parcouru par la procédure `Extension` pour mettre à jour une carte. Un prérequis sur le chemin fermé P traité par la procédure `MàJ-Carte` est que P puisse être parcouru par le robot sans l'aide du caillou. Comme la procédure `Extension` utilise le caillou, il semble impossible de satisfaire à ce prérequis. En fait, suivre un sous-chemin de P au lieu de P lui-même peut être suffisant pour l'algorithme si le sous-chemin a certaines propriétés. (Ce problème ne se pose pas dans [BFR$^+$02] car la mémoire n'est pas une préoccupation. Il est donc possible pour le robot de stocker la suite complète des numéros de port du chemin pendant que le robot a le caillou, et ensuite le robot peut utiliser cette suite pour mettre à jour la carte partielle).

Soit P un chemin. Un chemin P' est dit être un *sous-chemin complet de* P si P' vérifie les deux propriétés suivantes. Premièrement, tous les sommets et les arcs visités par P sont aussi visités par P'. Deuxièmement, tous ces sommets et tous ces arcs sont visités pour la première fois dans P' dans le même ordre que leur première visite dans P.

Lemme 3.8 *Soit u un sommet du graphe exploré. Supposons que, quand le robot applique la procédure* `Classifie` *à partir du sommet u, il explore un chemin P et la procédure retourne le mot w. Alors le robot, sans l'aide du caillou mais avec la connaissance de w, est capable de suivre depuis u un sous-chemin complet P' de P. Le résultat reste vrai pour la procédure* `Extension`.

Preuve. Supposons que le robot a la connaissance du mot w retourné par la procédure `Classifie` appelée en u. Le mot w est l'étiquette d'une unique feuille v de l'arbre étiqueté T. Soit $v_1 \cdots v_p$ le chemin simple dans T commençant à la racine et terminant au père de v. (Ce chemin peut ne contenir que la racine.) Par définition de la procédure `Classifie`, l'appliquer depuis u consiste à appliquer la procédure `Vérifie` avec les entrées successives $M(v_1), M(v_2), \ldots, M(v_p)$. D'après le point 2 du lemme 3.5, la procédure `Effectif-Vérifie`($M(v_i)$), pour $1 \leq i \leq r$, visite uniquement des sommets et des arcs déjà visités par la sous-procédure `Init-Vérifie`($M(v_i)$). Donc le chemin P' consistant en l'application de la procédure `Init-Vérifie` avec les entrées successives $M(v_1), \ldots, M(v_p)$ est un sous-chemin complet de P. De plus, d'après le point 1 du lemme 3.5, le robot est capable de suivre P' sans le caillou.

Puisque la procédure `Nouvelle-Arête` n'a pas besoin du caillou, le résultat reste vrai pour la procédure `Extension`. □

3.4.4 Algorithme `CEM-Borne-Sup`

Nous avons maintenant tous les éléments pour décrire l'algorithme `CEM-Borne-Sup` qui permet au robot d'explorer un graphe orienté inconnu avec un caillou quand une borne supérieure $\hat{n}$ sur le nombre de sommets est connue (voir Figure 3.2).

Algorithme `CEM-Borne-Sup`
  $T \leftarrow$ l'arbre à 3 sommets composé d'une racine r et de deux feuilles ;
  $M(r) \leftarrow$ la carte partielle minimale ;
  tant que aucune carte partielle n'est complète **faire**
    {*début de la phase*}
    écourter-phase $\leftarrow faux$;
    répète $\hat{n}$ fois ou **jusqu'à ce que** *écourter-phase* $= vrai$;
      $w \leftarrow$ `Extension`() ;
      si w finit par un 0 **alors**
        écourter-phase $\leftarrow vrai$;
    si *écourter-phase* $= vrai$ **alors**
      soit v la feuille de T étiquetée w ;
      ajouter deux fils à v ;
      $M(v) \leftarrow$ la carte partielle minimale ;
    sinon
      {*le robot est dans un cycle C d'appels à* `Extension()`}
      répète $\hat{n}$ fois `Extension`() pour stocker les informations nécessaires pour suivre C sans le caillou ;
      déposer le caillou ;
      répète procédure `Extension`() **jusqu'à ce que** le caillou soit trouvé ;
      {*le robot est maintenant capable de suivre C et de s'arrêter, sans l'aide du caillou*}
      $w \leftarrow$ `Extension`() ;
      soit v la feuille de T étiquetée w ;
      $M \leftarrow M(v')$ où v' est le père de v ;
      soit P'' le chemin fermé commençant à la position courante et correspondant à C ;
      $M' \leftarrow$ `MàJ-Carte`(M, P'') ;
      ajouter deux fils à v ;
      $M(v) \leftarrow M'$;
      si `Contrôle-Isomorphie`$(g(M'), v(M'))$ renvoie `succès` **alors** s'arrêter et retourner M' ;
  fin tant que

FIG. 3.2 – Algorithme `CEM-Borne-Sup`

Comme décrit précédemment, le robot stocke les cartes partielles dans un arbre étiqueté T. Au début de l'algorithme, l'arbre consiste simplement en une racine et deux

feuilles étiquetées 0 et 1. La carte partielle associée à la racine est la carte partielle minimale.

L'algorithme procède par phases. Une phase commence par une suite de $\hat{n}$ appels à la procédure `Extension`. Si, à un moment donné, l'un des appels retourne un mot w se terminant par un 0, la phase est écourtée de la façon suivante. Soit v la feuille de l'arbre étiquetée w. Le sommet v devient un sommet interne auquel deux feuilles sont attachées. La carte partielle associée à v est la carte partielle minimale. La phase se termine avec la mise à jour de l'arbre étiqueté T.

Nous pouvons donc maintenant supposer que les $\hat{n}$ appels à la procédure `Extension` retournent uniquement des mots finissant par un 1. D'après le fait 3.1, le robot est maintenant dans un chemin fermé P. Le reste de la phase a pour but d'identifier plus précisément P pour pouvoir l'utiliser (en fait, un chemin complet de celui-ci) pour mettre à jour une carte partielle par un appel à la procédure `MàJ-Carte`.

Le robot continue la phase par une nouvelle suite de $2\hat{n}$ appels à la procédure `Extension`. Ces appels sont effectués pour obtenir suffisamment d'informations sur le chemin fermé pour pouvoir le parcourir sans le caillou. Le robot mémorise les $2\hat{n}$ résultats des appels à la procédure `Extension`. Ensuite il calcule la période minimale p de cette suite de mots. Cela signifie que le robot n'a besoin de se souvenir que d'une période de p mots pour être capable de reconstruire la suite complète de résultats. Notons que $p \leq n$.

Le chemin fermé P est composé d'une suite de $p' \leq n$ appels à la procédure `Extension`. Par minimalité de p, nous savons que p' est un multiple de p. Pour appliquer la procédure `MàJ-Carte`, le robot doit pouvoir suivre le chemin fermé P et s'arrêter à la fin de P sans l'aide du caillou. Pour rendre possible ce dernier point, une autre suite d'appels à la procédure `Extension` est utilisée.

Pour compter le nombre p' d'appels à la procédure `Extension` dans le chemin fermé P, le robot dépose le caillou sur le sommet courant. Ensuite, sans le caillou, il suit un sous-chemin complet des appels à la procédure `Extension`. Ceci est possible d'après le lemme 3.8 et parce que le robot a mémorisé les mots retournés par les appels à la procédure. Le robot s'arrête quand il trouve le caillou à la fin d'un appel à la procédure `Extension`. A ce moment, le robot peut calculer $q = p'/p$. En utilisant q et les p mots stockés retournés par les appels à la procédure `Extension`, le robot est maintenant capable de suivre un sous-chemin complet P' de P sans le caillou, et de s'arrêter à la fin de P'.

Au début d'un appel à la procédure `Extension`, le robot est perdu dans le graphe en un sommet u. Le robot exécute donc la procédure `Classifie` pour obtenir un peu d'information sur l'endroit où il se trouve. Soit w la sortie de la procédure. Soit P'' le chemin fermé P' décalé de telle façon qu'il commence et se termine maintenant en le sommet courant u' et pas en u. Soit M la dernière carte partielle utilisée dans l'appel à la procédure `Classifie`. Par définition de P, le chemin P'' commence par un appel à la procédure `Nouvelle-Arête`(M).

Nous avions supposé que tous les mots du cycle finissaient par un 1. Donc la dernière vérification de carte partielle, à savoir `Vérifie`(M), a été un succès. Cela veut donc dire que M est une vue depuis la position courante u' du robot. Clairement, P'' commence

et finit en u'. De plus, par définition de la procédure `Nouvelle-Arête`, le robot visite au moins un arc qui n'est pas référencé dans M mais seulement après avoir exploré tous les sommets et tous les arcs référencés dans M. Le chemin P'' est donc compatible avec M. Finalement, le robot applique la procédure `MàJ-Carte`(M, P''). Soit M' la carte partielle retournée par la procédure. La feuille v étiquetée w devient un sommet interne et nous posons $M(v) = M'$. Le sommet v est maintenant le père de deux nouvelles feuilles de l'arbre.

Pour contrôler si la nouvelle carte partielle M' décrit le graphe entier, le robot exécute la procédure `Contrôle-Isomorphie`$(g(M'), v(M'))$ en u'. Si elle rend `succès`, alors le graphe orienté est exploré et cartographié, et le robot s'arrête définitivement. Sinon, le robot termine cette phase et passe à la suivante.

Théorème 3.4 *Pour tout graphe orienté à n sommets de degré sortant d, et étant donné une borne supérieure $\hat{n}$ sur n, l'algorithme* `CEM-Borne-Sup` *résout le problème de la cartographie en temps $O((\hat{n} + n^2 d)\hat{n}n^6 d^3)$, par un robot utilisant un caillou et une mémoire de taille $O(\hat{n} \log n + n^2 d(\log n + \log d))$ bits.*

Preuve. D'une part, à la fin de chaque phase, l'arbre étiqueté contient plus d'informations : une feuille de l'arbre devient un sommet interne auquel sont attachées deux nouvelles feuilles, et une carte partielle est stockée dans le nouveau sommet interne. D'autre part, l'algorithme ne peut pas se terminer avant d'avoir obtenu une une carte complète du graphe (grâce à l'appel à la procédure `Contrôle-Isomorphie` avant de s'arrêter). Donc pour prouver que l'algorithme `CEM-Borne-Sup` est correct, il suffit de montrer que l'arbre étiqueté ne peut pas devenir infiniment grand.

Pour toute feuille v d'étiquette $w = l(v)$, nous définissons la classe C_w composée de sommets de G. Un sommet u de G appartient à la classe C_w si et seulement si la procédure `Classifie` renvoie w lorsqu'elle est exécutée en u.

Assertion. Soit S l'ensemble des feuilles v telles que $l(v)$ termine par un 1. La classe associée à chaque feuille de S est non vide.

Au début de l'algorithme, le résultat est vrai car tous les sommets de G sont dans la classe C_1. Supposons que l'assertion est vraie au début d'une phase. Soit w l'étiquette de la feuille v qui devient un sommet interne de l'arbre à la fin de la phase. D'après la description de l'algorithme, w est le résultat d'un appel à la procédure `Classifie` réalisé dans la phase courante depuis un sommet u. Donc $u \in C_w$.

Supposons d'abord que w finit par un 0. Quand l'arbre étiqueté est mis à jour, une nouvelle feuille étiquetée $w' = w \circ 1$ est ajoutée à l'ensemble S. Puisque $M(v)$ est la carte partielle minimale, la procédure `Vérifie`$(M(v))$ ne fait rien et retourne 1. Par conséquent, si le robot applique la procédure `Classifie` depuis le sommet u, celle-ci retourne w'. Donc $C_{w'}$ contient u et est donc non vide.

Supposons maintenant que w termine par un 1. Quand l'arbre est mis à jour, une nouvelle feuille étiquetée $w' = w \circ 1$ est ajoutée à l'ensemble S, en remplacement de v, qui est devenu un sommet interne. Le chemin fermé P'' utilisé par la procédure `MàJ-Carte` commence au sommet u' atteint depuis u par l'application de la procédure `Classifie`.

Comme cet appel renvoie w, on a $u \in C_w$. La nouvelle carte partielle $M(v)$ est par construction une vue depuis u' et donc la procédure `Classifie` lancée en u retournera $w' = w \circ 1$. Donc $C_{w'}$ contient u et est donc non vide.

Dans les deux cas, les classes correspondant aux autres feuilles dans S restent inchangées. Ceci conclut la preuve de l'assertion.

Une conséquence directe de l'assertion est que la taille de l'ensemble S est au plus le nombre n de sommets dans G. Pour chaque sommet v de S, considérons l'ensemble S_v des ancêtres v' de v tels que $l(v) = l(v') \circ 11 \ldots 11$. Ces ensembles forment une partition des sommets internes car tout sommet interne a exactement deux fils. Soit w' le plus long préfixe de $l(v)$ tel que, soit w' finit par un 0, soit w' est le mot vide. Le sommet v' étiqueté w' est soit la racine soit un sommet devenu interne quand une phase a été écourtée car un appel à la procédure `Classifie` a retourné un mot finissant par 0. Donc $M'(v)$ est la carte partielle minimale. Toutes les cartes partielles des autres sommets de S_v sont des mises à jour successives de $M(v')$. Chaque mise à jour ajoute au moins un arc à la carte partielle précédente, grâce à la procédure `Nouvelle-Arête`. Par conséquent, S_v contient au plus nd sommets. L'arbre étiqueté T contient donc au plus n^2d sommets.

Ceci conclut la preuve de correction de l'algorithme. Nous nous concentrons maintenant sur le temps et l'espace utilisés par celui-ci.

Puisque la hauteur de T est au plus n^2d, la procédure `Classifie` consiste en au plus n^2d appels à la procédure `Vérifie` et donc s'exécute en temps $O(\hat{n}n^4d^2)$ et utilise $O(\log \hat{n} + n \log d)$ bits, d'après le lemme 3.7. La procédure `Nouvelle-Arête` s'exécute en temps $O(n^2d)$ et utilise $O(n \log d)$ bits de mémoire. Donc la procédure `Extension` s'exécute en temps $O(\hat{n}n^4d^2)$ et utilise $O(\log \hat{n} + n \log d)$ bits de mémoire.

Nous nous intéressons maintenant à une phase de l'algorithme `CEM-Borne-Sup`. Le pire cas en termes de temps et d'espace est lorsque la phase se termine normalement, par la mise à jour d'une carte partielle grâce à un chemin fermé P''. Ce chemin est composé d'au plus n appels à la procédure `Extension` et donc sa longueur est en $O(\hat{n}n^5d^2)$. Par conséquent la procédure `MàJ-Carte` s'exécute en temps $O(\hat{n}n^6d^3)$, d'après le point 3 du lemme 3.6. De plus $O(\hat{n})$ répétitions de la procédure `Extension` sont utilisées pour déterminer P pour un coût en temps en $O(\hat{n}^2n^4d^2)$. Le coût total de la phase est donc de $O((\hat{n} + n^2d)\hat{n}n^4d^2)$ traversées d'arcs.

Concernant la mémoire, le robot doit stocker le nombre q et les p mots retournés par les appels à la procédure `Extension` pour être capable de suivre le chemin P'' sans le caillou. Rappelons que T possède au plus n feuilles ayant une étiquette finissant par un 1. Ces étiquettes peuvent être rangées par ordre lexicographique. Donc, au lieu de mémoriser les p mots, le robot stocke seulement les rangs correspondants. En résumé, seuls $O(\hat{n})$ rangs de $O(\log n)$ bits sont stockés. Sans prendre en compte le coût en mémoire de l'arbre étiqueté, qui contient la carte partielle calculée dans la phase par la procédure `MàJ-Carte`, une phase utilise $O(\hat{n} \log n + n \log d)$ bits.

Cet espace peut être réutilisé pour chaque phase. Seul l'arbre étiqueté est gardé tout au long des différentes phases. Il a $O(n^2d)$ sommets internes et donc $O(n^2d)$ cartes partielles doivent être mémorisées. Nous pouvons sauver de l'espace en notant que toutes les cartes partielles correspondant aux sommets d'un ensemble S_v sont étroitement liées : la carte

partielle d'un sommet v' de S_v est un préfixe de la carte partielle d'un autre sommet v'' si v' est un ancêtre de v''. Par conséquent, il est presque suffisant de stocker uniquement la carte partielle $M = (N, L)$ du sommet le plus profond dans l'arbre de chaque S_v, c'est-à-dire la plus grande carte partielle d'un sommet de S_v. Pour pouvoir calculer les autres cartes partielles, nous utilisons deux nombres pour chaque carte partielle $M' = (N', L')$. Le premier est N' et le second est $|L'|$. La partie manquante L' peut être retrouvée à partir de la connaissance de $M = (N, L)$ et de $|L'|$ car L' est le préfixe de L de longueur $|L'|$. Donc l'espace utilisé pour les cartes partielles correspondant à un ensemble S_v est $O(nd(\log n + \log d))$ bits. Puisqu'il y a n ensembles de ce type, le stockage de l'arbre étiqueté utilise $O(n^2 d(\log n + \log d))$ bits.

Pour conclure, l'algorithme `CEM-Borne-Sup` utilise $O(\hat{n} \log n + n^2 d(\log n + \log d))$ bits et s'exécute en temps $O((\hat{n} + n^2 d)\hat{n} n^6 d^3)$ car il y a au plus $O(n^2 d)$ phases. □

3.4.5 Algorithme `Compacted-Explore-and-Map`

Nous supposons maintenant qu'aucune borne supérieure sur le nombre n de sommets du graphe orienté exploré n'est connue. Le robot devine différentes valeurs et exécute l'algorithme `CEM-Borne-Sup` pour chacune d'elles. L'algorithme `Compacted-Explore-and-Map` consiste en des appels successifs à `CEM-Borne-Sup` avec des valeurs croissantes de $\hat{n}$. Plus précisément, l'algorithme utilise la suite 2^{2^k}. Après $\log \log n$ essais, l'algorithme exécute `CEM-Borne-Sup` avec une valeur $\hat{n}$ plus grande ou égale à n et donc réussit à explorer et à cartographier le graphe orienté.

Preuve du théorème 3.3. L'algorithme `CEM-Borne-Sup` est exécuté $O(\log \log n)$ fois. La valeur de $\hat{n}$ pour chaque exécution est bornée par n sauf pour la dernière où $\hat{n}$ est borné par n^2. Par conséquent, l'algorithme `Compacted-Explore-and-Map` s'exécute en temps $O(n^{10} d^4)$ et utilise $O(n^2 d(\log n + \log d))$ bits.

Une exécution se termine lorsque le caillou est perdu. En effet, si n est sous-estimé, les $\hat{n}$ répétions préliminaires ne sont pas forcément suffisantes pour assurer au robot d'être dans un chemin fermé. Par conséquent, le robot n'utilise pas plus de $O(\log \log n)$ cailloux. Rigoureusement, un autre événement peut venir perturber les exécutions de l'algorithme `CEM-Borne-Sup`. Il peut arriver que le robot trouve un caillou perdu dans une exécution antérieure au lieu du caillou déposé en dernier. Comme il a été noté dans [BFR+02], ceci peut être évité en ramassant tous les cailloux présents sur un chemin fermé avant de travailler dessus avec le caillou. De plus, pour assurer que l'algorithme ne terminera pas avant d'avoir complètement exploré le graphe, l'algorithme demande au robot de parcourir le graphe pour ramasser tous les cailloux déposés dans les exécutions précédentes avant de vérifier pour la dernière fois que la carte calculée est bien une carte du graphe orienté exploré. □

3.5 Perspectives

Notre algorithme `Teste-toutes-les-cartes` nécessite de stocker la carte à tester. Cependant l'exploration avec arrêt est une tâche a priori plus facile que la cartographie. On pourrait donc espérer trouver un algorithme utilisant moins de mémoire que la quantité nécessaire pour stocker une carte. Une autre direction intéressante d'investigation est la recherche d'un algorithme compact d'exploration sous la contrainte d'une exécution en temps polynomial. Dans ce chapitre, nous avons décrit un algorithme d'exploration en temps polynomial utilisant un robot ayant une mémoire de taille $O(n^2 d(\log n + \log d))$ bits. Cette valeur est loin de la borne inférieure $\Omega(n \log d)$. L'algorithme exponentiel nécessite quant à lui une mémoire de $O(nd(\log n + \log d))$ bits. Il serait intéressant de trouver le compromis exact entre le temps et l'espace mémoire pour l'exploration des graphes orientés.

Chapitre 4

Impact de la taille mémoire sur les capacités d'exploration

Dans ce chapitre, nous étudions plus en détail la relation entre le nombre d'états d'un automate et le nombre de graphes que celui-ci peut explorer. En particulier, existe-t-il une hiérarchie stricte des automates, c'est-à-dire est-ce que toute augmentation du nombre d'états entraîne une augmentation de la capacité d'exploration ?

4.1 Introduction

Budach a montré qu'aucun automate fini ne peut explorer tous les graphes [Bud78]. Rollik a même montré l'existence d'un piège commun à tous les automates d'une taille donnée [Rol80] en montrant l'existence d'un piège pour toute équipe finie d'automates finis. Dans le chapitre 2, nous avons construit des pièges plus petits pour ce type d'équipes. Il découle de ces résultats l'existence d'un piège planaire 3-régulier de taille $K^{O(K)}$ commun à tous les automates à K états. Comme tout graphe est explorable par un automate ayant un nombre d'états polynomial en son nombre de sommets [Rei05], on en déduit qu'en augmentant exponentiellement le nombre d'états que peut avoir un automate, on augmente strictement le nombre de graphes que celui-ci peut explorer. Nous conjecturons que c'est le cas pour toute augmentation du nombre d'états, si petite soit elle.

Nous avons vu dans le chapitre 2 que les graphes réguliers sont difficiles à explorer. En effet, l'automate ne peut pas distinguer les différents sommets en utilisant leur degré. Dans ce chapitre, nous nous concentrons sur les graphes réguliers car ceux-ci semblent capturer les cas difficiles d'exploration.

Nous utilisons ici le modèle de l'automate de Mealy qui, comme nous l'avons noté dans l'introduction de cette thèse, capture mieux la quantité de mémoire nécessaire à l'exploration lorsque celle-ci est faible. Par contre, contrairement à la définition 8, nous considérons dans ce chapitre que l'automate débute en arrivant sur un certain sommet de départ depuis un certain port de départ. Autrement dit, l'automate ne décide pas du numéro de port d'entrée de départ. (Dans la définition 8, le port d'entrée initial est considéré être $\perp$.)

Puisque nous nous intéressons à l'exploration des graphes indépendamment de leur orientation locale et de la position de départ de l'automate, nous introduisons la notion suivante :

Définition 27 *Soit K un entier strictement positif. La* classe des graphes K-explorables *est l'ensemble $\mathcal{G}_K$ des graphes réguliers tel que pour tout $G \in \mathcal{G}_K$, il existe un automate $\mathcal{A}$ d'au plus K états visitant tous les sommets de G pour toute orientation locale de G, tout sommet de départ u_0 et tout numéro de port d'entrée initial i_0.*

Clairement, la suite d'ensembles $\mathcal{G}_K$ est croissante, c'est-à-dire $\mathcal{G}_K \subseteq \mathcal{G}_{K+1}$ pour tout $K > 0$. Nous pouvons maintenant formuler précisément notre conjecture :

Conjecture. *Pour tout $K > 0$, $\mathcal{G}_K \neq \mathcal{G}_{K+1}$.*

Les résultats de ce chapitre étayent partiellement cette conjecture. Nous prouvons qu'une augmentation polynomiale du nombre d'états entraîne une augmentation de la capacité d'exploration (Section 4.2). Plus précisément, nous montrons qu'il existe une fonction polynomiale $f : \mathbb{N} \to \mathbb{N}$, telle que $\mathcal{G}_K$ est strictement inclus dans $\mathcal{G}_{f(K)}$, pour tout $K > 0$. Notons qu'une telle augmentation du nombre d'états équivaut à seulement multiplier par une constante le nombre de bits de mémoire. Par ailleurs, nous montrons que la conjecture est vraie pour les petites valeurs de K (Section 4.3). Plus précisément, nous montrons que $\mathcal{G}_K$ est strictement inclus dans $\mathcal{G}_{K+1}$, pour $K = 1, 2$. De plus, nous caractérisons exactement les classes $\mathcal{G}_1$ et $\mathcal{G}_2$. La classe $\mathcal{G}_1$ des graphes réguliers pouvant être explorés par un automate sans mémoire (c'est-à-dire à un état) est constituée des cycles et des graphes complets à 1 et 2 sommets. La classe $\mathcal{G}_2$ des graphes réguliers pouvant être explorés par un automate à deux états est constituée de tous les cycles et de toutes les cliques.

4.2 Augmentation polynomiale de la mémoire

Dans cette section, nous prouvons qu'une augmentation polynomiale du nombre d'états d'un automate entraîne une augmentation de sa capacité d'exploration des graphes. Plus précisément, le résultat principal de cette section est le suivant.

Théorème 4.1 *Il existe une fonction polynomiale $f : \mathbb{N} \to \mathbb{N}$, telle que $\mathcal{G}_K$ est strictement inclus dans $\mathcal{G}_{f(K)}$, pour tout $K > 0$.*

Dans cette section, nous n'utilisons pas la méthode de l'automate réduit, décrite dans le chapitre 2, mais une nouvelle méthode, la méthode des pseudo-palindromes, pour construire des pièges. Nous commençons par les principales définitions et propriétés.

Une suite L d'étiquettes (numéros de port) est un *pseudo-palindrome* si l'une des deux conditions suivantes est vérifiée : (1) $L = \emptyset$, ou (2) $L = L' \circ (\ell, \ell) \circ L''$, où $L' \circ L''$ est un pseudo-palindrome, ℓ est une étiquette, et $\circ$ est l'opérateur de concaténation.

Une suite L' est une *réduction* de L si $L' = A \circ B$ et $L = A \circ L'' \circ B$ où L'' est un pseudo-palindrome non vide, et A et B sont deux suites arbitraires (éventuellement vides). Une

suite est dite *pp-libre* si elle n'admet aucune réduction. Une suite L' est la *pp-réduction* d'une suite L si L' est pp-libre et obtenu à partir de L par réductions successives. On peut aisément vérifier que la pp-réduction d'une suite finie est unique (voir, par exemple, la section 1.7 de [Bud78]). Par exmple, la pp-réduction de 11221211213223311313323112 21 est 1231. De façon évidente, étant donné un quelconque graphe homogène $G = (V, E)$ et un sommet $u \in V$, toute suite L de numéros d'arêtes définit un chemin P depuis u dans G. Si L est un pseudo-palindrome, alors P commence et finit en u.

Le principal outil pour prouver le théorème 4.1 est le lemme suivant, qui présente par ailleurs un intérêt propre.

Lemme 4.1 *Pour tout $K > 0$ il existe un graphe planaire régulier G de degré 3, ayant $O(K^3)$ sommets, tel que $G \notin \mathcal{G}_K$.*

Preuve. Pour $K = 1$ le lemme découle des Propositions 4.1 et 4.2 de la section suivante. Fixons un entier $K > 1$.

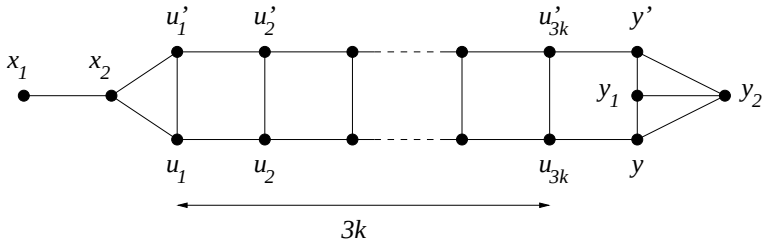

FIG. 4.1 – Le sous-graphe T

Nous définissons d'abord un graphe T qui est utilisé comme sous-graphe de G (cf. Figure 4.1). Le graphe T a $6K + 6$ sommets $x_1, x_2, y, y', y_1, y_2, u_1, \ldots, u_{3K}, u'_1, \ldots, u'_{3K}$ et $9K + 7$ arêtes $\{x_1, x_2\}$, $\{x_2, u_1\}$, $\{x_2, u'_1\}$, $\{y_1, y_2\}$, $\{y, y_1\}$, $\{y, y_2\}$, $\{y', y_1\}$, $\{y', y_2\}$, $\{y, u_{3K}\}$, $\{y', u'_{3K}\}$, $\{u_{3K}, u'_{3K}\}$ et $\{u_i, u'_i\}$, $\{u_i, u_{i+1}\}$, $\{u'_i, u'_{i+1}\}$ pour tout $1 \leq i < 3K$. Tous les sommets sont donc de degré 3 sauf x_1 qui est de degré 1. Le sous-graphe de T induit par les sommets y, y', y_1, y_2 est appelé la *queue* du graphe T.

Soit $\widehat{G}$ une chaîne de cycles disjoints de toutes les tailles de 3 à $3K$. Plus précisément, soient v_p et v'_p deux sommets différents du cycle de longueur p, pour $3 \leq p \leq 3K$. Connectons les cycles par les arêtes $\{v'_p, v_{p+1}\}$, pour $3 \leq p < 3K$. Maintenant attachons un graphe T à tout sommet w de $\widehat{G}$ de degré 2 en identifiant w de $\widehat{G}$ et x_1 de T. Soit G le graphe obtenu. G est régulier de degré 3. Puisque T a $O(K)$ sommets, le graphe G a $O(K^3)$ sommets.

Il reste à prouver que $G \notin \mathcal{G}_K$. Fixons un automate $\mathcal{A}$ avec K états ($K > 1$). Notre but est de trouver une orientation locale de G, un sommet de départ et un port d'entrée initial, tel que l'automate $\mathcal{A}$ ne visite pas tous les sommets de G. Nous utilisons une orientation locale spécifique qui est partiellement à arêtes colorées. L'automate $\mathcal{A}$ ne traversera que des arêtes colorées, même si toutes les arêtes du graphe ne le sont pas.

Si un automate ne traverse que des arêtes colorées dans un graphe régulier, alors la trajectoire de l'automate (la suite de numéros de port) et sa suite d'états dépendent

toutes deux exclusivement de la paire (s_0, i_0), où s_0 est l'état initial et i_0 le numéro de port d'entrée initial. En particulier, après $3K + 1$ étapes au plus, l'automate arrive sur un sommet dans un état s par le port i, tels que la paire (s, i) a déjà été rencontrée sur un certain sommet précédémment visité. Donc, après cette étape, la suite des états et la suite des couleurs d'arêtes $L = \{a_n\}_{n\geq 0}$ sont périodiques. Il existe q et p, avec $q+p \leq 3K$, tels que pour tout $i \geq q$, on a $a_i = a_{i+p}$. Soit $L' = b_1, b_2, \ldots$ la pp-réduction de la suite L. Nous avons $i_1 < i_2 < \ldots$ tels que $b_r = a_{i_r}$.

Supposons d'abord que la suite L' est finie. Alors la suite $a_{q+1}, \ldots, a_{q+2p}$ est un pseudo-palindrome. Choisissons un sommet quelconque w dans le cycle de taille 3 de $\widehat{G}$. Choisissons un 3-coloriage propre arbitraire des trois arêtes du cycle. Nous étendons maintenant le 3-coloriage propre au reste du graphe excepté pour la queue de chaque T. (Ceci peut facilement être fait, alors qu'un 3-coloriage propre des arêtes du graphe entier, même du sous-graphe T, est impossible). Etiquetons arbitrairement les ports restants du graphe (les ports dans les queues des copies de T) par les entiers 0,1,2 en chaque sommet. Il reste à calculer le sommet de départ de l'automate. Depuis le sommet w, traversons les arêtes colorées $a_q, a_{q-1}, \ldots, a_1$. Ce parcours mène à un sommet u_0. Nous affirmons que l'automate, démarrant en u_0 avec le port initial i_0, ne visite jamais aucune queue d'une copie de T et donc échoue dans l'exploration de G. Pendant les q premières étapes, l'automate est à distance au plus $3K$ de w car $q \leq 3K$. A l'étape $q + 2rp$, pour $r \geq 0$, l'automate est toujours en w car $a_q, a_{q+1}, \ldots, a_{q+2p}$ est un pseudo-palindrome. Puisque $p \leq 3K$, l'automate est de nouveau à distance au plus $3K$ de w. Puisque w est sur l'un des $3K - 2$ cycles, l'automate ne visite jamais aucune queue d'une copie de T, et donc ne traverse jamais des arêtes non colorées. Par conséquent, $\mathcal{A}$ ne réussit pas à explorer G.

Nous supposons à partir de maintenant que L' est infini. Alors L' hérite de la propriété de périodicité : il existe $q' \leq q$ et $p' \leq 3K$ tels que $b_r = b_{r+p'}$ pour tout $r \geq q'$. Si la période est 2, nous choisissons $p' = 4$ pour éviter les arêtes multiples (un cycle de longueur 2). En commençant d'un de ses sommets u, étiquetons les arêtes du cycle de taille p' dans G avec les couleurs $b_{q'+1}, b_{q'+2}, \ldots, b_{q'+p'}$. Nous étendons maintenant le coloriage au reste du graphe excepté pour la queue de chaque copie de T. Etiquetons arbitrairement les ports restants du graphe par les entiers 0,1,2 en chaque sommet. Il reste à calculer le sommet de départ de l'automate. Depuis le sommet u, traversons les arêtes colorées $b_{q'}, b_{q'-1}, \ldots, b_1$. Ce parcours mène à un sommet u_0. Nous affirmons que l'automate, démarrant en u_0 avec le port initial i_0, ne visite jamais aucune queue d'une copie de T et donc échoue dans l'exploration de G. Pendant les $i_{q'}$ premières étapes, l'automate est à distance au plus $3K$ de u car $i_{q'} \leq 3K$. A l'étape $i_{q'}$, l'automate est en u et il revient ensuite en u toutes les p étapes car u est sur le cycle de longueur p', qui est étiqueté pour piéger l'automate. Puisque $p \leq 3K$, l'automate est de nouveau à distance au plus $3K$ de u. Comme u est sur l'un des $3K - 2$ cycles, l'automate ne visite jamais aucune queue d'une copie de T et par conséquent ne réussit pas à explorer G.

Pour tout automate à K états, nous avons construit une orientation locale, un sommet de départ u_0 et un port d'entrée initial i_0 tels que l'automate ne réussit pas à explorer G. Donc $G \notin \mathcal{G}_K$. □

Preuve du théorème 4.1. Nous pouvons maintenant conclure la preuve du théorème comme suit. Pour tout K, prenons le graphe G avec $n \in O(K^3)$ sommets dont l'existence est prouvée par le lemme 4.1. Donc $G \notin \mathcal{G}_K$. Prenons une UTS de longueur $O(n^3 \log n)$ (cf. [AKL+79]) pour tous les graphes réguliers de degré 3 et de taille n. Cette suite peut être exécutée par un automate de $O(n^3 \log n)$ états comme nous l'avons montré dans la discussion précédant la définition 17. Il existe donc une fonction polynomiale $f(K)$ telle qu'un automate de $f(K)$ états explore le graphe G, quels que soient l'orientation locale, le sommet de départ et le numéro de port de départ. Cela implique $G \in \mathcal{G}_{f(K)}$. □

4.3 Puissance d'exploration des petits automates

Dans cette section, nous montrons que notre conjecture concernant la hiérarchie $\mathcal{G}_K$ est vraie pour $K = 1$ et 2, c'est-à-dire $\mathcal{G}_1$ est strictement inclus dans $\mathcal{G}_2$, qui est strictement inclus dans $\mathcal{G}_3$. Pour le prouver, nous déterminons exactement les éléments de $\mathcal{G}_1$ et $\mathcal{G}_2$.

Proposition 4.1 *La classe $\mathcal{G}_1$ des graphes réguliers pouvant être explorés par un automate sans mémoire (un automate à un état) consiste en : le graphe à un sommet, le graphe complet à deux sommets et tous les cycles.*

Preuve. Le fait que les graphes complets à un et deux sommets ainsi que tous les cycles peuvent être explorés par un automate à un état est immédiat (pour les cycles, la fonction de transition/sortie est $\delta(s_{init}, i, 2) = (s_{init}, i+1)$). Supposons qu'il existe un autre graphe d-régulier G appartenant à $\mathcal{G}_1$. Puisque les seuls arbres réguliers sont les graphes complets à un et deux sommets, le graphe G doit contenir un cycle C. Puisque G lui-même n'est pas un cycle, il y a deux cas : soit il existe des sommets de G qui ne sont pas dans C, soit G contient une corde de C. Dans ce dernier cas, G contient un cycle plus court C' contenant cette corde. Il s'ensuit que dans le second cas, il existe des sommets de G' qui ne sont pas dans C'. Donc, dans tous les cas, G contient un cycle C^* et un sommet v qui n'appartient pas à ce cycle. Soit δ la fonction de transition/sortie d'un automate de Mealy ayant un seul état s_{init} et explorant G. Si $\delta(s_{init}, i, d) = (s_{init}, i)$, pour tout i, alors l'automate ne peut explorer que deux sommets adjacents et donc il n'explore pas G. Autrement, soit i le numéro de port tel que $\delta(s_{init}, i, d) = (s_{init}, j)$, avec $i \neq j$. Nous étiquetons tous les ports sur le cycle C^* de telle façon que toute arête du cycle C^* ait le numéro de port i à une extrémité et j à l'autre. Pour cette orientation locale, l'automate entrant sur un sommet quelconque du cycle par le port d'entrée initial i explore perpétuellement le seul cycle C^* sans jamais visiter v. Donc il n'explore pas G. Cette contradiction implique que G ne peut pas appartenir à $\mathcal{G}_1$. □

Dans le but de caractériser la classe $\mathcal{G}_2$, nous avons besoin de quelques résultats de théorie des graphes. Rappelons qu'un cycle est dit élémentaire s'il ne contient pas deux fois le même sommet.

Lemme 4.2 *Tout graphe régulier de degré plus grand que 2 contient un cycle élémentaire de longueur paire.*

Lemme 4.3 *Tout graphe régulier de degré plus grand que* 2 *et différent de la clique à* 4 *sommets contient un cycle élémentaire non-hamiltonien de longueur paire.*

Preuve. Prenons le cycle élémentaire C de longueur paire dont l'existence est garantie par le lemme 4.2. S'il n'est pas hamiltonien, il n'y a rien à faire. Supposons donc que C est un cycle hamiltonien. Nous supposons d'abord que le graphe est de degré au moins 4. Prenons un sommet quelconque v de C. Il doit exister au moins deux cordes de C d'extrémité v, disons $\{v, x\}$ et $\{v, y\}$. Considérons le cycle C' composé de la corde $\{v, y\}$ et de la partie de C contenant x (cf. Figure 4.2 (a)). Si la longueur de C' est paire, alors c'est un cycle élémentaire non-hamiltonien de longueur paire. Autrement, l'un des cycles suivants doit avoir cette propriété : soit le cycle composé de $\{v, x\}$ et de la partie de C ne contenant pas y, soit le cycle contenant $\{v, x\}$ et $\{v, y\}$ et la partie de C entre x et y ne contenant pas v (la somme des longueurs de ces cycles est de la même parité que la longueur de C').

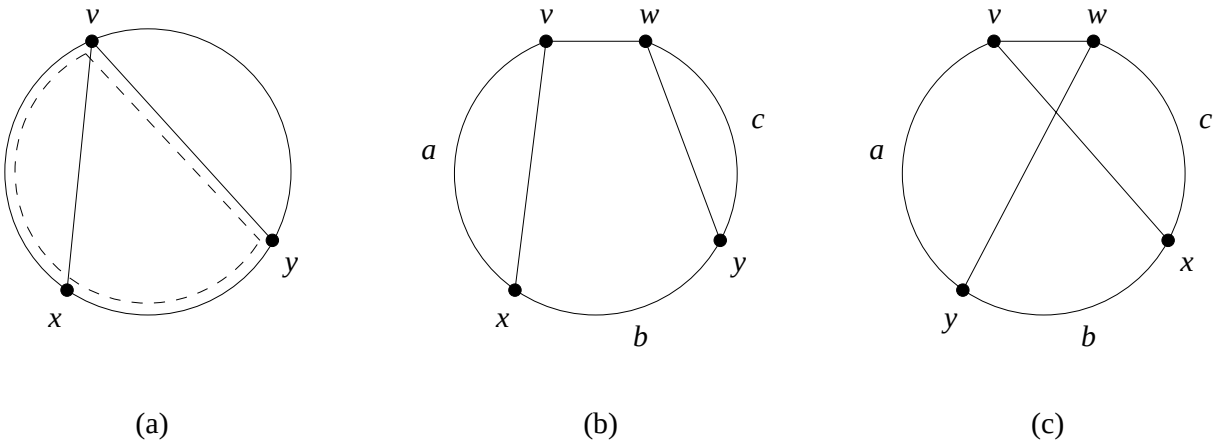

FIG. 4.2 – (a) degré au moins 4, (b) et (c) Cordes dans le cycle C

Il reste à considérer le cas où le degré de G est 3. Considérons deux sommets voisins v et w dans C (cf. Figure 4.2 (b) et (c)). Chacun de ces sommets est une extrémité d'une corde, $\{v, x\}$ et $\{w, y\}$, respectivement, avec $x \neq y$. Si ces cordes ne s'intersectent pas (lorsque le cycle C est représenté par un cercle dans le plan, cf. Figure 4.2 (b)), il y a deux cas. Si la partie a ou la partie c du cycle est paire alors le cycle (a, v, x) (resp. (c, y, w)) est comme requis. Si a et c sont tous deux de taille impaire alors la partie b doit être de taille paire et le cycle (v, w, y, b, x) est comme requis. Supposons donc que les cordes $\{v, x\}$ et $\{v, y\}$ s'intersectent (cf. Figure 4.2 (c)). Considérons deux cas.

Cas 1. La partie b est de taille paire. Si a ou c est non vide, le cycle (v, x, b, y, w) est non-hamiltonien. Il est de taille paire et donc nous avons le résultat dans ce cas. Autrement b doit être non vide car G n'est pas la clique à quatre sommets. Donc le cycle (v, x, w, y) est un cycle non-hamiltonien à 4 sommets.

Cas 2. La partie b est de taille impaire. Soit la partie a, soit la partie c doit être de taille paire car la somme des tailles de a, b et c doit être paire. Sans perte de généralité, supposons que a est de taille paire. Alors le cycle (v, a, y, b, x) est un cycle non-hamiltonien de longueur paire. □

Lemme 4.4 *Tout graphe régulier à n sommets de degré plus grand que 2 et différent de la clique à 4 sommets contient un cycle élémentaire de longueur au plus $n-2$.*

Preuve. Considérons un graphe régulier à n sommets de degré plus grand que 2 et différent de la clique à 4 sommets. D'après le lemme 4.3 il contient un cycle élémentaire non-hamiltonien C de longueur paire. S'il existe au moins deux sommets hors de C, nous avons le résultat. Supposons donc qu'il existe exactement un sommet v hors de C. Si C a une corde dans G, alors les deux cycles plus courts contenant cette corde ont la propriété requise. Donc nous supposons que C ne possède pas de corde. Puisque le degré de G est au moins 3, tous les sommets de C doivent être adjacents à v. Donc v est de degré $n-1$. Comme G est régulier, ce doit être une clique. Puique ce n'est pas la clique à 4 sommets, G doit être une clique d'au moins 5 sommets. Donc G contient un cycle de longueur au plus $n-2$. □

Notre prochain résultat caractérise la classe $\mathcal{G}_2$.

Proposition 4.2 *La classe $\mathcal{G}_2$ des graphes réguliers pouvant être explorés par un automate à deux états consiste en toutes les cliques et tous les cycles.*

Preuve. Puisque tous les cycles appartiennent à la classe $\mathcal{G}_1$, ils appartiennent aussi à la classe $\mathcal{G}_2$. Considérons une clique quelconque, de degré d, et l'automate à deux états s_0 et s_1 ayant la fonction de transition/sortie suivante : pour tout i,

$$\begin{cases} \delta(s_0, i, d) = (s_1, i+1) \\ \delta(s_1, i, d) = (s_0, i) \end{cases}$$

Il est facile de vérifier que l'automate ci-dessus, démarrant dans l'état s_0 sur un sommet quelconque v de la clique, pour tout port d'entrée initial, visite l'étoile centrée en v et donc explore la clique.

Il reste à prouver qu'aucun graphe régulier autre qu'un cycle ou qu'une clique n'appartient à la classe $\mathcal{G}_2$. Supposons qu'un tel graphe G de degré d existe, et soit $\mathcal{A}$ un automate à deux états explorant G. Soient s_0 et s_1 les états de $\mathcal{A}$, où s_0 est l'état initial. Nous définissons, pour tout i,

$$\begin{cases} \delta_0(i) = \delta(s_0, i, d) \\ \delta_1(i) = \delta(s_1, i, d) \end{cases}$$

où δ est la fonction de transition/sortie de $\mathcal{A}$.

Supposons d'abord que $\delta_0(i) = (s_0, j)$ pour un certain numéro de port i. Si $j = i$, l'automate ne peut explorer que deux sommets adjacents s'il est démarré sur une arête dont les deux ports sont étiquetés i. Si $j \neq i$ alors soit C un cycle non-hamiltonien dans G (cf. lemme 4.3). Etiquetons tous les ports du cycle C par la suite $(i, j, i, j, \dots)$. Pour cet étiquetage, l'automate entrant dans un sommet quelconque du cycle par le port d'entrée initial i dans l'état s_0 explore uniquement C et donc ne réussit pas à explorer G.

A partir de maintenant, nous supposons donc que $\delta_0(i) = (s_1, -)$ pour tout i.

Ensuite, supposons que $\delta_0(i) = (s_1, i)$, pour tout i.

Si $\delta_1(j) = (s_0, -)$, pour tous les numéros de port j, alors soit u un sommet quelconque de G. Soient $v_0, v_1, \ldots, v_{d-1}$ les d voisins de u. Si l'automate arrive sur un sommet v_k par le port i dans l'état s_0, alors il revient en u dans l'état s_1. Si l'automate arrive sur le sommet u dans l'état s_1, alors il quitte u dans l'état s_0 vers un certain v_s. Donc, pour cet étiquetage, l'automate entrant dans le sommet v_0 dans l'état s_0 par le port d'entrée initial i menant à u explore seulement u et tous ses voisins. Puisque G n'est pas une clique, l'automate ne réussit pas à explorer G.

Autrement, soit j le numéro de port pour lequel $\delta_1(j) = (s_1, k)$, pour un certain k. Si $k = j$, l'automate explore seulement deux sommets adjacents quand il démarre sur une arête dont les deux ports sont étiquetés j. Si $k \neq j$ alors soit C un cycle non-hamiltonien dans G. Etiquetons tous les ports du cycle C par la suite $(j, k, j, k, \ldots)$. Pour cet étiquetage, l'automate entrant dans un sommet quelconque du cycle par le port d'entrée initial k dans l'état s_0 explore uniquement C et donc ne réussit pas à explorer G.

Par conséquent, dans le reste de la preuve, nous supposons que $\delta_0(i) = (s_1, -)$ pour tout i, et qu'on n'a pas $\delta_0(i) = (s_1, i)$ pour tout i. Nous considérons trois cas.

Cas 1 : $\delta_1(j) = (s_0, j)$ pour un certain numéro de port j. Soit u un sommet arbitraire de G. Soient $v_0, v_1, \ldots, v_{d-1}$ les d voisins de u. A chaque v_k, donnons le numéro de port j à l'arête menant à u. Si l'automate arrive sur un sommet v_k par le port j dans l'état s_1, alors il revient en u dans l'état s_0. Si l'automate arrive sur le sommet u dans l'état s_0, alors il quitte u dans l'état s_1 vers un certain v_s. Donc, pour cet étiquetage, l'automate entrant dans le sommet u dans l'état s_0 par un port d'entrée initial quelconque explore seulement u et tous ses voisins. Puisque G n'est pas une clique, l'automate ne réussit pas à explorer G.

Cas 2 : $\delta_1(j) = (s_0, k)$, avec $k \neq j$, pour un certain numéro de port j. Soit i un numéro de port tel que $\delta_0(i) = (s_1, l)$ avec $l \neq i$. Soit C un cycle non-hamiltonien de taille paire dans G, dont l'existence est prouvée par le lemme 4.3. Etiquetons tous les ports du cycle C par la suite $(i, l, j, k, i, l, j, k, \ldots)$. Plus précisément, si le cycle C est $v_1, v_2, \ldots, v_{2k}, v_1$, alors une arête $\{v_{2r}, v_{2r+1}\}$ est étiquetée k en v_{2r} et i en v_{2r+1}. L'arête $\{v_{2r+1}, v_{2r+2}\}$ est étiquetée l en v_{2r+1} et j en v_{2r+2}. Pour cet étiquetage, l'automate entrant dans le sommet v_1 du cycle dans l'état s_0 par le port d'entrée initial i explore uniquement C et donc ne réussit pas à explorer G.

Cas 3 : $\delta_1(j) = (s_1, -)$ pour tous les numéros de port j. Pour tout i, notons $\lambda_0(i)$, resp. $\lambda_1(i)$, l'entier vérifiant $\delta_0(i) = (s, \lambda_0(i))$, resp. $\delta_1(i) = (s, \lambda_1(i))$, pour un certain s. Il y a deux sous-cas.

> **Cas 3.a :** $\lambda_0(\{0, \ldots, d-1\}) \cap \lambda_1(\{0, \ldots, d-1\}) = \emptyset$
> D'où λ_1 n'est pas bijective et donc pas injective. Par conséquent, il existe deux numéros de port j et j' tels que $\lambda_1(j) = \lambda_1(j') = k$. L'un des deux ($j$ ou j') est différent de k. Sans perte de généralité, supposons que $j' \neq k$. Si $j \neq k$, alors soit C un cycle de taille au plus $n-2$ dans G, dont l'existence est prouvée par le lemme 4.4. Etiquetons tous les ports du cycle C par la suite $(j, k, j, k, \ldots)$. Si $j = k$, soit C une arête de G. Etiquetons les deux ports de cette arête par k. Soit v un sommet de C voisin d'un sommet u hors de C. Un tel sommet existe puisque G est connexe.

Etiquetons les ports de l'arête $e = \{u, v\}$ par j' en v et par $\lambda_0(i)$ en u, pour un certain numéro de port i. Pour cet étiquetage, l'automate entrant dans le sommet u par le port d'entrée initial i dans l'état s_0 va en v et explore perpétuellement C. Donc l'automate visite u et tous les sommets de C. Puisque la taille de C est au plus $n-2$, l'automate ne réussit pas à explorer G.

Cas 3.b : $\lambda_0(\{0,\ldots,d-1\}) \cap \lambda_1(\{0,\ldots,d-1\}) \neq \emptyset$
Donc il existe des numéros de port i et j tels que $\lambda_0(i) = \lambda_1(j) = k$. Si $k = j$, nous considérons une arête $e = \{u, v\}$ dont les deux ports sont étiquetés j. L'automate entrant dans u par le port d'entrée initial i dans l'état s_0 visite seulement u et v. Si $k \neq j$ alors soit C un cycle non-hamiltonien dans G. Etiquetons tous les ports du cycle C par la suite $(j, k, j, k, \ldots)$. Pour cet étiquetage, l'automate entrant dans tout sommet du cycle par le port d'entrée intitial i explore uniquement C et donc ne réussit pas à explorer G.

Donc pour tout graphe régulier G qui n'est ni un cycle ni une clique, et pour tout automate $\mathcal{A}$ à deux états, nous avons construit une orientation locale, un sommet de départ et un port d'entrée initial tels que $\mathcal{A}$ ne visite pas tous les sommets de G. Donc $G \notin \mathcal{G}_2$. □

Les propositions 4.1 et 4.2 impliquent que la classe $\mathcal{G}_1$ est strictement incluse dans $\mathcal{G}_2$. Pour prouver l'inclusion stricte de $\mathcal{G}_2$ dans $\mathcal{G}_3$, il est suffisant de décrire un graphe de $\mathcal{G}_3$ qui n'est ni un cycle ni une clique.

Proposition 4.3 *La classe $\mathcal{G}_3$ contient un graphe qui n'est ni un cycle ni une clique.*

Preuve. Considérons le graphe G à 6 sommets consistant en deux cycles disjoints de longueur 3 dont les sommets sont connectés par des arêtes formant un couplage parfait (cf. Figure 4.3). Nous allons montrer que G peut être exploré par un automate à 3 états. Plus précisément, nous montrons que l'exploration peut être effectuée par l'automate suivant $\mathcal{A}$:

- Ensemble d'états $\{U, D, B\}$, où D est l'état initial ;
- Fonction de transition/sortie $\delta(D, i, 3) = (B, i+1)$, $\delta(B, i, 3) = (U, i)$, $\delta(U, i, 3) = (D, i+1)$, pour tout i.

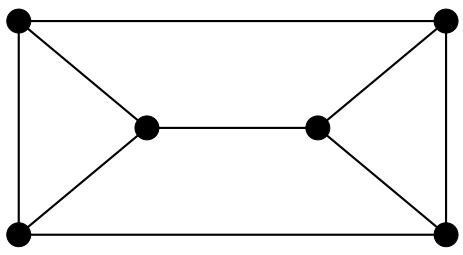

FIG. 4.3 – Un graphe dans $\mathcal{G}_3 \setminus \mathcal{G}_2$

Notons d'abord que si $\mathcal{A}$ visite un sommet u dans l'état D pendant l'exploration (en excluant la première étape), alors l'automate visite le sommet u, et chacun de ses trois

voisins, au moins une fois, pour toute orientation locale. L'un des voisins de u est visité juste avant d'entrer dans u dans l'état D, et les deux autres voisins sont visités durant les trois étapes suivantes.

Considérons le comportement de $\mathcal{A}$ dans G. A l'étape 4, $\mathcal{A}$ entre dans un certain sommet u dans l'état D. Après trois étapes supplémentaires, $\mathcal{A}$ entre dans un autre sommet v dans l'état D. Après de nouveau trois étapes supplémentaires, $\mathcal{A}$ entre dans un sommet w dans l'état D. Le sommet w est différent de v. Il est aussi différent de u car les arêtes $\{u, v\}$ et $\{v, w\}$ sont différentes et il n'y pas d'arêtes multiples dans G. Par conséquent, trois sommets de G différents sont visités par $\mathcal{A}$ dans l'état D pendant l'exploration. Ces trois sommets et tous leurs voisins sont donc visités au moins une fois par l'automate. Puisque tout ensemble de trois sommets de G est un ensemble dominant, tous les sommets de G sont visités par $\mathcal{A}$, quelles que soient l'orientation locale et la position de départ. □

Les propositions 4.1, 4.2 et 4.3 donnent le corollaire suivant.

Corollaire 4.1 *Les deux inclusions $\mathcal{G}_1 \subset \mathcal{G}_2 \subset \mathcal{G}_3$ sont strictes.*

4.4 Perspectives

Nous avons montré qu'une augmentation polynomiale du nombre d'états d'un automate entraîne une augmentation de sa capacité d'exploration. Le principal problème ouvert est de prouver ou de réfuter notre conjecture qui affirme que *toute* augmentation du nombre d'états permet à un automate d'explorer plus de graphes, c'est-à-dire que la hiérarchie $\mathcal{G}_K$ est stricte. Une autre question intéressante est de trouver la complexité du problème de décision suivant : "$G \in \mathcal{G}_K$?" c'est-à-dire "Etant donné un graphe G et un entier K, est-ce que G peut être exploré par un automate ayant au plus K états ?"

Deuxième partie

Exploration avec assistance

Contenu de la partie

Pour de nombreux problèmes distribués (tels que l'élection de leader, la construction d'un arbre couvrant de poids minimum, le rendez-vous entre agents mobiles, le réveil d'un ensemble de nœuds/d'agents, la diffusion d'un message, etc.), la qualité des solutions algorithmiques dépend souvent de la quantité d'informations sur la topologie donnée aux nœuds du réseau, ou donnée aux entités mobiles se déplaçant dans le réseau. Les connaissances *locales* données à chaque nœud et/ou à chaque entité mobile sont son identité et, pour un nœud, son degré (ou la liste des identités de ses voisins). Toute autre connaissance (par exemple, le nombre total de nœuds, le diamètre du réseau, le nombre total d'entités mobiles, une carte partielle du réseau, etc.) est une connaissance *globale*. De nombreux résultats illustrent l'impact des connaissances globales sur l'existence et l'efficacité de solutions à des problèmes de réseau. Par exemple, il est montré dans [BFR+02] que, si une borne supérieure $\hat{n}$ sur le nombre n de sommets d'un graphe orienté est connue, alors un robot peux explorer ce graphe en temps polynomial en $\hat{n}$, en utilisant un caillou, tandis que sans cette information, $\Theta(\log \log n)$ cailloux sont nécessaires et suffisants. La diffusion dans les réseaux radio est un autre problème où les connaissances globales influencent significativement l'efficacité des solutions. Dans [GPX05], il est prouvé que si les nœuds ont la connaissance complète du réseau, alors une diffusion déterministe peut être effectuée en temps $O(D + \log^3 n)$, pour les réseaux radio de n nœuds et de diamètre D. (Ce résultat a été récemment amélioré à $O(D + \log^2 n)$ dans [KP]). D'un autre côté, dans [CMS03], une borne inférieure de $\Omega(n \log D)$ est prouvée sur le temps de diffusion déterministe dans les réseaux radio dont les nœuds connaissent uniquement leur identité. (Une borne supérieure $O(n \log^2 D)$ est prouvée dans [CR03]). En fait, l'impact des connaissances globales est significatif pour de nombreux problèmes de calcul distribué, comme en témoigne [FR03, Lyn89], où des centaines de résultats d'impossibilité et de bornes inférieures pour le calcul distribué sont étudiés, beaucoup d'entre eux dépendant de si les nœuds ont connaissance ou non de valeurs approchées ou exactes de certains paramètres globaux. Finalement, notons que la quantité de connaissances globales a aussi un impact fort sur le calcul dans les réseaux anonymes (voir, par exemple, [YK96], où l'impact de la connaissance du nombre total de nœuds est étudié en profondeur).

Nous modélisons les connaissances globales données aux nœuds ou aux entités mobiles par un *oracle*. Etant donné un problème $\mathcal{P}$ et un ensemble d'instances $\mathcal{I}$, un oracle est une fonction $\mathcal{O} : \mathcal{I} \to \{0,1\}^*$ qui fait correspondre à tout instance I une chaîne binaire $\mathcal{O}(I)$. Résoudre le problème $\mathcal{P}$ en utilisant l'oracle $\mathcal{O}$ consiste à concevoir un algorithme qui, connaissant la chaîne binaire $\mathcal{O}(I)$ mais ignorant I, retourne un *$\mathcal{P}$-schéma* pour I, c'est-à-dire une séquence d'instructions exécutée par les nœuds ou les entités mobiles résolvant $\mathcal{P}$ pour I. Dans ce cadre, la quantité de connaissances globales est mesurée par la *taille* de l'oracle sur chaque instance I, c'est-à-dire par la longueur de la chaîne binaire $\mathcal{O}(I)$. Des questions intéressantes sont alors typiquement : "Quelle est la taille minimum d'un oracle pour résoudre le problème $\mathcal{P}$?" ou "Quelle est la taille minimum d'un oracle pour résoudre le problème $\mathcal{P}$ en un temps donné ?". La nouveauté et l'importance de notre modélisation des connaissances globales est que cette modélisation permet de poser

de telles questions *quantitatives* sur les connaissances requises, sans s'occuper de quelle *sorte* sont les informations fournies. Ceci doit être comparé à l'approche traditionnelle qui suppose la disponibilité de connaissances globales particulières.

Dans cette partie, nous illustrons ce concept d'oracle, utilisé comme mesure de complexité des problèmes, par le problème de l'exploration de graphes. Dans le chapitre 5, l'oracle fournit une chaîne binaire au robot, et nous nous intéressons à la taille minimum d'un oracle permettant la conception d'un algorithme d'exploration ayant un rapport compétitif strictement plus petit que 2 pour les arbres. Dans le chapitre 6, l'oracle fournit une chaîne binaire à chaque sommet du graphe exploré, sous la forme d'une couleur, et nous nous intéressons au nombre minimum de couleurs nécessaire à l'existence d'un automate fini capable d'explorer tous les graphes ainsi colorés. Enfin, dans le chapitre 7, nous supposons un type d'assistance plus fort encore en considérant que l'orientation locale est choisie pour aider un automate fini à réaliser l'exploration. Dans ce cadre, nous cherchons à minimiser le temps d'exploration.

Chapitre 5

Exploration d'arbres avec un oracle

Dans ce chapitre, nous étudions l'exploration d'arbres par un robot lorsque celui-ci est aidé par un oracle. Un robot (sans limite de mémoire) doit visiter tous les sommets d'un arbre inconnu le plus rapidement possible. La qualité d'un algorithme d'exploration $\mathcal{A}$ est mesurée par son *rapport compétitif*, c'est-à-dire en comparant son coût (en nombre de traversées d'arêtes) avec la longueur du plus court chemin contenant tous les sommets de l'arbre. Un parcours en profondeur d'abord a un rapport compétitif 2 et, en l'absence d'informations sur l'arbre, aucun algorithme ne peut faire mieux. En utilisant le formalisme de l'oracle, nous déterminons le nombre minimal de bits d'information qui doivent être fournis à un algorithme d'exploration exécuté par un robot pour que celui-ci ait un rapport compétitif strictement inférieur à 2.

5.1 Introduction

L'exploration de graphes étiquetés est le plus souvent effectuée quand le robot manque d'informations essentielles sur le graphe. Dans un tel cas, la qualité d'un algorithme d'exploration $\mathcal{A}$ peut être mesurée en comparant son coût (nombre de traversées d'arêtes) à la longueur de la plus courte *marche couvrante* (c'est-à-dire le plus court chemin contenant tous les sommets du graphe). Le rapport de ces deux nombres, maximisé sur tous les graphes et tous les sommets de départ, est appelé le *rapport compétitif* $\mathcal{R}(\mathcal{A})$ de l'algorithme $\mathcal{A}$. La situation ici est similaire au contexte des algorithmes à la volée, où la notion de rapport compétitif est apparu en premier. Dans les deux cas, la performance d'un algorithme manquant d'informations importantes sur l'environnement est comparée à celle d'un algorithme qui possède cette connaissance : dans le cas des algorithmes à la volée, cette connaissance concerne les événements futurs, et dans le cas de l'exploration, elle concerne la topologie du graphe et l'étiquetage des arêtes. (Un algorithme possédant une copie complètement étiquetée du graphe exploré, montrant quel port d'un sommet visité mène à quel voisin, peut pré-calculer la plus courte marche couvrante puis suivre cette marche pour effectuer l'exploration.)

Un parcours en profondeur d'abord (DFS) a un rapport compétitif 2 et il a été montré dans [DP04] qu'aucun algorithme d'exploration ne peut battre cette valeur pour les

graphes arbitraires, même lorsqu'on fournit à l'algorithme une copie isomorphe anonyme du graphe exploré, avec le sommet de départ marqué, mais sans l'orientation locale. L'absence des étiquettes de ports et de sommets sur la carte suffit à perturber tout algorithme sur certains graphes, les rendant aussi peu efficaces que le DFS. Par ailleurs, en l'absence de toute information globale, battre le rapport compétitif 2 a été montré impossible même pour la famille des arbres. Cela amène à la question de savoir s'il est possible d'obtenir un rapport compétitif plus petit que 2 si on fournit à l'algorithme quelques informations partielles concernant l'environnement exploré. Dans [DP04], une réponse positive à cette question est donnée dans le cas d'une très grande quantité d'informations : le robot dispose d'une carte sans numéros de port de l'arbre. Cependant, cette hypothèse n'est pas très réaliste. En effet, l'exploration est souvent utilisée comme un outil pour construire une carte du réseau inconnu, et habituellement les informations connues a priori sur le réseau exploré sont bien plus restreintes.

5.1.1 Le problème

Nous considérons le problème de la *quantité d'information* nécessaire pour effectuer l'exploration d'arbres avec un rapport compétitif plus petit que 2. (Rappelons que la raison pour laquelle nous nous intéressons seulement aux arbres est le résultat négatif sur les graphes arbitraires mentionné plus haut.)

Le problème est formalisé de la façon suivante. Dans le cadre de l'exploration d'arbres, nous définissons un *oracle* comme étant une fonction $\mathcal{O}$ de la classe des arbres dans la classe des chaînes binaires. Précisément, pour tout arbre T, on fournit à un algorithme d'exploration la chaîne $\mathcal{O}(T)$. Ensuite, cet algorithme retourne un *schéma d'exploration* pour T. Un tel schéma, débutant en un sommet quelconque u, traverse toutes les arêtes de T. La taille de l'oracle pour l'arbre T est la longueur de la chaîne $\mathcal{O}(T)$. Nous demandons quelle est la taille minimum d'un oracle pour lequel il existe un algorithme d'exploration ayant un rapport compétitif plus petit que 2, pour tous les arbres.

5.1.2 Résultats

Notre principal résultat dans ce chapitre établit un seuil exact sur la taille de l'oracle pour obtenir un rapport compétitif plus petit que 2. Ce seuil est approximativement $\log \log D$, où D est le diamètre de l'arbre. Plus précisément, nous construisons, pour toute constante c, un algorithme d'exploration de rapport compétitif plus petit que 2, utilisant un oracle de taille au plus $\log \log D - c$, et nous montrons que tout algorithme utilisant un oracle de taille $\log \log D - g(D)$, avec g une fonction quelconque non bornée supérieurement, a un rapport compétitif au moins égal à 2.

Il est intéressant de noter la *structure* de l'oracle dans notre résultat positif. Pour tout arbre T, l'oracle donne une chaîne de bits s dépendant seulement de D, qui en est une approximation, plus un bit supplémentaire b qui permet au robot de choisir entre deux types d'exploration. Ce bit supplémentaire b (qui dépend de D et de la taille de l'arbre) est très important. En effet, alors que la chaîne s dépend seulement de D et est de longueur

plus petite que $\log \log D$, nous montrons que même la pleine connaissance de D, mais sans le bit b, n'est pas suffisante pour battre le rapport compétitif 2. Plus précisément, nous montrons que tout algorithme d'exploration connaissant seulement le diamètre de l'arbre a nécessairement un rapport compétitif au moins égal à 2.

Ces résultats sont issus d'une collaboration avec Pierre Fraigniaud et Andrez Pelc et sont en cours de soumission à une conférence internationale.

5.2 Terminologie et préliminaires

Pour tout arbre T, nous notons $|T|$ le nombre de sommets de T, et nous l'appelons la *taille* de cet arbre. Pour un arbre donné T et un sommet de départ u, on note $opt(T, u)$ la longueur de la plus petite marche couvrant T commençant en u, c'est-à-dire la longueur du plus court chemin dans T commençant en u et contenant toutes les arêtes de T. Clairement $opt(T, u) = 2(n-1) - ecc(u)$, où n est la taille de T et $ecc(u)$ est l'excentricité du sommet de départ u, c'est-à-dire la distance de u à la feuille la plus éloignée. Un parcours en profondeur d'abord finissant sur la feuille la plus éloignée de u utilise ce nombre minimal de traversées d'arêtes.

Nous supposons l'arbre muni d'une orientation locale. Le robot ayant une mémoire non bornée, il peut stocker tous les numéros de port rencontrés et peut donc reconnaître les sommets déjà visités et les arêtes déjà traversées, car nous nous restreignons au cas des arbres. Cependant, il ne peut pas faire de différences entre des arêtes incidentes à sa position qui ne sont pas encore explorées. Le robot exécute un *schéma d'exploration* donné qui, en chaque sommet, prend l'une des décisions suivantes : prendre une arête spécifique déjà explorée, ou prendre une arête non explorée. Si le schéma décide de prendre une arête non explorée, le choix effectif de l'arête appartient à un adversaire, car nous nous intéressons à la performance dans le pire des cas.

Nous voulons que l'oracle fournisse des informations sur la topologie de l'arbre exploré, indépendamment de toute orientation locale. Nous le définissons donc comme une fonction $\mathcal{O}$ de la classe de tous les arbres *non étiquetés* dans la classe des chaînes binaires. Pour toute chaîne s, un arbre T tel que $\mathcal{O}(T) = s$ est appelé *compatible* avec s. Si un algorithme d'exploration d'arbres $\mathcal{A}$ prend la chaîne $\mathcal{O}(T)$ comme entrée pour tout arbre T, on dit que $\mathcal{A}$ *utilise* $\mathcal{O}$.

Considérons un algorithme d'exploration $\mathcal{A}$ utilisant l'oracle $\mathcal{O}$. Pour toute chaîne s atteignable par $\mathcal{O}$, l'algorithme $\mathcal{A}$ produit un schéma d'exploration qui explore tous les arbres compatibles avec s. Pour tout arbre T et tout sommet de départ u, le *coût* $\mathcal{A}(T, u)$ de ce schéma, exécuté sur l'arbre T depuis le sommet de départ u, est le nombre de traversées d'arêtes dans le pire cas, pris sur tous les choix de l'adversaire mentionnés plus haut. Le rapport compétitif de $\mathcal{A}$ est défini comme le rapport

$$\mathcal{R}(\mathcal{A}) = \sup_{T,u} \frac{\mathcal{A}(T, u)}{opt(T, u)} \ ,$$

où l'extremum est pris sur tous les arbres T et tous les sommets de départ u de T.

Le fait qu'un oracle soit défini sur des arbres non étiquetés plutôt que munis d'une orientation locale est une distinction importante. Par exemple, pour la classe des lignes, nous prouverons qu'un oracle de taille (asymptotique) $\log \log n$ est nécessaire pour obtenir un rapport compétitif plus petit que 2, où n est la longueur de la ligne. Cependant, pour une orientation locale *donnée*, un seul bit (indiquant le port du sommet de départ menant à l'extrémité de la ligne la plus proche) est suffisant pour obtenir le rapport compétitif 1 : un DFS débutant en direction de l'extrémité la plus proche l'atteint.

La remarque suivante sera utile pour prouver des bornes inférieures sur le rapport compétitif des algorithmes d'exploration. Supposons que le robot, à un certain moment de l'exploration, est sur un sommet v, puis se déplace le long d'une arête e déjà explorée incidente à v, et retourne immédiatement en v. Pour n'importe quel ensemble de décisions de l'adversaire, un algorithme effectuant de tels mouvements, lorsqu'il est lancé sur un arbre T depuis un sommet de départ u, a un coût strictement plus grand que l'algorithme qui saute ces deux déplacements. Nous nous restreignons donc aux algorithmes d'exploration qui n'effectuent jamais de tels retours. Nous les appelons *réguliers*.

Dans [DP04], les auteurs introduisent la classification suivante des algorithmes d'exploration pour la classe des lignes (ils considèrent les algorithmes d'exploration connaissant la longueur n des lignes). Fixons n et appelons *de type* k l'ensemble des algorithmes qui effectuent toujours au plus k demi-tours avant d'atteindre une extrémité, et qui font exactement ce nombre de demi-tours pour une certaine combinaison du sommet de départ et du choix (de l'adversaire) de la direction initiale. Ils prouvent le résultat suivant qui permet de restreindre l'attention sur des algorithmes explorant les lignes relativement simples, lorsque l'on s'intéresse au rapport compétitif minimum.

Lemme 5.1 [DP04] *Fixons $n \geq 11$. Pour tout algorithme d'exploration $\mathcal{A}$ pour la ligne L_n de longueur n, il existe un algorithme $\mathcal{A}'$ pour L_n, tel que $\mathcal{A}'$ est de type 1 et*

$$\max_{u \in L_n} \frac{\mathcal{A}'(L_n, u)}{opt(L_n, u)} \leq \max_{u \in L_n} \frac{\mathcal{A}(L_n, u)}{opt(L_n, u)}.$$

Dans notre cas, un algorithme ne connaît pas la longueur de la ligne mais seulement la valeur de l'oracle pour cette ligne. Nous modifions donc la notion de type de la façon suivante. Considérons un algorithme $\mathcal{A}$ utilisant l'oracle $\mathcal{O}$. Fixons une chaîne s atteignable par $\mathcal{O}$ et considérons le schéma d'exploration produit par $\mathcal{A}$ pour cette chaîne. Le schéma est de type k s'il effectue toujours au plus k demi-tours avant d'atteindre une extrémité, pour toute ligne L_n de longueur n compatible avec s et tout sommet de départ u, et s'il fait exactement ce nombre de demi-tours pour une certaine ligne compatible avec s, un certain sommet de départ et un certain choix de l'adversaire de la direction initiale.

Dans la preuve du lemme 5.1, l'algorithme $\mathcal{A}'$ est obtenu à partir de $\mathcal{A}$ indépendamment de n. Donc ce lemme implique que, dans notre cas, le meilleur rapport compétitif pour la classe des lignes est obtenu par un algorithme d'exploration qui, pour toute chaîne s, produit un schéma de type 1. Ce type consiste en de simples schémas d'exploration qui vont x pas dans une direction (à moins qu'une extrémité soit atteinte), puis font demi-tour et vont jusqu'à une extrémité, et finalement font demi-tour pour aller jusqu'à l'autre

extrémité. Pour tout schéma de type 1, l'entier x sera appelé la *distance d'investigation* du schéma.

Le lemme suivant décrit la performance des schémas de type 1 comme une fonction de la distance d'investigation.

Lemme 5.2 *Pour tout entier strictement positif n et pour tout $\alpha < 1$, soit $S_{\alpha,n}$ le schéma d'exploration de type 1 pour la ligne L_n de longueur n, de distance d'investigation $\lfloor \alpha n \rfloor$, et soit $t_{\alpha,n}(u)$ le coût de ce schéma pour le sommet de départ u. Enfin, soit $F_n(\alpha) = \max_{u \in L_n} \frac{t_{\alpha,n}(u)}{opt(L_n,u)}$. Alors il existe un entier strictement positif N_0, tel que pour tout $n \geq N_0$, la fonction F_n est strictement décroissante sur l'intervalle $]0, \frac{\sqrt{3}-1}{2}]$, et $\sup_{n>0} F_n(\alpha) < 2$, pour tout α dans cet intervalle.*

Preuve. Fixons un sommet de départ u dans L_n. Soit a la distance de u à l'extrémité de la ligne vers laquelle se dirige le robot au tout début de l'exploration, et soit $b = n - a$. Nous pouvons supposer $a, b > 0$, sinon $t_{\alpha,n}(u) = opt(L_n, u)$. Soit $x = \lfloor \alpha n \rfloor$. If $a \leq x$ alors, puisque $\alpha \leq \frac{\sqrt{3}-1}{2} < 1/2$, nous avons $t_{\alpha,n}(u) = opt(L_n, u)$. Donc supposons que $a > x$. Si $a \leq b$, alors $\frac{t_{\alpha,n}(u)}{opt(L_n,u)} = \frac{2x+b+n}{2n-b}$, qui est maximal pour $b = n-x-1$ et, dans ce cas, est égal à $\frac{2n+x-1}{n+x+1}$. Si $a > b$, alors $\frac{t_{\alpha,n}(u)}{opt(L_n,u)} = \frac{n+2x+b}{n+b}$, qui est maximal pour $b = 1$ et, dans ce cas, est égal à $\frac{n+2x+1}{n+1}$. Pour $\alpha \leq \frac{\sqrt{3}-1}{2}$ et pour n suffisamment grand, on a $\frac{2n+x-1}{n+x+1} \geq \frac{n+2x+1}{n+1}$, et donc $F_n(\alpha) = \frac{2n+x-1}{n+x+1}$. Cette fraction est une fonction décroissante de x, donc F_n est une fonction décroissante de α dans l'intervalle $]0, \frac{\sqrt{3}-1}{2}]$. Pour la seconde partie, notons que, pour n suffisamment grand, nous avons $F_n(\alpha) = \frac{2n+x-1}{n+x+1} \leq \frac{2+\alpha}{1+\alpha}$, qui est plus petit que 2 pour $\alpha > 0$. □

5.3 La borne supérieure

Dans cette section et la suivante, nous prouvons le résultat principal de ce chapitre. Nous établissons le seuil exact sur la taille de l'oracle pour laquelle un algorithme d'exploration peut avoir un rapport compétitif plus petit que 2. Ce résultat est présenté sous la forme de deux théorèmes. Le premier établit une borne supérieure sur la taille d'un tel oracle, en construisant un algorithme d'exploration approprié. Le second, dans la section 5.4, prouve une borne inférieure qui coïncide avec la borne supérieure. Dans cette section, nous construisons l'algorithme d'exploration `SKE`(c) (pour SMALL-KNOWLEDGE-EXPLORATION(c)), pour un entier arbitraire strictement positif c. Cet algorithme a un rapport compétitif plus petit que 2 et utilise un oracle $\mathcal{O}_c$ de taille au plus $\max(1, \log\log D - c)$, pour un arbre quelconque de diamètre D.

Nous décrivons d'abord l'oracle $\mathcal{O}_c$. Fixons $c \in \mathbb{N}^*$. Etant donné un arbre T de diamètre D, l'oracle $\mathcal{O}_c$ calcule un bit appelé `choice` et, si `choice` $= 1$, un entier k utilisant $\lceil \log \lceil \log D \rceil \rceil - (c+3)$ bits. Le bit `choice` est utilisé par l'algorithme pour prendre une décision concernant deux méthodes alternatives d'exploration, et l'entier k est utilisé pour obtenir une approximation D_0 du diamètre.

Soit N_0 un entier (dont l'existence est garantie par le Lemme 5.2) tel que, pour tout $n \geq N_0$, la fonction F_n est strictement décroissante dans l'intervalle $]0, \frac{\sqrt{3}-1}{2}]$, et $\sup_{n>0} F_n(\alpha) < 2$, pour n'importe quel α dans cet intervalle. Pour $\alpha \in]0, \frac{\sqrt{3}-1}{2}]$, soit $\beta(\alpha) = \sup_{n>0} F_n(\alpha)$. Soit T un arbre quelconque fixé. Soit n son nombre de sommets et D son diamètre. Prenons ϵ tel que $D = (1-\epsilon)n$. Dans la suite, nous utiliserons les abréviations suivantes : $\lambda = \frac{\sqrt{3}-1}{2}$, et $\gamma = 2^{2^{c+3}+1}$. Nous définissons maintenant un seuil ϵ^* sur la valeur de ϵ, qui servira à définir le bit `choice`. Soit $\epsilon_1 = \frac{\lambda}{16\gamma}$, $\beta_1 = \beta(\epsilon_1)$, $\epsilon_2 = \frac{2-\beta_1}{624}$, et $\epsilon^* = \min(\epsilon_1, \epsilon_2)$. L'oracle assigne 1 à `choice` si

$$(\epsilon < \epsilon^*) \wedge (D \geq 2^{2^{c+3}}) \wedge (n \geq N_0),$$

et assigne 0 à `choice` sinon. Si `choice` $= 1$, l'oracle calcule $k = \lfloor \frac{\lceil \log D \rceil}{2^{c+3}} \rfloor$.

Etant donné `choice` et k, l'algorithme `SKE`(c) retourne un schéma d'exploration. Si `choice` $= 0$, le schéma est un parcours en profondeur d'abord (DFS) arbitraire. Pour fixer les idées, nous prenons le DFS qui choisit toujours à chaque sommet le plus petit numéro de port non encore utilisé. Notons que `choice` est mis à 0 quand le diamètre de l'arbre est significativement plus petit que sa taille, ou quand le diamètre est borné, ou quand l'arbre lui-même est petit.

Nous décrivons maintenant le schéma $\mathcal{X}_c$, beaucoup plus subtile, produit par l'algorithme quand `choice` $= 1$. Le schéma $\mathcal{X}_c$ utilise la procédure `DPDFS`(v) (pour DOUBLING-PARTIAL-DEPTH-FIRST-SEARCH(v)) qui est appelé à un sommet v de l'arbre. La procédure renvoie les deux arêtes qui connectent v aux deux plus grands sous-arbres enracinés aux voisins de v, explore complètement les autres sous-arbres, et retourne finalement en v. Dans la suite, nous utiliserons la notion de sous-arbre pendant de v comme équivalent de la notion de sous-arbre enraciné en un voisin de v. La procédure `DPDFS`(v) est décrite dans la figure 5.1.

Lemme 5.3 *Soit v un sommet quelconque de degré au moins 3. Soit $T_1, \ldots, T_p$ l'énumération des sous-arbres pendants de v dans l'ordre décroissant de leur taille. La procédure* `DPDFS`(v) *renvoie deux arêtes correspondant aux deux plus grands sous-arbres (à égalité de taille près), et explore complètement tous les autres sous-arbres pendants de v. De plus, le coût de la procédure* `DPDFS`(v) *est au plus $22 \sum_{i \geq 3}^{p} |T_i|$.*

Preuve. La première partie découle directement de la description de la procédure. Pour la seconde partie, soit $x_i = \lfloor \log |T_i| \rfloor$, pour $i = 1, \ldots, p$. On appelle phase j l'exécution du j-ième passage dans la boucle "tant que" externe. Puisque $2^{x_i} - 1 < |T_i| \leq 2^{x_i+1} - 1$, le sous-arbre T_i est complètement exploré à la fin de la phase $x_i + 1$, et pas avant. Donc la dernière phase exécutée est la phase $x_3 + 1$.

Nous commençons par borner le coût d'exploration de T_1. Durant la phase j, le DFS visite au plus 2^j sommets de ce sous-arbre. Ce DFS coûte au plus $2 \cdot 2^j$, incluant le retour en v. Pour la procédure complète, le coût d'exploration de T_1 est au plus $\sum_{j=1}^{x_3+1} 2 \cdot 2^j \leq 2 \cdot 2^{x_3+2} = 8 \cdot 2^{x_3} \leq 8|T_3|$. Cette estimation reste valable en ce qui concerne l'exploration de T_2. En résumé, le coût d'exploration de T_1 et de T_2 est au plus $16|T_3|$.

Procédure DPDFS(v)
$i \leftarrow 1$;
$S \leftarrow$ ensemble des arêtes incidentes à v connectant v à des sous-arbres non complètement explorés ;
tant que $|S| \geq 3$ **faire**
$S' \leftarrow S$;
tant que $S' \neq \emptyset$ **faire**
soit $e \in S'$ et soit $T(e)$ le sous-arbre connecté à v par l'arête e ;
explorer $T(e)$ par DFS jusqu'à ce que $\min(|T(e)|, 2^i - 1)$ sommets soient visités ;
retourner en v ;
$S' \leftarrow S' \setminus \{e\}$;
si $T(e)$ est complètement exploré **alors** $S \leftarrow S \setminus \{e\}$;
$i \leftarrow i + 1$;
si $|S| = 2$ **alors retourner** S ;
si $|S| = 1$ **alors** soit e' l'arête connectant v au plus grand sous-arbre exploré et **retourner** $S \cup \{e'\}$;
si $S = \emptyset$ **alors** soient e' et e'' les arêtes connectant v aux deux plus grands sous-arbres explorés et **retourner** $\{e', e''\}$;

FIG. 5.1 – Procédure DPDFS

Soit $3 \leq i \leq p$. Le coût d'exploration de T_i lors de la phase $x_i + 1$ est exactement $2|T_i|$. Le coût total d'exploration de T_i est donc au plus $2|T_i| + \sum_{j=1}^{x_i} 2 \cdot 2^j \leq 2|T_i| + 4 \cdot 2^{x_i} \leq 6|T_i|$. En conséquence, le coût total d'exploration de tous les sous-arbres T_i, pour $i \geq 3$, est au plus $6 \sum_{i=3}^{p} |T_i|$.

Puisqu'on a trivialement $|T_3| \leq \sum_{i \geq 3}^{p} |T_i|$, le coût total de la procédure DPDFS(v) est borné par $22 \sum_{i \geq 3}^{p} |T_i|$. □

L'idée intuitive du schéma d'exploration $\mathcal{X}_c$ (retourné par l'algorithme SKE(c) quand choice $= 1$) est la suivante. Soit $D_0 = 2^{k \cdot 2^{c+3} - 1}$. Nous prouverons que D_0 approche le diamètre D de la façon suivante : $D_0 \leq D < \gamma D_0$. Le robot utilise la procédure DPDFS(v) pour identifier les deux arêtes reliant le sommet courant v aux plus grands sous-arbres pendants de celui-ci. Le robot traverse ensuite une de ces arêtes et applique de nouveau la procédure. Ces applications consécutives définissent un chemin de longueur approximativement égale au diamètre de l'arbre. Sur ce chemin, le robot exécute un schéma de type 1 pour les lignes : aller à la distance d'investigation $\lfloor \lambda D_0 / 2 \rfloor$, faire demi-tour et aller au bout du chemin, faire demi-tour et aller à l'autre bout du chemin. L'approximation D_0 du diamètre est suffisamment fine pour garantir une bonne performance du schéma sur ce chemin. Par ailleurs, la partie de l'arbre disjointe du chemin est de taille négligeable (ceci est une conséquence des critères d'affectation de la valeur 1 à la variable choice). Tout cela implique que le rapport compétitif du schéma $\mathcal{X}_c$ est plus petit que 2.

Le schéma d'exploration $\mathcal{X}_c$ est décrit dans la figure 5.2. Dans la description, les

déplacements effectués durant les appels à la procédure `DPDFS` sont appelés *internes*, et tous les autres déplacements sont appellés *externes*. Pendant toute l'exploration, le robot mémorise les résultats de toutes les actions précédentes, et construit une carte de la portion de l'arbre qui a été explorée jusqu'ici.

Schéma d'exploration $\mathcal{X}_c$
  tant que l'exploration n'est pas terminée **faire**
    soit v le sommet courant ;
    {*Déplacements internes :*}
    si $\deg(v) \geq 3$ **alors**
      à moins que la procédure `DPDFS`(v) n'ait déjà été exécutée précédemment, l'exécuter pour obtenir les arêtes e, e' reliant v aux deux plus grands sous-arbres pendants de v ;
    {*Déplacement externe :*}
    si il y a seulement une arête reliant v à un sous-arbre non encore complètement exploré **alors**
      quitter v par cette arête ;
    sinon
      {e, e' *sont les deux arêtes reliant* v *aux deux sous-arbres non encore complètement explorés*}
      si lors du dernier déplacement externe (s'il existe), le robot n'est pas venu en v par e ou e' **alors**
        quitter v par l'arête e ;
      sinon
        supposons, sans perte de généralité, que le robot est venu en v par l'arête e ;
        si le robot est pour la première fois à distance $\lfloor \lambda D_0/2 \rfloor$ du sommet de départ **alors**
          quitter v par l'arête e ;
        sinon quitter v par l'arête e' ;
  fin tant que

FIG. 5.2 – Schéma d'exploration $\mathcal{X}_c$

Lemme 5.4 *L'algorithme* `SKE`(c) *est correct.*

Preuve. Si l'oracle affecte 0 à `choice`, l'algorithme `SKE`(c) retourne un DFS comme schéma d'exploration. Clairement le DFS visite tous les sommets de l'arbre. Supposons que l'oracle affecte à `choice` la valeur 1. Nous allons prouver que le schéma $\mathcal{X}_c$ visite tous les sommets de l'arbre. Supposons que ce n'est pas le cas. Puisqu'à chaque exécution de la boucle "tant que" dans $\mathcal{X}_c$ le robot vérifie (avant de terminer) si l'arbre est complètement exploré, notre hypothèse implique que le robot ne s'arrête jamais. Il existe donc un ensemble non vide S de sommets visités infiniment souvent. S est un sous-arbre de T. Soit

v une feuille de S, et soit v' son unique voisin dans S. Considérons un instant t de l'exploration où chaque sommet de S a été visité au moins deux foix, et aucun sommet de $T \setminus S$ ne sera plus visité.

Après l'instant t, le robot se déplace (infiniment souvent) de v à v', lors de déplacements externes. Le robot n'effectue jamais de déplacement externe vers un sous-arbre qui est complètement exploré. Donc le sous-arbre U de T pendant de v' et contenant v n'est pas complètement exploré. Lorsque le robot atteint v, il n'est pas pour la première fois à distance $\lfloor \frac{\lambda D_0}{2} \rfloor$ du sommet de départ car, à l'instant t, le sommet v a déjà été visité au moins deux fois. De plus, le sommet v ne peut pas être une feuille de T parce que le sous-arbre U n'est pas complètement exploré. Donc la seule raison pour laquelle le robot revient de v à v' est que le sous-arbre pendant de v et contenant v' est l'unique sous-arbre pendant de v qui ne soit pas complètement exploré. Cela implique que le sous-arbre U est entièrement exploré, une contradiction. De ce fait, tous les sommets de l'arbre sont explorés par le schéma $\mathcal{X}_c$, ce qui prouve la correction du schéma d'exploration. □

Théorème 5.1 *Soit c une constante entière arbitraire et strictement positive. L'algorithme* $\texttt{SKE}(c)$ *utilise un oracle de taille au plus* $\max(1, \log\log D - c)$*, pour tout arbre de diamètre D, et a un rapport compétitif plus petit que 2.*

Preuve. Le résultat est vrai pour $n = 1$ ou $n = 2$ car tout algorithme est optimal dans ce cas. Dans la suite, nous supposerons que $n \geq 3$.

Rappelons que $k = \lfloor \frac{\lceil \log D \rceil}{2^{c+3}} \rfloor$. L'oracle $\mathcal{O}_c$ utilise au plus $\lceil \log k \rceil + 1$ bits, donc au plus $\max(1, \log\log D - c)$ bits. La définition de k implique l'inégalité $k \leq \frac{\lceil \log D \rceil}{2^{c+3}} < k + 1$, donc $k \cdot 2^{c+3} \leq \lceil \log D \rceil < k \cdot 2^{c+3} + 2^{c+3}$, et finalement $2^{k \cdot 2^{c+3} - 1} \leq D < 2^{k \cdot 2^{c+3}} \cdot 2^{2^{c+3}}$. De la définition de D_0 et de γ, nous avons $D_0 \leq D < \gamma D_0$.

Supposons d'abord que l'oracle affecte 0 à `choice`. Puisque le schéma d'exploration dans ce cas est un DFS, le coût du schéma est au plus $2(n-1) - 1 = 2n - 3$. Le coût $opt(T, u)$, où u est le sommet de départ, est $2(n-1) - ecc(u)$. Nous avons $ecc(u) \leq D = (1-\epsilon)n$. On obtient

$$opt(T, u) \geq 2(n-1) - (1-\epsilon)n = (1+\epsilon)n - 2.$$

Puisque $D \leq n - 1$, nous avons $\epsilon \geq 1/n$, ou, de façon équivalente, $\epsilon n \geq 1$. Le rapport est donc dans ce cas au plus

$$\frac{2n-3}{(1+\epsilon)n-2} = \frac{2n-3}{(1+\epsilon/2)n - 2 + \epsilon n/2} \leq \frac{2n-3}{(1+\epsilon/2)n - 1.5} \leq \frac{2}{1+\epsilon/2}.$$

Si $\epsilon \geq \epsilon^*$, alors $\frac{2}{1+\epsilon/2} \leq \frac{2}{1+\epsilon^*/2} < 2$. Supposons que $\epsilon < \epsilon^*$. Soit $D^* = 2^{2^{c+3}}$. `choice` est égal à 0 parce que $D < D^*$ ou $n < N_0$. Si $D < D^*$, alors $n < \frac{D^*}{1-\epsilon}$. Soit $N_1 = \frac{D^*}{1-\epsilon^*}$. Alors nous avons $n < \frac{D^*}{1-\epsilon} \leq \frac{D^*}{1-\epsilon^*} = N_1$. Donc, à la fois quand $D < D^*$ et quand $n < N_0$, nous avons $n < N^* = \max(N_0, N_1)$. Soit $\epsilon_3 = 1/N^*$. Nous avons $\epsilon \geq 1/n \geq 1/N^* = \epsilon_3$. On obtient $\frac{2}{1+\epsilon/2} \leq \frac{2}{1+\epsilon_3/2} < 2$. Le rapport "coût du DFS" (retourné par l'algorithme $\texttt{SKE}(c)$ quand $\texttt{choice} = 0$) sur $opt(T, u)$, est au plus $\max(\frac{2}{1+\epsilon^*/2}, \frac{2}{1+\epsilon_3/2}) < 2$.

A partir de maintenant, nous supposons que l'oracle a fixé `choice` à 1. L'algorithme `SKE`(c) retourne donc le schéma d'exploration $\mathcal{X}_c$.

Dans l'analyse du coût du schéma d'exploration $\mathcal{X}_c$, nous utilisons la terminologie suivante. Supposons que le robot arrive sur un sommet quelconque v de degré au moins 3 et suit la procédure `DPDFS`(v). Si la procédure renvoie deux arêtes différentes de e, nous disons que le sommet courant v est un *embranchement*. Considérons maintenant les arêtes traversées lors de déplacements externes. Ces arêtes forment un sous-arbre T' de T. Pour tout sommet v, il existe au plus deux arêtes incidentes telles que tout déplacement externe du robot quittant v prenne l'une d'elles. En conséquence, tous les sommets sont de degré au plus 3 dans ce sous-arbre. Les sommets de degré exactement 3 dans T' sont les embranchements. Soit $v_1, \ldots, v_q$ les embranchements de T', s'il en existe, dans l'ordre de leur première visite par le robot. Soit e_i l'arête connectant v_i au sous-arbre pendant de ce sommet et contenant le sommet de départ u. Au regard de la définition d'un embranchement, le robot n'effectue jamais un déplacement externe le long de e_i depuis le sommet v_i. Soit u' le dernier embranchement v_q, s'il existe, ou $u' = u$ si T' ne contient aucun embranchement. Ensuite, soit P un chemin de longueur D dans l'arbre et P' l'ensemble des sommets de T' visité par un déplacement externe après u'. P' est un chemin car il ne contient aucun embranchement autre qu'éventuellement u'. Finalement, soit C la longueur de la marche la plus petite couvrant le chemin P' commençant au sommet u'.

Dans notre analyse, le coût du schéma $\mathcal{X}_c$ est divisé en deux parties : le coût des déplacement internes, et le coût des déplacement externes.

Assertion 1. Le nombre total de déplacements internes est au plus $154\epsilon n$.

Pour prouver l'assertion 1, soit v un sommet quelconque de degré au moins 3 dans T. Soit $T_1, \ldots, T_p$ l'énumération des sous-arbres pendant de v, dans l'ordre décroissant de leur taille. Puisque $D = (1-\epsilon)n$, il y a au plus ϵn sommets dans $T \setminus P$. Cela implique $\sum_{i\geq 3}^{p} |T_i| \leq \epsilon n$. Donc `DPDFS`$(v)$ ne va jamais à distance plus que $2\epsilon n$ du sommet v. Un sommet de T est soit exploré lors d'un appel à `DPDFS`, soit il est dans T'. Donc tout sommet de T est à distance au plus $2\epsilon n$ d'un sommet de T'. En particulier, il existe deux sommets w, w' de T' tels que w est à distance au plus $2\epsilon n$ d'une extrémité de P, et w' est à distance au plus $2\epsilon n$ de l'autre extrémité de P. La distance entre w et w' est au moins $D - 4\epsilon n = (1-5\epsilon)n$. En conséquence, $|T'| \geq (1-5\epsilon)n$.

D'après le lemme 5.3, le coût de `DPDFS`(v) dépend de la somme des tailles des sous-arbres pendant de v, excepté les deux plus grands. Ces sous-arbres sont qualifiés de petits sous-arbres pendant de v. `DPDFS` est appelé exactement aux sommets de T' ayant un degré au moins 3 dans T. Pour tout sommet $v \in T'$ qui n'est pas un embranchement, les petits sous-arbres de v sont tous disjoints de T'. Pour un embranchement v_i, seulement un petit sous-arbre contient un sommet de T' : le sous-arbre pendant de v_i et contenant u. En résumé, tous les petits sous-arbres pendant de sommets de T', excepté les q sous-arbres pendant des embranchements et contenant u, sont disjoints de T' et aussi disjoints les uns des autres. La taille totale de ces sous-arbres est au plus $|T \setminus T'| \leq 5\epsilon n$.

Supposons que $T' \setminus P' \neq \emptyset$. Soit n_i, pour $i = 1, \ldots, q$, la taille du sous-arbre T_i pendant de v_i et contenant u. Fixons $i < q$. Puisque v_i est un embranchement, il existe

deux sous-arbres, pendant de v_i et ne contenant pas u, qui ont au moins n_i sommets chacun. L'un deux, appelé T'_i, ne contient pas le prochain embranchement v_{i+1}. T_i et T'_i sont tous deux inclus dans le sous-arbre T_{i+1}. Ceci implique que $n_{i+1} \geq 2n_i$. La somme des tailles des q sous-arbres pendant des embranchements et contenant u satisfait donc $\sum_{i=1}^{q} n_i \leq 2n_q$. Comme l'arête e_q connectant P' avec $T' \setminus P'$ n'est pas sélectionnée par `DPDFS` à l'embranchement $u' = v_q$, nous avons $|T' \setminus P'| \leq \epsilon n$. Donc, puisque $n_q = |T' \setminus P'|$, nous avons $\sum_{i=1}^{q} n_i \leq 2\epsilon n$.

Par conséquent, d'après le lemme 5.3, le nombre total de déplacements internes est au plus $22(5\epsilon n + 2\epsilon n) = 154\epsilon n$. Ceci conclut la preuve de l'assertion 1.

Assertion 2. Le nombre total de déplacements externes est au plus $\beta_1 C + 2\epsilon n$.

Pour prouver l'assertion 2, nous étudions séparément les déplacements externes dans P' et dans $T' \setminus P'$. Considérons d'abord les déplacements externes dans P'. Au moins un des deux sommets w, w' (définis dans le preuve de l'assertion 1) est dans P' parce que $(1 - 5\epsilon)n > \epsilon n$, puisque $\epsilon < \epsilon_1 < 1/8$. Donc, comme $|T' \setminus P'| \leq \epsilon n$, le chemin P' est de longueur au moins $(1 - 6\epsilon)n$.

Puisque $D \geq 2^{2^{c+3}}$, nous avons $D_0 \geq 16$. Couplé à $\epsilon < 1/2$, cela implique que

$$2\epsilon n < 2\epsilon_1 n = \frac{\lambda}{8\gamma} n = \frac{\lambda}{8\gamma} \frac{D}{1-\epsilon} \leq \frac{\lambda}{8} \frac{D_0}{1-\epsilon} \leq \lambda D_0/4 \leq \lambda D_0/2 - 1 \leq \lfloor \lambda D_0/2 \rfloor \ .$$

Donc lorsque le robot est pour la première fois à distance $\lfloor \lambda D_0/2 \rfloor$ du sommet de départ, il est nécessairement en un sommet de P'. De plus, ce sommet est à distance au moins $\epsilon_1 |P'|$ de u' parce que $\epsilon_1 |P'| \leq \epsilon_1 D < \epsilon_1 n \leq 2\epsilon_1 n - \epsilon n$. Il est aussi à distance au plus $\lfloor \lambda |P'| \rfloor$ de u' parce que $|P'| \geq (1 - 6\epsilon)n$ et $\epsilon < 1/16$ impliquent $|P'| \geq n/2 > D_0/2$. En résumé, dans le chemin P', le robot va à distance d'investigation x (depuis u') telle que $\lfloor \epsilon_1 |P'| \rfloor \leq x \leq \lfloor \lambda |P'| \rfloor$, ensuite fait demi-tour et va jusqu'à une extrémité de P', et finalement refait demi-tour pour aller à l'autre extrémité de P'. Il existe α vérifiant $0 < \epsilon_1 \leq \alpha \leq \lambda$, tel que $x = \lfloor \alpha |P'| \rfloor$. Rappelons que C est le coût optimal d'exploration de la ligne P' en débutant au sommet u' de cette ligne. D'après le lemme 5.2, le nombre de déplacements externes du robot dans P' est au plus $F_n(\alpha) \cdot C$. D'après le même lemme et étant donné le fait que $n \geq N_0$, nous avons $F_n(\alpha) \leq F_n(\epsilon_1) \leq \beta_1 < 2$. En conséquence, le nombre de déplacements externes du robot dans P' est au plus $\beta_1 \cdot C$.

Considérons maintenant les déplacements externes dans $T' \setminus P'$. Nous avons prouvé que, lorsque le robot est pour la première fois à distance $\lfloor \lambda D_0/2 \rfloor$ du sommet de départ, alors il est sur P'. Nous avons aussi prouvé que le robot ne visite jamais $T' \setminus P'$ par un déplacement externe après qu'il ait atteint P' par un déplacement externe. D'après la formulation du schéma $\mathcal{X}_c$ (cf. Figure 5.2), le robot effectue un déplacement externe sur chaque arête de $T' \setminus P'$ au plus deux fois. Puisque $|T' \setminus P'| \leq \epsilon n$, le nombre de déplacements externes du robot dans $T' \setminus P'$ est au plus $2\epsilon n$. Ceci conclut la preuve de l'assertion 2.

Les assertions 1 et 2 impliquent que le coût total du schéma $\mathcal{X}_c$ est au plus $\beta_1 \cdot C + (154 + 2)\epsilon n = \beta_1 \cdot C + 156\epsilon n$.

Il reste à borner le rapport ρ de ce coût sur $opt(T, u)$. La plus courte marche couvrant T et débutant au sommet u visite u' avant tout autre sommet de P'. Elle doit donc visiter

le chemin P' en commençant en u'. En conséquence, la longueur de la plus petite marche couvrant T et commençant en u ne peut pas être moins que C (le nombre optimal de déplacements dans P' en commençant en u'). Cela donne $\rho \leq \frac{\beta_1 \cdot C + 156\epsilon n}{C}$. Nous avons $\epsilon \leq \epsilon_2 = \frac{2-\beta_1}{624}$. Avec $C \geq |P'| \geq n/2$ (voir la preuve de l'assertion 2), on obtient

$$\beta_1 + \frac{156\epsilon n}{C} \leq \beta_1 + \frac{156\epsilon n}{n/2} \leq \beta_1 + 2 \cdot 156 \frac{2-\beta_1}{624} = \beta_1 + \frac{2-\beta_1}{2} = 1 + \frac{\beta_1}{2} < 2.$$

On déduit des estimations obtenues ci-dessus que le rapport compétitif de l'algorithme $\texttt{SKE}(c)$ est au plus

$$\max\left(\frac{2}{1+\epsilon^*/2}, \frac{2}{1+\epsilon_3/2}, 1 + \frac{\beta_1}{2}\right) < 2,$$

ce qui conclut la preuve de ce théorème. □

5.4 La borne inférieure

Cette section est dévolue à l'établissement d'une borne inférieure sur la taille d'un oracle pour lequel il existe un algorithme avec un rapport compétitif plus petit que 2. Cette borne inférieure coïncide exactement avec la borne supérieure montrée précédemment, et elle reste vraie même pour la classe des lignes. En fait, nous montrons que pour les oracles dont la taille pour toutes les lignes L_k, de diamètre (i.e., de longueur) $k \leq n$, est plus petite que $\log\log n$ et en diffère par un nombre non borné de bits, alors tout algorithme à un rapport compétitif d'au moins 2.

Théorème 5.2 *Soit $\mathcal{O}$ un oracle et soit $f(n)$ le maximum des tailles de $\mathcal{O}(L_k)$, pour $k \leq n$. Soit $g : \mathbb{N} \mapsto \mathbb{R}$ défini par la formule $f(n) = \log\log n - g(n)$. Si g est une fonction non bornée supérieurement, alors tout algorithme d'exploration utilisant l'oracle $\mathcal{O}$ a un rapport compétitif au moins égal à 2.*

Preuve. Nous prouvons d'abord l'assertion suivante.

Assertion 3. Pour toute paire d'entiers positifs (M, γ), il existe deux entiers $n_1 > n_2 \geq M$, tels que $\mathcal{O}(L_{n_1}) = \mathcal{O}(L_{n_2})$ et $n_1/n_2 \geq \gamma$.

Supposons que l'assertion est fausse. Prenons M et γ qui réfutent l'assertion. Soit $\psi : \{n : \ n > M\} \to \mathbb{R}$ la séquence définie par la formule $\psi(n) = \frac{\log\gamma \log n}{\log n - \log M}$. La séquence ψ converge vers $\log\gamma$, et donc est bornée. Soit A tel que $\psi(n) < A$ pour tout n. Puisque g est une fonction non bornée supérieurement, il existe $n_0 > M$ pour lequel $g(n_0) > \log A$. Soit x la taille de l'ensemble $\{\mathcal{O}(L_k) : k \leq n_0\}$. Nous avons

$$x \leq 2^{f(n_0)} = 2^{\log\log n_0 - g(n_0)} < 2^{\log\log n_0 - \log A} < 2^{\log\log n_0 - \log \frac{\log\gamma \log n_0}{\log n_0 - \log M}}$$

et donc $x < \frac{\log n_0 - \log M}{\log\gamma}$. Tous les entiers k avec $\mathcal{O}(L_k) = \mathcal{O}(L_{M\gamma^i})$ doivent être plus petits que $M\gamma^{i+1}$, pour $i \geq 0$. Donc toutes les valeurs de l'oracle pour les lignes $L_{M\gamma^i}$

sont distinctes, et ces valeurs sont au nombre de x. Nous avons $n_0 < M\gamma^x$ parce que $\mathcal{O}(L_{n_0}) = \mathcal{O}(L_{M\gamma^i})$, pour un $i < x$, et donc $n_0 < M\gamma^{i+1} \leq M\gamma^x$. En conséquence, $\log n_0 \leq \log M + x \log \gamma < \log M + \log n_0 - \log M$. Cette contradiction prouve l'assertion 3.

Nous montrons maintenant que tout algorithme utilisant l'oracle $\mathcal{O}$ doit avoir un rapport compétitif au moins égal à 2. D'après le lemme 5.1, il est suffisant de restreindre notre attention aux algorithmes produisant des schémas d'exploration de type 1 pour la classe des lignes. La distance d'investigation d'un tel schéma pour la ligne L_n dépend seulement de $\mathcal{O}(L_n)$. Considérons un algorithme $\mathcal{A}$ produisant un schéma de type 1 avec une distance d'investigation $\phi(\mathcal{O}(L_n))$. Soit β une constante quelconque fixée vérifiant $3/2 < \beta < 2$. Choisissons γ tel que $\frac{2\gamma}{\gamma+2} > \beta$, et M tel que $\frac{2M-1}{M+1} > \beta$. Donc $\gamma > 6$. Soient $n_1 > n_2 \geq M$ des entiers pour lesquels $\mathcal{O}(L_{n_1}) = \mathcal{O}(L_{n_2})$ et $n_1 \geq \gamma n_2$. Leur existence est garantie par l'assertion 3. Soit $y = \phi(\mathcal{O}(L_{n_1}))$. Le schéma effectue le premier changement de direction après y pas, aussi bien dans L_{n_1} que dans L_{n_2}, à moins qu'une extrémité soit rencontrée plus tôt. Considérons deux cas.

Si $y \leq n_2$, considérons le comportement de $\mathcal{A}$ sur L_{n_1}, avec le sommet de départ u à distance $y+1$ de l'extrémité vers laquelle se dirige le robot au départ. Comme $\gamma > 6$, c'est l'extrémité la plus proche de u. Alors

$$\frac{\mathcal{A}(L_{n_1}, u)}{opt(L_{n_1}, u)} = \frac{y + 2n_1 - 1}{y + n_1 + 1} \geq \frac{n_2 + 2n_1 - 1}{n_2 + n_1 + 1}$$
$$\geq \frac{(2\gamma+1)n_2 - 1}{(\gamma+1)n_2 + 1} \geq \frac{(2\gamma+1) - 1}{(\gamma+1) + 1} = \frac{2\gamma}{\gamma+2} > \beta \ .$$

Si $y > n_2$, considérons le comportement de $\mathcal{A}$ sur L_{n_2}, avec le sommet de départ u à distance $n_2 - 1$ de l'extrémité vers laquelle se dirige le robot au départ. Alors

$$\frac{\mathcal{A}(L_{n_2}, u)}{opt(L_{n_2}, u)} = \frac{2n_2 - 1}{n_2 + 1} \geq \frac{2M - 1}{M + 1} > \beta \ .$$

Cela prouve que le rapport compétitif de l'algorithme $\mathcal{A}$ est au moins 2. □

5.5 Exploration connaissant le diamètre

Nous avons montré dans la section 5.3 que très peu d'informations (moins de $\log \log D$ bits) étaient nécessaires pour battre le rapport compétitif 2, et en fait, la majorité de cette quantité d'information (tous les bits sauf un) concerne la valeur du diamètre D lui-même, et est utilisée pour établir une borne inférieure sur celui-ci. Ce bit supplémentaire, cependant, ne peut pas être déduit de D seul, et est en fait d'une importance cruciale. Dans cette section, nous prouvons le résultat surprenant que même un algorithme connaissant D *exactement* (c'est-à-dire recevant tous les $\lceil \log D \rceil$ bits de celui-ci), mais n'ayant pas de connaissance additionnelle, ne peut pas battre le rapport compétitif 2. Notons qu'un argument similaire prouve que la connaissance exacte du nombre n de sommets, sans autre information, n'est pas suffisante non plus.

Théorème 5.3 *Soit $\mathcal{A}$ un algorithme quelconque d'exploration à qui, pour tout arbre T, on donne le diamètre de T comme entrée. Alors le rapport compétitif de $\mathcal{A}$ est au moins 2.*

Preuve. Considérons un algorithme quelconque d'exploration $\mathcal{A}$ qui connaît seulement le diamètre D de l'arbre exploré. Fixons D. Soit S le schéma d'exploration retourné par $\mathcal{A}$ pour l'entrée D. Rappelons que nous pouvons considérer uniquement les schémas réguliers. Nous construisons un arbre T de diamètre D qui sera utilisé pour prouver une borne inférieure sur le rapport compétitif de l'algorithme. Supposons que le robot suive le schéma S.

La construction procède en phases. Par induction, après que la phase $i-1$ soit terminée, pour $i \geq 2$, il existe un sommet de degré 3, appelé v_i, qui a un voisin w_i avec le sous-arbre déjà construit enraciné en w_i. Les deux autres arêtes incidentes à v_i sont pendantes et la construction dans la phase i continue à partir d'elles. Dans la phase i, une ligne attachée au sommet v_i est ajoutée au sous-arbre construit précédemment. La phase 1 commence avec la partie précédemment construite consistant uniquement en le sommet de départ $v = v_1$ avec deux arêtes pendantes. La ligne ajoutée dans la phase i a deux *côtés*, correspondant aux deux arêtes pendantes de v_i. Par hypothèse d'induction, le sous-arbre enraciné en w_i est complètement exploré à la fin de la phase $i - 1$, donc par régularité du schéma, le robot n'entre pas dans ce sous-arbre pendant la phase i.

La phase i est décrite comme suit. Le robot commence en v_i dans une certaine direction (du côté 1) et, tant qu'il voit seulement des sommets de degré 2 sur son chemin, soit il avance indéfiniment sans se retourner, soit il fait demi-tour après x_i pas. Dans le premier cas, nous disons que $x_i = \infty$. Dans ce cas, nommé *Cas 1*, nous finissons la construction de l'arbre : une ligne de longueur $m + 1$ est ajoutée, avec le sommet v_i à distance 1 de l'extrémité du côté 2. L'entier m est ajusté de telle façon que le diamètre de l'arbre soit exactement D. Si le robot fait demi-tour après x_i pas, nous distinguons deux nouveaux cas : $x_i \geq 3$ et $x_i < 3$. Si $x_i \geq 3$, appelons-le *Cas 2*, la construction continue comme suit. Nous ajoutons une ligne de longueur $x_i + 2$, avec le sommet v_i à distance 1 de l'extrémité du côté 2, et l'autre extrémité remplacée par un sommet v_{i+1} de degré 3, à partir duquel la construction continuera dans la phase $i + 1$. La construction est continuée ainsi dans la phase i à moins qu'ajouter une telle ligne ne fasse dépasser le diamètre D de l'arbre, auquel cas nous procédons comme dans le Cas 1. Si $x_i < 3$, le robot fait demi-tour et va du côté 2 du sommet v_i. Tant qu'il ne voit que des sommets de degré 2 sur son chemin, soit il avance indéfiniment sans se retourner, soit il fait demi-tour après y_i pas du côté 2 de v_i. Dans la première situation, nommée *Cas 3*, nous finissons la construction de l'arbre en ajoutant une ligne de longueur $m + x_i + 1$, avec le sommet v_i à distance $x_i + 1$ de l'extrémité du côté 1. L'entier m est ajusté de telle façon que le diamètre de l'arbre soit exactement D. Si le robot fait demi-tour après y_i pas, nous distinguons deux nouveaux cas : $y_i \geq 3$ et $y_i < 3$. Si $y_i \geq 3$, appelons-le *Cas 4*, la construction continue comme suit. Nous ajoutons une ligne de longueur $x_i + y_i + 2$, avec le sommet v_i à distance $x_i + 1$ de l'extrémité du côté 1, et l'autre extrémité remplacée par un sommet v_{i+1} de degré 3, à partir duquel la construction continuera dans la phase $i+1$. La construction est continuée ainsi dans la phase i à moins qu'ajouter une telle ligne ne fasse dépasser le diamètre D

de l'arbre, auquel cas nous procédons comme dans le Cas 3. Si $y_i < 3$, considérons deux autre cas : le *Cas 5*, dans lequel le robot avance indéfiniment après le second demi-tour, en supposant qu'il ne voit que des sommets de degré 2 sur son chemin, et le *Cas 6*, s'il fait demi-tour après un certain nombre $x_1 + z_1$ de pas du côté 1 de v_i, sous la même condition. Dans le Cas 5, nous finissons la construction de l'arbre en ajoutant une ligne de longueur $m + x_i + y_i + 1$, avec le sommet v_i à distance $y_1 + 1$ de l'extrémité du côté 2. L'entier m est ajusté de telle façon que le diamètre de l'arbre soit exactement D. Dans le Cas 6, la construction continue comme suit. Nous ajoutons une ligne de longueur $x_1 + y_1 + z_1 + 2$, avec le sommet v_i à distance $y_1 + 1$ de l'extrémité du côté 2, et l'autre extrémité remplacée par un sommet v_{i+1} de degré 3, à partir duquel la construction continuera dans la phase $i+1$. La construction est continuée ainsi dans la phase i à moins qu'ajouter une telle ligne ne fasse dépasser le diamètre D de l'arbre, auquel cas nous procédons comme dans le Cas 5. Ceci conclut la description de la phase i de la construction. En résumé, dans chaque phase, excepté la dernière, un des Cas 2, 4, 6 a lieu, et dans la dernière phase, un des Cas 1, 3, 5 a lieu.

Notons P_i la partie de l'arbre construite pendant la phase i. Toute les parties P_i sont distinctes deux à deux. Puisque $P_1 \cup \cdots \cup P_{i-1}$ est entièrement exploré à la fin de la phase $i-1$, le robot n'entre pas dans ce sous-arbre, par régularité du schéma. Donc, lors de la phase i, le robot se déplace uniquement dans la partie P_i. Soit S_i le nombre de déplacements effectués par le robot dans la phase i, et opt_i la longueur de la plus courte marche couvrant la partie P_i.

Dans le Cas 1, on a $S_i \geq 2m+1$ et $opt_i \leq m+2$. Dans le Cas 2, on a $S_i \geq 3x_i + 3$ et $opt_i \leq x_i + 3$. Dans le Cas 3, on a $S_i \geq 3x_i + 2m + 1$ et $opt_i \leq 2x_i + m + 2$. Dans le Cas 4, on a $S_i \geq 4x_i + 3y_i + 3$ et $opt_i \leq 2x_i + y_i + 3$. Dans le Cas 5, on a $S_i \geq 4x_i + 3y_i + 2m + 1$ et $opt_i \leq x_i + 2y_i + m + 2$. Dans le Cas 6, on a $S_i \geq 5x_i + 4y_i + 3z_i + 3$ et $opt_i \leq x_i + 2y_i + z_i + 3$.

Soit $\alpha < 2$ quelconque fixé. Nous prouvons maintenant que, pour un D suffisamment large, le rapport $\mathcal{A}(T, v)/opt(T, v)$ est plus grand que α. Supposons que la dernière phase ait le nombre i. Soit $Z = S_1 + \cdots + S_{i-1}$. Dans les phases $j < i$, seuls les Cas 2, 4, 6 peuvent avoir lieu. Comme dans le Cas 2, on a $x_j \geq 3$, il s'ensuit que $S_j/opt_j \geq 2$. Comme dans le Cas 4, on a $y_j \geq 3$, il s'ensuit que $S_j/opt_j \geq 2$. Comme dans le Cas 6, on a $x_j \geq 1$, il s'ensuit que $S_j \geq 2x_j + 4y_j + 3z_j + 6$, et par conséquent $S_j/opt_j \geq 2$. Nous pouvons donc conclure que $Z \geq 2(opt_1 + \cdots + opt_{i-1})$.

Comme dans le Cas 3 on a $x_i < 3$ et dans le Cas 5 on a $x_i, y_i < 3$, il s'ensuit que dans chacun des Cas 1, 3, 5, on a $S_i \geq 2m + a$ et $opt_i \leq m + b$, pour certaines constantes a, b. Donc $\mathcal{A}(T, v) = S_1 + \cdots + S_i \geq Z + 2m + a$ et $opt(T, v) = opt_1 + \cdots + opt_i \leq Z/2 + m + b$. Cela implique que

$$\frac{\mathcal{A}(T, v)}{opt(T, v)} \geq \frac{Z + 2m + a}{Z/2 + m + b}.$$

D'autre part, Z est au moins la taille de $P_1 \cup \cdots \cup P_{i-1}$ et $m+6$ est au moins la taille de P_i, donc $Z + m + 6 \geq D$. Par conséquent, $\frac{\mathcal{A}(T,v)}{opt(T,v)}$ est plus grand que α, pour D suffisamment grand. □

5.6 Perspectives

Nous avons illustré dans ce chapitre l'utilisation qui peut être faite de notre concept d'oracle. En particulier, nous avons caractérisé très exactement la quantité d'informations que doit posséder a priori un algorithme d'exploration d'arbres pour avoir un rapport compétitif strictement inférieur à 2. Une approche différente serait de considérer que l'oracle ne fournit d'informations qu'à la demande, au cours de l'exécution de l'algorithme. Avec cette approche, calculer la taille de l'oracle optimum pour notre problème reste un problème ouvert. Par ailleurs, nous conjecturons que le concept d'oracle peut être utilisé comme mesure de difficulté pour une large classe de problèmes. Cette mesure quantitative pourrait permettre de comparer certains problèmes de calcul distribué. Dans [FIP06], nous comparons d'ailleurs deux problèmes classiques de communication : la diffusion d'un message et le réveil des nœuds d'un réseau.

Chapitre 6

Schéma d'étiquetage pour l'exploration

Dans ce chapitre, nous étudions le problème de l'exploration des graphes anonymes par un automate fini. Nous avons vu dans la sous-section 1.3.1 qu'aucun JAG (et donc aucun automate fini) ne pouvait explorer tous les graphes. Pour rendre possible l'exploration de tous les graphes par un automate fini, un oracle fournit à chaque sommet du graphe une chaîne binaire (c'est-à-dire une étiquette), accessible à l'automate. Ceci revient à donner une couleur à chaque sommet. Nous cherchons à minimiser le nombre de couleurs utilisées par l'oracle.

6.1 Introduction

Comme nous l'avons vu au début du chapitre 1, concevoir un agent mobile dédié à l'exploration et doté d'une mémoire de taille très réduite peut s'avérer utile pour l'exploration de grands réseaux comme internet. Les grands réseaux d'interaction ont souvent, en plus d'une grande taille, un grand degré maximum. De ce fait, nous désirons concevoir un automate de Mealy explorant tous les graphes et ayant un nombre d'états fini indépendant, non seulement du nombre de sommets du graphe, mais aussi de son degré.

Malheureusement, Budach [Bud78] a montré qu'aucun automate fini n'est capable d'explorer tous les graphes anonymes, même planaires et de degré borné. Chercher une machine plus complexe mais finie ne semble pas non plus être une solution. En effet, aucun JAG fini ne peut explorer tous les graphes [CR80]. Une solution est d'aider l'automate en lui fournissant des informations. Nous modélisons ces informations par le concept d'oracle que nous avons introduit au début de cette partie. Donner à un automate fini des informations que celui-ci pourrait utiliser librement n'a pas vraiment de sens. En effet, on ne pourrait plus considérer que l'automate a un nombre borné d'états. Il est par ailleurs envisageable de considérer un oracle fournissant à l'automate des informations que celui-ci ne peut pas utiliser librement, c'est-à-dire des informations sur lesquelles l'automate ne peut pas faire de calcul comme pourrait le faire une machine de Turing. A une chaîne donnée correspond donc une fonction de transition/sortie spécifique. Cette approche n'est

pas satisfaisante non plus car Rollik [Rol80] a montré que, pour un nombre d'états K donné, il existe un piège commun à tous les automates à K états. Par conséquent, nous décidons que l'oracle fournit une chaîne binaire aux sommets du graphe à explorer, et non à l'automate.

Plus formellement, l'oracle fournit un *schéma d'étiquetage pour l'exploration*. Un tel schéma est un algorithme d'étiquetage des sommets du graphe, calculé à partir du graphe muni de son orientation locale, tel qu'il existe un automate fini capable d'explorer tous les graphes ainsi étiquetés. Nous cherchons à minimiser la taille maximale d'une étiquette. Plus précisément, nous cherchons à minimiser le nombre d'étiquettes différentes possibles.

Comme conséquence du résultat de Budach, tout schéma d'exploration doit avoir au moins *deux* étiquettes. Dans ce chapitre, nous montrons que *trois* étiquettes sont suffisantes pour permettre à un automate fini d'explorer tous les graphes. Nous montrons de plus que cet étiquetage permet à l'automate de s'arrêter à la fin de l'exploration. Par ailleurs, l'automate fini que nous décrivons explore le graphe et s'arrête en temps $O(m)$.

Pour la classe des graphes de degré borné, nous concevons un schéma d'exploration utilisant des étiquettes encore plus petites. Plus précisément, nous montrons que seulement *deux* étiquettes différentes sont suffisantes pour permettre à un certain automate fini d'explorer tous les graphes de degré borné. En fait, nous prouvons qu'il existe un robot ayant une mémoire de $O(\log d)$ bits capable d'explorer, avec arrêt, tous les graphes de degré maximum au plus d en temps $O(d^{O(1)}m)$, en utilisant le schéma d'exploration à deux valeurs.

Tous ces résultats sont résumés dans le tableau 6.1. Les deux schémas que nous avons mentionnés s'exécutent en temps polynomial.

Etiquettes (#valeurs)	Mémoire du robot (#bits)	Temps (#traversées d'arêtes)
3	$O(1)$	$O(m)$
2	$O(\log d)$	$O(d^{O(1)}m)$

TAB. 6.1 – Résumé des principaux résultats.

Nous prouvons également plusieurs résultats d'impossibilité pour des automates à un état, c'est-à-dire sans mémoire. Le comportement d'un tel automate dépend uniquement du port d'entrée, du degré et de l'étiquette du sommet. En particulier, nous montrons que pour tout $d > 4$ et pour tout automate sans mémoire utilisant un étiquetage d'au plus $\lfloor \log d \rfloor - 2$ valeurs, il existe un graphe simple de degré maximum d que l'automate ne peut explorer. Cette borne inférieure sur le nombre d'étiquettes différentes nécessaires à l'exploration peut être augmentée exponentiellement à $d/2 - 1$ en autorisant des graphes avec boucles.

Les résultats de ce chapitre ont fait l'objet d'une publication [CFI+05], en collaboration avec Reuven Cohen, Pierre Fraigniaud, Amos Korman et David Peleg.

6.2 Un schéma d'étiquetage à 3 couleurs pour l'exploration des graphes arbitraires

Dans cette section, nous décrivons un schéma d'étiquetage pour l'exploration utilisant des étiquettes de seulement 2 bits (en fait 3 valeurs). Plus précisément, nous prouvons le résultat suivant.

Théorème 6.1 *Il existe un automate fini ayant la propriété que, pour tout graphe G, il est possible de colorer les sommets de G avec trois couleurs (ou alternativement de donner une étiquette de 2 bits à chaque sommet) de telle façon qu'en utilisant cet étiquetage, l'automate soit capable d'explorer le graphe G en entier, en partant de n'importe quel sommet et en s'arrêtant après avoir identifié que le graphe entier est exploré. De plus, le nombre total de traversées d'arêtes par l'automate est au plus $20m$.*

Afin de prouver le théorème 6.1, nous décrivons d'abord le schéma d'étiquetage $\mathcal{L}$ et ensuite l'algorithme d'exploration. L'étiquetage des sommets est en fait très simple. Il utilise trois étiquettes, appelées couleurs, et notées BLANC, NOIR, et ROUGE. Soit D le diamètre du graphe.

Le schéma d'étiquetage $\mathcal{L}$.

Prenons un sommet arbitraire r. Le sommet r est appelé la *racine* de l'étiquetage $\mathcal{L}$. Les sommets à distance d de r, $0 \leq d \leq D$, sont étiquetés BLANC si $d \bmod 3 = 0$, NOIR si $d \bmod 3 = 1$, et ROUGE si $d \bmod 3 = 2$.

L'ensemble $\mathcal{N}(u)$ des voisins de tout sommet u peut être partitionné en trois ensembles disjoints : (1) l'ensemble $\mathsf{pred}(u)$ des voisins plus proches de r que u ; (2) l'ensemble $\mathsf{succ}(u)$ des voisins plus éloignés de r que u ; (3) l'ensemble $\mathsf{cousins}(u)$ des voisins à la même distance de r que u. Nous identifions également deux sous-ensembles spéciaux de voisins :

- $\mathsf{père}(u)$ est le sommet $v \in \mathsf{pred}(u)$ tel que l'arête $\{u, v\}$ ait le plus petit numéro de port en u parmi les arêtes menant à un sommet de $\mathsf{pred}(u)$.
- $\mathsf{fils}(u)$ est l'ensemble des sommets $v \in \mathsf{succ}(u)$ tels que $\mathsf{père}(v) = u$.

Pour la racine, nous posons $\mathsf{père}(r) = \emptyset$. L'algorithme d'exploration est en partie basé sur les observations suivantes.

1. Pour la racine r, $\mathsf{fils}(r) = \mathsf{succ}(r) = \mathcal{N}(r)$.
2. Pour tout sommet u d'étiquette $\mathcal{L}(u)$, et pour tout voisin $v \in \mathcal{N}(u)$, l'étiquette $\mathcal{L}(v)$ permet de déterminer de façon unique si v appartient à $\mathsf{pred}(u)$, $\mathsf{succ}(u)$ ou $\mathsf{cousins}(u)$.
3. Une fois sur le sommet u, un automate peut identifier $\mathsf{père}(u)$ en visitant ses voisins successivement, en commençant par le voisin connecté par le port 0, puis le port 1, et ainsi de suite. D'après l'observation 2, les sommets de $\mathsf{pred}(u)$ peuvent être identifiés par leur étiquette. L'ordre dans lequel l'automate visite les voisins assure que $\mathsf{père}(u)$ est le premier sommet visité dans $\mathsf{pred}(u)$.

Remarque 6.1 *La difficulté, dans l'exploration de graphes, pour un automate fini (nombre d'états constant) est que l'automate, arrivant par le port p en un sommet u, et désirant quitter le sommet par le même port p après avoir effectué une opération d'exploration locale autour de u, n'a pas assez de mémoire pour stocker la valeur de p.*

Algorithme d'exploration.

Notre algorithme d'exploration utilise une procédure appelée `Typer-Arête`. Cette procédure est spécifiée de la façon suivante. Quand `Typer-Arête`(j) est initié en un sommet u, l'automate commence par visiter les voisins de u un par un, et revient finalement en u, en retournant un des trois résultats possibles : "fils", "père", ou "faux". Ces valeurs ont les interprétations suivantes :

(i) si "fils" est retourné, alors l'arête numérotée j en u mène à un fils de u ;

(ii) si "père" est retourné, alors l'arête numérotée j en u mène au père de u ;

(iii) si "faux" est retourné, alors l'arête numérotée j en u mène à un sommet de $\mathcal{N}(u) \setminus (\mathsf{père}(u) \cup \mathsf{fils}(u))$.

L'implémentation de la procédure `Typer-Arête` sera décrite plus tard. Auparavant, nous décrivons comment l'algorithme se sert de cette procédure pour effectuer l'exploration.

Supposons que l'automate $\mathcal{A}$ est initialement sur la racine r du 3-coloriage $\mathcal{L}$ des sommets. $\mathcal{A}$ quitte r par le numéro de port 0, dans l'état DESCENTE. Notons que, d'après les précédentes observations, le sommet à l'autre extrémité de l'arête numéroté 0 de r est un fils de r.

Supposons que $\mathcal{A}$ arrive en un sommet u par le port i dans l'état DESCENTE. Supposons que u est de degré d. Toutes les opérations arithmétiques dans la description suivante sont effectuées modulo d. $\mathcal{A}$ a pour but d'identifier un fils de u s'il en existe, ou de repartir en arrière par le port i de u s'il n'en existe pas. Pour ce faire, l'automate exécute la procédure `Typer-Arête`(j) pour tous les numéros de port $j = i+1, i+2, \ldots$ jusqu'à ce que la procédure finisse par renvoyer "fils" ou "père" pour un certain numéro de port j. L'automate $\mathcal{A}$ transite alors dans l'état DESCENTE dans le premier cas, ou dans l'état MONTEE dans le second, et quitte u par le port j.

Supposons que $\mathcal{A}$ arrive en un sommet u par le port i dans l'état MONTEE. Supposons que u est de degré d. Toutes les opérations arithmétiques dans la description suivante sont effectuées modulo d. $\mathcal{A}$ a pour but d'identifier un fils de u de numéro de port $j \in \{i+1, \ldots, p-1\}$ s'il en existe (où p est le numéro de port de l'arête menant à $\mathsf{père}(u)$), ou de continuer à remonter vers le parent de u s'il n'existe pas de tel fils. Pour ce faire, l'automate exécute la procédure `Typer-Arête`(j) pour tous les numéros de port $j = i+1, i+2, \ldots$ jusqu'à ce que la procédure finisse par renvoyer "fils" ou "père" pour un certain numéro de port j. L'automate $\mathcal{A}$ transite alors dans l'état DESCENTE dans le premier cas, ou dans l'état MONTEE dans le second, et quitte u par le port j.

Si l'automate ne part pas de la racine r de l'étiquetage $\mathcal{L}$, il commence par s'y rendre en utilisant la procédure `Typer-Arête` pour identifier le père de chaque sommet intermédiaire, et en identifiant la racine comme l'unique sommet tel que $\mathsf{pred}(r) = \emptyset$.

De plus, l'automate peut s'arrêter une fois l'exploration achevée. Plus précisément, cela peut se faire en introduisant une petite modification dans le comportement de l'automate lorsqu'il entre dans un sommet u de degré d par le port $d-1$ dans l'état MONTEE. Dans ce cas, $\mathcal{A}$ vérifie si u a un père. Si oui, alors il agit comme précédemment ($\mathcal{R}$ n'a pas besoin de stocker d car d est le degré du sommet). Sinon, l'automate arrête l'exploration.

Procédure `Typer-Arête`.

Nous décrivons maintenant les actions de l'automate $\mathcal{A}$ quand la procédure `Typer-Arête`(j) est initiée en un sommet u. L'objectif de $\mathcal{A}$ est de fixer la variable `arête` à l'une des valeurs {père, fils, faux}. Nous notons v l'autre extrémité de l'arête e ayant le numéro de port j en u. Tout d'abord, $\mathcal{A}$ va en v dans l'état "check_edge", en se souvenant de la couleur du sommet u. Soit i le numéro de port de l'arête e en v. Il y a trois cas à considérer.

(a) $v \in \mathsf{cousins}(u)$ **:** Alors $\mathcal{A}$ revient en u par le port i et retourne "`arête` = faux".

(b) $v \in \mathsf{pred}(u)$ **:** Alors $\mathcal{A}$ a comme objectif de vérifier si v est le père de u, c'est-à-dire si u est le fils de v. Pour ce faire, $\mathcal{A}$ revient en u et procède comme suit : $\mathcal{A}$ visite successivement les arêtes numérotées $j-1, j-2, \ldots$ de u jusqu'à ce que, soit l'autre extrémité de l'arête appartienne à $\mathsf{pred}(u)$, soit toutes les arêtes $j-1, j-2, \ldots, 0$ aient été visitées. $\mathcal{A}$ pose alors "`arête`=faux" dans le premier cas et "`arête`=père" dans le second. A cet instant de l'exploration, soit k le numéro de port en u de la dernière arête visitée par $\mathcal{A}$. Alors $\mathcal{A}$ visite successivement les arêtes numérotées $k+1, k+2, \cdots$ jusqu'à ce que l'autre extrémité appartienne à $\mathsf{pred}(u)$. Ensuite, il revient en u et retourne la valeur de `arête`.

(c) $v \in \mathsf{succ}(u)$ **:** Alors $\mathcal{A}$ a comme objectif de vérifier si u est le père de v. Pour ce faire, $\mathcal{A}$ procède d'une manière similaire au cas (b), c'est-à-dire il visite successivement les arêtes numérotées $i-1, i-2, \ldots$ de v jusqu'à ce que, soit l'autre extrémité de l'arête appartienne à $\mathsf{pred}(u)$, soit toutes les arêtes $i-1, i-2, \ldots, 0$ aient été visitées. $\mathcal{A}$ pose alors "`arête`=faux" dans le premier cas et "`arête`=père" dans le second. A cet instant de l'exploration, soit k le numéro de port de la dernière arête incidente à v visitée par $\mathcal{A}$. Alors $\mathcal{A}$ visite successivement les arêtes numérotées $k+1, k+2, \cdots$ jusqu'à ce que l'autre extrémité w appartienne à $\mathsf{pred}(v)$. $\mathcal{A}$ est alors en u et retourne la valeur de `arête`.

Ceci conclut la description de notre procédure d'exploration.

Preuve du théorème 6.1. Clairement, l'étiquetage de tous les sommets par $\mathcal{L}$ peut être effectué en temps linéaire en m, le nombre d'arêtes du graphe. De façon évidente, deux bits sont suffisants pour encoder l'étiquette de chaque sommet. Il reste à prouver que l'algorithme d'exploration est correct.

Il est facile de vérifier que si la procédure `Typer-Arête` satisfait ses spécifications, alors l'automate $\mathcal{A}$ effectue globalement un parcours en profondeur d'abord (DFS) du graphe en utilisant les arêtes $\{u, v\}$ telles que $u = \mathsf{père}(v)$ ou $u \in \mathsf{fils}(v)$. Nous nous concentrons

donc sur la justesse de la procédure `Typer-Arête`(j) initiée en un sommet u. Soit v l'autre extrémité de l'arête e de numéro de port j en u, et soit i le numéro de port de e en v. Nous vérifions séparément les trois cas considérés dans la description de la procédure. D'après les observations précédentes, comparer la couleur du sommet courant avec la couleur de u permet à $\mathcal{A}$ de distinguer ces cas.

Si $v \in \mathsf{cousins}(u)$, alors v n'est ni un père ni un fils de u, et donc retourner "faux" est correct. De plus, $\mathcal{A}$ revient bien en u par le port i, comme spécifié dans le cas (a).

Si $v \in \mathsf{pred}(u)$, alors $v = \mathsf{père}(u)$ si et seulement si pour tout voisin w_k connecté à u par une arête de numéro de port $k \in \{j-1, j-2, \ldots, 0\}$, on a $w_k \notin \mathsf{pred}(u)$. L'automate vérifie bien cette propriété dans le cas (b) de la description en revenant en u, et en visitant tous les w_k. En conséquence, la procédure `Typer-Arête` se comporte correctement dans ce cas.

Finalement, si $v \in \mathsf{succ}(u)$, alors $v = \mathsf{fils}(u)$ si et seulement si pour tout voisin z_l connecté à v par une arête de numéro de port $l \in \{i-1, i-2, \ldots, 0\}$, on a $z_l \notin \mathsf{pred}(v)$. Dans le cas (c), l'automate vérifie bien cette propriété en visitant tous les z_l. A cet instant de l'exploration, il reste pour $\mathcal{A}$ à retourner en u (de façon évidente, le numéro de port menant de v en u ne peut pas être stocké dans la mémoire de l'automate puisqu'il a seulement un nombre constant d'états). Soit k le numéro de port de la dernière arête incidente à v que $\mathcal{A}$ visite avant de fixer la variable `arête` à "faux" ou "fils". Nous avons $0 \leq k \leq i-1$, $z_l \notin \mathsf{pred}(v)$ pour tout $l \in \{k+1, \ldots, i-1\}$, et $u \in \mathsf{pred}(v)$. Donc u peut être identifié comme le premier sommet à être rencontré lors d'une visite de tous les voisins de v en traversant successivement les arêtes $k+1, k+2, \ldots$ de v. C'est précisément ce que fait $\mathcal{A}$ d'après la description de la procédure dans le cas (c). En conséquence, la procédure `Typer-Arête` se comporte correctement dans ce cas.

En résumé, la procédure `Typer-Arête` se comporte correctement dans tous les cas. Il en est de même de l'algorithme d'exploration global. Il reste à calculer le nombre de traversées d'arêtes effectuées par l'automate durant l'exploration (en incluant les différents appels à `Typer-Arête`).

Nous utilisons à nouveau les même notations, comme dans la description et la preuve de la procédure `Typer-Arête`. Considérons la procédure `Typer-Arête`(j) initiée en un sommet u. Soit v l'autre extrémité de l'arête e de numéro de port j en u, et soit i le numéro de port de e en v. Observons tout d'abord que lors de l'exécution de la procédure `Typer-Arête`, seules des arêtes incidentes à u et v sont traversées. Plus précisément :

Cas (a) : $v \in \mathsf{cousins}(u)$. L'arête $e = \{u, v\}$ est traversée deux fois et aucune autre arête n'est traversée durant cette exécution de la procédure `Typer-Arête`.

Cas (b) : $v \in \mathsf{pred}(u)$. $\mathcal{A}$ traverse uniquement des arêtes incidentes à u. Soit k le plus grand numéro de port des arêtes menant en un sommet de $\mathsf{pred}(u)$ et vérifiant $k < j$. Si un tel k n'existe pas, nous posons $k = 0$. L'automate $\mathcal{A}$ traverse deux fois chaque arête $j, j-1, \ldots, k+1$ de u, ensuite deux fois l'arête k, et finalement de nouveau deux fois les arêtes $k+1, \ldots, j-1, j$. En résumé, l'arête k de u est explorée deux fois et les arêtes $k+1, \ldots, j-1, j$ sont explorées quatre fois.

Cas (c) : $v \in \mathsf{succ}(u)$. $\mathcal{A}$ traverse uniquement des arêtes incidentes à v. Soit k le plus

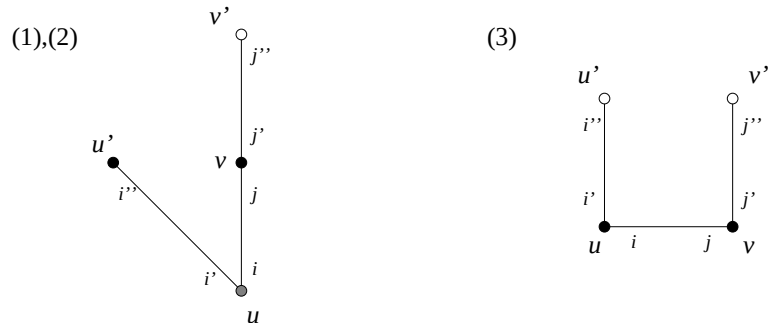

FIG. 6.1 – Notations pour les cas (1), (2) et (3) de l'analyse

grand numéro de port des arêtes menant en un sommet de $\mathsf{pred}(v)$ et vérifiant $k < i$. Si un tel k n'existe pas, nous posons $k = 0$. L'automate $\mathcal{A}$ traverse une fois l'arête j de u, deux fois chaque arête $i-1, i-2, \ldots, k+1$ de v, deux fois l'arête k, deux fois les arêtes $k+1, \ldots, i-2, i-1$ et finalement une fois l'arête i de v (i.e. l'arête j de u). En résumé, l'arête j de u et l'arête k de v sont explorées deux fois et les arêtes $k+1, \ldots, i-2, i-1$ de v sont explorées quatre fois.

Nous bornons maintenant le nombre de fois que chaque arête e du graphe est traversée. L'arête $e = \{u, v\}$ a le numéro de port i en u et j en v. Considérons différents cas (voir la figure 6.1) :

(1) $e = \{u, v\}$ avec $v = \mathsf{père}(u)$. L'arête e est dans l'arbre couvrant, et donc est explorée deux fois en dehors de toute exécution de la procedure `Typer-Arête`. Durant l'appel `Typer-Arête`(j) en v, l'arête e est explorée deux fois. e est aussi explorée quatre fois durant `Typer-Arête`(i) en u, sauf si $i = 0$, auquel cas e est seulement explorée deux fois durant `Typer-Arête`(i) en u. S'il existe une arête $\{u', u\}$ étiquetée i' en u et i'' en u' telle que $i' < i$ et $u' \in \mathsf{pred}(u)$, alors l'arête e est explorée deux fois durant la procédure `Typer-Arête`(i') en u et de nouveau deux fois durant la procédure `Typer-Arête`(i'') en u'. S'il existe une arête $\{v', v\}$ étiquetée j' en v et j'' en v' telle que $j' < j$ et $v' \in \mathsf{pred}(v)$, alors l'arête e est explorée quatre fois durant l'appel `Typer-Arête`(j') en v et de nouveau quatre fois durant l'appel `Typer-Arête`(j'') en v'. En résumé, l'arête e est explorée au plus 20 fois durant un DFS.

(2) $e = \{u, v\}$ avec $v \in \mathsf{pred}(u)$ mais $v \neq \mathsf{père}(u)$. Durant l'appel `Typer-Arête`(j) en v, l'arête e est explorée deux fois. e est aussi explorée quatre fois durant l'appel `Typer-Arête`(i) en u. S'il existe une arête $\{u', u\}$ étiquetée i' en u et i'' en u' telle que $i' < i$ et $u' \in \mathsf{pred}(u)$, alors l'arête e est explorée deux fois durant l'appel `Typer-Arête`(i') en u et de nouveau deux fois durant l'appel `Typer-Arête`(i'') en u'. S'il existe une arête $\{v', v\}$ étiquetée j' en v et j'' en v' telle que $j' < j$ et $v' \in \mathsf{pred}(v)$, alors l'arête e est explorée quatre fois durant l'appel `Typer-Arête`(j') en v et de nouveau quatre fois durant l'appel `Typer-Arête`(j'') en v'. En résumé, l'arête e est explorée au plus 18 fois durant un DFS.

(3) $e = \{u, v\}$ avec $v \in \mathsf{cousins}(u)$. Durant l'appel `Typer-Arête`$(j)$ en v, l'arête e est explorée deux fois. e est aussi explorée deux fois durant l'appel `Typer-Arête`(i)

en u. S'il existe une arête $\{u', u\}$ étiquetée i' en u et i'' en u' telle que $i' < i$ et $u' \in \mathsf{pred}(u)$, alors l'arête e est explorée quatre fois durant l'appel `Typer-Arête`(i') en u et de nouveau quatre fois durant l'appel `Typer-Arête`(i'') en u'. S'il existe une arête $\{v', v\}$ étiquetée j' en v et j'' en v' telle que $j' < j$ et $v' \in \mathsf{pred}(v)$, alors l'arête e est explorée quatre fois durant l'appel `Typer-Arête`(j') en v et de nouveau quatre fois durant l'appel `Typer-Arête`(j'') en v'. En résumé, l'arête e est explorée au plus 20 fois durant un DFS.

Notre algorithme d'exploration effectue donc l'exploration en temps au plus $20m$, où m est le nombre d'arêtes dans le graphe G. □

6.3 Un schéma d'étiquetage à 2 couleurs pour l'exploration des graphes de degré borné

Dans cette section, nous décrivons un schéma d'étiquetage pour l'exploration utilisant des étiquettes de seulement 1 bit. Ce schéma nécessite un automate de $O(\log d)$ bits de mémoire pour l'exploration des graphes de degré maximum d. Plus précisément, nous prouvons le résultat suivant :

Théorème 6.2 *Il existe un automate fini ayant la propriété que, pour tout graphe G de degré borné par une constante d, il est possible de colorer les sommets de G avec deux couleurs (ou alternativement de donner une étiquette de 1 bit à chaque sommet) de telle façon qu'en utilisant cet étiquetage, l'automate soit capable d'explorer le graphe G en entier, en partant de n'importe quel sommet et en s'arrêtant après avoir identifié que le graphe entier est exploré. L'automate a $O(\log d)$ bits de mémoire, et le nombre total de traversées d'arêtes par l'automate est au plus $O(d^{(O(1)}m)$.*

Afin de prouver le théorème 6.2, nous décrivons d'abord le schéma d'étiquetage à 1 bit $\mathcal{L}'$, c'est-à-dire un coloriage de chaque sommet en noir ou blanc. Ensuite nous montrons comment effectuer l'exploration en utilisant $\mathcal{L}'$.

Le schéma d'étiquetage $\mathcal{L}'$.

Comme pour $\mathcal{L}$, prenons un sommet arbitraire $r \in V$, appelé la *racine*. Les sommets à distance d de r sont étiquetés suivant une fonction de d mod 8. Partitionnons les sommets en huit *classes* en posant

$$C_i = \{u \in V \mid \mathrm{dist}_G(r, u) \bmod 8 = i\}$$

pour $0 \leq i \leq 7$. Un sommet u est coloré en blanc si $u \in C_0 \cup C_2 \cup C_3 \cup C_4$, et en noir sinon. Posons également

$$\tilde{C}_1 = \{u \mid \mathrm{dist}_G(r, u) = 1\}$$

$$\widehat{C} = \{r\} \cup \{u \in C_2 \mid \mathrm{dist}_G(r, u) = 2 \text{ and } \mathcal{N}(u) = \tilde{C}_1\}.$$

Lemme 6.1 *Il existe une procédure de recherche locale permettant à un automate de $O(\log \Delta)$ bits de mémoire de décider si un sommet u appartient à $\widehat{C}$ et à $\tilde{C}_1$, et d'identifier les classes C_i pour tout sommet $u \notin \widehat{C}$.*

Preuve. Soit $\mathbf{N}$ (resp., $\mathbf{B}$) l'ensemble des sommets noirs (resp. blancs) ayant tous leurs voisins noirs (resp. blancs). On peut facilement vérifier que la classe C_1 et les classes $C_3, \ldots, C_7$ peuvent être redéfinies comme suit :

- $u \in C_6 \Leftrightarrow u \in \mathbf{N}$ et il existe un sommet dans $\mathbf{B}$ à distance au plus 3 de u ;
- $u \in C_7 \Leftrightarrow u \notin C_6$, u a un voisin dans C_6, et il n'existe aucun sommet dans $\mathbf{B}$ à distance au plus 2 de u ;
- $u \in C_1 \Leftrightarrow u$ est noir, u n'a aucun voisin dans $\mathbf{N}$, et u a un voisin blanc v qui n'a aucun voisin dans $\mathbf{B}$;
- $u \in C_5 \Leftrightarrow u$ est noir, et $u \notin C_1 \cup C_6 \cup C_7$;
- $u \in C_3 \Leftrightarrow u \in \mathbf{B}$, et il existe un sommet dans C_1 à distance au plus 2 de u ;
- $u \in C_4 \Leftrightarrow u$ a un voisin dans $\mathbf{B}$, et il n'existe aucun sommet dans C_1 à distance au plus 2 de u.

En se basant sur les caractérisations précédentes, un automate de $O(\log \Delta)$ bits peut aisément identifier les classes C_1 et $C_3, \ldots, C_7$ en effectuant une recherche locale. De plus, les ensembles $\tilde{C}_1$ et $\widehat{C}$ peuvent aussi être caractérisés comme suit :

- $u \in \tilde{C}_1 \Leftrightarrow u \in C_1$, et il n'existe aucun sommet dans C_7 à distance au plus 2 de u ;
- $u \in \widehat{C} \Leftrightarrow N(u) \subseteq \tilde{C}_1$ et tout sommet v à distance au plus 2 de u vérifie $|N(v) \cap \tilde{C}_1| \leq |N(u)|$.

On peut en déduire :

- $u \in C_0 \setminus \widehat{C} \Leftrightarrow u \notin (\cup_{i=3}^{7} C_i) \cup C_1$ et u a un voisin dans C_7 ;
- $u \in C_2 \setminus \widehat{C} \Leftrightarrow u \notin \widehat{C} \cup C_1$, et u a un voisin dans C_1, mais n'a aucun voisin dans C_7.

Il s'ensuit qu'un automate de $O(\log \Delta)$ bits peut identifier la classe de chaque sommet excepté pour les sommets dans $\widehat{C}$. □

Preuve du théorème 6.2. L'algorithme d'exploration pour $\mathcal{L}'$ suit la même stratégie que l'algorithme d'exploration pour $\mathcal{L}$. En effet, pour $u \in C_i$, on a

$$\begin{aligned} \mathsf{pred}(u) &= \mathcal{N}(u) \cap C_{i-1} \pmod 8 \\ \mathsf{succ}(u) &= \mathcal{N}(u) \cap C_{i+1} \pmod 8 \\ \mathsf{cousins}(u) &= \mathcal{N}(u) \cap C_i \end{aligned}$$

En conséquence, d'après le lemme 6.1, toutes les instructions de l'algorithme d'exploration utilisant l'étiquetage $\mathcal{L}$ peuvent être exécutées en utilisant l'étiquetage $\mathcal{L}'$, sauf pour les cas non capturés dans le lemme 6.1, c'est-à-dire $\widehat{C}$.

Pour résoudre le problème de l'identification de la racine, nous remarquons que chacun des sommets de $\widehat{C}$ peut être utilisé comme racine, et tous les autres peuvent alors être considérés comme des feuilles dans C_2. Donc, en quittant la racine, l'automate mémorise le numéro de port p qu'il devrait prendre pour retourner à la racine. Quand l'automate quitte un sommet $u \in \tilde{C}_1$ par le port p, il le quitte dans l'état MONTEE et efface le contenu

de la variable contenant p. Ensuite, sur le sommet atteint, l'automate agit comme si le sommet était dans la classe C_0 (ce qui est d'ailleurs le cas puisqu'il est sur la racine) : il descend par le prochain port non exploré s'il en reste un. Quand l'automate quitte un sommet $u \in \tilde{C}_1$ par un port différent de p, il le quitte dans l'état DESCENTE et agit ensuite en supposant que le sommet atteint est dans C_2 : il revient en u.

Si l'exploration commence à la racine, alors les modifications précédentes suffisent. Pour gérer les explorations commençant dans des sommets arbitraires, il est nécessaire d'identifier la racine. Puisque tout sommet dans $\widehat{C}$ peut être utilisé comme racine, il suffit de trouver un sommet de $\widehat{C}$ en remontant et ensuite de débuter l'exploration comme il a été décrit précédemment. □

6.4 Résultats d'impossibilité

Théorème 6.3 *Pour tout $d > 4$, et pour tout automate à un unique état utilisant au plus $d/2 - 1$ couleurs, il existe un graphe (avec boucles) de degré maximum d et ayant au plus $d + 1$ sommets que l'automate n'arrive pas à explorer.*

Preuve. Fixons $d > 4$ et supposons par contradiction qu'il existe un automate à un état explorant tous les graphes de degré d colorés avec au plus $d/2 - 1$ couleurs. Rappelons que lorsqu'un automate à un état arrive en un sommet v par le port i, il quitte v par le port j où j dépend uniquement de i, d et de la couleur c de v. Donc pour un d fixé, chaque couleur correspond à une fonction des ports d'entrée vers les ports de sortie, à savoir une fonction de $\{0, 1, \cdots, d-1\}$ dans $\{0, 1, \cdots, d-1\}$. Partitionnons les fonctions correspondant aux couleurs des sommets de degré d en fonctions surjectives $f_1, f_2, \cdots, f_t$ et non-surjectives $g_1, g_2, \cdots, g_r$. On a $0 < t + r \leq d/2 - 1$. Soit c_i la couleur correspondant à f_i, et c_{t+i} la couleur correspondant à g_i. Pour chaque g_i, nous choisissons p_i parmi les numéros de port non atteints par g_i. Soit $p_0 \in \{0, 1, \cdots, d-1\} \setminus \{p_1, p_2, \cdots, p_r\}$ (c'est possible car $d - r \geq 1$).

Nous allons construire une famille $\{G_0, G_1, \cdots, G_t\}$ de graphes tels que, pour tout $k \in \{0, 1, \cdots, t\}$:

1. G_k a exactement un sommet v de degré d (éventuellement avec des boucles) ;
2. les autres sommets de G_k sont des voisins de degré 1 de v ;
3. toutes les arêtes sont soit des boucles incidentes à v, soit des arêtes menant de v à un sommet de degré 1 ;
4. les arêtes étiquetées $p_1, p_2, \cdots, p_r$ en v (s'il y en a, i.e., si $r > 0$) ne sont pas des boucles (et donc mènent à des sommets de degré 1) ;
5. l'arête étiquetée p_0 mène à un des sommets de degré 1, noté u_0 ;
6. il existe un ensemble $X_k \subseteq \{0, 1, \cdots, d-1\}$ tel que $\{p_0, p_1, \cdots, p_r\} \subseteq X_k$ et $d - |X_k| > 2(t - k)$, et pour lequel, dans G_k, les arêtes de numéro de port qui ne sont pas dans X_k mènent à des sommets de degré 1.

Nous allons prouver la propriété suivante pour $k = 0, \cdots, t$:

Propriété P_k. Dans G_k, si la couleur de v est dans $\{c_1, \cdots, c_k\}$, alors l'automate, démarrant en $u_0 \in V(G_k)$, ne peut pas explorer G_k. Plus précisément, tout sommet attaché en v par une arête de numéro de port qui n'est pas dans X_k n'est pas visité par l'automate.

Nous prouvons P_k par récurrence sur k. Soit G_0 l'étoile composée d'un sommet v de degré d et de d sommets feuille. Soit $X_0 = \{p_0, p_1, p_2, \cdots, p_r\}$. Rappelons que $t+r \leq d/2-1$. D'où $t \leq d/2-1$ et donc $2t+r+1 \leq d-1$. Par conséquent, on a $d-|X_0| = d-(r+1) > 2t$. P_0 est trivialement vrai.

Considérons $k > 0$. Soit G_{k-1} et X_{k-1} respectivement un graphe et un ensemble vérifiant la propriété de récurrence pour $k-1$. Supposons d'abord que v est coloré par la couleur c_k et que le robot démarre son parcours en u_0. Si le robot ne visite jamais des sommets attachés en v par des ports absents de X_{k-1}, alors le graphe G_{k-1} et l'ensemble X_{k-1} vérifient P_k. C'est-à-dire que $G_k = G_{k-1}$ et $X_k = X_{k-1}$. Sinon, soit p le premier port dans X_{k-1} utilisé par le robot en v, lorsqu'il démarre de u_0. Pour un port $i \in \{0, 1, \cdots, d-1\}$, posons $\mathsf{jumeau}(i) = j$ s'il existe un port j et une boucle étiquetée par i et j dans G_{k-1}. Posons $\mathsf{jumeau}(i) = i$ sinon. Nous définissons une suite de ports comme suit. Soit i_1 le port dans X_{k-1} tel que $f_k(i_1) = p$. Pour tout $l \geq 2$, soit i_l le port tel que $f_k(i_l) = \mathsf{jumeau}(i_{l-1})$. Cette suite est bien définie car f_k est surjectif.

Observons qu'il existe un certain l tel que $i_l \notin X_{k-1}$. En effet, supposons, pour obtenir une contradiction, que $i_l \in X_{k-1}$ pour tout l. Puisque X_{k-1} est fini, il existe un certain $i_l = i_{l+m}$ pour $m \geq 1$. Soit i_l le premier port répété deux fois dans la suite. Si $l > 1$, on a $f_k(i_l) = \mathsf{jumeau}(i_{l-1})$ et $f_k(i_{l+m}) = \mathsf{jumeau}(i_{l+m-1})$. Donc $\mathsf{jumeau}(i_{l-1}) = \mathsf{jumeau}(i_{l+m-1})$, ce qui donne $i_{l-1} = i_{l+m-1}$ par bijectivité de f_k, ce qui contredit la minimalité de l. Si $l = 1$, on a $i_1 = i_{1+m}$, et donc $i_m = p$, contredisant $i_j \in X_{k-1}$ pour tout j.

Par le résultat précédent, soit h le plus petit indice tel que $i_h \notin X$. Soit $q = i_h$. Si $q = p$, alors posons $G_k = G_{k-1}$ et $X_k = X_{k-1} \cup \{p\}$. Si $q \neq p$, alors connectons les ports p et q pour créer une boucle. Notons G_k le graphe obtenu et posons $X_k = X_{k-1} \cup \{p, q\}$.

Dans G_k, si v prend la couleur c_k, alors par le choix de p, en démarrant de u_0, l'automate entre et sort de v par des ports dans X_{k-1} jusqu'à ce qu'il finisse par quitter v par le port p. L'automate ressort alors de l'arête par le port q. Le port q a été choisi de telle façon que l'automate continue d'arriver en v par les ports $i_{h-1}, i_{h-2}, \cdots i_1$, après quoi l'automate quitte v par le port p, piégeant l'automate dans un cycle. Puisque les ports de v apparaissant dans ce cycle sont tous dans X_k, l'automate ne visite aucun des ports hors de X_k, comme affirmé. Par récurrence, nous avons $d - |X_{k-1}| > 2(t-(k-1))$. Par construction de X_k à partir de X_{k-1}, nous avons $|X_k| \leq |X_{k-1}|+2$. Donc $d-|X_k| > 2(t-k)$, ce qui conclut la correction de G_k et X_k.

Si la couleur de v dans G_k est dans $\{c_1, \cdots, c_{k-1}\}$, l'automate échoue dans l'exploration de G_k. En effet, puisque l'automate partant de u_0 dans G_{k-1} ne traverse aucun des sommets correspondant aux ports non dans X_{k-1}, alors dans G_k non plus, l'automate ne visite aucun des sommets correspondants aux ports absents de $X_k \supseteq X_{k-1}$, et donc échoue dans l'exploration de G_k car $d - |X_k| \geq 1$. Ceci termine la preuve de P_k, ainsi que la preuve de récurrence.

En particulier, G_t n'est pas exploré par l'automate si le sommet v est coloré par une

couleur dans $c_1, c_2, \cdots, c_t$. Si v est coloré par c_{t+i} avec $1 \leq i \leq r$, alors supposons que l'automate démarre en u_0. Puisque l'arête étiquetée p_i mène à un sommet de degré 1 dans G_t, ce sommet ne sera jamais visité par l'automate, par définition de p_i. Par conséquent, le graphe G_t ne peut pas être exploré par l'automate. □

Le théorème ci-dessus utilise des graphes avec des boucles. Pour les graphes simples, nous avons le théorème suivant.

Théorème 6.4 *Pour tout $d > 4$ et pour tout automate à un état utilisant au plus $\lfloor \log d \rfloor - 2$ couleurs, il existe un graphe simple de degré maximum d que l'automate ne peut pas explorer.*

Preuve. Comme dans la preuve du théorème 6.3, notons les fonctions correspondant aux couleurs par $f_1, f_2, \cdots, f_t, g_1, g_2, \cdots, g_r$ où les f_i sont des fonctions surjectives et les g_i des fonctions non surjectives. Pour chaque g_i, nous choisissons p_i parmi les numéros de port non atteints par la fonction g_i. Nous utilisons la même preuve que pour le théorème 6.3, à ceci près que nous remplaçons chaque boucle L de la construction de la preuve par un graphe sans boucle. Pour une boucle L du sommet v, avec les numéros de port i et j, nous considérons le graphe G_L suivant. Connectons v par des arêtes à deux nouveaux sommets w et w', ajoutons l'arête (w, w'), et connectons w (respectivement, w') à $d-2$ nouveaux sommets $n_1, n_2, \cdots, n_{d-2}$ (resp., $n'_1, n'_2, \cdots, n'_{d-2}$). Nous définissons maintenant l'orientation locale pour G_L. Donnons les numéros de port i et j aux arêtes $\{v, w\}$ et $\{v, w'\}$ en v. En w (respectivement, w') nous utilisons les numéros de port $p_1, p_2, \cdots, p_r$ pour les arêtes $\{w, n_1\}, \{w, n_2\}, \cdots, \{w, n_r\}$ (resp., $\{w', n'_1\}, \{w', n'_2\}, \cdots, \{w', n'_r\}$). Les numéros de port p et q que w utilise pour les arêtes $\{w, v\}$ et $\{w, w'\}$ sont les mêmes que les numéros de port que w' utilise pour les arêtes $\{w', v\}$ et $\{w', w\}$. Ils sont choisis d'une façon particulière dans l'ensemble $\{0, 1, \cdots, d-1\} \setminus \{p_1, p_2, \cdots, p_r\}$ qui sera décrite plus tard. Nous obtenons immédiatement que w et w' ne peuvent pas avoir une couleur dont la fonction correspondante appartient à l'ensemble $\{g_1, g_2, \cdots, g_r\}$ puisque ces fonctions ne sont pas surjectives et que les ports $\{p_1, p_2, \cdots, p_r\}$ mènent à des feuilles de w et w'. Pour conclure la preuve, nous affirmons le résultat suivant :

Assertion : Il existe deux ports $p, q \in \{0, 1, \cdots, d-1\} \setminus \{p_1, p_2, \cdots, p_r\}$ tels que si l'automate arrive soit en w soit en w' par le port p (respectivement, q), alors si plus tard l'automate quitte le sommet par le même port p (resp. q), entre temps il devra être sorti par le port q (resp. p).

Nous montrons le résultat pour w. Rappelons que w peut uniquement être coloré par une couleur donc la fonction correspondante est dans l'ensemble $f_1, f_2, \cdots, f_t$. Chaque fonction f_i est une permutation sur d éléments. Si pour certains f_i la décomposition en cycle de la permutation consiste en plus de deux cycles, alors w ne peut pas être coloré par f_i car quels que soient les ports p et q choisis, l'automate ne peut visiter que les ports correspondant aux cycles auxquels appartiennent p et q. Nous pouvons donc supposer que la décomposition en cycle de chaque f_i consiste en au plus deux cycles. Par conséquent, f_1 a un cycle de longueur au moins $d/2$. Par récurrence, f_k a un cycle C_k dont l'intersection avec $C_1, C_2, \cdots, C_{k-1}$ est de taille au moins $d/2^k$. Puisque $d \geq 2^{k+1}$, il existe au moins

deux ports p et q dans $\cap_{i=1}^{t} C_i$. Si l'automate arrive en w par le port p (respectivement, q) et si w est coloré par une couleur correspondant à f_k, alors puisque p et q sont dans le cycle C_k de f_k, l'automate doit atteindre le port q (resp. p) avant de revenir au port p (resp. q). Ceci conclut la preuve de l'assertion.

Il s'ensuit que, à chaque fois que l'automate entre dans G_L depuis le port i de v, il doit retourner en v par le port j (et vice versa). La boucle L dans la preuve du théorème 6.3 peut donc être remplacée par G_L. □

6.5 Perspectives

Rollik [Rol80] a montré qu'il n'existe pas de schéma d'exploration à une couleur (c'est-à-dire sans couleur), même dans les graphes de degré borné. Nous avons prouvé qu'il existe un schéma d'exploration à trois couleurs pour les graphes arbitraires et à deux couleurs pour les graphes de degré borné. Le problème de savoir s'il existe ou non un schéma d'exploration à deux couleurs pour les graphes arbitraires reste ouvert. D'autre part, un résultat prouvant que ces couleurs peuvent être calculées à la volée par un automate fini se déplaçant dans le graphe permettrait un rapprochement intéressant avec les résultats d'exploration lorsque les sommets sont munis de tableaux blancs.

Chapitre 7

Orientations locales pour l'exploration

Dans ce chapitre, nous considérons une aide à l'automate encore plus puissante. En effet, nous supposons ici que l'orientation locale peut être choisie pour faciliter l'exploration par un automate fini. Habituellement, un automate ou un robot est dit capable d'explorer tous les graphes s'il est capable d'explorer tout graphe, quel que soit le sommet de départ et quelle que soit son orientation locale. Nous considérons ici qu'un automate explore un graphe lorsqu'il existe une orientation locale telle que le graphe muni de celle-ci est exploré avec succès par l'automate, quel que soit le sommet de départ. Nous étudions plus particulièrement l'exploration *périodique* par un automate fini.

7.1 Introduction

Dans tous les autres chapitres, nous avons considéré qu'un automate explore avec succès un graphe G lorsque, partant d'un sommet quelconque du graphe, l'automate explore tous les sommets de G, quelle que soit l'orientation locale dont il est muni. Par conséquent, lorsqu'un adversaire veut mettre en défaut un automate, il a la liberté du choix du graphe, du sommet de départ *et* de l'orientation locale. Rappelons que si l'orientation locale est choisie par l'adversaire, aucun automate fini, et même aucun JAG, ne peut explorer tous les graphes [CR80]. Nous montrons dans ce chapitre que le choix de l'orientation locale a une importance capitale. En effet, si le choix de l'orientation locale est laissé au concepteur de l'automate plutôt qu'à l'adversaire, l'exploration de graphes devient beaucoup plus facile et il est simple de montrer qu'il existe un automate fini capable d'explorer tous les graphes munis d'une orientation locale convenablement choisie.

Dans ce chapitre, nous revisitons ce résultat en considérant le temps d'exploration par un automate de Mealy fini. Nous considérons en fait l'exploration périodique. Rappelons que l'exploration périodique consiste à visiter chaque sommet du graphe infiniment souvent. Ce type d'exploration est particulièrement utile pour la maintenance de réseaux : l'automate vérifie périodiquement le bon fonctionnement de tout nœud du réseau. De ce fait, nous cherchons à minimiser la période d'une telle exploration, c'est-à-dire le nombre

maximum de traversées d'arêtes séparant deux visites consécutives d'un quelconque sommet. Plus précisément, nous nous intéressons au problème suivant :

Problème. *Quelle est la fonction minimale $\pi(n)$ telle qu'il existe un algorithme fixant l'orientation locale et un automate fini l'utilisant pour explorer tous les graphes de taille n avec une période au plus $\pi(n)$?*

Une borne supérieure triviale sur cette période est $2m$, où m est le nombre d'arêtes du graphe exploré. On peut en effet choisir l'orientation locale de telle façon qu'une marche en utilisant la règle de la main droite (c'est-à-dire définie par $\delta(s, i, d) = (s, i + 1 \bmod d)$) induise un cycle eulérien du graphe où chaque arête est traversée deux fois, une fois dans chaque direction. Dobrev et al. [DJSS05] ont présenté un algorithme fixant l'orientation locale et un automate fini l'utilisant, tels que l'automate explore tous les graphes de taille n avec une période inférieure ou égale à $10n$. Donc $\pi(n) \leq 10n$. Le principal avantage de leur approche est que leur automate est ultimement simple : il n'a qu'un seul état (c'est-à-dire il n'a pas de mémoire). Utiliser un automate à un seul état résout bien évidemment le problème de l'initialisation de l'automate. Cependant, la bonne performance de l'automate dans [DJSS05] repose sur le fait que celui-ci démarre l'exploration en partant de l'arête de numéro de port 0.

Notre principal résultat est la conception d'un algorithme très simple pour fixer l'orientation locale de tout graphe, et la conception d'un automate à trois états réalisant l'exploration périodique de tout graphe muni de l'orientation locale calculée par l'algorithme. La période de cette exploration est d'au plus $4n-2$ maximum. Ceci prouve donc $\pi(n) \leq 4n-2$. De plus, les bonnes performances de cette exploration sont indépendantes de l'état initial et de la position de départ de l'automate.

Notre algorithme fixant l'orientation locale est basé sur le calcul d'un arbre couvrant du graphe et sur la construction d'une orientation locale à partir de cet arbre couvrant. Nous prouvons que notre algorithme d'étiquetage peut facilement être modifié en un algorithme distribué ou utilisé dans un environnement dynamique, répondant ainsi à des problèmes ouverts décrits dans [DJSS05].

Les résultats de ce chapitre sont publiés dans [Ilc06].

7.2 L'orientation locale et l'automate correspondant

Nous décrivons d'abord notre algorithme calculant l'orientation locale du graphe. Cet algorithme est principalement basé sur le codage d'un arbre couvrant du graphe en choisissant les petits numéros de port pour les arêtes de l'arbre couvrant.

Ensuite, nous présentons un automate de Mealy à trois états qui explore l'arbre couvrant construit (plus quelques arêtes additionnelles) à la manière d'un parcours en profondeur d'abord (DFS).

Nous concluons cette section en prouvant que notre algorithme et l'automate correspondant sont corrects.

7.2.1 Algorithme calculant l'orientation locale

Soit $G = (V, E)$ un graphe. Considérons un arbre couvrant T de G. Soit $F \subseteq E$ l'ensemble des arêtes de T. Pour tout sommet $v \in V$, on pose F_v l'ensemble des arêtes de F incidentes à v.

Définition 28 *Une orientation locale des arêtes du graphe G est* compatible *avec l'arbre couvrant $T = (V, F)$ si et seulement si :*

- *pour toute arête $e \in E$, au moins l'un de ses deux numéros de port est 0 si et seulement si $e \in F$;*
- *pour tout sommet $v \in V$, les arêtes dans F_v ont leur numéro de port de 0 à $|F(v)|-1$.*

Une orientation locale de G est dite arbre-orientée *s'il existe un arbre couvrant T de G tel que l'orientation locale est compatible avec T.*

Notre algorithme, appelé Petits-Ports, construit des orientations locales qui sont arbre-orientées. Pour fixer les idées, l'algorithme utilise l'orientation locale suivante.

Algorithme Petits-Ports *:*

1. Prendre un arbre couvrant enraciné T de G. Soit r sa racine.
2. Pour tout sommet $v \neq r$, affecter le numéro de port 0 à l'arête de T allant en direction de la racine. En r, affecter le numéro de port 0 à une arête arbitraire de F_r.
3. Pour tout sommet v de G, affecter arbitrairement les numéros de port de 1 à $|F_v|-1$ aux arêtes restantes de F_v, s'il y en a.
4. Finalement, affecter arbitrairement les numéros de port de $|F_v|$ à $d_v - 1$ aux arêtes auxquelles aucun numéro de port n'a encore été affecté, s'il en reste.

Cette orientation locale est clairement compatible avec T.

Remarque 7.1 Petits-Ports *est très simple puisqu'il ne requiert que le calcul d'un arbre couvrant pour fixer l'orientation locale. De plus, de nombreuses applications utilisent un arbre couvrant comme structure sous-jacente et dans ce cas,* Petits-Ports *peut l'utiliser sans surcoût. Les performances et la simplicité de* Petits-Ports *doivent être comparées à celles de l'algorithme présenté dans [DJSS05].* Petits-Ports *s'exécute en temps $O(m)$ tandis que l'algorithme dans [DJSS05] s'exécute en temps $O(n^3)$.*

Remarque 7.2 *Considérons un graphe G et une orientation locale arbre-orientée de G. Il existe un unique arbre couvrant T de G tel que cette orientation locale soit compatible avec T. Précisément, T est l'arbre composé des $n-1$ arêtes de G ayant au moins un de leurs deux numéros de port égal à 0. De plus, il existe exactement deux racines possibles pour T telles que l'orientation locale puisse être obtenue par l'exécution de l'algorithme* Petits-Ports *avec cet arbre enraciné. Ces deux racines sont les deux extrémités de l'unique arête ayant ses deux numéros de port égaux à 0.*

7.2.2 Description de l'automate

Notre automate explorant les graphes munis d'une orientation locale arbre-orientée est noté $\mathcal{A}$ et a trois états : N (pour *Normal*), T (pour *Test*), et R (pour *Retour*). La fonction de transition/sortie δ de l'automate est définie comme suit. Ici d représente le degré du sommet courant, et i le port d'entrée (éventuellement égal à $\perp$).

$$\begin{aligned}
\delta(N,i,d) &= \begin{cases} (N,0) & \text{if } i = d \\ (T,i+1) & \text{if } i \neq d \end{cases} \\
\delta(T,i,d) &= \begin{cases} (N,0) & \text{if } i = 0 \text{ and } d = 1 \\ (T,i+1) & \text{if } i = 0 \text{ and } d \neq 1 \\ (R,i) & \text{if } i \neq 0 \end{cases} \\
\delta(R,i,d) &= (N,0)
\end{aligned}$$

Intuitivement, l'automate traverse une arête dans l'état N quand il sait que l'arête appartient à l'arbre couvrant, dans l'état T quand il ne sait pas encore, et dans l'état R quand il sait que l'arête n'appartient pas à l'arbre couvrant.

7.2.3 Preuve

Théorème 7.1 *Soit G un graphe de taille n, avec une orientation locale arbre-orientée. Considérons l'automate $\mathcal{A}$ en un sommet quelconque, provenant d'une arête quelconque (ou du sommet), et étant dans un état quelconque. Après au plus deux étapes, l'automate entre dans un chemin fermé P et l'explore pour toujours. De plus, P est de longueur au plus $4n-2$ et contient tous les sommets de G.*

Preuve. Soit G un graphe arbitraire et n son nombre de sommets. Supposons que son orientation locale est compatible avec un certain arbre couvrant T de G. Nous étudions d'abord le comportement périodique de l'automate, et ensuite le régime transitoire initial.

Soit v un sommet arbitraire de G, et soit e son arête incidente de numéro de port 0. La suppression de e dans T a pour résultat deux composantes connexes (des sous-arbres). Soit T' la composante contenant v. Finalement, soit n' le nombre de sommets de T'.

Assertion Si l'automate $\mathcal{A}$ entre dans v par le port 0 dans un état différent de R, alors il finit par quitter v par le port 0 dans l'état N. De plus, entre ces deux événements, l'automate explore tous les sommets de T' en au plus $4n'-2$ étapes, et ne quitte aucun sommet hors de T' par le port 0 durant ces étapes.

Nous prouvons cette propriété par induction sur la hauteur h de T' enraciné en v, c'est-à-dire sur l'excentricité de v dans T'. Le cas $h = 0$ correspond à v feuille de T. Si v est aussi une feuille dans G, l'automate quitte immédiatement v par le port 0 dans l'état N et la propriété est prouvée. Nous supposons donc que $\deg(v) > 1$. Par hypothèse de l'assertion, l'automate arrive en v dans l'état T ou N. Dans les deux cas, il transite dans l'état T, et traverse l'arête e' de numéro de port 1. V est une feuille de T et comme e est dans T, e' ne l'est pas. Donc le numéro de port de e' à l'autre extrémité n'est pas égal

à 0. L'automate revient donc en v dans l'état R, et finalement quitte v par le port 0 après $4 \cdot 1 - 2 = 2$ étapes, ce qui prouve la base de la récurrence.

Considérons maintenant le cas $h > 0$. Soit d le degré de v. Nous avons $d \neq 1$ parce que v est incident à e et que la hauteur de T' est strictement positive. Pour $i \geq 1$, soit v_i le sommet à l'autre extrémité de l'arête e_i de numéro de port i en v. Si e_i est dans T, alors soit T_i la composante connexe de $T' \setminus \{e_i\}$ contenant v_i. Finalement, soit p le plus grand numéro de port d'une arête de T incidente à v. On a $p \geq 1$ parce que v n'est pas une feuille de v. Par hypothèse de récurrence, l'automate entre en v dans l'état T ou N. Dans les deux cas, il transite dans l'état T, et traverse l'arête e_1 de numéro de port 1. Supposons que l'automate quitte v par le port i dans l'état T, avec $1 \leq i \leq p$. Il atteint le sommet v_i. Par hypothèse de récurrence sur h, l'automate finit par revenir depuis v_i en v par le port i de v, dans l'état N, après au plus $4n_i - 2$ étapes. (Notons que lors de ce parcours, l'automate a pu avoir visité des sommets hors de T_i mais il n'a jamais quitté ces sommets par le port 0.) Si $i \neq d - 1$, l'automate quitte v par le port $i + 1$ dans l'état T. Donc l'automate explore successivement les sous-arbres T_i.

Si $p = d-1$, l'automate quitte finalement v par le port 0 dans l'état N après avoir fini l'exploration de T_p. Si $p < d$, l'automate sort par l'arête e_{p+1} dans l'état T. Puisque e_{p+1} n'appartient pas à l'arbre T, le numéro de port de e_{p+1} à son autre extrémité n'est pas égal à 0. L'automate revient donc en v dans l'état R et quitte finalement v par le port 0 dans l'état N. Dans les deux cas, il reste à borner le nombre d'étapes. L'automate a traversé $\sum_{i=1}^{p}(4n_i-2) = 4(n'-1)-2(p-1)$ arêtes durant l'exploration des sous-arbres T_i. Il a aussi traversé deux fois chaque arête e_i, $1 \leq i \leq p$. Finalement, il faut rajouter éventuellement deux traversées d'arêtes supplémentaires (si $p < d-1$). En résumé, le nombre de traversées d'arêtes, et donc d'étapes, est au plus $4(n' - 1) - 2(p - 1) + 2(p - 1) + 2 = 4n' - 2$. Ceci conclut la preuve de l'assertion.

Nous utilisons maintenant l'assertion précédente pour décrire le chemin fermé que l'automate parcourt périodiquement. Il existe une unique arête $e = \{v, v'\}$ ayant ses deux numéros de port égaux à 0. Supposons que l'automate arrive en v par l'arête e dans l'état N. Par application de l'assertion, l'automate explore le sous-arbre de $T\setminus\{e\}$ enraciné en v, revient en v et traverse l'arête e vers v' dans l'état N. Par une nouvelle application de l'assertion, l'automate explore le sous-arbre de $T \setminus \{e\}$ enraciné en v', revient en v' et traverse l'arête e vers v dans l'état N. En conséquence, l'automate parcourt un chemin fermé P de longueur au plus $4n - 2$, visitant tous les sommets de T, et donc de G.

Il reste à prouver que l'automate commence à suivre P après au plus deux étapes. L'automate est placé en un sommet quelconque, provenant d'une arête quelconque (ou du sommet), et étant dans un état quelconque. Par définition de la fonction de transition de l'automate, trois cas sont possibles :

- Cas 1 : l'automate quitte le sommet courant par le port 0 dans l'état N. Cela implique que l'automate commence à suivre immédiatement le chemin fermé P.
- Cas 2 : l'automate quitte le sommet courant dans l'état R. La prochaine traversée d'arête est alors le long de l'arête de numéro de port 0, dans l'état N. Cela implique que l'automate commence à suivre le chemin fermé P lors de la deuxième étape.
- Cas 3 : l'automate quitte le sommet courant par l'arête e de numéro de port i, avec

$i \geq 1$, dans l'état T. Supposons que, soit e est dans T, soit e est l'arête ayant le plus petit numéro de port qui n'est pas dans T. Dans ce cas, cette traversée d'arêtes est dans le chemin fermé P. Si ce n'est pas le cas, alors le numéro de port j à l'autre extrémité u de l'arête e n'est pas égal à 0 car e n'est pas dans T. Donc l'automate transite dans l'état R en u, et revient en v par e. Ensuite, il quitte v par le port 0 dans l'état N. Cette dernière traversée d'arête est dans P.

Finalement, dans tous les cas, l'automate commence à suivre le chemin fermé P après au plus deux étapes. □

7.3 Propriétés additionnelles

Dans la section précédente, nous avons présenté un algorithme simple, utilisant un arbre couvrant du graphe pour fixer l'orientation locale de celui-ci, et un automate à 3 états effectuant l'exploration périodique en temps au plus $4n$ en utilisant cette orientation, où n est le nombre de sommets du graphe exploré. Nous prouvons que grâce à la robustesse et à la simplicité de notre approche, il est possible d'utiliser notre algorithme dans un environnement distribué, et dans des réseaux dynamiques.

7.3.1 L'algorithme pour les environnements distribués

La construction distribuée d'un arbre couvrant un graphe anonyme peut être impossible si le graphe possède certaines symétries. Cependant cette tâche est possible si un unique sommet initie cette construction. Dans notre cas, nous utilisons l'automate pour briser la symétrie entre les sommets. La position de départ de l'automate est utilisée comme le sommet distingué, qui deviendra la racine de l'arbre couvrant. Ce sommet réveillera tous les autres sommets du graphe par inondation. Un sommet distinct de la racine choisit son père comme étant le sommet duquel il a reçu le premier message de réveil (les égalités sont tranchées arbitrairement). Finalement, la technique décrite dans la section 7.2.1 est utilisée pour fixer l'orientation locale, en se basant sur l'arbre couvrant construit.

Plus précisément, la variante distribuée de notre algorithme, appelée Petits-Ports-Distribué, se déroule comme suit. Au début, seul le sommet hébergeant l'automate est réveillé. Ce sommet est la racine r du futur arbre couvrant. r démarre le processus en envoyant un message "Hello" à chacun de ses voisins. Un sommet v, autre que la racine, est dit éveillé lorsqu'il a reçu au moins un message. Un sommet réveillé v choisit comme père l'expéditeur du premier message qu'il a reçu. Les égalités sont tranchées arbitrairement. Finalement v envoie un message "Père" au voisin choisi comme père dans l'arbre et un message "Hello" à chacun de ses autres voisins.

Quand un sommet u a reçu un message de tous ses voisins, il choisit les numéros de port de ses arêtes incidentes comme suit. Soit p le nombre de messages "Père" que le sommet u a reçus.

- Si u est la racine, il affecte arbitrairement les numéros de port de 0 à $p-1$ aux p arêtes menant aux expéditeurs des messages "Père" reçus par u. Il affecte arbitrairement les numéros de port restants, s'il y en a, aux arêtes restantes.
- Si u n'est pas la racine, il affecte le numéro de port 0 à l'arête menant au voisin choisi comme père. Ensuite, il affecte arbitrairement les numéros de port de 1 à p aux p arêtes menant aux expéditeurs des messages "Père" reçus par u, s'il y en a. Finalement, il affecte arbitrairement les numéros de port restants, s'il y en a, aux arêtes restantes.

Théorème 7.2 *L'algorithme* PETITS-PORTS-DISTRIBUÉ *construit un arbre couvrant du graphe et fixe une orientation locale compatible avec lui, en utilisant* $2m$ *messages.*

7.3.2 Exploration de graphes dynamiques

Comme il a été prouvé dans le théorème 7.1, l'automate explore périodiquement tout graphe de n sommets en au plus $4n$ étapes, quel que soient la position de départ et l'état initial, pourvu que l'orientation locale soit compatible avec un arbre couvrant quelconque du graphe. En conséquence, l'automate peut être utilisé dans les réseaux dynamiques sous l'unique contrainte que l'orientation locale du réseau reste arbre-orientée après chaque changement de topologie.

Nous considérons des changements du graphe qui le gardent connexe. Un changement du graphe peut être décomposé en une suite de changements élémentaires suivants :

- Ajout d'une nouvelle arête entre deux sommets existants.
- Ajout d'un nouveau sommet, connecté par une nouvelle arête à un sommet existant.
- Suppression d'une arête, sans déconnecter le graphe.
- Suppression d'un sommet de degré 1 et de son unique arête incidente.

Théorème 7.3 *Dans le cas d'une suppression d'une arête appartenant à l'arbre couvrant d'un graphe à n sommets, $\Theta(n)$ modifications de l'orientation locale sont nécessaires et suffisantes pour la maintenir arbre-orientée. Notre algorithme mettant à jour l'orientation locale s'exécute en temps $O(m)$ dans ce cas, où m est le nombre d'arêtes du graphe. Dans tous les autres cas, l'orientation locale peut être mise à jour en temps constant, et donc par un nombre constant de modifications.*

Preuve. L'orientation locale est mise à jour comme suit :

- *Ajout d'une arête.* Cette arête n'est pas ajoutée à l'arbre couvrant. Soit u et v les deux extrémités de la nouvelle arête e et soit d_u et d_v leur nouveau degré respectif. Nous fixons à $d_u - 1$, resp. $d_v - 1$, le numéro de port de l'arête e en u, resp. en v.
- *Ajout d'une feuille.* La nouvelle arête e reliant le nouveau sommet u au sommet v du graphe existant est nécessairement dans l'arbre couvrant. Soit d le degré de v et soit p le plus grand numéro de port en v correspondant à une arête de l'arbre couvrant. Ces deux paramètres sont pris avant la modification. Si $p = d - 1$, le numéro de port de e en v est d. Sinon ($p < d - 1$), l'arête avec le numéro de port $p + 1$ prend maintenant le numéro de port d, et la nouvelle arête e a le numéro de port $p + 1$ en v. On donne le numéro de port 0 à e en u.

- *Suppression d'une arête.* Si l'arête retirée e ne fait pas partie de l'arbre couvrant, alors soient u et v ses deux extrémités. Nous décrivons les modifications de l'orientation locale en u. Les modifications en v sont effectuées de manière similaire. Soit i le numéro de port de e en u. Soit d le degré de u avant la suppression de e. Enfin, soit e' l'arête incidente à u de numéro de port $d-1$. Si $e = e'$ (i.e., $i = d-1$), alors aucun numéro de port n'est modifié en u. Si $e \neq e'$, alors on donne le nouveau numéro de port i à l'arête e' en u.
  Si l'arête e appartient à l'arbre couvrant T, alors T sans l'arête e n'est pas connexe. Puisque nous supposons que le graphe reste connexe, il existe une arête e' dans le nouveau graphe connectant les deux parties de T. Cette arête e' est ajoutée à l'arbre. Certains numéros de port doivent être modifiés pour que l'orientation locale devienne compatible avec le nouvel arbre couvrant. Nous affirmons que seul un nombre constant de numéros de port doivent être modifiés en chaque sommet. Aux extrémités de e et de e', l'ensemble des arêtes de l'arbre incidentes à celles-ci est modifié. Cependant au plus deux arêtes sont concernées. Outre celles-ci, les seules modifications à faire concernent le choix de l'arête incidente ayant le numéro de port 0. Un échange de deux numéros de port est suffisant. En conclusion, au plus un nombre constant de numéros de port est modifié en chaque sommet.
- *Suppression d'une feuille.* Soit v le sommet connecté à la feuille supprimée u. Soit i le numéro de port en v de l'arête menant à u. Soit p le plus grand numéro de port en v d'une arête de T. Soit d le degré de v avant la suppression de e. Finalement, soit e', resp. e'', l'arête incidente à v de numéro de port p, resp. $d-1$. Puisque l'arête e est dans l'arbre T, nous avons $i \leq p \leq d-1$. Nous modifions le numéro de port de e', resp. e'', si et seulement si $i \neq p$, resp. $p \neq d-1$. Si $i \neq p$, nous fixons à i le nouveau numéro de port de e'. Si $p \neq d-1$, nous fixons à p le nouveau numéro de port de e''.

Dans tous les cas, les autres numéros de port dans le graphe restent inchangés.

Il n'est pas toujours possible d'éviter un nombre linéaire de modifications dans le cas de la suppression d'une arête de l'arbre. Par exemple, considérons un cycle C de longueur impaire $2n+1$. Pour simplifier la description, donnons des noms de 1 à $2n+1$ aux sommets. Pour tout sommet $i \leq n$, resp. $i > n$, 0 est le numéro de port menant à $i+1$, resp. $i-1$. Donc l'orientation locale est compatible avec le chemin allant du sommet 1 au sommet $2n+1$. Maintenant, supposons que l'arête $\{n, n+1\}$ est supprimée. Les numéros de port aux sommets n et $n+1$ sont fixés à 0 car ces deux sommets sont maintenant des feuilles. Toutes les arêtes sont nécessairement dans l'arbre couvrant mais l'arête $\{2n+1, 0\}$ a ses deux numéros de port égaux à 1. L'orientation locale n'est pas arbre-orientée. En fait, dans une orientation arbre-orientée, exactement une arête e doit avoir ses deux numéros de port égaux à 0. De plus, pour tout sommet v, excluant les deux extrémités de e, l'arête de numéro de port 0 doit pointer en direction de e, c'est-à-dire l'arête doit appartenir au chemin simple de v à la plus proche extrémité de e. En conséquence, l'orientation locale doit être modifiée en au moins n sommets pour obtenir une orientation locale arbre-orientée. □

7.4 Perspectives

Dans ce chapitre, nous avons prouvé la borne supérieure $4n-2$ sur la période minimale $\pi(n)$ pour l'exploration périodique de graphes par un automate fini. Notre algorithme utilise un arbre couvrant quelconque pour fixer l'orientation locale. L'automate explore cet arbre couvrant plus éventuellement une arête supplémentaire par sommet. Il semble difficile d'éviter les traversées d'arêtes supplémentaires. Donc $4n - O(1)$ pourrait s'avérer optimal pour les approches basées sur les arbres couvrants. Nous conjecturons que cette borne ne peut pas être améliorée, même avec d'autres techniques.

Conjecture. $\pi(n) = 4n - O(1)$.

Par ailleurs, trouver la période minimale pour des robots de mémoire non bornée peut être un problème intéressant.

Problème ouvert. *Quelle est la période minimale $\psi(n)$ telle qu'il existe un algorithme fixant l'orientation locale du graphe et un robot de mémoire non limitée utilisant cette orientation locale pour explorer périodiquement tout graphe de taille n avec une période d'au plus $\psi(n)$?*

Finalement, le problème reste ouvert de savoir si la période $10n$ prouvée dans [DJSS05] peut être améliorée dans le cas des automates sans mémoire, c'est-à-dire lorsque le nombre d'états de l'automate est restreint à 1.

Conclusion et perspectives

Conclusion

Les différents résultats montrés dans cette thèse ont eu pour objectif d'améliorer la compréhension de la complexité en espace de l'exploration de graphes. Il était connu que l'exploration des graphes de taille n nécessitait $\Omega(\log n)$ bits de mémoire pour un automate seul et sans caillou. Très récemment, Reingold a montré une borne supérieure correspondante, caractérisant ainsi complètement les besoins en mémoire de l'exploration des graphes non orientés. Dans cette thèse, nous avons complété ces résultats par une borne optimale en fonction du diamètre D et du degré maximum du graphe d en montrant qu'un DFS utilisant $O(D \log d)$ bits de mémoire est optimal en espace. Nous avons également montré que fournir un caillou à un automate ne lui permettait pas de réduire sa mémoire, en prouvant une borne inférieure de $\Omega(\log n)$ bits pour un automate muni d'un caillou. Concernant l'exploration avec arrêt, nous avons montré une borne supérieure $O(D \log d)$ bits pour un automate muni d'un caillou. En l'absence de borne inférieure correspondante, cela ne permet toutefois pas de conclure sur la complexité en espace de l'exploration avec arrêt dans les graphes non orientés. Enfin, le chapitre 4 nous a permis d'étudier plus finement la relation entre le nombre d'états d'un automate et sa capacité à explorer un grand nombre de graphes. Le problème n'est pas clos pour autant puisque la conjecture $\mathcal{G}_K \neq \mathcal{G}_{K+1}$ que nous avons énoncé au début de ce chapitre n'est ni prouvée ni réfutée à ce jour.

Dans les graphes orientés, où l'exploration est plus délicate à réaliser, pratiquement aucune borne sur la mémoire n'était connu. Nous avons prouvé une borne inférieure de $\Omega(n \log d)$ bits permettant d'affirmer que l'exploration des graphes orientés demande beaucoup plus de mémoire que celle des graphes non orientés. Nous avons également décrit un algorithme de cartographie optimal en espace et utilisant le nombre minimal de caillou, c'est-à-dire un seul caillou. Il reste cependant, en ce qui concerne l'exploration, à clore l'écart entre $\Omega(n \log d)$ et $O(nd(\log n + \log d))$ dans le cas des graphes orientés.

Dans la pratique, par exemple pour la maintenance de réseau, il est très appréciable de disposer d'un algorithme d'exploration très simple. Dans les chapitres 6 et 7, nous avons décrit deux algorithmes permettant soit d'étiqueter les sommets en utilisant au plus trois valeurs différentes, soit de choisir une orientation locale arbre-orientée, de telle façon qu'il existe un automate correspondant ayant un nombre constant d'états et explorant tous les graphes. Il serait très intéressant de concevoir une manière d'aider un automate qui soit

auto-stabilisante. Nous avons apporté un premier élément de réponse dans le chapitre 7 car notre automate est de fait auto-stabilisant. Cependant, l'algorithme fixant l'orientation locale ne l'est pas.

Une importante contribution de cette thèse est, selon notre jugement, l'introduction du concept d'oracle. Ce concept permet de quantifier précisément les informations à connaître sur la topologie pour résoudre un problème donné. La notion d'oracle, loin d'être artificielle, est parfaitement applicable comme nous l'avons montré dans les chapitres 5 et 6. L'oracle peut également être utilisé pour des problèmes fondamentaux de calcul distribué comme nous allons le détailler ci-dessous.

Perspectives

Outre tous les problèmes ouverts directement liés à nos travaux listés précédemment, des problèmes plus généraux concernant l'exploration restent ouverts. Tout d'abord, l'étude de la logique sur les graphes en utilisant la caractérisation par des automates à cailloux imbriqués est un axe de recherche très récent. De ce fait, beaucoup de résultats restent encore à montrer (cf. [EH06]). D'autre part, en complexité, un problème capital restant ouvert est de savoir si $\mathbf{NL} = \mathbf{L}$. La question $\mathbf{SL} = \mathbf{L}$ ayant été résolue en utilisant des résultats d'exploration des graphes non orientés, on peut penser que l'exploration des graphes orientés aura un impact sur cet important problème. Cook et Rackoff [CR80] ont néanmoins montré une borne inférieure de $\Omega(\log^2 n/\log\log n)$ bits sur la mémoire nécessaire à un JAG pour explorer les graphes orientés de taille n. Il semble donc difficile de prouver l'égalité entre **NL** et **L** par ce moyen. Toutefois, le problème de l'exploration pourrait donner des clés de compréhension permettant de mieux saisir la relation entre ces deux classes.

Plus généralement, les travaux de cette thèse s'inscrivent dans une problématique dépassant le cadre de l'exploration, à savoir l'étude du compromis entre la quantité d'informations que possèdent les nœuds ou les agents mobiles et leur capacité à effectuer une tâche. Plus spécifiquement, ce compromis peut être étudié dans deux grands domaines : le calcul mobile (où la tâche est effectuée par des agents mobiles, des robots ou des automates finis) et le calcul distribué.

Parmi les problèmes classiques du calcul mobile, on peut citer (outre l'exploration de graphes) le rendez-vous, la recherche de trou noir, la capture d'intrus, et l'élection d'un leader. Dans la plupart de ces problèmes, la quantité d'informations dont disposent les agents conditionne les performances des solutions que l'on peut y apporter. Par exemple, dans le problème de la recherche de trou noir dans sa version asynchrone, le nombre optimal d'agents dépent de la connaissance qu'ont les agents a priori sur la topologie du réseau. Dans [DFPS02], les auteurs montrent que deux agents permettent de localiser le trou noir si ceux-ci possèdent une carte complète du réseau. A l'inverse, en l'absence d'informations sur la topologie, $d+1$ agents sont nécessaires. Une question naturelle dans notre formalisme de l'oracle est de savoir quelle quantité d'informations doit être donnée à deux agents pour que ceux-ci soient capables de localiser le trou noir. Toujours concernant

le problème de la recherche de trou noir, il est possible d'étudier l'aide apportée au agents selon un point de vue différent. En effet, la formulation classique du problème suppose que les nœuds du réseau disposent de tableaux blancs, que l'on peut voir comme une aide forte pour les agents. Dobrev et al. [DFK06a] ont récemment considéré un modèle plus faible d'assistance aux agents en supposant que ceux-ci disposent simplement de marqueurs de demi-arêtes. Un problème ouvert très intéressant est de savoir si la recherche de trou noir en environnement asynchrone est possible lorsque les agents ne disposent que de cailloux, c'est-à-dire de marqueurs de sommets.

Comme nous l'avons déjà signalé dans l'introduction de la seconde partie de cette thèse, beaucoup de problèmes de calcul distribué dépendent fortement de la connaissance qu'ont les nœuds sur la topologie du réseau. Nous avons par exemple cité le problème de la diffusion d'un message dans les réseaux radio. Ce compromis connaissances/performances concerne également la dissémination d'informations dans les réseaux filaires. Dans le problème du réveil, un nœud source doit envoyer un message à tous les autres nœuds. A la différence du problème de diffusion, le problème du réveil stipule qu'un nœud n'est pas autorisé à envoyer un quelconque message avant d'avoir reçu le message à transmettre. Si tout nœud connaît la topologie du réseau à une distance ρ de lui, alors $\Theta(min\{m, n^{1+\Theta(1)/\rho}\})$ est le nombre minimum de messages de taille bornée permettant le réveil dans un réseau de n sommets et m arêtes [AGPV90]. En collaboration avec Pierre Fraigniaud et Andrzej Pelc, nous avons étudié et comparé la taille de l'oracle nécessaire et suffisante à la résolution des problèmes de diffusion et de réveil en utilisant un nombre de messages linéaire en n. En particulier, nous avons prouvé que le seuil sur la taille de l'oracle pour le problème du réveil est $\Theta(n \log n)$ bits tandis que $O(n)$ bits sont suffisants pour le problème de diffusion. Cette borne supérieure est presque optimale : nous avons prouvé qu'aucun oracle de taille $o(n)$ ne permet la diffusion en un nombre linéaire de messages. Ces résultats ne sont pas détaillés dans cette thèse car la thématique de ce document est l'exploration de graphes. Cependant, ces travaux vont donné lieu à une publication dans les actes de la conférence PODC 2006 [FIP06]. D'autres problèmes peuvent être étudiés dans le cadre de l'oracle, comme le coloriage distribué. Il est également possible, par exemple, de considérer l'élection de leader en modifiant la définition de l'oracle : l'oracle fournit la même chaîne binaire à tous les nœuds (i.e., une chaîne binaire unique est accessible à tous les nœuds). Pour conclure, notre approche de la quantité d'informations nécessaires à la réalisation d'une tâche peut s'appliquer à un large spectre de problèmes et donner des informations précises sur leur difficulté.

Bibliographie

[AAR90] N. Alon, Y. Azar, and Y. Ravid. Universal sequences for complete graphs. *Discrete Applied Mathematics*, 27 :25–28, May 1990.

[ABR79] H. Antelmann, L. Budach, and H.-A. Rollik. On universal traps. *Elektronische Informationsverarbeitung und Kybernetik*, 15(3) :123–131, 1979.

[ABRS99] B. Awerbuch, M. Betke, R. L. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. *Information and Computation*, 152(2) :155–172, 1999.

[AG94] Y. Afek and E. Gafni. Distributed algorithms for unidirectional networks. *SIAM Journal on Computing*, 23(6) :1152–1178, December 1994.

[AGPV90] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish. A trade-off between information and communication in broadcast protocols. *Journal of the ACM*, 37(2) :238–256, April 1990.

[AH00] S. Albers and M. R. Henzinger. Exploring unknown environments. *SIAM Journal on Computing*, 29(4) :1164–1188, August 2000.

[AK98] B. Awerbuch and S. G. Kobourov. Polylogarithmic-overhead piecemeal graph exploration. In *Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT)*, pages 280–286, 1998.

[AKL+79] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 218–223, 1979.

[AKS02] S. Albers, K. Kursawe, and S. Schuierer. Exploring unknown environments with obstacles. *Algorithmica*, 32(1) :123–143, 2002.

[ATSWZ00] R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. An $O(\log^{4/3} n)$ space algorithm for (s,t) connectivity in undirected graphs. *Journal of the ACM*, 47(2) :294–311, March 2000.

[BBFY94] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. Online navigation in a room. *Journal of Algorithms*, 17(3) :319–341, November 1994.

[BBR+99] P. Beame, A. Borodin, P. Raghavan, W. L. Ruzzo, and M. Tompa. A time-space tradeoff for undirected graph traversal by walking automata. *SIAM Journal on Computing*, 28(3) :1051–1072, June 1999.

[BCD+89a] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. Erratum : Two applications of inductive counting for complementation problems. *SIAM Journal on Computing*, 18(6) :1283, December 1989.

[BCD+89b] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal on Computing*, 18(3) :559–578, June 1989.

[BEGT02] S. N. Bhatt, S. Even, D. S. Greenberg, and R. Tayar. Traversing directed eulerian mazes. *Journal of Graph Algorithms and Applications*, 6(2) :157–173, July 2002.

[BFR+02] M. A. Bender, A. Fernández, D. Ron, A. Sahai, and S. P. Vadhan. The power of a pebble : Exploring and mapping directed graphs. *Information and Computation*, 176(1) :1–21, 2002.

[BH67] M. Blum and C. Hewitt. Automata on a 2-dimensional tape. In *Conference Record of 8th Annual Symposium on Switching and Automata Theory (FOCS)*, pages 155–160, 1967.

[BK78] M. Blum and D. Kozen. On the power of the compass (or, why mazes are easier to search than graphs). In *Proceeding of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 132–142, 1978.

[BNBK+89] A. Bar-Noy, A. Borodin, M. Karchmer, N. Linial, and M. Werman. Bounds on universal sequences. *SIAM Journal on Computing*, 18(2) :268–277, April 1989.

[BNS92] L. Babai, N. Nisan, and M. Szegedy. Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs. *Journal of Computer and System Sciences*, 45(2) :204–232, October 1992.

[Bou04a] I. B. Bourdonov. Backtracking problem in the traversal of an unknown directed graph by a finite robot. *Programming and Computer Software*, 30(6) :305–322, 2004.

[Bou04b] I. B. Bourdonov. Traversal of an unknown directed graph by a finite robot. *Programming and Computer Software*, 30(4) :188–203, 2004.

[Bri87] M. F. Bridgland. Universal traversal sequences for paths and cycles. *Journal of Algorithms*, 8(3) :395–404, September 1987.

[BRS95] M. Betke, R. L. Rivest, and M. Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18(2-3) :231–254, 1995.

[BRT92] A. Borodin, W. L. Ruzzo, and M. Tompa. Lower bounds on the length of universal traversal sequences. *Journal of Computer and System Sciences*, 45(2) :180–203, October 1992.

[BS77] M. Blum and W. J. Sakoda. On the capability of finite automata in 2 and 3 dimensional space. In *Proceeding of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 147–161, 1977.

[BS94] M. A. Bender and D. K. Slonim. The power of team exploration : Two robots can learn unlabeled directed graphs. In *Proceeding of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 75–85, 1994.

[BT95] J. F. Buss and M. Tompa. Lower bounds on universal traversal sequences based on chains of length five. *Information and Computation*, 120(2) :326–329, August 1995.

[Bud78] L. Budach. Automata and labyrinths. *Math. Nachrichten*, 86 :195–282, 1978.

[CCvM06] J.-Y. Cai, V. T. Chakaravarthy, and D. van Melkebeek. Time-space tradeoff in derandomizing probabilistic logspace. *Theory of Computing Systems*, 39(1) :189–208, 2006.

[CFI+05] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg. Label-guided graph exploration by a finite automaton. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *LNCS*, pages 335–346, 2005.

[CKMP05] J. Czyzowicz, D. R. Kowalski, E. Markou, and A. Pelc. Searching for a black hole in tree networks. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS 2004)*, volume 3544 of *LNCS*, pages 67–80, 2005.

[CKMP06] J. Czyzowicz, D. R. Kowalski, E. Markou, and A. Pelc. Complexity of searching for a black hole. *Fundamenta Informaticae*, 72 :1–14, 2006.

[CMS03] A. E. F. Clementi, A. Monti, and R. Silvestri. Distributed broadcast in radio networks of unknown topology. *Theoretical Computer Science*, 1-3(302) :337–364, June 2003.

[Coy77] W. Coy. Automata in labyrinths. In *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT)*, volume 56 of *LNCS*, pages 65–71, 1977.

[CR80] S. A. Cook and C. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM Journal on Computing*, 9(3) :636–652, August 1980.

[CR03] A. Czumaj and W. Rytter. Broadcasting algorithms in radio networks with unknown topology. In *Proceeding of the 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 492–501, 2003.

[CRR+97] A. K. Chandra, P. Raghavan, W. L. Ruzzo, R. Smolensky, and P. Tiwari. The electrical resistance of a graph captures its commute and cover times. *Computational Complexity*, 6(4) :312–340, 1997.

[CS02] B. Cubaleska and M. Schneider. A method for protecting mobile agents against denial of service attacks. In *Proceedings of the 6th International Workshop on Cooperative Information Agents (CIA)*, volume 2446 of *LNCS*, pages 297–311, 2002.

[Del97] M. Delorme. Automates à galets : un état de l'art. Technical Report 97-23, ENS Lyon, September 2 1997.

[DFK06a] S. Dobrev, P. Flocchini, and R. Kralovic. Exploring an unknown graph to locate a black hole using tokens. In *Proceedings of the 4th IFIP International Conference on Theoretical Computer Science*, to appear, 2006.

[DFK+06b] S. Dobrev, P. Flocchini, R. Kralovic, G. Prencipe, P. Ruzicka, and N. Santoro. Black hole in common interconnection networks. *Networks*, 47(2) :61–71, 2006.

[DFKP04] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory. *Journal of Algorithms*, 51(1) :38–63, 2004.

[DFNS05] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Distributed exploration of an unknown graph. In *Proceedings of the 12th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 3499 of *LNCS*, pages 99–114, 2005.

[DFPS01] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile search for a black hole in an anonymous ring. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC)*, volume 2180 of *LNCS*, pages 166–179, 2001.

[DFPS02] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Searching for a black hole in arbitrary networks : optimal mobile agent protocols. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 153–161, 2002.

[DFS04] S. Dobrev, P. Flocchini, and N. Santoro. Improved bounds for optimal black hole search with a network map. In *Proceedings of the 11th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 3104 of *LNCS*, pages 111–122, 2004.

[DJMW91] G. Dudek, M. R. M. Jenkin, E. E. Milios, and D. Wilkes. Robotic exploration as graph construction. *IEEE Transactions on Robotics and Automation*, 7(6) :859–865, December 1991.

[DJMW97] G. Dudek, M. R. M. Jenkin, E. E. Milios, and D. Wilkes. Map validation and robot self-location in a graph-like world. *Robotics and Autonomous Systems*, 22(2) :159–178, November 1997.

[DJSS05] S. Dobrev, J. Jansson, K. Sadakane, and W.-K. Sung. Finding short right-hand-on-the-wall walks in graphs. In *Proceedings of the 12th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 3499 of *LNCS*, pages 127–139, 2005.

[DKK01] C. A. Duncan, S. G. Kobourov, and V. S. A. Kumar. Optimal constrained graph exploration. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 807–814, 2001.

[DKP98] X. Deng, T. Kameda, and C. H. Papadimitriou. How to learn an unknown environment I : The rectilinear case. *Journal of the ACM*, 45(2) :215–245, March 1998.

[DM96] X. Deng and A. Mirzaian. Competitive robot mapping with homogeneous markers. *IEEE Transactions on Robotics and Automation*, 12(4) :532–542, August 1996.

[DM02] M. Delorme and J. Mazoyer. Pebble automata. figures families recognition and universality. *Fundamenta Informaticae*, 52(1-3) :81–132, 2002.

[DMM01] X. Deng, E. E. Milios, and A. Mirzaian. Robot map verification of a graph world. *Journal of Combinatorial Optimization*, 5(4) :383–395, December 2001.

[Döp71a] K. Döpp. Automaten in labyrinthen I. *Elektronische Informationsverarbeitung und Kybernetik*, 7(2) :79–94, 1971.

[Döp71b] K. Döpp. Automaten in labyrinthen II. *Elektronische Informationsverarbeitung und Kybernetik*, 7(3) :167–189, 1971.

[DP99] X. Deng and C. H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3) :265–297, 1999.

[DP04] A. Dessmark and A. Pelc. Optimal graph exploration without good maps. *Theoretical Computer Science*, 326(1-3) :343–362, October 2004.

[EH06] J. Engelfriet and H. J. Hoogeboom. Nested pebbles and transitive closure. In *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3884 of *LNCS*, pages 477–488, 2006.

[EJ73] J. Edmonds and E. L. Johnson. Matching, euler tours and the chinese postman. *Mathematical Programming*, 5 :88–124, 1973.

[FGKP04] P. Fraigniaud, L. Gasieniec, D. R. Kowalski, and A. Pelc. Collective tree exploration. In *Proceedings of the 6th Latin American Theoretical Informatics Symposium (LATIN)*, volume 2976 of *LNCS*, pages 141–151, 2004.

[FI04] P. Fraigniaud and D. Ilcinkas. Digraphs exploration with little memory. In *Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 2996 of *LNCS*, pages 246–257, 2004.

[FIP+04] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. In *Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 3153 of *LNCS*, pages 451–462, 2004.

[FIP+05] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. *Theoretical Computer Science*, 345(2-3) :331–344, November 2005.

[FIP06] P. Fraigniaud, D. Ilcinkas, and A. Pelc. Oracle size : a new measure of difficulty for communication tasks. In *Proceedings of the 25th Annual ACM*

Symposium on Principles of Distributed Computing (PODC), pages 179–187, 2006.

[FIRT05] P. Fraigniaud, D. Ilcinkas, S. Rajsbaum, and S. Tixeuil. Space lower bounds for graph exploration via reduced automata. In *Proceedings of the 12th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 3499 of *LNCS*, pages 140–154, 2005.

[FIRT06] P. Fraigniaud, D. Ilcinkas, S. Rajsbaum, and S. Tixeuil. The reduced automata technique for graph exploration space lower bounds. In *Essays in Memory of Shimon Even*, volume 3895 of *LNCS*, pages 1–26, 2006.

[FR03] F. E. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3) :121–163, September 2003.

[FT05] R. Fleischer and G. Trippen. Exploring an unknown graph efficiently. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 11–22, 2005.

[GPX05] L. Gasieniec, D. Peleg, and Q. Xin. Faster communication in known topology radio networks. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–137, 2005.

[Hem87] A. Hemmerling. Three-dimensional traps and barrages for cooperating automata (extended abstract). In *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT)*, volume 278 of *LNCS*, pages 197–203, 1987.

[Hem89] A. Hemmerling. *Labyrinth Problems : Labyrinth-Searching Abilities of Automata*, volume 114 of *Teubner-Texte zur Mathematik*. B. G. Teubner Verlagsgesellschaft, Leipzig, 1989.

[HIKK01] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem. *SIAM Journal on Computing*, 31(2) :577–600, April 2001.

[Hof81] F. Hoffmann. One pebble does not suffice to search plane labyrinths. In *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT)*, volume 117 of *LNCS*, pages 433–444, 1981.

[Hof85] F. Hoffmann. Four pebbles don't suffice to search planar infinite labyrinths. In *Colloquium on the Theory of Algorithms*, pages 191–206. North-Holland, 1985.

[HW93] S. Hoory and A. Wigderson. Universal traversal sequences for expander graphs. *Information Processing Letters*, 46(2) :67–69, 1993.

[Ilc06] D. Ilcinkas. Setting port numbers for fast graph exploration. In *Proceedings of the 13th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 4056 of *LNCS*, pages 59–69, 2006.

[Imm87] N. Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4) :760–778, August 1987.

[INW94] R. Impagliazzo, N. Nisan, and A. Wigderson. Pseudorandomness for network algorithms. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 356–364, 1994.

[Isl01] V. Isler. Theoretical robot exploration. Technical report, University of Pennsylvania, February 28 2001.

[Ist88] S. Istrail. Polynomial universal traversing sequences for cycles are constructible (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 491–503, 1988.

[Kil91] G. Kilibarda. A new proof of the Budach-Podkolzin theorem. *Diskretnaya Matematika*, 3(3) :135–146, 1991.

[Kil93] G. Kilibarda. On the minimum universal collectives of automata for plane labyrinths. *Discrete Mathematics and Applications*, 3(6) :555–586, 1993.

[Kle94] J. M. Kleinberg. On-line search in a simple polygon. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 8–15, 1994.

[KLNS89] J. D. Kahn, N. Linial, N. Nisan, and M. E. Saks. On the cover time of random walks on graphs. *Journal of Theoretical Probability*, 2(1) :121–128, January 1989.

[KMRS05a] R. Klasing, E. Markou, T. Radzik, and F. Sarracco. Approximation bounds for black hole search problems. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS)*, LNCS, page to appear, 2005.

[KMRS05b] R. Klasing, E. Markou, T. Radzik, and F. Sarracco. Hardness and approximation results for black hole search in arbitrary graphs. In *Proceedings of the 12th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, volume 3499 of *LNCS*, pages 200–215, 2005.

[Kou02] M. Koucký. Universal traversal sequences with backtracking. *Journal of Computer and System Sciences*, 65(4) :717–726, 2002.

[Kou03a] M. Koucký. Log-space constructible universal traversal sequences for cycles of length $O(n^{4.03})$. *Theoretical Computer Science*, 296(1) :117–144, March 2003.

[Kou03b] M. Koucký. On traversal and exploration sequences. Technical Report 2003-41, DIMACS, November 28 2003.

[Koz79] D. Kozen. Automata and planar graphs. In *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT)*, pages 243–254, 1979.

[KP] D. Kowalski and A. Pelc. Optimal deterministic broadcasting in known topology radio networks. manuscript.

[KPS88] H. J. Karloff, R. Paturi, and J. Simon. Universal traversal sequences of length $n^{O(logn)}$ for cliques. *Information Processing Letters*, 28(5) :241–243, August 1988.

[KPU85] V. B. Kudryavtsev, A. S. Podkolzin, and S. Ushchumlich. *Introduction to the Theory of Abstract Automata*. Moscow State Univ. Press, Moscow, 1985.

[Kut88] S. Kutten. Stepwise construction of an efficient distributed traversing algorithm for general strongly connected directed networks or : Traversing one way streets with no map. In *Proceedings of the 9th International Conference on Computer Communication (ICCC)*, pages 446–452, 1988.

[Kwe97] S. Kwek. On a simple depth-first search strategy for exploring unknown graphs. In *Proceedings of the 5th International Workshop on Algorithms and Data Structures (WADS)*, volume 1272 of *LNCS*, pages 345–353, 1997.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures : Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1992.

[LP82] H. R. Lewis and C. H. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19(2) :161–187, August 1982.

[Lyn89] N. A. Lynch. A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–28, 1989.

[Moo56] E. F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 129–153. Princeton University Press, New Jersey, 1956.

[MP06] E. Markou and A. Pelc. Efficient exploration of faulty trees. *Theory of Computing Systems*, 2006.

[Mül71] H. Müller. Endliche automaten und labyrinthe. *Elektronische Informationsverarbeitung und Kybernetik*, 7(4) :261–264, 1971.

[Mül79] H. Müller. Automata catching labyrinths with at most three components. *Elektronische Informationsverarbeitung und Kybernetik*, 15(1-2) :3–9, 1979.

[Nis92] N. Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4) :449–461, December 1992.

[NSW92] N. Nisan, E. Szemerédi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5}n)$ space. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 24–29, 1992.

[NTS95] N. Nisan and A. Ta-Shma. Symmetric logspace is closed under complement. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 140–146, 1995.

[PK98] V. A. Pham and A. Karmouch. Mobile software agents : An overview. *IEEE Communications Magazine*, 36(7) :26–37, July 1998.

[PY91] C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1) :127–150, July 1991.

[Rab67] M. O. Rabin. Maze threading automata. Seminar Talk presented at the University of California at Berkeley, October 1967.

[RDM00] I. M. Rekleitis, G. Dudek, and E. E. Milios. Graph-based exploration using multiple robots. In *Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 241–250, 2000.

[RDM01] I. M. Rekleitis, G. Dudek, and E. E. Milios. Multi-robot collaboration for robust exploration. *Annals of Mathematics and Artificial Intelligence*, 31(1-4) :7–40, 2001.

[Rei05] O. Reingold. Undirected st-connectivity in log-space. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 376–385, 2005.

[Rek03] I. M. Rekleitis. *Cooperative Localization and Multi-Robot Exploration.* PhD thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, February 2003.

[RKSI93] N. S. V. Rao, S. Kareti, W. Shi, and S. S. Iyengar. Robot navigation in unknown terrains : Introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Oak Ridge National Laboratory, January 29 1993.

[Rol80] H.-A. Rollik. Automaten in planaren Graphen. *Acta Informatica*, 13(3) :287–298, March 1980.

[Sak96] M. E. Saks. Randomization and derandomization in space-bounded computation. In *Proceedings of the 11th Annual IEEE Conference on Computational Complexity*, pages 128–149, 1996.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2) :177–192, April 1970.

[Sha51] C. E. Shannon. Presentation of a maze-solving machine. In *8th Conf. of the Josiah Macy Jr. Found.(Cybernetics)*, pages 173–180, 1951.

[SZ99] M. E. Saks and S. Zhou. $BP_H\text{Space}(S) \subseteq \text{DSPACE}(S^{3/2})$. *Journal of Computer and System Sciences*, 58(2) :376–403, April 1999.

[Sze82] A. Szepietowski. A finite 5-pebble-automaton can search every maze. *Information Processing Letters*, 15(5) :199–204, December 1982.

[Tom92] M. Tompa. Lower bounds on universal traversal sequences for cycles and other low degree graphs. *SIAM Journal on Computing*, 21(6) :1153–1160, December 1992.

[Tri05] V. Trifonov. An O(log n log log n) space algorithm for undirected st-connectivity. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 626–633, 2005.

[YK96] M. Yamashita and T. Kameda. Computing on anonymous networks : Part I-characterizing the solvable cases. *IEEE Transactions on Parallel and Distributed Systems*, 7(1) :69–89, January 1996.

Complexité en espace de l'exploration de graphes

Résumé : Le problème de l'exploration de graphes trouve ses motivations en informatique fondamentale, notamment en logique et en théorie de la complexité. Il possède également de nombreuses applications en robotique. Quel que soit le cadre, la quantité de mémoire utilisée par l'entité mobile (robot, automate fini, etc.) effectuant l'exploration est un des paramètres importants à considérer. Dans cette thèse, nous étudions en détail la complexité en espace de l'exploration de graphes, à travers différents modèles. Nous distinguons principalement deux cadres d'études.

Dans la première partie de la thèse, nous nous attachons à l'étude de l'exploration "sans assistance", c'est-à-dire lorsque l'entité mobile ne possède aucune information sur le graphe à explorer. Dans ce contexte, nous prouvons plusieurs bornes inférieures et supérieures sur la quantité de mémoire nécessaire et suffisante à l'entité pour explorer tous les graphes. En particulier, nous montrons que l'algorithme très simple de parcours en profondeur d'abord est optimal en mémoire lorsque la complexité est exprimée en fonction du degré et du diamètre.

Dans la seconde partie de la thèse, nous nous attachons à l'étude de l'exploration "avec assistance". Nous considérons un modèle supposant l'existence d'un oracle ayant une connaissance exhaustive du graphe exploré, et capable d'aider l'entité mobile en lui fournissant de l'information. Nous nous intéressons ainsi à la quantité minimale d'information (mesurée en nombre de bits) que l'oracle doit fournir à l'entité pour permettre l'exploration. Cette information peut être soit donnée directement à l'entité, soit codée sur les sommets du graphes.
Mots-clés : exploration, graphes, automate fini, complexité en espace, oracle

---

Space complexity of graph exploration

Abstract: Graph Exploration is a problem motivated by several aspects of theoretical computer science, including logic and computational complexity. It has also numerous applications in robotics. In all these contexts, the amount of memory used by the mobile entity (robot, finite automaton, etc.) performing exploration is a key parameter. In this thesis, we study in depth the space complexity of graph exploration, in two different frameworks.

In the first part of the thesis, we focus on exploration "without assistance", i.e., the case where the mobile entity does not possess any a priori information about the explored graph. In this context, we prove several lower and upper bounds on the amount of memory necessary and sufficient to the mobile entity for performing exploration of all graphs. In particular, we show that the depth-first search algorithm is optimal in space when the complexity is measured in term of degree and diameter.

In the second part of the thesis, we focus on exploration "with assistance", by assuming that an oracle entirely aware of the input graph can help the mobile entity, by providing some information about it. This information is measured by its bit-space complexity, and we are interested in the minimal number of bits that the oracle must provide so that the mobile entity can perform exploration. The information can be either given directly to the entity, or encoded on the vertices of the graph.
Keywords: exploration, graphs, finite automaton, space complexity, oracle