Algorithmes distribués pour l'optimisation de déploiement des microrobots MEMS

Hicham Lakhlef

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Titre

Algorithmes distribués pour l'optimisation du déploiement des microrobots MEMS

Hicham Lakhlef

N° | X | X | X

THÈSE présentée par

Hicham Lakhlef

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Titre

Algorithmes distribués pour l'optimisation du déploiement des microrobots MEMS

Soutenue publiquement le 24 Novembre 2014 devant le Jury composé de :

Pascal Felber	Rapporteur	Professeur à l'Université de Neuchâtel
Nathalie Mitton	Rapporteur	Chargée de recherche habilitée à diriger des recherches, INRIA Lille-Nord Europe
Raphaël Couturier	Examinateur	Professeur à l'Université de Franche-Comté
Pascale Minet	Examinateur	Chargée de recherche habilitée à diriger des recherches, INRIA Paris Rocquencourt
Julien Bourgeois	Directeur de thèse	Professeur à l'Université de Franche-Comté
Hakim Mabed	Co-encadrant de thèse	Maître de conférences à l'Université de Franche-Comté

Université de Franche-Comté **UFR Sciences, Techniques et Gestion de l'Industrie**

FEMTO-ST/DISC **Doctoral school SPIM**

Algorithmes distribués pour l'optimisation du déploiement des microrobots MEMS

Thèse de Doctorat

sera défendue publiquement le 24 Novembre 2014

pour obtenir

Grade de Docteur de l'Université de Franche-Comté

École Doctorale SPIM - Spécialité Informatique

par

Hicham Lakhlef

Composition du jury

Directeur de thèse :	Julien Bourgeois	Professeur à l'université de Franche-Comté
Rapporteurs :	Pascal Felber	Professeur à l'Université de Neuchâtel
	Nathalie Mitton	Chargée de recherche HDR, INRIA Lille-Nord Europe
Examinateurs :	Raphaël Couturier	Professeur à l'université de Franche-Comté
	Pascale Minet	Chargée de recherche HDR, INRIA Paris Rocquencourt
Encadreur :	Hakim Mabed	Maîtres de conférences à l'université de Franche-Comté

Remerciement

Je tiens tout d'abord à remercier le directeur de cette thèse, **M. Julien Bourgeois**, professeur à l'université de Franche-Comté, qui m'a confié cette thèse, puis je le remercie pour m'avoir guidé, encouragé, conseillé, fait beaucoup voyager pendant trois ans tout en me laissant une grande liberté et en me faisant l'honneur de me déléguer plusieurs responsabilités dont j'espère avoir été à la hauteur.

J'adresse de chaleureux remerciements à mon co-encadrant de thèse, **M. Hakim Mabed**, Maitre de conférences à l'université de Franche-Comté, pour son attention de tout instant sur mes travaux, pour ses conseils avisés et son écoute qui ont été prépondérants pour la bonne réussite de cette thèse. Son énergie et sa confiance ont été des éléments moteurs pour moi. J'ai pris un grand plaisir à travailler avec lui.

Je voudrais remercier les rapporteurs de cette thèse **M. Pascal Felber**, Professeur des Universités à l'université de Neuchâtel, et **Mme. Nathalie Mitton,** Chargée de recherche habilitée à diriger des recherches, INRIA Lille-Nord Europe, pour l'intérêt qu'ils ont porté à mon travail.

J'associe à ces remerciements **M. Raphaël Couturier**, professeur à l'université de Franche-Comté et **Mme. Pascale Minet**, Chargée de recherche habilité à diriger des recherches, INRIA Paris Rocquencourt, pour avoir accepté d'examiner mon travail.

Je tiens à remercier tous les membres de l'équipe OMNI, pour leur aide et leur bonne humeur. Nous avons partagé de bons moments.

J'exprime ma plus profonde reconnaissance à ma mère et mon père pour leur soutien sans faille tout au long de ces années. Je tiens à leur dire à quel point leur soutien affectif a été important à mes yeux, ce travail a pu être mené à son terme grâce à leur soutien. Ce travail leur est légitimement dédié.

Résumé

Les microrobots MEMS sont des éléments miniaturisés qui peuvent capter et agir sur l'environnement. Leur taille est de l'ordre du millimètre et ils ont une faible capacité de mémoire et une capacité énergétique limitée. Les microrobots MEMS continuent d'accroître leur présence dans notre vie quotidienne. En effet, ils peuvent effectuer plusieurs missions et tâches dans une large gamme d'applications telles que la localisation d'odeur, la lutte contre les incendies, le service médical, la surveillance, le sauvetage et la sécurité. Pour faire ces taches et missions, ils doivent appliquer des protocoles de redéploiement afin de s'adapter aux conditions du travail. Ces algorithmes doivent être efficaces, évolutifs, robustes et ils doivent utiliser de préférence des informations locales. Le redéploiement pour les microrobots MEMS mobiles nécessite actuellement un système de positionnement et une carte (positions prédéfinies) de la forme cible. La solution traditionnelle de positionnement comme l'utilisation d'un GPS consommerait trop d'énergie. De plus, l'utilisation de solutions de positionnement algorithmique avec les techniques de multilatération pose toujours des problèmes à cause des erreurs dans les coordonnées obtenues. Dans la littérature, si nous voulons une auto-reconfiguration de microrobots vers une forme cible constituée de P positions, chaque microrobot doit avoir une capacité mémoire de P positions pour les sauvegarder. Par conséquent, si P est de l'ordre de milliers ou de millions, chaque nœud devra avoir une capacité de mémoire de positions en milliers ou millions. Par conséquent, ces algorithmes ne sont pas extensibles ou évolutifs. Dans cette thèse, on propose des protocoles de reconfiguration où les nœuds ne sont pas conscients de leurs positions dans le plan et n'enregistrent aucune position de la forme cible. En d'autres termes, les nœuds ne stockent pas au départ les coordonnées qui construisent la forme cible. Par conséquent, l'utilisation de mémoire pour chaque nœud est réduite à une complexité constante. L'objectif des algorithmes distribués proposés est d'optimiser la topologie logique du réseau des microrobots afin de chercher une meilleure complexité pour l'échange de message et une communication peu coûteuse. Ces solutions sont complètement distribués. On montre pour la reconfiguration d'une chaîne à un carré comment gérer la dynamicité du réseau pour sauvegarder l'énergie, on étudie comment utiliser le parallélisme de mouvements pour optimiser le temps d'exécution et le nombre de mouvements. Ainsi, on propose une autre solution où la topologie physique initiale peut être n'importe quelle configuration initiale. Avec ces solutions, les nœuds peuvent exécuter l'algorithme indépendamment du lieu où ils sont déployés, parce que l'algorithme est indépendant de la carte de la forme cible. En outre, ces solutions cherchent à atteindre la forme de la cible avec une quantité minimale de mouvement.

Mots-clés: Microrobots MEMS, Auto-reconfiguration, Redéploiement, Algorithmes distribués, Algorithmes parallèles, Topologie logique, Énergie, Mobilité

Abstract

MEMS microrobots are miniaturized elements that can capture and act on the environment. They have a small size, low memory capacity and limited energy capacity. These inexpensive devices can perform several missions and tasks in a wide range of applications such as locating odor, fighting against fires, medical service, surveillance, search, rescue and safety. To do these tasks and missions, they have to carry out protocols of redeployment to adapt to the working conditions. These algorithms should be efficient, scalable, robust and should only use local information. Redeployment for mobile MEMS microrobots currently requires a positioning system and a map (predefined positions) of the target shape. Traditional positioning solutions such as using GPS consumes a lot of energy and it is no applicable in the micro scale. Also, the use of an algorithmic solution positioning with multilateration techniques causes problems due to errors in the coordinates obtained. In the literature works, if we want a microrobots self-reconfiguring to a target shape consisting of P positions, each microrobot must have a storage capacity of at least P positions to save them. Therefore, if P equals to thousands or millions, every node must have a storage capacity of thousands or millions of positions. However, these algorithms are not scalable. In this thesis, we propose protocols of self-reconfiguration where nodes are not aware of their position in the plane and do not record the positions of the target shape. Therefore, the memory space required for each node is significantly reduced at a constant complexity. The purpose of these distributed algorithms is to optimize the logical topology of the network of mobile MEMS microrobots to seek a better complexity for message exchange and inexpensive communication. In this work, we show for the reconfiguration of a chain into a square, how to handle the dynamicity of the network to save energy, and we study how to use parallelism in motion to optimize the execution time and the number of movements. Furthermore, another solution is proposed where the initial physical topology may be any connected configuration. With these solutions the nodes can execute the algorithm regardless of where they are deployed, because the algorithm is independent of the map of the target shape. Furthermore, these solutions seek to achieve the shape of the target with a minimum amount of movement.

Keywords: MEMS Microrobots, Self-reconfiguration, Redeployment, Distributed algorithms, Parallel algorithms, Logical topology, Energy, Mobility

TABLE DES MATIÈRES

TABLE DES FIGURES

Première partie

Généralités

CHAPITRE 1

INTRODUCTION

1.1 Énoncé du problème

Les robots sont conçus comme des outils automatiques pour effectuer des missions difficiles, dangereuses ou répétitives depuis les années 1950. Les axes de recherches principaux ont été consacrés au développement de robots avec des capacités de haute précision et/ou de grandes vitesses de contrôle dans le but de réaliser des travaux répétitifs, notamment dans les applications industrielles.

À la fin des années 1970, grâce au développement conjoint et à l'expansion des micro-ordinateurs et l'intelligence artificielle des robots ont été étudiés aussi bien en laboratoire universitaire que dans les centres de recherche et développement d'entreprises industrielles. Dans les années 1980, les robots ont été appliqués à différentes tâches et ont été utilisés d'une façon plus pratique.

Le développement des robots intelligents et l'extension de leurs domaines d'application ont donné naissance aux robots modulaires. Un réseau de microrobots modulaires désigne un ensemble de robots symétriques (identiques et interchangeables) capables de se rattacher et se détacher des autres robots pour atteindre des configurations différentes.

Depuis leur introduction, les systèmes robotiques modulaires ont été envisagés comme un domaine de recherche très prometteur.

Des recherches qui ont pour but la conception, la construction et le contrôle d'ensembles de plusieurs unités ou modules relativement autonomes qui dans un esprit de collaboration permettent d'effectuer des tâches collectives. Les robots modulaires ont le potentiel d'être plus faciles à entretenir et à réparer. Si un module est défectueux, il peut être identifié et remplacé en un temps relativement court. Tandis qu'une panne sur un robot non-modulaire nécessiterait son remplacement total ou sa réparation.

Dans la dernière décennie du 20éme siècle, des progrès récents dans les systèmes micro-électromécaniques (Micro Electro Mechanical Systems, MEMS) ont permis la conception de très petits robots modulaires, appelés microrobots MEMS. Ce sont des éléments mécaniques et électromécaniques miniaturisés fabriqués en utilisant des techniques de micro fabrication basées sur les microcontrôleurs et les systèmes mécaniques miniatures en silicium. Les dimensions

physiques critiques des dispositifs MEMS peuvent varier de moins d'un micron à plusieurs millimètres. De même, les types de dispositifs MEMS peuvent varier de structures relativement simples sans éléments mobiles, à des systèmes électromécaniques très complexes avec de multiples éléments mobiles sous le contrôle de la microélectronique intégrée. En raison de leur petite taille, les microrobots sont potentiellement peu chers, et pourraient être utilisés dans un grand nombre d'applications pour explorer des environnements qui sont trop petits ou trop dangereux pour les personnes et pour les grands robots. Globalement, les éléments fonctionnels des microrobots MEMS sont des composants miniaturisés, des capteurs, des actionneurs et de la microélectronique. Les micro-capteurs et micro-actionneurs sont des dispositifs qui permettent de convertir une quantité d'énergie en une autre. Dans le cas des micro-capteurs, le dispositif détecte une mesure physique en entrée telle que un mouvement, un signal électrique, une lumière ou une source thermique, tandis q'un micro-actionneur permet d'agir sur un périphérique externe tel que le mouvement ou le son. Grâce aux micro-capteurs et micro-actionneurs, les microrobots MEMS peuvent capter et agir sur l'environnement mais reste que leurs capacités de mémoire et d'énergie sont très limitées. Les microrobots modulaires sont souvent classés comme homogènes ou hétérogènes, selon que leurs modules sont similaires ou différents, bien que tous soient structurellement égaux. Certains modules peuvent intégrer ou réaliser des fonctions spéciales avec des équipements spéciaux tels que des pinces, caméras, antennes, etc. Selon l'autonomie de locomotion des composants du microrobot deux types de systèmes modulaires peuvent être identifiés. Dans le premier, le microrobot est complétement libre de se mouvoir dès lors que les contraintes d'espace (collision) sont respectées. Tandis que dans l'autre type de microrobots, le mouvement d'un microrobot est réalisé par la coopération des unités, sur la base du mouvement des articulations d'accueil et des liens entre microrobots.

Les microrobots ont des caractéristiques qui les rendent intéressants et qui leur permettent de faire des taches difficiles de façon efficace. Les caractéristiques suivantes ont été soulignées dans la littérature :

Polyvalence. Les modules peuvent être combinés de différentes manières permettant, au même système robotisé, d'effectuer une grande variété de tâches

Adaptabilité. Alors que le robot auto-reconfigurable accomplit sa tâche, il peut changer de forme physique pour s'adapter aux changements de l'environnement

Robustesse. Les robots auto-reconfigurables sont fabriqués à partir de nombreux modules identiques et donc si un module est défaillant, il peut être remplacé par un autre

L'échelle de l'extensibilité. La taille du robot modulaire peut être augmentée ou diminuée par l'ajout ou la suppression de modules

Coût de production. Les microrobots MEMS pourraient être produits en masse, rendant ainsi le coût d'un module individuel faible par rapport à sa complexité.

Les microrobots modulaires ont l'avantage d'être plus polyvalents, car ils peuvent se reconfigurer pour s'adapter à de nouveaux environnements et à de nouvelles tâches. Ils sont aussi plus robustes, car ils sont interchangeables ce qui les rend tolérant aux fautes. Ils sont également potentiellement moins cher qu'un robot monolithique, car leurs unités peuvent être réutilisés et, en principe, produites en grande série.

En conséquence, ils sont censés être utiles dans la construction de structures d'urgence, la réparation de machines inaccessibles, missions spatiales, et même dans la vie quotidienne

actuelle.

En contrepartie de cette flexibilité, il est difficile de concevoir des mécanismes de contrôle des systèmes de microrobots modulaires lors de la réalisation d'une tâche donnée.

Selon la répartition des modules dans l'espace lors du déploiement, les systèmes robotiques modulaires peuvent être organisés en réseau, chaîne ou architectures hybrides. Les robots modulaires à base de treillis peuvent être de forme hexagonale, triangulaire, sphérique, carrée ou cubique. Pour tous ces systèmes robotiques, des algorithmes d'actionnement ont été proposés, dans la littérature, avec des objectifs différents : la locomotion, la reconfiguration, l'autoréparation, etc. Bien que beaucoup d'entre eux soient centralisés, il est nécessaire d'utiliser un contrôle décentralisé, quand le nombre de modules augmente. Des algorithmes distribués ont été conçus pour différents systèmes modulaire, et plus particulièrement pour les robots modulaires à base de treillis tels que Proteo [102], Fracta [70], Cristalline [79] et Telecube [92], et les robots modulaires à grande échelle comme les Catoms du projet Claytronics [36].

L'auto-reconfiguration est un processus centralisé ou distribué qui permet de contrôler de façon dynamique l'ensemble des microrobots constituant au début une forme physique donnée pour arriver à une autre forme cible qui satisfait les exigences de la tâche donnée. Grâce aux algorithmes de reconfiguration et de redéploiement, les microrobots autonomes peuvent changer leur forme et s'adapter à un environnement de travail spécifique. Par conséquent, le changement de forme, dans ces systèmes composites est envisagé comme un moyen pour augmenter la versatilité de l'ensemble et d'accomplir diverses tâches, telles que la construction de ponts, un soutien structurel, la récupération de satellites, et l'excision d'une tumeur.

Pour achever la reconfiguration dans un système centralisé, les microrobots sont contrôlés par un module central différent des autres modules. Dans les systèmes distribués, un contrôleur s'exécute sur chaque module pour achever la reconfiguration. Les algorithmes de reconfiguration centralisés sont traditionnellement utilisés pour contrôler des robots, mais ne sont pas adaptés pour les microrobots modulaires. La raison est que si des changements surviennent sur un des modules ou sur l'environnement, la procédure de planification doit s'exécuter sur le module central avant que le robot modifié ne puisse continuer. Concrètement, cela signifie que les systèmes de contrôle centralisé manquent de robustesse et d'adaptabilité face aux modifications de l'environnement et au passage à l'échelle (extensibilité). En effet, puisque le module centralisé est responsable du contrôle des autres modules, il peut être surchargé s'il doit traiter un nombre croissant de modules. Cela implique que l'efficacité du système centralisé va diminuer et qu'il s'adapte mal à une augmentation du nombre de modules. Enfin, si le module contrôleur tombe en panne ou s'il commet une erreur, le processus de reconfiguration est alors compromis. Cela signifie que les systèmes de contrôle centralisés ne sont pas tolérants aux fautes et pannes.

Le contrôle distribué est utilisé pour pallier les inconvénients du contrôle centralisé. Cependant, il est difficile d'accomplir un objectif global de reconfiguration en utilisant seulement des informations locales. Si des informations globales sont nécessaires sur les différents modules, le système devient moins robuste et évolutif car les modules peuvent perdre beaucoup de temps à attendre des informations globales.

Un algorithme de reconfiguration doit être robuste à la défaillance des modules et à l'échec des communications. Il doit être extensible, en tenant compte de l'impact du nombre de modules sur la taille de la mémoire locale nécessaire à chaque module et de l'énergie consomé par chaque

microrobot. Ainsi,un algorithme de reconfiguration est jugé efficace si il utilise peu de mémoire, peu de messages, et peu d'énergie.

1.2 Contribution de la thèse

Cette thèse propose une conception et une implémentation d'algorithmes distribués efficaces pour le redéploiement des microrobots MEMS. L'objectif de ces algorithmes est l'optimisation de la topologie physique du réseau de microrobots pour améliorer la communication. La caractéristique commune de ces algorithmes est leur extensibilité puisqu'ils n'utilisent pas de carte de la forme cible (les positions prédéfinies de la forme cible). La reconfiguration des microrobots est obtenue grâce à des mécanismes de coopération entre les microrobots. La contribution de cette thèse comprend les éléments suivants :

- **Protocoles efficaces pour l'auto-reconfiguration**. Dans cette solution, j'ai proposé deux algorithmes distribués de redéploiement à partir d'une chaîne droite vers un carré. Ces algorithmes utilisent une complexité constante de mémoire (chaque algorithme a besoin seulement de trois états pour achever la reconfiguration). Le premier algorithme garantit une connexité non-continue du réseau (le réseau est initialement connexe et redevient connexe) avec $n - \sqrt{n}$ mouvements dans le pire cas où n est le nombre de modules. L'algorithme présente deux avantages supplémentaires : le non recours à l'échange de messages ni à une phase de pré-traitement. Le deuxième algorithme garantit une connexité continue du réseau tout au long de l'exécution de l'algorithme avec n mouvements dans le pire des cas. Cet algorithme utilise une structure distribuée (arbre dynamique) pour garantir la connexité du réseau où seulement les nœuds feuilles peuvent se déplacer. Dans cette deuxième solution, j'ai proposé un mécanisme de réveil/sommeil permettant de réduire la quantité d'énergie consommée. Dans ces deux algorithmes, je présente des techniques pour prédire le nombre de mouvements de chaque nœud. Par conséquent, chaque nœud peut s'assurer qu'il a bien exécuté l'algorithme de reconfiguration et chaque nœud est conscient de la quantité d'énergie qu'il va dépenser. Nos algorithmes sont donc robustes et énergie-conscients.
- **Protocoles de redéploiement parallèles**. J'ai aussi proposé deux approches distribuées permettant la construction parallèle du carré final à partir d'une chaîne droite. L'objectif est d'améliorer les performances des algorithmes précédents et notamment d'optimiser le temps de reconfiguration et le nombre de mouvements de chaque nœud pour réduire la consommation d'énergie. Ces algorithmes utilisent dix états pour achever la reconfiguration. La première approche assure une connexité non-continue du réseau et la deuxième assure une connexité continue en utilisant l'arbre couvrant. Dans ces deux algorithmes un pré-traitement est nécessaire pour choisir un module initiateur permettant un degré de parallélisme optimal, ce qui nécessite des échanges de messages. Nous montrons dans ce mémoire de thèse que les deux algorithmes sont aussi robustes et énergie-conscients.
- **Protocole d'auto-reconfiguration généralisé**. Dans cette solution, je propose un algorithme distribuée sans carte pour le redéploiement des microrobots à partir de n'importe quelle configuration initiale connexe vers une configuration carré. Ce protocole se compose de deux algorithmes : un algorithme pour l'élection d'un module initiateur et un algorithme

de reconfiguration. L'objectif de l'élection d'un initiateur est d'avoir un algorithme avec un nombre de mouvements minimum sur l'ensemble du réseau. L'algorithme de reconfiguration vise à convertir la topologie physique initiale vers le carré final par un processus incrémental.

1.3 Plan de la thèse

La thèse est organisée en six chapitres qui sont divisés en plusieurs sections. Ci-après, un aperçu sur les chapitres restants est fourni :

Chapitre 2 présente un état de l'art détaillé sur les domaines de la microrobotique et des problèmes et techniques de reconfiguration des réseaux de robots. Ce chapitre commence par donner un aperçu détaillé sur les microrobots (leurs origines, leurs caractéristiques, leurs domaines d'applications, et les perspectives futures). Ensuite, sont abordées les questions d'autoreconfiguration et de redéploiement pour les microrobots modulaires. Notamment, les caractéristiques et les défis pour les algorithmes de reconfiguration sont discutés et un examen des différentes techniques de redéploiement des microrobots proposées dans la littérature est présenté. Ces techniques ont été classifiées en trois catégories suivant les caractéristiques des robots modulaires utilisés. Ce chapitre se termine par une conclusion et une discussion sur les différents inconvénients de ces solutions.

Chapitre 3 introduit notre contribution. On propose dans ce chapitre deux algorithmes efficaces qui n'utilisent pas de carte pour la reconfiguration des microrobots. Ce chapitre commence par fournir des définitions et des outils qui seront utilisés pour la présentation des algorithmes. Après, un algorithme de reconfiguration à partir d'une chaîne droite à un carré est présenté. Cet algorithme assure une connexité non instantanée du réseau. On montre comment prédire le nombre de mouvements pour chaque nœud. Après, on présente un autre algorithme qui assure la connexité du réseau tout au long de l'algorithme. On montre pour cet algorithme comment prédire le nombre de mouvements pour chaque nœud et comment rendre l'algorithme dynamique dans le sens réveil/sommeil. Ensuite, les caractéristiques de ces algorithmes concernant le nombre de mouvements sont analysées, le nombre d'états utilisés et le nombre de messages. Enfin, les résultats des simulations avec le langage déclaratif MELD et le simulateur DPRSIM sont présentés ainsi qu'une comparaison entre les deux algorithmes.

Chapitre 4 présente deux algorithmes parallèles pour le redéploiement sans carte de la forme cible. Ce chapitre commence par fournir des définitions et des outils qui seront utilisés pour la présentation des algorithmes. Ensuite un algorithme parallèle de redéploiement qui assure une connexité non instantanée est présenté. On donne ses caractéristiques et on montre comment faire cet algorithme robuste et énergie-conscient. Après, un algorithme parallèle de redéployent qui assure une connexité instantanée est présenté. On donne ses caractéristiques et on montre comment faire cet algorithme robuste et énergie-conscient. Enfin, on montre les résultats de simulations avec le langage déclaratif MELD et le simulateur DPRSIM et on donne une comparaison entre les deux algorithmes.

Chapitre 5 présente un protocole efficace généralisé pour la reconfiguration à partir de n'importe quelle configuration initiale à une configuration carrée. Ce protocole se compose de plusieurs algorithmes ; un algorithme d'élection d'un initiateur et un algorithme de reconfigura-

tion. Ce chapitre commence par fournir des définitions et des outils utilisés pour la présentation du protocole. Ensuite, l'algorithme d'élection d'un initiateur est présenté avec une discussion sur ses caractéristiques. Après, l'algorithme de reconfiguration distribué est présenté. Enfin, les résultats de simulations faites avec le simulateur DPRSIM sont présentés et analysés.

Chapitre 6 fournit une conclusion générale de ce mémoire et discute des points forts des solutions proposées dans cette thèse. Les travaux futurs et les propositions visant à améliorer les solutions de reconfiguration des microrobots sont présentées.

CHAPITRE 2

ETAT DE L'ART SUR L'AUTO-RECONFIGURATION

Sommaire

2.1 Introduction

Dans ce chapitre, nous introduisons les concepts de base relatifs aux robots modulaires et à l'auto-reconfiguration permettant de fournir au lecteur les éléments nécessaires pour aborder ce mémoire de thèse.

Les microrobots modulaires distribués fonctionnent sur la base d'une décomposition du système en des éléments unitaires très petits appelés microrobots. Ce mode de fonctionnement

permet de pallier les inconvénients des robots centralisés résidant dans des problèmes de fiabilité, d'équilibrage de charge et de tolérance aux fautes.

L'évolution des systèmes microrobots distribués est étroitement liée au développement des équipements et réseaux informatiques, comme la diminution du coût de fabrication, l'augmentation des performances, et les systèmes de production de masse [32, 69]. Ces développements ont contribué aux avancées dans le domaine du contrôle et l'architecture de communication des microrobots autonomes distribués. Les microrobots MEMS peuvent être définis comme des dispositifs miniaturisés, de faible puissance, ayant une capacité mémoire très limitée, qui peuvent capter des données de l'environnement extérieur et agir en conséquence [41, 97]. La taille des microrobots peut varier d'un micron à quelques millimètres de diamètre. Un système de microrobots est défini comme un ensemble de microrobots mobiles, autonomes, et homogènes.

Nous distinguons les réseaux de microrobots des systèmes informatiques amorphes [1, 72, 25] construits à partir d'un très grand nombre de petits éléments, qui sont en mesure de communiquer localement. La portée de communication étant très inférieure à la taille du système, dans ces systèmes informatiques, les éléments ne se déplacent pas. Cependant, l'idée de construire un système complexe composé de nombreuses parties plus simples est partagée. Pour s'adapter aux limitations caractérisant les microrobots MEMS, en particulier une puissance de calcul réduite et des réserves d'énergie limitées, de nouvelles approches de programmation sont nécessaires, différentes de celles utilisées pour l'élaboration des robots classiques. De telles approches incluent par exemple les méthodes de conservation d'énergie et les protocoles de *veille-sommeil* par lesquels des capteurs alternent, plus ou moins uniformément, des périodes d'éveil associées à une puissance relativement faible et des périodes de sommeil. Les capteurs éveillés restent à l'écoute de leur environnement tandis que les capteurs en sommeil évitent les écoutes très longues. Au moyen d'un algorithme distribué défini par une règle locale, les capteurs coordonnent les horaires de réveil-sommeil avec leurs voisins dans la fourchette de communication ; ces communications contribuent à la consommation d'énergie dans l'état de réveil [30, 35, 81].

Les mécanismes de reconfiguration dépendent du matériel et peuvent inclure des microrobots déformables qui peuvent ramper sur les modules voisins [28, 73] ou peuvent se dilater et se contracter afin de glisser sur les voisins [79]. Alternativement, les microrobots mobiles peuvent être contraints de maintenir rigidement leur forme originale, les obligeant à rouler sur les modules voisins [70, 102, 109]. Globalement, les systèmes microrobots sont caractérisés par les propriétés suivantes :

1. Coordination et coopération
2. Décentralisation fonctionnelle, et
3. Composantes miniaturisées

– La coordination et coopération entre les nœuds sont nécessaires pour mener à bien les tâches et missions associées aux réseaux de microrobots [23]. Pour assurer la coordination et la coopération des microrobots, la communication mutuelle doit être garantie dans le système, où le travail coopératif et coordonné entre les microrobots permet non seulement le transport coordonné et la manipulation coopérative, mais aussi la détection distribuée ou la prise de décision coopérative [7, 8, 9].
– La décentralisation fonctionnelle est guidée par les limitations de la capacité à l'échelle d'un microrobot du réseau. L'idée de la décentralisation fonctionnelle diffère de la conception

classique des systèmes robotique, c'est-à-dire, les composants conçus avec pour principe la décentralisation fonctionnelle ne sont pas toujours complets pour travailler individuellement. Aujourd'hui, la décentralisation fonctionnelle a plusieurs avantages, comme la variation de charge, la simplicité, la modularité, la diversité, et l'interchangeabilité [46].

– La miniaturisation des composants permet d'avoir une production massive. La miniaturisation des composants et leur production en masse augmentent la diversité des performances du système et diminuent le coût de production [58]. En outre, la miniaturisation étend les domaines d'applications des systèmes robotiques puisque permet le développement des micro-processeurs à faible coût.

Ces caractéristiques ont donné lieu à plusieurs avantages :

1. *Simplicité*, c'est la décomposition du système robotique à un nombre d'unités élémentaires, où l'unité élémentaire est une unité robotique ayant des fonctions simples. La simplicité encourage le développement et la conception d'unités robotiques.
2. *Modularité*, c'est l'approche conventionnelle utilisée pour le développement et l'amélioration des systèmes robotiques. De plus, dans les systèmes distribués de microrobots, la modularité est employée efficacement, puisque l'architecture du système décentralisé est basée sur la modularité de la configuration du système [22, 105].
3. *Robustesse*, en raison de leur nature modulaire, les microrobots ont un degré élevé de redondance, qu'ils peuvent exploiter pour maintenir la robustesse du système en entier. Une défaillance matérielle ou une erreur logicielle peut provoquer la panne d'un module. Cependant, cette panne ne provoque pas de dysfonctionnement de l'ensemble du système. Autrement dit, les autres modules peuvent compenser la perte d'un module [104, 105].
4. *Variance de charge*, elle permet de réduire la charge du système et le rend plus tolérant aux fautes.
5. *L'interchangeabilité*, elle est basée sur les caractéristiques de modularité du système, permettant de concevoir une interface commune aux unités et en prévoir en réserve. De ce fait, l'interchangeabilité accentue la flexibilité et la tolérance aux fautes du système.
6. *La diversité*, la diversité de la configuration du système est une conséquence de la modularité, de l'interchangeabilité et de la simplicité des systèmes de microrobots. Notamment, la simplicité des composants élémentaires élargit la diversité de configuration du système, car la simplicité des fonctions des unités élémentaires augmente le nombre de combinaisons et de formations possible de ces unités.

Les applications des microrobots MEMS sont de plus en plus vastes, ceci peut expliquer l'attention croissante donnée par les chercheurs à ce domaine. Des essaims à grande échelle de robots peuvent effectuer plusieurs missions et tâches dans une large gamme d'applications telles que la localisation d'odeur, la lutte contre les incendies, le service médical, la surveillance, la recherche et le sauvetage, la sécurité ... [80, 5]. Les systèmes de microrobots peuvent être utiles dans des situations qui nécessitent un mouvement de plate-forme automatisée. En exemple, considérons un système de panneaux solaires auto-reconfigurables sur le toit d'un bâtiment. En fonction de la position du soleil et les ombres créées par d'autres bâtiments, les modules formant un panneau pourraient se déplacer pour être en mesure d'absorber la lumière du soleil et donc produire le plus d'électricité. Avoir un nombre limité de panneaux solaires auto-reconfigurables

serait potentiellement plus rentable que de couvrir un toit entier avec des panneaux. Les systèmes de microrobots peuvent être utilisés pour l'exploration spatiale, la construction et le maintien de structures dans l'espace, et le divertissement [44, 74].

Afin d'accomplir ces tâches les microrobots doivent appliquer des algorithmes de redéploiement et d'auto-reconfiguration afin d'adapter leur forme (topologie physique) aux conditions de travail. Le problème d'auto-reconfiguration est défini dans la section suivante.

2.2 Définition du problème

Les composants micro-électronique-mécanique, MEMS, sont de moins en moins coûteux à fabriquer, rendant possible la combinaison des circuits logiques, des micro capteurs, des actionneurs et des dispositifs de communications, intégrés sur la même puce minuscule. Les microrobots pourraient se réunir dans des formes différentes en fonction du besoin. Un système de microrobots consiste en un nombre massif de microrobots (nœuds), programmés identiquement et déployés sur une zone géographique donnée. Les nœuds individuels disposent de ressources (énergie, vitesse et mémoire) limitées et d'informations uniquement locales. Les nœuds ont tous le même programme, bien que chaque nœud exécute son programme de manière autonome et dispose de moyens pour stocker l'état local. Un nœud ne peut communiquer qu'avec ses voisins physiques. Les nœuds peuvent également détecter et agir sur l'environnement au niveau local. L'auto-reconfiguration des microrobots renvoie au processus dynamique permettant leur redéploiement sous une autre forme cible. En d'autres termes, l'auto-reconfiguration est un planificateur distribué qui contrôle les mouvements des nœuds constituant au début une configuration de départ afin d'arriver à une autre configuration cible. Ce processus est difficile à contrôler, car il implique la cofordination distribuée d'un grand nombre de modules identiques dont les liens sont volatiles [103, 31, 87].

Le problème d'auto-reconfiguration est désigné par d'autres termes dans la littérature. Le terme auto-organisation est lui généralement utilisé dans le cas des réseaux de capteur sans fils. Comme par exemple les travaux [62, 63, 101] où des protocoles sont proposés pour former un cercle ou un polygone autour d'un nœud central. Le terme redéploiement est aussi utilisé pour désigner la reconfiguration de la topologie physique d'un réseau de capteur sans fil [86, 67, 47].

L'auto-reconfiguration des microrobots est différente du problème d'entrepôt connexe (où des identifiants uniques sont attribués aux modules qui doivent être placés à des endroits souhaités)[14]. Dans [43], l'auto-reconfiguration est définie comme le processus qui transforme la topologie physique d'un réseau afin d'atteindre la topologie logique la plus optimale du point de vue de la communication.

Un algorithme de reconfiguration doit être robuste face à la défaillance des modules et à l'échec de la communication. Il doit prendre en compte les contraintes d'extensibilité du système, en tenant compte des changements sur le nombre de noeuds disponibles. Une autre caractéristique importante d'un algorithme de reconfiguration est l'évolutivité d'échelle, qui renvoie à la capacité de l'algorithme à maintenir ses performances quand le nombre de noeuds augmente. Un protocole de reconfiguration efficace doit tenir compte des caractéristiques des microrobots, notamment en ce qui concerne l'utilisation de la mémoire et la consommation énergétique. Ainsi, un algorithme est jugé efficace quand il utilise peu de mémoire, qu'il nécessite peu d'échanges

de messages, et qu'il consomme peu d'énergie. Les principales métriques pour évaluer la qualité d'un algorithme de reconfiguration sont les suivantes :

- **Robustesse**, est la faculté d'un algorithme de reconfiguration à poursuivre son exécution et à atteindre son objectif en situation de défaillance des noeuds et à l'échec de la communication.
- **Adaptabilité**, l'algorithme de reconfiguration doit exploiter l'entrée du (ou des) capteur(s) pour s'adapter aux changements de l'environnement. Si le robot n'est pas équipé de capteurs pour détecter des changements dans l'environnement, il ne peut pas s'adapter.
- **Polyvalence**, le système de reconfiguration doit permettre aux microrobots d'effectuer de nombreuses tâches différentes et de s'adapter à différentes situations.
- **L'échelle de l'extensibilité**, le système de reconfiguration doit tenir compte des changements sur le nombre de noeuds disponibles, et un contrôle qui ne dépend pas de la taille du réseau.
- **Évolutivité**, l'algorithme de reconfiguration doit être capable de gérer des systèmes constitués d'un très grand nombre de modules.
- **Vitesse**, l'algorithme de reconfiguration devrait s'achever rapidement pour optimiser la consommation d'énergie.
- **Équilibrage de charge**, le système de reconfiguration doit équilibrer l'effort de déplacement (le nombre de mouvements) sur les nœuds pour optimiser la durée de vie des nœuds.
- **Les ressources**, les nœuds formant le réseau de microrobots sont caractérisés par une durée de vie limitée de la batterie ainsi qu'une faible capacité de calcul et de mémoire. Ces limites constituent une question cruciale à laquelle l'algorithme de reconfiguration doit porter une grande attention.

Dans un système centralisé, les microrobots sont supervisés par un module centralisé doté de ressources élargies et contrôlant l'avancement de la reconfiguration. Dans les systèmes distribués un contrôleur s'exécute sur chaque noeud avec pour objectif d'achever la reconfiguration [11, 24]. Les algorithmes de reconfiguration centralisés sont traditionnellement utilisés pour contrôler des robots, mais ne sont pas adaptés pour les réseaux de microrobots. La raison est que si des changements surviennent sur des noeuds ou sur l'environnement, un processus de planification temps-réel doit avoir lieu dans l'unité centrale avant que les robots ne puissent continuer [106]. Concrètement, cela signifie que les systèmes de contrôle centralisé présentent un manque en robustesse, en adaptabilité et en extensibilité. En outre, puisque le module centralisé est responsable du contrôle de tous les modules, il est alors sujet à une surcharge liée au nombre croissant de modules. Cela peut induire un affaiblissement de l'efficacité du système centralisé qui ne s'adapte pas à une augmentation du nombre de modules. Enfin, en cas de panne ou erreur sur le module contrôleur, le processus de reconfiguration ne peut pas s'achever (les systèmes de contrôle centralisé ne sont pas tolérant au fautes).

Le contrôle distribué permet d'éliminer les inconvénients du contrôle centralisé. Mais si l'information globale est nécessaire entre les modules, le système devient moins robuste [102, 82]. En outre, l'évolutivité des systèmes distribués peut être remise en cause quand les modules passent énormément de temps à attendre des informations globales sur des grands réseaux [83, 85, 85, 12, 13]. Pour répondre à ces inconvénients, le contrôle distribué utilisant seule-

ment l'information locale est abordé afin de créer un système robuste. La panne d'un module ou une erreur de communication ne provoque pas l'arrêt du système. Cependant, dans ce cas, les microrobots doivent interagir localement [42, 14, 15]. Il peut s'avérer alors difficile de trouver des règles locales qui permettent au système de converger vers le comportement global désiré [61, 16, 68].

2.3 Auto-reconfiguration pour des robots hexagonaux

2.3.1 Modèle et Hypothèses

Le schéma hexagonal est l'un des formats adoptés pour la conception des microrobots modulaires [6, 99, 93, 94, 95, 96, 66, 59, 26, 27, 71]. En effet la forme hexagonale est la forme géométrique permettant le pavage régulier de l'espace 2D (avec le triangle isocèle et le carré) avec un maximum de connectivité (6 motifs voisins). L'espace géométrique dans lequel évoluent les microrobots de même taille est représenté par une grille de cellules désignant les positions stables où les microrobots peuvent se positionner. Dans ces algorithmes, les robots sont identiques, mais agissent comme des agents indépendants qui prennent des décisions en fonction de leurs positions actuelles et des données sensorielles provenant de leurs contacts avec les robots adjacents. Les modules peuvent être déformables (le robot change sa forme pour achever quelques types de mouvements), dans ce cas, chaque module se déplace en changeant les angles des articulations pour ramper sur un substrat immobile. Les modules peuvent être rigides utilisant des mouvements de glissement comme spécifié dans [68] pour se déplacer sur le substrat. Chaque module connait à tout moment : sa localisation (les coordonnées de la cellule qu'il occupe en ce moment), les positions des cellules dans la configuration finale, son orientation et les cellules voisines vides. Le module peut se déplacer sur un voisin substrat (S) vers une cellule $C1$ si : $C1$ est vide, le module substrat (S) est immobile à l'étape courante de l'algorithme permettant son utilisation comme support de déplacement et que la cellule au travers du mouvement, $C2$, est vide comme le montre la figure 2.1.

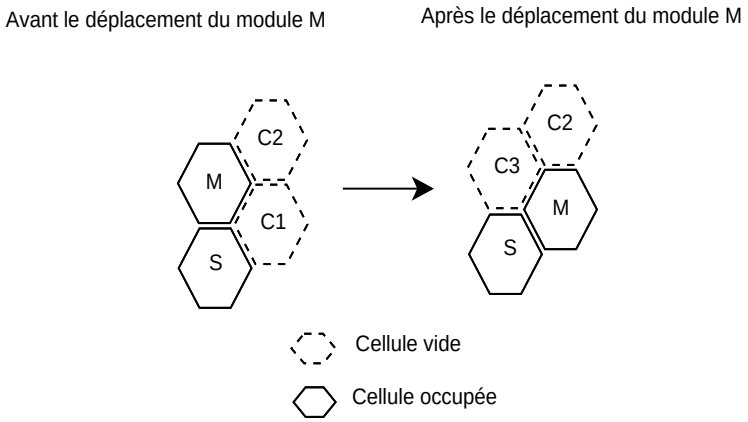

FIGURE 2.1 – Le déplacement du module

Une configuration finale : la configuration finale est modélisée sous forme d'un réseau connexe n'ayant pas des sections verticales vides. Les modules de la configuration finale sont orientés de

telle sorte qu'ils remplissent la surface du plan en regard du nord (N) et sud (S). De cette façon, nous pouvons clairement décrire les nœuds qui sont les plus à l'est, et celles qui sont le plus à l'ouest.

Module libre : un module est libre s'il a seulement un voisin dans la direction N, ou seulement des voisins dans les directions N et NE, ou seulement des voisins dans les directions N et NE et SE, ou seulement des voisins dans les directions N et NE et SE et S. Le module est libre quand il a au moins deux cotés vides (il n'y a pas de voisins) et quand son déplacement ne cause pas le dis-connectivité du réseau.

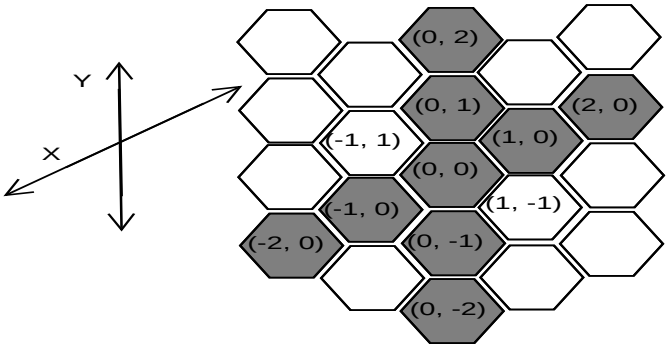

FIGURE 2.2 – Le système de coordonnées [26]

La distance réticulaire : décrit le nombre minimum de cellules que doit parcourir un module pour passer de la cellule $c1$ à $c2$. Il désigne aussi la distance en ligne directe entre deux cellules $c1$ et $c2$ de coordonnées respectives $(x1, y1)$ et $(x2, y2)$. La distance réticulaire entre deux cellules est mesurée comme suit [26] :

$$\Delta_x = (x1 - x2) \tag{2.1}$$

$$\Delta_y = (y1 - y2) \tag{2.2}$$

$$LD(c1, c2) = \begin{cases} max(|\Delta_x|, |\Delta_y|), si \Delta_x.\Delta_y < 0, \\ |\Delta_x| + |\Delta_y|, sinon. \end{cases} \tag{2.3}$$

Soit $G1, G2, ..., Gm$ le partitionnement des cellules occupées par la configuration finale en colonnes d'ouest en est, tel que dans la figure 2.3. Chaque colonne représente la liste des cellules, contiguës (figure 2.3 (a)) ou pas (figure 2.3 (b)), d'une même colonne verticale ordonnée depuis la cellule au plus nord vers la cellule au plus sud. La figure 2.3 (b) montre un exemple d'une configuration finale où les colonnes $G3$ et $G5$ sont composées de chaînes non contiguës de cellules.

Définition du chemin du substrat : on dit qu'une séquence, P, contiguë de cellules distinctes, $c_1, c_2, ..., c_k$ constitue un chemin de substrat si :

- P commence avec une cellule, c_1, dans la configuration initiale, de nord au sud et
- les cellules suivantes sont toutes dans la configuration finale et
- la dernière cellule, c_k, est au plus est de la configuration finale (dans la colonne Gm)

Un segment de P est une sous-séquence contiguë de P de longueur ≥ 2. Dans un segment dit sud, chaque cellule de la sequence est au sud de la cellule précédente. De manière analogue pour un segment de nord, chaque cellule de la sequence est au nord de la cellule précédente.

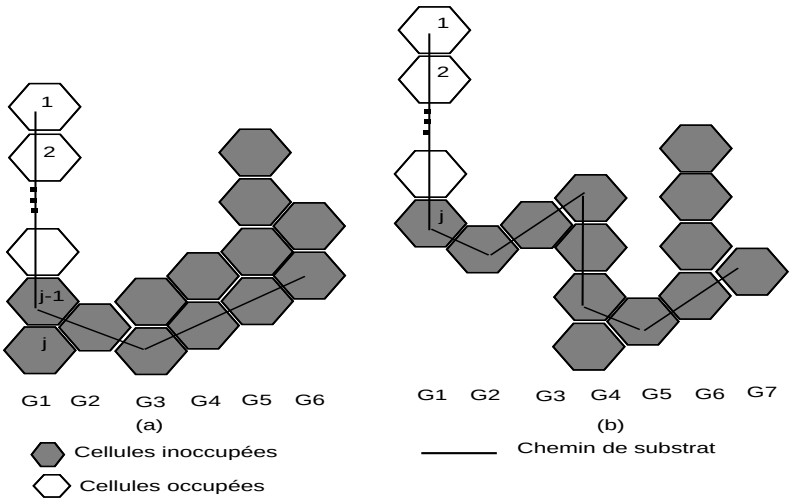

FIGURE 2.3 – Exemples : (a) d'une configuration finale admissible et (b) une configuration finale inadmissible, dans (a) toutes les colonnes sont composées de cellules contiguës, dans (b) $G3$ et $G5$ sont composés de cellules non-contiguës

Définition du chemin du substrat admissible (CS) : P est un chemin substrat admissible si :
1) chaque cellule de P est adjacente à la précédente, mais pas à l'ouest.
2) pour chaque segment nord de P se terminant par c_i :
a) les cellules consécutives X_i, Y_i, et Z_i se trouvant au sud-est de c_i ne sont pas des positions finales (voir la figure 2.4 (a)) et
b) les cellules $c_{i+1}, c_{i+2}, c_{i+3}$ et c_{i+4} ne font pas tous aucun segment de sud, et
3) pour chaque segment sud de p se terminant par c_i :
a) les cellules consécutives X_i, Y_i, et Z_i, se trouvant au nord-est de c_i, ne sont pas des positions finales (voir la figure 2.4 (b)) et
b) $c_{i+1}, c_{i+2}, c_{i+3}$ et c_{i+4} ne font pas tous aucun segment de nord.
Les segments nord ou sud de P peuvent être nommés des segments verticaux lorsque la direction spécifique du segment n'est pas importante.

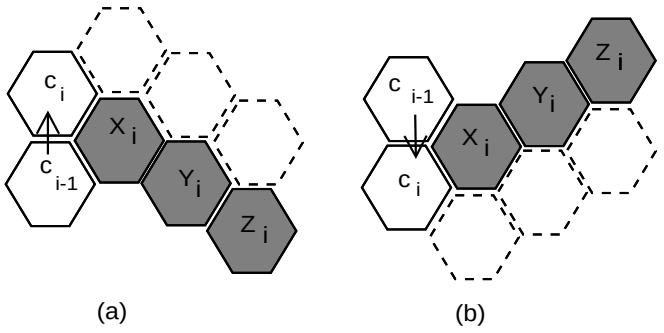

FIGURE 2.4 – étiquettes pour le segment nord finissant à c_i (a) et étiquettes pour le segment sud finissant à c_i (b) (les cellules qui ne doivent pas être des cellules cibles sont ombragées), les cellules en pointillés sont des cellules de but

Définition d'une configuration admissible : une configuration cible est admissible s'il existe un chemin de substrat admissible. Ce chemin de substrat admissible est une chaîne de cellules permettant des mouvements sans collision ni inter-blocage entre les modules, à condition que les choix de rotation du module et le retard sont appropriés. La figure 2.3 (a) montre un exemple d'une configuration admissible où le chemin de la configuration initiale à la configuration finale est un chemin de substrat admissible. La figure 2.3 (b) montre un exemple d'une configuration finale non admissible puisque le segment sud de la colonne $G3$ viole la définition du chemin de substrat, puisque chaque CS dans la figure 2.3 (b) doit inclure un segment dans $G4$; il n'y a aucun CS admissible avec cet exemple. Donc cette configuration n'est pas admissible.

2.3.2 Auto-reconfiguration chaîne à chaîne

Dans ce papier [93], les auteurs proposent un algorithme distribué pour la reconfiguration d'un système robotique métamorphique composé d'un nombre quelconque de modules hexagonaux, d'une forme initiale spécifique à une configuration cible spécifique. Il s'agit d'un protocole distribué de reconfiguration d'une chaîne droite de modules hexagonaux à une autre chaîne droite dans un autre emplacement dans le plan. Ils traitent le cas d'une chaîne colinéaire et la chaîne non colinéaire. La chaîne colinéaire est définie comme une chaîne où le nœud a un seul voisin dans la direction S ou un seul voisin dans la direction N ou deux voisins dans les deux directions S et N.

A) **Le cas des chaînes colinéaires :**

L'algorithme fonctionne en cycles synchrones. Initialement, chaque module détermine le sens de sa rotation autour de son voisin : dans le sens horaire (CW) ou dans le sens antihoraire (CCW). Le sens de rotation est déterminé en fonction de la distance réticulaire entre le module et la cellule de chevauchement. À chaque tour, le module se déplace s'il est libre dans le sens calculé initialement. Dans ce cas, le nœud est libre s'il a seulement un voisin dans la direction N ou deux voisins dans la direction N et NE. La figure 2.5 montre un exemple de reconfiguration.

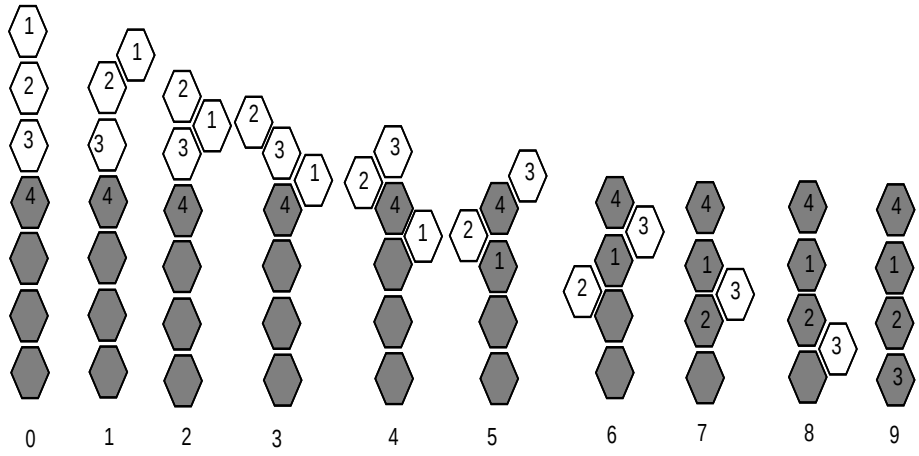

FIGURE 2.5 – Exemple d'une reconfiguration chaîne colinéaire [95]

B) **Le cas des chaînes non colinéaires :**

Cet algorithme est une extension de l'algorithme de la chaîne colinéaire, pour permettre une configuration à partir d'une chaîne droite à une autre chaîne droite qui chevauche avec la chaîne initiale dans n'importe direction. Cette extension ne nécessite pas de communication entre les modules, mais utilise des techniques de comptage associées à la connaissance des positions cibles pour déterminer la position de chaque module. Dans cette solution, la chaîne initiale est orientée nord-sud, et la chaîne finale est orientée sud-ouest-nord-est et le nœud plus au nord est dans la configuration finale. Les cas d'une reconfiguration chaîne à chaîne sont montrées dans la figure 2.6, où les modules de la configuration initiale ont des frontières solides et les cellules cibles sont ombragées. Dans la figure 2.6, les modules de la configuration initiale dans les cas 1 à 3 sont numérotés comme indiqué dans le cas 1 et les modules du cas 4 et 5 sont numérotés comme indiqué dans le cas 4. Les modules peuvent se déplacer seulement s'ils sont libres. Les algorithmes pour les cas 1 à 5 sont similaires à au cas 0. Les différences sont :

- déterminer si un module est libre,
- calculer la direction dans laquelle se déplacer, et
- calculer le retard, c'est à dire, combien de temps à attendre après être devenu libre jusqu'au début du mouvement.

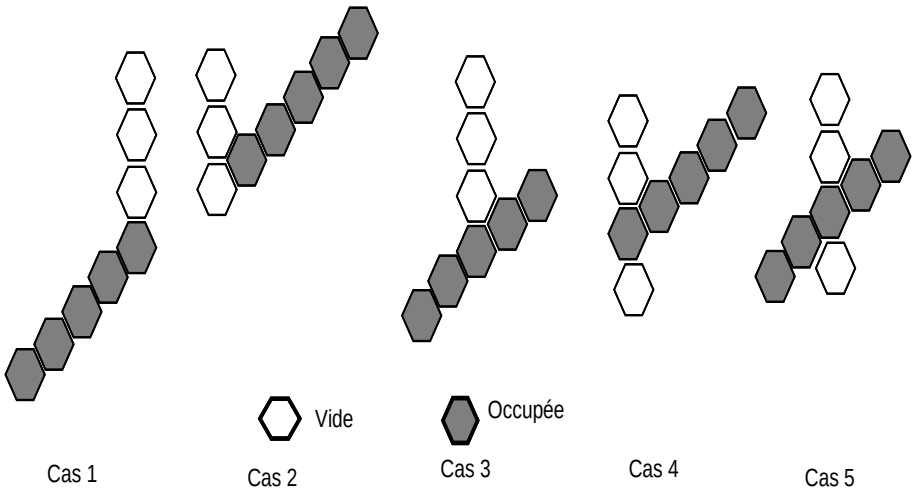

FIGURE 2.6 – Les différents cas de la chaîne non collinaire [95]

2.3.3 Auto-reconfiguration d'une forme quelconque en une chaîne

Dans cette solution [99], les auteurs présentent un algorithme d'auto-reconfiguration d'un réseau connexe de robots modulaires hexagonaux [27] ayant une forme quelconque en une chaîne droite.

Il s'agit d'un algorithme déterministe distribué parallèle pour la reconfiguration d'un système de modules d'une forme initiale quelconque vers une configuration finale représentant une chaîne droite. Les avantages de l'approche sont l'absence de procédures de prétraitement et de transmission de messages pour accomplir la reconfiguration. L'algorithme élimine les risques de collisions entre les modules en mouvement en considérant que les modules sont capables de détecter un autre module dans un périmètre d'une cellule autour. L'algorithme converge à condi-

tion que : le système démarre dans une configuration initiale admissible, les cellules cibles sont connues par tous les modules et chaque module est équipé de capteurs pour déterminer si la cellule adjacente est vide ou occupée. On note que cette solution nécessite que chaque module connaisse les positions prédéfinies de la chaîne. La complexité temporelle de l'algorithme est de $O(n^2)$, où n est le nombre de modules.

Les modules : Dans cette solution, le robot doit avoir la capacité de détecter les robots directement connectés à lui, ainsi que des modules qui sont à une distance d'une cellule d'espace dans une direction latérale adjacente. Ceci peut être réalisé avec des capteurs de contact et des capteurs infrarouges. Un module doit avoir au minimum deux côtés libres adjacents afin qu'il puisse se déplacer. Les modules peuvent se déplacer soit dans le sens horaire (CW) ou dans le sens antihoraire (CCW) ou glisser sur les deux faces d'un substrat pour obtenir le même effet (cf. figure 2.7 (f)-(j)). Les modules déformables se déplacent par une combinaison de rotation et de changement d'angles des articulations, [26] (cf. figure 2.7 (a)-(e)). Les modules rigides se déplacent en glissant sur deux faces d'un substrat [68] tout en conservant l'orientation de l'original (figure 2.7 (f)-(j)).

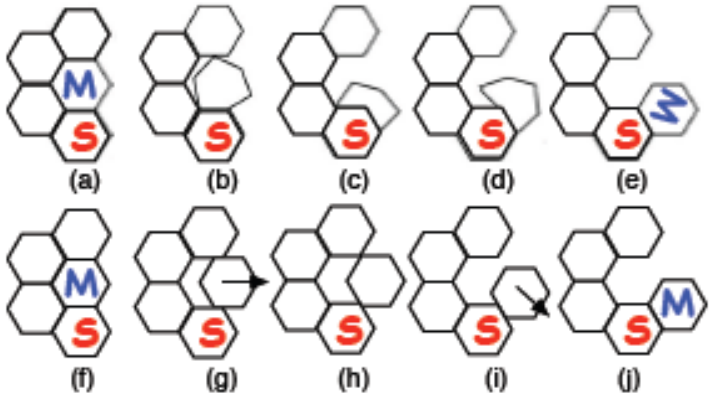

FIGURE 2.7 – (a)-(e) le déplacement d'un module déformable M sur un substrat S, (f)-(j) déplacement d'un module rigide sur un substrat S [99]

Conditions sur l'environnement de reconfiguration : Premièrement, pour commencer l'algorithme de reconfiguration il faut qu'au moins un module soit déjà dans la forme cible. Deuxièmement, l'auto-reconfiguration est possible si la forme initiale remplit certaines conditions à savoir :

- chaque colonne de la configuration initiale est contiguë du nord (N) vers le sud (S),
- chaque module de la configuration initiale ne se trouvant pas sur la colonne la plus à l'ouest (W) a initialement voisin du côté soit nord-ouest (NW), soit sud-ouest (SW), ou les deux en même temps et
- chaque module de la colonne la plus à l'ouest a initialement un module adjacent du côté soit NE, soit SE, ou les deux en même temps.

La figure 2.3 (a) montre un exemple d'une configuration initiale remplissant ces conditions et la figure 2.3 (b) montre un exemple qui ne remplit pas les conditions.

Principes de l'algorithme : Les modules se déplacent en phases synchrones, avec lors

de chaque phase le maximum possible (en fonction des contraintes physiques) de modules en déplacement. Les conditions spécifiques telles que la direction du dernier mouvement effectué par le module, les retards, les voisins courants, l'orientation de la chaîne cible, et l'emplacement des cellules cibles, sont utilisées dans l'algorithme pour garantir qu'un module n'est jamais en collision avec un autre module. Les modules de la colonne la plus l'ouest (W) commencent à se déplacer avec un espacement d'une cellule simple entre les modules mobiles. Les modules s'enroulent autour des colonnes en direction de l'est (E) en formant deux colonnes mobiles pour s'assurer que tous les modules (sauf un) atteignent des cellules de la chaîne cible par nord ou par sud (le dernier module pénètre dans la cellule objective la plus à l'est par NE par SE). Pour appliquer cette restriction avec les règles qui sont basées sur la configuration locale de chaque module, l'algorithme construit deux colonnes plus hautes jusqu'à ce que les modules commencent à pénétrer dans les cellules cibles, lorsque les colonnes commencent à diminuer.

Les modules qui ont au moins deux cellules consécutives voisines vides, des cellules libres au NW et au SW, ou ayant des cellules libres au N, S, NE, et SE peuvent se déplacer à n'importe quelle étape de l'algorithme. Le module subit un retard d'une étape quand il est dans une configuration où un module voisin peut ou pas être en mouvement, en particulier lors de la mise au coin dans une colonne à la direction E. Les modules arrêtent de bouger si aucune des règles de mouvement ne s'applique à la configuration locale, ou si le module est dans une cellule cible.

Un module bloqué qui n'est pas dans une position de la chaîne cible commence, à nouveau, à se déplacer après que les modules voisins au NW ou SW aient quitté les cellules qu'ils occupaient, et qu'une cellule adjacente dans la direction N, S, NE, ou SE devienne vide. Un module ayant effectué son premier déplacement dans le sens horaire CW est restreint à ce sens tout le long de l'algorithme ; de même pour un module qui lors de son premier mouvement, s'est déplacé dans le sens horaire inverse CCW.

2.3.4 Auto-reconfiguration d'une chaîne vers une forme quelconque

2.3.4.1 Reconfiguration avec espacement de deux cellules

Dans [95], les auteurs abordent le problème de la reconfiguration distribuée à partir d'une chaîne droite de robots vers des configurations cibles répondant à certaines conditions générales d'admissibilité. Il s'agit d'un protocole d'auto-reconfiguration concurrentielle constitué d'un algorithme centralisé qui vérifie si une configuration cible arbitraire est admissible. Si la configuration cible est admissible alors l'algorithme centralisé recherche un chemin de substrat sur lequel les modules robotiques peuvent se déplacer pour atteindre la configuration cible. Ensuite, un algorithme distribué est lancé pour reconfigurer la chaîne droite initiale à la configuration cible admissible à l'aide du chemin de substrat trouvé par l'algorithme centralisé. L'algorithme de planification des mouvements maintient une distance suffisante entre les modules en mouvement. D'autre part, le chemin substrat est généré de façon à ne pas contenir de poches ou de coins susceptibles de bloquer les modules.

Détection de configurations admissibles et les chemins de substrat : Pour déterminer si la configuration finale est admissible, les cellules cibles sont analysées du nord au sud puis du NW au SE et en fin du NE au SW, pour déterminer s'il existe une orientation (qui déterminera le sens des colonnes) pour laquelle toutes les colonnes de la forme cible sont contiguës. Si

cette orientation est inexistante alors la configuration finale est considérée non admissible. La deuxième étape de l'algorithme centralisé consiste à déterminer un chemin de substrat admissible. L'algorithme ci-dessous décrit les différentes étapes de cette procédure.

Algorithme ArêtesDirectes :

- Tout d'abord, chaque nœud de la configuration cible dans la colonne Gm est marqué, comme le montre la figure 2.8 (a).

- Les cellules au nord, au sud, au nord-est, et au sud-est de Gi, j (j ème nœud de la colonne i) sont étiquetées Ni, j, Si, j, NEi, j, et SEi, j, respectivement. - Chaque colonne à l'ouest de Gm, $(Gm - 1, ..., G1)$, est divisée en trois segments (notés sur la figure 2.8 (a)) par : (N) le segment nord de la colonne n'ayant pas de cellules cibles à l'est (éventuellement vide), (C) le segment central de la colonne, constitué de cellules ayant des cellules cibles à l'est, et (S) le segment sud de la colonne n'ayant aucune cellule cible à l'est (peut-être vide).

- Pour chaque colonne, $Gm - 1$ jusqu'à $G1$:

1. Les nœuds du segment (C) sont traités dans l'ordre du nord au sud. Si un nœud dans le segment (C) a, à l'est, un ou plusieurs voisins marqués, il est à son tour marqué et une arête directe depuis le noeud vers chacun de ses voisins marqués est ajoutée. Les seules exceptions sont quand une arête NE est dirigée vers un voisin avec une arête S sortante ou si une arête SE est dirigée vers un voisin avec une arête N sortante. Ces exceptions permettent d'assurer qu'aucun coin à angle aigu ne sera inclus dans n'importe quel chemin de substrat.
2. Les nœuds du segment (S) sont traités du nord au sud. Chaque nœud ayant un voisin nord marqué, est marqué à son tours et une arête dirigée vers son voisin nord est ajoutée si l'arête est un préfixe d'un chemin admissible.
3. Les nœuds dans le segment (N) sont traités du plus au sud au plus au nord. Chaque nœud est marqué et une arête dirigée vers son voisin sud est ajoutée si le voisin sud est un préfixe d'un chemin admissible.

La figure 2.8 (a) - (f) représente une exécution de la procédure ArêtesDirectes, sur six étapes de (a) à (f).

Algorithme TrouverChemin :

L'algorithme TrouverChemin, décrit ci-dessous, est utilisé pour construire un chemin de substrat admissible :

l'algorithme se déroule selon les règles suivantes jusqu'à ce que toutes les cellules de la colonne Gm soient ajoutées au chemin de substrat :

1. Si un nœud comporte une arête dirigée vers un seul voisin marqué (soit N, S, NE, ou SE), alors on traverse l'arête dans cette direction et on ajoute l'arête au chemin du substrat. Pour chaque arête verticale orientée (N) dont l'extrémité finale est une cellule cible Gi, j $(1 =< i < m-3)$, la cellule NE de la cellule NEi, j n'est pas marquée, si il y a un segment dans la colonne $Gi + 3$. Une action symétrique est prise pour un segment vertical
2. Si un nœud possède des arêtes dont les deux extrémités sont deux voisins au NE et SE marquées

Le temps nécessaire pour vérifier si les colonnes sont contigües est de l'ordre de $O(n)$. Le temps d'exécution de l'algorithme de construction des arêtes et du chemin substrat admissible est

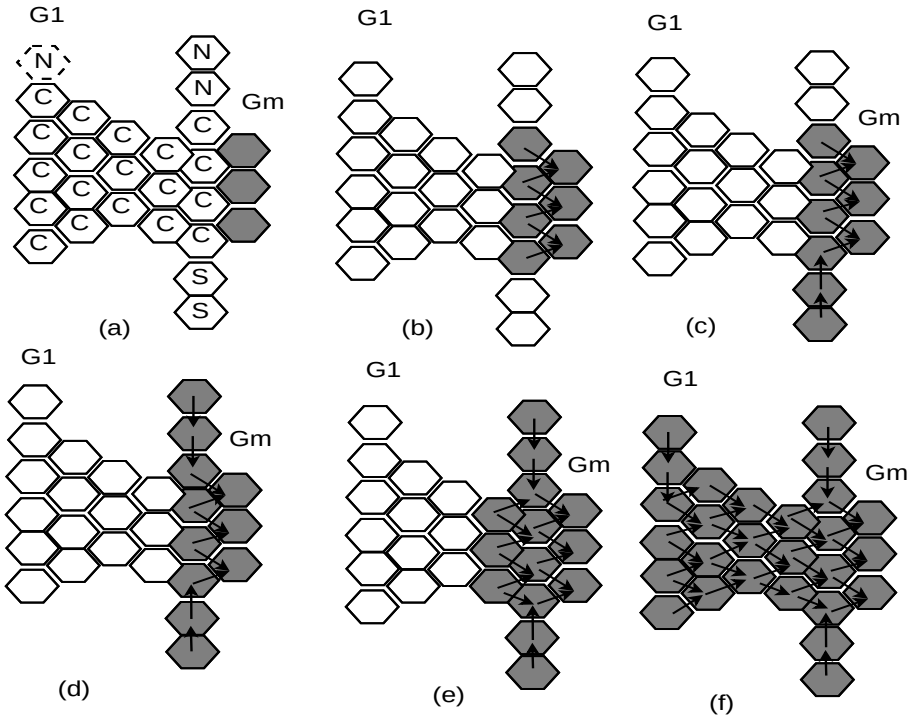

FIGURE 2.8 – Exemple de marquage des nœuds (cellules sombres) à partir de la dernière colonne, Gm, de la configuration cible [95]

de $O(n)$, étant donné que chaque nœud dispose d'un nombre constant de voisins (non orientés).

Algorithme de reconfiguration distribué : Dans cette section l'algorithme de reconfiguration d'une forme initiale I à une configuration cible G est présenté [95]. Cet algorithme utilise le chemin substrat (CS) trouvé dans la section précédente pour achever la reconfiguration. Dans cet algorithme chaque nœud connait dès le départ sa position finale, chaque module utilise seulement les informations locales et sa position pour déterminer son prochain mouvement à chaque étape de l'algorithme. Chaque module connait le nombre de nœuds dans la configuration initiale I et connait toutes les positions cible de G. Au départ un module $(G1, 1)$ doit appartenir à I et G en même temps.

L'algorithme fonctionne en cycles synchrones, à chaque étape, chaque module calcule s'il est libre. Tous les modules libres se déplacent simultanément. Les modules initialement dans I calculent leurs positions dans I, la direction de rotation, le retard possible, et les coordonnées finales dans G en déterminant leur distance réticulaire à partir de $G1, 1$. Le module calcule la cellule cible qu'il va occuper en comparant sa position dans I à la longueur des tableaux de coordonnées au nord, et au sud de CS.

Soit P le CS admissible trouvé, en commençant par la cellule qui possède une arête entrante de la cellule $G1, 1$. Les modules dont la position $\leq P$ remplissent CS en premier. Après avoir rempli P, les modules alternent le sens de rotation (CW ou CCW), remplissant les colonnes cibles au nord et au sud de P depuis l'est (la colonne Gm) vers l'ouest (la colonne $G1$). La figure 2.9 illustre l'itinéraire des nœuds des cellules occupées (cellules numérotées) à partir de la configuration initiale (a) pour atteindre les cellules cibles (cellules grisées) à l'étape (i).

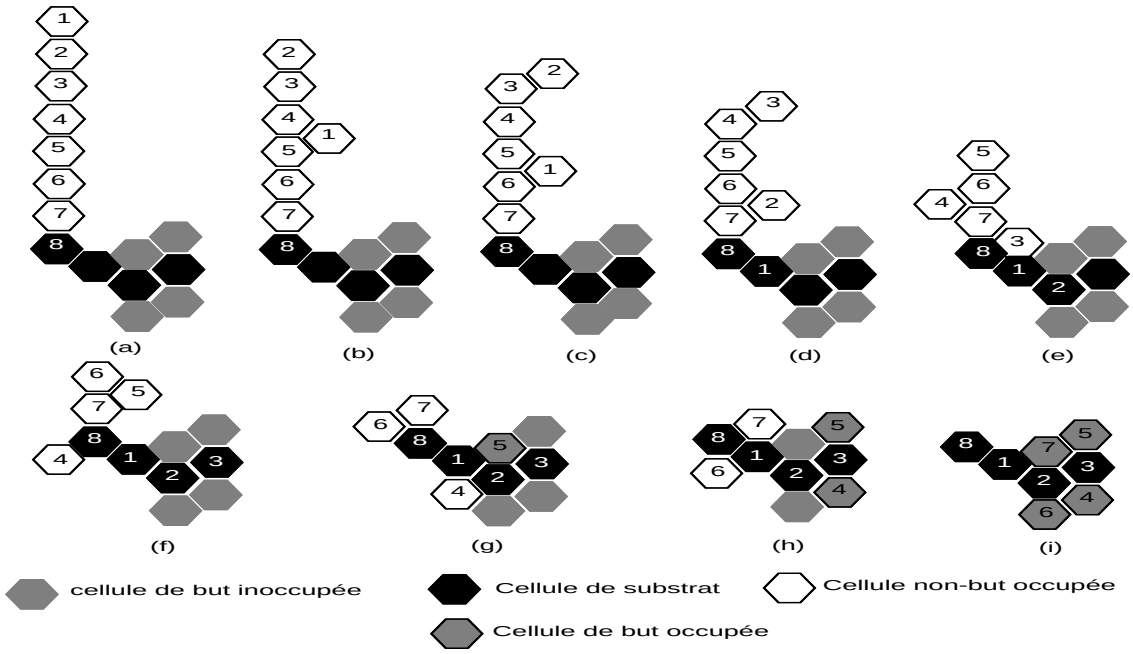

FIGURE 2.9 – Exemple d'exécution de l'algorithme de reconfiguration [95]

2.3.4.2 Reconfiguration avec un espacement simple-cellule

Dans ce papier [6] les auteurs proposent un algorithme pour la reconfiguration des systèmes de robots hexagonaux, partant d'une chaîne droite à une forme arbitraire qui représente la configuration cible. Dans cette solution, la configuration finale doit satisfaire certaines conditions d'admissibilité. L'algorithme se compose de deux parties, une partie de prétraitement centralisée et une partie de reconfiguration distribuée. La première partie centralisée sert à trouver le chemin de substrat qui doit être rempli par les robots pour se déplacer vers les positions de la configuration finale. Ainsi, l'objectif est de permettre le déplacement des modules voisins et d'éviter la collision entre les robots mobiles. Dans la partie distribuée, les modules utilisent les chemins contigus calculés dans la partie centralisée pour se déplacer vers les positions finales de la configuration cible. Dans cette partie les modules se déplacent en maintenant un espacement d'une cellule entre les modules mobiles, tout en veillant à ce que les modules mobiles ne soient pas en contact lors de leur passage par les coins d'angles aigus. L'algorithme exige que dans la configuration initiale, au moins un module soit déjà dans la configuration finale. Dans cette approche, certains modules peuvent être suspendus temporairement lors de la reconfiguration, formant des ponts pour d'autres modules à traverser. Une fois que tous les modules soient passés sur les modules pont, les modules pont reprennent le mouvement. L'algorithme assure qu'aucune paire de modules mobiles ne soient en contact au cours de la reconfiguration en maintenant un espace de séparation (cellule vide entre les modules mobiles). Ainsi les modules peuvent se déplacer avec une seule séparation sur les deux flancs du chemin de substrat.

A)Prétraitement centralisé

Les modules sont numérotés de 1 (le module le plus loin de la configuration finale) jusqu'à $n-1$ (le module adjacent au module déjà dans la configuration finale). Il reste donc à occuper

$n-1$ positions de la configuration cible. La stratégie utilisée pour remplir la configuration finale tout en évitant la collision, consiste à trouver un chemin de substrat. Après avoir rempli le CS, les cellules à l'extrémité N et S de CS sont remplies en parallèle, avec un espacement minimal entre modules. Le remplissage du CS garantit qu'un module occupant le N ne se heurtera pas à un module occupant le S, permettant aux deux segments un remplissage en parallèle. Les étapes pour trouver le CS sont les suivantes :

1. Trouver la cellule médiane dans chaque colonne de la configuration cible et considérer cette cellule comme une cellule appartenant déjà à CS. Si la taille de la colonne est paire, alors la cellule au nord ou au sud du point théorique médian est choisie comme faisant partie du CS.
2. Vérifier si le CS trouvé à l'étape 1 est contiguë. Sinon, utiliser des algorithmes présentés dans [95] pour tester chaque chemin possible dans G de gauche à droite. Si aucun CS sans segments verticaux n'est trouvé, continuer avec des algorithmes de reconfiguration donnés dans [95].
3. Si le seul CS trouvé contient des segments verticaux, on utilise les algorithmes de reconfiguration décrits dans [95] pour terminer la reconfiguration. Ces algorithmes utilisent un espacement de deux cellules pour éviter la collision et l'inter-blocage.
   Si le CS n'a pas de segments verticaux, il existe un algorithme RemplissageCheminSubsrat, permettant de remplir CS en se basant sur un espacement simple-cellule. Les modules commencent à se déplacer quand ils sont libres. On note NB la largeur de CS, sans compter le module initialement dans la configuration finale.

Algorithme RemplissageCheminSubsrtat :

Avant de commencer à décrire l'algorithme de reconfiguration on donne ici les deux définitions suivantes :

– $PenteInitiale$: la pente de la configuration initiale, SW à NE ou NW à SE

– $PenteFinale$: pente des deux dernières cellules dans CS (NE ou SE) La pente est la direction à partir du module numéro 1 à la cellule de la configuration initiale déjà sur une cellule cible (numéro n), elle doit être soit NE ou SE.

Les étapes de l'algorithme RemplissageCheminSubsrtat sont :

1. Si la $PenteInitiale$ est NE alors le module 1 se déplace dans le sens CW
2. Sinon, le module 1 se déplace dans le sens CCW
3. Si NB est pair ou que la $PenteFinale$ est identique à la $PenteInitiale$ alors les modules 2 jusqu'à NB alternent les directions, à partir du sens inverse à celui du module 1 immédiatement à l'étape où ils deviennent libres.
4. Sinon, les modules 2 jusqu'à $NB+1$ alternent les directions de rotation, commençant de se déplacer par le sens inverse de celui du module 1 sans retard à l'étape où ils deviennent libres.

Dans l'algorithme RemplissageCheminSubsrtat, le module NB (ou NB+1) est le premier module à atteindre une cellule du CS. Si NB est pair, les NB premiers modules atteignent le CS pas nécessairement dans l'ordre de leurs positions. Si NB est impair et le dernier sens de parcours

est égale à la direction du trajet initial, le dernier module sur le chemin (un module impair) atteindra le CS devant son homologue pair. La figure 2.10 montre un exemple.

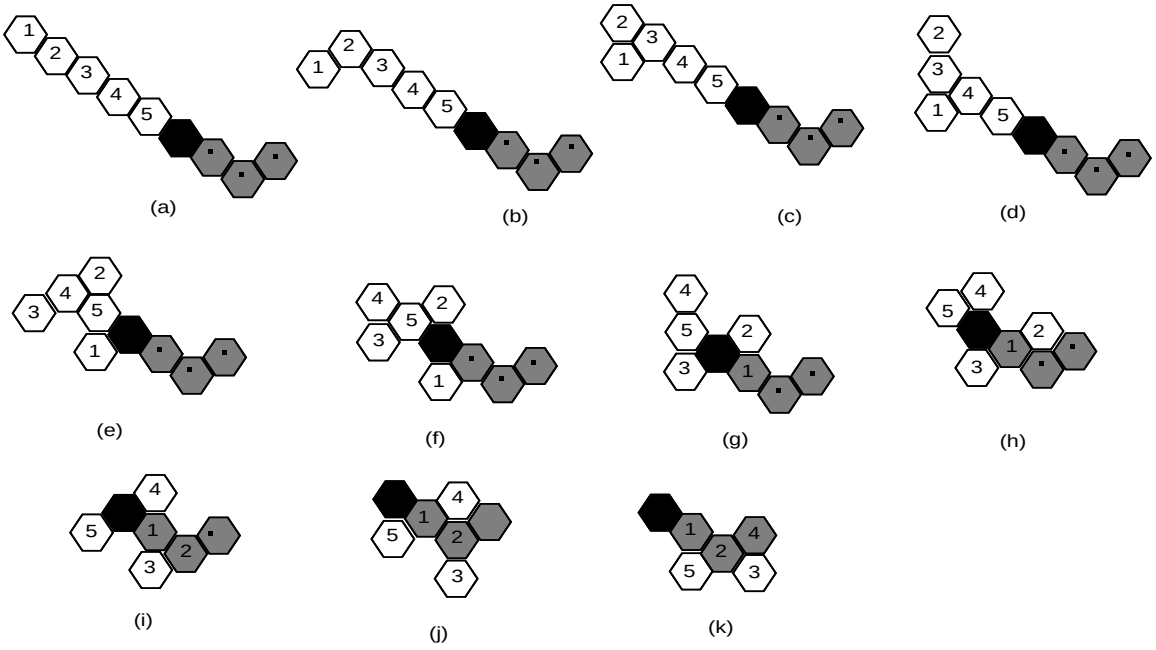

FIGURE 2.10 – illustration de placement des modules quand la longueur de chemin est impaire [6]

B) La phase de reconfiguration distribuée :

Dans cette phase, chaque module calcule son sens de rotation et le délai à attendre avant de se déplacer. Le module détermine alors sa position dans la configuration initiale. Les modules qui accomplissent la reconfiguration du CS n'ont pas à choisir une cible en particulier car le premier module atteignant une cellule du CS achève sa mission. Les modules suivants poursuivent leur parcours vers des cellules cibles sur le flanc (N) ou (S) de CS. Les modules remplissent les colonnes N et S de droite à gauche et de nord et sud de CS. L'espacement simple-cellule entre les modules mobiles est le motif en mouvement le plus efficace. Toutefois, cette distance peut causer des problèmes de blocage lorsque le CS forme un angle aigu avec une colonne de cellules cibles. Pour résoudre ce problème, on doit arrêter définitivement ou temporairement certains modules particuliers dans les zones où la collision ou le blocage peut se produire. Chaque module exécute un algorithme de mise en correspondance dans laquelle certaines cellules sont marquées par un indicateur Attendre (AI) ou Stop (SI). Chaque cellule (AI) a une cellule adjacente vide marquée AttendreCellule (AC) et chaque cellule (SI) a une cellule cible adjacente marquée StopCellule (SC). Les modules mis en correspondance avec une cellule (SC) choisissent les cellules (SC) comme leurs positions finales. Un module mis en correspondance avec une cellule de (AC) s'arrête temporairement à la cellule (AC) lorsque la cellule adjacente de (AI) est occupée. Les modules qui s'arrêtent temporairement dans des cellules (AC) sont associés à la cellule extrême N ou S de la colonne directement à leur droit. Ces modules s'arrêtent temporairement jusqu'à ce que toutes les cellules de l'extrême N ou S dans la colonne de droite sont remplis, lors de la reprise du mouvement. Pour éviter le blocage des modules dans les cellules (AC), des cellules

cibles particulières sont marquées comme RetardSet (RS). Chaque module associé à une cellule cible attend 3 étapes avant de se déplacer hors de la chaîne initiale. Les marqueurs SI et AI ne sont nécessaires que lorsque les modules entrent en conflit approximé des coins angle aigu. Un problème qui est évité si la colonne de la configuration cible immédiatement à droite est assez court. Un exemple dans la figure 2.11 (la quatrième colonne de gauche à droite).

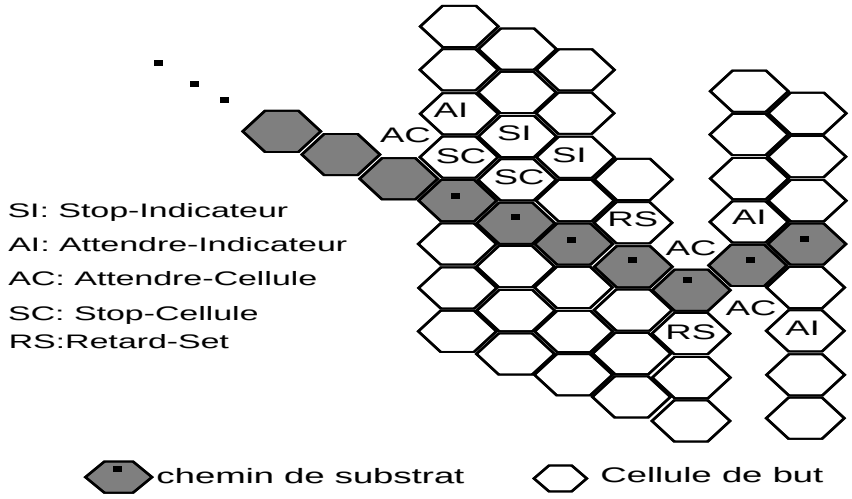

FIGURE 2.11 – Le placement des marqueurs dans la configuration finale pour éviter l'inter blocage [6]

Les modules de AC ou SC empêchent les modules mobiles d'être en contact dans les coins à angles aigus, ceci en occupant le coin avec un module, de manière permanente (SC) ou temporairement (AC). Remplir cette cellule de coin permet au reste des modules dans la colonne immédiatement à droite de se déplacer sur le coin sans blocage.

2.4 Auto-reconfiguration pour des robots cubiques

2.4.1 Auto-reconfiguration en utilisant la croissance dirigée

Dans [89], les auteurs proposent un algorithme centralisé qui utilise les positions prédéfinies de la forme cible pour guider la reconfiguration. Dans l'approche présentée, la configuration souhaitée est produite à partir d'un module graine initial. Les graines conduisent au déploiement progressif de la structure finale par un système de recrutement de gradient. En utilisant la communication locale, les modules libres escaladent le gradient pour atteindre des positions cibles. La progression de la construction est guidée par une nouvelle représentation de la configuration désirée, qui est générée automatiquement à partir d'un modèle 3D CAD. Cette approche présente deux avantages principaux : (1) la représentation est concise, avec une taille proportionnelle à la forme globale plutôt qu'en fonction du nombre de modules et (2) une séparation nette existe entre l'objectif et les règles locales utilisées par les modules.

2.4.1.1 Phase de représentation

Le but de cette phase est de représenter automatiquement le code des règles locales à partir du modèle CAD. Ces règles locales conduisent à la configuration souhaitée en cours d'assemblage. L'approximation de la forme d'entrée est faite en utilisant un ensemble de briques de différentes tailles qui se chevauchent. Ce choix est assez arbitraire, et d'autres formes géométriques de base, telles que des sphères ou des cônes, pourraient être utilisées aussi bien. L'ensemble des briques est généré en commençant par un point spécifié par l'utilisateur à l'intérieur du modèle CAD. L'algorithme détermine alors une brique, la plus grande possible, contenant ce point et qui ne coupe pas le modèle CAD.

De manière récursive, ce procédé est ré-appliqué pour tous les points à l'extérieur de cette brique, mais à l'intérieur du modèle CAD. Ce processus se poursuit jusqu'à ce que le volume ait été rempli de briques qui se chevauchent.

2.4.1.2 De la représentation à l'auto-reconfiguration

En partant d'une configuration aléatoire, les robots doivent se redéployer dans une configuration cible décrite par la représentation CAD. L'algorithme d'auto-reconfiguration se compose de trois étapes : un mécanisme pour propager les coordonnées, un mécanisme pour générer des graines dans le système, et un mécanisme qui permet le déplacement des modules sans se déconnecter de la structure.

A) Propager les coordonnées :

Au départ les modules sont déployés aléatoirement et chaque module a une copie de la configuration finale (chaque module enregistre toutes les positions prédéfinies de la forme cible). Tous les modules commencent dans un état errant. Un module arbitraire a initialement un échantillon aléatoire de coordonnées de points à l'intérieur de la représentation. L'idée est d'étendre la configuration à partir des modules graine. Le module graine va contrôler la reconfiguration en attirant un module errant vers une position voisine d'une cellule cible vacante. Quand un module atteint une cible vacante, en fonction de sa position il peut agir comme un module graine si la poursuite de la construction est nécessaire en cette position. Un module cesse d'agir comme une graine quand tous les modules voisins, spécifiés par la représentation et les coordonnées de la graine, sont en place. Afin de simplifier le problème de reconfiguration, une structure d'échafaudage est appliquée sur la configuration désirée ; les modules voisins ne sont nécessaires que dans des positions qui sont contenues dans la représentation de la brique et qui appartiennent à l'échafaudage. L'introduction de la sous-structure d'échafaudage dans la configuration désirée simplifie le problème de reconfiguration, car au cours de la reconfiguration, on peut supposer que la configuration ne contient pas de minima locaux.

B) La Création du gradient en utilisant des communications locales :

Le gradient est utilisé pour attirer les modules errants vers une position vacante. Le module graine est la source, il envoie un nombre entier, représentant la concentration d'un produit chimique artificiel, à tous ses voisins. Un module non-source calcule la concentration du produit chimique artificiel reçu localement en prenant la valeur maximale reçu par les voisins et en soustrayant 1. Cette valeur est ensuite propagée à tous les voisins et ainsi de suite. Lorsque la concentration atteint zéro, le gradient ne se propage pas davantage. Cela signifie que la source

peut décider de la plage du gradient. Si les modules errants doivent compter sur la valeur de base du gradient pour localiser la source, ils auraient à se déplacer de façon aléatoire pendant un certain temps afin de détecter la direction du gradient. Au lieu de cela, un vecteur gradient est introduit qui rend l'information de direction disponible localement, éliminant ainsi des mouvements inutiles. La mise en œuvre de base du gradient est prolongée par un vecteur indiquant la direction locale du gradient. Ce vecteur est mis à jour en prenant le vecteur du voisin avec la plus forte concentration, auquel est ajouté (dans le sens vectoriel) un vecteur unitaire dirigé vers ce voisin (le sens inverse du vecteur maximal). Les chemins d'accès aux positions cible vacantes passent soit par ou sur la surface de la structure. La structure ne contient pas de minima locaux, en raison de la structure de l'échafaudage. Par conséquent, les chemins d'accès aux postes vacants ne contiennent jamais de minima locaux.

C) Le déplacement des modules :

Les modules errants remontent le vecteur de gradient dans le sens inverse pour atteindre des postes vacants. Malheureusement, les modules errants peuvent ne pas se déplacer indépendamment les uns des autres (contrainte de connexité). Le problème est alors de maintenir le système connecté tout en permettant aux modules errants de se déplacer. Les modules ou groupes de modules, qui sont déconnectés, vont tomber et peuvent être endommagés (effet de la gravité). Le fondement de l'algorithme est que tous les modules finalisés sont connectés, en raison de la façon dont le mécanisme de propagation d'état fonctionne. Les modules errants utilisent un gradient de connexion pour s'assurer qu'ils peuvent se déplacer sans être déconnecté. Les robots restent connectés à condition qu'un module ne se déplace que lorsque : 1) la concentration du gradient de connexion est supérieur à zéro dans le module concerné et ses voisins ; 2) le déplacement ne change pas la concentration du gradient de connexion des modules voisins et 3) il est en outre supposé que quand un module se déplace, son gradient de connexion est mis à zéro.

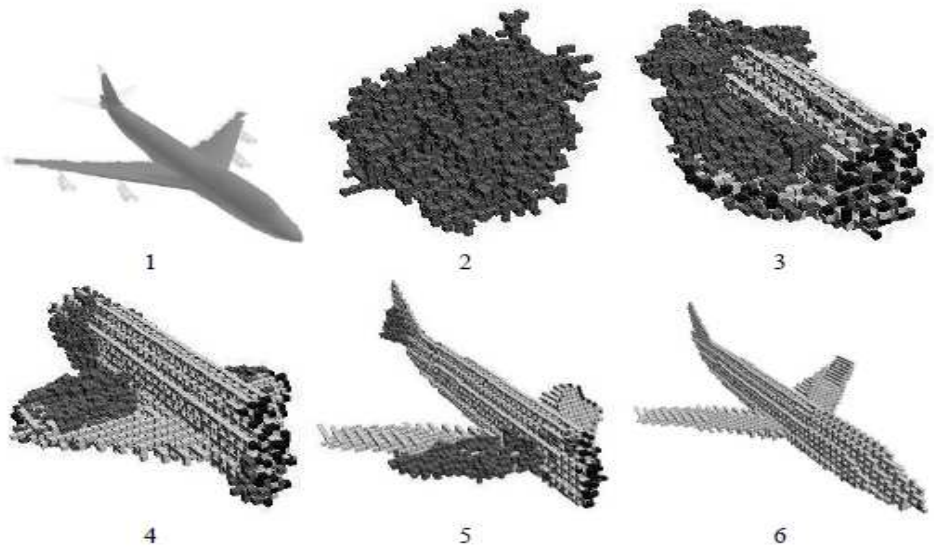

FIGURE 2.12 – Une représentation basée sur des briques générées par CAD, cette représentation est utilisé pour contrôler la reconfiguration [89]

2.4.2 Auto-reconfiguration contrôlée à l'aide d'automates cellulaires

Il s'agit d'une approche centralisée pour achever l'auto-reconfiguration pour des robots cubiques [90, 91]. La configuration souhaitée est produite à partir d'un module de départ initial appelé module graine. Ce module sera le contrôleur de base du processus de reconfiguration. Le module graine produit une croissance en créant un gradient dans le système en utilisant une communication locale, les modules grimpent pour trouver le module de graines. Dans cette solution, la croissance est guidée par un automate cellulaire (von Neumann, 1966), générée automatiquement à partir d'un modèle CAD en trois dimensions ou une description mathématique de la configuration souhaitée. L'intérêt du générateur de règles sous forme d'automates cellulaires est d'éliminer les aspects globaux du problème d'auto-reconfiguration et rendre possible l'utilisation d'un algorithme local pour contrôler le processus d'auto-reconfiguration.

2.4.2.1 Générateur d'automate cellulaire

Le générateur d'automate cellulaire (AC) prend en entrée un modèle en trois dimensions et en sortie produit les règles de l'automate cellulaire correspondant. Le générateur de règles de l'automate cellulaire prend un modèle tridimensionnel fermé et un point de départ indiqué à l'intérieur du modèle. L'algorithme fournit alors un ensemble de règles d'AC, qui précisent les relations de voisinage entre ces états. L'algorithme qui transforme le modèle de CAD tridimensionnel, représentant la configuration souhaitée, en un automate cellulaire, fonctionne comme suit :
1. Le modèle est approché avec une configuration de modules inter-connectés utilisant un procédé similaire au remplissage par diffusion.
2. Des modules sont supprimés de la configuration pour éviter des configurations avec des minima locaux, creux ou à des sous-configurations solides. Un moyen simple d'y parvenir est d'utiliser des échafaudages.
3. Pour chaque module i dans la configuration est attribué un numéro unique $s(i)$.
4. Pour chaque paire voisine de modules i, j, une règle est générée. Cette règle est de la forme : si l'automate cellulaire du module en direction $\longrightarrow ij$ est en l'état $s(j)$, alors cette AC devrait passer à l'état $s(i)$.
5. L'automate cellulaire final contient l'ensemble des règles de l'automate cellulaire et commence dans un état initial appelé errant qui est distinct de toute valeur $s(i)$.

L'introduction de la notion d'échafaudage dans la configuration désirée simplifie le problème de reconfiguration, car on peut supposer que la configuration ne contient pas de minima locaux, de creux ou des sous-configurations solides. Cette simplification signifie que le système est convergent par la conception.

2.4.2.2 De l'automate cellulaire à la configuration souhaitée

L'algorithme d'auto-reconfiguration se compose de trois éléments : un mécanisme de propagation de l'état, un mécanisme pour créer des graines dans le système, et un mécanisme pour se déplacer sans se déconnecté de la structure.

Principe de l'algorithme : Pour propager les états, chaque module a une copie de l'automate cellulaire, et commence par l'état errant. Un module aléatoire prend un nombre d'états

choisi aléatoirement des états $s(I)$ alloués par le générateur de l'automate cellulaire. L'idée est de faire croître la configuration de ce module graine. Les modules voisins reliés au module graine changent d'état selon les règles de l'automate cellulaire. Le module graine peut détecter si un voisin est absent à l'aide de capteurs et des règles de l'automate cellulaires. Si tel est le cas, la graine attire un module errant au poste vacant. Lorsqu'un module atteint une position but non occupée il change d'état et agit à son tour en tant que graine. Ces modules sont alors finalisés. Un module cesse d'agir comme une graine lorsque toutes les relations de voisinage, décrites par les règles d'automates cellulaires, sont satisfaites. Les techniques pour la création du gradient en utilisant des communications locales, et pour le déplacement des modules sont les mêmes que celles présentées dans la section 2.4.1

2.4.3 Auto-reconfiguration centralisée avec des modules unitaires compressibles

Le modèle de robots utilisés dans cette approche porte sur les robots cristallins [76, 88]. Un robot cristallin est constitué d'un ensemble de composants autonomes cristallins, appelé aussi Crystal Atoms [79]. Ces modules sont dotés d'une forme cubique et de connecteurs au centre de chaque face. Un Crystal Atom peut se déplacer par rapport à un autre par dilatation et contraction. Ainsi un composant peut se rétracter jusqu'à une taille équivalente à la moitié de sa taille originale. La figure 2.13 montre une conception de la mécanique d'un Crystal Atom dans sa version 2D.

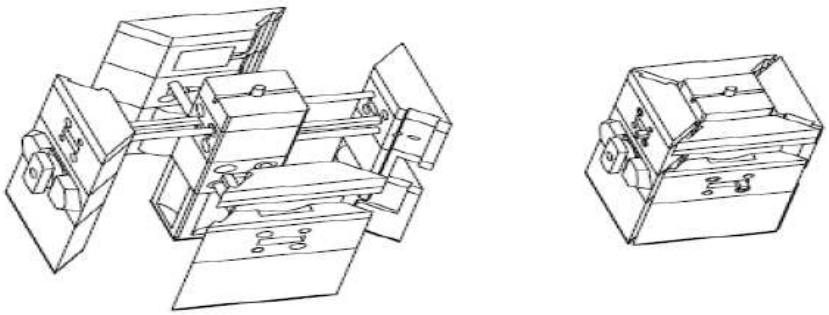

FIGURE 2.13 – Les atomes cristallins en état d'expansion. L'atome fait un carré de 4 pouces, après contraction il fait une longueur de 2 pouces seulement [76]

La figure 2.14 montre un exemple du fonctionnement (action de dilatation, contraction, liaison, libération). Un robot se compose de quatre composants cristallins connectés. Dans la première phase de l'algorithme, l'atome le plus à droite se fixe sur le substrat tandis que les 2 atomes du milieu se rétractent. Cette opération provoque l'avancement de l'atome de gauche d'une unité (où l'unité est représentée par la taille d'un atome). Dans la deuxième phase, l'atome le plus à gauche se fixe au substrat tandis que le module le plus à droite relâche sa connexion au substrat. Les deux atomes du milieu s'étendent alors causant l'avancée d'une unité de l'atome de droite.

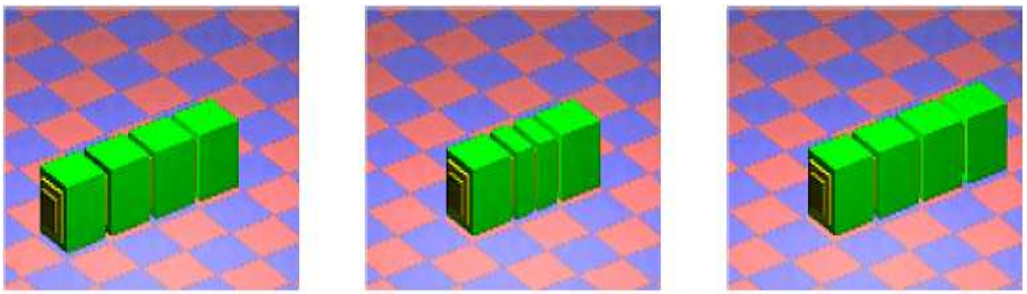

FIGURE 2.14 – Trois clichés d'une simulation utilisant des robots cristallins. L'image de gauche montre l'état initial. L'image du milieu montre le robot après avoir contracté deux atomes. L'image de droite montre le robot après le relâchement des atomes contractés [76]

2.4.3.1 Algorithme de reconfiguration

Cette section décrit le planificateur d'auto-reconfiguration dans les systèmes de robots cristallins. L'algorithme centralisé d'auto-reconfiguration a pour objectif de ramener une forme initiale S vers une autre forme cible G. L'algorithme utilise des cristaux de Grain (4). L'ensemble des grains (4) contient des grains de cristaux qui peuvent être appareillés par des cubes de 4x4, de sorte que l'ensemble de plans (ou les bords en 2D) qui coïncident avec les côtés de l'ensemble des grains se coupent uniquement au niveau des bords et coins des grains. La figure 2.15 montre un exemple d'une Grain(4) d'une autre qui n'est pas Grain(4).

Le sous-ensemble des grains (4) est utile pour la manipulation de l'échelle de l'atome, il est possible d'approcher une forme solide avec une précision arbitraire avec un cristal en grains (4). L'algorithme proposé appelé Fusionner-Grandir, est complet sur un sous-ensemble de Grains (4) de cristaux et s'exécute en temps $O(n^2)$, où n est le nombre d'atomes dans le cristal.

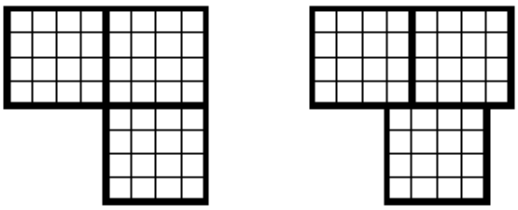

FIGURE 2.15 – à gauche un cristallin Grain(4), à droite cristallin qui n'est pas Grain(4)

L'algorithme Fusionner- Grandir : Dans cet algorithme [76] un cristal intermédiaire I est utilisé à la fois pour maintenir la stabilité au cours de la reconfiguration dans des environnements où la gravité est présente et afin d'éviter le retour en arrière (un autre planificateur pourrait générer des états intermédiaires dans lesquells aucun grain n'est mobile, ce qui nécessiterait un retour en arrière). L'algorithme peut être décomposé en deux parties principales. La partie Fusionner et la partie Grandir. Il est résumé en deux étapes :

1. Fusionner S dans un cristal intermédiaire I.
2. Grandir G à partir de I.

L'algorithme Fusionner fonctionne en trouvant un grain g mobile en S, la transporter à une place dans I, et répéter jusqu'à ce que tous les grains sont en I. De même, l'algorithme Grandir fonctionne en sélectionnant des grains mobiles à partir de I et de les transporter vers des endroits dans G jusqu'à ce que tous les grains soient dans G. Un grain est mobile si son déplacement ne cause pas de cristal déconnecté. La deuxième étape de l'algorithme consiste à faire fondre S en I, la troisième étape consiste à faire grandir I dans G. Ces étapes nécessitent de localiser un grain mobile, localiser une bonne destination pour ce grain, et trouver un chemin pour le grain à la destination. Les grains mobiles en cristal C peuvent être trouvés en recherchant des sommets qui ne sont pas des points d'articulation dans GCG (C), le graphe de connectivité de cristal ($GCG(C)$) est un graphe non orienté dont les sommets représentent les grains dans C et dont les arcs représentent les connexions actives entre les grains voisins. Dans l'algorithme Fusionner, tout grain mobile toujours en S est un moteur approprié. Dans l'algorithme Grandir, tout grain mobile toujours en I est un moteur adapté. Les grains parents (les positions dans G) peuvent être localisés par la recherche de grains qui sont à côté des postes encore à remplir. Dans l'algorithme Fusionner, un tel grain dans I est un parent approprié. Dans l'algorithme Grandir, un tel grain dans G est un parent approprié.

Après avoir localisé un grain mobile et un parent, le grain mobile est transporté dans un espace adjacent au parent avec (1) Trouver un itinéraire à travers le cristal (c'est à dire avec Recherche Profondeur D'abord, à partir du parent dans $GCG(C)$), (2) Décomposition de la route en une séquence de primitives de grains de mouvement, et à partir de là en une séquence de primitives de mouvement, et (3) d'exécution des primitives de mouvement d'atomes. Ce qui suit sont les primitives de mouvement des grains :

– (scrunch <grain, dimension, sense>) : créer une compression planaire en un grain mobile à l'une de ses six faces (+x, -x, +y, -y,+z, -z)
– (relax <grain>) : étendre une compression sur une face d'un grain à grain en progrès
– (transfer <grain>) : transférer une compression sur une face d'un grain à grain adjacent
– (propagate <grain>) : déplacer une compression sur une face d'un grain à la face opposée du grain
– (convert <grain, dimension, sense>) : déplacer une compression à une face d'un grain à l'une des faces dans une dimension orthogonale

Tout grain mobile dans un grain (4) peut toujours être chiffonné, après il peut toujours être transporté à des grains de parent d'une séquence de transfert, propager, et conversion d'opérations. La figure suivant montre un exemple.

2.4.4 Auto-reconfiguration distribuée avec modules unitaires compressibles

Dans l'algorithme, nommé PacMan [17, 18], une solution distribuée pour la reconfiguration de microrobots cristallins compressibles est proposée. Cet algorithme permet un parallélisme des mouvements pour la reconfiguration rapide et ainsi améliorer les performances de l'algorithme. Dans cette solution, un planificateur exécuté sur chaque atome détermine le chemin que doit suivre le module pour mettre en œuvre la forme cible. Les chemins sont ensuite parcourus

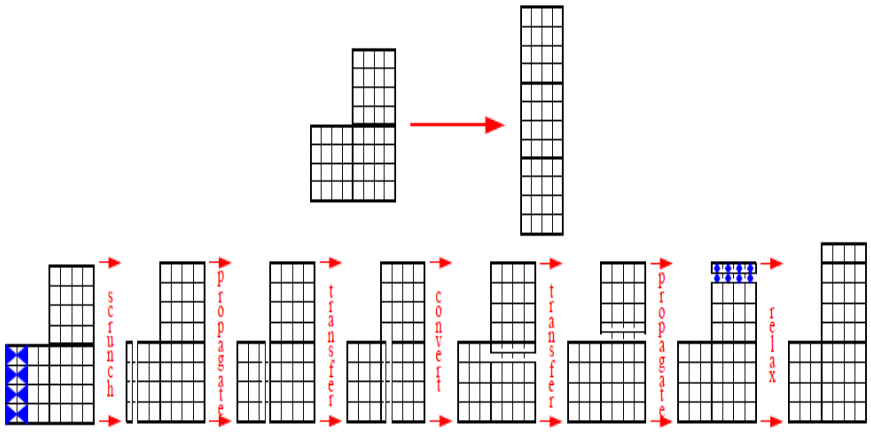

FIGURE 2.16 – Sept étapes pour la relocation de Grain à partir de 5 primitives de déplacement, illustrant comment les primitives peuvent être combinées pour affecter la relocation d'une grain aléatoire. En haut de la figure la relocation désirée est indiquée [76]

par les atomes dans un mode distribué parallèle. L'algorithme PacMan s'inspire du jeu vidéo de même nom, dans lequel le personnage principal mange des "pellets" tout en se déplaçant sur la carte. L'algorithme utilise la structure de données appelée pellets comme un moyen de marquage du chemin que doit suivre chaque module pour effectuer sa partie de reconfiguration. Cette représentation est bien adaptée à la notion d'actionnement de l'unité compressible, dans laquelle le mouvement des modules a lieu à l'intérieur de la structure, et non sur la surface. Avec PacMan, un module physique unique ne suit pas le chemin d'accès complet. Au lieu de cela, un module voyage virtuellement dans la voie tracée par ces pellets. Pour ce faire, il échange son identité avec d'autres modules le long du chemin, tandis que les modules physiques ne se déplacent que localement. Un exemple simple de reconfiguration avec PacMam est montré dans la figure 2.17. Le processus de PacMan est double. Tout d'abord, un chemin est planifié pour chaque module de manière distribuée, le résultat de la planification est un ensemble de pellets réparties à travers les atomes. Une fois que les pellets sont en place, l'actionnement se produira de façon asynchrone. Chaque atome cherche les pellets et les *mange* sans respecter un horaire strict. Cela signifie que la structure intermédiaire du cristal sera indéterminée, mais la structure finale sera déterminée.

2.5 Auto-reconfiguration pour des microrobots sphériques

2.5.1 Claytronics Atoms

Claytronics, est le nom d'un projet mené en collaboration par l'Université de Carnegie Mellon et la société Intel Corporation. Claytronics se veut une instance de matière programmable [37, 38] dont l'objectif principal est de s'organiser sous forme d'un objet de manière à fournir le meilleur rendu visuel depuis l'extérieur. Claytronics est constitué de composants individuels, appelés catoms (pour Claytronic Atomes) qui peuvent se déplacer dans un environnement 3D

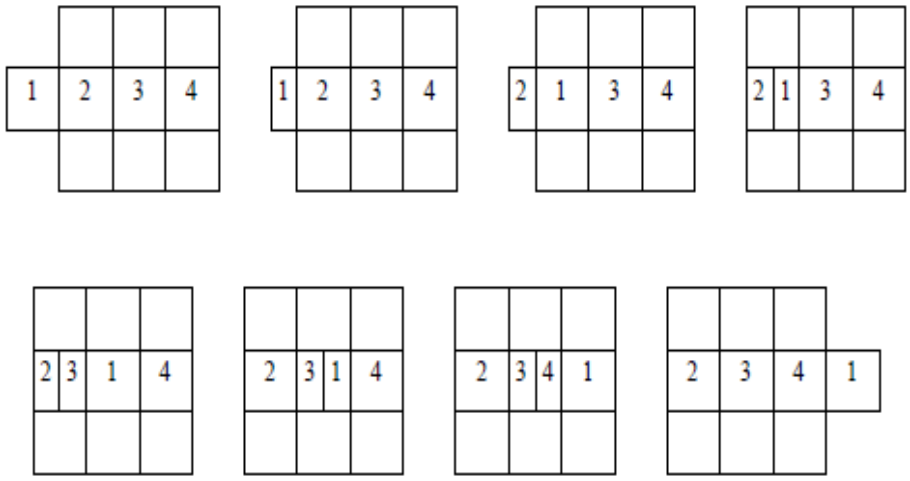

FIGURE 2.17 – Exemple de reconfiguration distribuée avec des modules unitaires compressibles. Les chiffres sont les identités des modules. [17]

à l'aide des autres catoms, adhérer à d'autres catoms pour maintenir une forme 3D, et communiquer avec les catoms voisins [48, 36]. La figure 2.18 présente une image de deux catoms cylindriques, la figure 2.19 présente une image de deux catoms sphériques et la figure 2.20 présente un exemple d'un catom millimétrique. Dans la mise en évidence, deux catoms se déplacent par rapport à l'autre en alimentant en coopération une paire d'aimants au long de leurs surfaces extérieures [45]. Chaque catom est une unité d'un seul bloc avec une unité centrale, un accumulateur d'énergie, un dispositif de télécommunication, un dispositif de sortie vidéo, un ou plusieurs capteurs, un moyen de déplacement, et un mécanisme pour adhérer à d'autres catoms. Une exigence fondamentale du Claytronics est que le système doit évoluer pour permettre la coordination d'un très grand nombre de catoms. Ces systèmes ont de nombreuses applications, telles que la télé-présence, l'interface homme-machine, et le divertissement. Les problématiques de recherche dans le domaine de l'auto-reconfiguration et l'auto-localisation occupent une bonne partie des travaux menés dans le cadre du projet Claytronics [31, 33, 34].

La surface d'un catom est quadrillée par des mécanismes de fixation et de détachement, en se basant sur des électro-aimants qui s'activent et se désactivent.

L'idée est donc d'avoir des centaines de milliers de microrobots formant des objets de forme quelconque, de la même façon que les cellules dans un organisme complexe où chaque petit élément de l'ensemble est dédié à un rôle spécifique réalisé à l'aide de communications entre les microrobots et conduisant à la construction de la forme finale [20, 21, 39, 2].

2.5.2 Les simulateurs pour Claytronics

Afin d'utiliser des systèmes robotiques modulaires à grande échelle comme dans Claytronics, des simulateurs censés reproduire le fonctionnement d'un grand ensemble de microrobots sont développés. Deux raisons principales pour ce besoin : les catoms n'existent pas encore dans leur forme finale raffinée, et deuxièmement, les ambitions du projet Claytronic nécessitent des algorithmes spécifiques qui ne peuvent pas être testés sur des entités robotiques modulaires plus

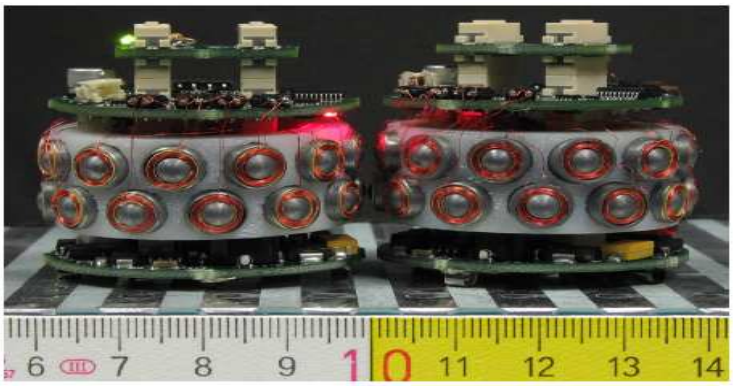

FIGURE 2.18 – Deux catoms cylindriques

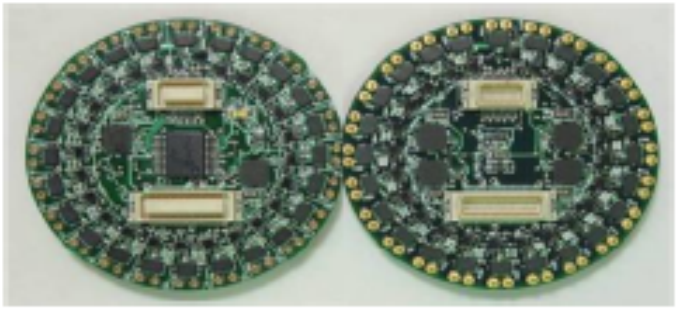

FIGURE 2.19 – Deux catoms sphériques

primitifes [2].

A cette fin, les équipes de recherche travaillant sur le projet ont développé un simulateur de calcul appelé DPRSsim [110] et un langage déclaratif MELD [3] pour les modèles Claytronics permettant ainsi aux programmeurs de tester différents algorithmes et paramètres.

2.5.2.1 Le langage déclaratif MELD

MELD est un langage logique de haut niveau pour la programmation des microrobots modulaires [3, 4]. MELD est un langage déclaratif [78] inspiré de P2 [60] avec une syntaxe similaire à Prolog. L'exécution d'un programme MELD est automatiquement distribuée, et le même programme est exécuté à travers l'ensemble des nœuds du système. Un programme MELD se compose de règles qui précisent les conditions préalables suffisantes pour tirer des faits nouveaux de ceux qui existent déjà. Un avantage clé de Meld est qu'il permet au programmeur de se concentrer seulement sur les aspects logiques de traitement de l'information d'un algorithme, tout en prenant automatiquement en charge les mécanismes de programmation distribuée, comme la communication. Par exemple, l'algorithme de construction de l'arbre couvrant peut être spécifié par deux règles : une règle qui détermine la racine de l'arbre, et une règle qui permet à un nœud de rejoindre l'arbre couvrant qui s'étend à l'un de ses voisins. D'une manière similaire, Meld simplifie la mise en œuvre d'autres structures de données distribuées.

FIGURE 2.20 – Catom millimétrique

A) Composition d'un programme MELD

Un programme Meld se compose d'une base de données de faits et un ensemble de règles de production et génère de nouveaux faits. Un fait Meld est un prédicat et un tuple de valeurs, le prédicat dénote une relation particulière dont le tuple est un élément. Les faits représentent l'état du monde basé sur des observations et des entrées (par exemple, des lectures de capteurs, de connectivité ou d'information de topologie, les paramètres d'exécution, etc), ou reflètent l'état interne du programme. A partir d'une première série d'axiomes, de nouveaux faits sont dérivés et ajoutés à la base de donnés que le programme exécute. En plus des faits, des actions sont également générées. elles sont syntaxiquement similaires à des faits mais provoquent des effets secondaires qui modifient l'état du monde plutôt que les dérivations de faits nouveaux. Dans une application de la robotique, par exemple, les mesures sont utilisées pour amorcer des dispositifs de déplacement ou de commande.

Un concept important dans Meld est l'agrégat (aggregate). Le but d'un agrégat est de définir un type de prédicat qui combine les valeurs dans une collection de faits. Les agrégats peuvent utiliser des fonctions arbitraires pour calculer la valeur totale. Dans la pratique, tel que décrit par [108], le programmeur implémente cette forme de trois fonctions : deux pour créer un ensemble et une pour récupérer la valeur finale. Les deux premières fonctions sont constituées d'une fonction pour créer un agrégat à partir d'une seule valeur et la seconde fonction met à jour la valeur actuelle d'un agrégat ayant une autre valeur. La troisième fonction, qui produit la valeur finale de l'agrégat, permet de conserver un état plus nécessaire pour calculer l'agrégat. Par exemple, un agrégat pour calculer la moyenne dépend de la somme de toutes les valeurs et le nombre de valeurs observées. Lorsque la valeur finale de l'ensemble est demandée, la valeur courante de la somme est divisée par le nombre total de valeurs observées pour produire la moyenne requise.

B) Exemple d'un programme MELD

Règle1 : $Dist(S, D) : -At(S, P),$
. $P_d = destination(),$
. $D = |P - P_d|,$
. $D > rayondurobot.$
Règle2 : $PluLoin(S, T) : -Neighbor(S, T),$
. $Dist(S, D_S),$

. $Dist(T, D_T)$,
. $D_S > D_T$.
Règle3 : $MoveAround(S, T, U) : -PluLoin(S, T)$,
. $PluLoin(S, U)$,
. $U \neq T$.

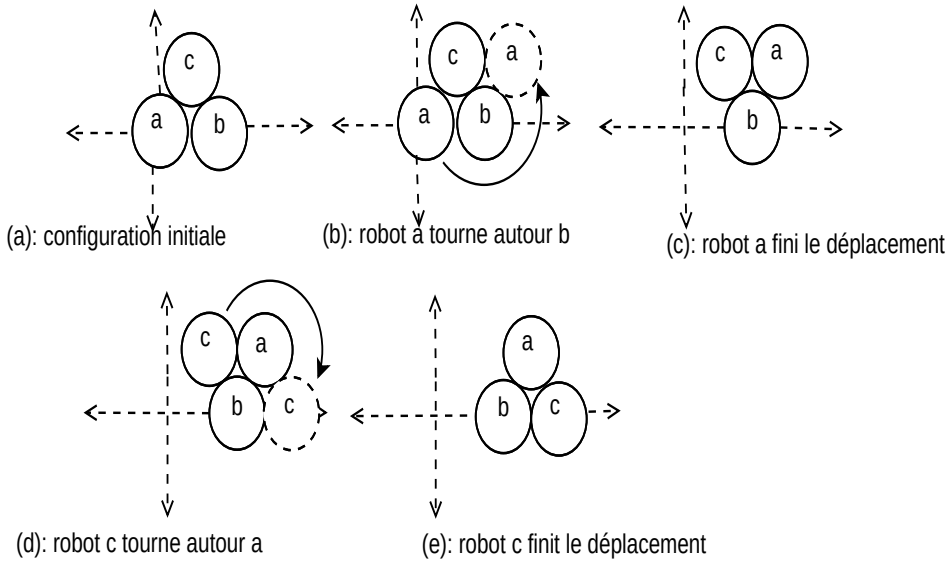

FIGURE 2.21 – Illustration du déplacement des robots

(a) Faits de base :

$Neighbor(a, b)$	$Neighbor(b, c)$
$Neighbor(a, c)$	$At(c, (1, \sqrt{3}))$
$Neighbor(b, a)$	$Neighbor(c, b)$
$Neighbor(c, a)$	$At(b, (0, 2))$

(b) Faits ajoutés après l'application de Regle1 :

$Dist(b, 2)$ $Dist(a, 4) Dist(c, 2.5)$

(c) Faits ajoutés après l'application de Regle2 :

$PlusLoin(a, b)$	$PlusLoin(c, b)$
$PlusLoin(a, c)$	

(d) Faits ajoutés après l'application de Regle3 :

$MoveAround(a, c, b)$ $MoveAround(a, b, c)$

2.5.2.2 DPRSIM (Dynamic Physical Rendering Simulator)

DPRSIM est un simulateur de Catoms développé par Carnegie Mellon University et Intel Labs Pittsburgh [110]. Il simule des microrobots qui sont en mesure d'effectuer le calcul des données, l'enregistrement local, et les mouvements. DPRSim fonctionne sur une seule machine et est capable de simuler des dizaines de milliers de robots.

DPRSim a été le premier simulateur développé qui permettait d'effectuer de grandes expérimentations dans Claytronics (2.22). Il s'agit d'un système intégré qui comprend l'exécution de code, la simulation des phénomènes physiques, la visualisation interactive, le débogage, et un monde basé sur une interface utilisateur graphique pour construire des scénarios d'expérimentation. Bien que destiné à être évolutive, l'objectif principal de son développement était de créer une plate-forme de simulation, qui est suffisamment riche en fonctionnalités pour décrire des applications et des prototypes initiaux pour Claytronics.

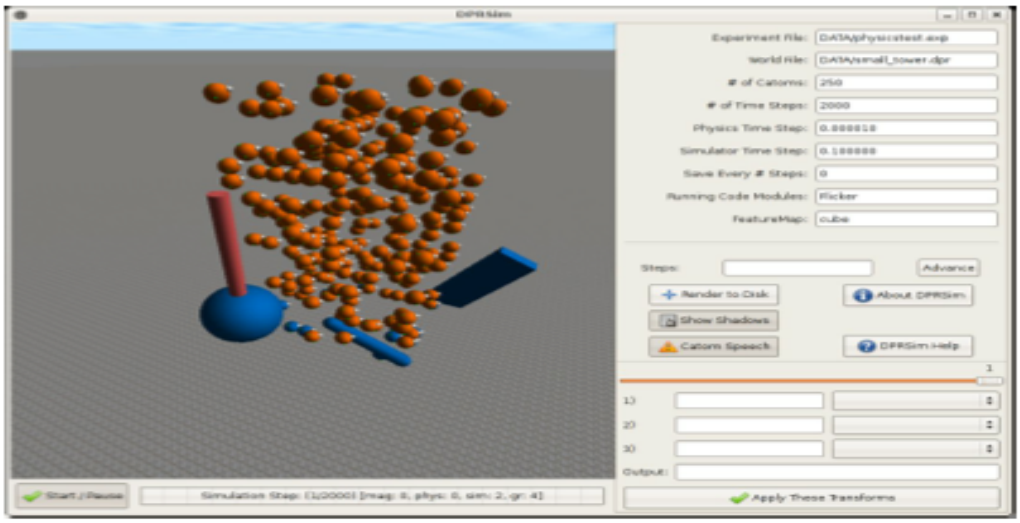

FIGURE 2.22 – Capture d'écran du simulateur de DPRSim original pour Claytronics. Il intègre la simulation de l'exécution distribuée de code, la physique, la visualisation, et le support de débogage interactif

2.5.3 Auto-reconfiguration sur les Claytronics Atoms

2.5.3.1 Auto-reconfiguration avec une décomposition médiane hiérarchique

Les auteurs dans ce papier [75] présentent un algorithme évolutif distribué de reconfiguration, utilisant un algorithme de décomposition médiane hiérarchique, pour atteindre une configuration cible quelconque. Cet algorithme nécessite de connaitre les positons prédéfinies ou la carte de la configuration cible. L'algorithme distribué contrôle le mouvement collectif d'un ensemble de robots par l'intermédiaire du contrôle de la forme du groupe. La solution présentée est entièrement distribuée et ne nécessite pas de communication globale. L'idée de base est que chaque nœud tente d'abord de connaître sa propre position au sein du réseau en utilisant un système GPS ou des techniques de multilateration [10, 98], puis calcule sa position cible. Ceci est accompli grâce à l'algorithme de décomposition médiane hiérarchique (HMD). L'algorithme de décomposition médiane hiérarchique est basé sur l'estimateur médian de consensus [64]. Un ensemble de nœuds qui ont peut-être des estimations initiales différentes sur une variable globale tentent de parvenir à un accord sur la valeur réelle de la variable. L'idée de base de l'algorithme est d'établir une bijection de la configuration initiale à la configuration cible. L'algorithme est constitué de deux étapes : la première établit la position relative de l'agent, et la seconde établit la position

définitive de l'agent dans la configuration cible. Dans la première étape les nœuds essayent de parvenir à un accord sur la valeur médiane des coordonnées x et y de leurs positions. L'algorithme construit itérativement un $kd-tree$, où l'espace est divisé en quadrants, les côtés gauche et droit sert à déterminer la coordonnée x, et les côtés haut et bas pour déterminer la coordonnée y, comme illustré dans la figure 2.24. La deuxième étape de l'algorithme est essentiellement l'inverse de la première étape.

Chaque robot exécute une procédure dont la complexité est de $O(log(n))$ de mémoire pour réaliser cette bijection.

Principe de l'algorithme : L'algorithme se déroule sur une configuration initiale connectée. Les nœuds ont un système de coordonnées uniforme et connaissent leur emplacement dans le système de coordonnées. Les nœuds ont au moins une représentation grossière de la configuration cible. L'algorithme HMD se compose de deux phases : 1) Chaque nœud calcule sa position par rapport à d'autres nœuds de manière répartie, et 2) Chaque nœud utilise ensuite cette position pour établir sa position dans la forme cible.

1) Établir la position relative

Les nœuds essayent de parvenir à un consensus sur la valeur médiane des valeurs x de coordonnées de leurs positions. Les nœuds initialisent d'abord leur identités par un littéral indiquant le côté de la médiane qui se trouvant sur la gauche (L) ou la droite (R). Après avoir atteint un consensus, chaque groupe d'agents ayant le même ID, en parallèle, atteint un consensus sur leurs valeurs y de coordonnées (figure 2.23, étape 2), et ajoute à leur ID d'un littérale vers le haut (U) ou vers le bas (D). Ce consensus médian se déroule de façon itérative entre les agents ayant la même identification, et à la fin de chaque itération, le nombre d'agents ayant le même ID réduit jusqu'à ce que chaque agent ait un identifiant unique et ainsi une position relative dans le système. La figure 2.23 montre un exemple.

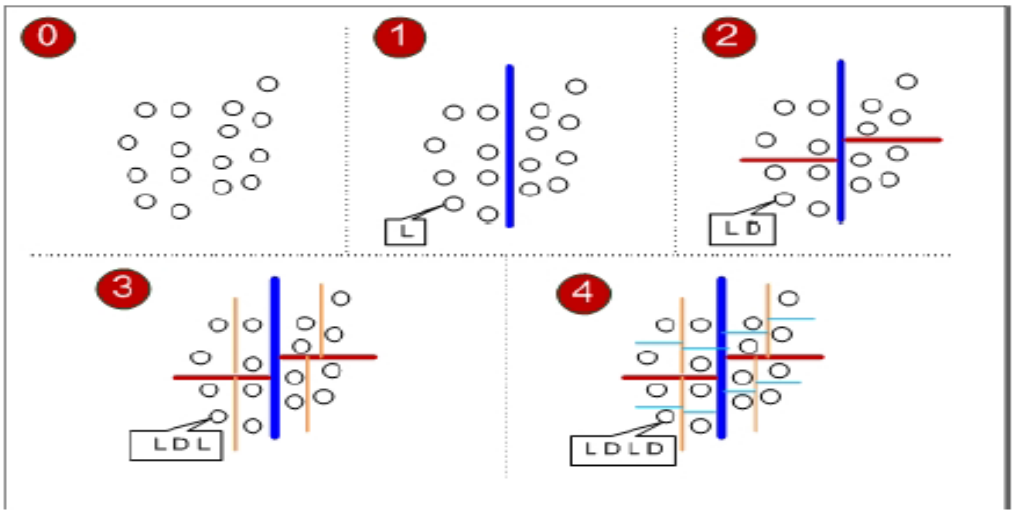

FIGURE 2.23 – montre les itérations pour trouver consensus médian, Décrit aussi la construction d'un ID pour un nœud arbitraire [75]

2) Détermination de la position finale

Dans cette partie de l'algorithme, le nœud tente de déterminer son emplacement final unique

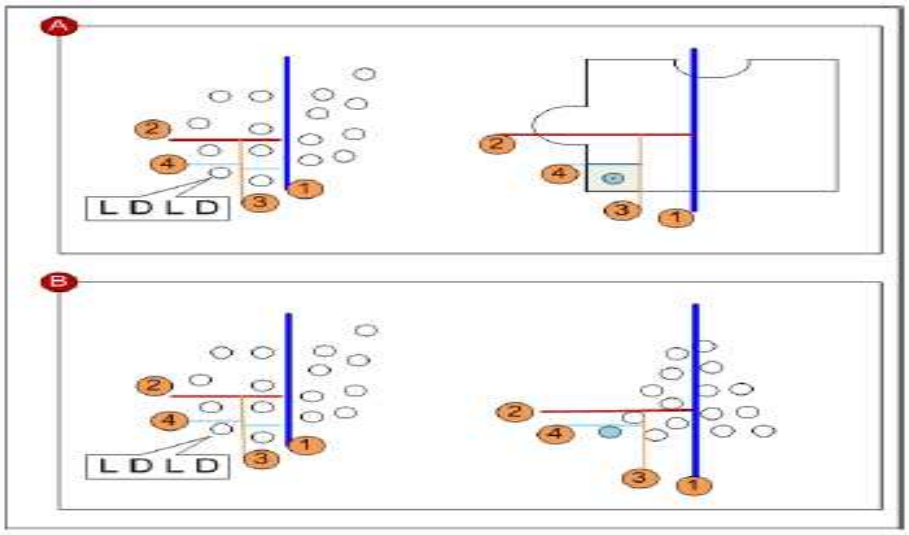

FIGURE 2.24 – en utilisant la position relative de la configuration initiale (côté gauche) pour déterminer la position dans la configuration finale (côté droit). Les chiffres indiquent l'ordre de plans de séparation. (A) représente un intance où une carte approximative est connue. (B) représente une application de formation finale est exacte [75]

dans la configuration cible. Cette étape est fondamentalement l'inverse de l'étape précédente (voir la partie droite de la figure 2.24). Chaque nœud boucle dans chaque littéral dans son ID (construit dans l'étape précédente), et il cycle à travers les axes de coordonnées pour sélectionner les plans de séparation dans la configuration cible. A chaque étape, le point choisi pour créer le plan de division est la médiane des points étant mis dans le kd-arbre, à l'égard de leurs coordonnées dans l'axe utilisé.

2.5.3.2 Un planificateur de mouvement hiérarchique

La planification de la reconfiguration est une tâche de recherche basée sur l'espace de la configuration. L'addition de chaque module non seulement augmente exponentiellement cet espace, mais aussi augmente sa dimension en raison de ses degrés de liberté. Les méthodes hiérarchiques sont un moyen de décomposer une tâche de recherche en sous-tâches plus faciles à gérer pour améliorer les chances de trouver une solution. Des méthodes hiérarchiques ont été proposés comme un moyen de résoudre les reconfigurations [19], en particulier pour les planificateurs déterministes. Une structuration naturelle d'une recherche hiérarchique implique un sous-programme qui détermine les délocalisations appropriées des modules, et appelle un sous-programme de niveau inférieur pour définir l'ensemble des actions valides permettant de déplacer avec succès le module vers sa position cible.

Dans cette solution les auteurs proposent une approche hiérarchique de planification de mouvement pour les systèmes avec plusieurs milliers de modules. Cette approche consiste en un planificateur de base qui calcule une solution optimale pour quelques modules et un planificateur hiérarchique qui appelle ce planificateur de base ou réutilise les plans pré-calculés à chaque niveau de la hiérarchie pour calculer finalement une solution optimale globale.

A) La Hiérarchie

Les hiérarchies sont construites en utilisant des groupes de catoms appelés meta-catoms. Un meta-catom MC dans un niveau i est une collection de sept meta-catoms dans le niveau $i-1$ formant une structure hexagonale. Le meta-catom dans le niveau i a six aimants actifs dans sa périphérie représentés par les grands cercles. Les autres aimants ne prennent pas part au mouvement au niveau i. Ainsi, un méta-catom à n'importe quel niveau est fonctionnellement similaire à un catom. Une configuration est une collection de catoms ou méta-catoms.

L'approche hiérarchique peut être perçue intuitivement, montant sur une feuille de route de la configuration de départ, traversant cette feuille de route jusqu'à ce que nous rapprochons de la configuration objectif, puis descendant la feuille de route pour la configuration de but. Le procédé de la montée et la descente de la feuille de route correspond à la configuration dans et hors d'une structure du modèle hiérarchique. Cette étape pourrait être très difficile à calculer. Il existe différentes approches pour résoudre ce problème, comme l'utilisation de règles locales soigneusement définies, pour passer rapidement d'une configuration à une Template de configuration hiérarchique. La structure imposée sur la hiérarchie nous permet de pré-calculer des mouvements optimaux déconnectés sous la forme de modèles de mouvement et de réutiliser ces modèles à n'importe quel niveau de la hiérarchie.

B) Modèles de mouvement

Les modèles de mouvement sont des mouvements pré-calculés qui sont réutilisés de façon récursive pour générer des plans au niveau des catoms. Un modèle prend en entrée un mouvement au niveau i et délivre une séquence de mouvements au niveau $i-1$ qui produisent le mouvement requis. Un exemple d'un modèle de mouvement est représenté sur la figure 2.25. Les grands cercles représentent les méta-modules au niveau i de la hiérarchie, chaque méta-module est constitué de sept modules de méta-modules de niveau $i-1$, comme indiqué par les petits cercles. Le mouvement d'entrée est une rotation à droite (CW) du méta-module du centre sur son ancre. Les configurations de début et cible sont présentées, respectivement, en haut à gauche et en bas à gauche. Les méta-modules au niveau $i-1$ du méta-module mobile sont numérotés de 0 à 6. La sortie du modèle est un plan de déplacement pour ces sept méta-modules. Le plan de mouvement est généré pour les sept méta-modules. Deux configurations intermédiaires du plan de mouvement résultant sont présentées en haut à droite et en bas à droite de la figure 2.25.

C) Planificateur de base

Le planificateur de base est au sommet du plus haut niveau de la hiérarchie et génère également des modèles de mouvement. Il utilise la méthode de recherche classique, guidée par des heuristiques pour planifier un petit nombre de modules. Dans l'heuristique gloutonne du plus proche voisin (greedy nearest neighbor), on calcule le coût d'une configuration en le comparant à la configuration cible de la manière suivante : Le coût de chaque module est la distance euclidienne entre lui-même et son voisin le plus proche dans la configuration cible, normalisé par le diamètre du catom. Le coût de l'ensemble de la configuration est la somme des coûts de tous les modules pour les modules de la configuration. Cette heuristique provoque la recherche pour élargir les nœuds avec un coût croissant. Ainsi, la méthode de recherche classique est garantie pour produire un chemin optimal en termes de nombre de mouvements.

Une autre heuristique a été étudiée, appelée **greedy nearest mismatched neighbor**. Dans cette heuristique, la configuration courante et la configuration objective sont d'abord trai-

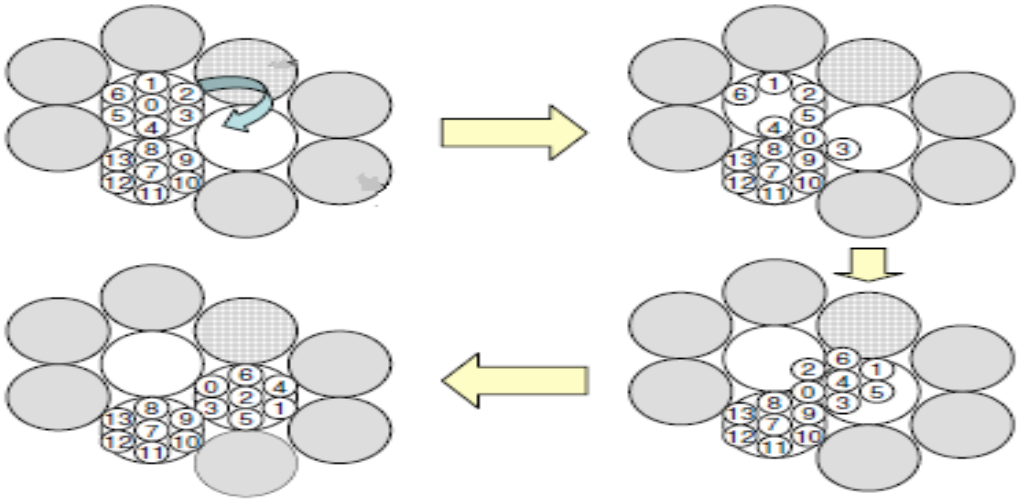

FIGURE 2.25 – Exemple d'un modèle de mouvement [19]

tées de sorte que tous les modules qui sont déjà dans des positions objectifs dans la configuration courante sont enlevés et les modules correspondants de la configuration de l'objectif sont également supprimés. Apres on calcule le coût avec la même méthode. Intuitivement, ceci signifie que les modules tentent de se déplacer vers les positions de but libre et non pas vers les postes qui sont déjà occupées. Aussi, l'heuristique n'essaye pas de déranger les modules qui sont déjà dans les positions finales. Les raisons principales pour choisir les deux heuristiques ci-dessus sont la simplicité, la facilité de mise en œuvre, la vitesse et l'optimalité.

D) Algorithme de planification hiérarchique

La configuration initiale et la configuration finale sont d'abord converties en une structure hiérarchique. Ce n'est pas fait automatiquement dans le planificateur de prototype. Une hiérarchie parfaite peut être formé avec des multiples de puissances de 7 catoms. Si, toutefois, la configuration n'est pas une puissances de 7 catoms, des modèles de mouvement doivent être créés pour les méta-catoms avec moins de 7 catoms. Le planificateur de base est alors appelé à planifier au sommet plus haut niveau de la hiérarchie pour reconfigurer les méta-modules de la forme de départ à la forme de but.

Une seule étape au niveau i est traduite en une séquence d'étapes au niveau $i-1$ soit par la réutilisation du plan donné par un modèle de mouvement, le cas échéant, ou en appelant le planificateur de base pour générer un plan. Chaque étape du plan généré au sommet plus haut niveau de la hiérarchie se traduit donc dans un mode en profondeur d'abord jusqu'à ce qu'il représente le déplacement au niveau des catoms. Enfin, on transforme à partir de la structure hiérarchique à la configuration de but avec les mêmes techniques utilisées pour générer les hiérarchies.

Le planificateur de base génère des plans optimaux, et les plans du modèle de mouvements pré-calculés sont également optimaux. Cependant, puisque les plans sont combinés à chaque niveau de la hiérarchie, les plans au niveau des catoms ne sont pas optimaux à l'échelle globale. Cette perte d'optimalité est un le prix à payer pour l'augmentation de la vitesse de la planification, et l'étendue des problèmes qui peuvent être résolus pratiquement. En outre, comme

l'approche est conçue pour des reconfigurations au niveau global, cette méthode converge de manière plus cohérente que les méthodes purement locales, à base de règles.

2.5.3.3 Méta-modules généralisés et planification de forme

L'élaboration des méta-modules a pour objectifs de réduire les contraintes de mouvement des modules à cause du blocage, et de réduire la complexité de la planification [29]. Le méta-module lui-même est utilisé comme unité de base de fonctionnement. Les auteurs dans cette solution développent des méta-modules généralisés et un planificateur d'auto-reconfiguration associé. Un méta-module peut être créé ou détruit à tout point du réseau, aussi longtemps que la création ou la suppression se produit à côté d'un méta-module existant. Ainsi, le système se comporte comme un fluide compressible qui peut se dilater ou se contracter librement. Les méta-modules sont capables de contenir un nombre variable de modules, tout en maintenant la stabilité physique et la connectivité de la structure globale. Les méta-modules se déplacent en échangeant des modules ou des méta-modules ou en absorbant les modules d'un voisin et les déplacer vers un autre endroit.

A) **Les contraints de mouvements**

Contraintes non holonomes : Une contrainte non holonome est une contrainte qui ne permet pas d'aller à une configuration adjacente en une seule étape (un seul déplacement) à cause des contraintes de blocage mécanique. La figure 2.26 présente un exemple d'une contrainte non holonome. Les deux états (a) et (e) diffèrent uniquement par la position d'un seul module. Toutefois, en raison des contraintes de blocage mécanique, quatre transitions sont nécessaires pour passer de la première configuration (a) à la configuration (e).

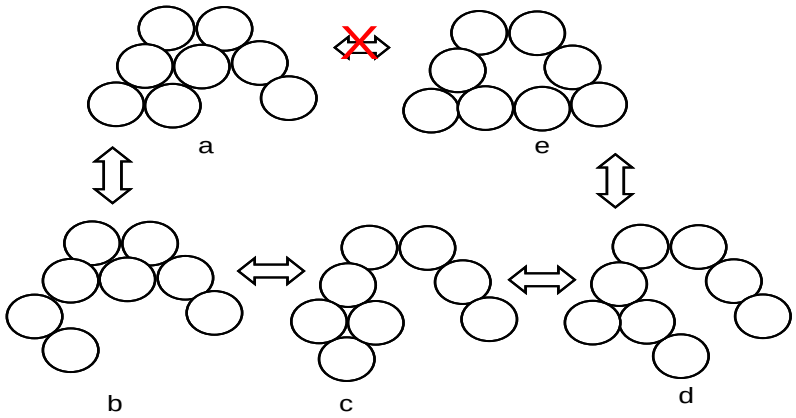

FIGURE 2.26 – La configuration (e) n'est pas accessible en une seule étape à partir de la configuration initiale (a), à cause d'une contrainte mécanique [29]

Contraintes locales et non-locales : La contrainte locale exprimée par ce générateur proposé est que l'action de se déplacer autour d'un voisin V à un de ces sens : haut, gauche, bas et droite, nécessite l'absence de tous les voisins dans ce sens. Par exemple, si un nœud A souhaite se déplacer vers la position NE de son voisin V qui situe au côté W, il faut que A n'ait pas de voisins dans les sens NE et NW.

Une contrainte non-locale, permet de déterminer si la configuration est admissible ou inadmissible. Les contraintes non-locales peuvent inclure n'importe quelle partie du système afin de déterminer l'admissibilité. Ces contraintes peuvent, par exemple, être utilisées pour maintenir des propriétés globales telles que la connectivité et la stabilité physique. Les contraintes non-locales sont holonome, chaque configuration adjacente est soit accessible par l'application d'un seul générateur à l'état actuel, soit inaccessible par le système de contraintes.

B) **Les méta-modules proposés**

Trois méta-modules ont été proposés : Pixel, Genral1, et Genraln.

1. Pixel :

Les modèles de méta-modules holonomes libèrent le système de contraintes de blocage. Pixel est un système métamorphique holonome de style de treillis. Chaque point du réseau est soit vide, soit titulaire d'un méta-module, qui est à son tour composé d'un groupe de modules de base. Au niveau 'méta-module', le système fournit un ensemble de modèles pour manipuler les méta-modules et l'état des points du réseau. Sous cette abstraction, le système fournit aussi des séquences planifiées de mouvements pour les modules individuels pour mettre en œuvre les modèles sur les méta-modules. Ce système a un seul modèle défini : créer/détruire, qui passe de l'état d'un point de maille entre "vide" et "méta-modules présent" :
creat/destroy : $\Delta \Leftrightarrow M$

Avec le modèle Pixel, les méta-modules peuvent apparaître et disparaître spontanément n'importe où, indépendamment de la présence ou de l'absence des méta-modules d'autres points du réseau. Le système de pixel a deux contraintes non-locales : la première est la connectivité. Seuls les états où les configurations dans lesquelles tous les modules présents forment un groupe connecté sont autorisées. La deuxième contrainte non-locale est la conservation (min, max), qui n'autorise que des configurations avec un nombre de modules inférieur ou égal à max et supérieur ou égal à min. Les contraintes de ce système sont holonomes, puisque n'importe quel module peut apparaître ou disparaître tant que les contraintes non-locales ne sont pas violées.

2. Général1 :

Le modèle de méta-module Pixel n'est pas réalisable dans la pratique, car il n'est pas conservateur de masse. En outre, la notion de pixel s'appuie fortement sur une contrainte non-locale pour forcer la connectivité. Pour rendre le système conservateur de masse en s'appuyant fortement sur une contrainte locale pour garantir la connectivité, un nouveau système est défini, général1 avec les modèles suivants :
creat/destroy : $C\Delta \Leftrightarrow DD$
transfer : $CD \Leftrightarrow DC$

Avec général1, un module est créé seulement par un voisin déjà présent. Les méta-modules peuvent être dans deux états différents : dans l'état D, le méta-module est composé d'un nombre minimum de modules, et il a une pièce interne pour tenir des modules supplémentaires, et dans l'état C, l'espace supplémentaire est rempli, de sorte que le méta-module est composé de deux fois autant de modules. Le modèle créer/détruire est maintenant conservateur de masse, puisque le nombre de modules (pas les méta-modules) reste inchangé. Un modèle supplémentaire pour transférer des modules supplémentaires (ressources) entre les méta-modules adjacents sans modifier le nombre de méta-modules est également introduit. La figure 2.27 montre un exemple d'un système de méta-modules Général1 de deux dimensions appliquée sur le même problème

de reconfiguration selon la figure 2.26.

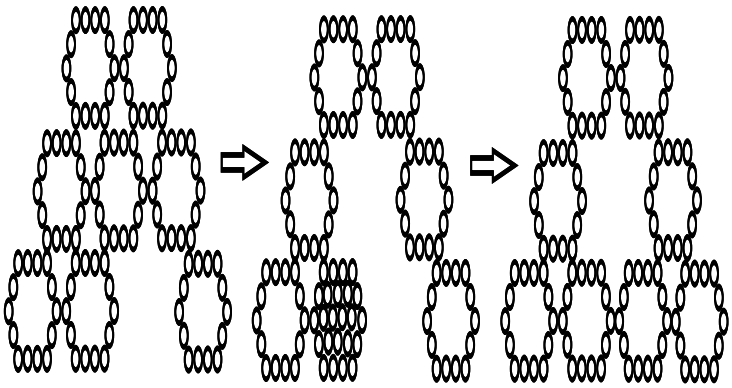

FIGURE 2.27 – Plan de mouvement dans des systèmes de méta-module : La configuration de la figure 2.26, mise à l'échelle et construite à partir de méta-modules. Le plan de mouvement en utilisant les méta-modules est beaucoup plus simple en raison de leur capacité à absorber / recréer d'autres méta-modules [29]

3. Généraln :

L'inconvénient de Général1 est qu'il doit avoir un volume interne suffisant pour contenir tous les modules nécessaires pour la création d'un deuxième méta-module. Comme ce n'est pas souhaitable, ni même possible, pour certains systèmes, généraln propose une généralisation de général1 qui utilise des modules supplémentaires à partir de n méta-modules pour créer un nouveau méta-modules. Ici, Les méta-modules sont construits/détruits progressivement au cours de plusieurs étapes. Cela nécessite n modèles différents de créer/détruire pour gérer tous les degrés de réalisation dans un méta-module partiel, et de multiples applications pour transférer les règles livrer/supprimer de toutes les ressources. L'ajout de n-étapes de calculs nécessite la participation d'un seul méta-module supplémentaire, dont les modules peuvent être balayés pour construire de nouveaux méta-modules. Un problème dans la mise en œuvre d'un tel système est que les méta-modules Bi partiellement construits doivent se connecter à tous les positions treillis de méta-module voisins. Cela garantit que les méta-modules Bi sont fonctionnellement équivalents à l'ensemble des méta-modules voisins, et de ce fait, conforment à l'ensemble de règles. Malheureusement, ceci limite considérablement la conception de méta-module qui fonctionne. Pour remédier à cela, la notion de verrouillage est introduite au niveau d'un méta-module : à partir d'une séquence de création ou de destruction, le système verrouille le méta-module partiel et le méta-module complet adjacent jusqu'à ce que la séquence complète soit achevée. La paire de méta-modules verrouillée ne peut participer à créer ou détruire des modèles avec le méta-module en partenariat, bien que le modèle de transfert est permis avec les autres méta-modules adjacents. Les modèles pour le général-lockn sont :

$creat/destroy_1 : \mathrm{C}\Delta \Leftrightarrow D\infty B_1$

$creat/destroy_i : C\infty B_{i-1} \Leftrightarrow D\infty B_i$

$creat/destroy_n : C\infty B_{n-1} \Leftrightarrow DD$

$transfer : CD \Leftrightarrow DC$

Les états supplémentaires $B_1 ... B_{n-1}$ représentent des méta-modules partiellement construits. Le symbole ∞ représente la serrure par paire entre les méta-modules. Ce système permet dé-

sormais une gamme beaucoup plus large de mise en œuvre, où un méta-module Bi a besoin seulement de maintenir la connectivité avec le module adjacent qui a initié la création ou le processus de destruction.

C) **L'algorithme de planification**

Cette section présente un algorithme distribué asynchrone de reconfiguration. Le planificateur tourne au-dessus de la notion de méta-modules. Étant donné une configuration de départ de méta-modules, le planificateur prend en charge la création et la suppression de méta-modules pour reconfigurer l'ensemble en une forme cible. Le planificateur conserve la propriété de connectivité globale, mais ne traite pas directement avec la contrainte de conservation (min, max). Toutefois, il fournit des garanties prouvables d'exhaustivité : s'il existe un plan globalement connecté pour parvenir à une forme cible, il sera trouvé.

Le planificateur, produit une séquence de réarrangements pour parvenir à une forme cible, T, tout en maintenant la connectivité du système. Initialement, tous les méta-modules sont dans une forme de départ connue, chaque méta-module est conscient de ses coordonnées dans le plan, et il connaît la configuration de la forme cible. Les méta-modules ne sont autorisés à communiquer qu'avec leurs voisins physiques. Le planificateur supprime progressivement les méta-modules qui ne sont pas dans la forme cible et crée des méta-modules à des espaces vides dans la forme cible. Le planificateur assure que la suppression n'affecte pas la connectivité globale en générant des arbres logiques qui relient chaque méta-module supprimé des sections qui seront conservées par induction grâce à son parent dans l'arborescence. Lorsque la suppression est limitée aux feuilles des arbres (c'est à dire, les méta-modules qui n'ont pas de fils), tous les autres méta-modules resteront connectés. Pour distribuer le travail de la génération et de modification des arbres logiques, chaque méta-module enregistre et communique une variable appelé *label*, à ses voisins physiques. Le planificateur utilise trois valeurs d'étiquettes - U (undecided), P (path), et F (final) pour désigner des méta-modules qui ne font pas partie d'un arbre, ceux qui sont dans un arbre, et ceux qui sont dans la forme cible, respectivement. Le module dans T à l'état F est noté TF.

Le plan commence avec un ensemble, E, consistant en un ensemble de méta-modules dans la forme de départ. Exactement un méta-module de semences à l'intersection de E et T, est marqué F, tandis que tous les autres sont marqués U. Un méta-module marqué F recrute chaque voisin dans T à devenir aussi F. Il marque chaque voisin qui n'est pas dans T en tant que candidat pour l'enlèvement appelé P. Ces nœuds recrutent de manière récursive autres méta-modules U. Chaque P est relié à son parent (le méta-module qu'il a recruté à l'état P). Tant que la séquence de connexions parent-enfant reste intact, les méta-modules P resteront connectés à la forme cible. Finalement, les arbres P n'auront plus de place pour étendre, à quel point, les feuilles (méta-modules sans enfants) peuvent être supprimées en toute sécurité sans perte de connectivité. La figure 2.28 montre des clichés à partir d'une séquence d'exécution de ce planificateur.

D) **Allocation des ressources**

Pour exécuter le planificateur sur un système de généraln il est nécessaire d'inclure une composante supplémentaire appelée allocateur de ressources. L'allocateur de ressources change les ressources autour (en déplaçant les états de C et D par de multiples appels à l'opération de transfert) impliquant que des créations et des suppressions indiquées par le planificateur peuvent se produire. Comme il existe une contrainte globale sur le nombre total d'étiquettes C et D, une bonne allocation des ressources doit les distribuer pour envoyer l'état D dans les

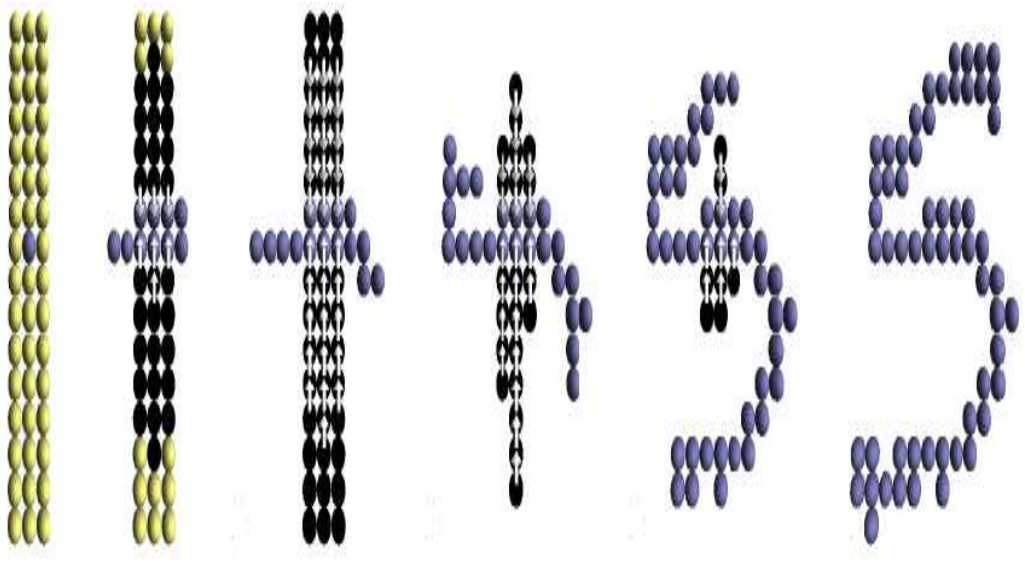

FIGURE 2.28 – Un exemple d'exécution de l'algorithme de reconfiguration en utilisant la création et la destruction des catoms. Les nœuds en jaune ce sont des nœuds qui n'appartiennent pas à la forme cible. Les nœuds en bleu ce sont des nœuds qui sont dans la forme cible nœuds en noire ce sont des nœuds à détruire (supprimer) [29]

régions de suppression anticipée et l'état C pour dans les régions de la création prévue. Une allocation simple et très sous-optimale est un générateur aléatoire qui transporte des ressources en échangeant au hasard des étiquettes adjacentes D et C. Sauf si un allocateur de ressources malveillant est utilisé, l'algorithme est prouvable tant que la forme de départ contient un certain nombre viable des ressources pour la forme cible. Cela fonctionne parce que le planificateur permet la création et la suppression en parallèle, et il permet d'empêcher les états intermédiaires de devenir coincés en raison des contraintes de ressources.

2.5.3.4 Sculpture de forme par trou de mouvement

Cette solution [77] proposée dans le cadre du projet Claytronics est basée sur la formation de forme résultant du mouvement aléatoire de trous réguliers dans une structure en treillis. Les modules sont emballés dans un réseau hexagonal. Un trou est l'entité logique formée par l'absence d'un module et ses six voisins. Un trou est entouré par un groupe de bergers, composé de 12 modules qui bordent le trou (figure 2.29). Des trous sont créés et supprimés au niveau du périmètre de la structure. La création d'un trou mène à l'enceinte de l'espace vide qui se traduit par un renflement à cette position. La suppression d'un trou donnera lieu à la structure spéléologie à ce point. Les procédés de création et de suppression sont représentés sur la figure 2.30. Une technique de lissage est appliquée pour empêcher le développement des zones où la suppression ne peut pas se produire. Avec un trou assez grand, le mouvement dans le voisinage immédiat d'un module sera toujours possible, donc beaucoup de contraintes de mouvement peuvent être évitées. L'algorithme est massivement parallèle et entièrement distribué. La construction de la forme par les modules ne nécessite pas de communication globale.

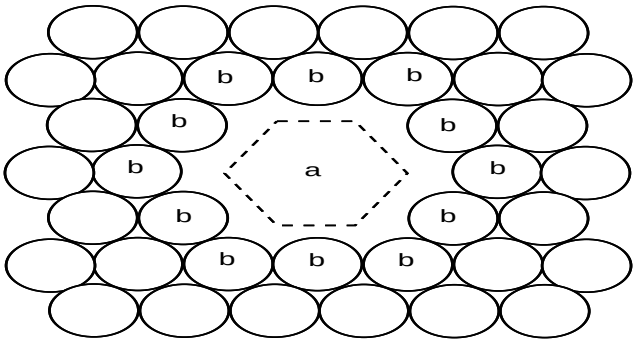

FIGURE 2.29 – (a) le trou, (b) le groupe de berger

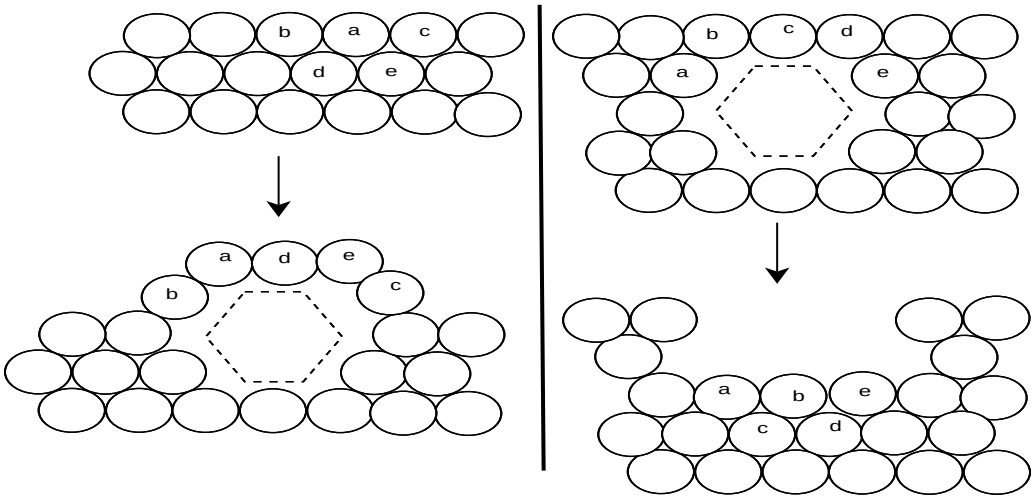

FIGURE 2.30 – Le planificateur de mouvement de trou de [77] crée des bulles à la surface (à gauche) et propage le vide résultant à travers le centre de l'ensemble jusqu'à ce qu'il atteigne une surface de rétrécissement où il apparaît, laissant un cratère (à droite)

2.6 Sommaire

Dans ce chapitre, on a fourni une vue détaillée des systèmes de microrobots modulaires et les différentes techniques proposées dans la littérature pour les reconfigurer. Les microrobots MEMS sont des éléments miniaturisés capables de détecter et d'agir sur l'environnement. Le réseau de microrobots est composé d'un nombre très grand de microrobots déployés dans une zone 2D ou 3D. Ils peuvent s'organiser dynamiquement pour changer leur topologie physique et logique, afin d'accomplir plusieurs missions et tâches. Les microrobots communiquent entre eux uniquement par contact physique et se déplacent en s'appuyant et en se synchronisant avec leurs voisins physiques. Les microrobots modulaires offrent l'avantage d'être simples, modulaires, polyvalents, peu coûteux, robustes et interchangeables.

L'auto-reconfiguration est le processus qui permet le redéploiement des systèmes de microrobots pour changer leur forme, afin de s'adapter aux conditions des tâches et missions qui leur

sont données.

La problématique d'auto-reconfiguration représente un défi dans les applications de microrobotique. Le contrôle dans les systèmes de microrobots implique la coordination et la coopération de grands nombres de modules identiques reliés d'une manière volatile au cours du temps. Le processus de contrôle centralisé ou celui basé sur l'acquisition d'informations globales sur le réseau donnent lieu à des problèmes de robustesse et d'évolutivité. L'approche de contrôle distribuée basée sur l'information locale pour achever la reconfiguration, permet d'obtenir des systèmes robustes, évolutifs et adaptatifs. En contre partie, il est difficile de maintenir un système cohérent en utilisant des règles locales. Dans la littérature, les techniques de reconfiguration pour les robots peuvent être classifiées en trois catégories suivant les types des robots et leurs caractéristiques :

– Auto-reconfiguration pour des robots hexagonaux : Plusieurs algorithmes de reconfiguration ont été proposés pour les microrobots hexagonaux. Deux algorithmes pour la reconfiguration vers une chaîne droite ont été proposés : un algorithme distribué de reconfiguration à partir d'une chaîne droite vers une autre chaîne droite cible, et un algorithme distribué de reconfiguration depuis une forme quelconque avec certaines caractéristiques d'admissibilité vers une chaîne droite. Le deuxième algorithme suppose que chaque nœud connait les voisins de second rayon (voisins des voisins). Ces deux algorithmes n'ont pas recours à une phase de prétraitement et ne nécessitent pas d'échange de messages pour accomplir la reconfiguration. Nous avons aussi décrit deux algorithmes de reconfiguration de microrobots hexagonaux depuis une chaîne vers une forme quelconque ayant certaines caractéristiques d'admissibilité. Le premier algorithme maintient un espacement de deux cellules libres entre les nœuds mobiles. Le deuxième algorithme, plus efficace, maintient un espacement d'une cellule entre les nœuds mobiles pour améliorer les performances. Ces deux algorithmes commencent avec un prétraitement centralisé pour déterminer si la configuration finale est admissible et pour trouver le chemin de substrat.
– Auto-reconfiguration pour des robots cubiques : Plusieurs approches ont été présentées. Une approche centralisée qui utilise les notions de grain et de croissance dirigée. Dans cette solution, un module de grains qui dirige la croissance pour achever la reconfiguration. La deuxième approche est un algorithme centralisé qui utilise un automate cellulaire pour guider l'expansion de la construction, où un module de grains dirigeant la croissance à partir de l'automate cellulaire. Nous avons aussi décrit d'autres approches utilisant des microrobots cristallins qui peuvent se dilater et se contracter par un facteur constant pour achever la reconfiguration. Un algorithme centralisé de la littérature a été décrit ainsi qu'un algorithme distribué à base d'informations locales.
– Auto-reconfiguration pour les Claytronics Atoms : Un algorithme de reconfiguration distribué utilisant seulement la communication locale a été décrit. Cet algorithme permet d'atteindre des formes cibles arbitraires en utilisant une décomposition médiane hiérarchique. Plusieurs planificateurs de mouvement hiérarchique ont été proposés, où l'objectif est la simplification des mouvements des catoms face aux contraintes sur les mouvements. D'autres solutions pour la reconfiguration utilisant les deux primitives Creation et Destruction de catoms ont été proposées. Ces deux primitives ont été utilisées pour simplifier le déplacement des catoms. En fin, une technique basée sur le mouvement aléatoire de trous réguliers dans

une structure en treillis pour achever la reconfiguration.
L'inconvénient majeur des solutions proposées dans la littérature est qu'elles utilisent les positions prédéfinies de la forme cible. Par conséquent, les algorithmes dépendent de la taille du système des microrobots. De plus, ells ne sont pas évolutifs quand la forme cible est constituée d'un nombre très grand de positions (des millions) puisque chaque nœud doit sauvegarder localement toutes ces positions nécessitant un espace mémoire très grand non disponible sur les microrobots. De plus chaque nœud doit connait dès le départ la taille du système (le nombre de microrobots). D'autre part, les solutions proposées exigent qu'au départ au moins un nœud soit déjà dans la forme cible. De plus, ces approches supposent également que chaque nœud connait sa position courante dans le plan. Une contrainte qui suppose l'utilisation d'un dispositif GPS énergivore ou d'une solution algorithmique peu précise.

Par conséquent, il y a un grand besoin pour des solutions d'auto-reconfiguration tenant compte des caractéristiques des microrobots, notamment en ce qui concerne l'utilisation des ressources. Dans nos approches de redéploiement, les algorithmes sont complètement distribués sans recours à l'acquisition d'information globale. Le déroulement des algorithmes proposés s'effectue sans carte (les positions prédéfinîtes de la forme cible). Nous présentons des algorithmes qui utilisent une complexité constante pour répondre aux contraintes de mémoire des microrobots MEMS. Cet avantage rend les algorithmes évolutifs car ils ne dépendent pas de la taille du système de microrobots. Ces algorithmes sont donc applicable quelle que soit la taille du système. Nous proposons aussi de nous affranchir de la contrainte portant sur l'appartenance initiale d'un nœud à la forme cible. De plus les nœuds ignorent leurs positions dans le plan, et ignorent l'orientation de la forme cible. Nous proposons aussi des approches originales pour introduire les notions de réveill et de sommeil des microrobots permettant la conservation d'énergie. En fin, des techniques de parallélisme sont proposées pour augmenter la vitesse de convergence de l'auto-reconfiguration.

Deuxième partie

Contributions

CHAPITRE 3

PROTOCOLES EFFICACES D'AUTO-RECONFIGURATION

Sommaire

3.1 Introduction

Les microrobots MEMS sont des éléments miniaturisés qui peuvent capter et agir dans l'environnement. Ils ont une taille trés petite et des ressources (mémoire et énergie) très limités. Les microrobots peuvent effectuer plusieurs missions et tâches dans un domaine d'applications large telles que la localisation d'odeur, la lutte contre les incendies, dans le service médical, la surveillance, la recherche de sauvetage et de sécurité. Pour réaliser ces tâches, les microrobots doivent se réorganiser en des formes physiques leur permettant de s'adapter aux conditions du travail. Le redéploiement est un processus difficile à contrôler, car il implique la coordination distribuée d'un grand nombre de modules identiques reliés d'une façon fluctuante dans le temps. Actuellement le redéploiement pour les microrobots MEMS mobile nécessite un système de positionnement et une carte (positions prédéfinies) de la forme cible. La solution traditionnelle de positionnement comme l'utilisation GPS consomme beaucoup d'énergie. Aussi, l'utilisation d'une solution de positionnement algorithmique avec les techniques de multilatération pose toujours des problèmes à cause des erreurs dans les coordonnées obtenues. Dans la littérature, l'auto-reconfiguration de microrobots vers une forme cible constituée de P positions, requiert

que chaque microrobot possède une capacité mémoire supérieure à P positions pour les sauvegarder. Malheureusement, quand P se chiffre en milliers ou en millions, cette capacité de stockage est indisponible rendant la solution algorithmique non efficace et non évolutive. Dans les solutions que je propose, les nœuds peuvent ignorer leurs positions courantes et n'enregistrent aucun position de la forme cible. Par conséquent, l'utilisation de mémoire pour chaque nœud est considérablement réduite à une complexité constante.

Une chaîne de microrobots représente le pire des cas en terme de complexité de la diffusion de messages avec $o(n)$ (où n est le nombre e nœuds). Dans ce cas, le redéploiement des noeuds en une organisation carrée permet d'atteindre la meilleure complexité de diffusion de messages avec $o(\sqrt{n})$. Dans ce chapitre on présente deux solutions pour le redéploiement d'une chaîne de noeuds vers un carré pour optimiser la topologie logique du réseau. On étudie deux algorithmes, le premier assure une connexité non instantanée durant l'exécution de l'algorithme avec $n - \sqrt{n}$ mouvements au pire des cas et le deuxième assure une connexité instantanée durant l'exécution de l'algorithme avec n mouvements au pire des cas. Nos algorithmes sont implémentés dans le langage déclaratif Meld et C++ et exécutés dans l'environnement de simulation DPRSim.

3.2 Modèle et définitions

Dans Claytronics, le nœud appelé Catom (figure 3.1) est modélisé comme une sphère qui peut avoir au plus six voisins 2D sans se chevaucher. Chaque nœud est capable de détecter la direction de ses voisins physiques (à l'est (E), à l'ouest (W), au nord-est (NE), au sud-est (SE), au sud-ouest (SW) et au nord-ouest (NW)). La topologie physique de départ est une chaîne droite de n nœuds reliés entre eux. Dans un premier temps, nous considérons que les nœuds initialement ont des voisins dans les directions SE ou NW ou dans les deux directions en même temps. Nous montrons, dans la section 3.3.5, comment généraliser l'algorithme à d'autres orientations de la chaîne droite. Un nœud A est dans la liste des voisins du nœud B si A touche physiquement B (figure 3.2). La communication n'est possible que par contact, ce qui signifie que seuls les voisins peuvent avoir une communication directe (0 saut).

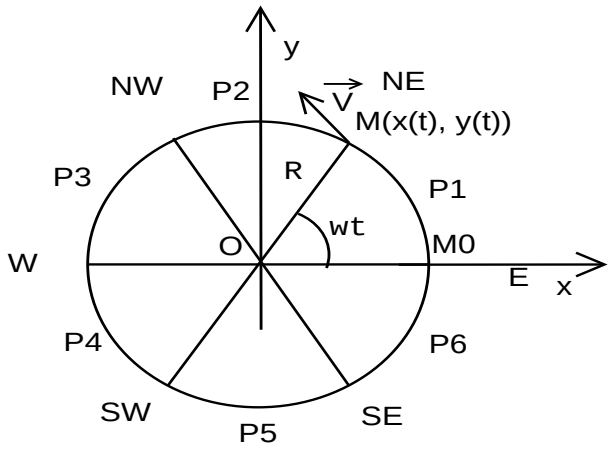

FIGURE 3.1 – Modélisation du Catom, dans chaque étape le nœud parcourt la même distance

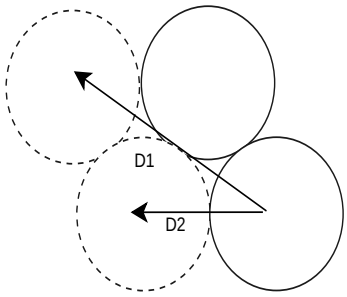

FIGURE 3.2 – La distance parcourue est $2R$, dans une étape de mouvement le nœud parcourt $2R$

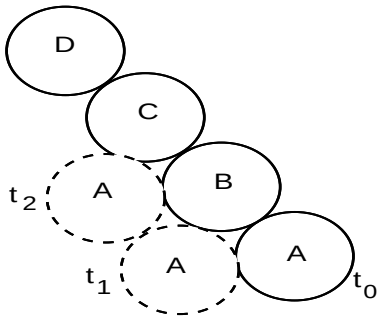

FIGURE 3.3 – La transmission de messages, il y aura un échange de messages si le nœud a besoin de connaître l'état d'un nœud non-voisin

Considérons le graphe non orienté connexe $G = (V, E)$ qui modélise le réseau, $v \in V$, est un nœud appartenant au réseau et $e \in E$ est une arête bidirectionnelle de communication entre les deux voisins physiques. Pour chaque nœud $v \in V$, on note l'ensemble des voisins de v comme $N(v) = \{u, (u, v) \in E\}$.
Connexité : dans un graphe $G = (V, E)$, si $\forall v \in V, \forall u \in V, \exists\, C_{v,u} \subseteq E : C_{v,u} = (e_{v,-}, ..., e_{-,-}, ..., e_{-,u})$, où $e_{x,y}$ est une arête de x à y et $C_{v,u}$ représente un chemin de v à u.
Connexité instantanée : soit T le temps total de l'exécution de l'algorithme distribué DA et $t_0..., t_m$ les rounds d'exécution de DA. DA garantit la connexité instantanée sur les graphes dynamiques $G_{ti} = (V_{ti}, E_{ti})$,si $\forall t_i, i \in \{1, ..., m\}$, le graphe G_{ti} est connexe.
Connexité non-instantanée : On dit que DA assure une connexité non-instantanée si il n'assure pas une connexité instantanée durant l'algorithme de reconfiguration mais le réseau sera connexe à la fin de l'algorithme.
Connexité B-non-instantanée : on dit que DA assure une connexité B-non-instantanée, s'il assure une connexité non-instantanée à ti ou à $t = t_i + ... + t_{m-1}$ et vérifie les conditions suivantes : Soit N_0 est l'ensemble des voisins de v à l'étape t_0, et N_s l'ensemble des voisins de v à l'étape t_s, avec N_s=N_0, et $N_{0/s}$ l'ensemble des voisins de v à l'étape t_0 ou à t_s, et N_i l'ensemble des voisins de v à l'étape $t_i, i \neq 0$.
- Si $r \in N_{0/s}$, et $r \notin N_i$ et $\forall e \in N_i$, $e \in N_{0/s}$ alors $r \in N_{i+B}$ à $t_{(i+B)}$. Ou
- Si $r \in N_{0/s}$, et $r \notin N_i$ et $\exists e \in N_i$, $e \notin N_{0/s}$ alors v assure une connexité instantanée avec tous ses voisins de $t_{(i+B+j)}$, $j = \{1, ..., m - i - B\}$.

Avec une Connexité B-non-instantanée l'information pourra toujour atteindre un destinataire même si elle reste bloquée un certain temps (B étapes) au niveau d'un noeud le temps que les liens se rétablissent.
Arbre couvrant : l'arbre est un graphe connexe sans cycle. Deux noeuds voisins, dans l'arbre, sont définis par une relation parent/enfants (un noeud est pris pour parent et l'autre pour enfant). Les feuilles de l'arbre sont les noeuds n'ayant pas d'enfants.
On appelle *le nombre propre de mouvements* d'un nœud donné le nombre de mouvements effectués par ce dernier. Il est intéressant de prédire le nombre de mouvements que doit effectuer un noeud avant d'atteindre sa position finale car ceci fournit un élément supplémentaire pour contrôler la bonne exécution de l'algorithme.

Pour calculer le nombre de mouvements on commence par formuler les suppositions suivantes :
Soit la figure 3.1 qui représente un nœud microrobot. On appelle un mouvement le déplacement d'un noeud d'une distance, D1 sur la figure 3.2, égale à deux fois le rayon d'un noeud ($D1 = 2R$). La figure 3.2 montre le cas où le nœud parcourt une distance D2 depuis son ancienne position faisant ainsi deux mouvements. Comme 360° est divisible en six angles égaux de 60°, le périmètre d'un angle a est de $P_a = \pi Ra/180$. Ainsi le périmètre total d'un noeud est de $P = 2\pi R$ subdivisé en 6 segments d'arc $P1 = P2 = P3 = P4 = P5 = P6$. Un nœud peut avoir au plus six voisins sans chevauchement.

Dans un plan cartésien, on considère la courbe de l'équation de coordonnées cartésiennes suivant :

$$\begin{cases} x(t) = Rcos(wt) \\ y(t) = Rsin(wt) \\ \text{Avec } wt \in [0..2\pi[\end{cases} \tag{3.1}$$

Il s'agit d'une courbe fermée puisque $x(0) = x(2\pi/w)$ et $y(0) = y(2\pi/w)$ où w est une constante définie par la vitesse de rotation d'un noeud autour de son voisin. Les coordonnées du point de contact M entre le noeud en mouvement et son pivot répondent à l'équation suivante : $x(t)^2 + y(t)^2 = R^2$. Cela signifie que M décrit un cercle de centre O et de rayon R (figure 3.1).

Le cercle est parcouru après une période $T = 2\pi/w$ correspondant à la période de révolution. Le vecteur de vitesse s'écrit :

$$\vec{V} = \begin{pmatrix} -Rsin\ wt \\ Rcos\ wt \end{pmatrix} \tag{3.2}$$

La longueur de l'arc parcourue par M est de $l(t) = \int || \vec{V} ||.dt = Rwt$. Dans un round, le microrobot de rayon R parcourt Ra.
Nous supposons que la communication entre deux voisins physiques s'effectue sans échange de messages, puisque la distance entre les deux voisins physiques est nulle. Si un nœud a besoin de connaitre l'état d'un autre nœud non voisin pour prendre une décision, un échange de messages est effectué et le nœud est amené à attendre. La figure 3.2 montre un exemple d'échange de messages :

– à t_0 : le noeud A a besoin de connaitre l'état du noeud B pour se déplacer vers une nouvelle position. Ce mouvement se fait sans échange de messages.

– à t_2 : si A est dans la nouvelle position et a besoin de connaitre l'état de D pour se déplacer, D envoie un message à C l'informant de son état puis le nœud C retransmet le message à A. L'échange de message conduit A à attendre l'information au moins deux tours avant de prendre une décision.

– si à t_0 ou à t_1 un message a été envoyé à partir de D à C, A peut à l'instant t_2 obtenir l'état de D avec une simple consultation de l'état de C, sans échange de messages.

La conception de l'algorithme d'auto-reconfiguration doit réduire au minimum le nombre de mouvements dont dépend la consommation énergétique du système et le temps d'exécution du programme. Il est également important de réduire l'espace mémoire utilisé et donc le nombre d'états par nœud.

3.3 Protocoles proposés

3.3.1 Algorithme avec connexité non-instantanée(ACNI)

Dans l'algorithme ACNI [52] (présenté ci-après), le nœud peut se déplacer autour de son voisin physique (voir la figure 3.4) ou s'éloigner d'un voisin d'une distance égale à deux fois le rayon d'un noeud (voir la figure 3.5). Ces deux types de mouvements sont effectués en présence d'au moins un voisin pour déterminer le sens du mouvement ('autour de' ou 's'éloigner de'). Un nœud sans voisin ne peut pas bouger, car il ignore sa position et celle des autres nœuds.

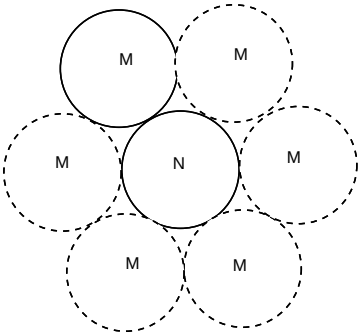

FIGURE 3.4 – Dans ce type de mouvement le nœud M peut se déplacer autour de son voisin N

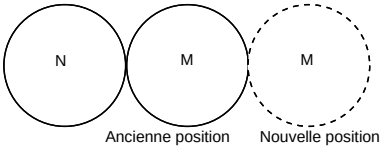

FIGURE 3.5 – Le nœud M peut se déplacer à la nouvelle position de distance égale à deux fois le rayon s'il a un voisin N

Variables et prédicats :

- v, $u1$, $u2$: des variables représentent un nœud qui appartient au réseau.
- $\{u\}$: ensemble de nœuds.
- $good, bad, spe$: états, un nœud peut prendre un ou deux états en même temps, sachant qu'il ne peut pas prendre les deux états *spe* et *bad* ou *good* et *bad* en même temps.
- $N_x(v)$: le voisin dans la direction x du nœud v, $x \in \{(N), (E), (W), (NE), (SE), or(NW)\}$.
- $thisRound$: entier représente l'étape courante.
- $connected_v$: $true$ si le nœud v est connecté au reseau, $false$ sinon (Booléen).
- $State_v(k)$: l'état du nœud v, prenant un ou deux de ces états $k \in \{good, bad, spe\}$.
- $N_x lastRound_v()$: le nœud v a un voisin dans la direction x à l'étape précédente.
- $State_v(n, good)$: le nœud v a n voisins qui ont l'état *good* ($State(good)$).
- $moveTo_v(P_{N_{nw}})$: v se déplacer vers la position $P_{N_{nw}}$ où était un voisin N_{nw} dans la direction nw à l'étape précédente.
- $moveAroundgood_v(u, P_x)$: le nœud v se déplace autour du voisin u de telle sorte que u devient un voisin dans la direction x pour v.

Prédicats vérifiés seulement dans la première étape

1 : Initiator$(v) \equiv N_{nw}(v) = \oslash \wedge connected_v$.
2 : State$_v(bad) \equiv connected_v \wedge \neg Initiator(v)$.
3 : State$_v(good) \equiv Initiator(v)$.
4 : State$_v(spe) \equiv Initiator(v)$.

Prédicats vérifiés à chaque étape

5 :(P1) : State$_v(good) \equiv (N_e(v) = u1 \wedge State_{u1}(good) \wedge N_{ne}(u1) = \oslash) \vee State_v(3, good) \vee (State_v(2, good) \wedge (N_{ne}(v) = u1 \wedge State_{u1}(spe)) \vee (N_w(v) = u1 \wedge State_{u1}(good))) \vee State_v(spe)$.
6 : State$_v(n, good) \equiv (N_x(v) = \{u\}, |u| = n \wedge State_{\{u\}}(good))$.
7 : (P1) : State$_v(spe) \equiv (N_{nw}(v) = u1)) \wedge (N_{ne}(v) = u2, State_{u2}(spe))$.
8 : thisRound$\equiv GetCurrentRound()$.
9 : has$N_{nw}v(thisRound) \equiv N_{nw}(v) = u1 \wedge State_{u1}(bad)$.
10 : N_{nw}lastRound$_v(LastRound) \equiv hasN_{nw}v(thisRound - 1)$.
11 : (P2) : moveTo$_v(P_{N_{nw}}) \equiv State_v(bad) \wedge N_{nw}lastRound_v(LastRound) \wedge \neg hasN_{nw}v(thisRound)$.
12 : cannotMove$_v() \equiv (N_x(v) = \{u\}, |u| = 1 \wedge State_u(good))$.
13 : (P2) : moveAround$good_v(u1, P_{ne}) \equiv \neg(cannotMove_v()) \wedge (State_v(bad)) \wedge (N_{nw}(v) = u1 \wedge State_{u1}(good))$.
14 :(P2) : moveAround$good_v(u1, P_e) \equiv \neg(cannotMove_v()) \wedge (State_v(bad)) \wedge (N_{ne}(v) = u1 \wedge State_{u1}(good))$.

Algorithme ACNI.

3.3.1.1 Description de l'algorithme

L'algorithme ACNI fonctionne en cycles asynchrones. À chaque tour, le démon (planificateur) d'un noeud choisit les prédicats vérifiés à exécuter. Nous introduisons dans l'algorithme un mécanisme de priorité entre les prédicats : le démon choisit le prédicat vérifié le plus prioritaire et ignore dans le présent cycle les prédicats de moindre priorité. On note que les prédicats marqués du label $P1$ sont considérés comme plus prioritaires que les autres marqués avec $P2$.

L'algorithme distribué ACNI utilise un processus incrémental construisant la forme finale couche par couche. Lors de chaque incrément, l'algorithme distribué positionne les nœuds dans la forme cible. Au début, l'initiateur est considéré comme appartenant à la forme cible et aide ses voisins à se positionner dans la forme cible. Les nœuds qui sont dans la forme cible agissent à leur tour comme des points de repère permettant à leurs voisins de remplir un nouvel incrément. Les nœuds qui atteignent leur position finale dans la forme cible, changent leurs états internes avec les prédicats (1) et (5) et deviennent fixes (ils ne bougent plus). Au début, tous les nœuds sont initialisés avec l'état *bad* (2) à l'exception de l'initiateur (1). Le nœud peut vérifier si son voisin est dans un état *good* à l'aide du prédicat (5). Avec le prédicat (12), un nœud dans l'état *bad* qui avait un voisin au NW à l'étape précédente, doit se déplacer vers le NW pour occuper l'ancienne position de son voisin (il parcourt exactement $2R$).

Les nœuds de la couche actuelle (en cours de construction) peuvent se déplacer soit vers l'ouest ou vers le NW à l'aide des trois derniers prédicats. Le nœud peut changer son état vers *good* s'il ne peut pas se déplacer à gauche ou NW en utilisant les prédicats de mouvement prédéfinis. Le prédicat (13) permet à un nœud de se déplacer vers la gauche autour d'un voisin ayant l'état *good*. Après l'exécution du prédicat (13) le voisin passe d'un voisin NW à un voisin NE. Le noeud en mouvement répète l'exécution du prédicat (13) jusqu'à ce qu'il devienne voisin d'un nœud dont l'état est *spe/good* (les noeuds constituant la diagonale du carré).

Le noeud peut se déplacer autour du noeud voisin dont l'état est *spe/good* se trouvant au NE seulement si le nœud diagonale ne possède pas de voisin dans sa direction W. Tous les nœuds de la diagonale ont l'état *spe/good* avec le prédicat (7). Et avec (14), le nœud se déplace à NW jusqu'à ce qu'il prenne une position correcte.

Le changement d'état est plus prioritaire que les actions de mouvement, ceci est pour éviter des mauvais mouvements, parce que quand le nœud est dans une bonne position (il peut changer son état à *good*), il doit ignorer le prédicat de mouvement. Pour ce faire, nous utilisons un mécanisme de priorité dans notre algorithme. Pour éviter l'échange de messages, le nœud peut changer son état à *good* s'il a trois voisins ayant l'état *good* ou un voisin á l'état *spe* et il á des voisins dans les deux directions NE et NW (6).

3.3.1.2 L'algorithme garantit une connexité 1-non-instantanée

La connexité 1-non-instantanée signifie qu'un message quelconque suivant un trajet $C_{v,u}$ ne peut pas être bloqué, au niveau d'un nœud, plus de deux rounds consécutifs avant d'être retransmis à un nouveau nœud. Pour le prouver il suffit de démontrer que l'algorithme n'assure pas une connexité instantanée et qu'il n'assure pas une connexité B-non-instantanée, avec $B > 1$.

ACNI n'assure pas une connexité instantanée puisque, par exemple, au second round de l'exécution de l'algorithme, le nœud voisin de l'initiateur se déplace autour de l'initiateur avec

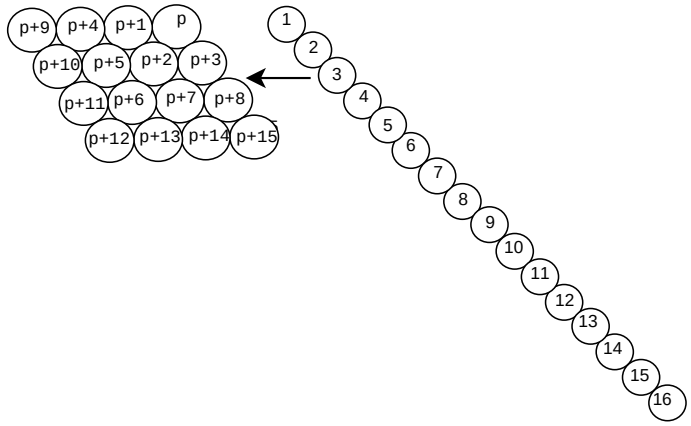

FIGURE 3.6 – Les positions des nœuds dans la forme finale

l'instruction $moveAroundgood_v(u1, P_e)$, il aura un seul voisin dans la direction E laissant un espace où il y avait deux voisins. Si nous supposons que le message est arrivé à son ancien voisin, le message doit attendre un tour ou plus pour qu'il puisse être transmis à l'initiateur, l'équivalent de dire $\exists v \in V, \exists u \in V \exists t_i,$ où $\nexists C_{v,u}$.

On montre par induction que ACNI ne garantit pas une connexité 2-non-instantanée. On suppose que ACNI garantit une connexité 2-non-instantanée, c'est-à-dire $\exists v \in V, \exists u \in V \exists t_i, \exists t_{i+1}$, où $\nexists C_{v,u}$, et le message doit attendre deux tours consécutifs. Supposons que le message est en attente à la fin de t_i, à ce moment le prédicat $moveTo_v(P_{N_{nw}})$ est disponibles pour le nœud ayant le message, et puisque le démon est équitable ce nœud exécute ce prédicat puisque le prédicat $State_v(good)$ n'est pas vérifiable, il se déplace à la position de son dernier voisin et il va trouver un nouveau voisin car ce dernier ne peut pas se déplacer ($cannotMove_v() \equiv (N_x(v))$, donc à t_{i+1} le message peut être transmis à un autre nœud. Depuis ACNI n'assure pas une connexité 2-non-instantanée, il n'assure pas une connexité B-non-instantanée, avec $B > 2$.
Pour compléter la preuve que ACNI garantit une connexité 1-non-instantanée, il reste à montrer pour le nœud où le message a été bloqué au moment t_i que ce nœud assure une connexité instantanée avec ses voisins de $t_{i+1+j}, j = \{1, ..., n-i-1\}$. Ceci peut être prouvé par récurrence puisque le nœud se déplace avec le prédicat $moveAroundgood_v(u1, P_e)$ autour des nœuds ayant l'état *good* et ayant des voisins qui ont l'état *good*. Ces nœuds (ayant l'état *good*) ne peuvent pas se déplacer ($State_v(good)$) le message peut être transmis à chaque étape à partir de ou vers des nœuds voisins de $t_{i+1+j}, j = \{1, ..., n-i-1\}$.

3.3.1.3 Prédire le nombre de mouvements pour ACNI

Pour former la matrice du carré final de dimension $N * N$, l'algorithme ACNI procède d'une façon incrémental à commencer par un seul nœud (l'initiateur) que nous supposons dans un carré 1x1. A chaque étape de l'algorithme, de nouveaux noeuds s'ajoutent à la couche en construction afin de former un carré plus grand. Chaque couche finit par contenir un nombre de nœuds égal au nombre de noeuds de la dernière couche plus deux. Par exemple, pour atteindre le carré 2x2, on doit ajouter une nouvelle couche avec trois nœuds. La figure 3.6 montre le repositionnement des noeuds dans la forme cible une fois atteinte. Le nœud i prend la position $p+x$, où les noeuds

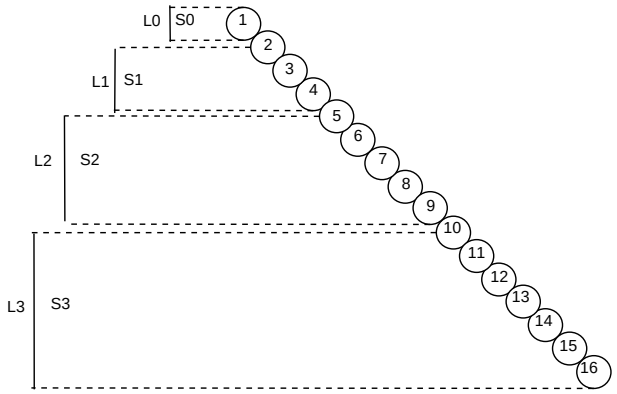

FIGURE 3.7 – Le partitionnement des nœuds à des niveaux

sont numérotés du nord au sud et les positions cibles sont numérotés couche par couche depuis la position NW jusqu'à la position SE. Un noeud commence à se déplacer avant les noeuds se trouvant plus au sud. Si le nœud A est initialement au nord du noeud B, et que la position finale de A est $p + c$ et celle de B est $p + d$ alors $d > c$.
Notons par U_j le nombre de noeuds dans la couche numéro j. Le nombre de noeuds d'une couche peut s'écrire sous la forme d'une série arithmétique sous la forme :

$$U_j = U_{j-1} + 2. \tag{3.3}$$

Nous définissons un partitionnement des noeuds de la chaîne initiale en niveaux. Un même niveau est associé à un ou plusieurs nœuds. Le calcul du niveau auquel appartient un nœud s'effectue suivant la procédure suivante : le niveau 0 est associé au premier nœud (l'initiateur), puis chaque niveau suivant est associé à un nombre de nœuds suivants égal au nombre de nœuds du niveau précédent plus deux (la figure 3.7 représente un exemple). La procédure est poursuivie jusqu'à ce que tous les nœuds soient traités.

Pour déterminer le nombre de mouvements, nous utilisons une autre séquence similaire à (3.3) mais avec une interprétation différente :

$$\begin{aligned} S_1 &= 2. \\ S_j &= S_{j-1} + 2. \end{aligned} \tag{3.4}$$

Avec : S_j est un nombre associé aux nœuds qui ont le niveau j.
Le nombre de mouvements pour chaque nœud i ayant le niveau j peut être donné par la composition de deux séquences $U_{i,j}$ et S_j.

$$\begin{aligned} U_1 &= 0. \\ U_{i,j} &= U_{j-1} + S_j. \end{aligned} \tag{3.5}$$

Avec : $U_{i,j}$ et U_j sont respectivement le nombre de mouvements du nœud i ayant le niveau j, et le nombre de mouvements des nœuds qui ont le niveau j.

Theorem 3.3.1 *$n - \sqrt{n}$ représente le nombre maximal de mouvements qu'un noeud est amené à faire dans l'algorithme quand le nombre $\sqrt{n}$ est entier. Cas particulier : Si $\sqrt{n}$ n'est pas un nombre entier, le plus grand nombre de mouvements est $\lceil\sqrt{n}\rceil\lceil\sqrt{n}\rceil - \lceil\sqrt{n}\rceil$, et le nombre de mouvements de chaque noeud est donné avec les mêmes séquences.*

3.3.2 Algorithme avec une connexité instantanée(ACI)

Dans cet algorithme [49, 52] un nœud ne peut se déplacer que autour de ses voisins physiques. Pour assurer une connexité instantanée seuls les nœuds dont le mouvement ne causent pas la dis-connectivité du réseau sont autorisés à se déplacer. Pour ce faire, l'algorithme ACI calcule au préalable un structure d'arbre géré dynamiquement où seuls les nœuds feuilles peuvent se déplacer.

Variables et prédicats :

– $Parent(v, u)$: le nœud v est parent du nœud u.
– $isLeaf(v)$: le nœud v est une feuille dans l'arbre.

Prédicats vérifiés seulement dans la première étape

1 : Initiator$(v) \equiv N_{nw}(v) = \oslash \wedge connected_v$.
2 : State$_v(bad) \equiv connected_v \wedge \neg Initiator(v)$.
3 : State$_v(good) \equiv Initiator(v)$.
4 : State$_v(spe) \equiv Initiator(v)$.

Prédicats vérifiés à chaque étape

5 : Parent$(v, v) \equiv Initiator(v)$.
6 : Parent$(v, u) \equiv (Parent(w, v), u \neq w) \wedge neighbor(v, u) \wedge State_u(bad) \wedge (\nexists z \in N(v), Parent(v, z))$.
7 : isLeaf$(v) \equiv (\forall u \in N(v), \neg Parent(v, u) \wedge \neg Parent(v, v))$.
8 : (P1) : State$_v(good) \equiv (N_e(v) = u \wedge State_u(good) \wedge N_{ne}(u) = \oslash) \vee State_v(3, good) \vee (State_v(2, good) \wedge (\mathrm{N}_{ne}(v) = u \wedge State_u(spe)) \vee (N_w(v) = u \wedge State_u(good))) \vee State_v(spe)$.
9 : State$_v(n, good) \equiv (N_x(v) = \{u\}, |u| = n \wedge State_{\{u\}}(good))$.
10 : (P1) : State$_v(spe) \equiv (N_{nw}(v) = u1)) \wedge (N_{ne}(v) = u2, State_{u2}(spe))$.
11 : (P2) : moveAround$bad_v(u1, P_e) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u1 \wedge State_{u1}(bad))$.
12 : (P2) : moveAround$bad_v(u1, P_{se}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) = u1 \wedge State_{u1}(bad))$.
13 : (P2) : moveAround$good_v(u1, P_{ne}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u1 \wedge State_{u1}(good))$.
14 : (P2) : moveAround$good_v(u1, P_e) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) = u1 \wedge State_{u1}(good))$.

Algorithme ACI.

3.3.2.1 Description de l'algorithme

Contrairement à ACNI, pour assurer une connexité instantanée, l'algorithme ACI autorise uniquement les nœuds feuilles à se déplacer. Pour ce faire, nous introduisons l'utilisation de l'arbre de gérer dynamiquement les nœuds feuilles. L'algorithme fonctionne en rounds asynchrones. Comme pour l'algorithme ACNI, à chaque round, les prédicats satisfaits sont exécutés selon une hiérarchie de priorités définies par les labels P1 (les prédicats prioritaires) et P2 (les prédicats moins priorioritaires). Quand un prédicat P1 est actif, les prédicats P2 satisfaits sont ignorés lors du round courant. L'algorithme ACI construit la forme cible par un processus incrémental. Une fois un incrément terminé, les nœuds de la couche construite appartiennent déjà à la forme cible (sont dans leur position finale). L'initiateur, initialise la construction de l'arbre en tant que racine et devient parent de lui-même (5). Un nœud, initialement sans parent ni enfants, choisit un parent parmi ses nœuds voisins ayant déjà un parent (6). Un nœud devient une feuille si il n'est le parent d'aucun de ses voisins (7). Au début tous les nœuds sont initialisés à l'état *bad* utilisant le prédicat (2). L'initiateur suppose alors appartenir à la forme cible et se met à l'état *good* (3). Il fonctionne alors comme un point repère aux voisins initiaux ou futurs pour se positionner correctement. Les nœuds suivants, agissent à leur tour comme des points repère pour remplir d'autres couches. Les nœuds qui atteignent leur position finale dans la forme cible, après vérification de l'état des voisins, deviennent fixes et changent leurs états avec les prédicats (3) et (8).

Le nœud le plus au sud de la chaîne initiale est celui qui entame les mouvements car c'est la feuille de la première arborescence construite. Il se dirige vers le nord par rotation autour des voisins jusqu'à devenir le voisin W de la racine (prédicats (11) and (12)). Les nœuds de la couche actuelle (couche en cours de construction) peuvent faire des mouvements soit vers le W ou vers le NW avec les trois derniers prédicats. Le nœud change d'état à *good* dans le prédicat (3) s'il ne peut pas se déplacer ni à gauche ni vers le NW. Avec le prédicat (13), le nœud se déplace à gauche, laissant son voisin pivot vers son coté NE. Il répète le même mouvement jusqu'à ce qu'il arrive au nœud en diagonale ayant l'état *spe*. Le noeud peut se déplacer autour du noeud *spe* seulement si le nœud en diagonale n'a pas un nœud voisin dans la direction E. Les nœuds en diagonale prennent l'état *spe* avec les prédicats (4) et (10). Avec le prédicat (14), le nœud se déplace jusqu'à ce qu'il atteigne une position finale.

Les prédicats de l'algorithme distribué sont régis par une règle de priorité consistant à privilégier les actions de changement d'état (étiquetées par P1) aux actions de mouvement(étiquetées P2), ceci afin d'éviter qu'un mouvement erroné ne se produise. Pour éviter l'échange de messages, un nœud passe à état *good* dès qu'il dispose de trois voisins dont l'état est *good* (9) ou qu'un de ses voisins est à l'état *spe* alors que les directions NE et NW sont occupées (10).

3.3.2.2 L'algorithme garantit une connexité instantanée

Cet algorithme garantit une connexité instantanée, étant donné que le graphe initial à l'état $t0$ est connexe et que l'algorithme utilise un arbre couvrant où seules les nœuds feuilles peuvent se déplacer. Le fait que seules les feuilles de l'arbre couvrant se déplacent fait que des discontinuité ne peuvent pas apparaitre dans le réseau. Nous divisons les mouvements de feuilles en deux familles : (1) un nœud tourne autour d'un autre sans avoir de nouveau voisin pour le remplacer

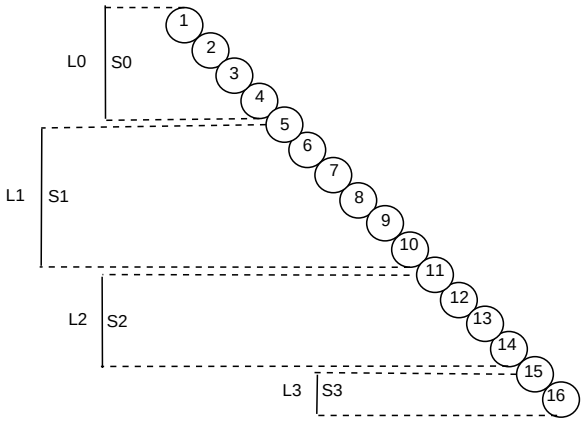

FIGURE 3.8 – Le partitionnement à des niveaux

dans sa position ancienne. Dans ce cas, il n'y a pas d'instant, ti, pendant lequel le message ne peut pas être envoyé car le réseau reste connexe. (2) le nœud en mouvement a un nouveau voisin après son déplacement. Dans ce cas le nœud obtient un autre chemin pour transmettre ses messages sans blocage pour tout ti, $i \in \{1, ..., m\}$.

Prédire le nombre de mouvements pour ACI

Comme dans ACNI, pour atteindre la forme cible on utilise un processus incrémental. Examinons la figure 3.6 qui représente l'ordre de remplissage de la forme cible. Contrairement à l'algorithme ACNI, un nœud i reste immobile (sa position initiale est identique à sa position finale) ou se déplace après les nœuds plus au sud. Si le nœud A est plus au nord que le nœud B et A prend la position $p + c$ et B prend la position $p + k$, alors $c > k$.

Comme dans l'équation (3.3) le nombre de nœuds ajoutés dans chaque couche peut être exprimé avec une suite arithmétique.

Dans la chaîne on partitionne les nœuds à des niveaux. Un niveau est associé à un ou plusieurs nœuds : les $\sqrt{n}$ premiers nœuds prennent le niveau racine ($L0$), les $(2\sqrt{n} - 2)$ nœuds suivants prennent le niveau ($L1$), puis les $(2\sqrt{n} - 4)$ nœuds suivants prennent le niveau $L2$. Ce processus est réitéré de façon que chaque niveau ait un nombre de noeuds égal au précédent moins 2 (sauf le dernier niveau) et ceci jusqu'à ce que tous les noeuds aient un niveau associé. Ce processus est illustré dans la figure 3.8. Le nombre de mouvements pour chaque nœud i ayant le niveau j peut être donné par la composition de deux séquences $U_{i,j}$ et S_j.

$$S_j = \begin{cases} 0, si\, j = 0 \\ 2\sqrt{n} - 5, si\, j = 1 \\ S_{j-1} - 2, autrement \end{cases} \tag{3.6}$$

Où S_j est un nombre associé aux nœuds ayant le niveau j.

$$U_{i,j} = \begin{cases} 0,\ si\, j = 0. \\ 2,\ si\, i = \sqrt{n} + 1, j = 1. \\ U_{i-1} - S_j,\ si\, l(i-1) \neq j. \\ U_{i-1} + 2, autrement. \end{cases} \tag{3.7}$$

Où $U_{i,j}$ et U_j sont, respectivement, le nombre de mouvements du nœud i ayant le niveau j et le nombre de mouvements des nœuds du niveau j.

Theorem 3.3.1 *n est le plus le nombre maximal de mouvements réalisé par un noeud dans l'algorithme ACI.*

Cas particulier : Pour calculer le nombre de mouvements d'un nœud quand n n'est pas le carré d'un nombre entier, nous considérons un système de partitionnement similaire au précédent. Le plus grand nombre de mouvements pour atteindre la forme finale reste n, mais la séquence $U_{i,j}$ s'écrit autrement.
Soit $q = \lfloor \sqrt{n} \rfloor$, et $diff = n - q^2$.

$$U_{i,j} = \begin{cases} 0, \, si \, i \leq q. \\ diff + 2q - 1, \, si \, i = q + 1. \\ U_{i-1} - 2, \, siq + diff \geq i > q \\ diff + 2, \, si \, i = q + diff + 1. \\ U_{i-1} - S_j, \, si \, i > q + diff, L(i+1) \neq j. \\ U_{i-1} + 2, autrement. \end{cases} \tag{3.8}$$

3.3.2.3 Sauvegarde d'énergie dans ACI

Les mesures montrent que l'écoute ralentie (l'état oisif du nœud) des dispositifs de détection conduit à une consommation importante d'énergie. Une approche efficace pour économiser l'énergie est de mettre le nœud en état sommeil pendant le temps d'inactivité et réveiller juste avant la transmission du message/réception [40]. Cela nécessite une synchronisation précise entre l'émetteur et le récepteur, de sorte qu'ils puissent se réveiller simultanément pour communiquer l'un avec l'autre.
Dans cette section on montre comment gérer dynamiquement (par un mode réveil/sommeil) les nœuds du réseau pour réduire la consommation d'énergie. Le mode sommeil permet d'économiser l'énergie du noeud pendant les moments où sa contribution n'est pas nécessaire au fonctionnement de l'algorithme [50, 51]. Après le changement de l'état d'un noeud vers *good*, il doit rester encore en mode éveillé car joue alors le rôle de référence permettant aux voisins ou aux futurs voisins, d'après son état, de prendre position dans la forme cible. L'objectif des fonctions suivantes est de présenter une méthode de calcul optimale et déterministe des intervalles de temps de réveil et de sommeil de chaque noeud. Ces fonctions de calcul prennent la forme de séquences mathématiques se traduisant algorithmiquement par une suite de messages entre les nœuds voisins. En recevant les informations de son voisin, le nœud peut déterminer l'intervalle de temps pour entrer en mode sommeil ou de réveil.

Nous reprenons la répartition précédente des nœuds en niveaux avec une particularité supplémentaire : certains nœuds prennent un niveau spécial noté ls. Le noeud numéro $x_1 = 4\sqrt{n} - 4$ prend le niveau ls. Puis le nœud numéro $x_2 = x + y$ où $y = 2\sqrt{n} - 5$ est désigné comme noeud ls. La procédure est réitérée en désignant comme noeud ls le noeud numéro $x_3 = x_2 + y - 2$. Ainsi les noeuds ls sont espacés de moins en moins, chaque fois en retirant 2 à la distance séparant un noeud ls du suivant.

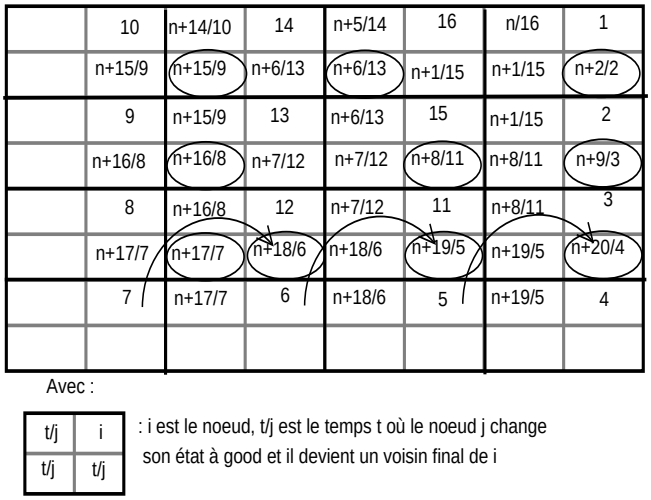

FIGURE 3.9 – Représente le moment où le voisin final change son état à *good*. Les valeurs dans les cercles représentent le temps où le nœud i peut entrer en état de veille, dans la dernière ligne les valeurs de certains nœuds sont indiquées par des flèches

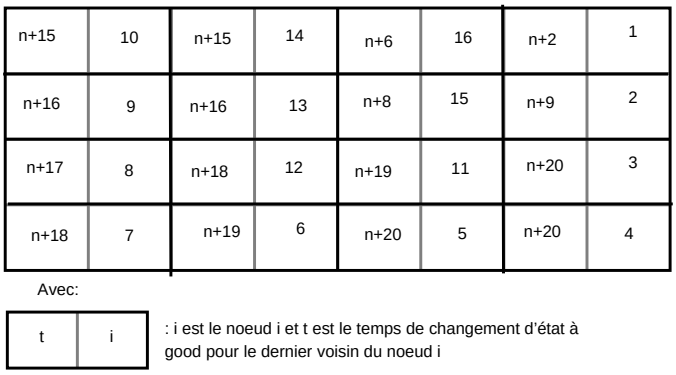

FIGURE 3.10 – Représente le moment du dernier changement d'état d'un voisin, le nœud i n'aura pas un nouveau voisin pour l'aider

Une fois que le nœud a désigné un enfant (dans l'arborescence) parmi ses voisins, il entre en mode sommeil et programme son réveil pour le moment prévu où de nouveaux voisins le rejoignent (le temps que les nœuds se dirigeant vers le nord l'atteignent). Le nœuds racine commence la construction de l'arbre au round $t0$, il devient un parent et entre en état de sommeil pour économiser l'énergie.

$$T_i = \begin{cases} 7, \, si \, i = 1. \\ T_{i-1} + 4, \, autrement. \end{cases} \tag{3.9}$$

Avec : T_i est un nombre associé au nœud i, $i \leq \sqrt{n} - 2$.

$$I_j = \begin{cases} 2\sqrt{n} - 1, \, si \, j = 3. \\ I_{j-1} - 2, \, autrement. \end{cases} \tag{3.10}$$

Avec : I_j est un nombre associé au nœud i de niveau $j > 1$.

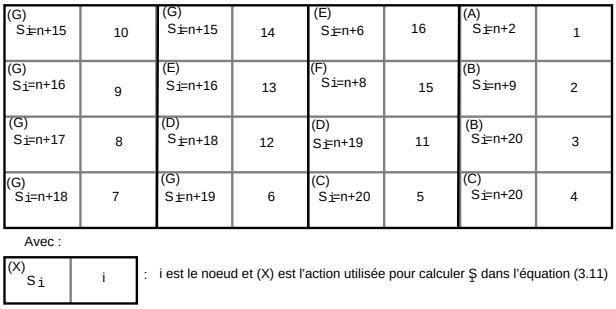

FIGURE 3.11 – Représente la manière dont le dernier temps correspond au temps de sommeil du nœud est calculé

$$S_i = \begin{cases} n+2,\ si\ i = 1.(A) \\ S_{i-1} + T_{i-1}, si\ i \leq \sqrt{n} - 1.(B) \\ S_{i-1}, si\ i = \sqrt{n} \vee i = \sqrt{n} + 1.(C) \\ S_{i-1} + 2\sqrt{n} - 4, si\ l(i) = 2 \wedge l(i-1) = 1.(D) \\ S_{i-1} - 2,\ si\ ls(i-1).(E) \\ S_{i-1} - I_i,\ if\ l(i-1) \neq l(i).(F) \\ S_{i-1} - 1, autrement.(G) \end{cases} \tag{3.11}$$

Avec : $S_i + O(n)$ désigne le temps d'entrer en sommeil pour le nœud i. Etant donné que le premier nœud feuille (à l'extrémité sud de la chaîne initiale) se déplace de n mouvements pour devenir le voisin W de la racine, il est donc voisin SW de la racine après $n-1$ rounds. Par conséquent, la racine règle son horloge locale pour se réveiller à l'instant $n-1+O(n)$ afin de collaborer avec ses nouveaux voisins, $O(n)$ étant le temps de la construction de l'arbre couvrant. De même, les autres nœuds sont initialement en attente pour la construction de l'arbre puis entrent en état de sommeil temporaire après avoir eu un enfant. Le nœud situé après z nœuds de la racine se réveille au round $n-z-1+O(n)$. La séquence S_i exprime en fonction de n le moment où un nœud peut de nouveau revenir à un état de sommeil définitif après avoir aidé ses voisins à prendre des positions correctes.

Exemple : les figures 3.9, 3.10 et 3.11 présentent un exemple pour une configuration de départ de 16 nœuds. Les figures montrant comment les valeurs S_i sont calculées. La figure 3.9 présente les moments où les nœuds deviennent des voisins finaux. La figure 3.10 présente le moment du dernier changement d'état d'un nœud voisin à *good*, qui correspond au moment où le nœud peut entrer en état de sommeil car il n'aura pas à collaborer avec de nouveaux voisins. La 3.11 présente comment les valeurs ont été déduites à partir des fonctions de calcul.

3.3.3 Le nombre d'états minimum

Dans cette section, nous donnons la preuve que trois états est un minimum pour achever les deux algorithmes proposés ACNI et ACI.
Avec un seul état, il est impossible pour les nœuds de s'auto-configurer, car ils ne peuvent pas distinguer s'ils ont atteint une position finale ou non. Autrement dit il n'existe aucun mécanisme

pour différencier les nœuds qui doivent se déplacer et ceux qui doivent rester stables ce qui empêche de reconnaître un bon mouvement d'un mauvais. Supposons maintenant que nous cherchons un algorithme qui utilise seulement deux états. Avec deux états le nœud sait s'il est dans une position finale ou pas, et dans ce deuxième cas se déplace vers la direction W ou NW autour d'autres nœuds dont le positionnement est bon (figure 3.12 (a)). Cependant, quand un noeud atteint la position diagonale de la couche en cours de construction, il a seulement deux voisins à état *good*. Par conséquent, il continue ses mouvements vers l'ouest puis vers le sud-ouest (figure 3.12 (b)) formant ainsi une nouvelle chaîne horizontale. Pour désigner cette situation particulière (noeuds de la diagonale) où le noeud doit devenir stable, il est nécessaire d'introduire un nouvel état spécial pour les nœuds formant la diagonale. un troisième état est essentiel pour savoir si le nœud peut se déplacer autour d'un autre ayant état *spe*. Ainsi, ce déplacement est possible si et seulement si le nœud de diagonale n'a pas de voisin à droite (3.12 (c)).

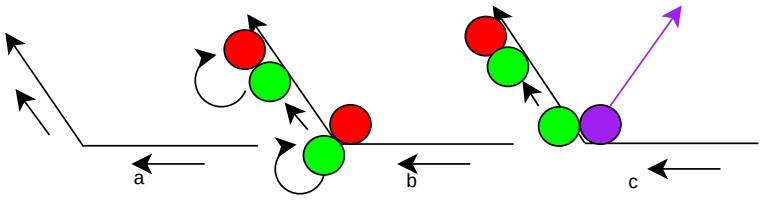

FIGURE 3.12 – Les états des nœuds, trois états sont nécessaires pour chaque nœud

3.3.4 Complexité des messages envoyés

L'information principale à propager au sein du réseau de noeuds est celle relative aux changements d'état. Si un nœud change d'état avant qu'il ne soit sûr de l'état approprié des autres nœuds de la couche courante ou de la couche précédente, le processus risque de ne pas aboutir à la configuration voulue. Le prédicat $State_v(good)$ permet sans échange de messages (sans latence), de changer l'état du nœud v si tous les nœuds le précédant dans la construction de la forme cible sont à l'état *good* ou *spe*. Le premier nœud qui commence la construction de la nouvelle couche n'a pas besoin d'attendre un message du premier nœud de la couche précédente, car il a cette information en consultant tout simplement l'état de ses voisins. En d'autres termes, l'information est propagée avant que le nœud en ait besoin. Le nœud consulte l'information par simple lecture de l'état de voisin physique. Cela signifie qu'il y a pas besoin de transmettre des informations entre deux nœuds non voisins de couches différentes ou entre deux nœuds non voisins de la même couche. Cette efficacité s'explique par le fait que ACNI et ACI font de la synchronisation du changement d'état une condition non nécessaire. On note que ACI a besoin de $O(n)$ messages pour construire l'arbre couvrant.

3.3.5 Généralisation des algorithmes

Nous avons présenté des algorithmes qui traitent d'une configuration initiale en forme de chaîne où les voisins des nœuds sont dans directions SE et/ou NW. Nous montrons ici comment l'algorithme d'auto-reconfiguration peut être généraliser à n'importe quelle chaîne droite. Nous commençons par décrire la généralisation de la procédure de sélection de l'initiateur.

Predicates :
Initiator$^r(v)$: le nœud a au début un seul voisin dans la direction r, avec r $\in \{nw, ne, w\}$.
Node$^d(v)$: le nœud a initialement au plus deux voisins orientés d, avec d $\in \{nw-se, ne-sw, w-e\}$.
x : désigne l'orientation de la chaîne initiale $\in \{nw-se, ne-sw, w-e\}$.

Prédicats vérifiés seulement dans la première étape

Initiator$^{nw}(v) \equiv (\neg N_{nw}(v)) \wedge (\neg N_{sw}(v)) \wedge (\neg N_{ne}(v)) \wedge (\neg N_e(v)) \wedge (\neg N_w(v)) \wedge (N_{se}(v))$.
Initiator$^{ne}(v) \equiv (\neg N_{nw}(v)) \wedge (\neg N_{se}(v)) \wedge (\neg N_{ne}(v)) \wedge (\neg N_e(v)) \wedge (\neg N_w(v)) \wedge (N_{sw}(v))$.
Initiator$^w(v) \equiv$
$(\neg N_{nw}(v)) \wedge (\neg N_{sw}(v)) \wedge (\neg N_{se}(v)) \wedge (\neg N_{ne}(v)) \wedge (\neg N_e(v)) \wedge (\neg N_w(v)) \wedge (N_e(v))$.
Node$^{nw-se}(v) \equiv ((N_{nw}(v) \vee N_{se}(v)) \wedge (\neg N_{ne}(v)) \wedge (\neg N_e(v)) \wedge (\neg N_w(v)) \wedge (\neg N_{sw}(v))$.
Node$^{ne-sw}(v) \equiv ((N_{sw}(v) \vee N_{ne}(v)) \wedge (\neg N_{nw}(v)) \wedge (\neg N_{se}(v)) \wedge (\neg N_e(v)) \wedge (\neg N_w(v))$.
Node$^{w-e}(v) \equiv ((N_w(v) \vee N_e(v)) \wedge (\neg N_{nw}(v)) \wedge (\neg N_{sw}(v)) \wedge (\neg N_{se}(v)) \wedge (\neg N_{ne}(v))$.
State$_v(bad) \equiv Node^x(v) \wedge \neg Initiator^x(v)$.
State$_v^x(good) \equiv Initiator^x(v)$.
State$_v^x(spe) \equiv Initiator^x(v)$.

Prédicats vérifiés à chaque étape

Parent(v, v)$\equiv Initiator^x(v)$.
Parent(v, u)$\equiv (Parent(w, v), u \neq w) \wedge neighbor(v, u) \wedge State_u(bad)$.
isLeaf(v)$\equiv (\forall u \in N(v), \neg Parent(v, u) \wedge \neg Parent(v, v))$.
(P1) :State$_v^{nw}(good) \equiv (N_e(v) =$
$u1 \wedge State_{u1}(good) \wedge \neg N_{nw}(u1)) \vee State_v(3, good) \vee (State_v(2, good) \wedge (N_{ne}(v) =$
$u1 \wedge State_{u1}(spe)) \vee (N_w(v) = u1 \wedge State_{u1}(good))) \vee State_v(spe) \wedge Node^{nw-se}(v)$.
(P1) :State$_v^{ne}(good) \equiv (N_w(v) =$
$u1 \wedge State_{u1}(good) \wedge \neg N_{ne}(u1)) \vee State_v(3, good) \vee (State_v(2,) \wedge (N_{nw}(v) =$
$u1 \wedge State_{u1}(spe)) \vee (N_e(v) = u1 \wedge State_{u1}(good))) \vee State_v(spe) \wedge Node^{ne-sw}(v)$.
(P1) :State$_v^w(good) \equiv (N_{ne}(v) =$
$u1 \wedge State_{u1}(good) \wedge \neg N_e(u1)) \vee State_v(3, good) \vee (State_v(2, good) \wedge (N_{nw}(v) =$
$u1 \wedge State_{u1}(spe)) \vee (N_{sw}(v) = u1 \wedge State_{u1}(good))) \vee State_v(spe) \wedge Node^{w-e}(v)$.
(P1) :State$_v(s,\ good) \equiv (N_x(v) = \{U\}, |U| = s \wedge State_{\{u\}}(good))$.
(P1) :State$_v^{nw}(spe) \equiv (N_{nw}(v) = u1)) \wedge (N_{ne}(v) = u2, State_{u2}(spe)) \wedge Node^{nw-se}(v)$.
(P1) :State$_v^{ne}(spe) \equiv (N_{ne}(v) = u1)) \wedge (N_{nw}(v) = u2, State_{u2}(spe)) \wedge Node^{ne-sw}(v)$.
(P1) :State$_v^w(spe) \equiv (N_w(v) = u1)) \wedge (N_{nw}(v) = u2, State_{u2}(spe)) \wedge Node^{w-e}(v)$.
(P2) :moveAround$^{nw} bad_v(u, P_e) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) =$
$u \wedge State_u(bad)) \wedge Node^{nw-se}(v)$.
(P2) :moveAround$^{nw} bad_v(u, P_{se}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) =$
$u \wedge State_u(bad)) \wedge Node^{nw-se}(v)$.

La généralisation de ACI : GACI

(P2) :moveAround^{nw} $good_v(u, P_{ne}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u \wedge State_u(good)) \wedge Node^{nw-se}(v)$.
(P2) :moveAround^{nw} $good_v(u, P_e) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) = u \wedge State_u(good)) \wedge Node^{nw-se}(v)$.
(P2) :moveAround^{ne} $bad_v(u, P_w) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) = u \wedge State_u(bad)) \wedge Node^{ne-sw}(v)$.
(P2) :moveAround^{ne} $bad_v(u, P_{sw}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u \wedge State_u(bad)) \wedge Node^{ne-sw}(v)$.
(P2) :moveAround^{ne} $good_v(u, P_{nw}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{ne}(v) = u \wedge State_u(good)) \wedge Node^{ne-sw}(v)$.
(P2) :moveAround^{ne} $good_v(u, P_w) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u \wedge State_u(good)) \wedge Node^{ne-sw}(v)$.
(P2) :moveAround^{w} $bad_v(u, P_w) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{w}(v) = u \wedge State_u(bad)) \wedge Node^{w-e}(v)$.
(P2) :moveAround^{w} $bad_v(u, P_{sw}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u \wedge State_u(bad)) \wedge Node^{w-e}(v)$.
(P2) :moveAround^{w} $good_v(u, P_{nw}) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{w}(v) = u \wedge State_u(good)) \wedge Node^{w-e}(v)$.
(P2) :moveAround^{w} $good_v(u, P_w) \equiv isLeaf(v) \wedge State_v(bad) \wedge (N_{nw}(v) = u \wedge State_u(good)) \wedge Node^{w-e}(v)$.

La généralisation de ACI : GACI

Les noeuds centraux de la chaîne initiale déterminent l'orientation de la chaîne en fonction de la direction des deux voisins. La racine peut être distinguée par le principe qu'il ne dispose que d'un voisin dans la direction SW, SE ou E. Quelle que soit l'orientation de la chaîne, il n'existe qu'un seul noeud répondant à cette condition dans la chaîne initiale. Les nœuds centraux ont deux voisins en même temps, un dans la direction D et l'autre dans le sens inverse que nous appellerons $-D$ où $(D, -D) \in \{(SE, NW), (SW, NE), (E, W)\}$. L'autre extrémité de la chaîne (noeud feuille) a un voisin dans la direction NW, NE ou W. Après identification de l'orientation

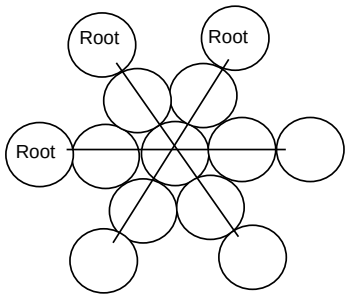

FIGURE 3.13 – Les trois cas possibles d'une chaîne linéaire

de la chaîne, un algorithme similaire à ceux présentés est appelé. Si la chaîne est de type où les nœuds peuvent avoir des voisins dans les directions NE, SE ou dans les deux directions, nous renvoyons aux mouvements nécessaires comme $ACNI^{-D}$ ou ACI^{-D} en fonction du type de mouvement considéré, sachant que on définit ACI^{-D} et $ACNI^{-D}$ comme les algorithmes

précédents avec des mouvements faits de NE à NW ou W, ou de NW à W. Selon le type de la chaîne, les nœuds doivent seulement changer les directions de mouvement dans leurs prédicats, l'algorithme *GACI* suivant traite avec tous les types de la chaîne.

3.3.6 Simulation et comparaison

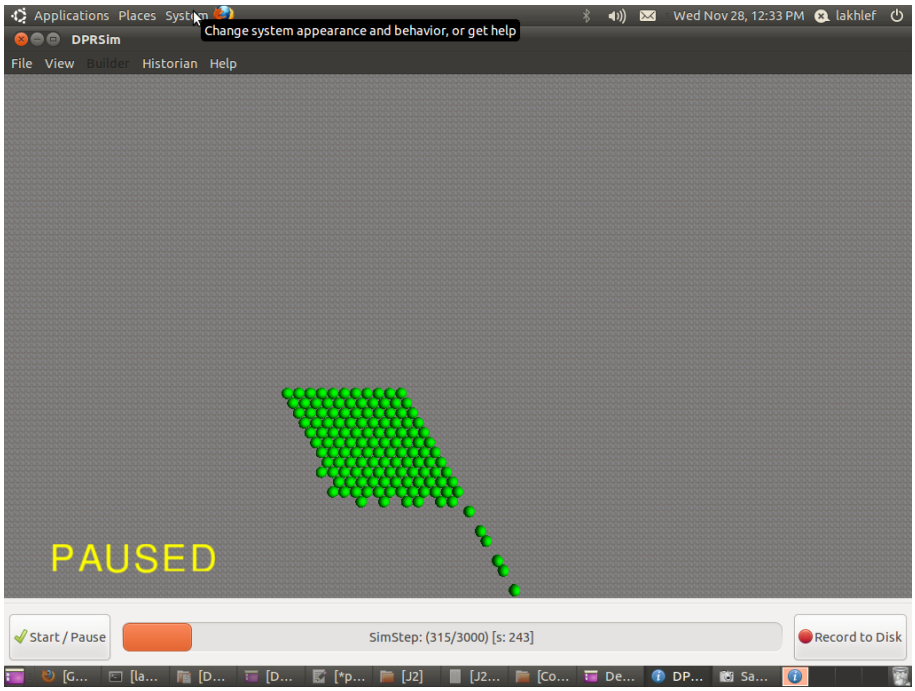

FIGURE 3.14 – Une instance d'exécution de l'algorithme ACI

La simulation a été faite avec le langage déclaratif Meld et le simulateur DPRSIM. Dprsim est un environnement pour la simulation des mouvements des microrobots. Meld est un langage déclaratif qui, en utilisant Dprsim, simule les algorithmes distribués pour les microrobots. Les figures 3.14 et 3.15 représentent, respectivement des exemples de l'algorithme ACNI et ACI en cours d'exécution. Dans nos simulations le rayon du nœud est de 1 mm [1]. Nous avons utilisé un portable Intel(R) Core(Tm) i5, 2.53 Ghz avec 4 GO de mémoire. Les résultats de ces simulations sont en accord avec les analyses numériques obtenues précédemment, en particulier en ce qui concerne le nombre de mouvements total et le nombre de mouvements par noeud. Les nœuds ont appliqué la procédure de partitionnement des nœuds en niveaux puis ont calculé le nombre de mouvements propres prévu. A la fin de l'algorithme, chaque nœud compare les résultats de la prédiction aux résultats de l'exécution. La figure 3.16 représente le nombre de mouvements en fonction du nombre de nœuds, avec $f(n) = (\lceil\sqrt{(n)}\rceil \lceil\sqrt{(n)}\rceil) - \lceil\sqrt{(n)}\rceil, g(n) = n - \sqrt{(n)}$, et $h(n) = n$. La figure 3.17 représente le temps d'exécution en fonction du nombre de nœuds.

1. Le temps d'un mouvement dépend de la taille (le diamètre) du microrobot, comme indiqué dans la section 4.

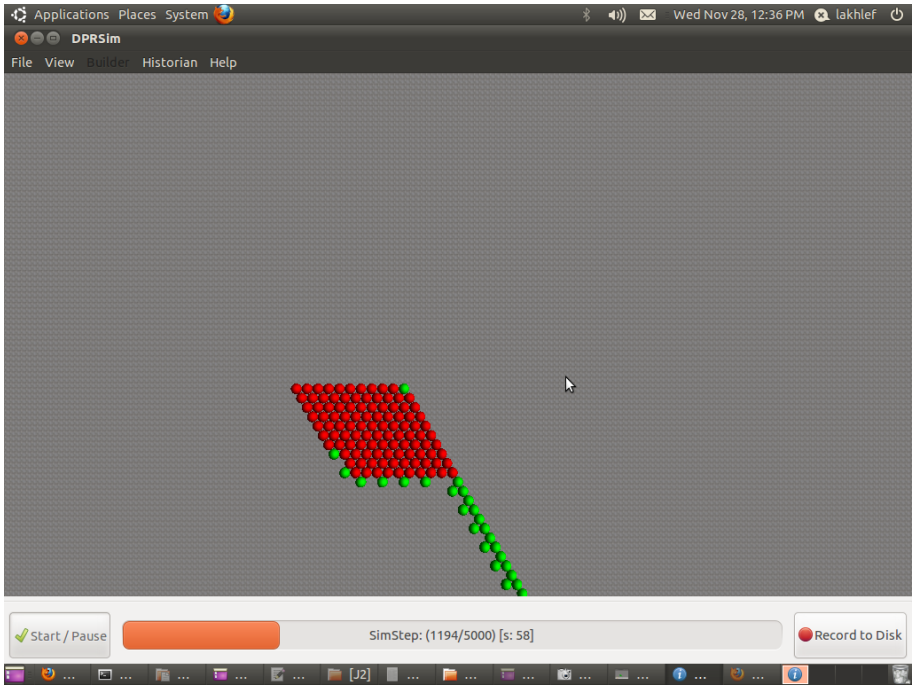

FIGURE 3.15 – Une instance d'exécution de l'algorithme ACNI

Pour analyser la courbe représentant le plus grand nombre de mouvement, on note une certaine valeur n comme la taille du réseau.

On remarque que si n est une racine carrée (n est le carré d'un nombre entier), alors le nombre de mouvements de ACNI est toujours inférieur à celui de ACI, mais si n n'est pas une racine carrée, alors on distingue deux cas : si n n'est pas une racine carrée, alors il y a un nombre Ms qui est la racine carrée minimum supérieure à n. Il existe également un certain nombre Mi, la racine carrée maximume inférieur à n. Dans cette courbe, jusqu'à 50 nœuds pour certaines valeurs de n on remarque que le plus grand nombre de mouvements de ACNI est inférieur au plus grand nombre de mouvement de ASC, sachant que dans ces cas, n est plus proche à Ms. Toutefois, si n est plus proche de Mi on remarque que le nombre de mouvements de ASC est inférieur à celle de ACNI pour les valeurs de $n = 100$. On note également que si n est au milieu de Mi et Ms comme les cas de 30 et 90 nœuds, alors les deux algorithmes ont le même plus grand nombre de mouvements. Pour les courbes qui représentent le temps d'exécution, on voit que si on compte le temps de construction de l'arbre de ACI la différence augmente avec une accélération quasi constante, mais si on compte le temps de l'arbre ($O(n)$) la différence sera très élevée.

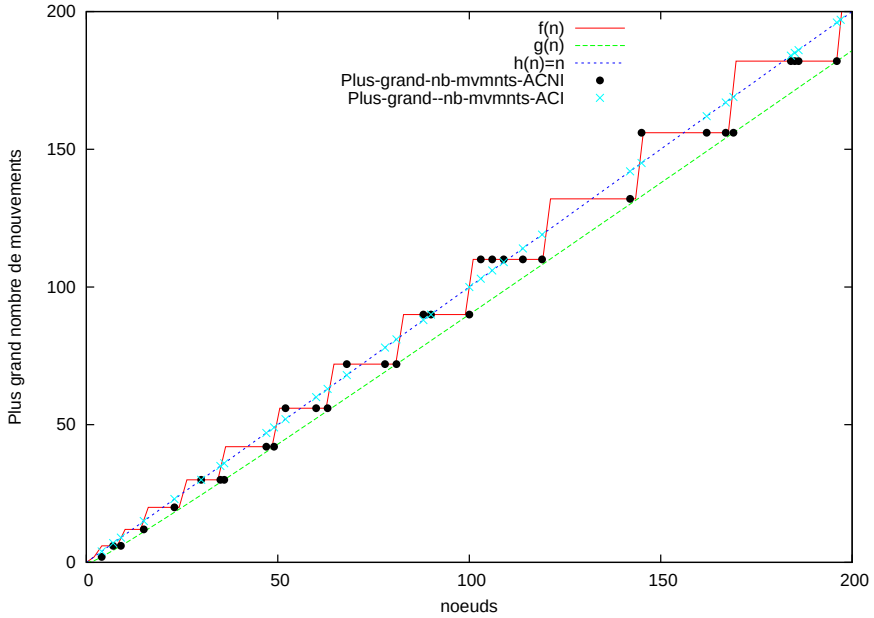

FIGURE 3.16 – Le plus grand nombre de mouvements dans ACI et ACNI

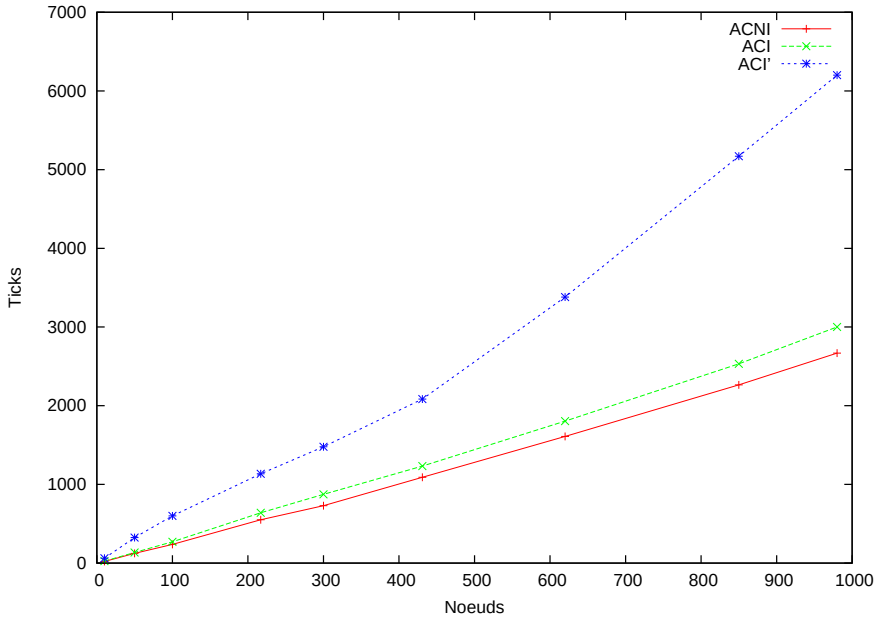

FIGURE 3.17 – Le temps d'éxecution de ACI et ACNI

3.4 Conclusion

Dans ce chapitre on a présenté deux solutions distribuées pour la reconfiguration des microrobots performantes en terme de consommation énergétique et de mémoire de stockage. Le but de ces algorithmes est d'optimiser la topologie logique du réseau des microrobots. Les deux solutions n'utilisent pas la carte de la forme cible. Le premier algorithme garantit une connexité non instantanée du réseau et le deuxième algorithme utilise l'arbre où seulement les feuilles peuvent se déplacer afin de garder le réseau connexe tout au long de l'algorithme. Nous avons étudié deux types de mouvements où le nœud peut recevoir de l'aide pour créer le mouvement et avoir les positions exactes souhaitées.

Les deux algorithmes proposés sont caractérisés par une complexité constante en mémoire de stockage (seulement trois états sont nécessaires pour chaque nœud) rendant l'algorithme à l'abri des problèmes d'échelle (le nombre de microrobots). Les échanges de messages sont limités à des consultations de l'état des voisins. Par conséquent la reconfiguration du système est rapide. On a fourni aussi une comparaison entre les deux solutions afin de montrer l'impact des types de mouvements sur les performances de la reconfiguration.

CHAPITRE 4

PROTOCOLES PARALLÈLES POUR L'AUTO-RECONFIGURATION

Sommaire

4.1 Introduction

Les algorithmes d'auto-reconfiguration utilisant une carte de la forme cible manquent d'évolutivité et d'efficacité dans l'utilisation de la mémoire. En effet, pour couvrir une forme cible définie par un ensemble de positions, il est nécessaire de subdiviser la forme cible en de très petites unités en fonction de la taille des microrobots, ce qui donne un nombre très élevé de positions. Par conséquent, chaque noeud doit avoir une très grande capacité mémoire pour stocker toutes ces positions. Ceci explique l'importance d'un protocole de reconfiguration sans carte de la forme cible.

Dans ce chapitre on présente deux solutions pour la reconfiguration parallèle des microrobots. L'objectif de ces solutions est d'optimiser le temps d'exécution de l'algorithme et minimiser le nombre de mouvements pour chaque nœud et par conséquent, donner plus d'efficacité énergétique pour les microrobots.

Dans ce chapitre, on étudie deux algorithmes distribués garantissant l'efficacité énergétique des microrobots avec le souci de l'amélioration des temps de convergence de la reconfiguration. L'objectif de ces deux algorithmes est d'introduire la parallélisme en mouvement sur les algorithmes précédents proposés, pour avoir moins de mouvements et moins de temps de convergence. Nous conservons ici les caractéristiques positives du modèle sans carte de la forme cible pour

la reconfiguration d'une chaîne de micrororbots en un carré. Les algorithmes proposés prennent en compte les contraintes technologiques sur des microrobots MEMS relatives à la mémoire et l'énergie limitées.

4.2 Modèle, définitions et outils

Dans cette section, on commence par rappeler le modèle et les définitions utilisés par l'algorithme de base décrit dans le chapitre précédent puis nous introduisons les notions nécessaires à la compréhension et la validation de nos algorithmes parallèles.

Lemme 1 *Soit x un nombre entier positif. Il est bien connu que si x est impair\pair, alors x^2 est un nombre impair\pair*[2]*.*

Puisque x est impair\pair, on peut écrire $x = 2n+1 \backslash x = 2n$. Donc, $x^2 = (2n+1)^2 \backslash x^2 = (2n)^2$. Ainsi, $x^2 = 4n^2 + 4n + 1 = 2(2n^2 + 2n) + 1 \backslash x^2 = 2(2x^2)$; qui est un nombre impair\pair.

Lemme 2 *Soit x un nombre carré (x est un entier qui est le carré d'un entier). Si x est pair, alors $\sqrt{x}$ est un nombre pair.*

Pour la preuve nous utilisons le raisonnement par l'absurde. Comme x est un nombre pair et il est le carré d'un nombre entier, on peut donc écrire $x = n^2$. Par conséquent, $\sqrt{x} = \sqrt{n^2}$. Supposons que n est impair, donc il existe un nombre k avec $n = 2k+1$. Ainsi, $\sqrt{x} = \sqrt{(2k+1)^2}$. Donc, $x = 2(2k^2 + 2k) + 1$ ce qui représente une contradiction, car cette valeur est impair alors que x est pair. Donc, $\sqrt{x}$ est pair.

Lemme 3 *Soit x un nombre carré. Si x est impair, alors $\sqrt{x}$ est un nombre impair.*

Comme x est impair, il existe un entier h avec $x = 2h + 1$. Donc $\sqrt{x} = \sqrt{2h+1}$. On note qu'il existe un entier η avec $\sqrt{2h+1} = \sqrt{2\eta} + 1$ où $h = \eta + \sqrt{2\eta}$.

Par conséquent, la valeur $\sqrt{2h+1} = 4\sqrt{\eta} + 1$ qui est un nombre impair.

Théorème 1 *Soit y un nombre carré impair\pair (y est un entier carré d'un entier), alors le prochaine nombre carré impair\ pair est $y + 4\sqrt{y} + 4$*

Puisque y est un nombre carré impair\pair, on a $\sqrt{y} = \rho$, with ρ est un entier impair\pair (à partir de lemme 5.10, lemme 5.11 et lemme 3). Puisque ρ est impair\pair, le prochaine nombre impair\pair est $r = \rho + 2$, et puisque $r^2 = (\rho + 2)^2 = \rho^2 + 4\rho + 4$ est impair\pair (de lemme 5.10, lemme 5.11 et lemme 3), on trouve $r^2 = y + 4\sqrt{y} + 4$

Théorème 2 *Soit y un nombre carré (y est un entier qui est le carré d'un entier), alors si y est impair\pair le prochaine nombre carré pair\impair est $y + 2\sqrt{y} + 1$*

Puisque y est un nombre entier carré impair\pair, nous déduisons des lemme 5.10, lemme 5.11 et lemme 3 que $\sqrt{y} = \rho$, avec ρ est un entier impair\pair. Comme ρ est impair\pair, le prochain nombre pair\impair est $r = \rho+1$, et puisque $r^2 = (\rho+1)^2 = \rho^2+2\rho+1$ est pair\impair (de lemme 5.10, lemme 5.11 et lemme 3), on trouve $r^2 = y + 2\sqrt{y} + 1$

2. le caractère "\" signifie 'respectivement' dans les lemmes et les théorèmes

4.3 Protocoles Proposés

Dans cette section on présente deux algorithmes de reconfiguration parallèles pour les microrobots : un algorithme parallèle pour la reconfiguration qui garantit une connexité instantanée du système des microrobots, tt un algorithme parallèle pour la reconfiguration qui garantit une connexité non instantanée du système des microrobots. Chacun des algorithmes se décompose en deux étapes. La première étape consiste à trouver le nœud du milieu de la chaîne *mi* qui initialise l'algorithme de reconfiguration. La deuxième étape constitue l'algorithme de reconfiguration.

4.3.1 Protocole parallèle avec une connexité instantanée (PPCI)

Dans l'algorithme PPCI [53, 54] (presenté ci-après), un nœud ne peut se déplacer que autour de son voisin physique. Pour former la matrice du carré avec $\sqrt{N} \times \sqrt{N}$ nœuds, avec N est la

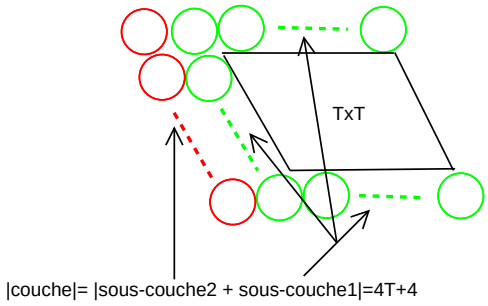

FIGURE 4.1 – Le nombre de nœuds ajoutés pour atteindre le prochaine carré lorsque n est impair

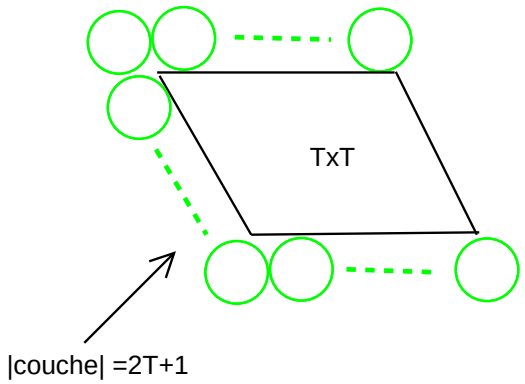

FIGURE 4.2 – Le nombre de nœuds ajoutés pour atteindre le prochaine carré lorsque n est pair

taille de la chaîne, l'algorithme utilise un processus incrémental qui commence (sur la base des théorèmes 1 et 2) par un carré correct (par exemple 1x1). Puis à chaque itération deux nouvelles sous-couches de nœuds sont ajoutées pour former le carré $T \times T$. La première sous-couche est composée de $3T + 2$ nœuds entourant le carré précédent $((T-2) \times (T-2))$ des 3 côtés (S), (N) et (W). Puis pour compléter le nouveau carré $T \times T$, une deuxième sous-couche de $T + 2$ nœuds encadre le côté (W) de forme partielle. Si N est pair, pour construire la dernière couche du carré final, l'algorithme ajoute $2\sqrt{N} + 1$ nœuds. Les figures 4.1 et 4.2 montrent un exemple du

fonctionnement de la dernière itération dans les cas respectivement où N est impair et pair. Dans cet algorithme (PPCI), chaque nœud ne peut se déplacer que autour de son voisin physique. Pour assurer une connexité instantanée seuls les nœuds qui empêchent la disconnectivité du réseau peuvent se déplacer autour des voisins. Pour ce faire, nous introduisons l'utilisation de l'arbre pour gérer dynamiquement les nœuds feuilles qui peuvent se déplacer.
Le protocole commence par trouver le nœud milieu de la chaîne pour faire des mouvements en parallèle. Soit N la taille du réseau, et $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$. Les nœuds de la chaîne initiale sont numérotés de 1 à N du Sud au Nord. Le nœud central de la chaîne d'indice (mi) initialise l'algorithme de reconfiguration. Si n est impair l'indice du nœud médian est $mi = \frac{n+1}{2}$, comme le montre l'exemple de la figure 4.4. Si n est pair alors $mi = \frac{n}{2} - (\frac{\sqrt{n}}{2} - 1)$, comme le montre l'exemple de la figure 5.4.
Pour que le nœud médian d'indice mi soit sélectionné nous supposons que les nœuds connaissent la taille du réseau et par conséquence connaissent l'indice du nœud initiateur. Le nœud de l'extrémité Sud de la chaîne initialise un compteur et le diffuse vers ses voisins. Chaque nœud qui reçoit ce message incrémente le compteur jusqu'à ce la valeur du compteur obtenu soit celle de l'indice mi. Le nœud mi active alors sont prédicat $medChain(v)$ pour signifier qu'il est initiateur.

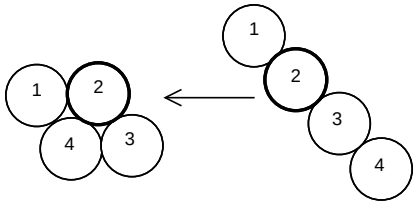

FIGURE 4.3 – Un exemple d'identification d'initiateur quand n est pair, dans cet exemple, l'initiateur est le nœud 2

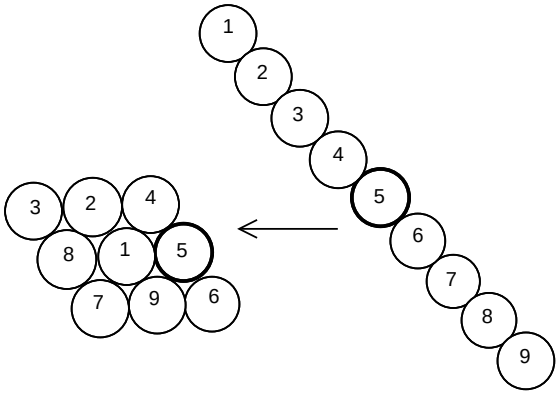

FIGURE 4.4 – Un exemple d'identification d'initiateur quand n est impair, dans cet exemple, l'initiateur est le nœud 5

Variables et prédicats :

- initiator$_v()$: le nœud v qui initialise l'algorithme.
- state$_v(X)$: le nœud v prend l'état X, avec :
  $X \in \{well, bad, int, nper, mnper, top, bottom, \neg nper, \neg mnper, \neg int\}$, v ne peut pas prendre les états *well* et *bad* en même temps.
- moveAround$state_v(u, P_x)$: se déplacer autour du nœud voisin u qui a l'état *state* de telle sorte u devient le voisin de v dans la direction x par rapport à v.
- parent(v, u) :le nœud v est parent du nœud u dans l'arbre.
- isLeaf(v) : v est un nœud feuille dans l'arbre.

Prédicats vérifiés seulement dans la première étape

1. initiator$_v() \equiv medChain(v)$.
2. state$_v(bad) \equiv connected_v \wedge \neg initiator_v()$.
3. parent$(v, v) \equiv initiator(v)$.
4. state$_v(well) \equiv initiator_v()$.
5. state$_v(nper) \equiv initiator_v()$.

Prédicats vérifiés à chaque étape

6. state$_v(top) \equiv (N_{se}(v) = u, initiator_u()) \vee (N_{se}(v) = u, state_u(top))$.
7. state$_v(bottom) \equiv (N_{nw}(v) = u, initiator_u()) \vee (N_{nw}(v) = u, state_u(top))$.
8. parent$(v, u) \equiv (parent(w, v), u \neq w) \wedge (u \in N(v)) \wedge (state_u(bad)) \wedge (/\exists z \in N(v), parent(v, z))$.
9. isLeaf$(v) \equiv ((\forall u \in N(v), \neg parent(v, u)) \wedge \neg parent(v, v))$.
10. state$_v(mnper) \equiv (((N_{se}(v) = u, state_u(nper)) \vee (N_e(v) = u, state_u(nper))) \wedge initiator_u())$.
11. state$_v(mnper) \equiv (N_e(v) = u, state_u(nper)) \wedge (\neg state_v(nper)) \wedge (state_v(int) \vee state_v(well))$.
12. state$_v(nper) \equiv (N_e(v) = u, state_u(mnper)) \wedge (\neg state_v(mnper)) \wedge (N_{se}(v))$.
13. state$_v(int) \equiv ((N_e(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{se}(v) = u, state_u(well)) \wedge (N_w(v))) \vee (N_e(v) = u1, N_{se}(v) = u2, state_{u1}(int), state_{u2}(int)) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_w(v)))$.
14. state$_v(well) \equiv ((N_e(v) = u, state_u(int)) \wedge (N_{nw}(v))) \vee ((N_e(v) = u, state_u(int)) \wedge (N_{se}(v)))$.
15. state$_v(well) \equiv (N_w(v) = u, state_u(well))$.
16. state$_v(well) \equiv state_v(bad) \wedge (N_{se}(v) = \oslash) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well))$.

17. state$_v(well) \equiv state_v(bad) \wedge (N_{se}(v) = \oslash) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well)) \wedge ((N_e(v) = u1, state_{u1}(well)))$.

18. moveAround$bad_v(u, P_e) \equiv canMove_v() \wedge (N_{se}(v) = u, state_u(bad), state_u(top))$.

19. moveAround$bad_v(u, P_{ne}) \equiv canMove_v() \wedge (N_e(v) = u, state_u(bad), state_u(top))$

20. moveAround$int_v(u, P_{se}) \equiv canMove_v() \wedge (N_{sw}(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper))$.

21. moveAround$well_v(u, P_{se}) \equiv canMove_v() \wedge (N_{sw}(v) = u, state_u(well), state_u(top))$.

22. moveAround$int_v(u, P_e) \equiv canMove_v() \wedge (N_{se}(v) = u, state_u(int), state_u(top))$.

23. moveAround$well_v(u, P_e) \equiv canMove_v() \wedge (N_{se}(v) = u, state_u(well), state_u(top))$.

24. moveAround$bad_v(u, P_{ne}) \equiv canMove_v() \wedge (N_{nw}(v) = u, state_u(bad)) \wedge state_u(bottom)$.

25. moveAround$bad_v(u, P_e) \equiv canMove_v() \wedge (N_{ne}(v) = u, state_u(bad), state_u(bottom))$.

26. moveAround$well_v(u, P_{ne}) \equiv canMove_v() \wedge (N_{nw}(v) = u, state_u(well), state_u(bottom))$.

27. moveAround$well_v(u, P_{se}) \equiv canMove_v() \wedge (N_e(v) = u, state_u(well), state_u(bottom), (\neg state_u(nper))$.

28. moveAround$well_v(u, P_e) \equiv canMove_v() \wedge (N_{ne}(v) = u, state_u(well), state_u(bottom)(\neg state_u(nper))$.

29. moveAround$int_v(u, P_{ne}) \equiv canMove_v() \wedge (N_{nw}(v) = u, state_u(int), state_u(bottom))$.

30. moveAround$int_v(u, P_e) \equiv canMove_v() \wedge (N_{ne}(v) = u, state_u(int), state_u(bottom), (\neg state_u(nper))$.

31. moveAround$well_v(u, P_e) \equiv canMove_v() \wedge state_v(bottom) \wedge (N_{ne}(v) = u, state_u(well), state_u(mnper), (\neg state_u(nper))$.

32. canMove$_v() \equiv isLeaf(v) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad)$.

L'algorithme PPCI

4.3.1.1 Description et analyse

A chaque étape de l'algorithme PPCI, les prédicats satisfaits sont choisis et exécutés. L'algorithme distribué converge vers la forme désirée en utilisant un processus incrémental. Après chaque incrément, les nœuds formant les couches construites font partie de la forme finale. De manière similaire aux algorithmes du chapitre précédent, les nœuds de forme construite aident les nœuds voisins à se positionner dans des positions correctes.
Le nœud du milieu de la chaîne se déclare comme initiateur avec le prédicat (1). L'initiateur, qui est la racine de l'arbre couvrant initialise la construction de l'arbre et devient un parent de lui-même (3). Un nœud s'il n'a pas de parent devient un enfant d'un des parents voisins (8) et un nœud est une feuille si tous ses voisins sont des parents (9). Les nœuds qui sont au-dessus de l'initiateur prennent l'état *top* avec le prédicat (6), les autres nœuds qui sont sous l'initiateur prennent l'état *bottom* (7). Initialement, tous les nœuds sont initialisés avec l'état *bad* sauf l'initiateur (2), qui prend les états *well* et *nper* avec (4) et (5). Les nœuds ayant les états *well* ou

int sont des nœuds déjà dans la forme cible et ne peuvent plus se déplacer.

Pour réduire d'une façon optimale le temps de convergence de l'algorithme, la ligne horizontale de la forme finale où se trouve l'initiateur doit comporter autant de nœuds ayant l'état *top* que de nœuds ayant l'état *bottom* si N est impair. Si N est pair, un nœud ayant l'état *nper* est ajouté aux nœuds ayant l'état *top*. L'initiateur prend l'état *nper* (5) pour empêcher les nœuds voisins de se déplacer autour de lui et rejoindre la ligne de l'initiateur (la garde $(\neg state_v(nper))$).

L'état *mnper* est un état intermédiaire utilisé pour propager l'état *nper* vers les autres nœuds afin de garantir un parallélisme optimal. Un nœud, dont le voisin côté (E) a l'état *nper*, prend l'état *mnper* (11). Un nœud, dont l'initiateur est le voisin (SE), prend l'état *mnper* (10). Les nœuds ayant dans le sens (E) un voisin à l'état *mnper* deviennent *nper* à leur tour (12). Par la suite, le nœud avec l'état *nper* empêche ses voisins de rejoindre la ligne de l'initiateur, puisque ces nœuds vérifient les prédicats (21), (22), (23), (26), (27), (28), (29) et (30).
L'état *int* est un état intermédiaire qui permet d'ajouter une couche non complète à la forme carrée. Par conséquent, les nœuds qui ont des voisins à l'état *well* prennent l'état *int* avec le prédicat (13). Le premier nœud à prendre l'état *int* est le nœud qui atteint la ligne de l'initiateur. L'état *int* se propage alors aux nœuds voisins ayant l'état *well*. On note que les nœuds ayant l'état *well* et les nœuds ayant l'état *int* ensemble, ne forment pas un carré, il sera un carré si tous les nœuds dont l'état *int* ont dans la direction (W) un voisin ayant l'état *well*. La vague de changement d'état à *well* commence à partir des prédicats (15), (16) et (17).

Avec les prédicats (18) et (19) les nœuds feuilles ayant l'état *top* descendent à l'axe de la chaîne. Ainsi que, les nœuds feuilles ayant l'état *bottom* montent vers le centre de la chaîne avec les prédicats (24) et (25). Avec les prédicats (20) / (21), le nœud feuille v à l'état *bad* se déplace autour du voisin u ayant l'état *top* et *int*/*well*, le nœud u devient un voisin dans la direction (SE) par rapport à v. Avec les prédicats (22) / (23), le nœud feuille v qui a l'état *bad* se déplace autour du voisin u ayant l'état *top* et *int*/*well*, le nœud u devient un voisin dans la direction (E) par rapport à v. Avec les prédicats (26) / (27) / (28), le nœud feuille v à l'état *bad* se déplace autour du voisin u ayant les états *well* et *bottom* states, le nœud u devient un voisin dans la direction NE / SE / E par rapport à v.

Théorème 3 *Si N est la taille du réseau et $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$ est impair, le plus grand nombre de mouvements effectués par un nœud est $((n+1)/2) + N - n$.*

Théorème 4 *Si N est la taille du réseau et $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$ est pair, avec $N \geq 5$, le plus grand nombre de mouvements effectués par un nœud est $(\sqrt{n}/2) + N - (n/2) - 1$.*

Exemple : La figure 4.5 montre un exemple qu'on explique, et la figure 4.6 montre un autre exemple sans explication. Dans la figure 4.5 :

– A $t0$: avec le prédicat (2) chaque nœud prend l'état b (*bad*), avec (6) les nœuds 1 qui est au-dessus de l'initiateur prend l'état t (*top*), avec le prédicat (7) les nœuds 3 et 4 situés au sud de l'initiateur prennent l'état B (*bottom*), avec le prédicat (4) l'initiateur prend l'état w (*well*), et avec (5) il prend l'état n (*nper*).

– Pour arriver à l'étape suivante $t1$, le nœud 1 se déplace autour du nœud 2 en utilisant le prédicat (18), et le nœud 4 se déplace autour du nœud 3 en utilisant le prédicat (26). Le nœud

1 prend l'état m (*mnper*) avec le prédicat (10). Le nœud 1 ne peut pas se déplacer autour du nœud 2 avec le prédicat (19) puisque le nœud 2 a l'état n. Le nœud 4 se déplace à la position nouvelle avec le prédicat (24).

– Pour arriver à l'étape suivante $t2$, le nœud 4 se déplace autour du nœud 3 en utilisant le prédicat (25). Après, le nœud 4 prend l'état i (*int*) avec le prédicat (13).

– A $t3$, la forme cible est obtenue. Les nœuds 1 et 4 prennent l'état i avec le prédicat (13).

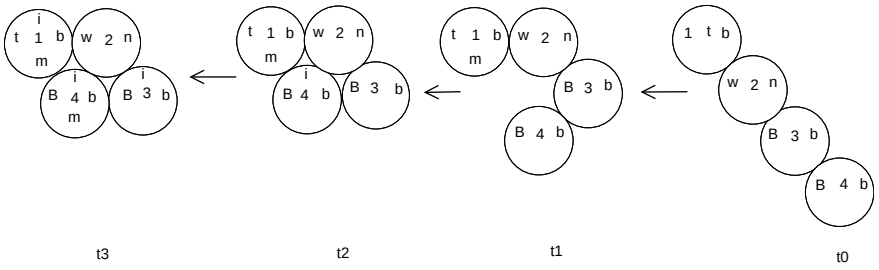

FIGURE 4.5 – Un exemple d'exécution de PPCI avec quatre nœuds

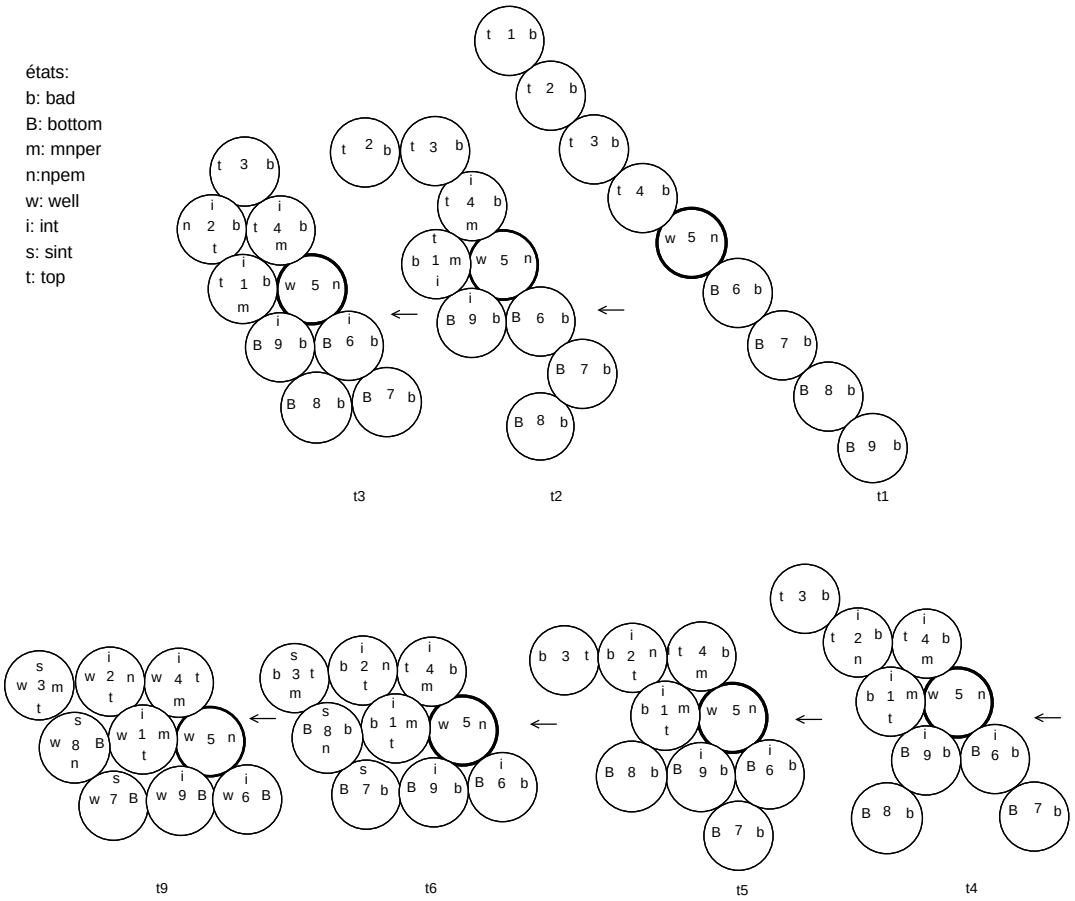

FIGURE 4.6 – Un exemple d'exécution de PPCI avec neuf nœuds, l'initiateur est le nœud 5

4.3.1.2 Nombre d'états nécessaires

Dans cette section, nous montrons que dix états sont nécessaires et suffisants pour obtenir la convergence de l'algorithme. Évidemment, avec un seul état, les nœuds n'ont aucun moyen de distinguer s'ils sont dans des bonnes positions ou non. Supposons une variante de PPCI avec seulement deux états *bad* et *well*. Avec ces deux états, on peut dire que le nœud qui a l'état *well* est un nœud stable (ne bouge pas) et il appartient à la forme cible, et le nœud ayant l'éat *bad* se déplace autour des nœuds ayant l'état *well*. Ainsi, avec ces deux états, les nœuds collaborent entre eux pour construire à chaque itération une nouvelle couche en modifiant l'état de *bad* à *well*. Supposons un ensemble S de nœuds ayant l'état *well* correctement positionnés dans la forme cible. En fonction de certaines conditions C l'ensemble B de nœuds ayant l'état *bad* vont changer leur état à *well* afin de stabiliser la nouvelle couche. Cependant, puisque il y a seulement deux états les autres nœuds qui sont voisins de l'ensemble B ont également ces conditions C. Ils vont changer leurs états à *well* et ils deviennent stables même s'ils ne sont pas dans la couche en cours de construction entraînant la non convergence de l'algorithme.
Deux états supplémentaires sont nécessaires, l'état *top* et *bottom*, ces deux états sont indispensables pour éviter l'inter-blocage de l'algorithme PPCI. Les prédicats (18) et (19) sont exécutés par les nœuds pour descendre au milieu de la chaîne, alors que les prédicats (24) et (25) sont utilisés pour remonter vers la ligne centrale. En l'absence des états *top* et *bottom* les nœuds oscilleront autours des mêmes positions en exécutant à la suite les deux prédicats (18,25), (25,18), (19,24) ou (24,19). Ceci conduit l'algorithme à des boucles d'exécutions qui empêchent sa convergence (terminaison). L'utilisation des quatre états *bad*, *well*, *top* et *bottom* est insuffisante car il est nécessaire d'ajouter un état intermédiaire *int* pour séparer les nœuds voisins ayant l'état *bad* et les nœuds ayant l'état *well*. En ajoutant cet état, le nœud qui a l'état *int* peut modifier les conditions C qui seront C'. De cette manière, les voisins de B ne peuvent pas changer leur état à *well* avec C', puisque ils ne forment pas une couche complète. Supposons une variante du PPCI avec six états *bad*, *well*, *top*, *bottom*, *int*, et $\neg int$. Avec cinq états, l'inter-blocage est évité, et les conditions pour changer l'état à *well* sont gérées. Cependant, les nœuds ayant l'état *int* font une nouvelle couche adjacente au carré correct courant $\sqrt{Z} * \sqrt{Z}$, le nombre de nœuds ayant *int* ajouté est $3\sqrt{Z}+2$. Par conséquent, puisque $\sqrt{Z} * \sqrt{Z} + 3\sqrt{Z}+2$ n'est pas une racine carré (suivant les théorèmes 1 et 2), la forme n'est pas un carré. Pour devenir un carré il faut ajouter $\gamma = \sqrt{Z}+2$ nœuds. Ces nœuds seront du côté (W) par rapport aux nœuds ayant l'état *int*. Les γ nœuds peuvent obtenir l'état *well* parce que la forme est un carré (intermédiaire ou finale). Avec six états *bad*, *well*, *top*, *bottom*, *int*, et $\neg int$, le parallélisme n'est pas optimal et la consommation d'énergie n'est pas bien équilibrée entre les nœuds. Pour faire un parallélisme optimal, PPCI fait deux rectangles en parallèle dont la combinaison donne un carré. Pour faire ça, il faut utiliser un autre état (*nper*) qui contrôle les mouvements des noeuds au milieu de la chaine. Aussi, pour propager l'état *nper* aux noeuds concernés du milieu, nous devons utiliser un autre état *mnper*. Les états $\neg nper$ et $\neg mnper$ doivent être utilisés pour vérifier si le noeud voisin a les états *nper* et *mnper* respectivement.

4.3.1.3 Complexité des messages envoyés

PPCI utilise $O(N)$ messages, les messages de la construction de l'arbre ($O(N/2)$) et les messages pour chercher le nœud du milieu ($O(N/2)$). L'action la plus intéressante pour l'échange

de messages dans l'algorithme est celle activée par le changement d'état de *bad* à *int* et de *int* ou *bad* à *well* avec les prédicats (15), (16) et (17). Si un nœud prend l'état *well* avant de vérifier que les s nœuds de la couche courante, qui se sont déplacés avant lui, sont à l'état *well*, la procédure n'aboutit alors pas à la forme souhaitée. Les prédicats (10), (11), (12), (13), (14), (15) et (16) assurent sans échange de messages que le nœud change d'état que si tous les nœuds qui sont passés avant ont changé leurs états à *int* ou à *well*. Par conséquent, le premier nœud qui commence la construction de la nouvelle couche n'a pas besoin d'attendre le message du premier nœud qui a commencé la construction de la couche précédente. Puisque le nœud qui est en train de vérifier les prédicats (10), (11), (12), (13), (14), (15) et (16) peut avoir cette information en consultant l'état de ses voisins. En d'autres termes, le message a été envoyé avant que le nœud ait eu besoin de connaître l'état de l'expéditeur. Tout au long de l'algorithme l'algorithme PPCI n'a pas besoin de transmettre des informations entre deux nœuds non voisins de la nouvelle couche. Cette efficacité s'explique par le fait que la synchronisation dans le changement d'état n'est pas requise pour les nœuds qui sont dans la même couche.

4.3.1.4 L'algorithme garantit une connexité instantanée

Cet algorithme garantit une connexité instantanée car l'utilisation de l'arbre couvrant fait qu'à partir d'un état connexe le déplacement de nœuds feuilles ne peut pas provoquer de discontinuité dans le réseau (produire différentes composantes connexes). En effet la nature des mouvements opérés par les feuilles peut être classifiée en deux catégorie. (1) Un nœud tourne autour d'un autre sans obtenir de nouveau voisin.

Dans ce cas, il n'y a pas de *ti* où ne peut être envoyé puisque le réseau est toujours connexe, (2) le nœud a un nouveau voisin après avoir effectué le déplacement. Dans ce cas le nœud aura ce nouveau voisin avant de quitter l'ancien voisin, cela signifie qu'il y a une autre route pour le message du chemin $C_{v,u}$ qui ne sera pas bloqué pour tout ti, $i \in \{1, ..., n\}$.

4.3.1.5 Prédire le nombre de mouvements pour chaque nœud

Dans cette section, nous présentons comment rendre l'algorithme robuste tout en tenant compte de la consommation d'énergie. Ceci est réalisé en prédisant le nombre de mouvements pour chaque nœud. Ainsi, chaque nœud connait la quantité d'énergie qu'il aura à consommer. En outre, le nœud, par cette prédiction peut s'assurer qu'il a correctement exécuté le protocole. Pour prévoir le nombre de mouvements pour chaque nœud nous prenons un partitionnement des nœuds en 3 groupes (A), (B) et (C) si $n = N$, ou à 3 groupes, (A), (B), (C') si $N > n$, comme le montre la figure 4.7. Pour appliquer les fonctions de prédiction, à chaque nœud est associé un niveau (L), un nombre spécial et éventuellement un indice spécial. Pour chaque groupe, nous donnons la formulation des fonctions de calcul du nombre de mouvements d'un nœud. On note que les procédures de partitionnement sont toujours réalisées du haut au bas de la chaîne.

– La taille du groupe (A) est $|(A)| = \frac{n-\sqrt{n}}{2}$ si n est impair. Ou $|(A)| = \frac{n}{2} - \sqrt{n} + 1$ si n est pair.

– La taille du groupe (B) est $|(B)| = \sqrt{n}$.

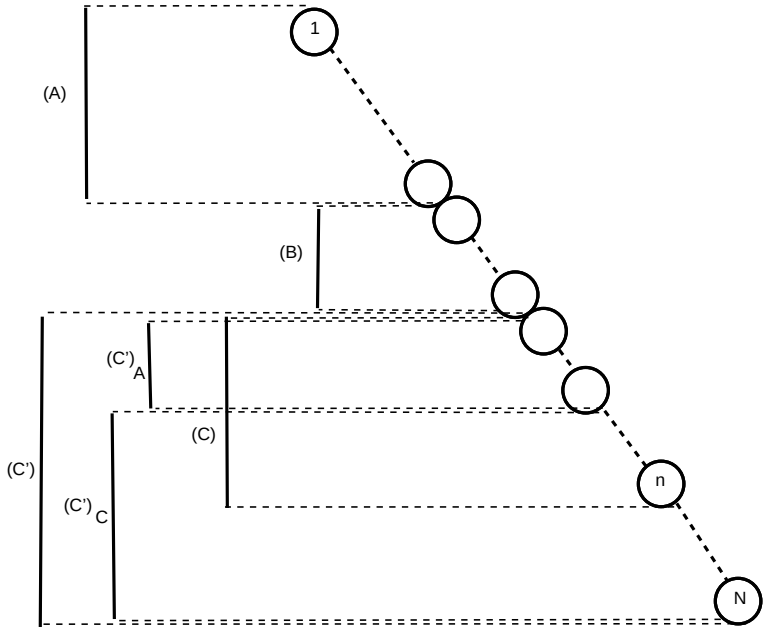

FIGURE 4.7 – Les modèles de la procédure de partitionnement pour calculer le nombre de mouvements

– La taille du groupe (C) est $|(C)| = \frac{n-\sqrt{n}}{2}$, si $n = N$ et n est impair. Ou $|(C)| = n - \frac{n}{2} - 1$, si $n = N$ et n est pair.

– Si $n > N$, les nœuds hors des groupes (A) et (B) forment le groupe (C') dont la taille est $|(C')| = \frac{n-\sqrt{n}}{2} + N - n$, si n est impair. Ou $|(C')| = \frac{n}{2} - 1 + N$, si n est pair.

4.3.1.6 Le cas n est impair

Pour le groupe (A) :

– Le premier groupe de $g = 2\sqrt{n} - 3$ nœuds prennent le premier niveau $L1$. Le groupe des $g = g - 4$ nœuds suivants prennent le niveau suivant $L2$, et le groupe des $g = g - 4$ nœuds suivants prennent le niveau suivant et ainsi de suite en réduisant à chaque fois le groupe de nœuds de 4. La figure 4.8 montre un exemple de cette procédure.

– Le nœud d'indice dans la chaîne $i = \left(\frac{n-2\sqrt{n}+3}{2}\right)$ prend le premier indice * (appelé $INDEX_*(i)$).

– Le premier nœud qui prend le second indice # (appelé $INDEX_\#(i)$) est le nœud $i = \left(\frac{n-4\sqrt{n}+7}{2}\right)$, le deuxième nœud qui aura l'indice # est le nœud $y = \left(\frac{n-4\sqrt{n}+7}{2}\right) - k$, avec $k = 2\sqrt{n} - 6$, et le nœud suivant qui prend l'indice # est le nœud $y = y - k$, avec $k = k - 4$, et le nœud suivant qui prend cet indice est le nœud $y = y - k$, avec $k = k - 4$ et ainsi de suite en soustrayant à chaque fois 4 à partir de la dernière k et soustraire cette valeur de la dernière y.

– Le premier nœud qui prend le troisième indice d (appelé $INDEX_d(i)$) est le nœud $x = \left(\frac{n-6\sqrt{n}+11}{2}\right)$, le deuxième nœud qui aura l'indice d est le nœud $x = \left(\frac{n-6\sqrt{n}+11}{2}\right) - p$, avec $p = \left(\frac{4\sqrt{n}-2}{3}\right)$, et le nœud suivant qui prend l'indice d est le nœud $x = x - p$, avec $p = p - 4$,

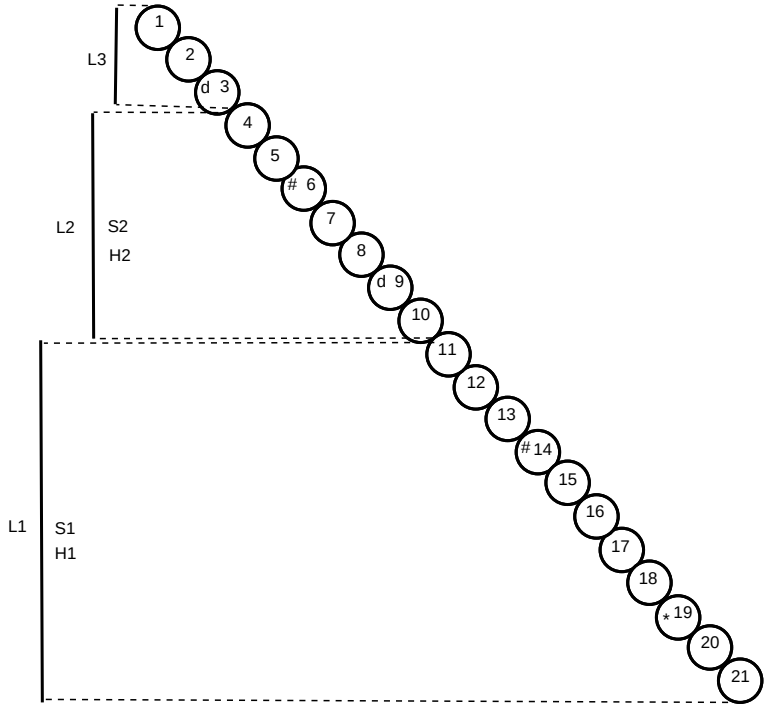

FIGURE 4.8 – Un exemple de partitionnement en niveaux des nœuds ayant l'état *top* et les nombres associés aux niveaux avec un exemple de $n = 49$

et le nœud suivant qui prend l'indice d est le nœud $x = x - p$, avec $p = p - 4$, et ainsi de suite en soustrayant à chaque fois 4 à partir de la dernière valeur k et en soustrayant cette valeur de la dernière valeur x.

– Pour chaque niveau j (sauf le dernier niveau) est associé deux nombres H_j et S_j.

$$H_j = \begin{cases} \dfrac{3\sqrt{n} - 17}{2}, si\, j = 1. \\ H_{j-1} - 3, sinon. \end{cases} \tag{4.1}$$

$$S_j = \begin{cases} \dfrac{\sqrt{n} - 3}{2}, si\, j = 1. \\ S_{j-1} - 1, sinon. \end{cases} \tag{4.2}$$

$$U_{j,i} = \begin{cases} U_{j-1,i+1} - S_{j-1}, si\, L(i+1) \neq j. \\ U_{j,i+1} - H_{j-1}, si\, INDEX_d(i+1). \\ \sqrt{n} - 1,\ si\, i = \left(\dfrac{n - \sqrt{n}}{2}\right). \\ U_{j,i+1} + 1,\ si\, i + 1 = \left(\dfrac{n - \sqrt{n}}{2}\right). \\ U_{j,i+1} - \left(\dfrac{3\sqrt{n} - 11}{2}\right), si\, INDEX_*(i+1). \\ U_{j,i+1} + 1, si\, INDEX_\#(i+1). \\ U_{j,i+1} + 2, sinon. \end{cases} \tag{4.3}$$

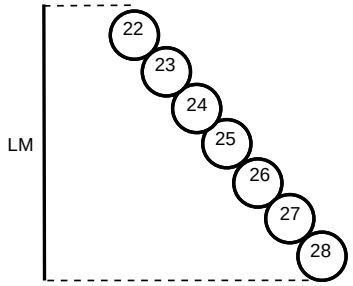

FIGURE 4.9 – Un exemple de partitionnement en niveaux pour $n = 49$. Les nœuds du milieu ont un niveau spécial LM, ces nœuds ne se déplacent pas, leur taille est de $\sqrt{n}$ nœuds

Avec $U_{j,i}$ est le nombre de mouvements du nœud i de niveau j.

Pour le groupe (B) :

– Les $\sqrt{n}$ nœuds après le nœud $i = \frac{n-\sqrt{n}}{2}$ sont associés au niveau du milieu appelé LM. Les nœuds de ce niveau ne bougent pas, leur nombre de mouvements est donc 0 (voir la figure 4.9).

Pour le groupe (C) :

– Les 7 derniers nœuds prennent le niveau LS. Les premiers $\left(\frac{\sqrt{n}+1}{2}\right)$ nœuds prennent le premier niveau $L1$. Les $a = 2\sqrt{n} - 4$ nœuds suivants prennent le niveau $L2$, et les $a = a - 4$ suivants prennent le niveau suivant $L3$ et ainsi de suite. La figure 4.10 montre un exemple.

– Le premier nœud qui prend l'indice \$ est le nœud $c = \left(\frac{5\sqrt{n}-5}{2}\right)$, le nœud suivant qui prend l'indice \$ est le nœud $c = c + p$, avec $p = (2\sqrt{n} - 7)$ et le prochain est le nœud $c = c + (p - 4)$ et le prochain est le nœud $c = c + (p - 8)$ et ansi de suite, la figure 4.10 présente une exemple.

$$\lambda_j = \begin{cases} \dfrac{\sqrt{n}-5}{2}, sij = 2. \\ \lambda_j = \lambda_{j-1} - 1, sinon. \end{cases} \tag{4.4}$$

$$\phi_j = \begin{cases} \dfrac{\sqrt{n}+5}{2}, sij = 2. \\ \phi_j = \phi_{j-1} - 3, sinon. \end{cases} \tag{4.5}$$

$$Z_{j,i} = \begin{cases} \sqrt{n}, siLM(i-1). \\ Z_{j,i-1} - \phi_j, siL(i-1) \neq Li. \\ Z_{j,i-1} + \lambda_j, siINDEX_D(i). \\ Z_{j,i-1} + 1, si\, i - 1 = n - 1. \\ Z_{j,i-1} + 2, sinon. \end{cases} \tag{4.6}$$

Avec $Z_{i,j}$ est le nombre de mouvements du nœud i de niveau j.

Pour le groupe (C') :

On partitionne le groupe (C') à deux sous-groupes $(C')_A$, $(C')_C$. Le groupe $(C')_A$ contient les premiers $N - n$ nœuds. Les $\frac{n-\sqrt{n}}{2}$ nœuds suivants sont dans le groupe $(C')_C$. Dans ce qui suit : $(C')_A(i)$ signifie que le nœud i appartient au groupe $(C')_A$.

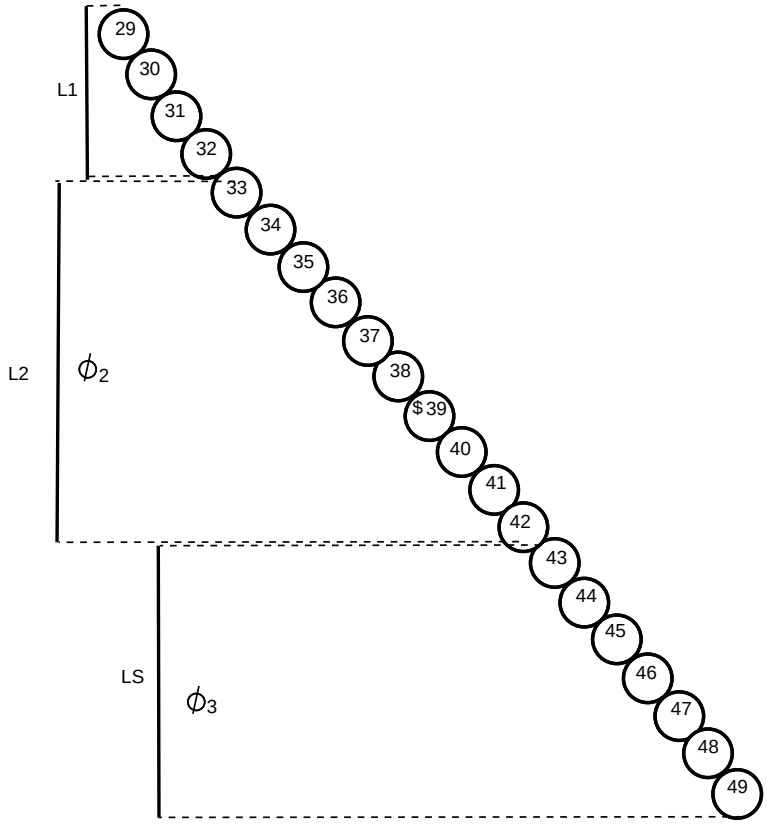

FIGURE 4.10 – Un exemple de partitionnement en niveaux des nœuds ayant l'état *bottom* et les nombres associés aux niveaux pour $n = 49$

Pour le groupe $(C')_A$:

$$\nu_i = \begin{cases} 2\sqrt{n} - (N-n), si LM(i-1). \\ \nu_{i-1} + 2, sinon. \end{cases} \tag{4.7}$$

Avec ν_i est le nombre de mouvements du nœud i.

Pour le groupe $(C')_C$:

Le nombre de mouvements d'un nœud est prédit avec les mêmes instructions (nombre, niveaux, et indices) du groupe C, avec cette nouvelle fonction $Z_{j,i}$.

$$Z_{j,i} = \begin{cases} \sqrt{n} + (N-n), si\, (C')_A(i-1). \\ Z_{j,i-1} - \phi_j, si\, L(i-1) \neq Li. \\ Z_{j,i-1} + \lambda_j, si\, INDEX_D(i). \\ Z_{j,i-1} + 1, si\, i-1 = n-1. \\ Z_{j,i-1} + 2, sinon. \end{cases} \tag{4.8}$$

4.3.1.7 Le cas n est pair

Pour le groupe (A) :

– Les premiers $\sqrt{n} - 1$ nœuds prennent le niveau racine (L0).

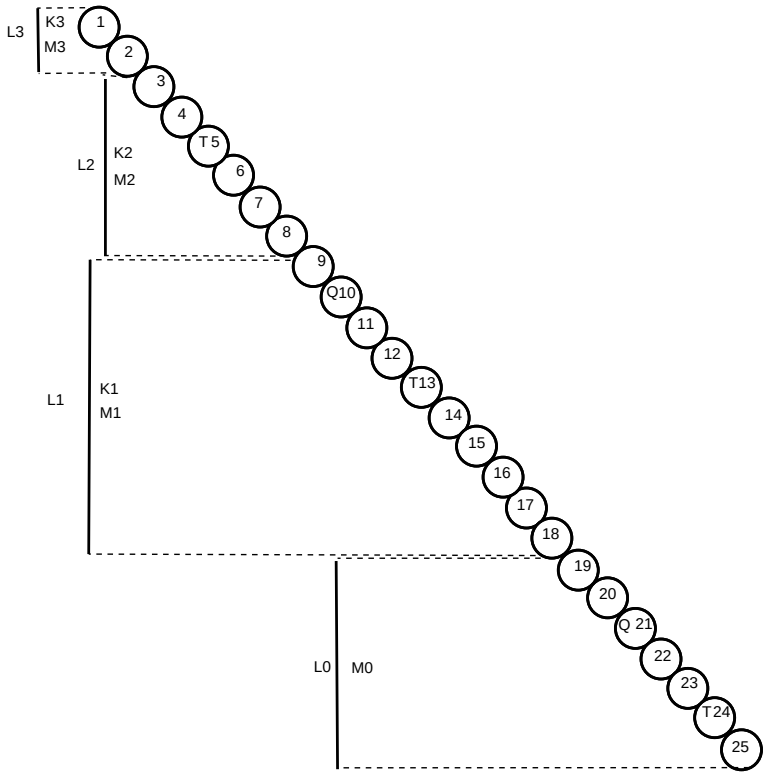

FIGURE 4.11 – Un exemple de partitionnement de 64 nœuds en niveaux du groupe (A) ainsi que les nombres associés aux niveaux

– Les $a = 2\sqrt{n} - 6$ nœuds suivants prennent le premier (L1).

– Les $a = a - 4$ nœuds prennent le niveau suivant (L3). Et les $a = a - 4$ suivants prennent le niveau suivant et ainsi de suite.

– Le nœud d'indice dans la chaîne $c1 = \frac{n}{2} - \frac{3\sqrt{n}}{2} + 1$ est le premier à prendre l'indice Q, il est noté $INDEX_Q(c1)$). Puis le nœud d'indice dans la chaîne $c2 = c1 - p1$ ($p1 = (2\sqrt{n} - 5)$) prend l'indice Q. La procédure est réitérée avec à chaque itération le nœud $c_{t+1} = c_t - p_t$ qui prend l'indice Q ($p_t = p_{t-1} - 4$).

– Le premier nœud qui prend l'indice T ($INDEX_T(i)$) est le nœud $b1 = \frac{n}{2} - \sqrt{n}$ puis le nœud $b2 = b1 - e1$ avec $e = (2\sqrt{n} - 5)$. De façon récursive, à chaque itération le le nœud le nœud $b_{t+1} = b_t - e_t$ avec $e_t = e_{t-1} - 4$ prend l'indice T (voir figure 4.11).

$$K_j = \begin{cases} \dfrac{3\sqrt{n}}{2} - 7, if j = 1. \\ K_j = K_{j-1} - 3, sinon. \end{cases} \tag{4.9}$$

$$M_j = \begin{cases} \dfrac{\sqrt{n}}{2} - 1, if j = 0. \\ M_j = M_{j-1} - 1, sinon. \end{cases} \tag{4.10}$$

$$\chi_{j,i} = \begin{cases} \sqrt{n} - 1, if\ i = \dfrac{n}{2} - \sqrt{n} + 1. \\ \chi_{j,i+1} - K_j, if\ L(i+1) \neq j. \\ \chi_{j,i+1} - M_j, if\ INDEX_Q(i). \\ \chi_{j,i+1} + 1, if\ INDEX_T(i). \\ \chi_{j,i+1} + 2, sinon. \end{cases} \tag{4.11}$$

Avec $\chi_{j,i}$ est le nombre de mouvements du nœud i de niveau j.

Pour le groupe (B)

– Les $\sqrt{n}$ nœuds après le nœud $i = \frac{n}{2} - \sqrt{n} + 1$ font partie du niveau du milieu LM. Ces nœuds ne se déplacent pas est ont par conséquent un nombre de mouvements égal à 0 (voir la figure 4.9).

Pour le groupe (C) :

– Les 7 derniers nœuds prennent un niveau spécial LS. Les premiers $w = 2\sqrt{n} - 2$ nœuds prennent le premier niveau $L1$, et les $w = w - 4$ nœuds suivants prennent le niveau suivant $L2$ et ainsi de suite. Pour chaque niveau il est associé un nombre B_j et un nombre O_j.

– Le premier nœud qui prend l'indice
H (appelé $INDEX_H(i)$) est le nœud $c = n - 10$, et le nœud suivant qui prend cet indice est $c = c - p$ avec $p = 11$, et le nœud suivant qui prend l'indice H est le nœud $c = c - p$ avec $p = p + 4$ et ainsi de suite en ajoutant à chaque fois 4 à p, (voir la figure 4.12).

$$B_j = \begin{cases} \dfrac{\sqrt{n}}{2} - 2, si\ j = 1. \\ B_j = B_{j+1} - 1, sinon. \end{cases} \tag{4.12}$$

$$O_j = \begin{cases} \dfrac{3\sqrt{n}}{2} - 5, si\ j = 1. \\ O_j = B_{j+1} - 3, sinon. \end{cases} \tag{4.13}$$

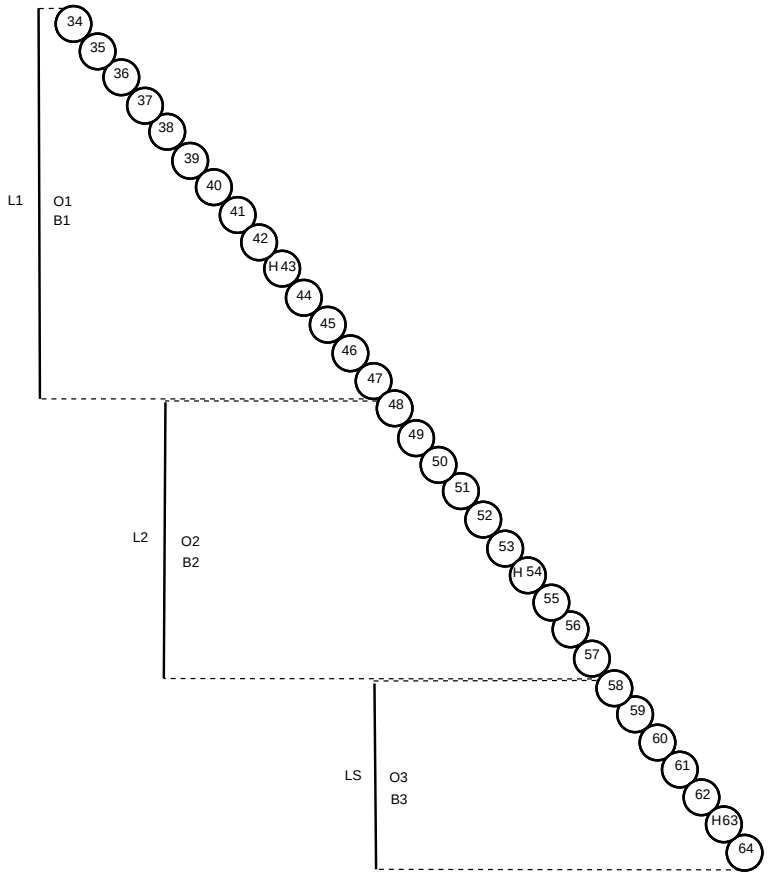

FIGURE 4.12 – Un exemple de partitionnement en niveaux de 64 nœuds pour le groupe (C). Les nombres associés aux niveaux sont donnés

$$\xi_{j,i} = \begin{cases} 2, LM(i-1). \\ \xi_{j,i-1} + 1, si\, i = n. \\ \xi_{j,i-1} - B_j, si\, INDEX_H(i-1). \\ \xi_{j,i-1} - O_j, si\, L(i-1) \neq j. \\ \xi_{j,i-1} + 2, sinon. \end{cases} \tag{4.14}$$

Avec : $\xi_{j,i}$ est le nombre de mouvements du nœud i de niveau j. Pour le groupe (C') :

On divise le groupe (C') en deux sous-groupes $(C')_A$, $(C')_C$. Le groupe $(C')_A$ contient les premiers $N - n$ nœuds. Les $n - \frac{n}{2} - 1$ nœuds suivants composent le groupe $(C')_C$.

Pour le groupe $(C')_A$:

$$\kappa_i = \begin{cases} 2\sqrt{n} + 1 - (N-n), si\, LM(i-1). \\ \kappa_{i-1} + 2, sinon. \end{cases} \tag{4.15}$$

Pour le groupe $(C')_C$:
le nombre de mouvements sont prévus avec les mêmes instructions (nombres, niveaux, et indices) du groupe (C), avec cette nouvelle fonction $\xi_{j,i}$:

$$\xi_{j,i} = \begin{cases} 2 + (N-n), si\, (C')_A(i-1). \\ \xi_{j,i-1} + 1, si\, i = n. \\ \xi_{j,i-1} - B_j, si\, INDEX_H(i-1). \\ \xi_{j,i-1} - O_j, si\, L(i-1) \neq j. \\ \xi_{j,i-1} + 2, sinon. \end{cases} \tag{4.16}$$

4.3.1.8 Généralisation de l'algorithme

L'algorithme PPCI présenté est spécifique à un cas de la chaîne où les nœuds forment d'abord une chaîne linéaire orientée vers les directions SE-NW. Dans cette section, nous décrivons comment l'algorithme peut être généralisé à tout type de chaîne initiale comme c'est montré dans la figure 4.13. Chaque nœud peut déduire l'orientation de la chaîne (l'un des trois cas représentés dans la figure 4.13) en analysant l'orientation de ses voisins. Par exemple, si un nœud correspond à un nœud d'extrémité (il a un seul voisin) dont le voisin direct est du côté (E), le nœud en déduit que la ligne droite est orientée E-W. De la même façon, les nœuds intermédiaires utilisent l'orientation de leurs deux voisins pour déterminer l'orientation de la chaîne formée. Généralement, chaque nœud après la détection de l'orientation de la chaîne, notée D-$\overline{D}$, exécute une variante de l'algorithme PPCI en fonction de l'orientation $D \in \{W, NW, NE\}$. La variante de l'algorithme PPCI, $PPCI^D$, représente une adaptation de l'algorithme original (correspondant à $PPCI^{NW}$) pour les deux autres orientations possibles. Cette adaptation prend la forme d'un changement des directions dans les prédicats. Par exemple, si la chaîne initiale est orientée NE-SW, l'algorithme $PPCI^{NE}$ est appelé, et la forme carrée est réalisée à l'aide de mouvements de

type $moveAroundbad_v(u, P_w)$, $moveAroundwell_v(u, P_w)$ et $moveAroundwell_v(u, P_{nw})$. L'utilisation de ces trois prédicats est décrite dans la figure 4.14 qui présente un exemple avec des nœuds dont l'état est *bottom*.

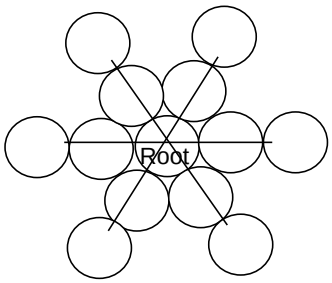

FIGURE 4.13 – Les trois cas possibles dune chaîne initiale

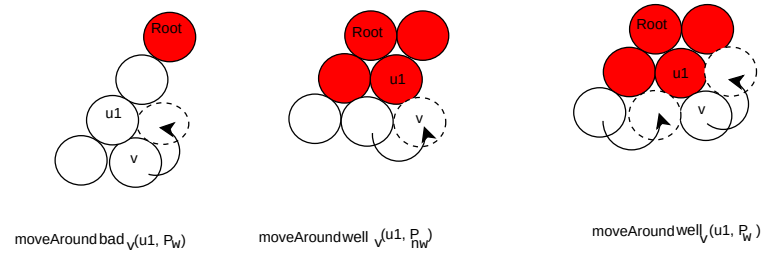

FIGURE 4.14 – L'adaptation des mouvements dans le cas d'une chaîne NE-SW. La figure présente le déplacement pour les nœuds ayant l'état *bottom*

4.4 Simulation et comparaison de PPCI

Pour évaluer les performances de l'algorithme, nous avons implémenté et simulé son exécution à l'aide du langage déclaratif MELD et le simulateur Dprsim (Dynamic Physical Rendering Simulator). Les simulations ont été effectuées avec un rayon de nœud égal à 1 mm sur ordinateur Intel(R) Core(Tm)i5, 2.53 Ghz avec 4 GO de mémoire. Les figures 4.15 et 4.16 représentent un exemple d'exécution de l'algorithme PPCI.

Nous désignons, dans les figures de la simulation, par $PPCI1$ les scénarios relatifs au cas de $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$ impair, et par $PPCI2$ les scénarios où n est pair (N est la taille du réseau). Les nœuds ont appliqué la procédure de partitionnement en niveaux et ont calculé le nombre de mouvements à effectuer. A la fin de l'algorithme, chaque nœud compare les résultats de la prédiction (avec les fonctions décrites dans ce chapitre) aux résultats de l'exécution. Les résultats des simulations confirment les analyses théoriques.

La figure 4.21 compare la vitesse de convergence des deux algorithmes distribués ACI et $PPCI$ sur la base du temps de construction des sous-carrés (les carrés intermédiaires générés avant la formation du carré final) sur un exemple de 1000 nœuds. La figure 4.17 représente le temps d'exécution en fonction du nombre de nœuds et compare les résultats de l'algorithme PPCI avec ceux de ACI. La figure 4.18 compare le nombre maximal de mouvements effectué

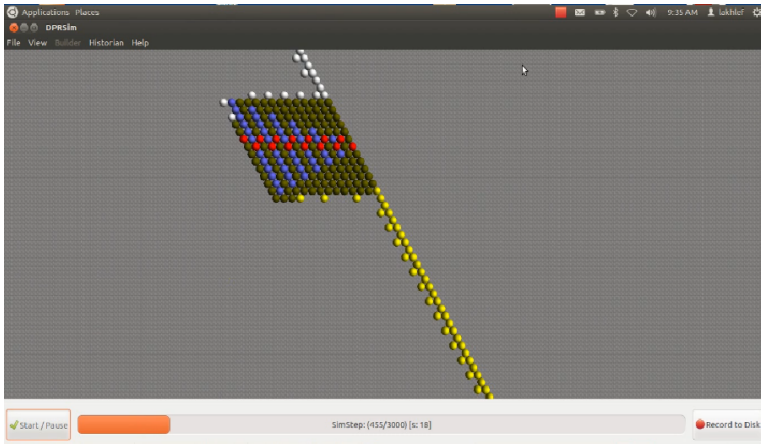

FIGURE 4.15 – Un exemple d'exécution de l'algorithme PPCI. Les couleurs des nœuds représentent leur état : blanc pour *top*, jaune pour *bottom*, bleu pour *well*, vert pour *int* et rouge pour *nper*. L'exemple n'indique pas tout les nœuds *well*, autrement, tous les nœuds seraient bleus

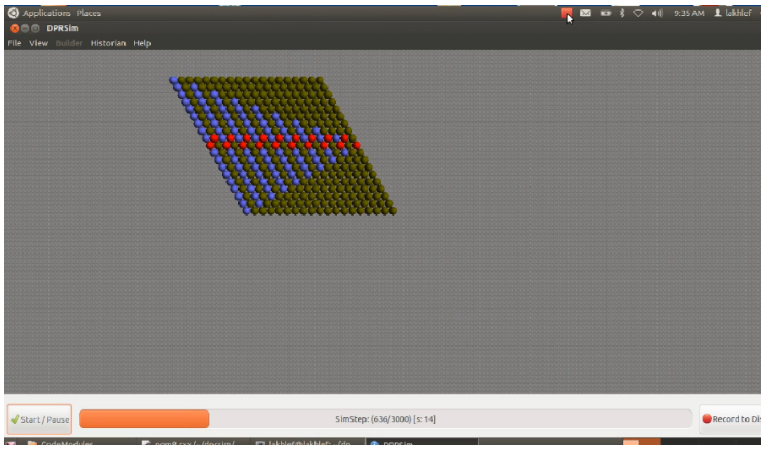

FIGURE 4.16 – Un exemple de l'exécution finale de PPCI

par un nœud pour les deux algorithmes PPCI et ACI. Avec, $g(N) = (\frac{\sqrt{n}}{2}) + N - (\frac{n}{2}) - 1$ et $f(N) = (\frac{n+1}{2}) + N - n$, où $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$.

La figure 4.19 représente le nombre total de mouvements dans les réseaux qui correspond à

$$(\sum Z_{i,j}) + (\sum U_{i,j}) \tag{4.17}$$

ou à

$$(\sum \chi_{j,i}) + (\sum \xi_{j,i}) \tag{4.18}$$

La figure 4.20 représente la moyenne du nombre total de mouvements qui correspond à :

$$\frac{(\sum Z_{i,j}) + (\sum U_{i,j})}{n} \tag{4.19}$$

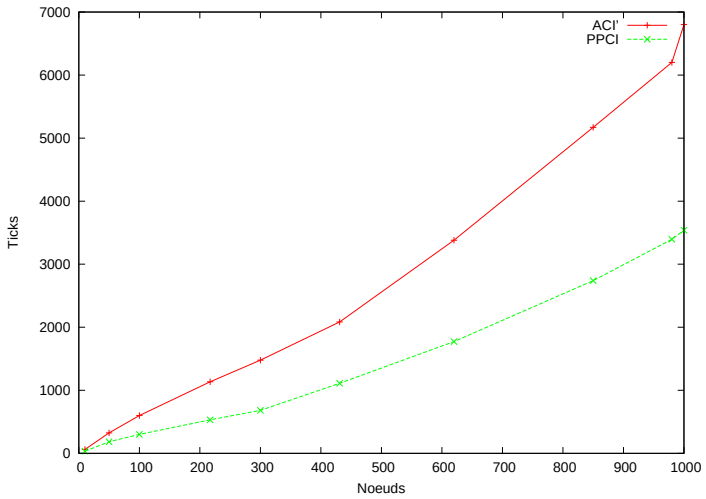

FIGURE 4.17 – Temps d'exécution

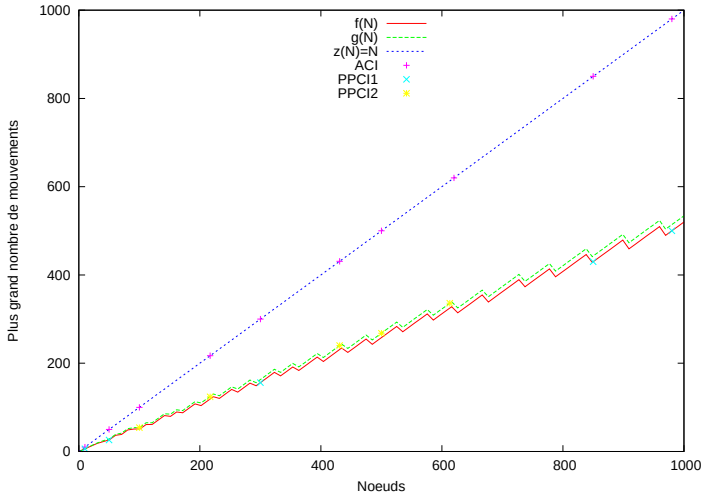

FIGURE 4.18 – Plus grand nombre de mouvements

ou à

$$\frac{(\sum \chi_{j,i}) + (\sum \xi_{j,i})}{n} \tag{4.20}$$

Les effets du parallélisme en mouvement apparaissent bien dans la courbe représentant le temps d'exécution, puisque PPCI construit deux rectangles en même temps. Nous observons aussi que le gain en temps obtenu avec l'algorithme PPCI augmente avec la taille du réseau.

On remarque dans la figure 4.18 que le nombre de mouvements de PPCI est beaucoup plus faible, ce qui permet d'augmenter la durée de vie des nœuds et par conséquent la probabilité que le nœud continue sa tâche (ses mouvements) sans panne. Le parallélisme a amélioré le nombre total de mouvements dans le réseau et par là la moyenne du nombre total de mouvements dans le réseau. Cependant, PPCI utilise dix états par nœud et les algorithmes ACI et ACNI utilisent seulement trois états par nœud.

On remarque dans la figure 4.21 que les sous-carrés sont obtenus tôt par rapport à l'autre protocole. Ceci s'explique par le fait que dans PPCI la construction de forme est en parallèle,

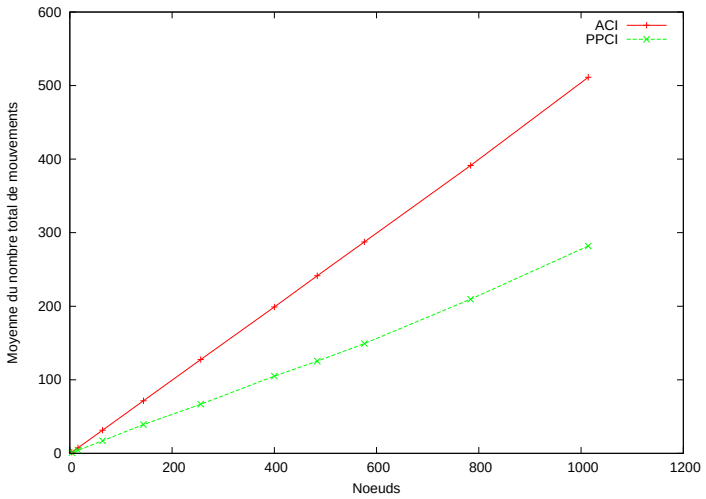

FIGURE 4.19 – La moyenne du nombre total de mouvements dans le réseau

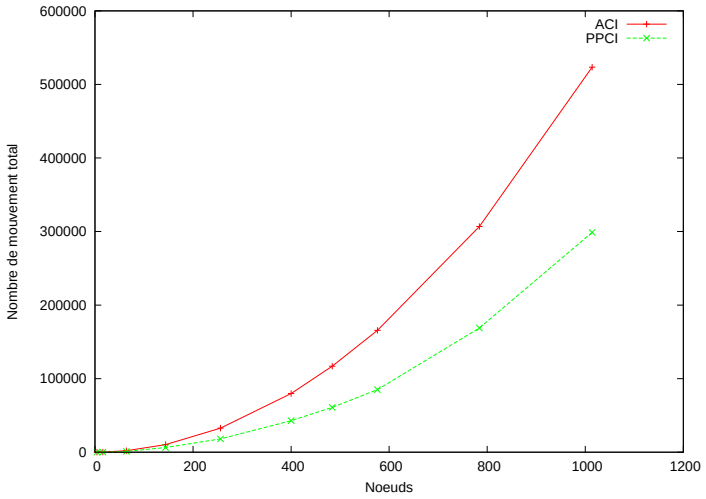

FIGURE 4.20 – Le nombre total de mouvements dans le réseau

de sorte que l'algorithme prend moins de temps à construire chaque sous-carré.

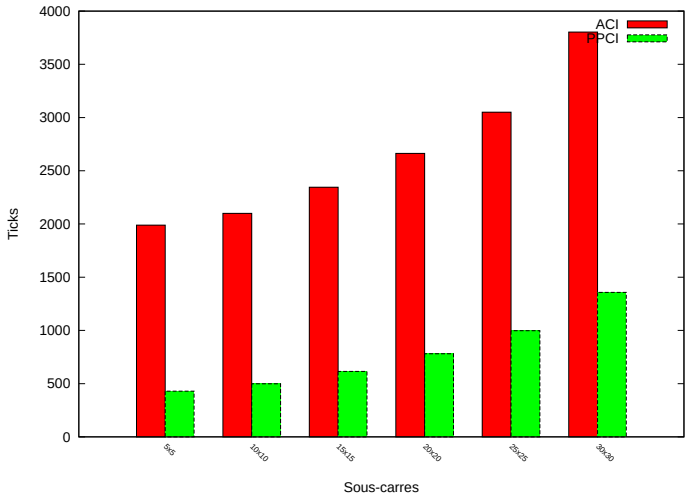

FIGURE 4.21 – Sous-carrés générés en fonction du temps pour une simulation de 1000 nœuds

4.4.1 Protocole parallèle avec une connexité non instantanée (PPCNI)

Dans l'algorithme PPCNI [55, 56] (présenté ci-après), le nœud peut se déplacer autour de son voisin physique ou glisser d'une position en ligne directe (d'une distance égale au diamètre du nœud). Ces deux types de mouvements sont effectués avec l'aide des voisins pour déterminer le sens et la nouvelle position. Le nœud qui n'a pas de voisin ne peut pas bouger, car il ne sait pas où est-il et où sont les autres nœuds.

L'optimalité en terme de nombre de mouvements dépend du choix du nœud du milieu. Pour appliquer un parallélisme optimal, le nœud du milieu mi qui est aussi l'initiateur de l'algorithme doit être trouvé comme suit :
Soit N la taille du réseau (nombre de microrobots), $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$ et $dif = N - n$ alors :

– Si n est impair et $dif < 2\sqrt{n}$ alors le nœud du milieu est :

$$mi = (n+1)/2 \tag{4.21}$$

– Si n est impair et $dif \geq 2\sqrt{n}$ alors mi est :

$$mi = ((n+1)/2) + \sqrt{n} - 2 \tag{4.22}$$

– Si n est pair et $dif < 2\sqrt{n}$ alors le nœud du milieu est :

$$mi = n/2 - ((\sqrt{n}/2) - 1) \tag{4.23}$$

– Si n est pair et $dif \geq 2\sqrt{n}$ alors le nœud du milieu est :

$$mi = (n/2 - ((\sqrt{n}/2) - 1)) + \sqrt{n} - 2 \tag{4.24}$$

Le nœud du milieu mi peut être trouvé en connaissant la taille du réseau. Un nœud de l'extrémité de la chaîne initialise un compteur et le diffuse, chaque nœud qui reçoit ce message incrémente le compteur jusqu'à ce qu'il arrive au nœud concerné mi, qui aura le prédicat satisfait $medChain(v)$.**Variables et prédicats**

– initiator$_v()$: le nœud v qui initialise l'algorithme.
– state$_v(X)$: v prend l'état $X \in \{well, bad, int, nper, sint, rint, mnper, top, bottom\}$, v ne peut pas prendre les états *well* et *bad* en même temps.
– moveAround$state_v(u, P_x)$: v se déplace autour du voisin u a l'état *state* de telle sorte que u devienne le voisin dans la direction x de v.
– moveTo$_v(\mathrm{P}_{N_x})$: v se déplace vers la précédente position de l'ancien voisin dans la direction x de v.

Prédicats vérifiés seulement dans la première étape

1. $\text{initiator}_v() \equiv medChain(v)$.
2. $\text{state}_v(bad) \equiv connected_v() \wedge \neg initiator_v()$.
3. $\text{state}_v(well) \equiv initiator_v()$.
4. $\text{state}_v(nper) \equiv initiator_v()$.

Prédicats vérifiés à chaque étape

5. thisRound$\equiv GetCurrentRound()$.
6. $\text{hasN}_{nw}v(thisRound) \equiv (N_{nw}(v) = u) \wedge state_u(bad)$.
7. $\text{N}_{nw}\text{lastRound}_v(LastRound) \equiv hasN_{nw}v(thisRound - 1)$.
8. $\text{hasN}_{se}v(thisRound) \equiv (N_{se}(v) = u) \wedge state_u(bad)$.
9. $\text{N}_{se}\text{lastRound}_v(LastRound) \equiv hasN_{se}v(thisRound - 1)$.
10. $\text{state}_v(top) \equiv (N_{se}(v) = u, initiator_u()) \vee (N_{se}(v) = u, state_u(top))$.
11. $\text{state}_v(bottom) \equiv (N_{nw}(v) = u, initiator_u()) \vee (N_{nw}(v) = u, state_u(top))$.
12. $\text{state}_v(sint) \equiv (N_e(v) = u, initiator_u())$.
13. $\text{state}_v(nper) \equiv (N_{se}(v) = u, state_u(sint))$.
14. $\text{state}_v(rint) \equiv (N_e(v) = u, state_u(well)) \wedge (N_{ne}(v) = u, \neg state_u(well), \neg state_u(int))$.
15. $\text{state}_v(mnper) \equiv (N_e(v) = u, state_u(nper)) \wedge (\neg state_v(nper)) \wedge (state_v(int) \vee state_v(well))$.
16. $\text{state}_v(nper) \equiv (N_e(v) = u, state_u(mnper)) \wedge (\neg state_v(mnper)) \wedge (N_{se}(v))$.
17. $\text{state}_v(int) \equiv ((N_e(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{se}(v) = u, state_u(well)) \wedge (N_w(v))) \vee ((N_{se}(v) = u, state_u(rint))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_w(v)))$.
18. $\text{state}_v(int) \equiv ((N_e(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{se}(v) = u, state_u(well)) \wedge (N_w(v))) \vee (N_e(v) = u1, N_{se}(v) = u2, state_{u1}(int), state_{u2}(int)) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_{nw}(v))) \vee ((N_{ne}(v) = u, state_u(well)) \wedge (N_w(v)))$.
19. $\text{state}_v(well) \equiv ((N_e(v) = u, state_u(int)) \wedge (N_{nw}(v))) \vee ((N_e(v) = u, state_u(int)) \wedge (N_{se}(v)))$.
20. $\text{state}_v(well) \equiv (N_w(v) = u, state_u(well)))$.
21. $\text{state}_v(well) \equiv state_v(bad) \wedge (\neg N_{se}(v)) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well))$.
22. $\text{state}_v(well) \equiv state_v(bad) \wedge (\neg N_{se}(v)) \wedge (N_w) \wedge (N_{nw}(v) = u, state_u(well)) \wedge ((N_e(v) = u1, state_{u1}(well)))$.
23. $\text{moveTo}_v(P_{N_{nw}}) \equiv state_v(top) \wedge state_v(bad) \wedge N_{nw}lastRound_v(LastRound) \wedge \neg hasN_{nw_v}(thisRound)$.

24. $\text{moveTo}_v(P_{N_{se}}) \equiv state_v(bottom) \wedge state_v(top) \wedge state_v(bad) \wedge N_{se}lastRound_v(LastRound) \wedge \neg hasN_{se_v}(thisRound)$.

25. $\text{moveAround}int_v(u, P_{se}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{sw}(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper)) \wedge (((N_{se}(v) = u1, \neg N_e(u1) = u) \wedge N_{nw}(v)) \vee (N_{se}(v) = u1, N_e(u1) = u))$.

26. $\text{moveAround}well_v(u, P_{se}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v)) \wedge state_v(top) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{sw}(v) = u, state_u(well), state_u(top))$.

27. $\text{moveAround}int_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v) \wedge \neg N_e(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{se}(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper))$.

28. $\text{moveAround}well_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{nw}(v) \wedge \neg N_e(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{se}(v) = u, state_u(well), state_u(top))$.

29. $\text{moveAround}well_v(u, P_{ne}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{sw}(v) \wedge \neg N_{se}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(well), state_u(top)) \wedge (\neg state_u(nper))$.

30. $\text{moveAround}int_v(u, P_{ne}) \equiv (\neg N_w(v) \wedge state_v(top) \wedge \neg N_{sw}(v) \wedge \neg N_{se}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(int), state_u(top)) \wedge (\neg state_u(nper))$.

31. $\text{moveAround}well_v(u, P_{ne}) \equiv (\neg N_e(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{nw}(v) = u, state_u(well), state_u(bottom))$.

32. $\text{moveAround}well_v(u, P_{se}) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_e(v) = u, state_u(well), state_u(bottom), (\neg state_u(nper))$.

33. $\text{moveAround}well_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{ne}(v) = u, state_u(well), (\neg state_u(nper))$.

34. $\text{moveAround}int_v(u, P_{ne}) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{nw}(v) = u, state_u(int), state_u(bottom)) \wedge (N_{ne}(v) \vee N_{se}(v))$.

35. $\text{moveAround}int_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge (N_{ne}(v) = u, state_u(int), state_u(bottom), (\neg state_u(nper))$.

36. $\text{moveAround}well_v(u, P_e) \equiv (\neg N_w(v) \wedge state_v(bottom) \wedge \neg N_{sw}(v)) \wedge (\neg state_v(well)) \wedge (\neg state_v(int)) \wedge state_v(bad) \wedge state_v(bottom) \wedge (N_{ne}(v) = u, state_u(well), state_u(mnper), (\neg state_u(nper))$.

4.4.1.1 Description et analyse

L'algorithme PPCNI fonctionne par étapes, à chaque étape les prédicats satisfaits sont choisis pour l'exécution. L'algorithme distribué cherche la forme désirée en utilisant un processus incrémental. Dans un incrément terminé, les nœuds qui le construisent appartiennent déjà à la forme ; ces nœuds vont aider leurs voisins ou futurs voisins à se positionner correctement.
Le nœud du milieu (*mi*) de la chaîne se déclare comme un initiateur avec le prédicat (1). Initialement, tous les nœuds sont initialisés avec l'état *bad* à l'exception de l'initiateur (2). L'initiateur prend les états *well* et *nper* avec les prédicats (3) et (4). Les nœuds qui sont au-dessus de l'initiateur prennent l'état *top* avec le prédicat (10). Les autres nœuds qui sont sous l'initiateur prennent l'état *bottom* avec le prédicat (11). Les nœuds ayant l'état *well* ou *int* sont des nœuds déjà dans la forme cible, il ne peuvent pas bouger, ils deviennent fixes.

Pour faire un parallélisme optimal et un carré correct, le nombre de nœuds ayant l'état *top* (10) finissant sur la même ligne horizontale que l'initiateur (dans la direction E par rapport à l'initiateur) doit être égal au nombre de nœuds possédant l'état *bottom* (11) qui finissent sur la même ligne que l'initiateur si N est impair. Si N est pair, un autre nœud est ajouté aux nœuds ayant l'état *top*. L'état *nper* est utilisé pour réaliser cet objectif. C'est-à-dire, le nœud ayant l'état *nper* ne permet pas à certains nœuds de se déplacer autour de lui. L'initiateur prend l'état *nper* (4). L'état *nper* empêche les voisins du nœud ayant l'état *bottom* de rejoindre la ligne de l'initiateur (se positionner à l'Est de l'initiateur). Cela se fait avec la garde ($\neg state_v(nper)$). L'état *mnper* est un état intermédiaire utilisé pour propager l'état *nper* vers les autres nœuds concernés, pour assurer un parallélisme optimal. L'état *sint* (12) est un état intermédiaire utilisé comme référence pour le premier nœud qui peut obtenir l'état *nper*. L'état *sint* est un état indispensable parce que le deuxième nœud à prendre l'état *nper* (13) n'est pas un nœud voisin de l'initiateur (qui est le premier nœud *nper*).

Le nœud avec un voisin dans la direction E ayant l'état *nper* prend l'état *mnper* (15). Le nœud qui a l'initiateur comme nœud voisin dans la direction E prend l'état *sint* (12). Les autres (prochains) nœuds qui auront l'état *nper* sont les nœuds ayant un voisin (E) avec l'état *mnper* (16), les nœud ayant l'état *bottom* vérifient si le nœud voisin a l'état *nper*. Par conséquent, un nœud ayant l'état *nper* empêche les nœuds voisins avec l'état *bottom* de rejoindre la ligne de l'initiateur car ces nœuds vérifient les prédicats (30), (32), (35) et (36). L'état *int* est un état intermédiaire qui permet d'ajouter une couche non-complète à la forme carrée. Ainsi, les nœuds qui ont des voisins ayant l'état *well* prennent l'état *int* avec le prédicat (17). Le nœud sur la ligne de l'initiateur change son état à *int*. Puis l'état *int* se propage aux nœuds voisins à l'état *well*. On note que les nœuds prennent l'état *int* s'ils ont au moins un voisin ayant l'état *well*. Mais en faisant une nouvelle couche, il y a un nœud qui n'aura pas un voisin dont l'état est *well*, et il doit prendre l'état *int*, l'état *rint* (14) est utilisé pour traiter ce cas. Notez aussi, les nœuds ayant l'état *well* et les nœuds ayant l'état *int* ensemble ne forment pas un carré. Le carré est complété quand tous les nœuds à l'état *int* ont un voisin (W) à l'état *well*. La vague de changement d'état à *well* commence avec les prédicats (19), (20), (21) et (22).

Avec le prédicat (23) les nœuds ayant les états *top* et *bad* descendent vers le centre de la chaîne. Aussi, les nœuds ayant les états *bottom* et *bad* montent vers le centre de la chaîne avec le prédicat (24). Avec le prédicat (23), le nœud à l'état *top* et *bad* prend la position de son ancien voisin dans la direction NW. Avec le prédicat (23) le nœud qui est à l'état *bottom* et *bad* prend

la position de son ancien voisin dans la direction SE (24). Avec les prédicats (6), (7), (8) et (9), le nœud vérifie s'il avait un voisin dans la direction SE ou NW à l'étape précédente.
Avec le prédicat (25)/(26), un nœud avec les états *top* et *bad* se déplace autour du voisin ayant les états *top* et *int*/*well*. Ce voisin devient un voisin SE du nœud en mouvement. Avec le prédicat (27)/(28), un nœud avec les états *top* et *bad* se déplace autour du voisin ayant les états *top* et *int*/*well*. Le nœud voisin devient un voisin E du nœud en mouvement. Avec le prédicat (29)/(30), un nœud avec les états *top* et *bad* se déplace autour du voisin ayant les états *bottom* et *well*/*int*. Le nœud voisin devient un voisin NE du nœud en mouvement.

Avec le prédicat (31)/(32), un nœud avec les états *bottom* et *bad* se déplace autour du voisin à l'état *bottom* et *well*/*int*. Le nœud voisin devient un voisin NE/SE du nœud en mouvement. Avec le prédicat (33)/(34), un nœud v avec les états *bottom* et *bad* se déplace autour du voisin ayant les états *bottom* et *well*/*int*. Le nœud voisin devient un voisin E/NE du nœud en mouvement. Le prédicat (35)/(36) permet à un nœud avec les états *bottom* et *bad* de se déplace autour d'un voisin ayant les états *bottom* et *int*/*well*. Le nœud voisin devient alors un voisin E du nœud en mouvement.

Théorème 5 *Soit N la taille du réseau, $n = \left\lfloor \sqrt{N} \right\rfloor \left\lfloor \sqrt{N} \right\rfloor$ et $\eta = \left\lceil \sqrt{N} \right\rceil \left\lceil \sqrt{N} \right\rceil$, alors :*

- *Si n est impair et $n = N$ alors le plus grande nombre de mouvements est $((n+\sqrt{n}-2)/2)$.*
- *Si n est impair et $n < N$ alors le plus grande nombre de mouvements est $(\eta/2-1)$.*
- *Si n est pair et $n = N$ alors le plus grande nombre de mouvements est $(n/2-1)$.*
- *Si n est pair et $n < N$ alors le plus grande nombre de mouvements est $((\eta+\sqrt{\eta}-2)/2)$.*

Exemple :
La figure 4.22 montre un exemple d'exécution de l'algorithme avec explication et la figure5.4 montre un exemple sans explication. Dans la figure 4.22 :

– à $t0$: avec le prédicat (2) chaque nœud prend l'état b (*bad*), avec le prédicat (10) les nœuds (nœud 1) au-dessus de l'initiateur (nœud 2) prennent l'état t (*top*). Avec le prédicat (11), les nœuds (nœud 3 et 4) au sud de l'initiateur prennent l'état B (*bottom*). Avec le prédicat (3), l'initiateur prend l'état w (*well*), et avec le prédicat (4) il prend l'état n (*nper*).

– Pour arriver à l'étape suivante $t1$: Le nœud 1 se déplace autour du nœud 2 en utilisant le prédicat (28), et le nœud 3 se déplace autour le nœud 2 en utilisant (31). Le nœud 1 prend l'état s (*sint*)
avec le prédicat (12). Le nœud 1 ne peut pas faire un autre mouvement autour du nœud 2 puisque le nœud 2 a l'état n.

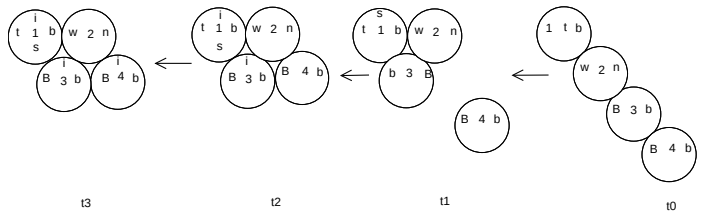

FIGURE 4.22 – Un exemple d'exécution de PPCNI avec quatre nœuds

– Pour arriver à l'étape suivante $t2$: le nœud prend la position de son ancien voisin (nœud 3) en utilisant le prédicat *(23)*.

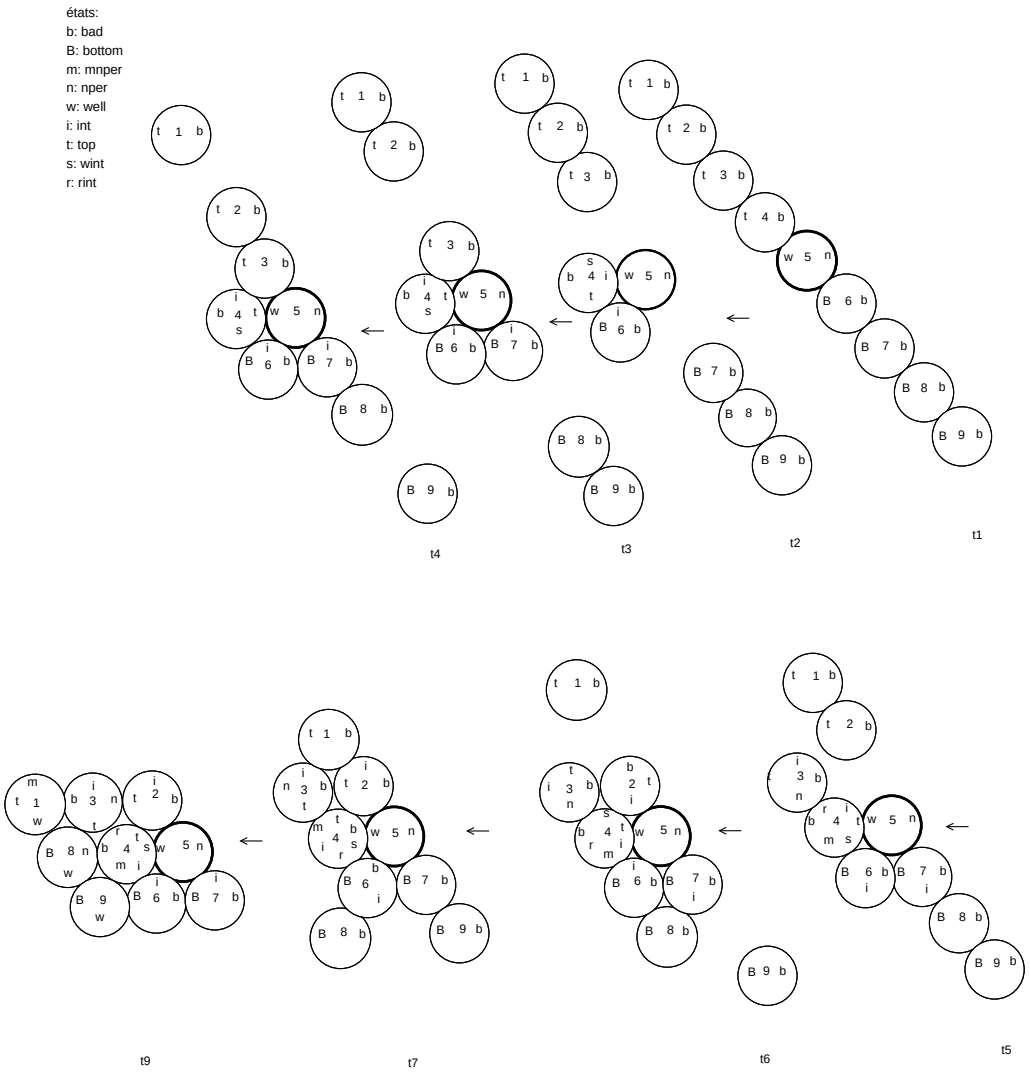

FIGURE 4.23 – Un exemple d'exécution de PPCNI avec neuf nœuds. L'initiateur est le nœud 5

Le nœud 1 et le nœud 3 prennent l'état i (*int*) avec les prédicats (17) et (18).

– à $t3$: la forme cible est obtenue. Le nœud 4 prend l'état i avec le prédicat (17).

4.4.1.2 Les onze états minimum

Dans cette section, nous montrons que onze états sont nécessaires et suffisants pour assurer la convergence de l'algorithme. Évidemment, avec un seul état, les nœuds ne peuvent pas distinguer s'ils sont dans des bonnes positions ou non et donc si le nœud doit se déplacer ou pas. Supposons une variante de PPCNI avec seulement deux états *bad* et *well*. Avec ces deux états, un nœud à l'état *well* est stable (ne bouge pas) car il appartient à la forme cible et un nœud à l'état *bad* se déplace autour des nœuds stables. Avec ces deux états, les nœuds collaborent pour construire à chaque itération une nouvelle couche dont les nœuds passent de l'état de *bad* à *well*. Soit l'ensemble S de nœuds ayant l'état *well*. En fonction de certaines conditions C l'ensemble B des nœuds voisins de S ayant l'état *bad* vont changer d'état à *well* afin de fixer la nouvelle couche.

Cependant, avec deux états les autres nœuds voisins de l'ensemble B vérifient aussi les conditions C dans les étapes suivantes. Ils vont alors changer d'états à *well* pour devenir stables provoquant le dysfonctionnement de l'algorithme. Deux états supplémentaires sont donc nécessaires : l'état *top* et *bottom*. Ces deux états sont indispensables pour éviter des boucles d'exécution dans PPCNI. En effet, le prédicat(23) permet aux nœuds de descendre aux milieu de la chaîne alors que le prédicat(24) permet aux nœuds de monter vers le milieu de la chaîne. Sans la présence des états *top* et *bottom*, les nœuds risquent d'enchaîner l'exécution des prédicats (23) puis(24), (24) puis (23), (29) puis (33) ou (33) puis (29). Les actions de ces prédicats s'annulent faisant que le nœuds oscillent autour des mêmes positions empêchant ainsi la convergence de l'algorithme. Une variante de PPCNI avec quatre états *bad* et *well* et *top* et *bottom* restent cependant insuffisante. Les raisons sont identiques à ceux utilisés pour prouver l'impossibilité avec deux états *well* et *bad*.

La solution consiste à ajouter un état intermédiaire *int* pour séparer les nœuds voisins ayant l'état *bad* et les nœuds ayant l'état *well*. Les conditions C pour passer de l'état *bad* à *well* sont modifiées pour intégrer le test sur l'état *int* (notons ces conditions par C'). Ainsi les voisins de B ne peuvent pas changer d'états à *well* car ils ne forment pas une nouvelle couche complète.
Supposons une variante de PPCNI avec six états *bad*, *well*, *top*, *bottom*, *int*, et $\neg$*int*. Les nœuds ayant l'état *int* forment une nouvelle couche adjacente au carré parfait intermédiaire de taille $\sqrt{Z} * \sqrt{Z}$. Le nombre de nœuds ayant l'état *int* est de $3\sqrt{Z} + 2$. Comme $\sqrt{Z} * \sqrt{Z} + 3\sqrt{Z} + 2$ n'est pas une racine carrée (de Théorème 1 et Théorème 2), la forme obtenue n'est pas un carré. Pour compléter le carré, il faut ajouter $\gamma = \sqrt{Z} + 2$ nœuds. Ces nœuds se positionnent sur le côté W des nœuds ayant l'état *int*. Ces γ nœuds peuvent prendre l'état *well* puisque la forme est un carré ou un carré intermédiaire.

Avec seulement six états *bad*, *well*, *top*, *bottom*, *int* et $\neg$*int*, la consommation d'énergie n'est pas bien répartie entre les nœuds. Pour mieux répartir la charge de mouvement, PPCNI construit deux rectangles en parallèle dont l'union forme un carré. En outre, pour propager l'état *nper* aux nœuds concernés du milieu horizontale du carré, on doit utiliser un autre état *mnper*. Les états $\neg$*nper* et$\neg$*mnper* sont utilisés pour vérifier si le nœud voisin a l'état *nper* et *mnper* respectivement. L'état *sint* est un état intermédiaire utilisé comme une référence au deuxième nœud qui prendra l'état *nper*. L'état *sint* est indispensable car le deuxième nœud à prendre l'état *nper* n'est pas voisin de l'initiateur qui est le premier à prendre l'état *nper*. D'abord le nœud dont le voisin est à l'état *nper* prend l'état *int*. Puis l'état *int* se propage aux nœuds voisins ayant l'état *well*. Notez que, les nœuds prennent l'état *int* s'ils ont au moins un voisin dont l'état est *well*. C'est pourquoi au début d'une nouvelle couche (le nœud initial qui n'a pas un voisin à l'état *well*) prend l'état *rint*.

4.4.1.3 Prédire le nombre de mouvements pour chaque nœud

Pour tenir compte de la consommation énergétique des nœuds, chaque nœud pré-calcule le nombre de mouvements qu'il devrait faire. Ainsi, chaque nœud sait la quantité d'énergie qu'il va consommer. En outre, le nœud, par cette prédiction peut s'assurer qu'il a exécuté correctement le protocole améliorant ainsi la robustesse de l'algorithme.
Pour prévoir le nombre de mouvements pour chaque nœud, les nœuds sont partitionnés en deux groupes (A) et (B) :

– La taille du groupe (A) est $|(A)| = \frac{n+1}{2}$ si n est impair. Ou $|(A)| = (n/2 - ((\sqrt{n}/2) - 1))$ si n est pair.

– La taille du groupe (B) est $|(B)| = \frac{n-1}{2}$ si n est impair. Ou $|(B)| = (n/2 + ((\sqrt{n}/2) - 1))$ si $n = N$ et n est pair.

Pour appliquer les fonctions de prédiction, on partitionne les nœuds en niveaux FL_j, SL_j et TL_j. Les partitionnement se fait selon l'ordre des nœuds dans la chaîne initiale du haut vers le bas. Dans ce qui suit, $FL_x(i)$ signifie que le nœud i a le niveau FL_x, $TL_x(i)$ signifie que le nœud i a le niveau TL_x et $SL_x(i)$ signifie que le nœud i a le niveau SL_x.

Le cas n est impair :

Groupe (A) :

Le nœud $((n+1)/2)$ prend un niveau spécial $PL0$ et le nœud $((n+1)/2)-1$ prend un niveau spécial $PL1$. Les premièrs $a = \sqrt{n}$ nœuds prennent le premier niveau FL_0. Les $B = (\sqrt{n} - 3)/2$ nœuds suivants prennent le premier niveau SL_0. Les $C = (\sqrt{n} - 1)/2$ nœuds suivants prennent le premier niveau TL_0. Par la suite :

– Les $a = a - 2$ nœuds suivants prennent le niveau FL_j si $TL(a-1)$.

– Les $B = B - 1$ nœuds suivants prennent le niveau SL_j si $FL(B-1)$.

– Les $C = C - 1$ nœuds suivants prennent le niveau TL_j si $SL(C-1)$.

La figure 4.24 montre un exemple de partitionnement en niveaux pour le groupe (A) avec $n = 49$.

$$O_j = \begin{cases} (\sqrt{n} - 1)/2, si\, j = 0. \\ O_{j-1} - 1, sinon. \end{cases} \tag{4.25}$$

$$W_j = \begin{cases} (3\sqrt{n} - 7)/2, si\, j = 0. \\ W_{j-1} - 3, sinon. \end{cases} \tag{4.26}$$

$$V_{i,j} = \begin{cases} (n + 1/2) - ((\sqrt{n} + 1)/2), si\, FL_0(i). \\ 0, si\, PL0(i). \\ 1, si\, PL1(i) \\ V_{i-1} - W_{j-1}, si\, FL(i) \wedge TL(i-1). \\ V_{i-1} - 1, si\, SL(i) \wedge FL(i-1). \\ V_{i-1} - O_j, si\, TL(i) \wedge SL(i-1). \\ V_{i-1}, sinon. \end{cases} \tag{4.27}$$

Avec $V_{i,j}$ le nombre de mouvements du nœud i qui a le niveau j. Pour chaque niveau TL_j est associé un certain nombre O_j, et pour chaque niveau FL_j est associé un certain nombre W_j. Un exemple dans la figure 4.24.

Groupe (B) :

Les deux premiers nœuds prennent le premier niveau FL_0. Les $A = 5$ nœuds suivants prennent le premier niveau SL_0. Les $B = 2$ nœuds suivants prennent le niveau FL_1. Les derniers $(\sqrt{n}+1)/2$ nœuds prennent le niveau SL_L. Par la suite :

– Les $A = A + 3$ nœuds suivants prennent le niveau SL_j si $FL(A-1)$.

– Les $B = B + 1$ nœuds suivants prennent le niveau FL_j si $SL(B-1)$.

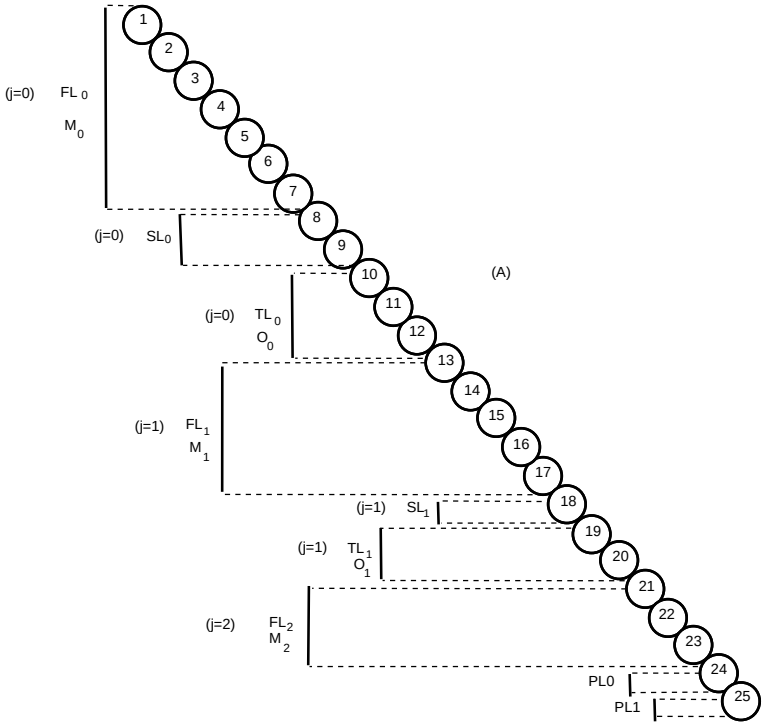

FIGURE 4.24 – Un exemple de partitionnement en niveaux du groupe (A) avec 49 nœuds

$$S_j = \begin{cases} 5, si\, j = 0. \\ S_{j-1} + 3, sinon. \end{cases} \tag{4.28}$$

$$D_j = \begin{cases} 4, si\, j = 0. \\ D_{j-1} + 1, sinon. \end{cases} \tag{4.29}$$

$$Z_{i,j} = \begin{cases} 1, si\, \mathrm{FL}_0(i). \\ 5, si\, \mathrm{SL}_0(i). \\ Z_{i-1} + S_{j-1}, si\, FL(i) \wedge SL(i-1). \\ Z_{i-1} + D_{j-1}, si\, SL(i) \wedge FL(i-1). \\ Z_{i-1}, sinon. \end{cases} \tag{4.30}$$

Avec : $Z_{i,j}$ est le nombre de mouvements du nœud i qui a le niveau j, S_j et D_j sont des fonctions utilisées pour calculer $V_{i,j}$. Pour chaque niveau FL_j est associé un certain nombre S_j, et pour chaque niveau SL_j est associé un certain nombre D_j.

Le cas n est pair :

Groupe (A) :

Le nœud $(n/2 - ((\sqrt{n}/2) - 1))$ prend un niveau spécial $PL0$ et le nœud $(n/2 - ((\sqrt{n}/2) - 2))$ prend un niveau spécial $PL1$. Les premiers $A = \sqrt{n}/2$ nœuds prennent le niveau FL_0. Les $B = (\sqrt{n}/2) - 1$ nœuds suivants prennent le premier niveau SL_0. Les $C = (\sqrt{n}/2) - 2$ nœuds suivants prennent le premier niveau TL_0. Par la suite :

- Les $A = A - 1$ nœuds suivants prennent le niveau FL_j si $TL(A-1)$.
- Les $B = B - 2$ nœuds suivants prennent le niveau SL_j si $FL(B-1)$.

– Les $C = C - 1$ nœuds suivants prennent le niveau TL_j si $SL(C-1)$.

Où $FL_x(i)$ indique que le nœud i a le niveau FL_x, $SL_x(i)$ signifie que le nœud i a le niveau SL_x et $TL_x(i)$ signifie que le nœud i a le niveau TL_x

$$P_j = \begin{cases} (3\sqrt{n}/2) - 2, si\, j = 0. \\ P_{j-1} - 4, sinon. \end{cases} \tag{4.31}$$

$$K_j = \begin{cases} (\sqrt{n}/2) - 1, si\, j = 0. \\ K_{j-1} - 1, sinon. \end{cases} \tag{4.32}$$

$$H_{i,j} = \begin{cases} (n/2) - 1, si\, FL_0(i). \\ 0, si\, PL0(i). \\ 1, si\, PL1(i) \\ H_{i-1} - P_j, si\, SL(i) \wedge FL(i-1). \\ H_{i-1} - 1, si\, TL(i) \wedge SL(i-1). \\ H_{i-1} - K_{j-1}, si\, FL(i) \wedge TL(i-1). \\ H_{i-1}, sinon. \end{cases} \tag{4.33}$$

Avec $H_{i,j}$ le nombre de mouvements du nœud i qui a le niveau j, P_j et K_j sont des fonctions utilisées pour calculer $V_{i,j}$. Pour chaque niveau TL_j est associé un nombre P_j et pour chaque niveau FL_j est associé un nombre K_j.

Groupe (B) :

Les deux premiers nœuds prennent le premier niveau FL_0. Les $A = 5$ nœuds suivants prennent le premier niveau SL_0. Les $B = 2$ nœuds suivants prennent le niveau FL_1. Par la suite :

– Les $A = A + 3$ nœuds suivants prennent le niveau SL_j si $FL(A-1)$.

– Les $B = B + 1$ nœuds suivants prennent le niveau FL_j si $SL(B-1)$.

$$U_{i,j} = \begin{cases} 1, si\, \mathrm{FL}_0(i). \\ 5, si\, \mathrm{SL}_0(i). \\ U_{i-1} + S_{j-1}, si\, FL(i) \wedge SL(i-1). \\ U_{i-1} + D_{j-1}, si\, SL(i) \wedge FL(i-1). \\ U_{i-1}, sinon. \end{cases} \tag{4.34}$$

Avec $U_{i,j}$ est le nombre de mouvements du nœud i qui a le niveau j, S_j et D_j sont des fonctions utilisées pour calculer $V_{i,j}$. Pour chaque niveau FL_j est associé un certain nombre S_j, et pour chaque niveau SL_j est associé un certain nombre D_j.

4.4.1.4 Complexité des messages envoyés

PPCNI utilise seulement les $(O(N/2))$ messages pour trouver le nœud du milieu. Comme pour PPCI, si un nœud change d'état à *well* sans vérifier que les nœuds l'ayant précédé sont dans l'état *well*, alors l'algorithme ne convergera pas vers la configuration souhaitée. Les prédicats (15), (16), (17), (18), (19), (20), (21), et (22) assurent, sans échange de messages, que le nœud ne change d'état que si tous les nœuds le précédant sont à l'état *int* ou *well*. Par conséquent,

le premier nœud qui commence la construction de la nouvelle couche n'a pas besoin d'attendre le message du premier nœud qui a commencé la couche précédente. Puisque le nœud qui est en train de vérifier les prédicats (15), (16), 17), (18), (19), (20), (21), et (22) peut avoir cette information en consultant (message) l'état de ses voisins. En d'autres termes, le message a été envoyé avant que le nœud a besoin de connaître l'état de son expéditeur ; lorsque le nœud a besoin de connaître cet état, il va trouver le message au niveau de voisin physique. Donc tout au long de l'algorithme en tout cas, nous n'avons pas besoin de transmettre des informations entre deux nœuds non voisins de la nouvelle couche. Cette efficacité s'explique par le fait que la synchronisation dans le changement d'état n'est pas requise pour les nœuds qui sont dans la même couche. En conséquence, PPCNI utilise seulement les $(O(N/2))$ messages pour trouver le nœud du milieu.

4.4.1.5 Généralisation de l'algorithme

L'algorithme présenté PPCNI est spécifique à un cas de la chaîne où les nœuds forment d'abord une ligne droite orientée vers des directions SE-NW. Dans cette section, nous décrivons comment l'algorithme peut être généralisé à tout type de chaîne initiale avec n'importe quelle direction. Le nœud avec un seul voisin du côté SW, SE ou E est identifié comme la première extrémité de la chaîne et le nœud avec un seul voisin du côté NW, NE ou W est désigné comme la seconde extrémité de la chaîne. Les autres nœuds identifient l'orientation de la chaîne (l'un des trois cas représentés sur la figure 4.13) en vérifiant l'orientation de ses deux voisins. Après la détection de l'orientation de la chaîne, noté D-$\overline{D}$, les nœuds exécutent une variante de l'algorithme de PPCNI en fonction de l'orientation $D \in \{W, NW, NE\}$. La variante de l'algorithme PPCNI, $PPCNI^D$, représente une adaptation de l'algorithme de PPCNI original pour les deux autres orientations possibles avec un changement des orientations dans les prédicats. Par exemple, si la chaîne initiale est orientée NE-SW, l'algorithme $PPCNI^{NE}$ est invoqué, et la forme carré est réalisée à l'aide de mouvements de type $moveAroundbad_v(u, P_w)$, $moveAroundwell_v(u, P_w)$ et $moveAroundwell_v(u, P_{nw})$. L'utilisation de ces trois prédicats est décrite dans la figure 4.25 qui présente un exemple avec des nœuds dont l'état est *bottom* se déplaçant autour des nœuds ayant l'état *well* ou *int*.

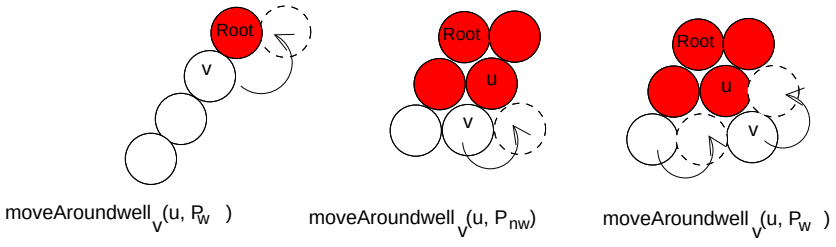

FIGURE 4.25 – L'adaptation des déplacements dans le cas de la chaîne NE-SW. Les nœuds sombres sont des nœuds ayant l'état *well* ou *int*

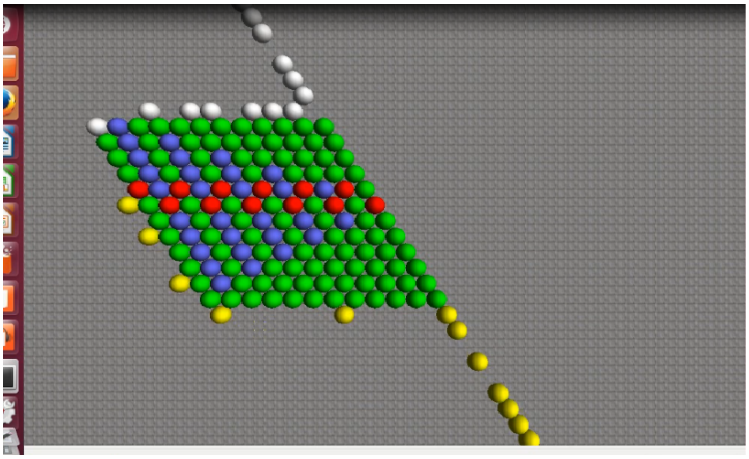

FIGURE 4.26 – Un exemple d'instance d'exécution de PPCNI. La couleur des nœuds désigne l'état du nœud : blanc pour *top*, jaune pour *bottom*, bleu pour *well*, vert pour *int*, et rouge pour *nper*. Seuls quelques nœuds *well* sont colorés en bleu.

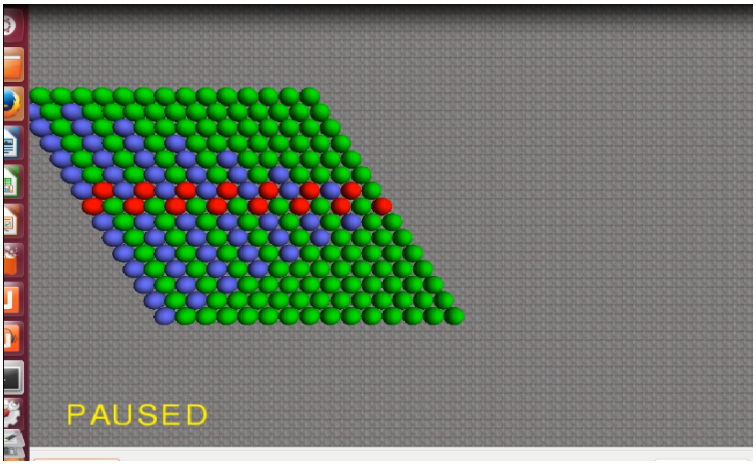

FIGURE 4.27 – Un exemple du résultat final de l'éxecution de PPCNI

4.5 Simulation et comparaison de PPCNI

L'exécution du programme a été simulée sous Dprsim [110]. Pour cela nous avons considéré un rayon de nœud égal à 1 mm. Les tests ont été effectués sur ordinateur Intel(R) Core(Tm)i5, 2.53 Ghz avec 4 GO de mémoire. On note dans les figures de la simulation, $PPCNI1$ pour les tests avec n impair avec $t(\eta) = \eta/2-1$ et $p(n) = ((n+\sqrt{n}-2)/2)$. Et $PPCNI2$ pour les valeurs pair de n, avec $v(n) = (n/2-1)$ et $s(\eta) = ((\eta+\sqrt{\eta}-2)/2)$.

Les résultats de simulation coïncident avec nos calculs analytiques. La figure 4.28 présente le temps d'exécution en fonction du le nombre de nœuds de l'algorithme.

Cette figure compare le temps d'exécution de l'algorithme proposé dans ce chapitre et les algorithmes ACI et PPCI. La figure 4.29 présente et compare le plus grand nombre de mouvements. La figure 4.30 donne les temps de génération des sous-carrés intermédiaires avec un scénario de 1000 nœuds.

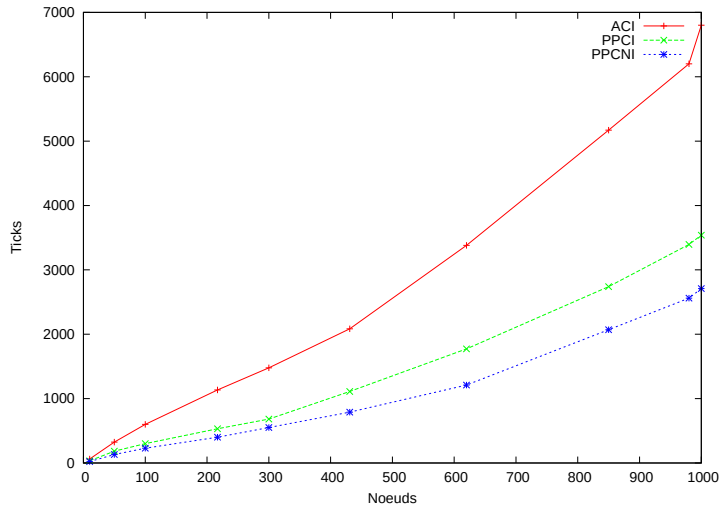

FIGURE 4.28 – Temps d'exécution de l'algorithme PPCNI

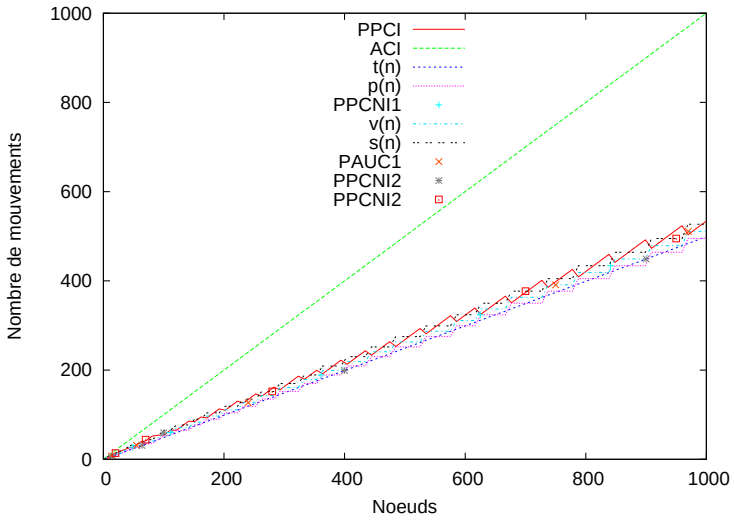

FIGURE 4.29 – Le plus grand nombre de mouvements

Les effets du parallélisme apparaissent clairement dans la courbe représentant le temps d'exécution de PPCNI (figures 4.30 et 4.28). La parallélisation des mouvements des microrobots réduit le temps d'exécution de l'algorithme. Ce qui a un effet direct sur les messages et un gain dans la communication. En effet si l'algorithme est rapide alors l'information critique arrive tôt au nœud concerné. Dans la figure 4.28 on remarque que chaque fois que la taille du réseau augmente la différence augmente considérablement.

On remarque dans la figure 4.29 que le nombre de mouvements de PPCNI est beaucoup plus faible que ACI, ce qui permet d'augmenter la durée de vie des nœuds et par conséquent la probabilité que les nœuds arrivent au terme de leur tâche. Cependant, PPCNI a besoin de onze états par nœud et l'algorithme ACI utilise seulement trois états.

On observe dans la figure 4.30 que les sous-carrés intermédiaires sont obtenus plus vite comparés aux autres protocoles. Ceci s'explique par le fait que dans les autres solutions, seuls

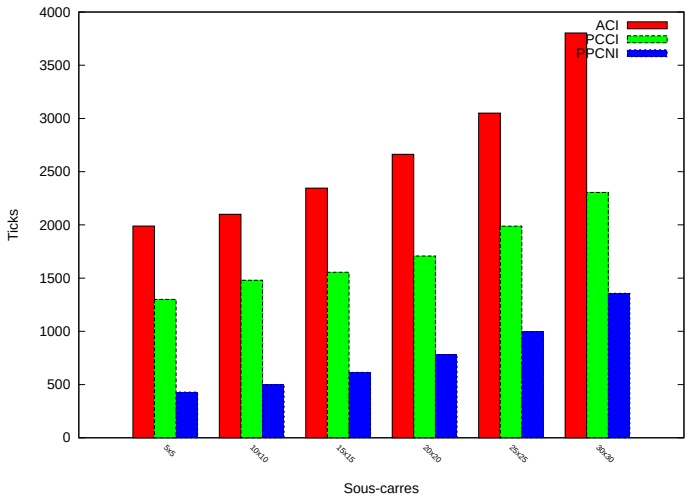

FIGURE 4.30 – Les carrés intermédiaires générés en fonction du temps d'une simulation sur 1000 nœuds

les nœuds feuilles peuvent se déplacer utilisant pour cela un arbre couvrant dont la construction est de complexité *(O(N))*. Dans l'algorithme ACI, le premier sous-carré est obtenu après la construction de l'arbre *(O(N))* et l'arrivée du nœud à l'extrémité sud de la chaîne au niveau de l'initiateur *(O(N))*. Dans PPCI, le premier carré intermédiaire est construit après la génération de l'arbre couvrant *(O(N))* et après l'arrivée des deux nœuds extrêmes au centre de la chaîne avec *(O(N/2))* .

4.6 Conclusion

Dans ce chapitre, j'ai proposé deux solutions parallèles pour achever la reconfiguration à partir d'une chaîne à un carré. Ces deux solutions ne nécessite pas la connaissance d'une carte de la forme cible. On a présenté un algorithme qui garantit la connexité du réseau durant le processus de reconfiguration et l'autre algorithme qui garantit la connexité à la fin de processus de reconfiguration. L'objectifs de ces deux algorithmes est d'améliorer les algorithmes ACI et ACNI du chapitre précédent en terme de performances dans le temps et l'énergie.

CHAPITRE 5

UN PROTOCOLE GÉNÉRALISÉ POUR LA RECONFIGURATION

Sommaire

5.1 Introduction

Les nœuds MEMS gagnent une attention croissante puisque ils peuvent effectuer diverses missions et tâches dans une large gamme d'applications, y compris la localisation d'odeur, la lutte contre les incendies, service médical, la surveillance et la sécurité, et de recherche et sauvetage. Afin d'aider les microrobots dans la réalisation de leur mission, il convient de réorganiser la forme géométrique des microrobots en fonction de leur tâche ainsi que l'environnement. Le problème d'auto-reconfiguration et la construction de la topologie logique optimale (en terme d'échange de messages) à partir d'une topologie physique aléatoire est un problème très difficile. La plupart des recherches dans le domaine des configurations d'essaims n'ont pas encore examiné les aspects de l'énergie et d'optimisation de la mémoire. Notre recherche vise donc à développer des essaims de microrobots auto-reconfigurables qui peuvent être déployés dans un environnement contraint. Spécifiquement, nous avons fait les hypothèses suivantes pour améliorer les travaux de la littérature :

• aucune carte de la forme cible,

• communication directe est limitée à la proximité immédiate du nœud (voisin physique),

- le même algorithme est exécuté sur chaque nœud,
- et la plage de détection est limitée.

Sur la base de ces hypothèses, nous abordons ici le problème de la réorganisation d'essaims de micro-robots. L'auto-configuration adaptative permet aux microrobots de tendre vers la réalisation de leur mission sans carte (positions finales prédéfinies) de la forme cible. Optimiser le coût de l'énergie des algorithmes auto-reconfiguration aura un impact direct sur l'efficacité énergétique de l'essaim. Dans la littérature, l'auto-reconfiguration peut avoir deux définitions. D'un côté, elle est définie comme un protocole, centralisé ou distribué, qui réorganise un ensemble de nœuds pour atteindre une topologie logique optimale Dans nos travaux, la forme carrée a été choisie car elle représente une topologie physique optimale pour l'échange de messages. Par exemple, un ensemble de micro-robots n connectés organisés comme une chaîne présente une complexité de diffusion de $O(n)$ dans le pire des cas (temps nécessaire pour atteindre tous les autres nœuds). Lorsque les n micro robots sont configurés en une forme carrée, la complexité de diffusion au pire des cas diminue à $O(\sqrt{n})$. Par conséquent, l'amélioration de la topologie logique de communication permettra d'améliorer les procédures de propagation et de la convergence dans le réseau. D'autre part, l'auto-reconfiguration est construite à partir de modules qui sont autonomes capables de changer la façon dont ils sont connectés, ce qui modifie la forme globale du réseau.

Dans ce chapitre on présente un protocole généralisé pour améliorer la topologie logique des systèmes de microrobots. Dans les deux chapitres précédents la configuration initiale est une chaîne droite. L'avantage de ces solutions proposées est leur caractère massivement distribués et la non utilisation d'une carte de la forme cible ce qui les rend efficaces et évolutives. Nous proposons dans ce chapitre une solution d'auto-reconfiguration où la forme initiale peut être quelconque. L'objectif est de réorganiser l'ensemble des nœuds sous forme d'un carré sans stockage des positions finales. Par conséquent, les nœuds peuvent exécuter l'algorithme indépendamment du lieu où ils sont déployés. En outre, la solution doit garantir l'atteinte de la forme cible avec un nombre réduit de mouvements. Le protocole est constitué de deux algorithmes principaux, *élection de l'initiateur* et *changement de forme*. L'élection de l'initiateur vise à choisir le meilleur initiateur pour initier l'algorithme de *changement de forme*. Le choix de l'initiateur se fait sur la base de l'estimation du nombre de mouvements nécessaires aux autres nœuds pour former le carré final. L'algorithme de changement de forme vise à convertir la topologie physique initiale vers la topologie finale par un processus incrémental de collaboration entre les nœuds. Pour cela l'algorithme se base sur l'idée de "*ramener la forme cible aux nœuds*" afin de réduire le nombre de mouvements. Ce protocole a été implementé dans DPRSim (Dynamic Physical Rendering Simulator) [110].

Le reste de ce chapitre est organisé comme suit : La section 2 discute du modèle et la terminologie et l'objectif. La section 3 discute le protocole proposé et analyse ses caractéristiques. La section 4 analyse les résultats de la simulation. Finalement, la section 5 présente nos conclusions et illustre nos suggestions pour les travaux futurs.

5.2 Énoncé du problème

5.2.1 Modèle et définitions

Dans Claytronics, le nœud Catom est modélisé comme une sphère qui peut avoir au plus six voisins 2D sans se chevaucher. Chaque nœud est capable de détecter la direction de ses voisins physiques (à l'est (E), à l'ouest (W), au nord-est (NE), au sud-est (SE), au sud-ouest (SW) et le nord-ouest (NW)). Dans ce travail, la topologie de départ peut être une topologie physique connectée arbitraire avec n nœuds reliés. Les communications ne sont possibles que par contact, ce qui signifie que seuls les voisins physiques peuvent avoir une communication directe.

Considérons le graphe non orienté connecté $G = (V, E)$ modélisant le réseau, où $v \in V$, est un nœud qui fait partie du réseau et, $e \in E$ une arête bidirectionnelle de communication entre deux voisins physiques. Chaque nœud $v \in V$ connait l'ensemble de ses voisins dans G, noté $N(v)$. Un *arbre couvrant* est un sous graphe connexe $G' = (V, E')$ sans cycles avec $E' \subset E$. Les deux nœuds d'une arête de l'arbre $e \in E'$ sont pour l'un parent et pour l'autre enfant. Les nœuds feuilles sont des nœuds sans enfants.

Pour faciliter l'explication des algorithmes, on modélise le réseau comme suit :

– L'espace 2D de déploiement et de mouvement des nœuds est modélisé par une matrice $J * I$. On note $N(j, i)$ le nœud à la ligne i et colonne j si la position est occupée (figure 5.1. Les lignes de la matrice sont numérotés de haut en bas et les colonnes de droit à gauche.

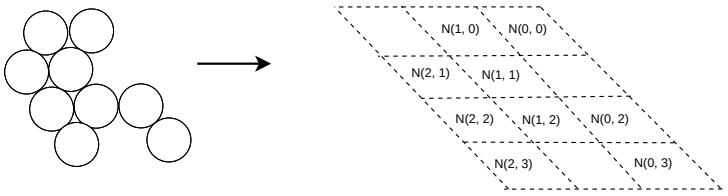

FIGURE 5.1 – Un exemple d'un réseau modélisé par une matrice

– On appelle *meilleure largeur* d'un nœud $N(j, i)$, la valeur $cntW = J - j$ et *meilleure hauteur* la valeur $cntH = I - i$.

– On appelle *meilleur initiateur absolu* le nœud $N(0, 0)$ (coin supérieur droite de l'espace de mouvement).

5.2.2 Objectif :

Comme décrit précédemment, nous proposons un protocole de reconfiguration sans carte (sans positions prédéfinies de la forme cible) optimisant la quantité de déplacements. Ce protocole a un autre objectif en même temps qui est d'optimiser la topologie physique. L'objectif de notre protocole est également d'améliorer la procédure de propagation, la tolérance aux pannes et de la convergence. Ce protocole permet de convertir n'importe quelle configuration d'un réseau connecté de microrobots dans une configuration carrée.

5.3 Protocole Proposé

Le protocole est constitué de deux algorithmes, *élection de l'initiateur* et *changement de forme* [57]. L'algorithme de l'élection de l'initiateur se base sur l'estimation du nombre de déplacements nécessaire pour atteindre la forme finale. En d'autres termes, nous cherchons avant le début de la reconfiguration à rapprocher la forme finale de la position initiale des nœuds. Comme pour les algorithmes précédents, l'algorithme de reconfiguration utilise un processus itératif, commençant par un nœud considéré comme un carré intermédiaire. A chaque itération, l'algorithme ajoute une nouvelle couche constituée d'un nombre de nœuds égal au nombre de nœuds dans la couche précédente plus deux. Les nœuds de chaque couche ajoutés changent leurs états pour devenir stables. L'algorithme de changement de forme synchronise la reconfiguration des nœuds en se basant sur l'état des nœuds et les messages.

5.3.1 Election de l'initiateur

Vue générale : L'objectif de l'algorithme de l'élection de l'initiateur est d'identifier simplement et rapidement le point de départ de l'algorithme de changement de forme (le premier carré intermédiaire) afin de garantir un minimum de mouvements nécessaires pour atteindre la forme cible. L'idée est double : D'abord le nœud initiateur choisi représente le coin supérieur droit de la forme cible, limitant la surface de redéploiement à un quart du plan. Deuxièmement, le nœud initiateur est choisi de manière à minimiser le nombre de nœuds en dehors de ce quart du plan. L'action combinée de ces deux principes fait que presque tous les nœuds sont dans la même zone et se déplacent dans cette zone réduisant ainsi la quantité de déplacement nécessaire. En outre, l'impact de l'algorithme de l'élection de l'initiateur sera discuté et analysé dans la partie expérimentale. Dans ce protocole, les nœuds seront organisés dans seulement un quart du plan (voir l'exemple de la figure 5.2). Les mouvements des nœuds seront effectués uniquement dans la zone A1 du plan pour éviter de parcourir de longues distances. Avec l'exemple de la figure 5.2, le meilleur initiateur est le nœud 1 car aucun nœud n'est hors du quart de plan défini par ce nœud. Si ce nœud n'est pas présent (figure 5.3), le meilleur initiateur est le nœud 4, car il y a seulement deux nœuds hors du quart de plan A1-a1 défini par le nœud 4 (le nœud 2 et le nœud 3). De même, si le nœud 4 n'est pas présent le meilleur initiateur est le nœud 2, et ainsi de suite.

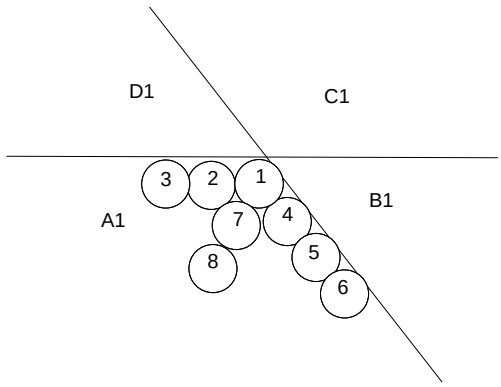

FIGURE 5.2 – Les nœuds se déplacent dans la zone $A1$

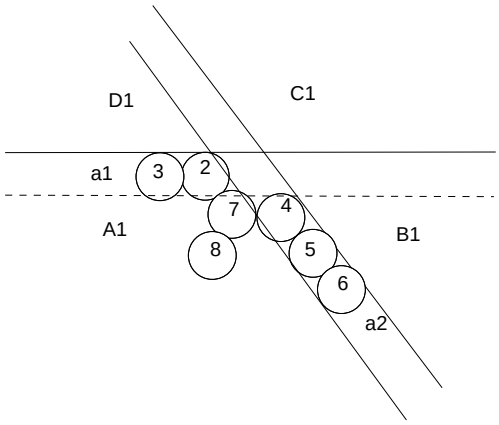

FIGURE 5.3 – Les nœuds se déplacent dans la zone $A1 - a1$, les nœuds qui sont en dehors de cette zone se joindront à elle, comme le cas des nœuds 2 et 3

Avec l'algorithme de l'élection de l'initiateur, tous les nœuds convergent vers le même meilleur initiateur. L'algorithme consiste en deux étapes. Dans l'étape 1 appelée *recherche de coordonnés* un algorithme distribué est exécuté. Dans l'étape 2 un algorithme heuristique appelé *recherche de meilleur initiateur* est exécuté localement sur chaque nœud en utilisant les informations obtenues par l'étape 1. Dans l'étape 1, chaque nœud calcule les valeurs de sa meilleure largeur et sa meilleure hauteur et enregistre celles des autres nœuds. Le but de l'algorithme d'élection de l'initiateur est de trouver le nœud qui va initialiser l'algorithme de changement de forme.

5.3.1.1 Recherche de coordonnées

Cette section présente l'algorithme de recherche de coordonnés. L'algorithme fonctionne par itération. A chaque itération, un prédicat satisfait est choisi et exécuté. L'objectif de la recherche de coordonnées est que chaque chaque nœud calcule sa meilleure largeur ($cntW$) et sa meilleure hauteur ($cntH$) et enregistre celles des autres nœuds. Les informations obtenues dans cette section seront utilisées à la prochaine étape pour rechercher le meilleur initiateur. A la fin de l'algorithme de recherche de coordonnées chaque nœud a enregistré une liste L contenant toutes les coordonnées des nœuds en sélectionnant pour chaque id reçu en retour ($receiveRESP_v(id, cntH, cntW)$) le $MAX(cntH)$ et $MAX(cntW)$.

Description de l'algorithme de recherche de coordonnées

Le prédicat $BroadcastMSG_v(id, 0, 0)$ est vrai pour chaque nœud d'identifiant id connecté au réseau au début de l'algorithme. Le nœud envoie le message $(id, 0, 0)$ à tous ses voisins. Avec le prédicat $ReceiveMSG - X_v(id, cntH, cntW)$ les voisins du nœud id reçoivent le message et la garde $visitedB_v(id)$) assure qu'un nœud ne reçoit le message en diffusion provenant d'un autre nœud qu'une seule fois. Après réception du message, le nœud incrémente, décrémente ou maintient les paramètres $cntH$, $cntW$. Pour calculer la meilleure largeur, à chaque message reçu depuis le voisin E ou NE, le nœud incrémente le paramètre $cntW$ reçu. A chaque message reçu depuis le voisin W ou SW, le nœud décrémente le paramètre $cntW$ reçu.

Variables et Prédicats

- initiator$_v()$: chaque nœud initie l'algorithme.
- broadcastMSG$_v(id, 0, 0)$: chaque initiateur envoie un message contenant son identitéid, une variable pour compter le meilleure hauteur $cntH = 0$, et une variable pour compter le meilleur largeur $cntW = 0$.
- receiveRESP$(id, cntH, cntW)$: le nœud reçoit le message $(id, cntH, cntW)$ en retour si c'était le premier message avec les mêmes paramètres $id, cntH, cntW$.
- receiveMSG$-X_v(id, cntH, cntW)$: with $X \in \{NW, NE, E, SE, SW, W\}$. Le nœud reçoit le message en diffusion si c'était le premier message ayant l'identité id.
- visitedB$_v(id)$: $true$ si le nœud a déjà reçu un message en diffusion avec le même id.
- visitedF$_v(id, cntH, cntW)$: vosité en retour (réponse), $true$ si le nœud a reçu en retour la même id avec le même $cntH$ et $cntW$.
- broadcastMSG$_v(id, cntH, cntW + 1)$: incrémenter $cntW$ et diffuser le message reçu par le nœud en direction E.
- broadcastMSG$_v(id, cntH + 1, cntW + 1)$: incrémenter $cntH$ et $cntW$ et diffuser le message reçu par le nœud en direction NE.
- broadcastMSG$_v(id, cntH + 1, cntW)$: incrémenter $cntH$ et diffuser le message reçu par le nœud en direction NW.
- broadcastMSG$_v(id, cntH, cntW - 1)$: decrémenter $cntH$ et diffuser le message reçu par le nœud dans le sens W.
- broadcastMSG$_v(id, cntH - 1, cntW)$: decrémenter $cntH$ et diffuser le message reçu par le nœud dans le sens SE.
- broadcastMSG$_v(id, cntH - 1, cntW - 1)$: decrémenter $cntH$ et $cntW$ et diffuser le message reçu par le nœud en direction de SW.
- broadcastRESP$_v(id, cntH, cntW)$: nœuds qui n'ont pas des voisins dans la direction E, SW et SE, et d'autres qui n'ont pas les voisins dans les directions W, SE et SW commenceront l'envoie des messages en retour.
- receiveRESP$_v(id, cntH, cntW)$: le nœud peut diffuser le retour si c'était le premier message avec le même id, $cntH$ et $cntW$.
- broadcastRESP$_v(id, cntH, cntW)$: le nœud diffuse le retour vers les autres nœuds jusqu'à ce que le message arrive au nœud id.

Prédicats vérifiés seulement dans le premier tour

$$\text{initiator}_v() \equiv Connected_v.$$
$$\text{broadcastMSG}_v(id, 0, 0) \equiv initiator_v().$$

Prédicats vérifiés à chaque tour

$$\text{receiveMSG-NE}_v(id, cntH, cntW) \equiv (Vne(v)) \wedge (\neg visitedBv(id)).$$
$$\text{receiveMSG-NW}_v(id, cntH, cntW) \equiv (Vnw(v)) \wedge (\neg visitedBv(id)).$$
$$\text{receiveMSG-SE}_v(id, cntH, cntW) \equiv (Vse(v)) \wedge (\neg visitedBv(id)).$$
$$\text{receiveMSG-SW}_v(id, cntH, cntW) \equiv (Vsw(v)) \wedge (\neg visitedBv(id)).$$
$$\text{receiveMSG-E}_v(id, cntH, cntW) \equiv (Ve(v)) \wedge (\neg visitedBv(id)).$$

receiveMSG-W$_v(id, cntH, cntW) \equiv (Vw(v)) \wedge (\neg visitedBv(id))$.
visitedB$_v(id) \equiv receiveMSG - X_v(id, -, -)$.
broadcastMSG$_v(id, cntH, cntW + 1) \equiv receiveMSG - E_v(id, cntH, cntW)$.
broadcastMSG$_v(id, cntH + 1, cntW + 1) \equiv receiveMSG - NE_v(id, cntH, cntW)$.
broadcastMSG$_v(id, cntH + 1, cntW) \equiv receiveMSG - NW_v(id, cntH, cntW)$.
broadcastMSG$_v(id, cntH, cntW - 1) \equiv receiveMSG - W_v(id, cntH, cntW)$.
broadcastMSG$_v(id, cntH - 1, cntW) \equiv receiveMSG - SE_v(id, cntH, cntW)$.
broadcastMSG$_v(id, cntH - 1, cntW - 1) \equiv receiveMSG - SW_v(id, cntH, cntW)$.
visitedF$_v(id, cntH, cntW) \equiv ReceveRESP_v(id, cntH, cntW)$.
broadcastRESP$_v(id, cntH, cntW) \equiv (\neg Ne(v)) \wedge (\neg Nsw(v)) \wedge (\neg Nse) \wedge \neg visitedF_v(id, cntH, cntW)$.
broadcastRESP$_v(id, cntH, cntW) \equiv (\neg Nw(v)) \wedge (\neg Nse(v)) \wedge (\neg Nsw) \wedge \neg visitedF_v(id, cntH, cntW)$.
receiveRESP$_v(id, cntH, cntW) \equiv \neg visitedF_v(id, cntH, cntW)$.
broadcastRESP$_v(id, cntH, cntW) \equiv receiveRESP_v(id, cntH, cntW)$.

L'algorithme de recherche de coordonnées

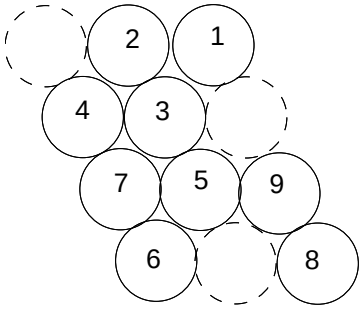

FIGURE 5.4 – Un exemple de réseau avec neuf nœuds

La même procédure est réalisée pour le calcul de la meilleure hauteur. A chaque message reçu depuis le voisin NE ou NW, le nœud incrémente le paramètre *cntH* reçu et à chaque message reçu depuis le voisin SE ou SW, le paramètre *cntH* est décrémenté. Quand le message diffusé par le nœud *idS* atteint un nœud *idD* qui n'a pas de voisins du côté E, SW et SE, le nœud *idD* diffuse un message de réponse. Pour cela, le nœud *idD* diffuse le message $BroadcastRESP_v(idS, cntH, cntW)$. Les nœuds voisins qui reçoivent ce message le rediffusent à leur tour. La garde $visitedF_v(id, cntH, cntW)$ est utilisée pour éviter de retransmettre une réponse concernant un même nœud *idS* avec les mêmes paramètres *cntH* et *cntW*.

Exemple

On prend l'exemple avec le nœud 1 dans la figure 5.4 sachant que les autres nœuds suivent la même procédure. Le nœud 1 envoie à ses voisins 2 et 3 le message (1, 0 ,0) avec le prédicat $broadcastMSG_v(id, 0, 0)$ et attend des réponses avec le prédicat $receiveRESP(id, cntH, cntW)$. A la réception de ce message, le nœud 2 incrémente *cntW* avec $broadcast_v(id, cntH, cntW+1)$, et le nœud 3 incrémente les paramètres *cntW* et *cntH* puis diffuse le message avec $broadcast_v(id, cntH+$

$1, cntW + 1$). Les messages retransmis sont donc respectivement (1, 0, 1) et (1, 1, 1) pour les nœuds 2 et 3. A la réception du message (1, 0, 1) par le nœud 4 envoyé par le nœud 2 ou le message (1, 1, 1) envoyé par le nœud 3, il le retransmet à ces voisins via le prédicat $broadcast_v(id, 1cntH + 1, cntW + 1)$ (si le message est reçu du nœud 2) ou via le prédicat $broadcast_v(id, cntH, cntW + 1)$ (si le message reçu du nœud 3). Le message retransmis devient (1, 1, 2). Le nœud 7 à la reception du message (1, 1, 2) envoyé par le nœud 4 ou (1, 1, 1) envoyé par le nœud 3 modifie les paramètres *cntW* et *cntH* en fonction de la direction de réception et rentransmet à ces voisins. Le message devient (1, 2, 2) et le nœud 7 diffuse ce nouveau message. de son côté le nœud 5, reçoit le message (1, 1, 1) envoyé par le nœud 3, puis incrémente le paramètre *cntH* et diffuse le message (1, 2, 1) via le prédicat $BroadcastMSG_v(id, cntH + 1, cntW)$. Le nœud 6 à la réception du message (1, 2, 2) envoyé par le nœud 7 ou (1, 2, 1) envoyé par le nœud 5, retransmet les message (1, 3, 2). De même, le nœud 9 reçoit puis décrémente le paramètre *cntW* avec le prédicat $broadcastMSG_v(id, cntH, cntW - 1)$ pour transmettre le message (1, 2, 0). Le nœud 9 envoie le message au nœud 8, ce dernier incrémente le *cntH* et transmet le message (1, 3, 0). Les nœuds 6 et 8 peuvent commencer la procédure de réponse avec l'envoi du message (1, 3, 2) et (1, 3, 0) aux autres nœuds(avec $broadcastRESP_v(id, cntH, cntW)$). Un nœud ignore les réponses identiques et ne garde que les valeurs maximales de *cntH* et *cntW* associé à un nœud donné. . Par exemple, le nœud 5 reçoit en réponse les messages (1, 3, 0) et (1, 3, 2) et ne stocke que le message (1, 3, 2).
La liste *Ln* représente la liste des coordonnées des nœuds enregistrés par chaque nœud à la fin de l'algorithme :
$Ln = \{(1,3,2),(2,3,1),(3,2,1),(4,2,0),(5,1,1),(6,0,0)\} \cup \{(7,1,0),(8,0,2),(9,1,2)\}$.

Lemme 4 *Puisque chaque nœud enregistre pour chaque id, les valeurs cntH et cntW, l'algorithme de recherche de coordonnées engendre une complexité mémoire de* $3n$.

Lemme 5 *La forme en chaîne représente topologie physique la moins avantageuse pour de nombreux algorithmes distribués en termes de tolérance aux pannes, des procédures de diffusion et de convergence. La latence maximale dans le réseau concerne les deux extrémités de la chaîne avec* $2n$ *messages (*n *messages pour la diffusion et* n *messages pour la réponse).*

5.3.1.2 Algorithme de recherche du meilleur initiateur

L'algorithme de recherche du meilleur initiateur (présenté ci-après) est exécuté par chaque nœud dans le réseau. Cet algorithme utilise les informations acquises par l'algorithme de recherche de coordonnées afin d'élire l'initiateur qui lancera l'algorithme de changement de forme. Le meilleur initiateur dans l'absolu s'il existe est celui dont les valeurs $(cntH, cntW) = (J, I)$, avec $J = MAX(cntH)$ et $I = MAX(cntW)$ et $cntW$ et $cntH \in Ln$. Ce nœud prend les coordonnées relatives (0, 0).

Ensuite, chaque nœud prend les coordonnées relatives $(J - cntH, I - cntW)$ et convertit de la même façon les paramètres *cntH* et *cntW* des nœuds stockés dans la liste *Ln*. La liste *Ln* prend alors la forme : $Ln = \{(x1, y1), (x2, y2), ..., (xn, yn)\}$. Soit la fonction $ID(x, y)$ qui retourne l'identifiant du nœud à la position (x, y).

Algorithme pour chaque nœud

Variables
- liste : $A, li, LC, R1, R2$.
- entier : Nli initialisé à 0, représente le nombre de nœuds hors la zone si le nœud ayant li comme coordonnées a été choisi

$A = \{(0,0)\}$;
si $\exists a = (x,y) \in A \wedge a \in L$ **alors**
fin de l'algorithme, l'initiateur est $ID(a)$;
sinon
début
$LC = \oslash$;
pour tous $a = (x,y) \in A, LC = LC \cup \{(a.x+1, a.y) \cup (a.x+1, a.y+1) \cup (a.x, a.y+1)\}$;
fin pour tous
$LC = \{l1, l2, l3, ..., ln\}$, avec $\forall li \in LC, li \notin A$;
pour chaque $li = (x,y)$ de LC le nœud calcule NLi :
début
calculer la liste $R1$ et $R2$ pour li ;
Soit $X = MAX(xi, i \in \{1..n\})$,avec $(xi, yi) \in A$
Soit $Y = MAX(yi, i \in \{1..n\})$,avec $(xi, yi) \in A$
$R1 = \{(X, y1), (X, y2), ..., (X, yn)\}$. Avec : $y1 \neq y2 \neq, ..., \neq yn$, $yi < y$ et $R1 \subseteq A$

$R2 = \{(x1, Y), (x2, Y), ..., (xn, Y)\}$. Avec : $x1 \neq x2 \neq, ..., \neq xn$, $xi < x$ et $R2 \subseteq A$

pour tous $xi \in R1$:
pour $m = xi + 1$ à $J\,NLi++$;
pour tous $yi \in R2$:
pour $m = yi + 1$ à $I\,NLi++$;
fin
$BEST = li$ ayant $(MIN\,NLi)$;
si $BEST \in L$ **alors**
Fin de l'algorithme, le meilleur est $ID(BEST)$.
sinon
début
$A = A \cup li$;
Répéter l'algorithme ;
fin
fin

Algorithme de recherche sur le meilleur initiateur

Le meilleur initiateur absolu, s'il existe, a pour coordonnées (0, 0). Au début, chaque nœud vérifie s'il existe un nœud avec les coordonnées (0, 0) dans la liste locale. Si c'est le cas, le nœud ID(0, 0) se déclare initiateur, et les autres nœuds reconnaissent l'initiateur. Si la position (0,0) est inoccupée, les positions candidates sont insérées dans la liste LC. A chaque itération, si aucune

des positions candidates n'est occupée alors la liste est remplacée par les positions voisines de la liste LC courante. Pour chaque position candidate dans LC le nombre de nœuds hors du quart de plan Nli est calculé et la position avec le plus petit Nli est élue. Si cette position est inoccupée (en vérifiant dans la liste L) une nouvelle liste de candidats LC est considérée dans le tour suivant et ainsi de suite.

5.3.2 Algorithme de changement de forme

L'algorithme de changement de forme (présenté ci-après) est un algorithme itératif. A chaque itération, les prédicats satisfaits sont exécutés. Pour décrire l'algorithme on désigne par *première ligne* tous les nœuds dont l'état est FR qui se trouvent à gauche de l'initiateur et par *première colonne* tous les nœuds dont l'état est FC qui sont au sud-est de l'initiateur. Une *nouvelle couche* est constituée d'une *nouvelle colonne* et une *nouvelle ligne* comme le montre la figure 5.6. Un nœud devient *stable* dés que il obtient l'état *well* signifiant qu'il a atteint une position définitive appartenant à la forme cible.
L'algorithme distribué cherche la forme désirée en utilisant un processus incrémental. Dans un incrément terminé, les nœuds qui la construisent appartiennent déjà à la forme ; ces nœuds vont aider les nœuds voisins et futurs nœuds voisins à obtenir des positions correctes. C'est-à-dire, le nœud prend des décisions et des actions en fonction des états des voisins. Au début, l'initiateur prend l'état *well* et devient stable avec le prédicat (2). Le prédicat (1) permet à l'initiateur de lancer la construction de l'arbre couvrant. L'arbre couvrant est utilisé pour assurer la connectivité du réseau et éviter de perdre le contact avec d'autres nœuds étant donné que la communication n'est possible que par contact. Si un nœud s'isole, il lui devient impossible de retrouver ou rejoindre les autres nœuds. L'utilisation de l'arbre couvrant permet également d'éviter les cycles infinis de mouvement. En effet, s'il y a un cycle dans le réseau, il y aura un mouvement infini, parce que si on suppose pour assurer la connectivité de réseau, le nœud suit le nœud voisin qui s'est déplacé dans la précédente étape, on peut trouver un cas où les nœuds se déplacent dans un cycle, comme le cas de la figure 5.5. Par définition, l'arbre est un réseau connecté sans cycle. Par conséquent, avec l'arbre, l'un des nœuds fils suit son parent.
Après avoir effectué un mouvement, un nœud parent ne peut de nouveau se déplacer qu'une fois qu'il est rejoint par le nœud fils et redevient de nouveau son parent. Avec le prédicat (4) les autres nœuds prennent des fils, sauf les feuilles qui ne peuvent être parents, parce que tous les voisins sont des parents (3). A l'aide du prédicat (5), les nœuds de la première ligne horizontale du carré prennent (au rythme de la construction des couches du carré) l'état FR et avec (6) les nœuds de la première colonne du carré prennent l'état FC. Ces deux derniers états sont utilisés pour ramener les nœuds en dehors du carré final vers le quart du plan en utilisant les prédicats (23), (24), (25) et (26).

L'état BE est utilisé pour remplir une nouvelle ligne horizontale du carré (les nœuds prennent l'état BE de droite à gauche). Le nœud de la couche courante avec le voisin du côté NW ayant l'état FC est le premier nœud de la nouvelle couche qui prend l'état BE(7). Ce nœud ne se déplace que s'il a un fils (ce fils a nécessairement qu'un état : *bad*) car ce nœud commence la construction d'une nouvelle ligne de la nouvelle couche.

Variables et prédicats

- parent(v, v) : l'initiateur (la racine de l'arbre) est un parent de lui-même.
- isLeaf(v) : le nœud v est une feuille, pas un parent.
- parent(v, u) : le nœud v est parent de son voisin u.
- state$_v(X)$: v a l'état $X \in \{wel, bad, FR, FC, BE, BE+, BNW, BNW+\}$, on note que le nœud ne peut pas prendre l'état *well* et *bad* en même temps. Au début, tous les nœuds sont initialisés avec l'état *bad*. On note que le nœud perd son état $BE+$ et $BNW+$ s'il se déplace et il réserve les autres états une fois obtenus.
- moveAround$well_v(u1, P_X)$: v se déplace autour $u1$ jusqu'à ce que ce dernier devient dans la direction X par rapport à v.
- moveTo$_v(u1, pos_{u1})$: le nœud v se déplace à l'ancienne position de son parent $u1$.
- r :un nombre entier représente l'étape actuel de l'algorithme pour le nœud

1. parent$(v, v) \equiv initiator_v()$.
2. state$_v(well) \equiv initiator_v()$.
3. isLeaf$(v) \equiv (\nexists u, parent(u, v)) \wedge state_v(bad)$.
4. parent$(v, u) \equiv (parent(w, v), u \neq w) \wedge (N(v) = u) \wedge state_u(bad) \wedge (/\exists z \in N(v), parent(v, z))$.
5. state$_v(FR) \equiv initiator_v() \vee (Ne(v) = u \wedge state_u(FR) \wedge state_v(well))$.
6. state$_v(FC) \equiv initiator_v() \vee (Nnw(v) = u \wedge state_u(FC) \wedge state_v(well))$.
7. state$_v(BE) \equiv (Nnw(v) = u \wedge state_u(well) \wedge state_u(FC)$.
8. state$_v(BE) \equiv (Ne(v) = u1 \wedge state_{u1}(bad) \wedge state_{u1}(BE)) \wedge (Nne(v) = u2 \wedge state_{u2}(well)) \vee ((Nse(v) = u1 \wedge state_{u1}(bad) \wedge state_{u1}(BE)) \wedge (Ne(v) = u2 \wedge state_{u2}(well)))$.
9. state$_v(BNW) \equiv (Ne(v) = u \wedge state_u(well) \wedge state_u(FR)$.
10. state$_v(BNW) \equiv (Nnw(v) = u1 \wedge state_{u1}(BAD) \wedge state_{u1}(BNW)) \wedge (Ne(v) = u2 \wedge state_{u2}(well)) \vee ((Nw(v) = u1 \wedge state_{u1}(bad) \wedge state_{u1}(BE)) \wedge (Nne(v) = u2 \wedge state_{u2}(well)))$.
11. state$_v(BE+) \equiv (Nnw(v) = u \wedge state_u(well)) \wedge \neg IsLeaf(v)$.
12. state$_v(BE+) \equiv (Ne(v) = u1 \wedge state_{u1}(bad) \wedge state_{u1}(BE+) \wedge state_{u1}(BE)) \wedge (Nnw(v) = u2 \wedge state_{u2}(well)))$.
13. state$_v(BNW+) \equiv (Ne(v) = u \wedge state_u(well)) \wedge \neg IsLeaf(v)$.
14. state$_v(BNW+) \equiv (Nnw(v) = u1 \wedge state_{u1}(bad) \wedge state_{u1}(BNW+) \wedge state_{u1}(BNW)) \wedge (Ne(v) = u2 \wedge state_{u2}(well)))$.
15. state$_v(well) \equiv (Nnw(v) = u1 \wedge state_{u1}(BNW) \wedge Nse(v) = u2 \wedge state_{u2}(BE)) \vee (Nw(v) = u1 \wedge state_{u1}(BNW) \wedge Ne(v) = u2 \wedge state_{u2}(BE))$.
16. state$_v(well) \equiv state_v(BEW) \wedge Nse(v) = u1 \wedge state_{u1}(BE)) \vee (state_v(BE) \wedge Nw(v) = u2 \wedge state_{u2}(BNW))$.
17. moveAround$well_v(u1, P_{nw}) \equiv state_v(bad) \wedge ((Nne(v) = u1 \wedge Nnw(v) = u2 \wedge state_{u1}(well) \wedge state_{u2}(well))$.
18. moveAround$well_v(u1, P_e) \equiv state_v(bad) \wedge ((Nne(v) = u1 \wedge Ne(v) = u2 \wedge state_{u1}(well) \wedge state_{u2}(well))$.

19. moveAround$well_v(u1, P_{ne}) \equiv state_v(bad) \wedge state_v(BNW+) \wedge (Ne(v) = u1 \wedge state_{u1}(well))$.

20. moveAround$well_v(u1, P_{ne}) \equiv state_v(bad) \wedge state_v(BE+) \wedge (Nnw(v) = u1 \wedge state_{u1}(well))$.

21. moveAround$well_v(u1, P_{ne}) \equiv state_v(bad) \wedge ((Nne(v) = u1 \wedge Ne(v) = u2 \wedge state_{u1}(well) \wedge state_{u2}(BE)) \wedge state_v(BE+)$.

21. moveAround$well_v(u1, P_{ne}) \equiv state_v(bad) \wedge ((Ne(v) = u1 \wedge Nnw(v) = u2 \wedge state_{u1}(well) \wedge state_{u2}(well)) \wedge state_v(BEW+)$.

22. moveAround$well_v(u1, P_{se}) \equiv state_v(bad) \wedge (Nsw(v) = u1 \wedge state_{u1}(FR))$.

23. moveAround$well_v(u1, P_w) \equiv state_v(bad) \wedge (Nsw(v) = u1 \wedge state_{u1}(FC))$.

24. moveAround$well_v(u1, P_e) \equiv state_v(bad) \wedge (Nse(v) = u1 \wedge state_{u1}(FR))$.

25. moveAround$well_v(u1, P_{nw}) \equiv state_v(bad) \wedge (Nsw(v) = u1 \wedge state_{u1}(FC))$.

26. moveTo$_v(u1, pos_{u1}) \equiv state_v(bad) \wedge (\exists x Nx(v) = u1 \wedge parent(u1, v) \wedge, r = r - 1) \wedge (\forall x, \neg \exists Nx(v) = u, state_u(well)) \wedge (\exists x Nx(v) = u2 \wedge parent(v, u2))$.

27. moveTo$_v(u1, pos_{u1}) \equiv state_v(bad) \wedge (\exists x Nx(v) = u1 \wedge parent(u1, v) \wedge isLeaf(v), r = r - 1)$.

Algorithme de changement de forme

Un nœud prend l'état BE s'il a un voisin dans la direction E ayant l'état BE grâce au prédicat (8). L'état BNW est utilisé pour remplir une nouvelle colonne commençant du haut vers le bas avec les deux prédicats (9) et (10). Le nœud a l'état BE ou BNW se déplace seulement si : il a un fils dont l'état $BE+$ ou $BNW+$ ou qu'il a reçu un message d'un nœud parent qui ne peut pas se déplacer.
Ainsi, avec (11), le nœud prend l'état $BE+$ si deux conditions sont remplies : le nœud est un nœud parent et appartient à la couche en cours de construction, i.e. le voisin E/NW est à l'état *well*. Si ces deux conditions sont remplies et que le nœud peut se déplacer vers la direction W/SE, il effectue alors le mouvement et l'un de ses voisins ou son fils le remplace dans son ancienne position (prédicats (27) et (28)).

Le nœud peut prendre l'état $BE+$ après avoir reçu un message (prédicat (12)) l'informant qu'il existe un ou plusieurs nœuds parents qui ne peuvent pas se déplacer. Si le nœud est libre de se mouvoir, il se déplace autour d'un nœud ayant l'état *well* avec le prédicat (20). De même

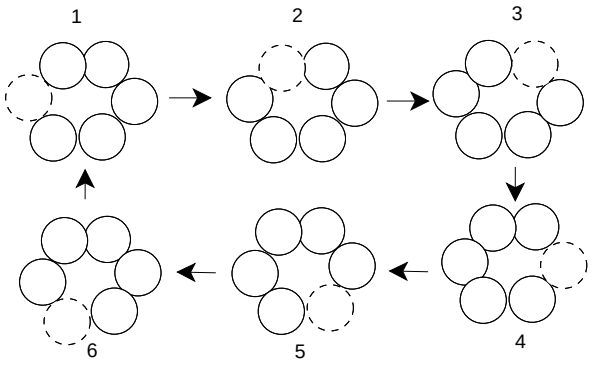

FIGURE 5.5 – Les nœuds ne peuvent pas appliquer le principe du *nœud suit son voisin* pour assurer la connexité du réseau et pour éviter de perdre des nœuds en raison des cycles. En outre, les nœuds peuvent entrer en état de collision quand ils se déplacent au même instant et au même endroit. Par conséquent, les nœuds peuvent se déplacer dans une boucle et ne peuvent pas converger vers la forme cible. La solution consiste à utiliser un arbre couvrant dont la racine est l'initiateur et le nœud suit son père, et après chaque mouvement le père attend son fils.

un nœud parent à l'état BNW peut prendre l'état $BNW+$ (prédicat 13). Ce nœud se déplace dans la direction SE, si possible en utilisant le prédicat (19), sinon il transmet un message de blocage avec le prédicat (14).

Grâce aux prédicats (15) et (6), un nœud change d'état vers *well* lorsque il est sûr que la nouvelle couche est complètement construite. Le prédicat (15) permet au nœud ayant un voisin NW à l'état BNW et un voisin SE à l'état BE de prendre l'état *well*. En effet, les états BE/BMW sont propagés à partir des nœuds de la couche courante avec les états FL/FC. Par conséquent, quand un nœud a deux voisins avec ces deux états, il est sûr que la nouvelle couche (ligne et colonne) est construite et le changement d'état à *well* est possible. Le prédicat (16) est similaire au prédicat (15), excepté que dans (16), le nœud vérifie si son état est BNW ou BE. Ces mécanismes assurent la synchronisation du passage des nœuds de la couche courante à l'état *well* (les nœuds de la couche en construction ont une vision locale du réseau et sans ces procédures ils ne savent pas quand la nouvelle couche est complètement remplie).

Le nœuds de la nouvelle couche commence par se déplacer vers la droite à l'aide du prédicat (17) jusqu'à ce qu'il ait un voisin dans la direction NW ayant l'état FC ou jusqu'à ce qu'il ait un voisin dans le sens E dont l'état est BE. De même pour construire la colonne le nœud se déplace avec le prédicat (18). Avec les prédicats (19) et (20) le nœud qui construit la nouvelle couche (ayant des voisins avec l'état *well*) se déplacent vers le gauche/ le bas que s'ils ont un fils ou après avoir reçu un message $BNW+/BE+$ à partir d'un nœud dans la nouvelle couche ayant un fils. Les prédicats (21) et (22) permettent aux nœuds dans une nouvelle couche de passer d'une ligne à une colonne ou vice versa. Le prédicat (21) permet le déplacement autour du nœud diagonal pour construire une nouvelle ligne. Le prédicat (22) permet aux nœuds d'aller de ligne en colonne afin de construire une nouvelle colonne. Les prédicats (23), (24), (25) et (26) permettent aux nœuds qui sont en dehors de la surface d'y entrer. Ces nœuds se déplacent autour d'un nœud ayant l'état FR ou FC pour entrer dans la surface et contribuer à construire une nouvelle couche. Les prédicats (27) et (28) aident à assurer la connectivité réseau en utilisant l'arbre. Avec le

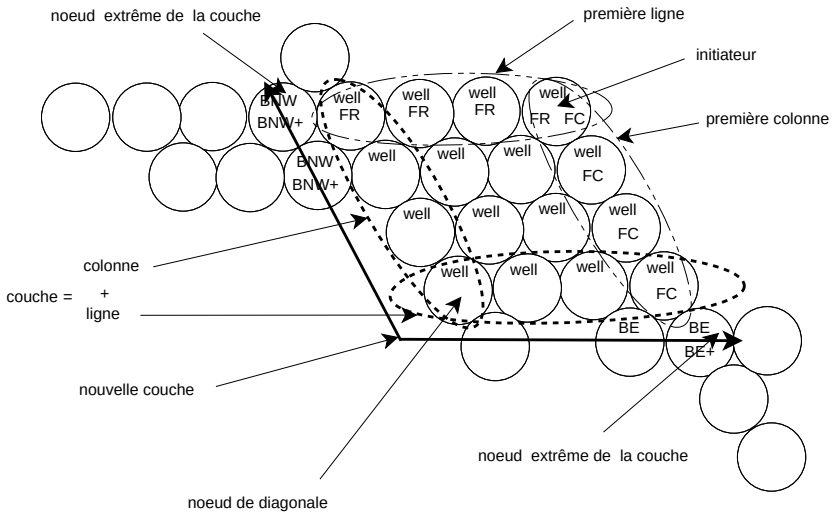

FIGURE 5.6 – Une instance de nœuds avec leurs états

prédicat (27), le nœud se déplace vers la position de son parent après le déplacement de l'un de ses fils à son ancienne position. Avec le prédicat (28) le nœud feuille suit son parent.

5.3.3 L'analyse de transmission du message

Dans cette section, on analyse le nombre de messages requis pour que les nœuds atteignent l'état *well*.

5.3.3.1 Le meilleur des cas

Lemme 6 *Dans le meilleur des cas, changement de forme utilise* $2n-2\sqrt{n}+O(n)-1$ *messages.*

Le nœud racine représente à lui seul la couche initiale change d'état sans utilisation de messages. Si la forme de départ est déjà un carré, aucun nœud n'aura à se déplacer et l'algorithme consiste en un processus permettant aux nœuds de prendre l'état *well*. Dans ce cas, la propagation des états $BE(BNW)$ se fait en parallèle à partir des deux nœuds extrêmes de la couche courante. Le message de propagation de l'état $BE(BNW)$ passe par tous les nœuds de la couche actuelle engendrant $n-1$ messages.

Les nœuds de la couche suivante attendent que les nœuds de la couche actuelle prennent l'état *well*. Un temps équivalent à la transmission d'un nombre de messages égal à la taille de la couche courante. La dernière couche représente un cas particulier car le changement d'état à *well* des nœuds la composant n'est pas utilisé par des couches suivantes. Ainsi on doit ajouter $O(n-2\sqrt{n})$ messages.

5.3.3.2 Le pire des cas

Lemme 7 *L'algorithme de changement de forme utilise* $3n-2\sqrt{n}+O(n)$ *messages dans le pire des cas.*

La chaîne droite représente le pire des cas en termes de nombre de messages échangés car la transmission des messages ne peut pas être effectuée en parallèle. Par conséquent, dans chaque couche de taille c, c messages sont nécessaires, ce qui correspond exactement à $n-1$ messages.

Theorem 5.3.1 *De Lemme 1, Lemme 2, Lemme 3 et Lemme 4, le protocole proposé utilise un espace mémoire de* $3n$ *: l'algorithme de l'élection de l'initiateur et l'algorithme de changement de forme ne sont pas exécutés en parallèle et utilisent* $4n - 2\sqrt{n} + O(n) - 1$ *messages dans le meilleur des cas et* $5n - 2\sqrt{n} + O(n)$ *dans le pire des cas.*

5.4 Simulation

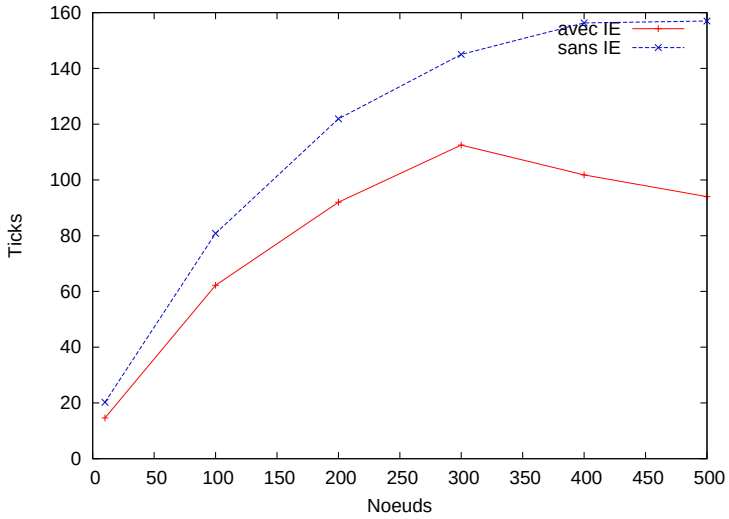

FIGURE 5.7 – Le temps d'exécution de l'algorithme avec et sans élection de l'initiateur

Le protocole proposé a été implémenté sous DPRSim (Dynamic Physical Rendering Simulator). Il s'agit d'une plate-forme multithread sur lequel on peut développer et tester de nouveaux algorithmes distribués pour les grands ensembles de nœuds. Il représente un environnement pratique pour implémenter les algorithmes distribués dédiés aux microrobots MEMS. Dans nos simulations le rayon du nœud est de 1 mm. Les simulations ont été exécutées sur un ordinateur portable avec Intel(R) Core(Tm) i5, processeur 2.53 GHz et 4 Go de mémoire. Les résultats des simulations suivantes sont la moyenne sur 100 exécutions avec un nombre de nœuds fixes (10, 100, 200, 300, 400 et 500) et différentes topologies physiques de réseaux connexes générés de manière aléatoire. Les nœuds sont placés sur une surface de 50 x 50 (mm^2). Les figures 5.10, 5.11, 5.12, 5.13 and 5.14 représentent des instances d'exécution de l'algorithme d'auto-reconfiguration. Les figures de cette section comparent les résultats de simulation obtenus avec l'algorithme d'élection de l'initiateur (IE pour élection de l'initiateur) et les résultats de l'algorithme de reconfiguration avec choix aléatoire de l'initiateur dans les réseaux (sans élection de l'initiateur). On rappelle que l'objectif de l'élection de l'initiateur est d'obtenir un nombre minimum de mouvements, avec un bon temps d'exécution et une réduction de la consommation d'énergie. Les figures comparent le temps d'exécution, la moyenne du nombre de mouvements (énergie consommée), et la moyenne du plus grand nombre de mouvements.

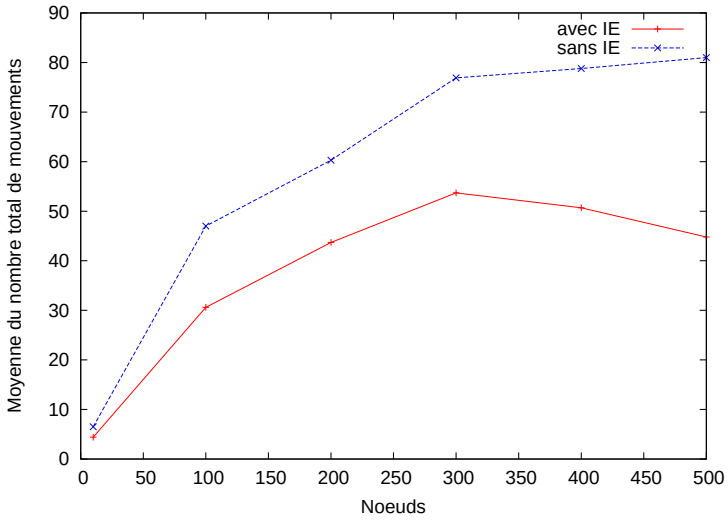

FIGURE 5.8 – Moyenne du nombre total de mouvements dans le réseau en fonction du nombre de nœuds

La figure 5.7 représente le temps d'exécution sur la base du nombre de nœuds. La figure 5.8 représente la moyenne du nombre total de mouvements dans le réseau sur la base du nombre de nœuds. La figure 5.9 représente la moyenne du plus grand nombre de mouvements dans le réseau sur la base du nombre de nœuds. Dans la figure 5.7 on observe clairement l'effet de l'algorithme d'élection de l'initiateur sur l'optimisation du temps d'exécution du protocole. On remarque que l'augmentation de la taille du réseau est accompagnée par l'augmentation de la différence en termes de temps d'exécution. Il est intéressant de noter que l'optimisation du temps d'exécution de l'algorithme a un impact direct sur le nombre de messages échangés. En effet quand l'algorithme est rapide, l'information critique arrive rapidement au nœud concerné. En outre, si la tâche à réaliser est un calcul distribué lourd, la rapidité de l'algorithme garantit la rapidité du calcul parallèle et la bonne répartition de la tâche sur les nœuds.

Dans les figures 5.8 and 5.9 on voit que, lorsque la taille du réseau augmente la différence de nombre de mouvements augmente de façon spectaculaire, ce qui impacte la durée de vie des nœuds. Par conséquent, la probabilité que le nœud continue sa tâche (ses mouvements), ceci améliore également la consommation d'énergie. On observe aussi sur les figures correspondant aux scénarios avec plus de 200 nœuds, les courbes diminuent en termes de temps et nombre de mouvements. En effet, quand le nombre de nœuds présent sur une même surface de simulation est très grand, le réseau devient dense. Par conséquent, ces bonnes performances sont dues à l'effet de la densité du réseau et de l'algorithme d'élection de l'initiateur.

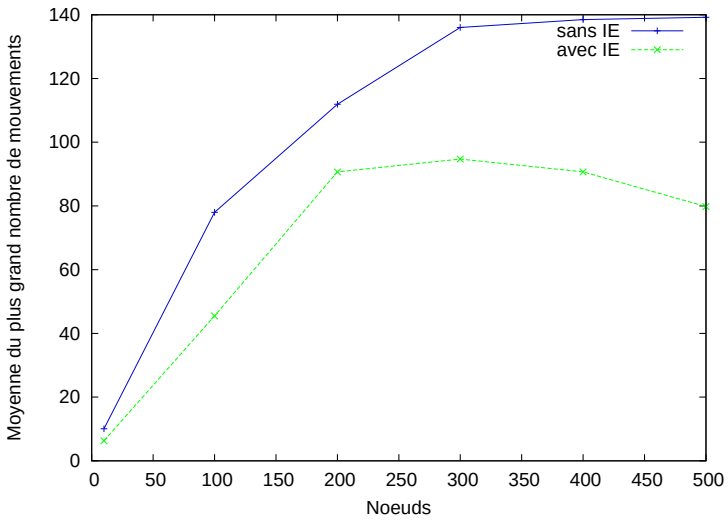

FIGURE 5.9 – Moyenne du plus grand nombre de mouvements dans le réseau sur la base du nombre de nœuds

FIGURE 5.10 – Instance d'exécution : T1

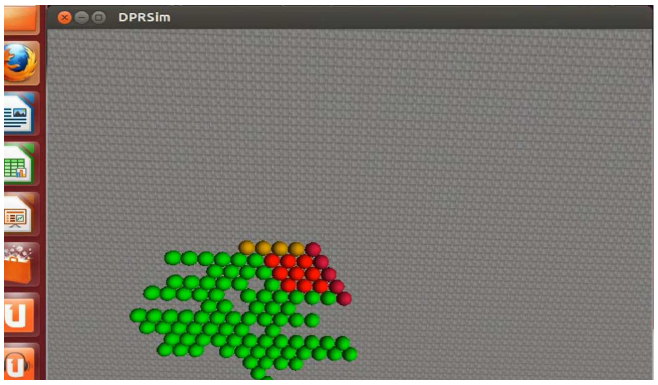

FIGURE 5.11 – Instance d'exécution : T2

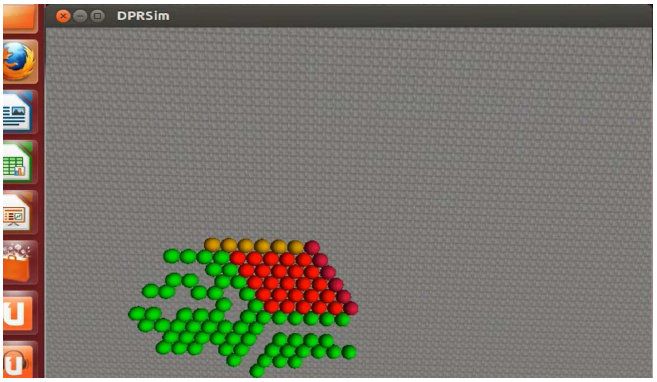

FIGURE 5.12 – Instance d'exécution : T3

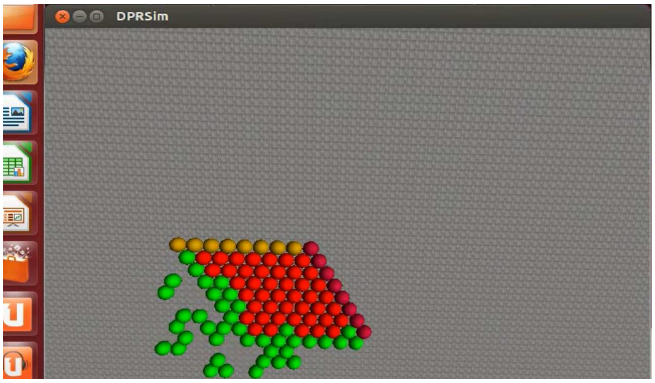

FIGURE 5.13 – Instance d'exécution : T4

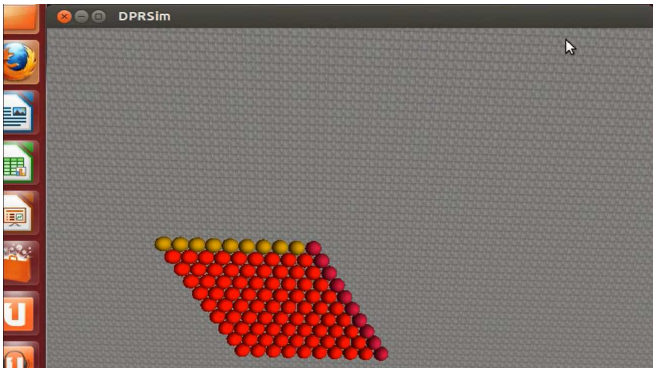

FIGURE 5.14 – Instance d'exécution : T5

5.5 Conclusion

Dans ce chapitre on a proposé une amélioration de la topologie logique d'un réseau de microrobots par une auto-reconfiguration vers un carré et ceci quelque soit la forme initiale. La principale différence de cet algorithme par rapport aux solutions présentées dans les chapitres précédents est que dans cette solution la forme initiale peut être n'importe quel réseau connexe. Dans ce travail, on a proposé un protocole d'auto-reconfiguration pour les microrobots sans carte de la forme cible ce qui rend l'algorithme efficace et évolutive. Le protocole présenté est le premier qui fournit une auto-reconfiguration autonome et portable, car il est indépendant de la carte qui construit la forme de la cible à partir de tout réseau initial connecté. Les positions prédéfinies de la forme cible sont remplacées par la collaboration entre les nœuds, ce qui rend l'algorithme plus intelligent que les algorithmes basés sur l'utilisation de la carte de la forme cible.

CHAPITRE 6

CONCLUSION GÉNÉRALE

Les microrobots MEMS représentent un des enjeux technologiques majeurs des dernières décennies en raison des prévisions sur leur usage massif dans la vie quotidienne notamment dans le but daméliorer le bien-être des personnes. En effet, les microrobots MEMS gagnent une attention croissante due à leur utilisation dans diverses missions et tâches dans une large gamme d'applications, y compris la localisation d'odeur, la lutte contre les incendies, le service médical, la surveillance et la sécurité, et la recherche et le sauvetage. Ils peuvent accomplir des tâches dans des environnements non structurés, complexes, dynamiques et inconnus. Pour réaliser ces missions les microrobots MEMS doivent appliquer des protocoles de redéploiement afin de s'adapter aux conditions de travail. Ces protocoles de reconfiguration doivent composer avec les limites des nœuds MEMS notamment celles relatives aux ressources mémoire et d'énergie. Aussi, ces algorithmes devraient être efficaces, évolutifs, robustes et utilisent seulement les informations locales. Par conséquent, dans cette thèse, on a proposé des algorithmes de reconfiguration efficaces pour des microrobots MEMS modulaires. L'objectif de ces algorithmes est d'optimiser la topologie logique des microrobots en utilisant des protocoles de redéploiement distribués de la topologie physique. Notre étude de l'état de l'art a montré qu'un nombre grandissant de recherches sont effectuées sur le contrôle du redéploiement des microrobots. Des algorithmes de reconfiguration centralisés et distribués ont été proposés pour des robots hexagonaux utilisant les principes de mouvements sur un chemin de substrat. D'autres solutions centralisées et distribuées ont été proposées dans la littérature pour des robots cubiques. Ces solutions utilisent les principes de graines, la croissance dirigée et les automates cellulaires pour aboutir à des configurations arbitraires.

Nous avons orienté notre étude vers les approches traitant des microrobots réalisés dans le cadre du projet Claytronics appelés Catoms. Certaines de ces méthodes abordent le problème par décomposition hiérarchique où l'objectif est la simplification des mouvements des catoms. Pour cela les deux primitives « création » et « destruction » des catoms ont été proposées pour accomplir la reconfiguration à des formes arbitraires. Ces solutions de la littérature utilisent les positions prédéfinies de la forme cible. Par conséquent, les algorithmes dépendent de la taille du système des microrobots limitant ainsi l'évolutivité. En effet si la forme cible est constituée

d'un nombre très grand de nœuds (des millions) chaque nœud doit sauvegarder dans sa mémoire locale toutes ces positions. Ce qui n'est pas possible car les microrobots sont dotés d'un espace de stockage très limité. De plus, dans ces approches, chaque nœud doit connaître dès le départ la taille du système (le nombre de microrobots) ainsi que sa position courante à tout moment. Certaines des solutions proposées nécessitent qu'au moins un nœud de la configuration initiale soit déjà dans la forme cible. Etant donné les ressources limitées qu'ont les nœuds MEMS, dans cette thèse, j'ai proposé des protocoles de reconfiguration sans carte de la forme cibles, ce qui a amené à l'utilisation d'une complexité constante de mémoire. Les solutions proposées représentent mes avantages suivants : réduction de la consommation énergétique, robustesse et évolutivité car ne stockant pas la carte de la forme cible (les positions prédéfinies de la forme cible), fonctionnement par collaboration entre les microrobots.
Le premier algorithme est basé sur un mécanisme de réveil-sommeil et assure la connexité du réseau le long de la procédure de reconfiguration. Le deuxième algorithme est plus rapide mais nassure pas une connexité continue du réseau lors de la phase de reconfiguration. Ces algorithmes fonctionnent par évolution de l'état des nœuds du réseau. Trois états sont nécessaires permettant de maintenir la complexité spatiale des algorithmes. Pour optimiser le temps de convergence des algorithmes, des solutions ont été proposées pour exploiter les possibilités de parallélisassions de la construction de la forme cible. Les tests effectués montrent une plus rapide convergence tout en maintenant le caractère évolutif des solutions et un nombre de mouvements minimum. En fin, on a présenté un protocole distribué plus général pour optimiser la topologie logique partant d'une configuration connexe arbitraire.

Perspectives

A partir des recherches que nous avons menées lors de ces travaux, plusieurs nouvelles directions de recherche s'offrent à nous. Dans un premier temps nous nous intéressons à l'extension de l'algorithme de reconfiguration au départ de n'importe quelle forme afin d'inclure des mécanismes de sauvegarde d'énergie et de maintien de la connexité instantanée du réseau, cela pourrait etré achevé en utilisant une décomposition de la forme final à plusieures forme (carré, triangle), apres dans une autre étape faire l'assemblage pour faire la forme final. L'objectif final étant de concevoir des algorithmes de reconfiguration pour des formes plus générales tout en gardant les atouts obtenus dans ces algorithmes à savoir l'évolutivité, la robustesse et l'efficacité.

Pour cela des principes du routage par permutation peuvent être exploités avec la reconfiguration et la permutation des données s'exécutant en parallèle.Which generates a large number of messages exchanged. L'autre grand axe de recherche reste l'étude des mécanismes de tolérance aux pannes dans les réseaux de microrobots. En effet, les algorithmes présentés dans cette thèse supposent que les nœuds MEMS sont non défectueux et ne sont pas sujet à des fautes d'exécution. Pour cela, on envisage de commencer par introduire une probabilité qu'un nœud soit en panne ou qu'il y ait une erreur de reconfiguration dans le cas des chaînes puis dans le cas plus général de formes quelconques. L'objectif est alors d'enrichir le fonctionnement des algorithmes afin de répondre au mieux aux scénarios de panne les plus probables.

BIBLIOGRAPHIE

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss *Amorphous computing.* Communications of the ACM, 43(5) :7482, 2000).

[2] M.P. Ashley-Rollman, P. Pillai, and M. L. Goodstein. *Simulating multi-million-robot ensembles.* In Proceedings of the 2011 IEEE International Conference on Robotics and Automation, pages 10061013, 2011.

[3] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. *Meld : A declarative approach to programming ensembles.* In Proc. of the IEEE Intl Conf. on Intelligent Robots and Systems, Oct. 2007.

[4] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell *A Language for Large Ensembles of Independently Executing Nodes.* In Proceedings of the International Conference on Logic Programming (ICLP 09), July, 2009.

[5] T. Balch and M. Hybinette. *Social potentials for scalable multi-robotformations,.* In Proc. IEEE International Conference on Robotics and Automation, 2000

[6] J. Bateau, A. Clark, K. McEachern, E. Schutze, and J. Walter, *Increasing the Efficiency of Distributed Goal-Filling Algorithms for Self-Reconfigurable Hexagonal Metamorphic Robots.* Proc. of the International Conference on Parallel and Distributed Techniques and Applications, PDPTA 2012, Las Vegas, July 2012

[7] J. Bourgeois, J. Cao, M. Raynal, D. Dhoutaut, B. Piranda, E. Dedu, A. Mostefaoui, and H. Mabed. *Coordination and Computation in distributed intelligent MEMS.* In AINA 2013, 27th IEEE Int. Conf. on Advanced Information Networking and Applications, Spain, p 118–123, 2013.

[8] J. Bourgeois and S.C. Goldstein. *Distributed Intelligent MEMS : Progresses and perspectives*, IEEE Systems Journal, 2014

[9] J. Bourgeois and S.C. Goldstein. *Distributed Intelligent MEMS : Progresses and perspectives*, In Ljupco Kocarev, editor, ICT Innovations 2011, volume 150 of Advances in Intelligent and Soft Computing, Ohrid, Macedonia, pages 15–25, 2012. Springer. Note : Keynote talk at the ICT Innovations 2011 conference

[10] N. Bulusu, J. Heidemann, D. Estrin, *GPS-less Low-Cost Outdoor Localization for Very Small Devices.* IEEE Personal Communications Magazine,7 :28-34, 2000

[11] Z. Butler, R. Fitch, D. Rus, and Y. Wang *Distributed goal recognition algorithms for modular robots.* In Proceedings, IEEE International Conference on Robotics and Automation (ICRA02), volume 1, pages 110116,Washington, DC, USA, 2002

[12] Z. Butler, R. Fitch, and D. Rus *Experiments in locomotion with a unit-compressible modular robot.* In Proceedings, IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, pages 28132818, Lausanne, Switzerland, 2002

[13] Z. Butler, R. Fitch, and D. Rus *Experiments in distributed control of modular robots.* In Siciliano, B. and Dario, P., editors, Proceedings, Experimental Robotics VIII, volume 5 of Springer Tracts in Advanced Robotics, 2003

[14] Z. Butler, K. Kotay, D. Rus, and K. Tomita *Cellular automata for decentralized control of self-reconfigurable robots.* In Proceedings, IEEE Int. Conf. on Robotics and Automation (ICRA01), workshop on Modular Self-Reconfigurable Robots, Seoul, Korea, 2001

[15] Z. Butler, S. Murata, and D. Rus *Distributed replication algorithms for self-reconfiguring modular robots.* In Proceedings, Distributed Autonomous Robotic Systems, pages 3748, Fukuoka, Japan, 2002

[16] Z. Butler, K. Kotay, D. Rus, and K. Tomita *Generic decentralized control for a class of self-reconfigurable robots.* In Proceedings, IEEE Int.Conf. on Robotics and Automation (ICRA02), volume 1, pages 809815,Washington, DC, USA, 2002

[17] Z. Butler, S. Byrnes, D. Rus *Distributed motion planning for modular robots with unit-compressible modules.* In Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 790-796, 2001.

[18] Z. Butler, S. Byrnes, D. Rus *Distributed Motion Planning for 3-D Modular Robots with Unit-Compressible Modules.* Workshop on the Algorithmic Foundations of Robotics, 2002.

[19] P. Bhat, J. Kuffner, S. Goldstein, S. Srinivasa *Hierarchical Motion Planning for Self-Reconfigurable Modular Robots.* In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems IROS, 2006.

[20] J. D. Campbell and P. Pillai. *Collective Actuation* . International Journal of Robotics Research, 27(3-4) :299314,2008.

[21] J. D. Campbell and P. Pillai. *Collective Actuation* . In RSS 2006 Workshop on Self-Reconfigurable Modular Robots, August, 2006.

[22] J. D. Campbell, P. Pillai, and S. C. Goldstein. *The Robot is the Tether : Active, Adaptive Power Routing for Modular Robots With Unary Inter-robot Connectors.* n IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005), pages 410815, August, 2005.

[23] Y. Cao, A. S. Fukunaga, A. Kahng , *Cooperative Mobile Robotics : Antecedents and Directions*, Journal of Journal Autonomous Robots,Springer Volume 4 Issue 1, Pages 7-27, 1997.

[24] A. Castano and P. Will *Representing and discovering the configuration of conro robots.* In Proceedings, Int. Conf. on Robotics and Automation, Seoul,Korea, 2001

[25] D. Coore. *Introduction to Amorphous Computing.*. UPP, volume 3566 of Lecture Notes in Computer Science, page 99-109. Springer, (2004)

[26] G. Chirikjian. *Kinematics of a metamorphic robotic system.* In Proc. of IEEE Intl. Conf. on Robotics and Automation, pages 449455, 1994.

[27] G. Chirikjian, A. Pamecha, and I. Ebert-Upho *Evaluating efficiency of self-reconfiguration in a class of modular robots.* Journal of Robotic Systems, Vol. 13, No. 5, pages 317-338, May 1996.

[28] G. Chirikjian and A. Pamecha. *Bounds for self-reconfiguration of metamorphic robots.* In Proc. of IEEE Intl. Conf. on Robotics and Automation, pages 1452, 1457, 1996.

[29] D. Dewey, S. S. Srinivasa, M. P. Ashley-Rollman, M. De Rosa, P. Pillai, T. C. Mowry, J. D. Campbell, and S. C. Goldstein, *Generalizing metamodules to Simplify Planning in Modular Robotic Systems.* In Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems IROS 08, September, 2008.

[30] J. Elson and K. Romer *Wireless sensor networks : a new regime for time synchronization.* SIGCOMM Comput. Commun. Rev. , 33(1) :149154, 2003.

[31] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein, *Distributed Localization of Modular Robot Ensembles.* In Proceedings of Robotics : Science and Systems,June, 2008.

[32] T. Fukuda, K. Sekiyama, T. Ueyama, F. Arai *Efficient communication method in the cellular robotic system*, In Proceedings of the IEEE International Conference on Intelligent Robots and Systems, 1993.

[33] S. Funiak, P. Pillai, Michael P. Ashley-Rollman, Jason D. Campbell, and Seth Copen Goldstein. *Distributed Localization of Modular Robot Ensembles.* International Journal of Robotics Research, 28(8) :946961,2009.

[34] S. Funiak, P. Pillai, J. D. Campbell, and S. C. Goldstein. *Internal Localization of Modular Robot Ensembles.* In Workshop on Self-Reconfiguring Modular Robotics at the IEEE International Conference on Intelligent Robots and Systems (IROS) 07, October, 2007.

[35] S. Ganeriwal, R. Kumar, and M. B. Srivastava *Timing-sync protocol for sensor networks.* In Proceedings, SenSys 03 , pages 138149, New York, NY, USA, 2003.

[36] S. C. Goldstein and T. C. Mowry. *Claytronics : A scalable basis for future robots.* In RoboSphere 2004, November, 2004.

[37] S. C. Goldstein and T. C. Mowry. *Claytronics : An Instance of Programmable Matter.* IIn Wild and Crazy Ideas Session of ASPLOS, October, 2004.

[38] S. C. Goldstein, J. D. Campbell, and T. C. Mowry. *Programmable Matter.* IEEE Computer, 38(6) :99101,June, 2005.

[39] S. C. Goldstein, T. C. Mowry, J. D. Campbell, P. Lee, P. Pillai, J. F. Hoburg, P. B. Gibbons, C. Guestrin, J. Kuffner, B. Kirby, B. D. Rister, M. De Rosa, S. Funiak, B. Aksak, and R. Sukthankar. *The Ensemble Principle.* In 13th Foresight Conference of Advanced Nanotechnogy, October, 2005.

[40] C. Gui and P. Mohapatra. *Power conservation and quality of surveillance in target tracking sensor networks.* In Proceedings, MobiCom '04, pages 129.143, New York, NY, USA, 2004. ACM Press.

[41] S. Hollar, A. Flynn, C. Bellew, and K.S.J. Pister, *Solar powered 10mg silicon robot*, In MEMS, Japan, January 2003.

[42] K. Hosokawa, T. Tsujimori, T. Fujii, H. Kaetsu, H., Asama, Y. Kuroda, and I. Endo *Self-organizing collective robots with morphogenesis in a vertical plane.* In Proceedings, IEEE Int. Conf. on Robotics and Automation (ICRA98), pages 28582863, Leuven, Belgium, 1998

[43] C. Jones, M. J. Mataric, *From local to global behavior in intelligent self-assembly.In : Proceedings of the 2003 IEEE International Conference on Robotics and Automation,*ICRA 2003, vol. 1, pp. 721-726, Los Alamitos, 2003.

[44] M. Karagozler, B. Kirby, W. Lee, E. Marinelli, T. C. Ng, M. Weller, and S.C. Goldstein *Ultralight modular robotic building blocks for the rapid deployment of planetary outpostst.* in Revolutionary Aerospace Systems Concepts Academic Linkage Forum, 2006, pp. 115.

[45] B. Kirby, B. Aksak, S. C. Goldstein, J. F. Hoburg, T. C. Mowry, and P. Pillai. *A Modular Robotic System Using Magnetic Force Effectors.* In Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS 07), October, 2007.

[46] K. Kosuge, D. Taguchi, T. Fukuda, M. Sakai, K. Kanitani, *Decentralized control of robots for dynamic coordination*, In Proceedings of the IEEE International Conference on Intelligent Robots and Systems, October 1995.

[47] F. Kribi, P. Minet, A. Laouiti, *Redeploying mobile wireless sensor networks with virtual forces*, IFIP Wireless Days, Paris, France, December 2009.

[48] B. Kirby, J. D. Campbell, B. Aksak, P. Pillai, J. F. Hoburg, T. C. Mowry, and S. C. Goldstein. *Catoms : Moving Robots Without Moving Parts.* In AAAI (Robot Exhibition), pages 17301, July, 2005.

[49] H. Lakhlef, H. Mabed, and J. Bourgeois *Distributed and Efficient Algorithm for Self-reconfiguration of MEMS Microrobots.* In SAC 2013, 28-th ACM Symposium On Applied Computing, Coimbra, Portugal, pages 560–566, March 2013

[50] H. Lakhlef, H. Mabed, and J. Bourgeois *Dynamicity to Save Energy in Microrobots Reconfiguration.* In UIC 2013,10th IEEE Int. Conf. on Ubiquitous Intelligence and Computing, Salerne, Italy, pages 246–253, December 2013

[51] H. Lakhlef, H. Mabed, and J. Bourgeois *Distributed and Dynamic Map-less Self-reconfiguration for Microrobot Networks.*In NCA 2013, 12th IEEE International Symposium on Network Computing and Applications, Cambridge, MA, United States, pages 55–60, August 2013

[52] H. Lakhlef, H. Mabed, and J. Bourgeois *Energy and Memory-efficient distributed self-recon guration for modular sonsor/robot networks* Journal of SuperComputing, Springer, Accpted To Appear, 2014.

[53] H. Lakhlef, J. Bourgeois, H. Mabed, and Seth Copen Goldstein *Energy-Aware Parallel self-reconguration for microrobots networks* JPDC, Journal of Parallel and Distributed Computing - Elsevier, 2015

[54] H. Lakhlef, H. Mabed, and J. Bourgeois *Parallel Self-reconguration for MEMS Microrobots* In IEEE EUROCON 2013, 7-th IEEE International conference on Computer as a Tool, Zagreb, Croatia, IEEE Computer Society, pages 283-290, July 2013.

[55] H.Lakhlef, J. Bourgeois, and H. Mabed *Robust Parallel Redeployment Algorithm for MEMS Microrobots* 28th IEEE International Conference on Advanced Information Networking and Applications, Victoria, Canada, pages ******, IEEE Computer Society, May 2014.

[56] H. Lakhlef, J. Bourgeois, and H. Mabed *Efficient Parallel Self-reconguration Algorithm for MEMS Microrobots* In Euromicro PDP 2014, 22-nd Euromicro. Conf. on Parallel, Distributed, and Network-Based Processing, Turin, Italy, pages ******, IEEE Computer Society, February 2014.

[57] H. Lakhlef, H. Mabed, and J. Bourgeois *Optimization of the Logical Topology for Mobile MEMS Networks,* International Journal of Network and Computer Applications(JNCA), Elsevier, Accpted To Appear, 2014.

[58] Y. Lin, A. Asakura, T. Fukuda, F. Arai *Design and fabrication of miniaturized force sensor with quartz crystal resonators.* IEEE/RSJ International Conference on Intelligent Robots and Systems, October 29 - November 2, 2007, Sheraton Hotel and Marina, San Diego, California, USA, 2007

[59] D. Little and J. Walter. *Using hexagonal metamorphic robots to form temporary bridges* . In Proc. of the IEEE International Conference on Intelligent Robots and Systems, pages 26522657, 2005.

[60] B.T. Loo, T. Condie, M. Garofalakis, D.E. Gay, J.M. Hellerstein,P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. *Declarative networking : language, execution and optimization.* In Proc. of the 2006 ACM SIGMOD int. conf. on Management of data, pages 97108, New York, NY, USA, ACM Press,2006.

[61] H. Lund, R. Larsen, and E. Østergaard *Distributed control in selfreconfigurable robots.* In Proceedings, 5th International Conference on Evolvable Systems : From Biology to Hardware (ICES03), pages 296307, Trondheim, Norway, 2003

[62] M. Mamei, A. Roli, F. Zambonelli, *Emergence and Control of Macro Spatial Structures in Perturbed Cellular Automata, and Implications for Pervasive Computing Systems,* IEEE Transactions on Systems, Man, and Cybernetics, 36(5), May 2005.

[63] M. Mamei, M. Vasirani, F. Zambonelli, *Experiments of Morphogenesis in Swarms of Simple Mobile Robots*, Journal of Applied Artificial Intelligence, 8(9-10) :903-919, Oct. 2004.

[64] M. Mehyar, D. Spanos, J. Pongsajapan, S. Low, and R. Murray. *Asynchronous Distributed Averaging on Communication Networks..* IEEE Transactions of Networking, August 2007.

[65] Y. Miao, G. Yan, and Z. Lin. *A distributed reconfiguration strategy for target enveloping with hexagonal metamorphic modules.* In Proc. of the IEEE International Conference on Robotics and Automation, pages 48044809, 2011.

[66] S. Matysik and J. Walter. *Using a pocket-filling strategy for distributed reconfiguration of a system of hexagonal metamorphic robots in an obstacle-cluttered environment* . In Proc. of the IEEE International Conference on Robotics and Automation, pages 42654272, 2009.

[67] P. Minet, S. Mahfoudh, *Energy, bandwidth and time efficiency in data gathering applications*, IFIP Wireless Days, Paris, France, December 2009.

[68] Y. Miao, G. Yan, and Z. Lin. *A distributed reconfiguration strategy for target enveloping with hexagonal metamorphic modules.* In Proc. of the IEEE International Conference on Robotics and Automation, pages 48044809, 2011.

[69] A. Milella, G. Reina and R. Siegwart *Computer Vision Methods for Improved Mobile Robot State Estimation in Challenging Terrains.* Journal of Multimedia (JMM), Vol. 1, No. 7, November/December, pp. 49-61, 2006

[70] S. Murata, H. Kurokawa, and S. Kokaji *Self-assembling machine.* In Proc. of IEEE Intl. Conf. on Robotics and Automation, pages 441-448, 1994.

[71] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji. *3-d self-reconfigurable structure.* SIn Proc. of IEEE Intl.Con. Robotics and Automation, pages 432439, 1998.

[72] R. Nagpal, A. Kondacs, and C. Chang *Programming methodology for biologically-inspired self-assembling systems..* In Proceedings, AAAI Spring Symposium on Computational Synthesis : From Basic Building Blocks to High Level Functionality, 2003

[73] A. Pamecha, I. Ebert-Upho, and G. Chirikjian. *Useful metrics for modular robot motion planning.* IEEE Transactions on Robotics and Automation,13(4) :531545, 1997.

[74] D. Paola, A. Milella, G. Cicirelli, and A. Distante *An Autonomous Mobile Robotic System for Surveillance of Indoor Environments.* International Journal of Advanced Robotic Systems, ISSN : 1729-8806, 2010.

[75] Ra. Ravichandran, G.Gordon, and S. C. Goldstein. *A Scalable Distributed Algorithm for Shape Transformation in Multi-Robot Systems.* Proceedings of the IEEE International Conference on Intelligent Robots and Systems, October, 2007.

[76] D. Rus and M. Vona. *Crystalline robots :Self-reconfiguration with compressible unit modules.* Autonomous Robots Journal, special issue on self-reconfigurable robots, Vol. 10, No. 1, pages 107124, Jan. 2001

[77] M. De Rosa, S. C. Goldstein, Peter Lee, J. D. Campbell, and P. Pillai. *Scalable Shape Sculpting via Hole Motion : Motion Planning in Lattice-Constrained Module Robots.* In Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA 06), May, 2006.

[78] M. De Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai. *Programming Modular Robots with Locally Distributed Predicates.* In Proceedings of the IEEE International Conference on Robotics and Automation ICRA 08, 2008.

[79] D. Rus and M. Vona. *Self-reconfiguration planning with compressible unit modules.* In Proc. of IEEE Intl. Conf. on Robotics and Automation, pages 2513-2520, 1999.

[80] E. Sahin. *Swarm robotics : from sources of inspiration to domains of application.* in Revolutionary Aerospace Systems Concepts Academic Linkage Forum, 2006, pp. 115.SAB 2004 International Workshop (Revised Selected Papers) E. Sahin and W. M. Spear (Eds.), Lecture Notes in Computer Science 3342, Springer, 2005

[81] M. L. Sichitiu and C. Veerarittiphan *Simple, accurate time synchronization for wireless sensor networks.*In Proceedings of the Wireless Communication and Networking Conference, pages 12661273, 2003

[82] B. Salemi, W. Shen, and P. Will *Hormone controlled metamorphic robots.* In Proceedings, IEEE Int. Conf. on Robotics and Automation, pages 41944199, Seoul, Korea, 2001

[83] W.M Shen, B. Salemi, and P. Will *Hormone-based control for self-reconfigurable robots.* In Proceedings, Int. Conf. on Autonomous Agents,pages 18, Barcelona, Spain, 2000

[84] W.M Shen, B. Salemi, and P. Will *Hormones for self-reconfigurable robots.*In Proceedings, Int. Conf. on Intelligent Autonomous Systems (IAS-6), pages 918925, Venice, Italy, 2000

[85] W.M Shen, B. Salemi, and P. Will *Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots.* IEEE Transactions on Robotics and Automation, 18(5) :700712, 2002

[86] R. Soua, L. Saidane, P. Minet, *Sensors deployment enhancement by a mobile robot in wireless sensor networks*, IEEE ICN 2010, Les Menuires, France, April 2010.

[87] K.Stoy, R.Nagpal, *Self-reconguration using Directed Growt.* 7th International Symposium on Distributed Autonomous Robotic Systems (DARs), France, June23-25, 2004.

[88] J. Suh, S. Homans, and M. Yim, *Telecubes : mechanical design of a module for self-reconfigurable robotics.* . In Proceedings, IEEE International Conference on Robotics and Automation (ICRA02), volume 4, pages 4095 4101, Washington, DC, USA, 2002

[89] K. Støy, R.Nagpal *Self-Reconfiguration Using Directed Growth.* Proc., Int. conf. on distributed autonomous robot systems (DARS-04), pp. 1-10, Toulouse, France, 2004.

[90] K. Støy *Controlling Self-Reconfiguration using Cellular Automata and Gradients.*Proc., 8th Int. conf. on intelligent autonomous systems, pp. 693-702, Amsterdam, The Netherlands, 2004.

[91] K. Støy *Using cellular automata and gradients to control self-reconfiguration.*Robotics and Autonomous Systems 54(2) : 135-141, 2006

[92] S. Vassilvitskii, M. Yim, and J. Suh *A complete, local and parallel reconfiguration algorithm for cube style modular robots.* In Proceedings, IEEE International Conference on Robotics and Automation (ICRA02), volume 1, pages 117122, Washington, DC, USA, 2002

[93] J. Walter, J. Welch, and N. Amato, *Distributed Reconfiguration of Metamorphic Robot Chains.* in Proceedings of the Nineteenth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2000), Portland, Oregon, 2000

[94] J. Walter, J. Welch, and N. Amato, *Distributed Reconfiguration of Metamorphic Robot Chains.*Springer-Verlag Journal on Distributed Computing, vol. 17, pp. 171189, 2004.

[95] J. Walter, J. Welch, and N. Amato, *Concurrent metamorphosis of hexagonal robot chains into simple connected configurations.*IEEE Transactions on Robotics and Automation, Vol. 18, No. 6, pp. 945-956, 2002

[96] J. Walter, B. Tsai, and N. Amato *Choosing good paths for fast distributed reconfiguration of hexagonal metamorphic robots.* In Proceedings,IEEE International Conference on Robotics and Automation (ICRA02), volume 1, pages 102109, Washington, DC, USA, 2002

[97] B. Warneke, M. Last, B. Leibowitz, and K.S.J Pister *Smart Dust : Communicating with a Cubic-Millimeter Computer*, Computer Magazine, pp. 44-51, 2001.

[98] A. Ward, A. Jones, A. Hopper, *A New Location Technique for the Active Office.* IEEE Personal Communications Magazine 4 :42-7, 2002.

[99] S. Wong and J. Walter, *Deterministic Distributed Algorithm for Self-Reconfiguration of Modular Robots from Arbitrary to Straight Chain Configurations.* Proc. the IEEE International Conference on Robotics and Automation, ICRA 2013, May 2013

[100] M. Yim, J. Lamping, E. Mao, and J. G. Chase. *Rhombic dodecahedron shape for self-assembling robots.* SPL TechReport P9710777, Xerox PARC, 1997.

[101]

[102] M. Yim, *New locomotion gaits.* International Conference on Robotics and Automation (ICRA 94), pages 2508 2514, San Diego, California, USA, 1994

[103] M. Yim, Y. Zhang, K. Roufas, D. Duff, and C. Eldershaw *Connecting and disconnecting for chain self-reconfiguration with polybot.* IEEE/ASME Transactions on Mechatronics, 7(4) :442451, 2002

[104] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian. *Modular self-reconfigurable robot systems.* EEE Robotics and Automation, 14(1) :4352, 2007.

[105] M. Yim, Y. Zhang, and D. G. Du. *Modular robots.* IEEE Spectrum, 39(2) :3034, 2002.

[106] M. Yim, S. Homans, and K. Roufas *Climbing with snake-like robots.* Proceedings, IFAC Workshop on Mobile Robot Technology, pages , Jejudo, Korea, 2001

[107] F. Zambonelli, M.P. Gleizes, M. Mamei, R. Tolksdorf, *Spray Computers : Explorations in Self-Organization*, Journal of Pervasive and Mobile Computing, Elsevier, Vol. 1, p. 1-20, 2005.

[108] Carlo Zaniolo, Natraj Arni, and KayLiang Ong. *Negation and aggregates in recursive rules :the LDL++ approach.* In DOOD, pages 204221, 1993

[109] Y. Zhang, M. Yim, J. Lamping, and E. Mao. *Distributed control for 3D shape metamorphosis.* To appear in Autonomous Robots Journal, 1999

[110] *http ://www.pittsburgh.intel-research.net/dprweb/*

Résumé :

Les microrobots MEMS sont des éléments miniaturisés qui peuvent capter et agir sur l'environnement. Ils ont une taille trés petite, une faible capacité de mémoire et une capacité énergétique limitée. Les microrobots MEMS continuent d'accroître leur présence dans notre vie quotidienne. En effet, ils peuvent effectuer plusieurs missions et tâches dans une large gamme d'applications telles que la localisation d'odeur, la lutte contre les incendies, le service médical, la surveillance, le sauvetage et la sécurité. Pour faire ces tâches et missions ils doivent appliquer des protocoles de redéploiement afin de s'adapter aux conditions du travail. Ces algorithmes doivent être efficaces, évolutifs, robustes et utilisent seulement les informations locales. Le redéploiement pour les microrobots MEMS mobiles actuellement nécessite un système de positionnement et une carte (positions prédéfinies) de la forme cible. La solution traditionnelle de positionnement comme l'utilisation GPS consomme beaucoup d'énergie. Aussi, l'utilisation d'une solution de positionnement algorithmique avec les techniques de multilatération comporte toujours des problèmes à cause aux erreurs dans les coordonnées obtenues. Dans la littérature, si nous voulons une auto-reconfiguration de microrobots en une forme cible constituée de P positions, chaque microrobot doit avoir une capacité de mémoire de P positions pour les sauvegarder. Par conséquent, si P est en milliers ou des millions, chaque nœud doit avoir une capacité de mémoire de positions en milliers ou millions. Par conséquent, ces algorithmes ne sont pas évolutifs. Dans cette thèse on propose des protocoles de reconfiguration oú les nœuds ne sont pas conscients de leurs positions dans le plan et n'enregistrent aucune position de la forme cible. En d'autres termes, les nœuds ne stockent pas au départ les coordonnées qui construisent la forme cible. Par conséquent, l'utilisation de mémoire pour chaque nœud est considérablement réduite à une complexité constante. L'objectif des algorithmes distribués proposés est d'optimiser la topologie logique du réseau des MEMS microrobots mobiles afin de chercher une meilleure complexité pour le nombre de messages échangés et une communication peu coûteuse. Ces solutions sont complétement distribués. On montre pour la reconfiguration d'une chaine en un carré comment gérer la dynamicité du réseau pour sauvegarder l'énergie, on étudie comment utiliser le parallélisme en mouvement pour optimiser le temps d'exécution et le nombre de mouvements. Ainsi, on propose une autre solution oú la topologie physique initiale peut être n'importe quelle configuration initiale. Avec ces solutions les nœuds peuvent exécuter l'algorithme indépendamment du lieu oú ils sont déployés, parce que l'algorithme est indépendant de la carte de la forme cible. En outre, ces solutions cherchent à atteindre la forme de la cible avec une quantité minimale de mouvement.

Mots-clés : Microrobots MEMS, Auto-reconfiguration, Redéploiement, Algorithmes distribués, Algorithmes parallèles, Topologie logique, Énergie, Mobilité

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex

■ tél. +33 (0)3 81 66 66 02 ■ **ed-spim@univ-fcomte.fr** ■ www.ed-spim.univ-fcomte.fr

UFC UNIVERSITÉ DE FRANCHE-COMTÉ