UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

LE TEMPS DU RENDEZ-VOUS SYNCHRONE DANS LES TRÈS GRANDS ENVIRONNEMENTS

MÉMOIRE PRÉSENTÉ
COMME EXIGENCE PARTIELLE
MAÎTRISE EN SCIENCE ET TECHNOLOGIES DE L'INFORMATION

PAR

JALAL AMAROF

SOUS LA SUPERVISION DU PROFESSEUR ANDRZEJ PELC

FÉVRIER 2019

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

LE TEMPS DU RENDEZ-VOUS SYNCHRONE DANS LES TRÈS GRANDS ENVIRONNEMENTS

Jalal Amarof

pour l'obtention du grade de maître ès science (M.Sc.)

a été évalué par un jury composé des personnes suivantes :

Dr Andrzej Pelc ..... Directeur de recherche
Dr Mohand Said Allili ..... Président du jury
Dr Jurek Czyzowicz ..... Membre du jury

Projet de mémoire accepté le : .....

À ALLAH Le Très-Haut,le très miséricordieux qui ma donné la volonté et le courage pour la réalisation de ce travail.

À ma mère Elrami Hajjou qui grâce à ces sacrifices, son amour et sa bénédiction j'ai pu réaliser ce travail.

À mon père Amarof Mohammed (paix à son âme) dont la mémoire m'a toujours donné force et inspiration.

À ma femme Jamila pour sa patience, sacrifices et amour.

À ma soeur Dr. Khadija et son mari François Guyot pour leurs soutien.

À mon frère Lahsen qui m'a aidé à écrire mon premier algorithme à l'âge de dix ans, et a allumé ainsi la flamme d'une passion.

À mes frères et soeurs Aziza, AbdelAli, Drissia, Houssaine, Mostapha, Karim et Mouly Ali.

À mon fils Omar et mes deux filles Sara et Lina, dont la présence et les éclats de rire sont une grande source de motivation

À Samir Elouasbi pour son soutien et aide précieuse.

Remerciements

Je tiens à remercier mon directeur de recherche, Professeur Andrzej Pelc, pour ça confiance et sa patience, ainsi que pour ça disponibilité et ces conseils toujours pertinents.
Je remercie également les membres de mon jury pour le temps et l'attention qu'ils ont accordé à mon travail.
Je tiens aussi à remercier Samir Elouasbi, Issam Boutahar, Othmane Himadi et ma soeur Dr. Khadija Amarof pour leur contribution et aide.
Enfin, je remercie tous mes amis qui ont contribué à ce travail par leur soutient et leur conseil.

Merci à vous tous

Table des matières

Résumé

Ce mémoire aborde le sujet du rendez-vous synchrone de deux agents mobiles avec des traces distinctes (jetons). Les deux agents, identifiés par des étiquettes différentes, sont placés sur des nœuds marqués par des jetons de couleurs différentes dans un graphe anonyme. L'exploration de ce dernier est nécessaire pour avoir une carte complète du graphe et, par la suite, les agents doivent se rencontrer dans le même nœud. Dans une ronde, l'agent mobile peut aller dans un des nœuds adjacents à son nœud actuel ou rester immobile dans ce dernier. Nous cherchons un algorithme déterministe qui permet de faire le rendez-vous dans un temps optimal.

mots clés : agent mobile, rendez-vous, traces distinctes, algorithme déterministe.

Abstract

This thesis addresses the subject of the synchronous rendezvous of two mobile agents with distinct traces (tokens). The two agents, identified by different labels, are placed at nodes marked by different tokens in an anonymous graph. The exploration of the latter is necessary to have a complete map of the graph and subsequently, the agents must meet at the same node. In a round, a mobile agent can go to one of the nodes adjacent to its current node or remain motionless in the current location. We seek a deterministic algorithm that accomplishes rendezvous in optimal time.

Keywords: mobile agent, rendezvous, distinct traces, deterministic algorithm.

Chapitre 1

Introduction

Chacun de nous a déjà vécu une expérience semblable à celle-ci : En magasinant entre les rayons d'un grand supermarché, vous vous rendez compte que vous avez perdu l'ami qui vous accompagne. Vous êtes pressé de retourner chez vous et sans moyen de communication, vous pensez à la meilleure façon de le retrouver. Vous avez le choix entre le chercher à travers les rayons ou rester sur place à attendre qu'il vous trouve, et si votre ami décide lui aussi d'attendre, vous risquez de passer toute la journée au supermarché.

Le problème du rendez-vous peut être plus complexe que l'exemple ci-dessus. Imaginez deux robots dans un grand environnement tel que la planète Mars, sans communication et qui doivent se rencontrer pour partager une information cruciale à la réussite de leur mission. Dans le domaine de l'informatique, plus précisément dans le calcul distribué, le problème du rendez-vous est souvent associé aux agents mobiles, qui sont des entités logicielles qui se déplacent entre les ordinateurs d'un réseau informatique et qui peuvent échanger de l'information ou exécuter des tâches. Dans d'autres applications, des robots mobiles se déplacent dans un terrain contenant possiblement des obstacles et ils doivent se rencontrer pour échanger des échantillons du sol pris et mesurer la contamination du terrain.

Dans cette recherche, nous sommes motivé par la question suivante : Quel est le temps optimal du rendez-vous synchrone de deux agents mobiles dans un très grand environnement ? Les algorithmes proposés dans la littérature ont comme modèle différents environnements géométriques avec obstacles (terrain, plan, graphe), mais contenant souvent une borne supérieure sur la grandeur du terrain, et la complexité de l'algorithme

dépend de cette borne. Par contre, nous voulons concevoir des algorithmes de rendez-vous dont la complexité ne dépend pas de la taille de l'environnement.

Afin de répondre à cette question, nous proposons différents algorithmes déterministes synchrones pour différents environnements. Nous analysons en détail les algorithmes pour chaque environnement. Dans ce mémoire, nous commençons par un algorithme qui résout le problème du rendez-vous sur un graphe arbitraire avec un degré maximal Δ, puis nous présentons un algorithme qui résout le problème du rendez-vous sur une grille infinie orientée. Enfin nous présentons un algorithme qui résout le problème de l'approche dans un plan euclidien.

Ce mémoire est organisé comme suit :

- Le chapitre 2 comporte une revue de la littérature concernant le problème du rendez-vous.
- Le chapitre 3 présente le modèle pour les graphes arbitraires avec un degré maximal Δ et l'algorithme de rendez-vous pour ce modèle, ainsi que son analyse.
- Le chapitre 4 présente le modèle pour les grilles infinies orientées et l'algorithme de rendez-vous pour ce modèle, ainsi que son analyse.
- Le chapitre 5 présente le modèle pour le plan euclidien et l'algorithme de l'approche pour ce modèle, ainsi que son analyse.
- Le chapitre 6 présente le logiciel qui permet la simulation des algorithmes des chapitre 4 et 5.
- Le chapitre 7 présente la conclusion du mémoire, ainsi que les problèmes ouverts.
- Le mémoire se termine par la bibliographie.

Chapitre 2

Revue de la littérature

2.1 Introduction

Le problème du rendez-vous est largement étudié dans la littérature. Il est décrit comme suit : Deux agents mobiles placés dans deux endroits différents d'un environnement doivent se rencontrer [23]. La résolution de ce problème concerne une large panoplie d'applications, allant du comportement animalier, par le domaine de la robotique, jusqu'au calcul distribué dans les réseaux informatiques.

La première étude sur le problème du rendez-vous est celle de Schelling [24] en 1960. Il présente le problème sous une forme très simple : dans une ville, deux personnes veulent se rencontrer et chacune a une seule tentative. [24] a démontré qu'un endroit spécifique connu préalablement par les deux personnes peut aider à résoudre le problème du rendez-vous. Dans le domaine de l'informatique, la connaissance de l'environnement est aussi importante pour l'étude du rendez-vous que la façon dont les agents mobiles se déplacent dans celui-ci. Les algorithmes du problème du rendez-vous sont classés en deux catégories majeures, algorithmes déterministes et algorithmes aléatoires. Compte tenu du fait que la littérature est très vaste, Dans cette revue de littérature, nous aborderons seulement les algorithmes déterministes.

Briser la symétrie est le principal problème que les algorithmes déterministes essaient de régler pour réaliser le rendez-vous. Prenons l'exemple d'un environnement symétrique tel qu'un graphe avec deux nœuds liés par une arête, où les deux agents mobiles sont placés dans des positions différentes. Si les deux agents commencent en même temps et exécutent le même algorithme déterministe, ils conservent la même distance initiale après chaque déplacement et ils ne peuvent donc pas se rencontrer dans un nœud.

Dans cette revue de littérature, nous allons détailler différentes solutions qui ont permis de briser cette symétrie. Deux catégories sont abordées : les algorithmes déterministes synchrones et les algorithmes déterministes asynchrones. Nous allons examiner le rendez-vous déterministe asynchrone dans deux environnements différents : le rendez-vous dans les réseaux et le rendez-vous dans les terrains. De plus nous considérerons le rôle des pannes dans la réalisation du rendez-vous. Enfin, nous allons aborder un problème lié à celui du rendez-vous : l'exploration des graphes et des terrains.

2.2 Le rendez-vous déterministe synchrone dans les réseaux

Briser la symétrie est la tâche la plus importante pour un algorithme de rendez-vous déterministe synchrone. Il existe trois façons de le faire [23] :

La première façon est d'assigner une étiquette différente à chaque agent. Considérons l'exemple de l'anneau de grandeur connue où l'on attribue une étiquette numérique L différente à chacun des agents mobiles. Si chaque agent exécute l'algorithme qui consiste à faire L tours de l'anneau qu'il s'immobilise, alors l'agent qui a l'étiquette la plus grande va nécessairement rencontrer l'autre agent, qui devient immobile.

La deuxième façon de briser cette symétrie est de marquer les nœuds du graphe par un jeton ou d'utiliser un tableau de bord. En ce qui concerne le tableau de bord, il s'agit d'une mémoire que possède chaque nœud et que chaque agent peut effacer ou dans laquelle il peut lire et écrire tout en respectant sa taille maximale. Pour les jetons, il existe deux types, un jeton immobile qui ne peut être enlevé une fois placé par un agent et un jeton mobile qui peut être déplacé par un autre agent ou l'agent qui l'a placé.

Enfin, la troisième façon est d'exploiter la non-symétrie du graphe par exemple une ligne comprenant un nombre impair de nœuds où le rendez-vous peut se faire au milieu, ou d'exploiter la non-symétrie des positions initiales des agents si le réseau est symétrique.

Le mouvement des agents mobiles est un aspect important des algorithmes déterministes. Un des deux types de mouvement que nous allons examiner dans cette section est le mouvement synchrone. Celui-ci est le déplacement des agents à travers les nœuds du graphe par rondes synchrones. Durant une ronde, chaque agent peut soit rester sur place, soit se déplacer vers un nœud adjacent. Le rendez-vous doit se faire dans un nœud

et non dans une arête. Les agents qui se croisent sur une arête ne se rendent pas compte du croisement.

Dessmark, Fraigniaud, Kowalski et Pelc [11] nous proposent un algorithme déterministe synchrone qui permet de résoudre le problème de rendez-vous dans un graphe connexe quelconque. Deux variantes du mouvement synchrone des agents sont proposées, un départ simultané où les agents commencent à exécuter l'algorithme au même moment et un départ arbitraire où un adversaire choisit à quel moment aura lieu le début de l'exécution de l'algorithme de chaque agent. Les agents se déplacent dans un graphe anonyme (les deux agents ignorent la topologie du graphe) et chaque agent porte une étiquette l différente de l'étiquette de l'autre agent. Chaque agent connaît son étiquette et ignore l'étiquette de l'autre.

Les auteurs du [11] présentent d'abord un algorithme qui s'applique aux arbres et qui permet le rendez-vous dans un temps de $O(n+\log l)$, où l est la plus petite étiquette pour n'importe quel arbre d'ordre n, et même pour un départ arbitraire. Ils prouvent aussi, pour la classe de tous les arbres d'ordre n, qu'il n'existe aucun algorithme déterministe qui permette de résoudre le problème du rendez-vous dans un temps meilleur que $O(n+\log l)$ même si le départ des agents est simultané.

Les auteurs présentent un deuxième algorithme qui travaille dans un temps de $\Theta(D \log l)$, qui est optimal et qui s'applique à n'importe quel anneau à n nœuds pour un départ simultané. Pour un départ arbitraire, les auteurs proposent deux algorithmes : le premier travaille dans un temps de $O(n \log l)$ quand les agents connaissent le nombre n de nœuds qui forment l'anneau, et le deuxième dans un temps de $O(l\tau + ln^2)$ dans le cas contraire, τ étant la différence de temps qui existe entre le départ du premier agent et celui du deuxième agent. Ils prouvent que $\Omega(n + D \log l)$ est une borne inférieure sur le temps d'un rendez-vous avec départ arbitraire, où D est la distance entre les positions initiales des agents. Enfin, les auteurs présentent un algorithme pour n'importe quel graphe dans un temps de $O(n^5\sqrt{\tau \log l} \log n + n^{10} \log^2 n \log l)$.

Ta-Shma et Zwick [25] soumettent une meilleure solution pour assurer le rendez-vous dans n'importe quel graphe. Le temps de leur algorithme ne dépend pas de τ : il est de $\tilde{\mathrm{O}}(n^5)$ [1]. De plus, ils proposent un autre algorithme ayant un temps d'au plus $\tilde{\mathrm{O}}(d^2n^3)$, où d est le degré maximal du graphe.

Les solutions envisagées par [11] et [25] considèrent que les agents mobiles utilisent une mémoire interne illimitée, ce qui est impossible dans le cadre des applications réelles

1. $\tilde{\mathrm{O}}$ signifie un O à un facteur logarithmique près.

du problème de rendez-vous. Le problème est abordé par Czyzowicz, Kosowski et Pelc [8], qui émettent une solution pour contrer le problème du rendez-vous en prenant en considération la taille de la mémoire des agents mobiles. Ils répondent à une question simple : Pour n'importe quel graphe, quelle est la taille minimale de la mémoire des agents mobiles anonymes (sans étiquette) qui assure un rendez-vous déterministe synchrone ? Les auteurs prouvent qu'avec un graphe de n nœuds et des positions initiales non symétriques des agents, la mémoire minimale est de $\Theta(\log n)$, peu importe si les agents mobiles ont un départ simultané ou arbitraire. Ils prouvent aussi que $\Omega(\log n)$ est une borne inférieure.

Miller et Pelc [21] s'interrogent sur la taille minimale de l'information qu'on peut fournir aux agents mobiles portant des étiquettes différentes (la même information pour les deux agents) afin d'assurer le rendez-vous. Ils prouvent que, pour un graphe de n nœuds et deux agents avec des étiquettes distinctes d'au plus L et une distance initiale D qui les sépare, l'information minimale pour qu'ils puissent réaliser un rendez-vous déterministe synchrone dans un temps de $O(D)$ est de taille $\Theta(D \log(n/D) + \log\log L)$, peu importe si les agents mobiles ont un départ simultané ou arbitraire.

Dans une autre étude [22], Miller et Pelc s'interrogent sur le compromis à faire entre la taille de l'information fournie aux agents mobiles et le coût (nombre d'arêtes traversées) du rendez-vous et de la chasse au trésor. Le problème de la chasse au trésor est une variante du problème du rendez-vous. Comme son nom l'indique, il s'agit d'un seul agent mobile qui doit rejoindre une cible, nommée trésor, située dans un nœud inconnu du graphe. Le but des auteurs de [22] est de trouver la taille minimale de l'information fournie aux agents mobiles afin de résoudre le problème du rendez-vous et de la chasse au trésor avec un coût C dans un graphe de e arêtes. Ils proposent une solution pour n'importe quel graphe dont les nœuds ont des étiquettes distinctes. Les agents n'ont pas d'étiquettes, mais l'information fournie peut être différente pour chacun d'eux. Ils prouvent une borne supérieure de $O(D \log(D.\frac{e}{C}) + \log\log e)$ et une borne inférieure de $\Omega(D \log \frac{e}{C})$ sur la taille de l'information nécessaire. Pour les arbres, ils prouvent une borne supérieure de $O(D \log(\frac{e}{C}) + \log\log e)$ et une borne inférieure de $\Omega(D \log \frac{e}{C})$ sur la taille de l'information.

2.3 Le rendez-vous déterministe asynchrone

Le scénario synchrone est basé principalement sur le mouvement des agents mobiles par rondes synchrones. Dans cette section, nous verrons un autre scénario qui rend le problème du rendez-vous plus compliqué à résoudre : le scénario asynchrone. Dans ce scénario, l'agent choisit le nœud suivant à visiter dans le cas des graphes et le point cible suivant dans le cas des terrains, mais la vitesse du mouvement de l'agent peut être contrôlée par un adversaire, contrairement au scénario synchrone (dans les graphes), où le mouvement instantané des agents est assuré à chaque ronde en donnant à l'adversaire l'unique possibilité de contrôler les moments de départ des agents.

Dans cette section, nous examinerons le rendez-vous déterministe asynchrone dans deux environnements différents : les réseaux et les terrains. Également, nous allons voir l'impact des pannes sur le rendez-vous déterministe asynchrone.

2.3.1 Le rendez-vous dans les réseaux

Si l'adversaire contrôle la vitesse de déplacement des agents dans un graphe, le rendez-vous dans un nœud peut être impossible. C'est pour cela que le scénario asynchrone permet aussi la rencontre des agents à l'intérieur d'une arête. Les auteurs de [20] élaborent des algorithmes déterministes asynchrones du rendez-vous pour une ligne infinie, pour l'anneau et pour des graphes arbitraires. L'algorithme assure le rendez-vous de deux agents mobiles en utilisant une étiquette différente pour chaque agent et en supposant une distance initiale D entre les deux. Le coût de l'algorithme est le nombre maximal d'arêtes traversées par les deux agents. Les auteurs proposent un algorithme pour une ligne infinie ayant un coût de $O(D|Lmin|^2)$, où la distance initiale D est connue, et un algorithme ayant un coût $O((D + |Lmax|)^3)$ quand la distance initiale D est inconnue, où $|Lmin|$ est l'étiquette des deux agents la plus courte et où $|Lmax|$ l'étiquette la plus longue. En ce qui concerne l'anneau, les auteurs conçoivent un algorithme ayant un de temps $O(n|Lmin|)$, où n est le nombre de nœuds connu de l'anneau, et un autre algorithme ayant un temps de $O(n|Lmax|)$ quand le nombre de nœuds n est inconnu. Pour les graphes arbitraires, ils montrent que le rendez-vous est possible si les agents connaissent une borne supérieure sur le nombre de nœuds et ils proposent un algorithme ayant un coût de $O(D|Lmin|)$, où D est la distance entre les positions initiales des deux agents, si ces derniers connaissent la topologie du graphe et les nœuds du départ. Les auteurs soulèvent une question sur la faisabilité du rendez-vous détermi-

niste asynchrone pour n'importe quel graphe et sans connaître une borne supérieure de la taille du graphe.

Czyzowicz, Labourel et Pelc [10] répondent par l'affirmative à cette question. Ils élaborent un algorithme déterministe asynchrone qui assure le rendez-vous pour n'importe quel graphe et sans connaître une borne supérieure sur la taille du graphe. Ils prouvent que les deux agents ont juste besoin de connaître leurs propres étiquettes. Cependant, d'après les auteurs, le coût de cet algorithme est exponentiel. Ils soulèvent une autre question sur la possibilité de construire un algorithme déterministe asynchrone du rendez-vous pour n'importe quel graphe à un coût polynomial. Trois ans plus tard, Dieudonné, Pelc et Villain [12] répondent par l'affirmative à cette question en élaborant un algorithme déterministe asynchrone qui permet le rendez-vous pour n'importe quel graphe en temps polynomial en taille du graphe et en longueur de la plus petite étiquette des deux agents.

Les algorithmes déterministes asynchrones précédents résolvent le problème du rendez-vous pour des agents mobiles portant des étiquettes différentes. Guilbault et Pelc [18] créent un algorithme déterministe asynchrone qui résout le problème du rendez-vous pour des agents anonymes (sans étiquette). Afin de briser la symétrie, l'algorithme utilise la position initiale des agents et la vue de chaque agent depuis cette position dans le graphe. La vue d'un agent est l'arbre infini des séquences de ports des nœuds qui forment les chemins possibles, où la racine est la position initiale de l'agent. D'après [18], l'algorithme assure le rendez-vous déterministe asynchrone pour des agents anonymes, pour toutes les positions initiales dans lesquelles le rendez-vous est faisable. Les auteurs prouvent que le rendez-vous des agents anonymes est faisable si et seulement si leurs positions initiales ont des vues différentes ou si leurs positions initiales sont liées par un chemin en forme de palindrome.

Enfin, les auteurs de [2] se sont intéressés au scénario d'un réseau de nœuds anonymes ayant des arêtes avec étiquettes, mais sans connaître la topologie du réseau et encore moins la position initiale des agents mobiles. Les solutions proposées s'appliquent aussi au problème de l'élection, qui est un problème semblable à celui du rendez-vous en matière de complexité et qui consiste à choisir un agent mobile comme chef d'un groupe d'agents. Pour r agents mobiles identiques et asynchrones, dans un réseau de nœuds anonymes de taille n où chaque nœud a un tableau de bord dans lequel les agents peuvent lire et écrire, les auteurs prouvent que les problèmes du rendez-vous et de l'élection n'ont aucune solution si le plus grand commun diviseur $pgcd(n, r) > 1$, et cela, peu importe si

les arêtes ont des étiquettes ou un sens de direction [2]. Par contre, si $pgcd(n, r) = 1$, alors une solution est possible seulement si les arêtes ont un sens de direction. La solution se base sur un mécanisme qui s'appelle *Dynamic Name Mutation*, pour assurer que chaque agent mobile a un nom unique. Chaque fois qu'un agent visite un nœud, il change de nom (en privé) en utilisant le sens de la direction de l'arête. Ainsi l'algorithme brise la symétrie du réseau anonyme.

2.3.2 Le rendez-vous dans les terrains

Les auteurs de [6] proposent un algorithme déterministe asynchrone pour résoudre le problème du rendez-vous pour n'importe quel terrain avec une borne. Leur solution considère deux agents mobiles (robots) asynchrones avec étiquettes, modélisés comme des points et qui doivent faire le rendez-vous dans un terrain modélisé comme polygone. Celui-ci peut contenir des obstacles sous forme de polygone. Les robots ont des compas qui peuvent être incohérents, mais les deux compas des deux agents mobiles ont la même orientation que celle des aiguilles d'une montre. La trajectoire du robot est décidée par le robot, mais la vitesse est contrôlée par un adversaire. Les auteurs de [6] soumettent leurs solutions en se basant sur trois facteurs importants : présence d'obstacle ou non, cohérence ou incohérence des compas des deux robots et, enfin, si les robots possèdent ou non une carte du terrain. Le coût du rendez-vous est la somme de la taille des trajectoires (au pire cas) effectuées pour faire le rendez-vous. Si les deux robots utilisent une carte du terrain, avec un compas cohérent, le coût du rendez-vous est de D, qui est la distance entre les positions initiales des deux robots, et cela, peu importe si le terrain a des obstacles ou non. Avec des compas incohérents, le coût est de D sans présence d'obstacles et de $\Theta(D|l|)$ avec présence d'obstacles, où $|l|$ est la longueur de l'étiquette la plus petite des deux robots. Si les deux robots n'utilisent pas une carte du terrain, avec des compas cohérents, le coût du rendez-vous est de $\Theta(P)$, où P est le périmètre du terrain, et cela, peu importe si le terrain a des obstacles ou non. Avec des compas incohérents, le coût est de $\Theta(P)$ sans présence d'obstacles et de $\Theta(P + x|L|)$ avec présence d'obstacles, où $|L|$ est la longueur de l'étiquette la plus grande des deux robots et x est le périmètre le plus grand d'un obstacle.

Les auteurs de [17] proposent un algorithme qui permet aux agents mobiles (robots) de se rassembler dans un même point (non défini à l'avance) du terrain en un temps

2. Le sens de direction est une propriété des graphes ayant des arêtes avec étiquettes et qui permet d'avoir une orientation du genre Nord, Sud, Est, Ouest.

fini, mais cette fois-ci, ils considèrent deux facteurs importants : les robots disposent d'une visibilité limitée et ils n'ont pas de mémoire interne. Par conséquent, les robots sont capables de voir seulement à une distance fixe V et ils ne se rappellent pas des observations et calculs faits durant les étapes précédentes (inconscient). Les robots sont asynchrones, anonymes et se déplacent dans le terrain selon un cycle *Attendre Regarde Calcule Bouge*. Le cycle commence par l'étape *Attendre*, où le robot est immobile. Ensuite arrive l'étape *Regarde*, où le robot est en mode observation. Grâce à des capteurs, il peut capturer instantanément les positions des autres robots. Il faut noter que le robot ne peut pas détecter si un seul robot ou plusieurs robots se trouvent dans le même point. Après l'étape *Regarde* arrive l'étape *Calcule*, où le robot exécute localement son algorithme déterministe pour décider ou non de se déplacer vers un point. L'exécution se fait sans connaître le résultat des calculs des mouvements précédents. Enfin, si le robot décide de se déplacer, il passe à l'étape *Bouge* où il se dirige vers le point de destination. Pour que le rassemblement (deux robots et plus) se fasse, il faut que les positions initiales des robots forment un graphe connexe, c'est-à-dire la taille de chacune des arêtes entre les nœuds du graphes doit être inférieure ou égale au rayon du champ de visibilité V du robot.

Les auteurs de [5] proposent aussi une solution pour le problème du rassemblement, mais cette fois-ci avec des suppositions plus restreintes. Comme dans le cas de [17], les robots se déplacent d'une façon asynchrone selon les cycles *Attendre Regarde Calcule Bouge* et sans mémoire interne (inconscient). Ils sont donc incapables de se rappeler les calculs et observations faits aux cycles précédents. Mais cette fois-ci, les robots sont désorientés, ils ne disposent d'aucun système de navigation commun. Les auteurs prouvent que le rassemblement est faisable pour tous $n > 2$ robots.

Les auteurs de [19] se sont intéressés à la cohérence nécessaire et suffisante des compas des deux robots inconscients afin de résoudre le problème de rassemblement dans un mode semi-synchrone et un mode asynchrone. Les deux robots se déplacent par les cycles *Attendre Regarde Calcule Bouge*. Le mode semi-synchrone est celui où l'exécution du cycle *Attendre Regarde Calcule Bouge* est instantané. Le robot ne sera jamais observé quand il est en mouvement. Par contre, le mode asynchrone, c'est quand le cycle *Attendre Regarde Calcule Bouge* d'un robot peut interférer avec celui de l'autre robot. Ainsi, quand un robot est en mouvement, il peut être observé par l'autre robot. Chaque robot possède un système de coordonnées local basé sur un compas comprenant certaines déviations ϕ. La solution proposée par [19] se fonde sur deux types de compas :

compas statique et compas dynamique. Un compas statique comporte une déviation constante alors qu'un compas dynamique peut avoir une déviation arbitraire après ou avant chaque cycle *Attendre Regarde Calcule Bouge*. Pour le mode semi-synchrone, les auteurs de [19] prouvent que le rassemblement des deux robots est impossible s'ils ont un compas statique ayant une déviation de $\phi \geq \pi/2$, et impossible avec un compas dynamique ayant une déviation de $\phi \geq \pi/4$. Toujours dans le mode semi-synchrone, les auteurs démontre que le rassemblement des deux robots est possible s'ils ont un compas statique ayant un déviation de $\phi < \pi/2$, et possible avec un compas dynamique ayant une déviation de $\phi < \pi/4$. Pour le mode asynchrone, le rassemblement des deux robots est impossible s'ils ont un compas statique ayant une déviation de $\phi \geq \pi/2$, et impossible avec un compas dynamique ayant une déviation de $\phi \geq \pi/4$. Toujours dans le mode asynchrone, les auteurs prouvent que le rassemblement des deux robots est possible s'ils ont un compas statique ayant une déviation de $\phi < \pi/2$, et possible avec un compas dynamique yant une déviation de $\phi < \pi/6$.

Enfin, une autre solution du problème du rassemblement proposée par [1] est basée sur le scénario suivant : il y a n robots sur un plan qui se déplacent en mode asynchrone et dans les cycles *Attendre Regarde Calcule Bouge*. Les robots sont anonymes (sans étiquette) et identiques sous forme de disques fermés. Chaque robot dispose d'un système de vision à 360 degrés et tous les robots se concordent sur le même axe d'orientation. Les auteurs de [1] proposent un algorithme qui permet le rassemblement de n robots en trois étapes : Dans la première étape, les robots forment une enveloppe convexe. Dans la deuxième étape, chaque robot arrive à voir les autres robots. Dans la dernière étape, les robots se rassemblent en gardant la formation de l'enveloppe convexe.

Dans une autre étude [9], les auteurs se sont intéressés au problème du rendez-vous asynchrone de deux agents mobiles équipés d'une mémoire limitée et qui se déplacent dans un terrain. Leur objectif était de vérifier la faisabilité du rendez-vous de deux agents mobiles avec et sans étiquettes. Le terrain est modélisé comme un polygone qui peut contenir des obstacles sous forme polygonale. Les deux agents sont présentés comme des points et devant se rencontrer. Les auteurs de [9] ont étudié deux variantes du rendez-vous : le rendez-vous exact où les deux points qui représentent les deux robots doivent se rencontrer et le rendez-vous approximatif (ε-rendez-vous) où les deux points doivent se rencontrer à une distance inférieure à ε. Pour se déplacer, chaque robot a un compas qui peut être incohérent, mais les compas respectifs des deux robots ont la même orientation que celle des aiguilles d'une montre. La trajectoire du robot est décidée par

le robot lui-même, mais la vitesse est contrôlée par un adversaire. Aussi l'adversaire peut arrêter l'agent mobile et même le faire avancer ou reculer. Pour les agents mobiles sans étiquettes, les auteurs démontrent que le rendez-vous exact est impossible si le terrain a une forme symétrique rotationnelle. Si le terrain est connu et ne contient pas de formes symétriques rotationnelles, alors le rendez-vous exact est possible. Les auteurs démontrent aussi que même si les agents mobiles portent des étiquettes, le rendez-vous exact est impossible pour un nombre arbitraire de sommets du polygone du terrain. Par contre, si le nombre de sommets k est connu, alors le rendez-vous exact est possible pour toutes les classes de polygones, même avec obstacles, à condition que ces derniers aient une forme polygonale triangulaire. Ils prouvent aussi que si le terrain est connu, le rendez-vous des agents avec étiquettes est toujours faisable. Enfin, pour le ε-rendez-vous, les auteurs prouvent que le rendez-vous est possible pour les agents sans étiquettes seulement si le terrain est sous forme de polygone régulier avec diamètre limité. Si le polygone est régulier avec un diamètre illimité, le ε-rendez-vous est impossible. Par contre, dans le cas où les agents mobiles auraient des étiquettes différentes, le ε-rendez-vous deviendrait possible même pour un polygone régulier avec diamètre illimité.

Une autre solution proposée par les auteurs de [16] apporte une contribution importante concernant le problème du rendez-vous des robots sans mémoire persistante (inconscients). Ils ajoutent deux extensions au modèle existant : agents mobiles avec mémoire constante et agents mobiles avec communication constante. Le modèle proposé par les auteurs de [16] est basé sur deux robots anonymes modélisés comme des points et qui bougent dans un terrain selon le principe *Attendre Regarde Calcule Bouge*. Chaque robot a son propre système de coordonnées et ces déplacements se font en mode asynchrone ou semi-synchrone. Pour la mémoire des robots, les auteurs proposent deux scénarios. Dans le scénario FState, chaque robot a une mémoire constante et persistante durant les cycles *Attendre Regarde Calcule Bouge*, mémoire appelée lumière et qui n'est visible que par le robot lui-même. Dans le scénario FComm, la lumière du robot n'est visible que par les autres robots, et elle est oubliée par ceux-ci après chaque étape *Calcule* du cycle *Attendre Regarde Calcule Bouge*. Le mouvement des robots peut être non rigide ou rigide. Un mouvement non rigide, c'est quand un adversaire empêche le robot d'atteindre sa destination : si la distance entre le robot et sa destination est inférieure ou égale à δ, alors le robot arrive à destination, sinon il avancera vers sa destination au moins pour une distance δ. Le mouvement rigide, c'est quand le robot atteint sa destination sans aucune interruption de la part de l'adversaire. Les auteurs

de [16] prouvent que dans le scénario FState, avec mouvement rigide des robots et un modèle semi-synchrone, le rendez-vous est possible avec six états de mémoire interne. Ils prouvent également que dans le scénario FComm, avec un mouvement rigide et un modèle asynchrone, le rendez-vous est possible avec communication de douze messages, et trois messages sont suffisants pour un modèle semi-synchrone. Ils établissent en outre que pour un mouvement non rigide, trois états de mémoire interne sont suffisants pour FComm en mode asynchrone. Enfin, les auteurs démontrent que pour un rendez-vous en temps optimal, c'est mieux de communiquer que de mémoriser l'information.

2.3.3 Le rendez-vous avec pannes

Dans la littérature, le problème du rendez-vous est largement étudié dans différents modèles et scénarios. Dans cette section, nous aborderons quelques solutions au problème du rendez-vous avec pannes. Les auteurs de [13] proposent une solution pour le rendez-vous asynchrone de k agents mobiles sans étiquettes dans un anneau avec n nœuds. La particularité de l'anneau, c'est qu'il peut être orienté ou non, et qu'il contient un trou noir. Un trou noir est un nœud qui détruit tout agent qui le visite, sans laisser de traces. Les agents sont conscients de l'existence du trou noir, mais ils ne connaissent pas sa position. Avec ce scénario, les auteurs de [13] se sont intéressés au nombre maximal d'agents mobiles qui assurent le rendez-vous, en sachant k ou n (au moins un doit être connu). Les auteurs de [13] prouvent que pour un anneau orienté avec n inconnu et k connu, le temps du rendez-vous est de $(k-1)$; pour n connu et k inconnu, le temps du rendez-vous est de $(k-2)$. Ils démontrent aussi que pour un anneau non orienté avec n inconnu et k connu, si k est impair, alors le temps du rendez-vous est de $(k-2)$; si k est pair, alors le temps du rendez-vous est de $\frac{k-2}{2}$. Toujours pour un anneau non orienté avec n connu et k inconnu, si k est impair ou n est pair, le temps du rendez-vous est de $(k-2)$; si k est pair ou n est impair, le temps du rendez-vous est de $\frac{k-2}{2}$. Les auteurs démontrent que ces résultats sont stricts, c'est-à-dire que le rendez-vous est impossible si l'on ajoute un seul agent.

Les auteurs de [4] proposent une solution pour un autre scénario du problème du rendez-vous avec pannes. Ils considèrent plusieurs agents mobiles sans étiquettes et qui doivent faire le rendez-vous dans un graphe anonyme quelconque. Le graphe peut contenir des arêtes défectueuses qui détruisent tout agent qui le visite sans laisser de traces. Si toutes les arêtes d'un nœud sont défectueuses, la solution du problème du rendez-vous est celle proposée précédemment par les auteurs de [13]. Les agents se déplacent d'une façon asynchrone et leurs positions initiales sont distinctes. Chaque nœud a un tableau de bord où les agents peuvent lire et écrire, et durant leur déplacement, un seul agent à la fois peut traverser une arête. Les auteurs de [4] prouvent qu'avec τ arêtes défectueuses et k agents, $(k - \tau)$ agents peuvent faire le rendez-vous. Ils prouvent également que le rendez-vous est impossible avec $(k - \tau)$ agents si les agents ne connaissent pas la taille du graphe, ou une borne supérieure B. Si B est connue, alors le problème du rendez-vous est résolu si $n \leq B \leq 2n$, où n est le nombre de nœuds du graphe. Ils démontrent de plus que si m est le nombre d'arêtes, alors le nombre de déplacements des agents est de $O(m(m + k))$, qui est aussi une borne inférieure. Enfin, ils démontre que le rendez-vous maximal (tous les agents se retrouvent au même nœud) est impossible.

2.4 L'exploration des graphes et des terrains

L'exploration des terrains inconnus par des robots a plusieurs applications importantes, surtout quand le terrain à explorer est dangereux ou inaccessible par les humains. Comme exemple, on peut citer l'exploration des fonds marins, les terrains après un accident nucléaire, les zones dangereuses à haut taux de radioactivité. Dans de nombreux cas, un robot doit inspecter un terrain inconnu et revenir à son point de départ. En raison des exigences d'économie d'énergie et de coût, la longueur de la trajectoire du robot doit être minimisée. Souvent, l'environnement à explorer est modélisé par un graphe, s'il s'agit d'un réseau de corridors d'un bâtiment ou d'une usine contaminée. Dans cette section, nous examinerons l'exploration des graphes et des terrains dans différents modèles et scénarios.

Les auteurs de [14] répondent à une question simple : Combien de robots inconscients peuvent explorer un graphe anonyme ayant la forme d'une ligne ? Ils considèrent le scénario suivant : un graphe avec n nœuds formant une ligne et chaque nœud doivent être visités au minimum par un robot. k robots identiques et inconscients se déplacent selon les cycles asynchrones *Regarde Calcule Bouge*. Durant la procédure *Regarde*, les robots peuvent détecter si plus d'un robot est présent dans un nœud, mais ils ne peuvent pas en déterminer le nombre exact. Durant la procédure *Bouge*, chaque robot peut soit rester immobile, soit se déplacer sur un nœud adjacent. L'inconscience des robots signifie que, durant la procédure *Calcule*, un robot peut utiliser seulement le résultat de la dernière procédure *Regarde*. Une fois l'exploration terminée, tous les robots doivent s'arrêter. Les auteurs de [14] prouvent que pour n nœuds, k robots et $k < n$, l'exploration est possible si et seulement si $k = 3$ ou $k \geq 5$ ou $k = 4$ et que n est impair. Ils proposent un algorithme d'exploration et ils prouvent que l'exploration est impossible pour les autres valeurs de k et de n.

Dans [15], les mêmes auteurs proposent une réponse à la même question de [14], mais cette fois pour un anneau : combien de robots inconscients peuvent explorer un graphe anonyme ayant la forme d'un anneau ? Ils considèrent le même scénario que celui du [14]. Ils prouvent que pour n nœuds, k robots et $k < n$, l'exploration est impossible si k et n ne sont pas relativement premiers. De plus, ils prouvent que l'exploration se fait en temps fini si $pgcd(n, k) = 1$ pour $k \geq 17$. Ils démontrent aussi que si $\rho(m)$ est le nombre minimal de robots qui peuvent explorer un anneau de n nœuds, alors $\rho(m)$ est de $\Theta(\log n)$.

Les auteurs de [3] ont étudié le problème de la construction de la carte d'un graphe anonyme par un robot. Depuis un nœud de départ, le robot doit parcourir le graphe, sans marquer ses arêtes, et retourner à son nœud de départ. Les auteurs se sont intéressés au coût de la construction de la carte du graphe en termes de nombre d'arêtes traversées par le robot et aussi en termes de mémoire utilisée par le robot pour faire le parcours. Les auteurs ont prouvé que la construction de la carte du graphe se fait en temps polynomial si les agents connaissent une borne n sur le nombre de nœuds du graphe et une borne d sur le degré tel que $d \leq n$. Ils proposent un algorithme ayant un temps de $O(n^6 d^2 \log n)$ et utilisant une mémoire de $O(n^6 d^2 \log n)$. Ils proposent aussi un autre algorithme plus avancé ayant un temps de $O(n^6 d^3 \log n)$ et utilisant une mémoire de $O(n^3 d^2 \log n \log d)$. Ils prouvent également que pour construire la carte du graphe sans connaître la borne sur le degré ou la borne sur le nombre de nœuds du graphe, il faut marquer le nœud du départ, sinon la construction est impossible. Si le nœud de départ est marqué, ils proposent un algorithme en temps polynomial qui permet de déduire n et de construire la carte du graphe en mémoire optimale de $\Theta(\log n)$. Enfin, ils proposent un algorithme qui requiert plus de mémoire ($O(nd \log n)$) pour les agents, mais qui est plus performant en matière de temps $O(n^3 d)$.

Les auteurs de [7] se sont intéressés à l'exploration des terrains avec obstacles. L'obstacle et le terrain sont modélisés par des polygones. Le robot est représenté par un point qui doit explorer le terrain. La mémoire du robot est illimitée et les auteurs de [7] proposent deux scénarios : celui où le robot a une vision illimitée et celui où le robot a une vision limitée. La vision illimitée du robot lui permet de voir tous les autres points du terrain joints au robot par un segment qui ne passe pas par un obstacle. La vision limitée du robot l'empêche de voir au-delà de 1. Tous les points du terrain doivent être explorés. La complexité de l'algorithme proposé par les auteurs est mesurée par la taille de la trajectoire du robot. Pour le scénario avec vision illimitée, les auteurs élaborent un algorithme avec le coût de l'exploration de $O(P + D\sqrt{k})$ où P est le périmètre total du terrain incluant le périmètre des obstacles, où D est le diamètre de l'enveloppe convexe du terrain et où k est le nombre d'obstacles. Pour le scénario avec vision limitée, ils proposent un algorithme avec le coût de l'exploration de $O(P + A + \sqrt{Ak})$ où A est la surface du terrain. Ils prouvent aussi que $\Omega(P + A + \sqrt{Ak})$ est une borne inférieure même si le terrain est connu du robot.

Chapitre 3

Rendez-vous dans les graphes arbitraires

3.1 Le modèle

Deux agents mobiles sont placés sur deux nœuds différents d'un graphe arbitraire de degré maximal Δ connu des agents et exécutent un algorithme déterministe. Une distance initiale connue d'au plus D les sépare. Afin de briser la symétrie, les agents portent des étiquettes différentes qui sont des entiers d'un ensemble $\{0, 1, ..., L-1\}$. Les agents se déplacent dans le graphe en rondes synchrones. À chaque ronde, l'agent décide s'il doit rester immobile sur sa position actuelle ou se déplacer vers un nœud adjacent. Quoique les agents se déplacent en rondes synchrones, un adversaire peut contrôler le moment de départ de chaque agent. Ainsi il y aura un délai θ qui est la différence entre le temps de départ du premier agent et le temps de départ du deuxième agent. Si $\theta = 0$, c'est un départ simultané des agents et si $\theta > 0$, c'est un départ arbitraire des agents. Ces derniers ignorent la valeur de θ et c'est l'adversaire qui décide sa valeur.

Les agents ne peuvent faire le rendez-vous que dans un nœud. Cependant ils peuvent se croiser dans une arête sans s'en apercevoir. Les nœuds ne sont pas étiquetés et leurs ports sont numérotés de façon arbitraire par les entiers $0, 1, ..., d-1$ pour un nœud de degré d. Le degré maximal des nœuds du graphe est Δ. La mémoire des agents est illimitée. Le temps du rendez-vous est défini comme le nombre de rondes entre le départ du premier agent et la rencontre.

3.2 L'algorithme

L'agent commence par la transformation de son étiquette [20] pour en faire une suite binaire avec la propriété suivante : Une étiquette transformée n'est jamais un préfixe d'une autre étiquette transformée. Chaque bit de la représentation binaire de l'étiquette est dupliqué. Ensuite, 10 est ajouté au début et à la fin de la suite binaire obtenue.

Nous définissons la $Boule(D)$ à rayon D avec un centre à la position initiale de l'agent comme l'ensemble de tous les chemins de longueur D codés comme suites des ports. La procédure Parcourir $Boule(D)$ consiste à parcourir tous les chemins de la $Boule(D)$ en ordre lexicographique en revenant au nœud du départ par le chemin rebours. Si un chemin particulier est infaisable (le port qu'il faudrait prendre à un nœud n'existe pas), l'agent abandonne ce chemin en revenant au nœud du départ par le chemin rebours. Le temps du parcours de la $Boule(D)$ est d'au plus $T = 2D\Delta(\Delta-1)^{D-1}$. L'agent lit les bits consécutifs de l'étiquette transformée.

Si le bit est 0, l'agent s'immobilisera sur son nœud de départ pendant un temps de $2T$. Si le bit est de 1, l'agent exécute la procédure $ParcourirBoule(D)$ en temps α, s'immobilise pour un temps $t = 2T - 2\alpha$ et exécute une deuxième fois la procédure $ParcourirBoule(D)$ en temps α.

Nous présentons un algorithme déterministe synchrone qui permet le rendez-vous en un temps de $O(D\Delta(\Delta-1)^{D-1}\log L)$ où Δ est le degré maximal du graphe, où D est la distance initiale entre les deux agents et où L est la grandeur de l'espace des étiquettes.

Parcourir Boule(D)

Générer toutes les suites des ports de longueur D.

Parcourir tous les chemins correspondant en ordre lexicographique en revenant au nœud du départ par le chemin rebours.

Si une suite est infaisable, revenir par le chemin rebours.

Procédure Transformer étiquette l

Transformer l'étiquette en binaire

Dupliquer chaque bit de l'étiquette

Retourner l'étiquette transformée en ajoutant un 10 au début et à la fin

Exécution du bit 1

Exécuter la Procédure Parcourir Boule(D)

α := temps d'exécution de la procédure Parcourir Boule(D)

$T := 2D\Delta(\Delta - 1)^{D-1}$

Rester immobile pendant le temps $t = 2T - 2\alpha$

Exécuter la Procédure Parcourir Boule(D)

L'algorithme suivant est interrompu au moment du rendez-vous.

Algorithme RV-déterministe Graphe Arbitraire

Exécuter la **Procédure Transformer étiquette** l

$C_1...C_k$ <- étiquette transformée

Pour $i = 1$ jusqu'à k **Faire**

  Si $C_i = 1$ **alors**

    Exécuter la **Procédure Exécution du bit 1**

  Sinon

    Rester immobile pour un temps de $2T$

  Fin Si

Fin Pour

3.3 Preuve d'exactitude et analyse

Dans cette section, nous prouvons que l'algorithme **RV-déterministe Graphe Arbitraire** est correct et nous estimons son temps de fonctionnement.

Théorème 3.3.1 *L'algorithme* ***RV-déterministe Graphe Arbitraire*** *résout le problème du rendez-vous de deux agents avec une distance initiale D dans un graphe arbitraire de degré maximal Δ, même dans le cas d'un départ arbitraire.*

Démonstration

Nous allons montrer que l'algorithme est correct pour deux scénarios possibles, d'abord dans le cas le plus simple du départ simultané des agents et ensuite dans le scénario général où le départ des agents est arbitraire.

Soit A_1 et A_2, deux agents mobiles dans un graphe arbitraire, et D, la distance initiale qui les sépare. Δ est le degré maximal du graphe. C et C' sont les étiquettes transformées des agents A_1 et A_2, respectivement.
Soit : $C = (C_1...C_k)$ et $C' = (C'_1...C'_j)$.

Départ simultané

Si le départ est simultané, les deux agents exécutent le bit i de leur étiquette transformée exactement en même temps. Il existe un indice i tel que $C_i \neq C'_i$. Sans perte de généralité supposons que $C_i = 1$ et $C'_i = 0$. Dans ce cas, l'agent A_1 explore la boule $Boule(D)$ pendant que l'agent A_2 reste inerte sur sa position initiale. Puisque cette position est dans la boule $Boule(D)$ de l'agent A_1, le rendez-vous aura lieu au plus tard quand les agents termineront l'exécution du i-ème bit de leur étiquette transformée.

Départ arbitraire

Considérons le départ arbitraire et soit T_1, T_2, T_3,..., les intervalles de temps de longueur $2T$ à partir du départ du premier agent. Soit T_k, l'intervalle de temps pendant lequel le deuxième agent débute, et soit δ, la distance entre le début de T_k et le début du deuxième agent, appelé temps de décalage. Soit T'_1, T'_2, T'_3,..., les intervalles de temps de longueur $2T$ à partir du départ du deuxième agent. Si $\delta \leq T$, nous définissons T_{k+i-1} et T'_i comme des intervalles jumelés et si $\delta > T$, nous définissons T_{k+i} et T'_i comme des intervalles jumelés. Dans les deux cas, les intervalles jumelés ont une intersection de longueur d'au moins T. voir Figure 3.1.

Il existe des intervalles jumelés I et I' qui correspondent aux exécutions des bits différents des étiquettes transformées. Sans perte de généralité supposons que I correspond au bit 1 et que I' correspond au bit 0. Si $\delta <= T$, le second agent reste inerte pendant la deuxième exploration de la boule $Boule(D)$ par le premier agent dans l'intervalle jumelé et si $\delta > T$, le second agent reste inerte pendant la première exploration de la boule $Boule(D)$ par le premier agent dans l'intervalle jumelé. Dans les deux cas, les agents se rencontrent. □

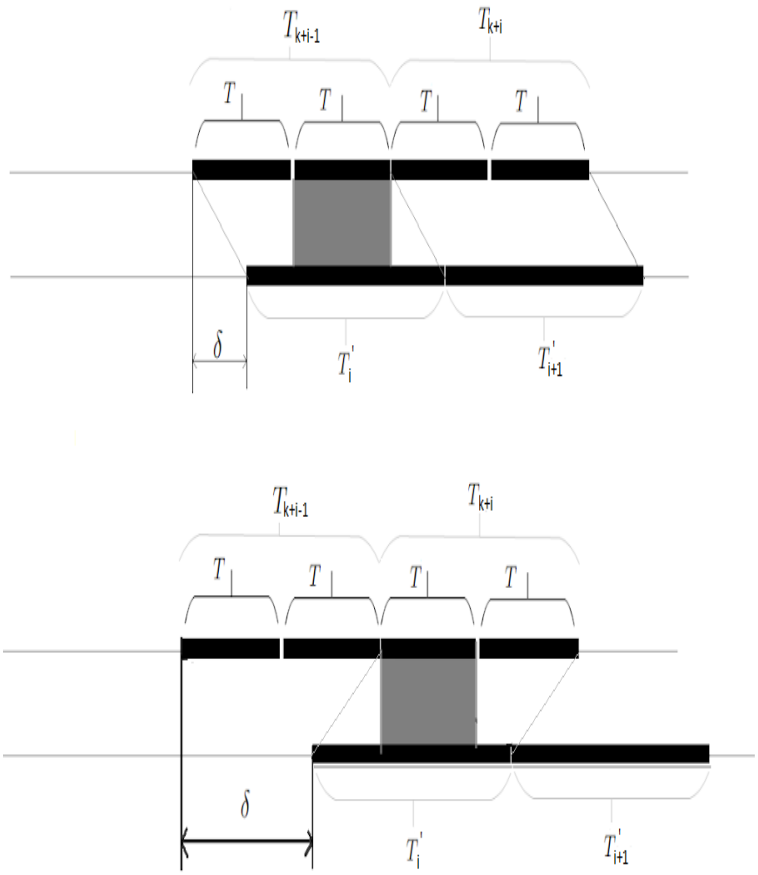

FIGURE 3.1 – Intervalles jumelés durant le départ arbitraire

Théorème 3.3.2 *L'algorithme* ***RV-déterministe Graphe Arbitraire*** *résout le problème du rendez-vous en un temps de* $O(D\Delta(\Delta-1)^{D-1}\log L)$ *où* D *est la distance initiale entre les agents, où* Δ *est le degré maximal du graphe et où* L *est la grandeur de l'espace des étiquettes.*

Démonstration

Pour que l'agent parcoure la boule $Boule(D)$, il doit traverser tous les chemins de longueur D qui commencent à sa position initiale et chaque chemin doit être parcouru deux fois. Il existe au plus $\Delta(\Delta-1)^{D-1}$ chemins. Donc le parcours de la boule $Boule(D)$ prend un temps d'au plus $T = 2D\Delta(\Delta-1)^{D-1}$. Puisque le rendez-vous doit avoir lieu pendant l'exécution du bit de l'étiquette transformée d'un des agents et que la longueur de cette étiquette est de $O(\log L)$, alors le temps du rendez-vous est de $O(D\Delta(\Delta-1)^{D-1}\log L)$.□

3.3.1 Borne inférieure

Nous avons prouvé que le temps du rendez-vous de l'algorithme **RV-déterministe Graphe Arbitraire** est de $O(D\Delta(\Delta-1)^{D-1}\log L)$. Dans cette section, nous allons vérifier si nous pouvons améliorer ce temps de façon significative. Le théorème suivant montre qu'il y a des graphes dans lesquels le rendez-vous prend un temps d'au moins $\Omega(\Delta(\Delta-1)^{D-1})$.

Théorème 3.3.3 *Soit* T*, l'arbre où chaque nœud autre que les feuilles a le degré* Δ *et où toutes les feuilles sont à une distance* D *d'un nœud* v*. Le rendez-vous dans* T *prend un temps d'au moins* $\Omega(\Delta(\Delta-1)^{D-1})$.

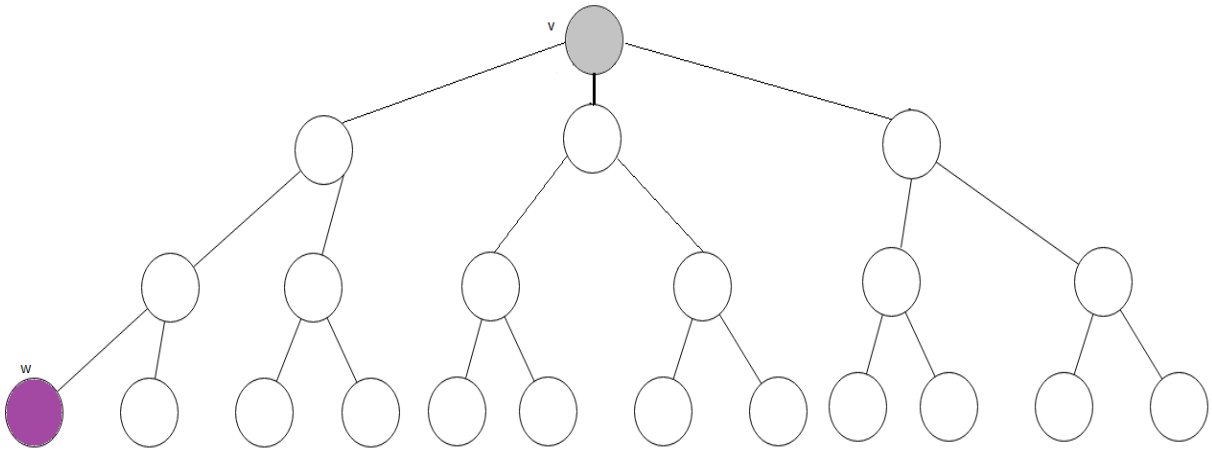

FIGURE 3.2 – Arbre avec degré $\Delta = 3$ et toutes les feuilles sont à la même distance $D = 3$ d'un nœud v

Démonstration

Considérons un algorithme de rendez-vous A et soit P, la trajectoire d'un agent avec l'étiquette 1 placé au nœud v. Soit w, la dernière feuille de T (voir Figure 3.2) visitée par cet agent exécutant l'algorithme A sans rencontrer un autre agent. L'adversaire place l'agent avec l'étiquette 1 dans v, place l'agent avec l'étiquette 2 dans le nœud w, et retarde le départ de cet agent jusqu'au moment de la visite de w par le premier agent. Le rendez-vous aura lieu au nœud de la visite de w par le premier agent. Puisqu'il y a $\Theta(\Delta(\Delta-1)^{D-1})$ feuilles dans T, la feuille w sera visitée après un temps de $\Omega(\Delta(\Delta-1)^{D-1})$. □

Il s'ensuit de [11] que dans l'anneau le temps du rendez-vous est de $\Omega(D \log L)$. Donc nous obtenons la borne inférieure $\Omega(\Delta(\Delta-1)^{D-1} + D \log L)$ pour le temps de rendez-vous dans la classe des graphes arbitraires de degré maximal Δ. Cette borne inférieure est assez proche du temps de fonctionnement de notre algorithme. Par exemple, pour la distance initiale D constante, le ratio des deux bornes est logarithmique en L.

3.3.2 Conclusion

Dans ce chapitre, nous avons présenté l'algorithme **RV-déterministe Graphe Arbitraire** qui résout le problème de rendez-vous de deux agents avec une distance initiale au plus D dans un graphe arbitraire de degré maximal Δ. Nous avons introduit la notion d'intervalles jumelés pour prouver le fonctionnement de notre algorithme pour un

départ arbitraire. Enfin, nous avons prouvé que cet algorithme résout le problème du rendez-vous en un temps de $O(D\Delta(\Delta-1)^{D-1}\log L)$ et nous avons établi une borne inférieure $\Omega(\Delta(\Delta-1)^{D-1}+D)$ sur le temps du rendez-vous dans la classe des graphes de degré maximal Δ avec distance initiale d'au plus D. Maintenant, posons les questions suivantes : En supposant que les agents sont à distance initiale au plus D, est-ce qu'il existe un algorithme déterministe synchrone qui permet le rendez-vous dans une grille avec direction (Nord, Sud, Est, Ouest) ? Si oui, en quel temps peut-on le faire ? Cet algorithme peut-il nous aider à résoudre le problème de rapprochement dans le plan ? Les réponses à ces questions feront l'objet de nos recherches futures.

Chapitre 4

Rendez-vous dans la grille infinie

4.1 Le modèle

Deux agents mobiles sont placés sur deux nœuds différents dans une grille infinie orientée. La grille est un graphe où chaque nœud contient quatre ports, dont chacun représente une direction (Nord, Sud, Est, Ouest). Ces directions sont les mêmes dans chaque nœud. Les agents se déplacent en traversant les arêtes de la grille. Nous appelons une droite horizontale l'ensemble des nœuds qui peuvent être atteint à partir d'un nœud v en employant seulement les ports E et W. La définition d'une droite verticale est semblable. Les agents exécutent un algorithme déterministe. Une distance initiale connue d'au plus D les sépare. La distance entre deux nœuds est la longueur du chemin le plus court entre eux dans la grille. Afin de briser la symétrie, les agents portent des étiquettes différentes qui sont des entiers d'un ensemble $\{0, 1, ..., L-1\}$. Chaque agent connaît L et son étiquette mais pas celle de l'autre agent. Les agents se déplacent dans la grille en rondes synchrones. À chaque ronde, l'agent décide s'il doit rester immobile sur sa position actuelle ou se déplacer vers un nœud adjacent. Quoique les agents se déplacent en rondes synchrones, un adversaire peut contrôler le moment de départ de chaque agent. Ainsi, il y aura un délai θ qui est la différence entre le temps de départ du premier agent et le temps de départ du deuxième agent. Si $\theta = 0$, c'est un départ simultané des agents et si $\theta > 0$, c'est un départ arbitraire des agents. Ces derniers ignorent la valeur de θ, et c'est l'adversaire qui décide sa valeur.

Les agents ne peuvent faire le rendez-vous que dans un nœud. Cependant, ils peuvent se croiser dans une arête sans s'en apercevoir. Les nœuds ne sont pas étiquetés et les ports dans chaque nœud sont numérotés par les entiers $0, 1, 2, 3$ qui représentent respec-

tivement les directions Nord, Sud, Est, Ouest. La mémoire des agents est illimitée. Le temps du rendez-vous est défini comme le nombre de rondes entre le départ du premier agent et la rencontre.

4.2 L'algorithme

L'agent commence par la même transformation de son étiquette que dans le chapitre 3, pour en faire une suite binaire avec la propriété suivante : une étiquette transformée n'est jamais un préfixe d'une autre étiquette transformée. Chaque bit de la représentation binaire de l'étiquette est dupliqué. Ensuite, 10 est ajouté au début et à la fin de la suite binaire obtenue.

Nous définissons la $Spirale(D)$ à rayon D avec un centre à la position initiale de l'agent comme le chemin codé par la suite de ports définie comme suit : soit N, S, E, O, les ports qui représente respectivement les directions Nord, Sud, Est, Ouest. Soit N^k, pour $k \geq 1$, le chemin de l'agent à distance k en direction N. Les symboles S^k, E^k, O^k sont définis pareillement. Le point dénote la concaténation des chemins. La $Spirale(D)$ est définie comme le chemin $(E^1.N^1).(O^2.S^2)...(E^{2D-1}.N^{2D-1}).(O^{2D}.S^{2D}).(E^D)$.

À titre d'exemples :

$Spirale(1) := ENOOSSE$

$Spirale(2) := ENOOSSEEENNNOOOOSSSSEE$

$Spirale(3) := ENOOSSEEENNNOOOOSSSSEEEEENNNNNOOOOOOSSSSSSSEEE$

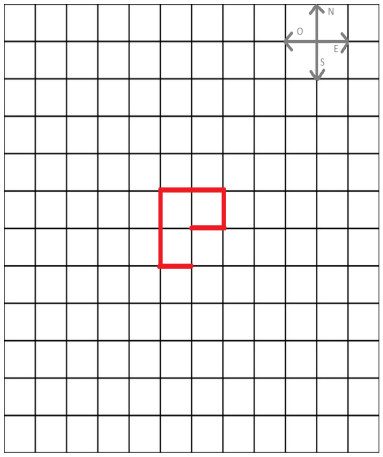

FIGURE 4.1 – Le chemin $Spirale(1)$

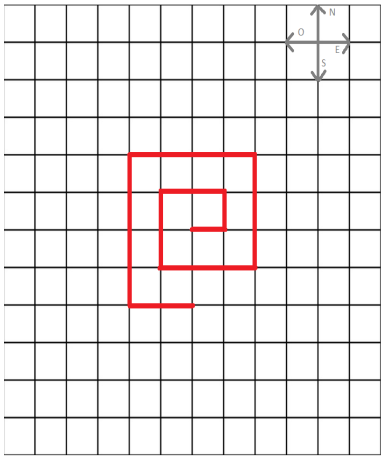

FIGURE 4.2 – Le chemin $Spirale(2)$

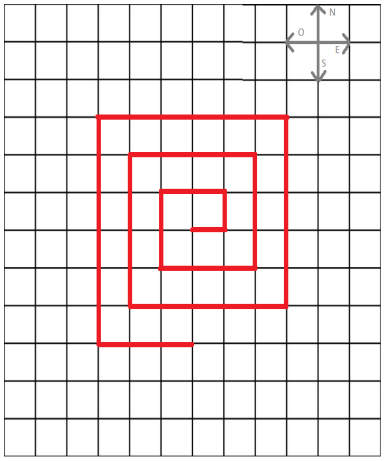

FIGURE 4.3 – Le chemin $Spirale(3)$

La procédure *Parcourir Spirale*(D) consiste à suivre le chemin $Spirale(D)$ en revenant au nœud du départ par le chemin N^D. Le temps d'exécution de cette procédure est $T = 4D^2 + 4D$. L'agent lit les bits consécutifs de son étiquette transformée. Si le bit est 0, l'agent s'immobilise sur son nœud de départ pendant un temps de $2T$. Si le bit est de 1, l'agent exécute deux fois la procédure *Parcourir Spirale*(D).

Nous présentons un algorithme déterministe synchrone qui permet le rendez-vous en un temps de $O(D^2 \log L)$ où D est une borne supérieure sur la distance initiale entre les deux agents et L, la grandeur de l'espace des étiquettes.

Parcourir Spirale(D)

Parcourir le chemin $Spirale(D)$.

Revenir par le chemin N^D.

Procédure Transformer étiquette l

Transformer l'étiquette en binaire

Dupliquer chaque bit de l'étiquette

Retourner l'étiquette transformée en ajoutant un 10 au début et à la fin

L'algorithme suivant est exécuté par l'agent avec l'étiquette l et il est interrompu au moment du rendez-vous.

Algorithme RV-déterministe Grille Infinie

Exécuter la **Procédure Transformer étiquette** l

$C_1...C_k$ <- étiquette transformée

Pour $i = 1$ jusqu'à k **Faire**

Si $C_i = 1$ **alors**

Exécuter deux fois consécutives la **Procédure Parcourir Spirale(D)**

Sinon

Rester immobile pour un temps de $2T$

Fin Si

Fin Pour

4.3 Preuve d'exactitude et analyse

4.3.1 Estimation de la complexité de l'algorithme

Dans cette section, nous prouvons que l'algorithme **RV-déterministe Grille Infinie** est correct et nous estimons son temps de fonctionnement.

Théorème 4.3.1 *L'algorithme* ***RV-déterministe Grille Infinie*** *résout le problème du rendez-vous de deux agents avec une distance initiale au plus D dans une grille infinie, même dans le cas d'un départ arbitraire. La complexité de l'algorithme est de $O(D^2 \log L)$ où L est la grandeur de l'espace des étiquettes.*

Démonstration

Nous allons démontrer que l'algorithme est correct pour deux scénarios possibles, d'abord dans le cas du départ simultané des agents et ensuite, dans le scénario général où le départ des agents est arbitraire.

Soit A_1 et A_2, deux agents mobiles dans une grille infinie, à distance initiale au plus D, et soit $Boule(v, D)$ l'ensemble de sommets, à distance au plus D du nœud de la grille v. C et C' sont respectivement les étiquettes transformées des agents A_1 et A_2. Soit : $C = (C_1...C_k)$ et $C' = (C'_1...C'_j)$.

Départ simultané

Si le départ est simultané, les deux agents exécutent le bit i de leur étiquette transformée exactement en même temps. Il existe un indice i tel que $C_i \neq C'_i$. Sans perte de généralité, supposons que $C_i = 1$ et $C'_i = 0$. Dans ce cas, l'agent A_1 explore la boule $Boule(v, D)$ pendant que l'agent A_2 reste inerte sur sa position initiale. Puisque cette position est dans la boule $Boule(v, D)$ de l'agent A_1, le rendez-vous aura lieu au plus tard quand les agents termineront l'exécution du i-nième bit de leur étiquette transformée.

Départ arbitraire

En suivant le même raisonnement que celui présenté au chapitre 3.3. (voir Figure 3.1), il existe toujours un intervalle de temps correspondant à l'exécution d'un bit de l'étiquette transformée où l'un des agents reste inerte pendant que l'autre explore la boule $Boule(v, D)$. Les agents se rencontrent pendant cet intervalle de temps. □

L'agent parcoure la boule $Boule(v, D)$, en suivant le chemin $Spirale(D)$. Donc le parcours de la boule $Boule(v, D)$ prend un temps d'au plus $T = 4D^2 + 4D$. Puisque le rendez-vous doit avoir lieu pendant l'exécution du bit de l'étiquette transformée d'un des agents et que la longueur de cette étiquette est de $O(\log L)$, alors le temps du rendez-vous est de $O(D^2 \log L)$.□

4.3.2 Borne inférieure

Dans cette section, nous allons vérifier si nous pouvons améliorer le temps de rendez-vous de l'algorithme **RV-déterministe Grille Infinie** de façon significative. Le théorème suivant montre que le rendez-vous prend un temps d'au moins $\Omega(D^2)$.

Théorème 4.3.2 *Soit G une grille infinie orientée. Deux agents sont placés sur la grille à distance au plus D. Le rendez-vous dans G prend un temps d'au moins $\Omega(D^2)$.*

Démonstration

Considérons un algorithme de rendez-vous A et soit $Boule(v, D)$ l'ensemble de sommets à distance au plus D du nœud de la grille. Un agent avec l'étiquette 1 est placé au nœud v. Soit w, le dernier nœud de $Boule(v, D)$ visité par cet agent exécutant l'algorithme A sans rencontrer un autre agent. L'adversaire place l'agent avec l'étiquette

2 dans le nœud w et retarde le départ de cet agent jusqu'au moment de la visite de w par le premier agent. Le rendez-vous aura lieu au nœud de la visite de w par le premier agent. Puisqu'il y a $1+2D(D+1)$ nœuds dans $Boule(v, D)$, le nœud w sera visité après un temps de $\Omega(D^2)$ et donc, le rendez-vous doit prendre un temps de $\Omega(D^2)$. □

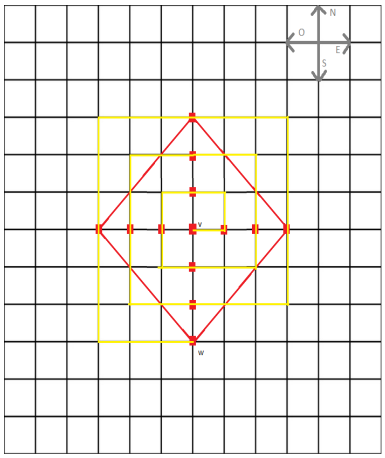

FIGURE 4.4 – $Boule(v, D)$ avec $D = 3$

Le théorème suivant montre que le temps $\Omega(DLogL)$ est aussi nécessaire.

Théorème 4.3.3 *Soit G une grille infinie orientée. Deux agents sont placés sur la grille à distance au plus D. Le rendez-vous dans G prend un temps d'au moins $\Omega(DLogL)$.*

Démonstration

Soit deux agents mobiles placés à une distance initiale D divisible par 6 dans deux nœuds d'une grille infinie sur la même droite horizontale. Appelons segment vertical de largeur x l'ensemble de tous les nœuds de la grille sur les $x+1$ lignes verticales consécutives. Divisons la grille en segments verticaux de largeur $\frac{D}{6}$ ainsi que les positions initiales des agents étant sur les bords des segments verticaux. Soit A, un algorithme qui permet le rendez-vous en temps T d'au plus $\frac{D}{6\log 3}\log L$. Les agents exécutent A avec un départ simultané. On divise le temps en périodes successives de longueur $\frac{D}{6}$. On considère le *code de comportement* [11] d'un agent mobile. Ce code est une suite de termes dans l'ensemble $\{-1, 0, 1\}$ définie de la manière suivante : soit S, le segment où l'agent a commencé la période i. Le $i-me$ terme du code est -1 si l'agent termine sa période dans le segment précédant S, il est 1 s'il termine la période dans le segment

suivant le segment S et il est 0 si l'agent termine la période dans le segment S. Notons que ce sont les seules possibilités, car l'agent ne peut pas terminer une période dans un segment S' séparé par un autre segment du segment S. Soit $\{1, ..., L\}$, l'ensemble des étiquettes et soit Z, la longueur du code de comportement généré par un agent. Alors, $Z \leq \frac{T}{\frac{D}{6}} < \frac{DlogL}{6log3.\frac{D}{6}} = \frac{logL}{log3}$, d'où $3^Z < L$. Il existe donc au moins deux étiquettes différentes l_1 et l_2 qui génèrent le même code de comportement.

L'adversaire assigne les étiquettes l_1 et l_2 aux deux agents. Puisque les agents ont le même code de comportement à la fin de chaque période, ils seront séparés par au moins quatre segments. Pendant chaque période, chacun d'eux peut pénétrer seulement un segment voisin, alors ils seront toujours séparés par au moins deux segments. Donc, les deux agents ne vont jamais se rencontrer. C'est une contradiction avec le fait que A permet de faire le rendez-vous. Donc, $T \geq \frac{1}{6 \log 3} D \log L$. D'où le temps du rendez-vous est à $\Omega(DlogL)$. □

Les théorèmes 4.3.2 et 4.3.3 impliquent le corollaire suivant :

Corollaire 1 *Le temps optimal du rendez-vous de deux agents placés à distance au plus D dans une grille infinie orientée est $\Omega(D^2 + D \log L)$.*

4.3.3 Conclusion

Dans ce chapitre, nous avons présenté l'algorithme **RV-déterministe Grille Infinie** qui résout le problème de rendez-vous de deux agents avec une distance initiale au plus D dans une grille infinie. Nous avons introduit la notion de parcours en spirale de la $Boule(v, D)$ qui est l'ensemble de sommets à distance au plus D du nœud v de la grille. Enfin, nous avons prouvé que cet algorithme résout le problème du rendez-vous en un temps de $O(D^2 \log L)$ et nous avons établi une borne inférieure $\Omega(D^2 + D \log L)$ sur le temps du rendez-vous dans la grille infinie avec distance initiale d'au plus D. Trouver un algorithme optimal pour la tâche de rendez-vous dans une grille infinie orientée reste un problème ouvert. Maintenant, posons la question suivante : l'algorithme **RV-déterministe Grille Infinie** peut-il nous aider à résoudre le problème de l'approche dans le plan ? Une réponse sera étudiée dans le chapitre suivant.

Chapitre 5

Approche dans le plan

5.1 Le modèle

Deux agents mobiles sont placés dans le plan. Ils commencent l'exécution d'un algorithme déterministe à des temps possiblement différents choisis par l'adversaire. Chaque agent est équipé d'une mesure commune de longueur et d'une boussole. Chaque agent à une horloge qui démarre quand l'agent commence l'exécution de l'algorithme. L'agent peut rester immobile pendant le temps qu'il souhaite ou se déplacer à une distance et dans une direction de son choix. Lorsqu'il se déplace, il bouge avec la vitesse constante 1. Chaque agent est représenté par un disque de diamètre 1 qui se déplace par une suite des mouvements élémentaires définis par une distance donnée x et une direction donnée α. Un exemple de déplacement d'un agent serait de : se déplacer vers un point à une distance $x = 5$ en direction $\alpha = Ouest$. Les agents commencent à une distance initiale plus grande que 1 et au plus D, où D est un nombre positif connu. Cette distance est mesurée entre les centres des disques. Afin de briser la symétrie, les agents portent des étiquettes différentes qui sont des entiers d'un ensemble $\{0, 1, ..., L-1\}$. Chaque agent connaît L et connaît son étiquette, mais pas celle de l'autre agent. Le délai θ est défini comme la différence entre le temps de départ du premier agent et le temps de départ du deuxième agent. Si $\theta = 0$, c'est un départ simultané des agents et si $\theta > 0$, c'est un départ arbitraire des agents. Ces derniers ignorent la valeur de θ et c'est l'adversaire qui décide sa valeur. Le but des agents est d'exécuter l'approche qui a lieu quand les centres des disques sont à distance 1, c'est-à-dire quand les disques se touchent. La mémoire des agents est illimitée. Le temps de l'approche est défini comme le temps entre le départ du premier agent et le moment de l'approche.

5.2 L'algorithme

Chaque agent A commence l'exécution de l'algorithme par la création de sa propre grille virtuelle orientée G_A (voir Figure 5.1.). La grille G_A est un graphe où chaque nœud contient quatre ports, dont chacun représente une direction (Nord, Sud, Est, Ouest). Ces directions sont les mêmes dans chaque nœud. Chaque arête de la grille G a une taille de $\frac{1}{2}$. La position initiale de l'agent A est un des nœuds de la grille G_A. Afin de réaliser l'approche, chaque agent exécute l'algorithme **RV-déterministe Grille Infinie** du chapitre précédent dans sa grille virtuelle à l'aide de la boussole. Plus précisément, si l'algorithme **RV-déterministe Grille Infinie** instruit l'agent de prendre le port Nord, Sud, Est, Ouest, l'agent dans le plan doit aller à distance $\frac{1}{2}$ dans la direction respective Nord, Sud, Est, Ouest.

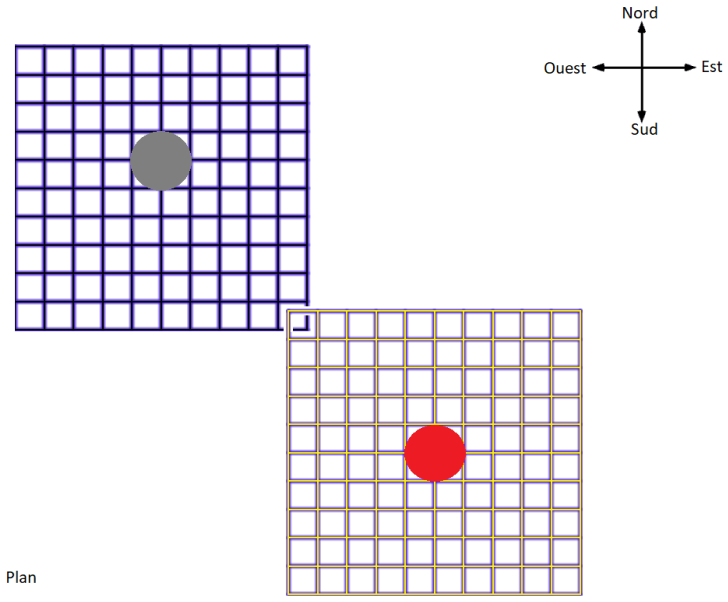

FIGURE 5.1 – Les grilles virtuelles G_{A_1} et G_{A_2}

Nous présentons un algorithme déterministe synchrone qui permet l'approche dans le plan en un temps de $O(D^2 \log L)$ où D est une borne supérieure sur la distance initiale entre les deux agents et L est la grandeur de l'espace des étiquettes.

L'algorithme suivant est exécuté par l'agent A avec l'étiquette l et il est interrompu au moment de l'approche.

Algorithme Approche-Plan

Créer la grille virtuelle G_A

Exécuter algorithme **RV-déterministe Grille Infinie** dans la grille G_A

5.3 Preuve d'exactitude et analyse

5.3.1 Estimation de la complexité de l'algorithme

Dans cette section, nous prouvons que l'algorithme **Approche-Plan** est correct et nous estimons son temps de fonctionnement.

Théorème 5.3.1 *L'algorithme* ***Approche-Plan*** *résout le problème de l'approche de deux agents avec une distance initiale au plus D dans le plan, même dans le cas d'un départ arbitraire. La complexité de l'algorithme est de $O(D^2 \log L)$ où L est la grandeur de l'espace des étiquettes.*

Démonstration

Soit A_1 et A_2, deux agents mobiles dans un plan à distance initiale au plus D et soit $G_1 = G_{A_1}$ et $G_2 = G_{A_2}$, les deux grilles virtuelles des agents A_1 et A_2 respectivement. Soit p le point de départ de l'agent A_2 et soit p' le nœud de la grille G_1 le plus proche du point p.(Dans le cas où plusieurs points seraient les plus proches, on choisit arbitrairement l'un d'eux.) Soit v, le vecteur $\overrightarrow{pp'}$ (voir Figure 5.2). La longueur du vecteur v est plus petite que 1. Considérons l'exécution de l'algorithme **Approche-Plan** dans la grille G_1 par l'agent A_1. Simulons l'exécution de l'algorithme **Approche-Plan** dans la grille G_2 par l'agent A_2 en le transposant dans la grille G_1 de la façon suivante : la trajectoire de l'agent A_2 est décalée par le vecteur v. En particulier, la simulation commence dans un nœud de la grille G_1 et les mouvements simulés se font dans cette grille. Puisque l'algorithme **RV-déterministe Grille Infinie** assure le rendez-vous dans la grille G_1, l'agent A_1 et l'agent simulé A_2 se rencontrent dans un nœud q de cette grille. En ce moment, les agents A_1 et A_2 sont à distance plus petite que 1, donc avant ce mouvement, ils ont dû être à distance 1 (par la continuité des mouvements), donc l'approche a eu lieu. □

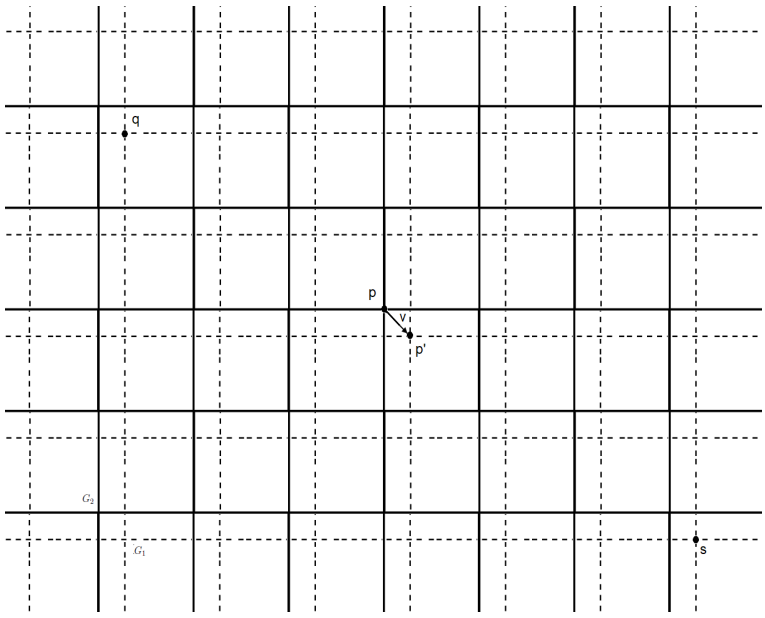

FIGURE 5.2 – Les grilles virtuelles G_1 et G_2

Il reste à estimer le temps de l'approche. Puisque la distance euclidienne entre les positions initiales des agents est au plus D, la distance dans la grille G_1 entre le nœud s où l'agent A_1 commence et le nœud p' où l'agent simulé A_2 commence, est $D' \in O(D)$. L'algorithme **RV-déterministe Grille Infinie** prend un temps dans $O(D'^2 \log L)$ pour le rendez-vous. Donc, le temps de l'approche est $O(D'^2 \log L) = O(D^2 \log L)$.

5.3.2 Conclusion

Dans ce chapitre, nous avons présenté l'algorithme **Approche-Plan** qui résout le problème de l'approche de deux agents dans un plan avec une distance euclidienne initiale au plus D. Nous avons introduit la notion de grille virtuelle qui nous a permis de réaliser l'approche des deux agents en appliquant l'algorithme **RV-déterministe Grille Infinie**. Enfin, nous avons prouvé que l'algorithme **Approche-Plan** résout le problème de l'approche dans le plan en un temps de $O(D^2 \log L)$, où L est la grandeur de l'espace des étiquettes.

Chapitre 6

Logiciel de simulation

6.1 Introduction

Le logiciel de simulation implémente les deux algorithmes **RV-déterministe Graphe Arbitraire** et **RV-déterministe Grille Infinie**. Le logiciel est développé en Angular [1] qui est un cadriciel (*framework*) *open source* basé sur TypeScript [2] dirigé par Google [3]. Deux facteurs majeurs ont influencé le choix d'Angular. Le premier facteur est de rendre le logiciel de simulation *open source* [4] ce qui ouvre la porte à des contributions futures. Le deuxième facteur est la portabilité. En effet, le cadriciel Angular permet l'exécution du logiciel de simulation dans n'importe quel dispositif qui contient un fureteur Web (ordinateur, tablette, téléphone intelligent). Enfin, une librairie importante utilisée dans le logiciel de simulation est VisJS [5]. Elle permet la présentation visuelle ainsi que la manipulation des graphes dans un fureteur Web.

6.2 Défis et limitations

Le choix d'utiliser un fureteur Web comme plate-forme d'exécution du logiciel de simulation était un grand défi durant le développement de celui-ci. En effet, un fureteur Web par conception est exécuté dans un *(Thread)* processus unique ce qui rend impossible l'implantation de nos algorithmes. Ceux-ci sont principalement exécutés en

1. `https://angular.io`
2. `https://www.typescriptlang.org`
3. `https://fr.wikipedia.org/wiki/Angular`
4. `https://github.com/amarof/rendezvous-demo`
5. `http://visjs.org`

parallèle par les deux agents *(multithreading)*. Pour contourner cette limitation majeure, l'utilisation d'un nouveau concept appelé *web worker*[6] a rendu possible l'exécution de nos algorithmes dans un environnement *(multithreading)* et ainsi permettre la simulation de nos algorithmes.
Une des limitations connues du logiciel de simulation est celle de la performance qui est étroitement liée à la performance du fureteur Web. Si, par exemple l'utilisateur lance plusieurs instances de son fureteur, le temps d'exécution de l'algorithme peut être affecté. Aussi, le logiciel de simulation ne peut pas être exécuté dans les anciennes versions des fureteurs Web (Chrome, Internet Explorer, Firefox) qui ne supportent pas le *web worker*.

6.3 Fonctionnalités

Les fonctionnalités du logiciel de simulation sont présentée sous six sections (illustrées par la Figure 6.1).

6. `https://en.wikipedia.org/wiki/Web_worker`

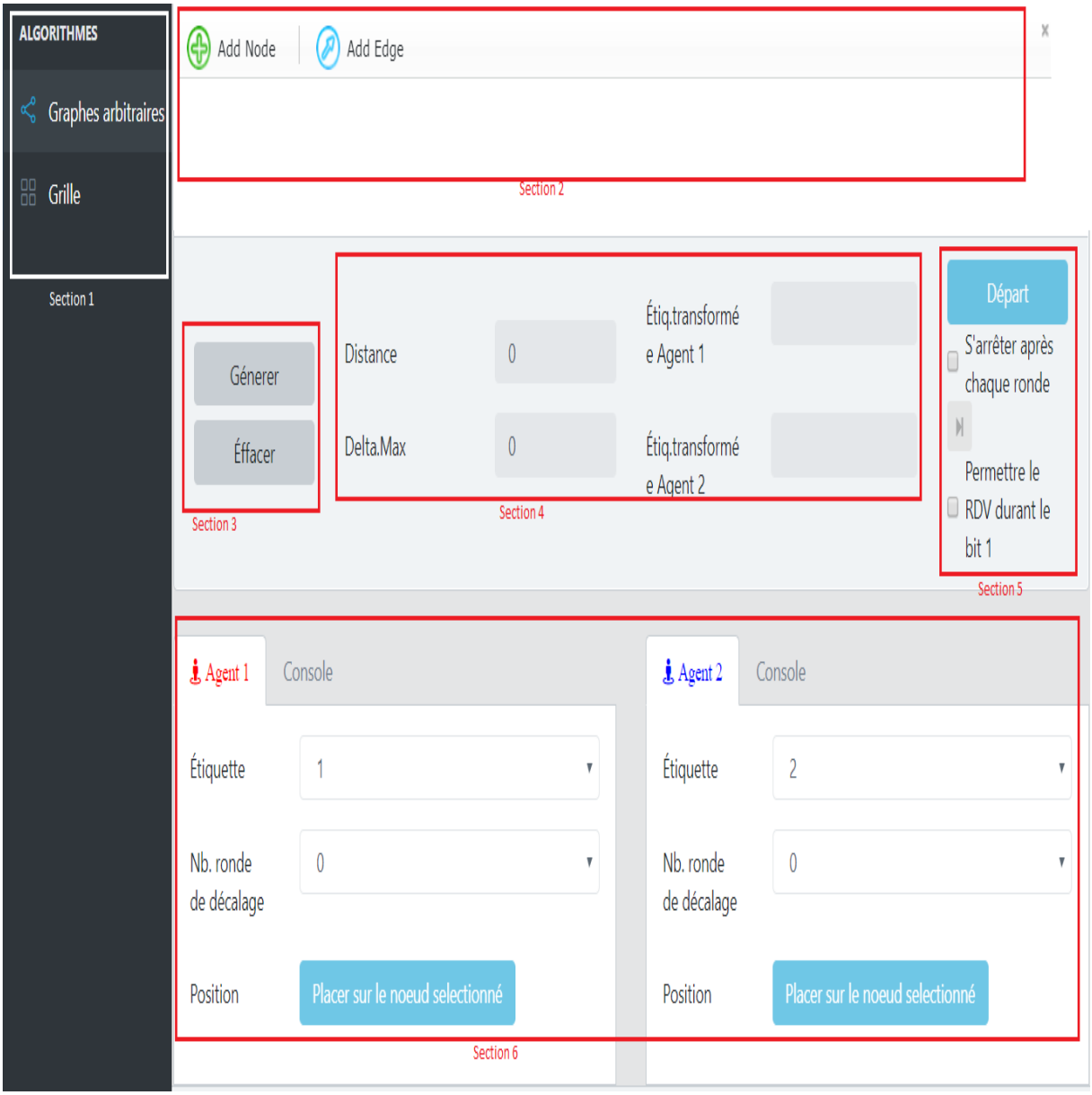

FIGURE 6.1 – Les différentes sections du logiciel de simulation

— (1) Menu qui permet de naviguer entre la simulation de l'algorithme pour les graphes arbitraires et la simulation de l'algorithme pour la grille infinie.
— (2) Boîte à outils qui permet la création du graphe.
— (3) Bouton «Générer»qui permet de construire un graphe arbitraire complexe et le Bouton «Effacer»permet de supprimer un graphe existant.

— (4) Section affichant la distance entre les deux agents, le degré maximal Δ et l'étiquette transformée des deux agents.
— (5) Bouton «Départ»qui permet de lancer l'exécution de la simulation. Une case à cocher qui permet le rendez-vous durant un bit 1 et une case à cocher qui permet l'arrêt de l'agent après la visite de chaque nœud.
— (5) Section qui permet de choisir l'étiquette de chaque agent, le nombre de rondes de décalage et la position de chaque agent dans le graphe. Enfin, une zone console qui permet d'afficher les étapes de l'algorithme exécuté par chaque agent.

6.3.1 Étapes de la simulation pour un graphe arbitraire

1. Construire un graphe (Figure 6.2).
2. Choisir une étiquette pour chaque agent (Figure 6.3).
3. Choisir le nombre de rondes de décalage (optionnel) (Figure 6.2).
4. Sélectionner un nœud pour placer l'agent-1 et cliquer sur le bouton «Placer sur le nœud sélectionné»de la zone Agent 1 (Figure 6.2).
5. Sélectionner un nœud pour placer l'agent-2 et cliquer sur le bouton «Placer sur le nœud sélectionné»de la zone Agent 2 (Figure 6.2).
6. Cliquer sur le bouton «Départ »pour commencer la simulation (Figure 6.3) .
7. Exécuter l'algorithme : chaque agent va se déplacer à travers les nœuds du graphe. La zone console affiche les étapes d'exécution de l'algorithme (Figure 6.4).
8. La simulation est terminée une fois le rendez-vous est effectué (Figure 6.5) .

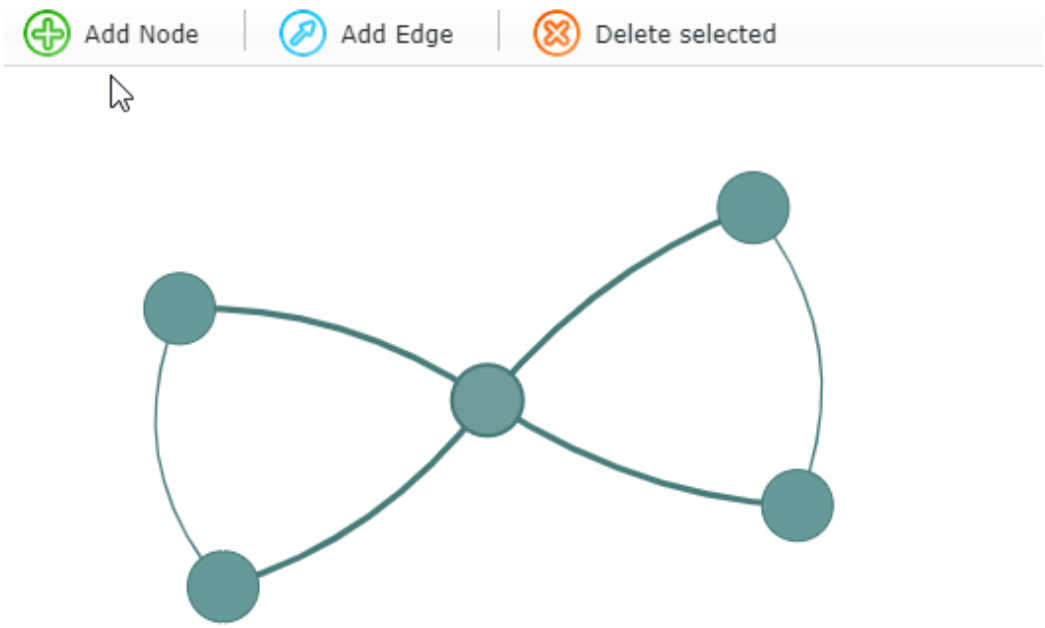

FIGURE 6.2 – Construire un graphe

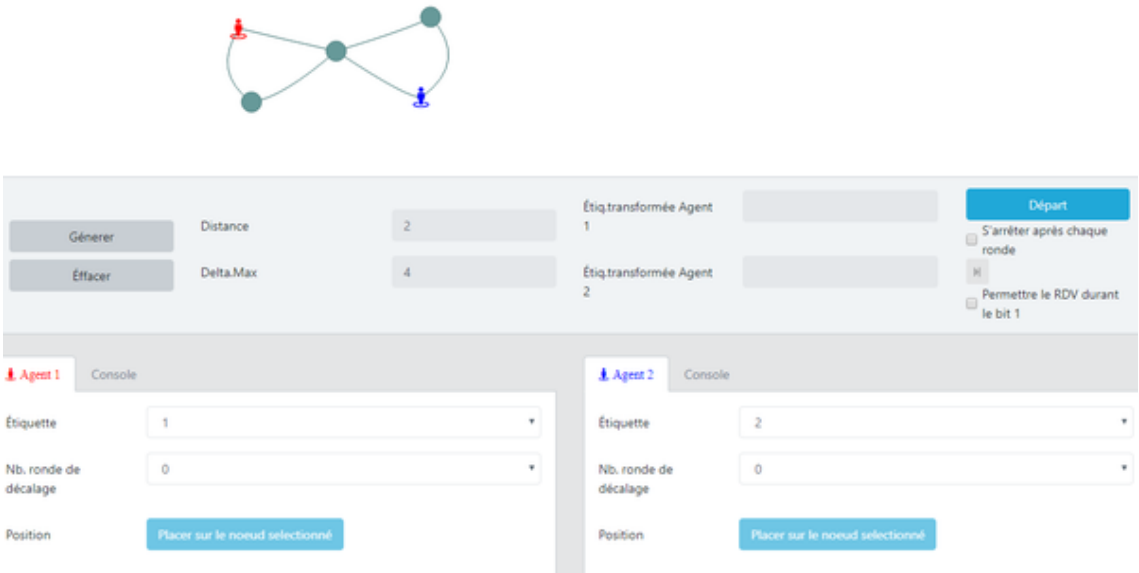

FIGURE 6.3 – Choisir les étiquettes et placer les agents dans le graphe

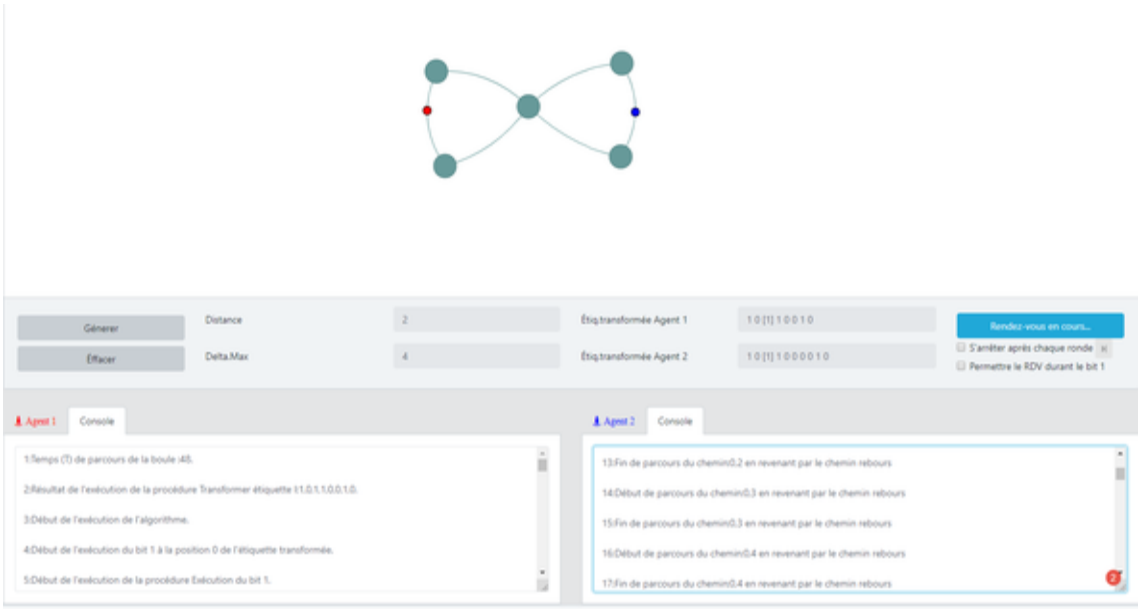

FIGURE 6.4 – Exécution de la simulation de l'algorithme de rendez-vous pour les graphes arbitraires

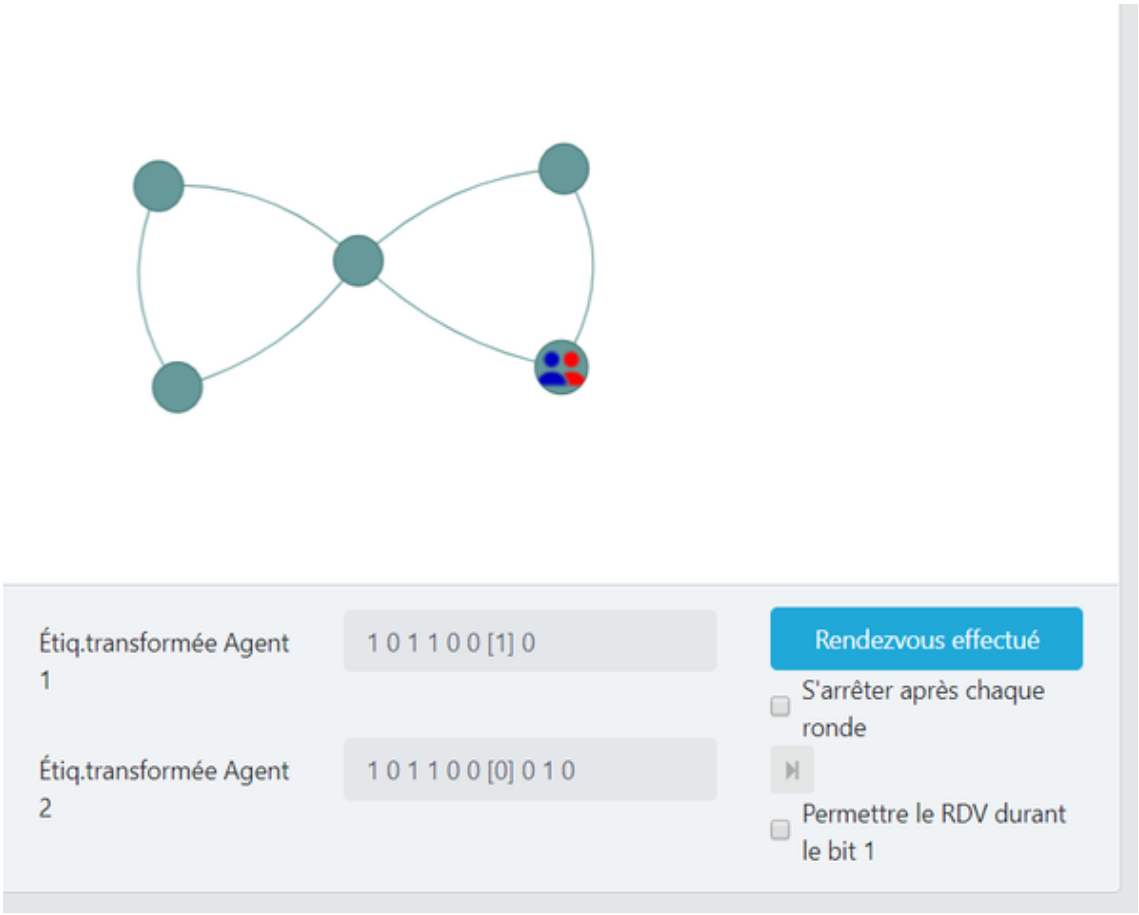

FIGURE 6.5 – Fin de la simulation de l'algorithme de rendez-vous pour les graphes arbitraires

6.3.2 Étapes de la simulation pour une grille infinie

Par souci de simplicité, le logiciel de simulation va construire une grille de taille limitée à 10 x 10. (Figure 6.6).

1. Choisir une étiquette pour chaque agent (Figure 6.6).
2. Choisir le nombre de rondes de décalage (optionnel) (Figure 6.6).
3. Sélectionner un nœud pour placer l'agent-1 et cliquer sur le bouton «Placer sur le nœud sélectionné »de la zone Agent 1 (Figure 6.6).
4. Sélectionner un nœud pour placer l'agent-2 et cliquer sur le bouton «Placer sur le nœud sélectionné»de la zone Agent 2 (Figure 6.6).
5. Cliquer sur le bouton «Départ»pour commencer la simulation (Figure 6.7)

6. Exécuter l'algorithme : chaque agent va se déplacer à travers les nœuds de la grille. La zone console affiche les étapes d'exécution de l'algorithme .
7. Fin de la simulation une fois le rendez-vous effectué (Figure 6.8).

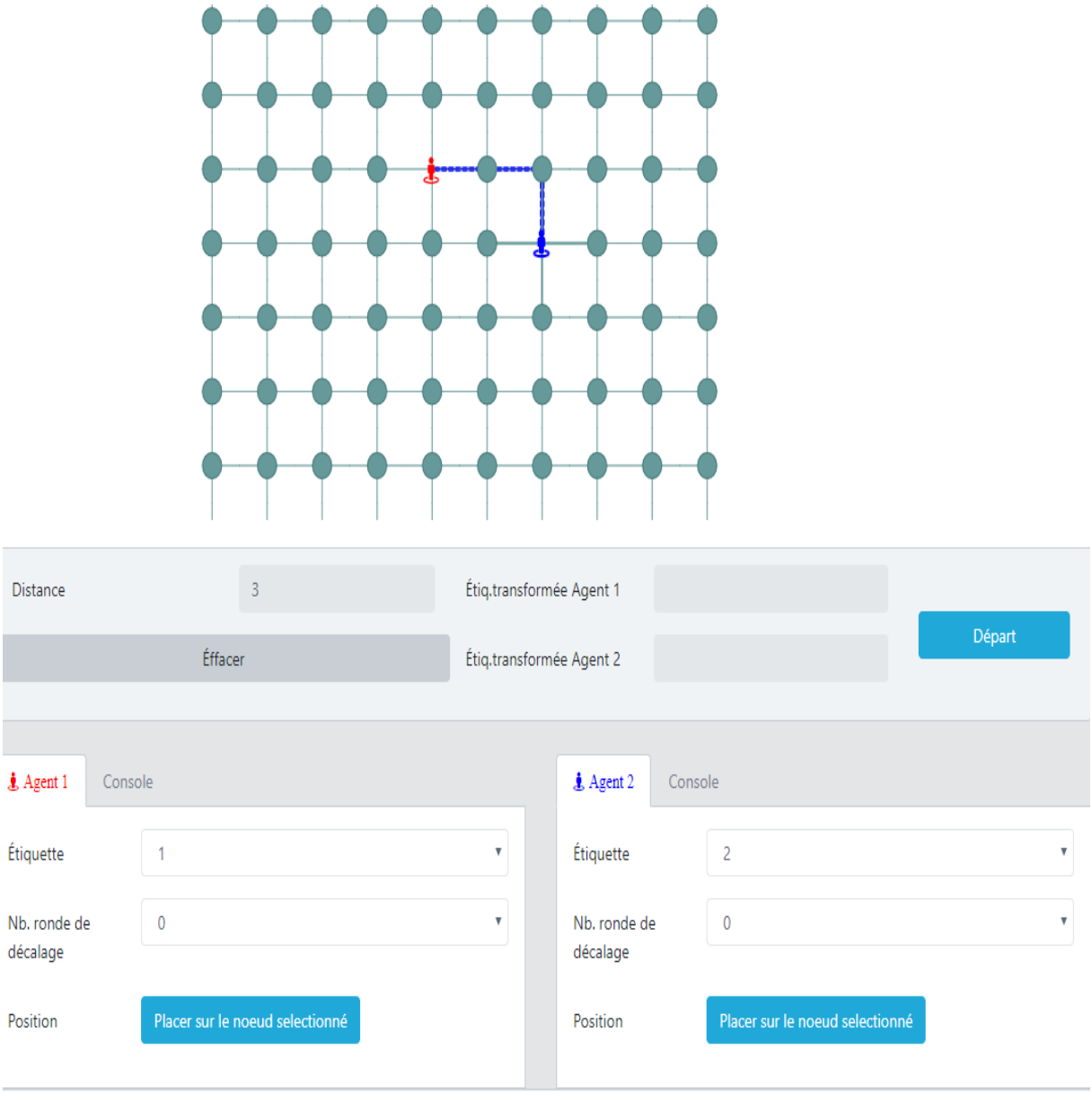

FIGURE 6.6 – Placer les agents dans la grille

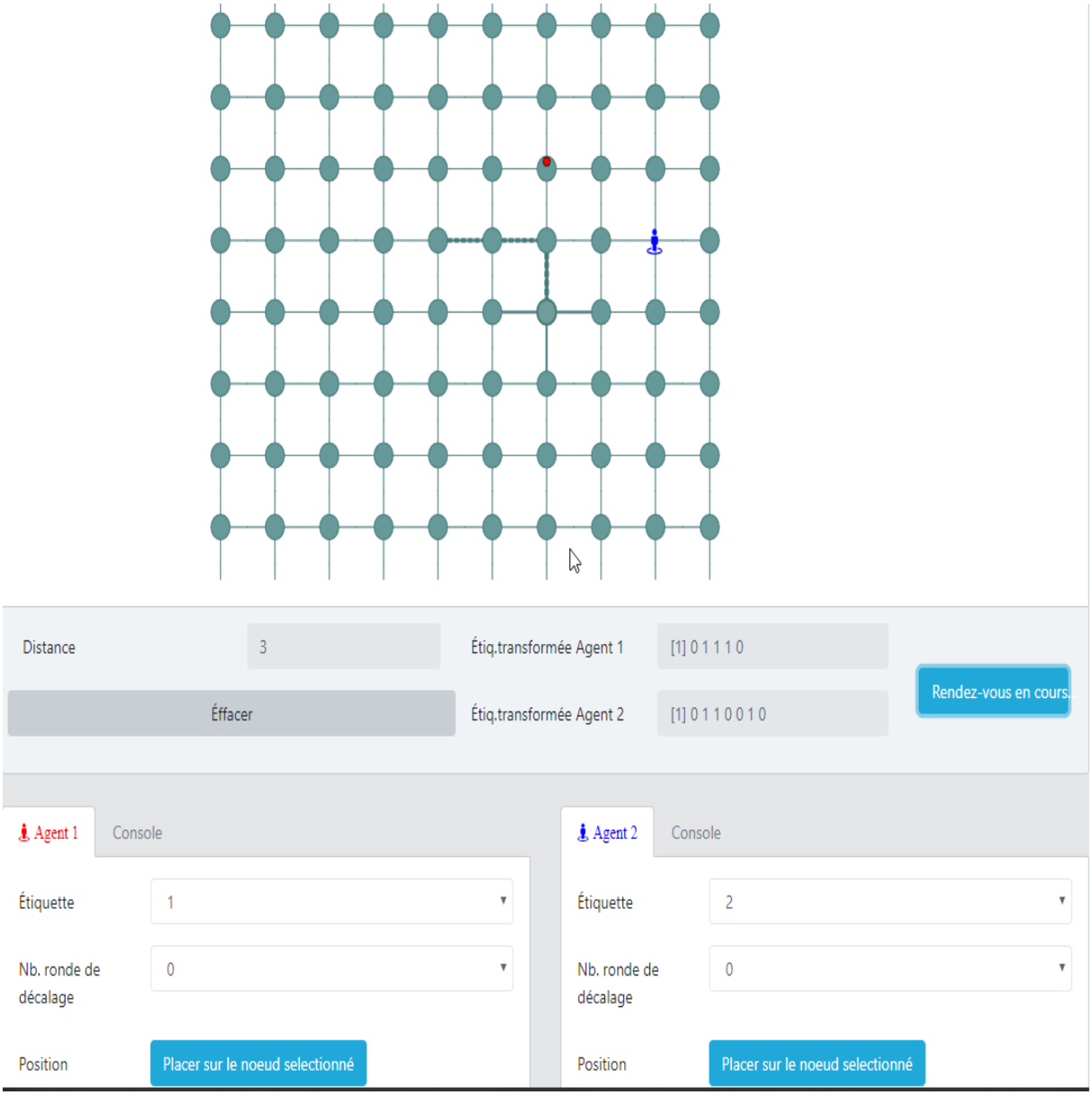

FIGURE 6.7 – Exécution de la simulation de l'algorithme de rendez-vous dans une grille de taille 10x10

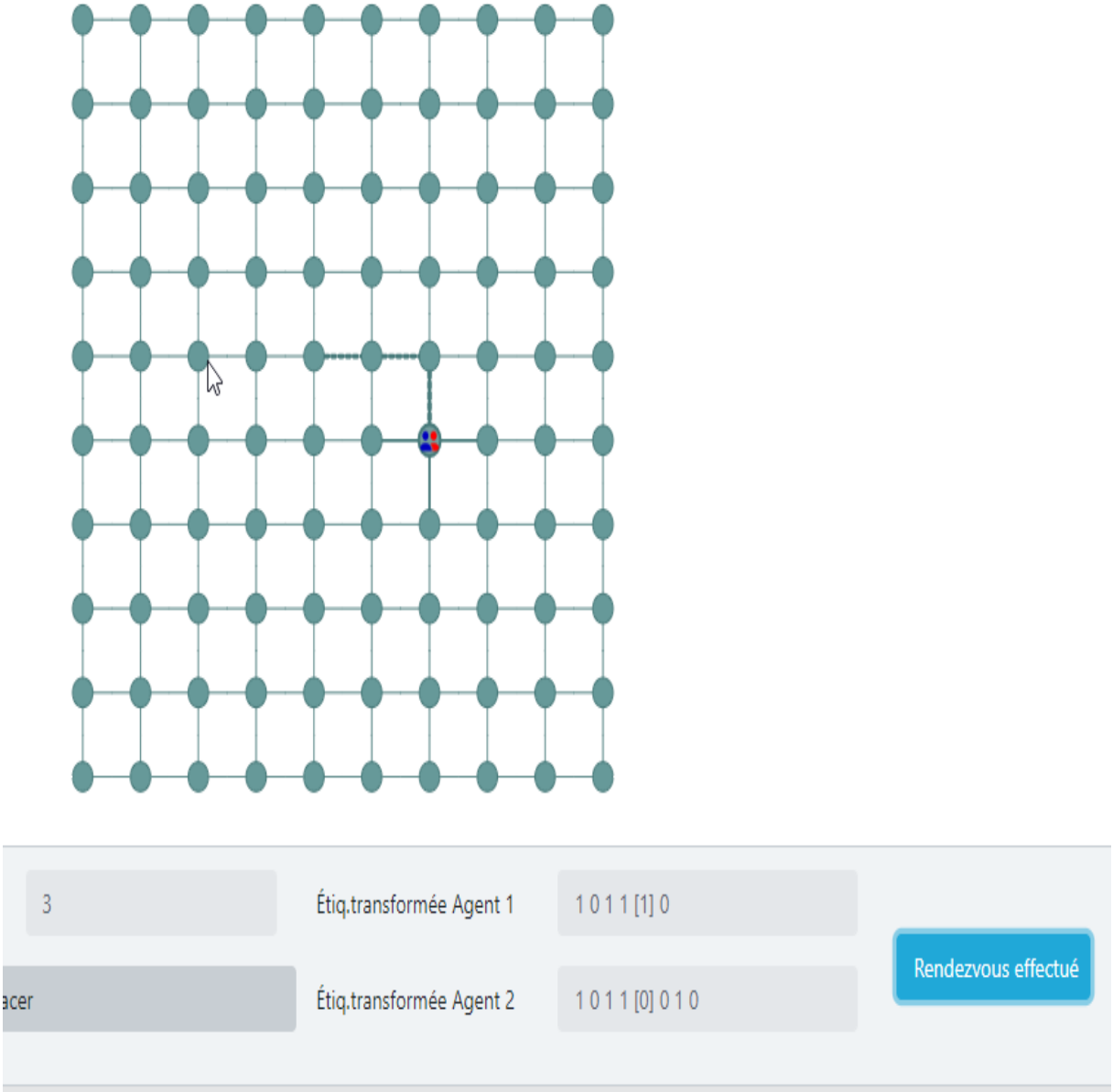

FIGURE 6.8 – Fin de la simulation de l'algorithme de rendez-vous dans une grille de taille 10x10

Chapitre 7

Conclusion

Dans ce mémoire, nous avons étudié des algorithmes de rendez-vous, dont la complexité ne dépend pas de la grandeur de l'environnement, ce qui permet de les appliquer dans de très grands environnements tout en préservant l'efficacité du rendez-vous. Nous avons présenté un algorithme déterministe qui résout le problème de rendez-vous de deux agents mobiles avec une distance initiale d'au plus D dans un graphe arbitraire de degré maximal Δ. Nous avons prouvé que cet algorithme résout le problème du rendez-vous en un temps de $O(D\Delta(\Delta-1)^{D-1}\log L)$ avec une borne inférieure $\Omega(\Delta(\Delta-1)^{D-1}+D\log L)$, où L est la grandeur de l'espace des étiquettes des agents. Dans un graphe spécifique qui est la grille infinie orientée, nous avons construit un algorithme qui résout le problème de rendez-vous de deux agents mobiles avec une distance initiale d'au plus D. Nous avons prouvé que cet algorithme résout le problème du rendez-vous en un temps de $O(D^2\log L)$ et nous avons établi une borne inférieure $\Omega(D^2 + D\log L)$ sur le temps du rendez-vous dans la grille infinie avec distance initiale d'au plus D. Enfin, en appliquant l'algorithme précédent de la grille infinie, nous avons construit un algorithme qui résout le problème de l'approche de deux agents mobiles dans un plan avec une distance euclidienne initiale au plus D. Nous avons prouvé que cet algorithme résout le problème du rendez-vous en un temps $O(D^2\log L)$.

Les problèmes ouverts ci-dessous sont énumérés pour la recherche future :

— Trouver un algorithme optimal qui résout le problème du rendez-vous dans un graphe arbitraire avec un degré maximale Δ inconnu.

— Trouver un algorithme optimal qui résout le problème du rendez-vous dans une grille infinie orientée.

— Trouver un algorithme optimal qui résout le problème de l'approche dans le plan.

Annexe A

Code Source du logiciel de simulation

Listing A.1 – Class Agent

```
1 import {Node} from './node';
2 import {Point} from './point';
3 import { angularMath } from 'angular-ts-math';
4 export class Agent {
5     public Label: string ;
6     public CurrentNode: Node;
7     public CurrentPoint: Point;
8     public Shift = 0;
9     public agent2Shit = 0;
10    constructor(label: string) {
11        this.Label = label;
12    }
13    getTransformedLabel(): Array<string> {
14        let binaryString = angularMath.numberToBinary(Number(this.
              Label));
```

```
15          let result = [];
16          const table  = binaryString.split('');
17          binaryString = table.map(function(elem){
18              return elem.repeat(2);
19          }).join('');
20          binaryString = '10' + binaryString + '10';
21          result = binaryString.split('');
22          console.log(result);
23         return result;
24      }
25      getTransformedLabelWithShift(): Array<string> {
26          let binaryString = angularMath.numberToBinary(Number(this.
               Label));
27          let result = [];
28          const table  = binaryString.split('');
29          binaryString = table.map(function(elem){
30              return elem.repeat(2);
31          }).join('');
32          binaryString = '10' + binaryString + '10';
33          result = binaryString.split('');
34          const shift = [];
35          for ( let i = 0; i < this.Shift; ++ i ) {
36              shift.push('*');
37          }
38          result = shift.concat(result);
39          console.log(result);
40          return result;
41      }
42  }
```

Listing A.2 – Class Graph

```
1  import { Node } from './node';
2  export class Graph {
3      private Nodes: Node[] = [];
4      constructor() {}
5
6      getMaxNeighbours(): number {
7          let max = 0;
8          for (let index = 0; index < this.Nodes.length; index++) {
9              if (this.Nodes[index].Neighbours.length > max) {
10                 max = this.Nodes[index].Neighbours.length;
11             }
12         }
13         return max;
14     }
15     addNode(id: string) {
16         this.Nodes.push(new Node(id));
17     }
18     deleteNode(id: string) {
19         // remove all node edges
20         this.Nodes.forEach(node => {
21             node.deleteNeighbour(id);
22         });
23         // remove the node
24         const nodeToDelete: Node =  this.getNodeById(id);
25         const index = this.Nodes.indexOf(nodeToDelete);
26         if (index > -1) {
27             this.Nodes.splice(index, 1);
28         }
29     }
```

```
30      addEdge(source: string, destination: string) {
31          // tslint:disable-next-line:triple-equals
32          const sourceNode: Node = this.Nodes.find(n => n.Id == source
                );
33          // tslint:disable-next-line:triple-equals
34          const destinationNode: Node = this.Nodes.find(n => n.Id ==
                destination);
35          if (sourceNode !== null && destinationNode !== null) {
36              sourceNode.addNeighbour(destinationNode);
37              destinationNode.addNeighbour(sourceNode);
38          }
39      }
40      deleteEdge(source: string, destination: string) {
41          // tslint:disable-next-line:triple-equals
42          const sourceNode: Node = this.Nodes.find(n => n.Id == source
                );
43          // tslint:disable-next-line:triple-equals
44          const destinationNode: Node = this.Nodes.find(n => n.Id ==
                destination);
45          if (sourceNode !== null && destinationNode !== null) {
46              sourceNode.deleteNeighbour(destinationNode.Id);
47              destinationNode.deleteNeighbour(sourceNode.Id);
48          }
49      }
50
51      getNodeById(nodeId: string): Node {
52          // tslint:disable-next-line:triple-equals
53          const node: Node = this.Nodes.find(n => n.Id == nodeId);
54          if (node === null  ||  node === undefined) {
55              throw new Error('graph with id ' + nodeId + ' not found.
                    ');
```

```
56          }
57          return node;
58      }
59
60  }
```

Listing A.3 – Class Grid

```
1  import { Point } from './point';
2  export class Grid {
3      private Points: Point[] = [];
4      constructor() {}
5
6      addPoint(id: string) {
7          this.Points.push(new Point(id));
8      }
9      setNorth(source: string, destination: string) {
10         // tslint:disable-next-line:triple-equals
11         const sourcePoint: Point = this.Points.find(n => n.Id ==
               source);
12         // tslint:disable-next-line:triple-equals
13         const destinationPoint: Point = this.Points.find(n => n.Id
               == destination);
14         if (sourcePoint !== null && destinationPoint !== null) {
15             sourcePoint.North = destinationPoint;
16             destinationPoint.South = sourcePoint;
17         }
18     }
19     setWest(source: string, destination: string) {
20         // tslint:disable-next-line:triple-equals
```

```
21          const sourcePoint: Point = this.Points.find(n => n.Id ==
                source);
22          // tslint:disable-next-line:triple-equals
23          const destinationPoint: Point = this.Points.find(n => n.Id
                == destination);
24          if (sourcePoint !== null && destinationPoint !== null) {
25              sourcePoint.West = destinationPoint;
26              destinationPoint.East = sourcePoint;
27          }
28      }
29      getPointById(pointId: string): Point {
30          // tslint:disable-next-line:triple-equals
31          const point: Point = this.Points.find(n => n.Id == pointId);
32          if (point === null  ||  point === undefined) {
33              throw new Error('grid with id ' + pointId + ' not found.
                    ');
34          }
35          return point;
36      }
37
38  }
```

Listing A.4 – Class Movement

```
1  export class Movement {
2      public AgentId: string;
3      public From: string;
4      public To: string;
5      public EdgeId: string;
6      constructor() { }
7
```

```
8       IsSamePosition(): Boolean {
9           return String(this.From) === String(this.To);
10      }
11  }
```

Listing A.5 – Class Node

```
1  export class Node {
2      public Id: string;
3      public Neighbours: Node[] = [];
4
5      constructor(id: string) {
6          this.Id = id;
7      }
8      getNeighbourIdAtPort(portId: number) {
9          let nodeId;
10         if (portId < this.Neighbours.length) {
11             const node: Node = this.Neighbours[portId];
12             nodeId = node.Id;
13         }
14         return nodeId;
15     }
16     addNeighbour(node: Node) {
17         this.Neighbours.push(node);
18     }
19
20     deleteNeighbour(id: string) {
21         // tslint:disable-next-line:triple-equals
22         const index = this.Neighbours.findIndex(n => n.Id == id);
23         if (index > -1) {
24             this.Neighbours.splice(index, 1);
```

```
25            }
26        }
27    }
```

Listing A.6 – Class Point

```
1  export class Point {
2      public Id: string;
3      public North: Point = null;
4      public South: Point = null;
5      public East: Point = null;
6      public West: Point = null;
7      constructor(id: string) {
8          this.Id = id;
9      }
10 }
11 export enum PointDirection {
12     NORTH,
13     SOUTH,
14     EAST,
15     WEST
16 }
```

Listing A.7 – Class Dijkstra

```
1  // credits: https://github.com/SergeiGalkovskii/Dijkstra-s-algorithm
       -implementation-typescript
2  export class NodeVertex {
3      nameOfVertex: string;
4      weight: number;
5  }
```

```
6
7  export class Vertex {
8      name: string;
9      nodes: NodeVertex[];
10     weight: number;
11
12     constructor(theName: string, theNodes: NodeVertex[], theWeight:
           number) {
13         this.name = theName;
14         this.nodes = theNodes;
15         this.weight = theWeight;
16     }
17 }
18
19 export class Dijkstra {
20
21     vertices: any;
22     constructor() {
23         this.vertices = {};
24     }
25
26     addVertex(vertex: Vertex): void {
27         this.vertices[vertex.name] = vertex;
28     }
29
30     findPointsOfShortestWay(start: string, finish: string, weight:
           number): string[] {
31
32         let nextVertex: string = finish;
33         const arrayWithVertex: string[] = [];
34         while (nextVertex !== start) {
```

```
35
36              let minWeigth: number = Number.MAX_VALUE;
37              let minVertex = '';
38              for (const i of this.vertices[nextVertex].nodes) {
39                  if (i.weight + this.vertices[i.nameOfVertex].weight
                        < minWeigth) {
40                      minWeigth = this.vertices[i.nameOfVertex].weight
                            ;
41                      minVertex = i.nameOfVertex;
42                  }
43              }
44              arrayWithVertex.push(minVertex);
45              nextVertex = minVertex;
46          }
47          return arrayWithVertex;
48      }
49
50
51      findShortestWay(start: string, finish: string): string[] {
52
53          const nodes: any = {};
54          const visitedVertex: string[] = [];
55
56          // tslint:disable-next-line:forin
57          for (const i in this.vertices) {
58              if (this.vertices[i].name === start) {
59                  this.vertices[i].weight = 0;
60
61              } else {
62                  this.vertices[i].weight = Number.MAX_VALUE;
63              }
```

```
64              nodes[this.vertices[i].name] = this.vertices[i].weight;
65          }
66
67          while (Object.keys(nodes).length !== 0) {
68              const sortedVisitedByWeight: string[] = Object.keys(
                    nodes).sort((a, b) => this.vertices[a].weight - this.
                    vertices[b].weight);
69              const currentVertex: Vertex = this.vertices[
                    sortedVisitedByWeight[0]];
70              for (let j of currentVertex.nodes) {
71                  const calculateWeight: number = currentVertex.weight
                        + j.weight;
72                  if (calculateWeight < this.vertices[j.nameOfVertex].
                        weight) {
73                      this.vertices[j.nameOfVertex].weight =
                            calculateWeight;
74                  }
75              }
76              delete nodes[sortedVisitedByWeight[0]];
77          }
78          const finishWeight: number = this.vertices[finish].weight;
79          const arrayWithVertex: string[] = this.
                findPointsOfShortestWay(start, finish, finishWeight).
                reverse();
80          arrayWithVertex.push(finish, finishWeight.toString());
81          return arrayWithVertex;
82      }
83
84  }
```

Listing A.8 – Class RendezVousManager

```
1  import { Injectable, Input } from '@angular/core';
2  import { Graph } from '../data/graph';
3  import { Agent } from '../data/agent';
4  import { Network, DataSet, Node, Edge, IdType } from 'vis';
5  import { Dijkstra, Vertex } from '../algorithm/dijkstra';
6  import { Movement } from '../data/movement';
7  @Injectable()
8  export class RendezVousManager {
9
10
11   public RendezVousNodeId: String = '';
12   public distance = 0;
13   public transofrmedLabel1 = '';
14   public transofrmedLabel2 = '';
15   public FirstAgent: Agent;
16   public SecondAgent: Agent;
17   public stopAfterEachRound = false;
18   public allowRDVWithSameBit = false;
19   public agent1Console  = '';
20   public agent2Console  = '';
21   private Network: Graph = new Graph();
22   private agent1Label = '1';
23   private agent2Label = '2';
24   private initialAgent1Node: any;
25   private initialAgent2Node: any;
26   private nodes: DataSet;
27   private edges: DataSet;
28   private visNetwork: Network;
29
```

```
30  private agent1Worker;
31  private agent2Worker;
32  private agent1CurrentBit = 0;
33  private agent2CurrentBit = 0;
34  private currentAgent1Bit = '';
35  private currentAgent2Bit = '';
36  private agent1transformedLabel: any;
37  private agent2transformedLabel: any;
38  private agent1bitCounter  = 0;
39  private agent2bitCounter  = 0;
40  private syncBits = 0;
41  private traficAnimationSpeed = 8;
42  private nodeVisitTime = 1500;
43  private timer;
44  private updateTransformedLabels: () => void;
45  private updateCounter: () => void;
46  private rendezvousDone: () => any;
47  private getAgent1Console: () => any;
48  private getAgent2Console: () => any;
49  private isRDV = false;
50  private rdvNodeId = '';
51  constructor() {}
52
53  register(nodes: DataSet, edges: DataSet, visNetwork: Network) {
54    this.nodes = nodes;
55    this.edges = edges;
56    this.visNetwork = visNetwork;
57   }
58   clear(): any {
59    this.Network = new Graph();
60    this.FirstAgent = new Agent(this.agent1Label);
```

```
61      this.SecondAgent = new Agent(this.agent2Label);
62      this.agent1Console = '';
63      this.agent2Console = '';
64      this.isRDV = false;
65    }
66    addNode(id: string) {
67      this.Network.addNode(id);
68    }
69    deleteNode(id: string) {
70      this.Network.deleteNode(id);
71    }
72    addEdge(source: string, destination: string) {
73      this.Network.addEdge(source, destination);
74    }
75    deleteEdge(source: string, destination: string) {
76      this.Network.deleteEdge(source, destination);
77    }
78    setSecondAgentLabel(label: any): any {
79      this.agent2Label = label;
80      this.SecondAgent.Label = label;
81    }
82    setFirstAgentLabel(label: any): any {
83      this.agent1Label = label;
84      this.FirstAgent.Label = label;
85    }
86    setFirstAgent(label: string){
87      this.agent1Label = label;
88      this.FirstAgent = new Agent(label);
89    }
90    setSecondAgent(label: string){
91      this.agent2Label = label;
```

```
92      this.SecondAgent = new Agent(label);
93    }
94    setFirstAgentPosition(nodeId: string){
95      const node = this.Network.getNodeById(nodeId);
96      this.FirstAgent.CurrentNode = node;
97      this.calculateDistance();
98    }
99    setSecondAgentPosition(nodeId: string) {
100     const node = this.Network.getNodeById(nodeId);
101     this.SecondAgent.CurrentNode = node;
102     this.calculateDistance();
103     if (this.RendezVousNodeId !== '') {
104       const n: Node = this.nodes.get(this.RendezVousNodeId);
105       n.shape = 'ellipse';
106       n.image = '';
107       n.x = undefined;
108       n.y = undefined;
109       this.nodes.update(n);
110     }
111   }
112   calculateDistance() {
113     if (this.SecondAgent.CurrentNode != null && this.FirstAgent.
          CurrentNode != null) {
114       const dijkstra = new Dijkstra();
115        this.nodes.forEach(node => {
116         const connectedEdges = this.edges.get({
117           filter: function (edge) {
118             return (edge.from === node.id || edge.to === node.id);
119           }
120         });
121         const connectVertex = [];
```

```
122         connectedEdges.forEach(e => {
123           if (e.to === node.id) {
124             connectVertex.push({ nameOfVertex: e.from, weight: 1 });
125           } else {
126             connectVertex.push({ nameOfVertex: e.to, weight: 1 });
127           }
128         });
129         dijkstra.addVertex(new Vertex(node.id, connectVertex, 1));
130       });
131       const path = dijkstra.findShortestWay( this.FirstAgent.
              CurrentNode.Id , this.SecondAgent.CurrentNode.Id);
132       this.distance = Number(path[path.length - 1]);
133     }
134   }
135   clearPreviousEdgeColor() {
136       // clear previous color
137       const ids = this.edges.getIds();
138       for (let index = 0; index < ids.length; index++) {
139         const id = ids[index];
140         const edge = this.edges.get(id);
141         edge.color = {color: '#669999', highlight: '#669999'};
142         edge.dashes = false;
143         edge.width = 1;
144         this.edges.update(edge);
145       }
146   }
147   run(updateTransformedLabels: () => any , updateCounter: () => any,
          rendezvousDone: () => any,
148    getAgent1Console: () => any, getAgent2Console: () => any) {
149     this.updateTransformedLabels = updateTransformedLabels;
150     this.updateCounter = updateCounter;
```

```
151     this.rendezvousDone = rendezvousDone;
152     this.getAgent1Console = getAgent1Console;
153     this.getAgent2Console = getAgent2Console;
154     this.isRDV = false;
155     this.clearPreviousEdgeColor();
156     this.initWorker();
157     this.start();
158   }
159   stop() {
160     if (this.agent2Worker !== undefined) {
161       this.agent2Worker.terminate();
162     }
163     if (this.agent1Worker !== undefined) {
164       this.agent1Worker.terminate();
165     }
166   }
167   moveAgent1ToNode(nodeId: string) {
168     const  move: Movement = new Movement();
169     move.AgentId = '1';
170     move.From = this.FirstAgent.CurrentNode.Id;
171     move.To = nodeId;
172     const tt =  this.getEdgeBetweenNodes(Number(move.From), Number(
          nodeId));
173     if (tt.length  === 0) {
174       console.log('aie 1 :' + move.From + '->' + nodeId);
175       move.EdgeId  = undefined;
176     } else {
177       move.EdgeId  = tt[0];
178     }
179     this.updateAgent1Position(move);
180
```

```
181   }
182   moveAgent2ToNode(nodeId: string) {
183     const  move: Movement = new Movement();
184     move.AgentId = '2';
185     move.From = this.SecondAgent.CurrentNode.Id;
186     move.To = nodeId;
187     const bb =  this.getEdgeBetweenNodes(Number(move.From), Number(
          nodeId));
188     if (bb.length === 0) {
189       console.log('aie 2 :' + move.From + '->' + nodeId);
190       move.EdgeId = undefined;
191     } else {
192       move.EdgeId = bb[0];
193     }
194     this.updateAgent2Position(move);
195   }
196   getEdgeBetweenNodes(source, dest) {
197     return this.edges.get().filter(function (edge) {
198         return (edge.from === source && edge.to === dest );
199     });
200   }
201   setRendezVous(nodeId: any) {
202       const rendezVousNode: Node = this.nodes.get(nodeId);
203       rendezVousNode.shape = 'circularImage';
204       rendezVousNode.image = 'assets/img/rdv.png';
205       rendezVousNode.size = 15;
206       rendezVousNode.x = undefined;
207       rendezVousNode.y = undefined;
208       this.nodes.update(rendezVousNode);
209   }
210   updateAgent1Position(move: Movement) {
```

```
211     const agent1OldNode: Node = this.nodes.get(move.From);
212     agent1OldNode.shape = 'ellipse';
213     agent1OldNode.image = '';
214     agent1OldNode.icon = undefined;
215     agent1OldNode.x = undefined;
216     agent1OldNode.y = undefined;
217     const agent1NewNode: Node = this.nodes.get(move.To);
218     agent1NewNode.shape = 'icon';
219     agent1NewNode.x = undefined;
220     agent1NewNode.y = undefined;
221     const targetNode: Node = this.Network.getNodeById(move.To);
222     if (move.EdgeId === undefined) {
223       agent1OldNode.shape = 'icon';
224       agent1OldNode.icon = {
225         face: 'FontAwesome',
226         code: '\uf21d',
227         size: 30,
228         color: 'red'
229       };
230       this.nodes.update(agent1OldNode);
231       this.FirstAgent.CurrentNode = targetNode;
232     } else {
233       this.nodes.update(agent1OldNode);
234       this.visNetwork.animateTraffic([{edge: move.EdgeId,
          trafficSize: 5, isBackward : false}],
235         this.traficAnimationSpeed * 3 , 'red', function() {},
            function() {}, function() { } , () => {
236           const graphicNode: Node = this.nodes.get(move.To);
237           if (graphicNode.shape === 'icon')  {
238             agent1NewNode.shape = 'circularImage';
239             agent1NewNode.image = 'assets/img/rdv.png';
```

```
240             agent1NewNode.size = 15;
241           } else {
242             agent1NewNode.shape = 'icon';
243             agent1NewNode.icon = {
244               face: 'FontAwesome',
245               code: '\uf21d',
246               size: 30,
247               color: 'red'
248             };
249           }
250           this.nodes.update(agent1NewNode);
251           this.FirstAgent.CurrentNode = targetNode;
252           if (this.isRDV) {
253             this.theRendezVousIsDone();
254           }
255       });
256     }
257   }
258   updateAgent2Position(move: Movement) {
259
260     const agent2OldNode: Node = this.nodes.get(move.From);
261     agent2OldNode.shape = 'ellipse';
262     agent2OldNode.image = '';
263     agent2OldNode.icon = undefined;
264     agent2OldNode.x = undefined;
265     agent2OldNode.y = undefined;
266     const agent2NewNode: Node = this.nodes.get(move.To);
267     agent2NewNode.shape = 'icon';
268     agent2NewNode.icon = {
269       face: 'FontAwesome',
270       code: '\uf21d',
```

```
271         size: 30,
272         color: 'blue'
273       };
274       agent2NewNode.x = undefined;
275       agent2NewNode.y = undefined;
276       const targetNode = this.Network.getNodeById(move.To);
277
278       if (move.EdgeId === undefined){
279         agent2OldNode.shape = 'icon';
280         agent2OldNode.icon = {
281           face: 'FontAwesome',
282           code: '\uf21d',
283           size: 30,
284           color: 'blue'
285         };
286         this.nodes.update(agent2OldNode);
287         this.SecondAgent.CurrentNode = targetNode;
288       } else {
289         if ( String(move.From) === String(move.To)) {
290           agent2OldNode.shape = 'icon';
291           agent2OldNode.icon = {
292             face: 'FontAwesome',
293             code: '\uf21d',
294             size: 30,
295             color: 'blue'
296           };
297         }
298         this.nodes.update(agent2OldNode);
299         this.visNetwork.animateTraffic([{edge: move.EdgeId,
              trafficSize: 5, isBackward : false}],
```

```
300         this.traficAnimationSpeed * 3 , 'blue', function() {},
               function() {}, function() { } , () => {
301           const graphicNode: Node = this.nodes.get(move.To);
302           if (graphicNode.shape === 'icon' )  {
303             agent2NewNode.shape = 'circularImage';
304             agent2NewNode.image = 'assets/img/rdv.png';
305             agent2NewNode.size = 15;
306           } else {
307             agent2NewNode.shape = 'icon';
308             agent2NewNode.icon = {
309               face: 'FontAwesome',
310               code: '\uf21d',
311               size: 30,
312               color: 'blue'
313             };
314           }
315           this.nodes.update(agent2NewNode);
316           this.SecondAgent.CurrentNode = targetNode;
317           if (this.isRDV) {
318             this.theRendezVousIsDone();
319           }
320       });
321     }
322   }
323   getAgent1TransofrmedLabel() {
324     this.FirstAgent.getTransformedLabel();
325   }
326   getAgent2TransofrmedLabel() {
327     this.FirstAgent.getTransformedLabel();
328   }
329   getDeltaMax(): number {
```

```
330     return this.Network.getMaxNeighbours();
331   }
332   getBouleTime(): number {
333     const delta = this.Network.getMaxNeighbours();
334     return 2 * this.distance * delta * (Math.pow( delta - 1, this.
          distance - 1));
335   }
336   initWorker() {
337     this.initialAgent1Node = this.FirstAgent.CurrentNode;
338     this.initialAgent2Node = this.SecondAgent.CurrentNode;
339     const delta = this.Network.getMaxNeighbours();
340     console.log('Delta is:' + delta);
341     const visitedNodeCount = this.getBouleTime(); // 2 * this.
          distance * delta * (Math.pow( delta - 1, this.distance - 1));
342     const roundTime = (visitedNodeCount * this.nodeVisitTime * 2);
343
344     this.agent1Worker = new Worker('assets/algo-graph-delta.js');
345     this.agent2Worker = new Worker('assets/algo-graph-delta.js');
346     const agent1params = {
347       action: 'init',
348       params: {
349         agent : this.FirstAgent,
350         distance: this.distance,
351         maxNeighbours : delta,
352         graph : this.Network,
353         nodeVisitTime: this.nodeVisitTime,
354         shift : this.FirstAgent.Shift
355       }
356     };
357     this.agent1Worker.postMessage(agent1params);
358     const agent2params = {
```

```
359       action: 'init',
360       params: {
361         agent : this.SecondAgent,
362         distance: this.distance,
363         maxNeighbours : delta,
364         graph : this.Network,
365         nodeVisitTime : this.nodeVisitTime,
366         shift : this.SecondAgent.Shift
367       }
368     };
369     this.agent2Worker.postMessage(agent2params);
370   }
371   getTransofrmedLabelWithPos(label: Array<string>, pos: number):
          string {
372     const tmp = Object.assign([], label);
373     const shifted = [];
374     let transfLabel = '';
375     let emptyBitCounter = 0;
376     for (let index = 0; index < tmp.length; index++) {
377       const element = tmp[index];
378       if (element !== '*') {
379         shifted.push(element);
380       } else {
381         emptyBitCounter++;
382       }
383     }
384     if (tmp[pos] === '*') {
385       transfLabel = shifted.join(' ');
386     } else {
387       shifted[pos - emptyBitCounter] = '[' + shifted[pos -
            emptyBitCounter] + ']';
```

```
388         transfLabel = shifted.join(' ');
389       }
390       return transfLabel;
391     }
392     start() {
393       const  tracker = [];
394       let counter = 3;
395       this.agent1transformedLabel = this.FirstAgent.
            getTransformedLabelWithShift();
396       this.agent2transformedLabel = this.SecondAgent.
            getTransformedLabelWithShift();
397       this.agent1bitCounter  = 0;
398       this.agent2bitCounter  = 0;
399       this.transofrmedLabel1 = this.getTransofrmedLabelWithPos(this.
            agent1transformedLabel, this.agent1bitCounter);
400       this.transofrmedLabel2 = this.getTransofrmedLabelWithPos(this.
            agent2transformedLabel, this.agent2bitCounter);
401       this.updateTransformedLabels();
402       this.timer = setTimeout(() => {
403         if (this.agent1bitCounter < this.agent1transformedLabel.length
              ) {
404           this.currentAgent1Bit = this.agent1transformedLabel[this.
                agent1bitCounter];
405           this.agent1Worker.postMessage({action: 'exec'});
406           this.agent1bitCounter++;
407         }
408         if (this.agent2bitCounter < this.agent1transformedLabel.length
              ) {
409           this.currentAgent2Bit = this.agent2transformedLabel[this.
                agent2bitCounter];
410           this.agent2Worker.postMessage({action: 'exec'});
```

```
411         this.agent2bitCounter++;
412       }
413     }, 3000);
414     const counterTimer = setInterval(() => {
415       this.updateCounter();
416       counter--;
417       if ( counter === 0 ){
418         counter = 3;
419         clearInterval(counterTimer);
420       }
421     }, 1000);
422     this.agent1Worker.onmessage = (event) => {
423       if (event.data.moveToNode !== undefined) {
424         const nodeId = event.data.moveToNode;
425         tracker.push(nodeId);
426         this.moveAgent1ToNode(nodeId);
427         if (this.allowRDVWithSameBit) {
428           if ( this.SecondAgent.CurrentNode.Id === nodeId) {
429             this.isRDV = true;
430           }
431         } else {
432           if (this.currentAgent2Bit === '0'  && this.SecondAgent.
                CurrentNode.Id === nodeId &&
433           this.initialAgent2Node.Id === nodeId) {
434             this.isRDV = true;
435           }
436         }
437         if (this.isRDV) {
438           this.rdvNodeId = nodeId;
439           this.theRendezVousIsDone();
440         }
```

```
441       } else if (event.data.bit !== undefined) {
442         this.syncBits += 1;
443         if (this.syncBits === 2) {
444           console.log('agent 1 bit ' + this.currentAgent1Bit + '.' +
                  this.agent1CurrentBit + ' execution done' + this.
                 syncBits);
445           if (!this.stopAfterEachRound) {
446             this.executeCurrentBit();
447           }
448         }
449         this.agent1CurrentBit = Number(event.data.bit);
450       } else if (event.data.messageToConsole !== undefined) {
451         this.agent1Console += event.data.messageToConsole;
452         this.getAgent1Console();
453       }
454     };
455     this.agent2Worker.onmessage = (event) => {
456       if (event.data.moveToNode !== undefined) {
457         const nodeId = event.data.moveToNode;
458         this.moveAgent2ToNode(nodeId);
459         if (this.allowRDVWithSameBit) {
460           if (this.FirstAgent.CurrentNode.Id === nodeId) {
461             this.isRDV = true;
462           }
463         } else {
464           if (this.currentAgent1Bit === '0' &&  this.FirstAgent.
                 CurrentNode.Id === nodeId  &&
465           this.initialAgent1Node.Id === nodeId) {
466             this.isRDV = true;
467           }
468         }
```

```
469         if (this.isRDV) {
470           this.rdvNodeId = nodeId;
471           this.theRendezVousIsDone();
472         }
473       } else if (event.data.bit !== undefined) {
474         this.syncBits += 1;
475         if (this.syncBits === 2) {
476           console.log('agent 2 bit ' + this.currentAgent2Bit + '.' +
                  this.agent2CurrentBit + ' execution done ' + this.
                 syncBits);
477           if (!this.stopAfterEachRound) {
478             this.executeCurrentBit();
479           }
480         }
481         this.agent2CurrentBit = Number(event.data.bit);
482       } else if (event.data.messageToConsole !== undefined) {
483         this.agent2Console += event.data.messageToConsole;
484         this.getAgent2Console();
485       }
486     };
487   } // start
488
489   executeCurrentBit() {
490     this.syncBits = 0;
491     this.currentAgent1Bit = this.agent1transformedLabel[this.
          agent1bitCounter];
492     this.currentAgent2Bit = this.agent2transformedLabel[this.
          agent2bitCounter];
493     this.transofrmedLabel1 = this.getTransofrmedLabelWithPos(this.
          agent1transformedLabel, this.agent1bitCounter);
```

```
494     this.transofrmedLabel2 = this.getTransofrmedLabelWithPos(this.
            agent2transformedLabel, this.agent2bitCounter);
495     this.updateTransformedLabels();
496     if ((this.agent1bitCounter + 1) < this.agent1transformedLabel.
            length) {
497       this.agent1bitCounter++;
498       this.agent1Worker.postMessage({action: 'exec'});
499     } else {
500       this.agent1Worker.postMessage({action: '*'});
501     }
502     if ((this.agent2bitCounter + 1) < this.agent2transformedLabel.
            length) {
503       this.agent2bitCounter++;
504       this.agent2Worker.postMessage({action: 'exec'});
505     } else {
506       this.agent2Worker.postMessage({action: '*'});
507     }
508   }
509   theRendezVousIsDone() {
510     console.log('OUTSIDE: agent2 find agent 1: RDV DONE:');
511     this.agent1Worker.terminate();
512     this.agent2Worker.terminate();
513     this.setRendezVous(this.rdvNodeId);
514     this.rendezvousDone();
515   }
516
517 }
```

Listing A.9 – Class GridRendezVousManager

```
1 import { Injectable, Input } from '@angular/core';
```

```
2 import { Grid } from '../data/grid';
3 import { Agent } from '../data/agent';
4 import { Network, DataSet, Node, Edge, IdType } from 'vis';
5 import { Dijkstra, Vertex } from '../algorithm/dijkstra';
6 @Injectable()
7 export class GridRendezVousManager {
8
9
10   public RendezVousNodeId: String = '';
11   public distance = 0;
12   public transofrmedLabel1 = '';
13   public transofrmedLabel2 = '';
14   public FirstAgent: Agent;
15   public SecondAgent: Agent;
16   public stopAfterEachRound = false;
17   public allowRDVWithSameBit = false;
18   public agent1Console  = '';
19   public agent2Console  = '';
20   private Network: Grid = new Grid();
21   private agent1Label = '1';
22   private agent2Label = '2';
23   private initialAgent1Node: any;
24   private initialAgent2Node: any;
25   private nodes: DataSet;
26   private edges: DataSet;
27   private visNetwork: Network;
28
29   private agent1Worker;
30   private agent2Worker;
31   private agent1CurrentBit = 0;
32   private agent2CurrentBit = 0;
```

```
33   private currentAgent1Bit = '';
34   private currentAgent2Bit = '';
35   private agent1transformedLabel: any;
36   private agent2transformedLabel: any;
37   private agent1bitCounter  = 0;
38   private agent2bitCounter  = 0;
39   private syncBits = 0;
40   private timer;
41   private updateTransformedLabels: () => void;
42   private updateCounter: () => void;
43   private rendezvousDone: () => any;
44   private getAgent1Console: () => any;
45   private getAgent2Console: () => any;
46   private isRDV = false;
47   private rdvNodeId = '';
48   private traficAnimationSpeed = 4;
49   private nodeVisitTime = 1000;
50   constructor() {}
51
52   register(nodes: DataSet, edges: DataSet, visNetwork: Network) {
53     this.nodes = nodes;
54     this.edges = edges;
55     this.visNetwork = visNetwork;
56    }
57    clear(): any {
58     this.Network = new Grid();
59     this.FirstAgent = new Agent(this.agent1Label);
60     this.SecondAgent = new Agent(this.agent2Label);
61     this.agent1Console = '';
62     this.agent2Console = '';
63     this.isRDV = false;
```

```
64   }
65   addPoint(id: string) {
66     this.Network.addPoint(id);
67   }
68   setNorth(s: string, d: string) {
69     this.Network.setNorth(s, d);
70   }
71   setWest(s: string, d: string) {
72     this.Network.setWest(s, d);
73   }
74   setSecondAgentLabel(label: any): any {
75     this.agent2Label = label;
76     this.SecondAgent.Label = label;
77   }
78   setFirstAgentLabel(label: any): any {
79     this.agent1Label = label;
80     this.FirstAgent.Label = label;
81   }
82   setFirstAgent(label: string){
83     this.agent1Label = label;
84     this.FirstAgent = new Agent(label);
85   }
86   setSecondAgent(label: string){
87     this.agent2Label = label;
88     this.SecondAgent = new Agent(label);
89   }
90   setFirstAgentPosition(pointId: string){
91     const point = this.Network.getPointById(pointId);
92     this.FirstAgent.CurrentPoint = point;
93     this.calculateDistance();
94   }
```

```
95   setSecondAgentPosition(pointId: string) {
96     const point = this.Network.getPointById(pointId);
97     this.SecondAgent.CurrentPoint = point;
98     this.calculateDistance();
99     if (this.RendezVousNodeId !== '') {
100      const n: Node = this.nodes.get(this.RendezVousNodeId);
101      n.shape = 'ellipse';
102      n.image = '';
103      n.x = undefined;
104      n.y = undefined;
105      this.nodes.update(n);
106    }
107  }
108  calculateDistance() {
109    if (this.SecondAgent.CurrentPoint != null && this.FirstAgent.
         CurrentPoint != null) {
110      const dijkstra = new Dijkstra();
111       this.nodes.forEach(node => {
112        const connectedEdges = this.edges.get({
113          filter: function (edge) {
114            return (edge.from === node.id || edge.to === node.id);
115          }
116        });
117        const connectVertex = [];
118        connectedEdges.forEach(e => {
119          if (e.to === node.id) {
120            connectVertex.push({ nameOfVertex: e.from, weight: 1 });
121          } else {
122            connectVertex.push({ nameOfVertex: e.to, weight: 1 });
123          }
124        });
```

```
125         dijkstra.addVertex(new Vertex(node.id, connectVertex, 1));
126       });
127       const path = dijkstra.findShortestWay( this.FirstAgent.
              CurrentPoint.Id , this.SecondAgent.CurrentPoint.Id);
128       this.distance = Number(path[path.length - 1]);
129     }
130   }
131   run(updateTransformedLabels: () => any , updateCounter: () => any,
          rendezvousDone: () => any,
132    getAgent1Console: () => any, getAgent2Console: () => any) {
133     this.updateTransformedLabels = updateTransformedLabels;
134     this.updateCounter = updateCounter;
135     this.rendezvousDone = rendezvousDone;
136     this.getAgent1Console = getAgent1Console;
137     this.getAgent2Console = getAgent2Console;
138     this.isRDV = false;
139     this.initWorker();
140     this.start();
141   }
142   stop() {
143     if (this.agent2Worker !== undefined) {
144       this.agent2Worker.terminate();
145     }
146     if (this.agent1Worker !== undefined) {
147       this.agent1Worker.terminate();
148     }
149   }
150   moveAgent1ToNode(nodeId: string) {
151      const oldNodePos: Node = this.nodes.get(this.FirstAgent.
           CurrentPoint.Id);
152      oldNodePos.shape = 'ellipse';
```

```
153       oldNodePos.image = '';
154       oldNodePos.x = undefined;
155       oldNodePos.y = undefined;
156       const newNodePos: Node = this.nodes.get(nodeId);
157       newNodePos.shape = 'icon';
158       newNodePos.icon = {
159         face: 'FontAwesome',
160         code: '\uf21d',
161         size: 30,
162         color: 'red'
163       };
164       newNodePos.x = undefined;
165       newNodePos.y = undefined;
166     this.FirstAgent.CurrentPoint = this.Network.getPointById(nodeId)
          ;
167     this.nodes.update([oldNodePos]);
168     // animate
169     const oldNodeEdges = this.visNetwork.getConnectedEdges(
          oldNodePos.id);
170     const newNodeEdges = this.visNetwork.getConnectedEdges(nodeId);
171     const edges = oldNodeEdges.filter(value => -1 !== newNodeEdges.
          indexOf(value));
172     const edgeId = edges[0];
173     const edge  = this.edges.get(edgeId);
174     let backward =  false;
175     if (edge.from !== oldNodePos.id ) {
176       backward = true;
177     }
178     this.visNetwork.animateTraffic([
179               {edge: edgeId, trafficSize: 5, isBackward : backward}
```

```
180     ], this.traficAnimationSpeed * 3 , 'red', function() {},
            function() {}, function() { } , () => {
181             this.nodes.update([oldNodePos, newNodePos]);
182             if (this.isRDV) {
183               this.theRendezVousIsDone();
184             }
185           });
186   }
187   moveAgent2ToNode(nodeId: string) {
188     const oldNodePos: Node = this.nodes.get(this.SecondAgent.
            CurrentPoint.Id);
189     oldNodePos.shape = 'ellipse';
190     oldNodePos.image = '';
191     oldNodePos.x = undefined;
192     oldNodePos.y = undefined;
193     const newNodePos: Node = this.nodes.get(nodeId);
194     newNodePos.shape = 'icon';
195     newNodePos.icon = {
196       face: 'FontAwesome',
197       code: '\uf21d',
198       size: 30,
199       color: 'blue'
200     };
201     newNodePos.x = undefined;
202     newNodePos.y = undefined;
203     this.SecondAgent.CurrentPoint = this.Network.getPointById(nodeId
            );
204     this.nodes.update([oldNodePos]);
205     // animate
206     const oldNodeEdges = this.visNetwork.getConnectedEdges(
            oldNodePos.id);
```

```
207     const newNodeEdges = this.visNetwork.getConnectedEdges(nodeId);
208     const edges = oldNodeEdges.filter(value => -1 !== newNodeEdges.
            indexOf(value));
209     const edgeId = edges[0];
210     const edge  = this.edges.get(edgeId);
211     let backward =  false;
212     if (edge.from !== oldNodePos.id ) {
213       backward = true;
214     }
215     this.visNetwork.animateTraffic([
216               {edge: edgeId, trafficSize: 5, isBackward : backward}
217           ], this.traficAnimationSpeed * 3, 'blue', function() {},
                function() {}, function() { } , () => {
218           this.nodes.update([oldNodePos, newNodePos]);
219           if (this.isRDV) {
220             this.theRendezVousIsDone();
221           }
222           });
223   }
224   setRendezVous(nodeId: any) {
225       const rendezVousNode: Node = this.nodes.get(nodeId);
226       rendezVousNode.shape = 'circularImage';
227       rendezVousNode.image = 'assets/img/rdv.png';
228       rendezVousNode.size = 15;
229       rendezVousNode.x = undefined;
230       rendezVousNode.y = undefined;
231       this.nodes.update(rendezVousNode);
232   }
233   getAgent1TransofrmedLabel() {
234     this.FirstAgent.getTransformedLabel();
235   }
```

```
236   getAgent2TransofrmedLabel() {
237     this.FirstAgent.getTransformedLabel();
238   }
239   initWorker() {
240     this.initialAgent1Node = this.FirstAgent.CurrentPoint;
241     this.initialAgent2Node = this.SecondAgent.CurrentPoint;
242
243     this.agent1Worker = new Worker('assets/algo-grid-spiral.js');
244     this.agent2Worker = new Worker('assets/algo-grid-spiral.js');
245     const agent1params = {
246       action: 'init',
247       params: {
248         agent : this.FirstAgent,
249         distance: this.distance,
250         graph : this.Network,
251         nodeVisitTime: this.nodeVisitTime,
252         shift : this.FirstAgent.Shift
253       }
254     };
255     this.agent1Worker.postMessage(agent1params);
256     const agent2params = {
257       action: 'init',
258       params: {
259         agent : this.SecondAgent,
260         distance: this.distance,
261         graph : this.Network,
262         nodeVisitTime : this.nodeVisitTime,
263         shift : this.SecondAgent.Shift
264       }
265     };
266     this.agent2Worker.postMessage(agent2params);
```

```
267   }
268   getTransofrmedLabelWithPos(label: Array<string>, pos: number):
          string {
269     const tmp = Object.assign([], label);
270     const shifted = [];
271     let transfLabel = '';
272     let emptyBitCounter = 0;
273     for (let index = 0; index < tmp.length; index++) {
274       const element = tmp[index];
275       if (element !== '*') {
276         shifted.push(element);
277       } else {
278         emptyBitCounter++;
279       }
280     }
281     if (tmp[pos] === '*') {
282       transfLabel = shifted.join(' ');
283     } else {
284       shifted[pos - emptyBitCounter] = '[' + shifted[pos -
              emptyBitCounter] + ']';
285       transfLabel = shifted.join(' ');
286     }
287     return transfLabel;
288   }
289   start() {
290     let counter = 3;
291     this.agent1transformedLabel = this.FirstAgent.
          getTransformedLabelWithShift();
292     this.agent2transformedLabel = this.SecondAgent.
          getTransformedLabelWithShift();
293     this.agent1bitCounter  = 0;
```

```
294     this.agent2bitCounter  = 0;
295     this.transofrmedLabel1 = this.getTransofrmedLabelWithPos(this.
            agent1transformedLabel, this.agent1bitCounter);
296     this.transofrmedLabel2 = this.getTransofrmedLabelWithPos(this.
            agent2transformedLabel, this.agent2bitCounter);
297     this.updateTransformedLabels();
298     this.timer = setTimeout(() => {
299       if (this.agent1bitCounter < this.agent1transformedLabel.length
              ) {
300         this.currentAgent1Bit = this.agent1transformedLabel[this.
                agent1bitCounter];
301         this.agent1Worker.postMessage({action: 'exec'});
302         this.agent1bitCounter++;
303       }
304       if (this.agent2bitCounter < this.agent1transformedLabel.length
              ) {
305         this.currentAgent2Bit = this.agent2transformedLabel[this.
                agent2bitCounter];
306         this.agent2Worker.postMessage({action: 'exec'});
307         this.agent2bitCounter++;
308       }
309     }, 3000);
310     const counterTimer = setInterval(() => {
311       this.updateCounter();
312       counter--;
313       if ( counter === 0 ){
314         counter = 3;
315         clearInterval(counterTimer);
316       }
317     }, 1000);
318     this.agent1Worker.onmessage = (event) => {
```

```
319         if (event.data.moveToNode !== undefined) {
320           const nodeId = event.data.moveToNode;
321           this.moveAgent1ToNode(nodeId);
322           if (this.allowRDVWithSameBit) {
323             if ( this.SecondAgent.CurrentPoint.Id === nodeId) {
324               this.isRDV = true;
325             }
326           } else {
327             if (this.currentAgent2Bit === '0'  && this.SecondAgent.
                  CurrentPoint.Id === nodeId &&
328             this.initialAgent2Node.Id === nodeId) {
329               this.isRDV = true;
330             }
331           }
332           if (this.isRDV) {
333             this.rdvNodeId = nodeId;
334             // this.theRendezVousIsDone();
335           }
336         } else if (event.data.bit !== undefined) {
337           this.syncBits += 1;
338           if (this.syncBits === 2) {
339             console.log('agent 1 bit ' + this.currentAgent1Bit + '.' +
                   this.agent1CurrentBit + ' execution done' + this.
                  syncBits);
340             if (!this.stopAfterEachRound) {
341               this.executeCurrentBit();
342             }
343           }
344           this.agent1CurrentBit = Number(event.data.bit);
345         } else if (event.data.messageToConsole !== undefined) {
346           this.agent1Console += event.data.messageToConsole;
```

```
347           this.getAgent1Console();
348         }
349       };
350       this.agent2Worker.onmessage = (event) => {
351         if (event.data.moveToNode !== undefined) {
352           const nodeId = event.data.moveToNode;
353           this.moveAgent2ToNode(nodeId);
354           if (this.allowRDVWithSameBit) {
355             if (this.FirstAgent.CurrentPoint.Id === nodeId) {
356               this.isRDV = true;
357             }
358           } else {
359             if (this.currentAgent1Bit === '0' &&  this.FirstAgent.
                  CurrentPoint.Id === nodeId  &&
360             this.initialAgent1Node.Id === nodeId) {
361               this.isRDV = true;
362             }
363           }
364           if (this.isRDV) {
365             this.rdvNodeId = nodeId;
366             // this.theRendezVousIsDone();
367           }
368         } else if (event.data.bit !== undefined) {
369           this.syncBits += 1;
370           if (this.syncBits === 2) {
371             console.log('agent 2 bit ' + this.currentAgent2Bit + '.' +
                    this.agent2CurrentBit + ' execution done ' + this.
                  syncBits);
372             if (!this.stopAfterEachRound) {
373               this.executeCurrentBit();
374             }
```

```
375         }
376         this.agent2CurrentBit = Number(event.data.bit);
377       } else if (event.data.messageToConsole !== undefined) {
378         this.agent2Console += event.data.messageToConsole;
379         this.getAgent2Console();
380       }
381     };
382   } // start
383
384   executeCurrentBit() {
385     this.syncBits = 0;
386     this.currentAgent1Bit = this.agent1transformedLabel[this.
          agent1bitCounter];
387     this.currentAgent2Bit = this.agent2transformedLabel[this.
          agent2bitCounter];
388     this.transofrmedLabel1 = this.getTransofrmedLabelWithPos(this.
          agent1transformedLabel, this.agent1bitCounter);
389     this.transofrmedLabel2 = this.getTransofrmedLabelWithPos(this.
          agent2transformedLabel, this.agent2bitCounter);
390     this.updateTransformedLabels();
391     if ((this.agent1bitCounter + 1) < this.agent1transformedLabel.
          length) {
392       this.agent1bitCounter++;
393       this.agent1Worker.postMessage({action: 'exec'});
394     } else {
395       this.agent1Worker.postMessage({action: '*'});
396     }
397     if ((this.agent2bitCounter + 1) < this.agent2transformedLabel.
          length) {
398       this.agent2bitCounter++;
399       this.agent2Worker.postMessage({action: 'exec'});
```

```
400     } else {
401       this.agent2Worker.postMessage({action: '*'});
402     }
403   }
404   theRendezVousIsDone() {
405     console.log('OUTSIDE: agent2 find agent 1: RDV DONE:');
406     this.agent1Worker.terminate();
407     this.agent2Worker.terminate();
408     this.setRendezVous(this.rdvNodeId);
409     this.rendezvousDone();
410   }
411
412 }
```

Listing A.10 – Class Graph Arbitraire

```
1  var roundTimer;
2  var subscription;
3  var agent;
4  var graph;
5  var _maxNeighbours = 3;
6  var _distance = 1;
7  var transformedLabel = [];
8  var roundCounter  = 0;
9  var visitedNodeCount = 0;
10 var rendezVousHappen = false;
11 var nodeVisitTime = 1000;
12 var shift = 0;
13 var consoleCounter = 1;
14 onmessage = function(e) {
15     if (e.data.action === 'exec'){
```

```
16          executeRoundBit();
17      }else if (e.data.action == 'init') {
18          init(e.data.params);
19      }else if (e.data.action == '*'){
20          var params = {
21              bit : '*'
22          }
23          postMessage(params);
24      }
25
26  }
27
28  function init(params){
29      this.nodeVisitTime = params.nodeVisitTime;
30      this.agent = params.agent;
31      this._distance = params.distance;
32      this._maxNeighbours = params.maxNeighbours;
33      this.graph = params.graph;
34      this.shift = params.shift;
35      this.transformedLabel = this.getTransformedLabel(this.agent.
            Label);
36      this.visitedNodeCount = 2 * this._distance * this._maxNeighbours
            *
37       (Math.pow( this._maxNeighbours - 1, this._distance - 1));
38       this.roundTime = this.visitedNodeCount * this.nodeVisitTime *
            2;
39       this.consoleCounter = 1;
40       writeToConsole('Temps (T) de parcours de la boule :' +  this.
            visitedNodeCount + '.');
41       writeToConsole('Résultat de l\'exécution de la procédure
            Transformer étiquette l:' +  this.transformedLabel + '.');
```

```
42      writeToConsole('Début de l\'exécution de l\'algorithme.');
43      console.log('T =' + this.visitedNodeCount);
44      console.log('Round Time:' + this.roundTime);
45      console.log('Agent label :' + this.agent.Label + ' transformed
            to: ' + this.transformedLabel);
46      console.log(agent.Label + 'EXECUTION INITIALIZED');
47 }
48
49 function executeRoundBit(){
50     var bit = this.transformedLabel[this.roundCounter];
51     console.log(this.agent.Label + ' : Execution BIT:' +  bit  + '
           at Position: ' + this.roundCounter);
52     writeToConsole('Début de l\'exécution du bit ' + bit + ' à la
           position ' + this.roundCounter + ' de l\'étiquette
           transformée.');
53     if (bit === '1') {
54        this.executeBitOne();
55     }else if(bit === '0'){
56         writeToConsole('Rester immobile pour un temps de 2T:'+ this.
               roundTime +'.');
57         this.wait(this.roundTime);
58     }else{
59         console.log(this.agent.Label + ' : en attente....');
60     }
61
62     if ((this.roundCounter + 1) < this.transformedLabel.length) {
63         this.roundCounter++;
64     }
65     var params = {
66         bit : bit
67     }
```

```
68      writeToConsole('Fin de l\'exécution du bit ' + bit + ' à la
            position ' + this.roundCounter + ' de l\'étiquette
            transformée.');
69      postMessage(params);
70  }
71  function executeBitOne(){
72      // console.log(this.agent.Label + ' :Start excution of bit 1');
73      writeToConsole('Début de l\'exécution de la procédure Exécution
            du bit 1.');
74      var realVisitedNodeCount = this.runThroughBall();
75      writeToConsole('Temps (alpha) de l\'exécution de la procédure
            Parcourir Boule(D):'+ realVisitedNodeCount +'.');
76      if (this.rendezVousHappen) {
77          return;
78      }
79      var timeToWait = ( 2 * this.visitedNodeCount * this.
            nodeVisitTime) - ( 2 * realVisitedNodeCount * nodeVisitTime);
80      writeToConsole('Rester Immobile pendant le temps (2T-2 alpha):'
            + timeToWait);
81      this.wait(timeToWait);
82      this.runThroughBall();
83      writeToConsole('Fin de l\'exécution de la procédure Exécution du
             bit 1.');
84  }
85  function runThroughBall(){
86      // console.log(this.agent.Label + ' :Start excution of
            Parcourir_Boule:');
87      writeToConsole('Début de l\'exécution de la procédure Parcourir
            Boule(D).');
88      writeToConsole('Début de la Génération de toutes les suites des
            ports de longueur D.(ordre lexicographique)');
```

```
89      var nodeCount = 0;
90      var pathCount = Math.pow( this._maxNeighbours, this._distance );
91      var list = [];
92      // initialize path
93      for (var i = 0; i < this._distance; i++)
94      {
95          list.push(0);
96      }
97      for (var index = 0; index < pathCount; index++) {
98          var c = list.join(',');
99          writeToConsole('Début de parcours du chemin:' + c + ' en
                revenant par le chemin rebours' );
100         nodeCount += this.goAndBackThroughPath(list);
101         writeToConsole('Fin de parcours du chemin:' + c + ' en
                revenant par le chemin rebours' );
102         list = this.getPath(list);
103         if (this.rendezVousHappen) {
104             break;
105         }
106     }
107     // console.log(this.agent.Label + 'Excution of Parcourir_Boule
            Ends');
108     writeToConsole('Fin de l\'exécution de la procédure Parcourir
            Boule(D).');
109     return nodeCount;
110 }
111 function goAndBackThroughPath(list) {
112     var nodeCount = 0;
113     var inversePath = [];
114     for (var portIndex = 0; portIndex < list.length; portIndex++) {
115         var portId = list[portIndex];
```

```
116         inversePath.push(Number(this.agent.CurrentNode.Id));
117         const nodeId = getNeighbourIdAtPort(portId);
118         // the node has a portId and it's a port to a node that was
                never visited
119         if (nodeId !== undefined && inversePath.indexOf(nodeId) < 0)
                {
120             this.agent.CurrentNode = getNodeById(nodeId);
121             // console.log('forward agent' + this.agent.Label + '
                    current node:' + this.agent.CurrentNode.Id);
122             moveToNode(nodeId);
123             nodeCount++;
124         } else {
125             break;
126         }
127         if (this.rendezVousHappen) {
128             return;
129         }
130     }
131     if (this.rendezVousHappen) {
132         return;
133     }
134     inversePath = inversePath.reverse();
135     var r = inversePath.join(',');
136     // console.log(this.agent.Label + 'Start go backward path:' +
        inversePath.join('=>'));
137     // go back through Path
138     for (var index = 0; index < inversePath.length; index++) {
139         var nId = String(inversePath[index]);
140         this.agent.CurrentNode = getNodeById(nId);
141         // console.log('backward agent' + this.agent.Label + '
                current node:' + this.agent.CurrentNode.Id);
```

```
142         moveToNode(nId);
143         nodeCount++;
144     }
145     return nodeCount;
146 }
147 function getPath(list){
148     for (var index = list.length - 1; index >= 0; index--) {
149         if ( list[index] < this._maxNeighbours) {
150             list[index]++;
151             for (var k = index + 1; k < list.length; k++) {
152                 list[k] = 0;
153             }
154             break;
155         }
156     }
157     return list;
158 }
159 function getTransformedLabel(label){
160     var binaryString = numberToBinary(label);
161     var result = [];
162     var table  = binaryString.split('');
163     binaryString = table.map(function(elem){
164         return elem.repeat(2);
165     }).join('');
166     binaryString = '10' + binaryString + '10';
167     result = binaryString.split('');
168     const shift = [];
169     for ( let i = 0; i < this.shift; i++ ) {
170         shift.push('*');
171     }
172     result = shift.concat(result);
```

```
173     return result;
174 }
175 function numberToBinary(number){
176     number = parseInt(number.toString(), 10);
177     return number.toString(2);
178 }
179 function getNeighbourIdAtPort(portId) {
180     var nodeId;
181     if (portId < this.agent.CurrentNode.Neighbours.length) {
182         var node = this.agent.CurrentNode.Neighbours[portId];
183         nodeId = node.Id;
184     }
185     return nodeId;
186 }
187 function moveToNode(nodeId){
188     var params = {
189         moveToNode : nodeId
190     }
191     postMessage(params);
192     wait(this.nodeVisitTime);
193
194 }
195 function getNodeById(nodeId) {
196     var node = graph.Nodes.find(n => n.Id == nodeId);
197     if (node === null  ||  node === undefined) {
198         throw new Error('graph with id ' + nodeId + ' not found.');
199     }
200     return node;
201 }
202 function wait(ms){
203     var start = new Date().getTime();
```

```
204     var end = start;
205     while(end < start + ms) {
206       end = new Date().getTime();
207    }
208  }
209  function writeToConsole(message){
210     var params = {
211         messageToConsole : this.consoleCounter +':' + message + "\n\
                r"
212     }
213     this.consoleCounter++;
214     postMessage(params);
215  }
```

Listing A.11 – Class Grille

```
1  var roundTimer;
2  var subscription;
3  var agent;
4  var graph;
5  var _distance = 1;
6  var transformedLabel = [];
7  var roundCounter  = 0;
8  var visitedNodeCount = 0;
9  var rendezVousHappen = false;
10 var nodeVisitTime = 1000;
11 var shift = 0;
12 var consoleCounter = 1;
13 onmessage = function(e) {
14     if (e.data.action === 'exec'){
15         executeRoundBit();
```

```
16      }else if (e.data.action == 'init') {
17          init(e.data.params);
18      }else if (e.data.action == '*'){
19          var params = {
20              bit : '*'
21          }
22          postMessage(params);
23      }
24
25  }
26  function init(params){
27      this.nodeVisitTime = params.nodeVisitTime;
28      this.agent = params.agent;
29      this._distance = params.distance;
30      this._maxNeighbours = params.maxNeighbours;
31      this.graph = params.graph;
32      this.shift = params.shift;
33      this.transformedLabel = this.getTransformedLabel(this.agent.
            Label);
34      this.visitedNodeCount = 4 * Math.pow(this._distance,2) + 4 *
            this._distance;
35       this.roundTime = this.visitedNodeCount * this.nodeVisitTime *
             2;
36       this.consoleCounter = 1;
37       writeToConsole('Temps (T) de parcours de la spirale :' +  this.
            visitedNodeCount + '.');
38       writeToConsole('Résultat de l\'exécution de la procédure
            Transformer étiquette l:' +  this.transformedLabel + '.');
39       writeToConsole('Début de l\'exécution de l\'algorithme.');
40       console.log('T =' + this.visitedNodeCount);
41       console.log('Round Time:' + this.roundTime);
```

```
42      console.log('Agent label :' + this.agent.Label + ' transformed
            to: ' + this.transformedLabel);
43      console.log(agent.Label + 'EXECUTION INITIALIZED');
44
45 }
46
47 function executeRoundBit(){
48     var bit = this.transformedLabel[this.roundCounter];
49     console.log(this.agent.Label + ' : Execution BIT:' +  bit  + '
           at Position: ' + this.roundCounter);
50     writeToConsole('Début de l\'exécution du bit ' + bit + ' à la
           position ' + this.roundCounter + ' de l\'étiquette
           transformée.');
51     if (bit === '1') {
52        this.executeBitOne();
53     }else if(bit === '0'){
54         writeToConsole('Rester immobile pour un temps de 2T:'+ this.
               roundTime +'.');
55         this.wait(this.roundTime);
56     }else{
57         console.log(this.agent.Label + ' : en attente....');
58     }
59
60     if ((this.roundCounter + 1) < this.transformedLabel.length) {
61         this.roundCounter++;
62     }
63     var params = {
64         bit : bit
65     }
```

```
66      writeToConsole('Fin de l\'exécution du bit ' + bit + ' à la
            position ' + this.roundCounter + ' de l\'étiquette
            transformée.');
67      postMessage(params);
68  }
69  function executeBitOne(){
70      // console.log(this.agent.Label + ' :Start excution of bit 1');
71      writeToConsole('Début de l\'exécution de la procédure Exécution
            du bit 1.');
72      this.runThroughSpiral();
73      if (this.rendezVousHappen) {
74          return;
75      }
76      this.runThroughSpiral();
77      writeToConsole('Fin de l\'exécution de la procédure Exécution du
            bit 1.');
78  }
79  function runThroughSpiral(){
80      writeToConsole('Début de l\'exécution de la procédure Parcourir
            Spirale(D).');
81      writeToConsole('Début de la Génération du chemin de la spirale')
            ;
82      var step = 1;
83      var max  = 2 * this._distance;
84      var path = '';
85      for(var i = 1; step < max ; i++)
86      {
87          path += 'E'.repeat(step) + 'N'.repeat(step) + 'W'.repeat(
                step+1) + 'S'.repeat(step+1);
88        step += 2;
89      }
```

```
90      path += 'E'.repeat(this._distance);
91      path += 'N'.repeat(this._distance);
92      var list = path.split('');
93      this.goAndBackThroughPath(list);
94      writeToConsole('Fin de l\'exécution de la procédure Parcourir
            Spirale(D).');
95  }
96  function goAndBackThroughPath(list) {
97
98      for (let i = 0; i < list.length; i++) {
99          var currentPoint = this.agent.CurrentPoint;
100         var direction  = list[i];
101         if (direction === 'E') {
102             this.agent.CurrentPoint  = currentPoint.East;
103         }else  if (direction === 'N') {
104             this.agent.CurrentPoint  = currentPoint.North;
105         }else  if (direction === 'W') {
106             this.agent.CurrentPoint  = currentPoint.West;
107         }else  if (direction === 'S') {
108             this.agent.CurrentPoint  = currentPoint.South;
109         }
110         moveToNode(this.agent.CurrentPoint.Id);
111         if (this.rendezVousHappen) {
112             return;
113         }
114     }
115 }
116 function getTransformedLabel(label){
117     var binaryString = numberToBinary(label);
118     var result = [];
119     var table  = binaryString.split('');
```

```
120     binaryString = table.map(function(elem){
121         return elem.repeat(2);
122     }).join('');
123     binaryString = '10' + binaryString + '10';
124     result = binaryString.split('');
125     const shift = [];
126     for ( let i = 0; i < this.shift; i++ ) {
127         shift.push('*');
128     }
129     result = shift.concat(result);
130     return result;
131 }
132 function numberToBinary(number){
133     number = parseInt(number.toString(), 10);
134     return number.toString(2);
135 }
136 function moveToNode(nodeId){
137     var params = {
138         moveToNode : nodeId
139     }
140     postMessage(params);
141     wait(this.nodeVisitTime);
142 
143 }
144 function wait(ms){
145     var start = new Date().getTime();
146     var end = start;
147     while(end < start + ms) {
148       end = new Date().getTime();
149    }
150  }
```

```
151 function writeToConsole(message){
152     var params = {
153         messageToConsole : this.consoleCounter +':' + message + "\n\
                r"
154     }
155     this.consoleCounter++;
156     postMessage(params);
157 }
```

Bibliographie

[1] Agathangelou, C., Georgiou, C., and Mavronicolas, M. A distributed algorithm for gathering many fat mobile robots in the plane. *PODC* (2013), 250–259.

[2] Barrière, L., Flocchini, P., Fraigniaud, P., and Santoro, N. Rendezvous and election of mobile agents : impact of sens of direction. *Theory Comput. Syst. 40*, 2 (2007), 143–162.

[3] Chalopin, J., Das, S., and Kosowski, A. Constructing a map of an anonymous graph : applications of universal sequences. *International Conference On Principles Of DIstributed Systems (OPODIS)* (2010), 119–134.

[4] Chalopin, J., Das, S., and Santoro, N. Rendezvous of mobile agents in unknown graphs with faulty links. *DISC* (2007), 108–122.

[5] Cieliebak, M., Flochini, P., Prencipe, G., and Santoro, N. Distributed computing by mobile robots : Gathering. *SIAM J. COMPUTING 41* (2012), 829–879.

[6] Czyzowicz, J., Ilcinkas, D., Labourel, A., and Pelc, A. Asynchronous deterministic rendezvous in bounded terrains. *Theory Comput. Syst. 412* (2011), 6926–6937.

[7] Czyzowicz, J., Ilcinkas, D., Labourel, A., and Pelc, A. Worst-case optimal exploration of terrains with obstacles. *Information and Computation 225* (2013), 16–28.

[8] Czyzowicz, J., Kosowski, A., and Pelc, A. How to meet when you forget : log-space rendezvous in arbitrary graphs. *Distributed Computing 25* (2012), 165–178.

[9] Czyzowicz, J., Kosowski, A., and Pelc, A. Deterministic rendezvous of asynchronous bounded memory agents in polygonal terrains. *Theory Comput. Syst. 52* (2013), 179–199.

[10] Czyzowicz, J., Labourel, A., and Pelc, A. How to meet asynchronously (almost) everywhere. *ACM Transactions on Algorithms 8*, 37 (2012).

[11] Dessmark, A., Fraigniaud, P., Kowalski, D. R., and Pelc, A. Deterministic rendezvous in graphs. *Algorithmica 46* (2006), 69–96.

[12] Dieudonné, Y., Pelc, A., and Villain, V. How to meet asynchronously at polynomial cost. *SIAM J. COMPUTING 44* (2015), 844–867.

[13] Dobrev, S., Flocchini, P., Prencipe, G., and Santoro, N. Multiple agents rendezvous in a ring in spite of a black hole. *International Conference On Principles Of DIstributed Systems (OPODIS)* (2003), 34–46.

[14] Flocchini, P., Ilcinkas, D., Pelc, A., and Santoro, N. How many oblivious robots can explore a line. *Information Processing Letters 111* (2011), 1027–1031.

[15] Flocchini, P., Ilcinkas, D., Pelc, A., and Santoro, N. Computing without communicating : Ring exploration by asynchronous oblivious robots. *Algorithmica 65* (2013), 562–583.

[16] Flocchini, P., Santoro, N., Viglietta, G., and Yamashita, M. Rendezvous of two robots with constant memory. *SIROCCO* (2013), 189–200.

[17] Flochini, P., Prencipe, G., Santoro, N., and Widmayer, P. Gathering of asynchronous oblivious robots with limited visibility. *Theoretical Computer Science 337* (2005), 147–168.

[18] Guilbault, S., and Pelc, A. Asynchronous rendezvous of anonymous agents in arbitrary graphs. In *International Conference On Principles Of DIstributed Systems (OPODIS)* (2011), pp. 162–173.

[19] Izumi, T., Souissi, S., Katayama, Y., Inuzuka, N., Défago, X., Wada, K., and Yamashita, M. The gathering problem for two oblivious robots with unreliable compasses. *SIAM J. COMPUTING. 41*, 1 (2012), 26–46.

[20] Marco, G. D., Gargano, L., Kranakis, E., Krizanc, D., Pelc, A., and Vaccaro, U. Asynchronous deterministic rendezvous in graphs. *Theoretical Computer Science 355* (2006), 315–326.

[21] Miller, A., and Pelc, A. Fast rendezvous with advice. *Theoretical Computer Science 608* (2015), 190–198.

[22] Miller, A., and Pelc, A. Tradeoffs between cost and information for rendezvous and treasure hunt. *Journal of Parallel and Distributed Computing 83* (2015), 159–167.

[23] Pelc, A. Deterministic rendezvous in networks : a comprehensive survey. *Networks 59* (2012), 331–347.

[24] Schelling, T. The strategy of conflict. *Oxford University Press* (1960).

[25] Ta-Shma, A., and Zwick, U. Deterministic rendezvous, treasure hunts and strongly universal exploration sequences. In *SODA* (2007), pp. 599–608.