UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

EXPLORATION OPTIMALE D'UN ANNEAU PAR DES AGENTS MOBILES

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

ALEXANDRE LEMIEUX, ing.

DÉCEMBRE 2017

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Mémoire intitulé :

EXPLORATION OPTIMALE D'UN ANNEAU PAR DES AGENTS MOBILES

présenté par
Alexandre Lemieux, ing.

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

a été évalué par un jury composé des personnes suivantes :

Dr. Jurek Czyżowicz ..... Directeur de recherche
Dr. Kamel Adi ..... Président du jury
Dr. Andrzej Pelc ..... Membre du jury

À mon épouse, pour son support.

Table des matières

Liste des figures

Résumé

Des agents mobiles, aussi appelés *robots*, débutant à des positions distinctes sur un anneau, doivent parcourir tous les points de ce dernier. Dans la littérature, ce type de problèmes est appelé *exploration*. Dans les différents cas explorés ici, les agents se déplacent tous à la même vitesse. Nous cherchons un mouvement des agents qui complète l'exploration de l'anneau dans un temps minimal. Nous démontrons d'abord la solution pour deux agents mobiles.

Pour n agents, nous faisons l'observation cruciale qu'un programme d'exploration optimal peut être converti en un programme où un des agents se déplace dans une seule direction. Nous nommons ce déplacement *direct*. Cette observation nous permet d'utiliser l'algorithme d'exploration sur un segment proposé par Stec [21], résultant en un algorithme d'exploration s'exécutant en $O(n^3)$.

Nous avons par contre été en mesure de développer un algorithme original permettant à un ensemble d'agents d'explorer l'anneau en temps $O(n^2)$. Nous prouvons l'exactitude et l'optimalité de cet algorithme.

L'algorithme a été implanté et une animation contrôlée par l'utilisateur permettant de visualiser la solution a été rendue disponible en ligne.

Mots-clés : agent mobile, robot, anneau, exploration

Abstract

Mobile agents, also known as *robots*, starting at distinct positions on a ring, have to explore all its points. This is known in the literature as an *exploration* problem. In the many cases studied here, the agents move at the same speed. We want to determine the agent's movements achieving the exploration in the shortest possible time. We first demonstrate the solution for two mobile agents.

For n agents we made a crucial observation that an optimal exploration may be converted to one in which one of the agents walks in one direction only (we call such exploration *direct*). Such observation permits to use the segment exploration algorithm (ref. Stec [21]) resulting in a $O(n^3)$ ring exploration algorithm.

We were able, however, to design an original, $O(n^2)$-time algorithm generating an optimal exploration schedule using a given collection of agents. We proved the correctness and the optimality of this algorithm.

The algorithm was implemented and a user-controlled animation showing its execution was put on the Web for on-line usage.

keywords : mobile agent, robot, ring, exploration

Chapitre 1

Introduction

Ce mémoire s'inscrit dans une lignée de recherches sur l'exploration par des agents mobiles dirigées par les professeurs J. Czyżowicz et A. Pelc. Nous poursuivrons ici les études qu'ont faites Lessard [17] et Stec [21] à l'Université du Québec en Outaouais pour leur mémoire respectif.

Avec l'arrivée des véhicules autonomes sur nos routes, l'étude des agents mobiles n'aura jamais été aussi pertinente. L'exploration d'un anneau peut être imagée, par exemple, par une flotte de véhicules autonomes patrouillant les frontières d'une propriété, faisant le déneigement d'un circuit ou la collecte des ordures. Afin de minimiser les impacts écologiques ou économiques de tels travaux, il est important de déterminer la stratégie optimale à adopter pour chaque agent.

Dans le cadre de nos recherches, nous nous attarderons à l'exploration d'un anneau par un ensemble d'agents mobiles. Il est important de préciser que dans la pratique, le circuit que les agents ont à parcourir n'a pas à être circulaire.

Note : La littérature parle souvent « d'agent » et de « robot ». Dans ce mémoire le terme « agent » est privilégié, mais les deux peuvent être considérés comme synonymes.

1.1 Objectifs

L'objectif principal de ce mémoire consiste à implanter un algorithme capable de déterminer le parcours optimal nécessaire pour l'exploration d'un anneau par un ensemble d'agents mobiles. L'algorithme doit fonctionner pour n'importe quelle configuration initiale d'agents.

Dans ce mémoire, nous établirons les définitions nécessaires à l'étude des problèmes d'exploration sur l'anneau. Nous démontrerons un théorème permettant de déterminer le parcours optimal pour l'exploration d'un anneau par deux agents mobiles.

Afin de réaliser notre objectif principal pour n agents, nous planifions réaliser les objectifs intermédiaires suivants :

1. Construire un algorithme qui cherche un temps minimal nécessaire pour l'exploration ;
2. Prouver l'exactitude de cet algorithme ;
3. Montrer la complexité de temps de l'algorithme ;
4. Implanter l'algorithme.

Pour les bienfaits pédagogiques de la chose, nous planifions aussi présenter l'exécution de notre algorithme de façon visuelle.

1.2 Le modèle et le problème

Un anneau doit être exploré par un ensemble d'agents mobiles. Initialement, les agents sont placés à des positions quelconques de l'anneau. Les agents peuvent se déplacer sur l'anneau avec la même vitesse unitaire. Afin de résoudre le problème d'exploration de l'anneau, chaque point de l'anneau doit être visité par au moins un agent. L'objectif de ce travail est de planifier les mouvements de tous les agents afin que tous les points de l'anneau soient explorés en un temps minimum. L'efficacité du programme d'exploration est mesurée par le temps où l'exploration de l'anneau est complétée ou, autrement dit, le moment de l'exploration du dernier point de l'anneau. Dans ce sens, on suppose que tous les agents collaborent afin que le temps d'exploration soit minimisé.

Dans notre travail, le critère de minimisation concerne le temps nécessaire pour terminer l'exploration. Cependant, comme les agents ont la même vitesse, le temps utilisé, la distance parcourue et éventuellement l'énergie dépensée par un agent durant son parcours, sont les quantités *commensurables*. Par conséquence, la solution présentée dans notre texte sert à minimiser la distance maximale à parcourir par un agent ou la quantité d'énergie dont un agent doit posséder au départ pour que l'exploration de l'anneau soit possible.

On suppose que les caractéristiques de l'anneau (c'est-à-dire sa taille) ainsi que les positions initiales de tous les agents sont connues d'une autorité centrale qui planifie les mouvements de tous les agents.

Pour simplifier l'algorithme, nous utilisons un cercle de rayon unitaire. Si l'algorithme avait à résoudre un problème pour un cycle quelconque, il devrait en connaître les caractéristiques (circonférence.) Les agents se déplacent tous à la même vitesse. Cette vitesse est arbitraire. Les agents ont le droit de rester immobiles.

Les agents n'ont pas à communiquer entre eux pour échanger de l'information. Ils se contentent de suivre les commandes résultant de l'exécution du présent algorithme.

On utilisera le terme « programme » pour représenter le mouvement ou la séquence de mouvements qu'un agent exécutera. Puisque l'environnement est un anneau, nous parlerons de déplacement en sens horaire et antihoraire.

Chapitre 2

État des connaissances

Ce mémoire s'inscrit dans la recherche en algorithmique, plus précisément les problèmes reliés à l'exploration et la recherche par des agents mobiles.

Les différents types de territoire, les contraintes, les caractéristiques des agents et les objectifs des algorithmes font de ce domaine de recherche un champ riche. Voici un survol des écrits choisis se rapportant au sujet de l'exploration et de la fouille, avec un accent sur les recherches plus proches du problème décrit dans ce mémoire.

2.1 Mouvements discrets et continus

Selon les modèles que les chercheurs décident d'étudier, on pourra dire que les agents effectuent des mouvements discrets ou continus.

Dans le modèle discret, les agents se déplacent pas à pas. Lors de chaque déplacement (*tour* ou *round*), un agent se déplace d'une ou plusieurs unités de distance. Il n'y a pas de « demi-pas. » Ce modèle est utile pour étudier les graphes ou les arbres non pondérés [2, 4, 12].

Sur une ligne, un modèle discret pourrait s'avérer pertinent dans les cas où nous ne considérons que les valeurs entières pour représenter les distances [6, 14].

Des agents aux mouvements discrets sont aussi impliqués, sur un anneau, cette fois, dans les travaux de Gorry [16].

Lorsqu'on étudie des agents aux déplacements continus, les agents ne sont pas contraints de se déplacer pas à pas. Un agent peut couvrir une distance représentée par un nombre réel. Le temps est également représenté par un nombre réel ; on ne parlera pas ici de *tours* ou de *rounds*.

Les déplacements discrets et continus peuvent avoir lieu dans le modèle synchrone ou asynchrone. Dans le modèle synchrone, le temps utilisé pour chaque mouvement est fixe. De l'autre côté, dans le modèle asynchrone, les actions des agents (mouvements, communications, etc.) prennent un temps qui ne peut pas être prévu d'avance. Par conséquent, dans le modèle asynchrone, il est beaucoup plus difficile de synchroniser les actions de différents agents.

Un modèle asynchrone est utilisé dans plusieurs travaux [1, 3, 8, 13, 9, 17, 21]. Dans son rapport sur l'état de la recherche concernant les problèmes de rendez-vous [18], Pelc aborde à la fois les modèles synchrones et asynchrones.

2.2 Problèmes d'exploration et de fouille

2.2.1 Chasse au trésor

Le problème classique de fouille est la recherche d'un trésor. Un trésor est placé par un adversaire à une position inconnue du terrain à explorer. La solution consiste à déplacer les agents de manière à fouiller l'ensemble du terrain. On emploie parfois le terme *couverture* puisqu'il faut *couvrir* tout le terrain.

Dans la majorité des travaux où le terrain à explorer est une ligne, le périmètre d'une forme ou un graphe, les agents explorent les segments lors de leurs déplacements, c'est-à-dire : un segment parcouru par un agent est exploré.

Lors de la résolution du problème de chasse au trésor, nous assumons que le trésor est trouvé au tout dernier moment, alors que l'un des agents explore le dernier point inexploré. Plusieurs travaux traitent de problèmes de chasse au trésor (par exemple, voir [3, 8, 14, 17, 12, 21]).

La recherche d'un trou noir, tel que décrit par Dobrev et coll. [13], est un cas particulier de chasse au trésor. Ce problème a la particularité que, lorsqu'un agent trouve le trou noir, il est absorbé par ce dernier sans laisser de traces.

Les travaux de ce mémoire s'inscrivent dans la catégorie de la *chasse au trésor*.

2.2.2 Création de la carte d'un environnement

Une catégorie de problèmes à laquelle les chercheurs se sont intéressés est celle de l'exploration d'un territoire inconnu avec comme objectif d'en tracer la carte. On s'intéresse plus souvent ici à des graphes, des arbres ou des surfaces planaires. Les techniques

employées pour cartographier une surface plane comportant des obstacles [10] sont très différentes des algorithmes permettant de naviguer dans un arbre ou dans un graphe [4, 18].

2.2.3 Rendez-vous

Dans le problème du rendez-vous, plusieurs agents doivent parcourir le territoire afin de se rencontrer, c'est-à-dire occuper la même position à un moment donné t. Le rendez-vous peut avoir lieu sur une ligne, un cercle, un graphe ou une surface plane. La nature du territoire à explorer influence grandement les algorithmes nécessaires pour résoudre le problème. Dans son rapport, Pelc [18] rassemble des résultats concernant différents problèmes de rendez-vous sur un graphe.

L'anonymat des agents, la quantité de mémoire disponible pour chaque agent, la présence ou l'absence d'étiquettes à chaque nœud d'un graphe ainsi que la quantité d'information que les agents peuvent laisser dans le nœud sont toutes des facettes de ce problème discutées par Pelc [18] dans son rapport.

De leur côté, Czyżowicz et coll. [11] étudient le problème du rendez-vous lorsque les agents ont une quantité limitée de mémoire. Ils démontrent que pour se rencontrer, il est nécessaire pour les agents de posséder $\Theta(\log n)$ bits de mémoire, où n correspond à la borne supérieure du nombre de nœuds dans le graphe. Il est important que la valeur de n soit connue de l'algorithme.

2.2.4 Transport d'information

Le transport d'information est un sujet pertinent lié de près aux problèmes d'exploration et de recherche. Les agents mobiles sont positionnés dans un environnement et peuvent transporter de l'information. Lorsque deux agents se rencontrent, ils partagent leurs informations.

Les chercheurs se sont penchés sur différentes variantes du problème de transport d'information.

Les recherches de Chalopin et coll. [6] s'intéressent au transport d'une information sur une ligne, d'un point s à un point t, par des agents ayant des contraintes énergétiques. Ils arrivent à la conclusion qu'il s'agit d'un problème NP-complet.

On peut également s'intéresser à la convergence d'information (*convergecast*). Dans ces problèmes, les agents doivent se déplacer et échanger leurs informations de manière

à ce que, éventuellement, l'un d'entre eux possède toutes les informations. C'est le problème étudié par Anaya et coll. [1]. Les auteurs prouvent que, si l'environnement est une ligne, on peut déterminer si la convergence d'information est possible en temps $O(n)$. Le cas échéant, il est possible de construire une stratégie de déplacement des agents qui résulte en la convergence de l'information. Par contre, si l'environnement est un arbre, on montre dans [1] que la convergence d'information est un problème NP-difficile. Les auteurs arrivent tout de même à un algorithme distribué 2-compétitif.

Les problèmes de dissémination de l'information (*broadcast*) constituent un autre problème de transport d'information. Dans ce cas, le résultat à obtenir et la propagation (dissémination) de l'information vers tous les agents. Anaya et coll. [1] se sont aussi penchés sur la question dans le même article. Ils en arrivent à la conclusion que, comme pour la convergence de l'information, le problème peut être résolu en temps $O(n)$ lorsque l'environnement est une ligne. Lorsque l'environnement est un arbre, le problème est encore une fois NP-difficile. L'algorithme distribué qu'ils proposent est cette fois-ci 4-compétitif.

2.2.5 Évacuation

Dans les problèmes d'évacuation, les agents doivent repérer un point de sortie dans le terrain, partager cette information et, finalement, s'y rendre. On peut penser à des robots prisonniers d'un édifice en proie aux flammes.

Baeza-Yates et Schott [3] étudient ce problème pour un agent sur une droite (le problème du *chemin de vache*, ou « cow-path » en anglais) et démontrent que le ratio compétitif de ce problème est de $9d$, où d est la distance séparant la position initiale de l'agent de la sortie, et que cette dernière est inconnue de l'agent. Chrobak et coll. [7] étudient aussi ce problème pour un ensemble d'agents. Lorsque la distance d est connue de l'agent, le ratio est de $3d$. Lorsque 2 agents sont impliqués, la découverte de la sortie nécessitera des déplacements totalisant $2d$. Si tous les agents doivent également atteindre cette sortie, et que les agents peuvent communiquer à distance, la distance de $4d$ est nécessaire. Si les agents ne peuvent communiquer à distance, le ratio compétitif est de $9d$ (voir [7]).

Gorry [16] s'intéresse aussi au problème d'évacuation sur un disque. À priori, tout porte à croire que ses travaux se rapprocheront des nôtres. Après tout, la première étape de l'algorithme consiste à trouver la sortie sur la circonférence du disque (c'est-à-dire

effectuer une exploration sur un cercle.) Par contre, étant donné que, dans le problème étudié, les agents partent du centre du disque, les chercheurs peuvent faire en sorte que les agents se déplacent vers des endroits précis sur la circonférence du disque. Ils n'ont donc pas à chercher une manière d'explorer la circonférence de manière optimale puisque leur position sur la circonférence du disque est optimale au départ. Gorry [16] suggère d'ailleurs, dans la conclusion du chapitre 5, un problème d'évacuation où les agents commenceraient leur recherche à des positions arbitraires sur la circonférence du disque. Les algorithmes présentés dans ce mémoire seraient alors applicables.

2.2.6 Autres problèmes

Plusieurs autres problèmes font usage d'agents mobiles.

Dans le problème de la patrouille, les agents patrouillent la frontière d'un territoire, comme dans les travaux de Gorry [16]. On tente alors de minimiser le temps séparant deux passages d'un agent en tous points de la frontière.

Lors de l'étude des problèmes de *policiers et voleurs*, deux types d'agents mobiles existent : des policiers et des voleurs. Les policiers doivent mettre la main sur les voleurs en occupant la même position qu'eux sur le terrain à explorer. Bonato et coll. [5] approfondissent davantage l'étude de ce problème et démontrent que, lorsque les policiers peuvent *voir* ou *faire feu* sur le voleur à une distance k, le problème pour déterminer le nombre de policiers nécessaires pour assurer la capture du voleur $c_k(G)$ (où G est l'ordre du graphe) est NP-difficile.

2.3 Modèle centralisé et distribué

Lorsque les chercheurs proposent des solutions aux problèmes de recherche et d'exploration, ils le font selon un modèle centralisé ou distribué (ou les deux).

Dans un modèle centralisé, c'est une entité centrale qui exécutera les calculs nécessaires afin d'établir le programme de déplacement de chaque agent. Les agents n'ont pas à prendre de décisions. Un modèle centralisé, par exemple, est employé dans les travaux de [6, 8, 17, 12, 21]

Dans un modèle distribué, chaque agent exécute un algorithme afin de déterminer quelle stratégie de mouvement il devra appliquer [3, 14, 16]. L'algorithme exécuté par les agents est souvent le même pour tous les agents, mais ce n'est pas obligatoirement le cas.

Parfois, le déplacement de l'agent sera déterministe (paramétré par la position initiale de l'agent, son index, son étiquette, etc.) alors que dans d'autres cas, les chercheurs proposent des agents dont les déplacements seront dictés de manière aléatoire. On parle d'agents anonymes lorsque tous les agents exécutent exactement le même programme.

D'autres chercheurs abordent les deux aspects de la résolution du problème. Par exemple, c'est le cas des recherches de Anaya et coll. [1].

Les modèles centralisés s'appliquent généralement lorsque le terrain à explorer est connu. Les modèles distribués, quant à eux, sont plus appropriés pour l'exploration de terrains inconnus.

2.4 Algorithmes probabilistes

Dans l'étude des agents mobiles, on retrouve parfois les algorithmes probabilistes dans les modèles distribués. Les agents mobiles exécutent alors des algorithmes souvent simples afin de résoudre le problème donné. Après un certain temps, selon la topologie de l'environnement, les étiquettes ou paramètres des agents, ou le hasard, il devient fortement probable que les agents aient résolu le problème. Ce type d'algorithme est employé par [4, 16].

2.5 Algorithmes déterministes et approximatifs

Les algorithmes déterministes permettent de calculer avec précision la solution d'un problème [8, 17, 21].

Les algorithmes approximatifs permettent d'arriver à une réponse suffisamment proche de la solution exacte. Leur utilisation est avantageuse lorsque leur complexité est plus petite qu'un algorithme déterministe qui résoudrait le même problème. Typiquement, pour beaucoup de problèmes NP-difficiles, on cherche des solutions approximatives (c'est le cas, par exemple, dans les travaux de Arkin et coll. [2]).

Par exemple, les algorithmes approximatifs peuvent être utilisés lorsque nous cherchons à déterminer un maximum ou un minimum et que nous avons à notre disposition une fonction déterminant si une valeur x est suffisante pour résoudre le problème. En faisant une recherche binaire, il est possible de converger vers un minimum ou un maximum. La recherche se poursuit jusqu'à ce que la variation du résultat soit inférieure à la

précision souhaitée, généralement notée par ϵ. Les travaux de [2, 21] emploient des algorithmes approximatifs. Il faut par contre faire attention : si la fonction étudiée possède un « minimum local », une recherche binaire ne sera pas suffisante puisqu'elle pourrait donner un minimum local et ignorer une valeur plus petite située ailleurs. Il faudra alors utiliser des techniques plus complexes telles que des algorithmes « d'escalade de collines »(*hill climbing*). C'est une technique algorithmique typiquement appliquée en intelligence artificielle (voir [19, 20]).

2.6 Contraintes sur les agents

Les problèmes de recherche et d'exploration par des agents mobiles peuvent être complexifiés davantage afin d'obtenir des situations plus proches de la réalité. On imagine en effet que les agents auront des capacités limitées afin de réduire d'éventuels coûts de production.

2.6.1 Contraintes énergétiques

Dans les travaux de recherche sur les agents mobiles, on assume que la plus grande dépense énergétique d'un agent mobile n'est pas dédié au calcul, ni à la communication, mais plutôt à son déplacement. Plusieurs chercheurs ont donc étudié les problèmes d'exploration et de recherche pour des agents mobiles avec des contraintes énergétiques [1, 6] en supposant que l'énergie est dépensée proportionnellement à la distance parcourue par un agent.

Bien que nos recherches tentent d'établir la solution permettant de parcourir un anneau dans un temps minimum, l'étude des articles traitant des contraintes énergétiques demeure très pertinente. En effet, puisque tous les agents se déplacent à une vitesse constante, la solution offrant un temps d'exploration minimum sera aussi celle qui nécessitera la plus faible réserve d'énergie individuelle (seulement si tous les agents possèdent la même réserve d'énergie au départ.)

2.6.2 Contraintes de communication

Les variations quant à la portée des dispositifs de communication des agents mobiles complexifient les problèmes d'exploration et de recherche.

Plusieurs chercheurs se sont penchés sur l'étude de problèmes pour lesquels les agents n'ont pas de capacités de communication sans fil. Les agents ne peuvent alors communiquer que lorsqu'ils occupent la même position (le même nœud d'un graphe, la même position sur une ligne ou un anneau, etc.) [1, 6, 16].

2.6.3 Contraintes de mémoire

Même si la miniaturisation et le coût de la mémoire des ordinateurs ne cessent de s'améliorer, on peut imaginer que les agents soient restreints quant à la quantité de mémoire dont ils disposent. Ce pourrait être le cas, par exemple, pour des nano robots. Citons à titre d'exemple les travaux de Czyżowicz et coll. [11] et ceux de Fraigniaud et Pelc [15].

Il est intéressant de noter que, selon les travaux de Fraigniaud et Pelc [15], si les agents débutent leurs mouvements simultanément, ils peuvent réaliser le rendez-vous dans un arbre en utilisant $O(\log l + \log\log n)$ bits de mémoire, où n est le nombre de sommets et l le nombre de feuilles dans l'arbre. Cependant, Fraigniaud et coll. nous montrent que $\Omega(\log n)$ bits sont nécessaires pour réaliser un rendez-vous si les agents commencent l'exécution de leur algorithme dans un délai.

2.6.4 Contraintes liées aux essaims

Une autre contrainte que nous pouvons imaginer aux nano robots est celle de former un essaim. Dans les problèmes s'intéressant aux essaims, on suppose parfois la présence d'un très grand nombre d'agents. Par contre, on impose que l'essaim demeure « connexe. » On entend par là que, en tout temps, un agent ne peut s'éloigner de tout autre agent de l'essaim d'une distance supérieure à d. C'est le modèle employé par Czyżowicz et coll. [12]. Les auteurs montrent qu'en temps $O(n)$, où n est le nombre de sommets d'un arbre, on peut construire un parcours optimal d'un essaim explorant cet arbre.

2.7 Variations quant aux terrains explorés

Les différents types de terrains explorés seront abordés aux points 2.8, 2.9, 2.10 et 2.11. Précisons par contre ici les variations suivantes :

- La présence d'un nœud corrompu dans un graphe : Ce nœud pourrait, tel que souligné par Pelc [18], perdre le jeton qu'on y a déposé. Il pourrait aussi s'agir d'un *trou noir* qui détruirait tout agent qui visiterait ce nœud [13].
- La présence d'obstacles empêchant le déplacement ou la vision des agents : Citons par exemple les travaux de Czyżowicz et coll. [10].

2.8 Exploration de plans

Czyżowicz et coll. [10] étudient un agent mobile explorant un environnement polygonal dans un plan. Des obstacles sont présents et l'agent doit créer la carte de son environnement. Les algorithmes proposés s'intéressent à l'exploration par un agent à vision illimitée ainsi que par un agent à vision limité. L'algorithme se repose sur la décomposition du terrain en « arbres quaternaires »(*quadtree decomposition*). Les mouvements de l'agent se distinguent par trois phases : la reconnaissance, l'exploration et l'approche. Les auteurs font la preuve que les algorithmes proposés permettent l'exploration complète du plan et qu'ils convergent tous vers l'exploration (c'est-à-dire qu'ils se terminent).

Czyżowicz et coll. [10] proposent un algorithme qui génère la trajectoire d'un agent à visibilité non bornée explorant un environnement polygonal en temps $O(P+D\sqrt{k})$ où P est la valeur du périmètre, D le diamètre et k le nombre d'obstacles de l'environnement. Si l'agent a une visibilité bornée, Czyżowicz et coll. donnent un algorithme d'exploration optimale en temps $\Omega(P + A + \sqrt{Ak})$ où A est l'aire de l'environnement à explorer.

Bien qu'il s'agit ici aussi d'un problème d'exploration, le fait qu'il s'agit d'un plan (plutôt qu'une ligne) et que ce dernier soit exploré par un seul et unique agent fait en sorte que les résultats ne peuvent être appliqués à nos recherches.

2.9 Exploration de graphes

Les problèmes relatifs aux agents mobiles s'appliquent souvent aux graphes [1, 2, 4, 11, 18]. Ces graphes peuvent être orientés ou pas, pondérés ou pas.

Plusieurs des problèmes énoncés plus tôt s'appliquent aux graphes : couverture [2], convergence et distribution de l'information [1], cartographie [4], rendez-vous [11, 18].

S'il est possible de représenter l'anneau étudié dans ce mémoire comme un graphe, aucun des travaux cités plus tôt dans cette section ne s'applique à nos travaux. Tout

d'abord, notre modèle est continu alors que la majorité des travaux cités plus tôt emploient un modèle discret.

2.10 Exploration d'arbres

L'exploration d'un arbre nécessite des algorithmes différents de ceux permettant l'exploration d'un segment linéaire. En raison de sa complexité, les chercheurs qui se sont penchés sur ce type de problèmes [18, 12] modélisent les arbres de manière simplifiée.

On fera souvent l'hypothèse que les nœuds d'un arbre sont équidistants. L'unité de temps utilisée dans ces cas-ci sera le temps nécessaire pour un agent de passer d'un nœud à un nœud voisin. Les algorithmes optant pour cette modélisation ont l'avantage d'œuvrer dans le domaine des mathématiques discrètes.

2.10.1 Exploration d'un arbre par un essaim d'agents

Czyżowicz et coll. [12] étudient l'exploration d'un arbre par un essaim d'agents. Dans le cas étudié, ils ne dénombrent pas les agents et ne les suient pas de manière individuelle. Les agents doivent garder une certaine cohésion : Czyżowicz et coll. étudient les essaims « avec noyau ». Dans ce genre d'essaims, il existe un nœud c tel que pour l'ensemble des nœuds occupés par les agents, leur distance de c est inférieure ou égale au rayon r de l'essaim.

Czyżowicz et coll. présentent leur algorithme et font la preuve qu'il est exact et optimal. Ils démontrent que, pour un essaim délimité par une borne paire $d = 2p$ où $p > 1$, un arbre peut être exploré en $2(n - H - 1) - h + d$ tours. Pour un essaim délimité par une borde impaire $d = 2p + 1$ où $p > 1$, un arbre peut être exploré en $p + 2n' - m' - h' - 1$ tours. (Où n est le nombre de nœuds, m est le nombre de feuilles, h est la hauteur de l'arbre T et, finalement, H est le nombre de nœuds dont la distance de la racine est inférieure à p.)

2.11 Exploration de segments linéaires

Nous discuterons maintenant des recherches dans le domaine de l'exploration de segments linéaires. Plusieurs chercheurs [3, 6, 8, 14, 17, 21] se sont penchés sur la question et fournissent les bases de nos recherches. Il est important de noter que, si l'exploration

d'un segment linéaire semble à priori éloignée de l'exploration d'un anneau, les observations et conclusions de Lessard [17] et Stec [21] seront d'une influence majeure pour ce mémoire.

Le modèle des problèmes étudiés par Lessard et Stec se rapproche du problème présent sur plusieurs axes :

- La position des agents et les caractéristiques de l'environnement sont connues par l'algorithme ;
- Les déplacements des agents sont gouvernés par un algorithme central ;
- L'agent peut commencer, s'arrêter, changer de direction, ou terminer son mouvement en n'importe quelle position du segment ;
- L'algorithme vise à obtenir une solution optimale, c'est-à-dire un ensemble de déplacements qui feront en sorte que tout le territoire sera exploré dans un délai minimum.

2.11.1 Exploration d'un segment linéaire par deux agents

Lessard démontre dans [17] que pour un segment linéaire et deux agents, les déplacements optimaux (f_1, f_2) peuvent être calculés mathématiquement avec exactitude à l'aide de l'algorithme qu'il propose.

2.11.2 Exploration d'un segment linéaire par plusieurs agents

Stec se penche quant à lui sur l'exploration d'un segment linéaire par un nombre quelconque d'agents [21]. Il propose deux algorithmes afin de déterminer les déplacements optimaux.

Un premier algorithme (dit approximatif) applique une recherche binaire et permet d'obtenir un résultat de plus en plus précis au fur et à mesure que les itérations s'enchaînent. L'algorithme repose sur les fonctions ExploPossible(T) et PortéeRobot(...). La fonction ExploPossible(T) évalue si l'ensemble des agents peut explorer tout le segment en T unités de temps. En faisant varier T selon l'algorithme de la recherche binaire, on se rapproche de plus en plus de la valeur optimale T_{OPT}.

Un second algorithme (dit exact) permet de déterminer les déplacements optimaux en faisant la prédiction des « changements d'état à venir ». Ainsi, chaque agent reçoit un état (1, 2, 3, ou 4.) L'état d'un agent détermine sa stratégie actuelle afin d'explorer le segment. Au fur et à mesure que le temps passe, les agents changent d'état. Chaque

agent détermine quand surviendra son prochain changement d'état. À chaque itération, on détermine quel agent sera le prochain à changer d'état. Une fois l'agent dans son nouvel état, on réévalue les prédictions. On répète le tout jusqu'à ce que tout le segment ait été exploré.

Stec démontre la validité de ces deux algorithmes et évalue la complexité algorithmique de chacun d'eux. L'algorithme approximatif s'exécute en $O(n \log(\frac{L}{\epsilon}))$, n représentant le nombre d'agents, L la longueur du segment à explorer et ϵ la précision à atteindre. L'algorithme exact, quant à lui, s'exécute en $O(n^2)$, n représentant toujours le nombre d'agents.

L'algorithme exact de Stec sera d'une importance majeure pour ce mémoire.

Flocchini et coll. [14] étudient également le problème d'exploration d'un segment de droite par plusieurs agents, mais cette fois-ci les agents n'ont pas de mémoire. L'algorithme qu'ils proposent est basé sur la formation de « tas » (*towers*) et implique des mouvements discrets (voir section 2.1). Les résultats de Flocchini et coll. [14] ne sont pas directement applicables dans nos travaux.

L'exploration de segments linéaires présente quelques variantes que nous verrons maintenant.

2.11.2.1 Exploration par des agents capables de marcher et de chercher

Czyżowicz et coll. [8] proposent une variante intéressante au problème de l'exploration d'un segment linéaire. Dans le cas qu'ils présentent, les agents ont deux vitesses : une vitesse de recherche, et une vitesse de déplacement (en anglais : « walking » et « searching »). Lors de mouvements à la vitesse de déplacement, les capteurs des agents ne sont pas actifs et le territoire survolé n'est pas considéré comme « exploré ». Les agents débutent tous à l'origine. Les chercheurs étudient ici deux versions de l'algorithme : l'une explore un segment de ligne (avec une origine et une fin données) alors que la seconde explore une ligne possédant une origine, mais de longueur indéterminée. Czyżowicz et coll. proposent deux algorithmes fonctionnant en temps $O(n)$, où n est le nombre d'agents, qui calculent le temps minimal nécessaire à explorer un segment dans chacun des deux cas mentionnés ci-dessus.

2.12 Exploration d'anneaux

Les travaux de Gorry [16] se rapprochent des travaux de ce mémoire en raison de l'environnement concerné. Les problèmes et les algorithmes proposés sont différents, mais les rapprochements sont évidents. Gorry résout d'abord le problème de « localisation » : les agents sont situés à des positions arbitraires sur un anneau unitaire, mais chaque agent ne connaît que sa position et pas celle des autres agents. Le problème consiste à se déplacer sur l'anneau de manière à déterminer la position de tous les autres agents. Pour parvenir à leurs fins, les agents échangent des « témoins » chaque fois qu'ils se rencontrent sur l'anneau, un peu comme lors d'une course à relais. Il est important de noter que les agents ne se croisent jamais. Lorsqu'ils se rencontrent, ils changent de direction et se déplacent dans la direction opposée. Les algorithmes proposés par Gorry résolvent le problème de localisation *avec une forte probabilité* en $O(\log^2 n)$.

Malgré tous ces rapprochements, les résultats de Gorry sont difficilement applicables au problème d'exploration décrit dans ce mémoire.

2.13 Récapitulatif

De tous les travaux mentionnés dans ce chapitre, ceux de Gorry [16], Lessard [17] et Stec [21] sont ceux qui se rapprochent le plus de nos recherches.

Puisque les problèmes étudiés par Gorry sont différents de celui présenté dans ce mémoire, les algorithmes qu'il propose ne pourront pas servir de bases aux nôtres.

Les rapprochements entre travaux de Lessard et Stec et ceux de ce mémoire sont détaillés à la section 2.11. Plus spécifiquement, les algorithmes développés par Stec peuvent être adaptés pour résoudre le problème de recherche sur un anneau.

Chapitre 3

Préliminaires

Dans ce texte, nous avons n agents placés sur le périmètre d'un anneau.

Nous travaillerons en radian. La circonférence d'un cercle de rayon 1 est de 2π. Puisque tous les agents voyagent à la même vitesse, on peut les considérer comme égaux ou équivalents. De plus, sans perte de généralité de nos résultats, nous fixerons cette vitesse à 1 rad/sec dans la suite de ce texte.

Définition 3.1. *L'anneau A est représenté par l'ensemble des points suivants :*

$$A = [0, 2\pi) \tag{3.1}$$

On appellera le point 0 l'origine de l'anneau A.

Définition 3.2. *Le parcours d'un agent i, pour $i = 0, 1, ...n-1$, est défini par une fonction :*

$$f_i : [0, T] \to [0, 2\pi) \tag{3.2}$$

La position initiale d'un agent est définie par :

$$p_i \in [0, 2\pi) \tag{3.3}$$

donc

$$f_i(0) = p_i \tag{3.4}$$

Définition 3.3. *La vitesse d'un agent v est fixée à la vitesse unitaire 1 rad/sec.*

Définition 3.4. *Le* segment d'opération *d'un agent est défini par l'ensemble des points X visités par l'agent tel que :*

$$X = \left\{ x : \underset{0 \leq t \leq T}{\exists} x = f_i(t) \right\} \tag{3.5}$$

La position initiale de chaque agent fait partie du segment d'opération de chaque agent.

Définition 3.5. *Le coût du parcours f_i est égal à t_k, soit la durée du parcours, et est dénoté $|f_i|$. Un point $x \in [0, 2\pi)$ est visité par l'agent effectuant le parcours f_i ssi*

$$\underset{0 \leq t \leq T}{\exists} \text{ tel que } f_i(t) = x \tag{3.6}$$

Définition 3.6. *Le temps d'opération $[0, T_i]$ d'un agent est le temps lors duquel l'agent est en mouvement. Une fois T_i atteint, l'agent s'immobilise. Plus formellement :*

$$\underset{T_i}{\exists} \text{ tel que } f_i(t) = f_i(T_i) \text{ pour } t \geq T_i \tag{3.7}$$

Définition 3.7. *Un parcours d'exploration est un ensemble de fonctions $F = (f_0, f_1, f_2, ..., f_{n-1})$ tel que chaque point $x \in [0, 2\pi)$ est visité par au moins un agent.*

$$\underset{x \in A}{\forall} \underset{0 \leq i \leq n-1}{\exists} \underset{0 \leq t \leq T}{\exists} \text{ tel que } f_i(t) = x \tag{3.8}$$

Définition 3.8. *Le coût du parcours d'exploration est le maximum entre tous les coûts des parcours $f_0, f_1, f_2, ..., f_{n-1}$, et est dénoté $|F|$.*

$$|F| = \max_{0 \leq i \leq n-1} |f_i| \tag{3.9}$$

Définition 3.9. *Un parcours d'exploration $F = (f_0, f_1, f_2, ..., f_{n-1})$ est dit optimal s'il n'existe pas un parcours d'exploration $F' = (f'_0, f'_1, f'_2, ..., f'_{n-1})$ ayant un coût plus petit.*

Dans un parcours F non optimal, il est possible pour un agent s'étant immobilisé d'aider son voisin et ainsi créer un parcours d'exploration ayant un coût plus petit.

Il est évident que l'ensemble des points explorés par un agent forme un sous-ensemble connexe de points de l'anneau. Nous nommerons ce sous-ensemble le *segment d'opération* d'un agent.

Définition 3.10. *Par un segment d'anneau $[a, b]$ où $a, b \in A$, nous comprenons tous les points dans le sens horaire à partir du point a jusqu'à la position du point b.*

Notons que si $a > b$ alors le segment $[a, b]$ contient l'origine de l'anneau.

Définition 3.11. *La longueur du segment $[a, b]$ de l'anneau est donnée par la formule suivante :*

$$\text{longueur}([a, b]) = (b - a) \bmod 2\pi \tag{3.10}$$

3.1 Programmes de déplacement

Le déplacement optimal d'un robot exprimé par f_i peut être caractérisé, selon la configuration de départ, par un programme de déplacement.

Lessard [17] et Stec [21] démontrent qu'il existe des programmes de déplacement (aussi appelés *stratégies*, *état* ou, en anglais *schedule*) que les agents doivent adopter pour explorer un segment de droite de manière optimale. Ces programmes s'appliquent également pour l'exploration d'un anneau.

Ils sont basés sur les observations suivantes :

Observation 3.1. *L'ensemble de points de l'anneau explorés par un agent est un segment de l'anneau.*

Observation 3.2. *Il existe un parcours optimal de l'anneau où les segments explorés par deux agents différents ont des intérieurs disjoints.*

Observation 3.3. *Un parcours optimal de l'anneau peut être converti en un parcours dans lequel chaque agent marche à la vitesse maximale jusqu'à l'extrémité la plus rapprochée de son segment d'opération, fait volte-face, et marche ensuite (toujours à la vitesse maximale) jusqu'à l'autre extrémité de son segment d'opération.*

Comme conséquence des observations 3.1, 3.2 et 3.3 nous pouvons constater que chaque agent exécute un des deux programmes de déplacement : programme de déplacement A ou programme de déplacement B, décrits dans les sous-sections 3.1.1 et 3.1.2 respectivement.

3.1.1 Programme A

Lors du programme de déplacement A, l'agent se déplace uniquement dans une direction (voir figure 3.1.)

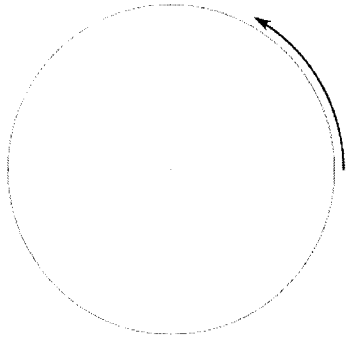

FIGURE 3.1 – Programme de déplacement A

3.1.2 Programme B

Lorsqu'un agent exécute le programme de déplacement B, il se déplace tout d'abord dans vers la borne g_i ou d_i la plus proche de la position initiale P_i, puis change de direction et se déplace dans le sens horaire jusqu'à ce qu'il atteigne l'autre borne (voir figure 3.2.)

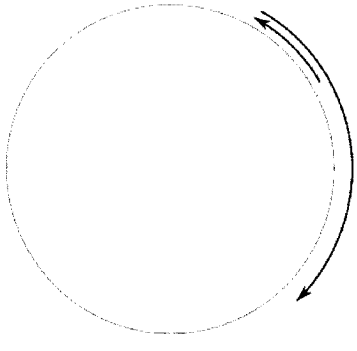

FIGURE 3.2 – Programme de déplacement B

Il est important de noter que le programme de déplacement A est un cas limite du programme de déplacement B où la borne g_i ou d_i coïncide avec la position initiale P_i de l'agent.

Chapitre 4

Exploration de l'anneau par deux agents

À l'aide des concepts décrits plus haut, étudions maintenant l'exploration de l'anneau par deux agents. Les observations que nous allons faire dans ce chapitre donneront des bases qui seront utiles pour analyser la solution pour n agents. Comme nous le verrons bientôt, il existe trois cas d'intérêt.

4.1 Exploration de l'anneau par deux agents antipodaux

Le cas de deux agents antipodaux (tout comme celui de n agents distribués de manière homogène sur l'anneau) est simple. Les deux agents effectuent des mouvements en sens horaire (ou antihoraire) jusqu'à ce qu'ils atteignent la position initiale de leur voisin. À ce moment, tout l'anneau est exploré et le problème est résolu. Par exemple, voir la figure 4.1.

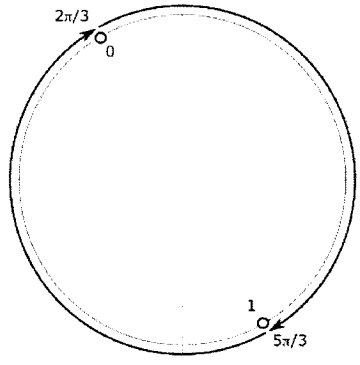

FIGURE 4.1 – Deux agents antipodaux

4.2 Exploration de l'anneau par deux agents rapprochés

Prenons une situation où deux agents 0 et 1 sont situés respectivement à $\frac{\pi}{3}$ et $\frac{2\pi}{3}$. Dans une première solution, chaque agent se dirige d'abord vers son voisin par le chemin le plus court. Les deux agents se rencontrent à $\frac{\pi}{2}$ (90°). Ils font chacun volte-face et poursuivent leur chemin jusqu'à ce qu'ils se rencontrent à nouveau à $\frac{3\pi}{2}$ (270°.) Voir la figure 4.2. Le coût d'une telle exploration est de $\frac{7\pi}{6}$.

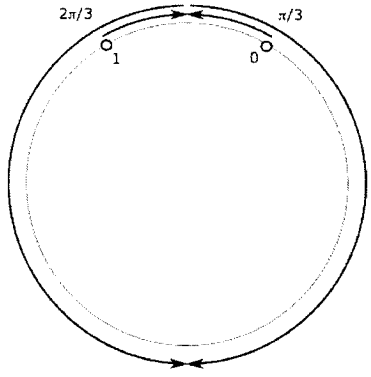

FIGURE 4.2 – Deux agents rapprochés, solution A

Reprenons la même configuration de départ, mais démontrons une autre solution optimale. Ici, l'agent 1 se déplace en sens horaire jusqu'à la position initiale de l'agent

0. Puis il fait volte-face et poursuit son exploration jusqu'à ce qu'il rencontre l'agent 0. L'agent 0, quant à lui, se déplace en sens horaire jusqu'à ce qu'il rencontre l'agent 1. Voir la figure 4.3. Le coût d'une telle exploration est également de $\frac{7\pi}{6}$.

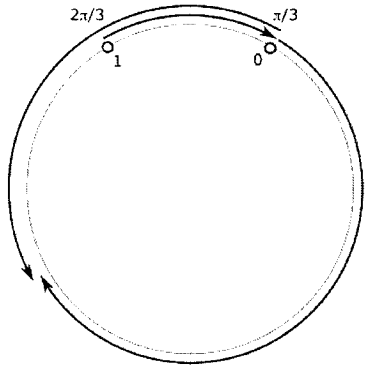

FIGURE 4.3 – Deux agents rapprochés, solution B

La figure 4.4 illustre la solution inverse, qui a toujours pour coût $\frac{7\pi}{6}$.

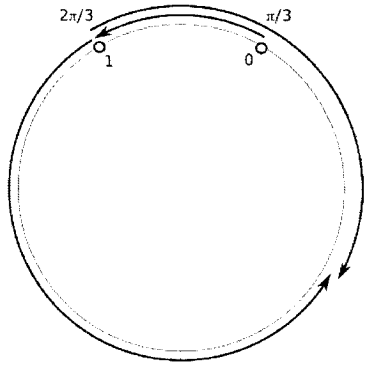

FIGURE 4.4 – Deux agents rapprochés, solution C

Il est intéressant de noter qu'il est possible de choisir n'importe quel point $x \in [\frac{\pi}{3}, \frac{2\pi}{3}]$ tel que d'abord chaque agent se dirige par la distance la plus courte vers x (l'agent 0 dans la direction antihoraire et l'agent 1 dans la direction horaire.) En atteignant le point x, chaque agent fait volte-face et continue jusqu'au temps $\frac{7\pi}{6}$. Indépendamment du point x choisi, les deux agents se rencontrent au temps $\frac{7\pi}{6}$ et l'anneau est exploré. Voir figure 4.5.

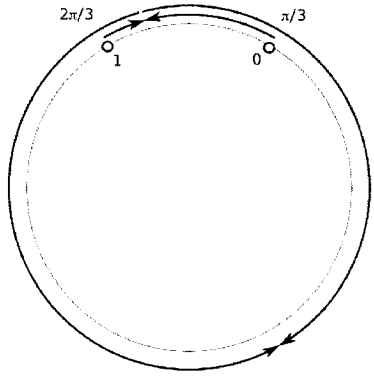

FIGURE 4.5 – Deux agents rapprochés, solution D

Peu importe le choix du point x, les deux agents explorent l'anneau en :

$$|F| = \pi + \frac{d}{2} \tag{4.1}$$

où d est la distance séparant les deux agents. L'exactitude de l'équation 4.1 sera démontrée plus tard.

Nous allons montrer plus tard que la stratégie pour deux agents rapprochés est optimale si la distance d entre les positions initiales des agents est :

$$d \leq \frac{2\pi}{5} \tag{4.2}$$

4.3 Exploration de l'anneau par deux agents éloignés

Prenons un exemple de deux agents 0 et 1 aux positions initiales 0 et $\frac{\pi}{2}$. Dans une solution optimale, l'agent 0 fait un parcours en sens horaire en s'arrêtant au point x, tandis que l'agent 1 se dirige en sens antihoraire vers le point x pour ensuite faire volte-face et continuer vers le point 0. Pour minimiser le temps de parcours, le point x doit être choisi de manière à ce que les deux agents complètent leur parcours en même temps. Conséquemment, nous devons résoudre l'équation suivante :

$$\begin{aligned} 2(x - \frac{\pi}{2}) + (\frac{\pi}{2} - 0) &= 2\pi - x \\ x &= \frac{5\pi}{6} \end{aligned}$$

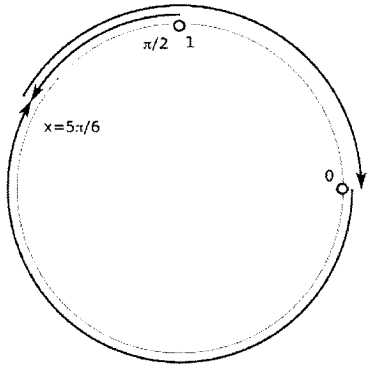

FIGURE 4.6 – Deux agents éloignés

Le coût d'exploration est déterminé par l'équation 4.3. L'exactitude de cette équation sera démontrée plus tard.

$$|F| = \frac{4\pi - d}{3} \tag{4.3}$$

Nous prouverons plus tard que la stratégie pour deux agents éloignés est optimale si la distance d entre les positions initiales des agents est telle que :

$$\frac{2\pi}{5} \leq d \leq \pi \tag{4.4}$$

Il est possible d'observer que pour $d \in [0, \frac{2\pi}{5}]$ le coût d'exploration $|F|$ de l'anneau par deux agents est une fonction linéaire de la distance séparant leur position initiale d. Cette propriété est aussi vraie pour $d \in [\frac{2\pi}{5}, \pi]$.

Si nous reprenons les équations 4.1, 4.4 et 4.3, nous pouvons alors affirmer que :

$$|F| = \begin{cases} \pi + \frac{d}{2} & \text{pour } d \in [0, \frac{2\pi}{5}] \\ \frac{4\pi - d}{3} & \text{pour } d \in [\frac{2\pi}{5}, \pi] \end{cases} \tag{4.5}$$

Ce que nous représentons graphiquement à la figure 4.7.

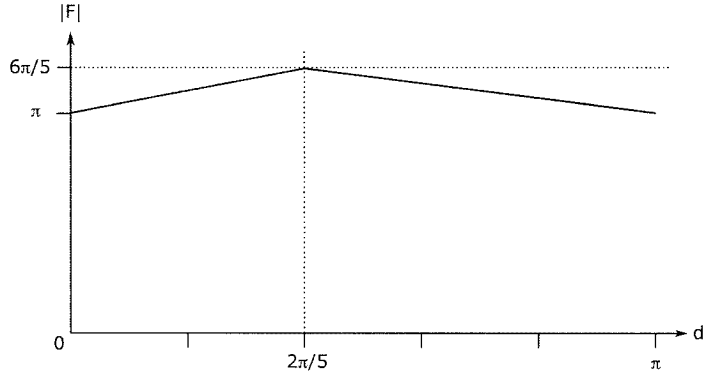

FIGURE 4.7 – Coût d'exploration de l'anneau par deux agents

Nous concluons que, peu importe l'emplacement initial des deux agents sur l'anneau, le coût d'exploration sera $\pi \leq |F| \leq \frac{6\pi}{5}$.

4.4 Preuve d'optimalité de l'exploration d'un anneau par deux agents

Nous prouverons ici l'équation 4.5 (voir aussi la figure 4.7.) Pour cela, nous prouverons tout d'abord le lemme suivant :

Lemme 4.1. *Il existe un parcours optimal d'anneau utilisant deux agents, où l'un des agents exécute le programme de déplacement A (c'est-à-dire qu'il ne change pas de direction de parcours pendant l'exploration de son segment d'opération. Voir 3.1.1.)*

Démonstration. Le preuve se fait par l'absurde. Supposons qu'il existe un parcours d'exploration F pour lequel les deux agents exécutent le programme de déplacement B. Nous allons prouver que ce parcours peut être modifié en un parcours d'exploration F' qui respecte la thèse du lemme 4.1 (c'est-à-dire au moins un agent exécute le programme de déplacement A) tel que :

$$\left|F'\right| \leq |F| \tag{4.6}$$

Nous supposons que les agents 0 et 1 ont les segments d'opération suivants : $[g_0, d_0]$ et $[g_1, d_1]$. Supposons également que ces segments d'opération couvrent tout l'anneau, de sorte que $g_0 = d_1$ et $g_1 = d_0$.

Il existe alors deux cas possibles : soit les agents débutent leur déplacement dans le même sens, soit ils le débutent dans en sens opposés.

Cas 1 : Les deux agents commencent leur déplacement dans la même direction (voir figure 4.8.)

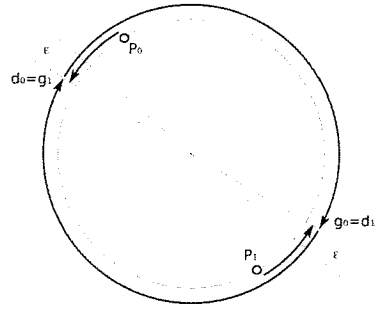

FIGURE 4.8 – Deux agents exécutant le programme de déplacement B avec départ dans le même sens

Les segments d'opération des agents sont représentés par :

$$\begin{aligned} agent\ 0 &: [g_0, d_0] \\ agent\ 1 &: [g_1, d_1] \end{aligned}$$

Supposons que nous déplacions le point de volte-face de chacun des agents de ϵ vers la position d'origine de l'agent pour $0 < \epsilon < |d_0 - P_0|$. Les segments d'opération résultants seraient alors représentés par :

$$\begin{aligned} agent\ 0 &: [g_0 - \epsilon, d_0 - \epsilon] \\ agent\ 1 &: [g_1 - \epsilon, d_1 - \epsilon] \end{aligned}$$

Le parcours de chaque agent est plus court que le parcours original par ϵ. L'existence d'un tel parcours plus court démontre que la proposition du cas 1 est invalide.

Cas 2 : Les deux agents commencent leur déplacement en sens opposés. Nous avons vu à la section 3.1.2 que les agents doivent tout d'abord se diriger vers la borne la plus

proche. Cette borne est obligatoirement située dans l'arc de cercle le plus court séparant l'agent 0 et 1. Ce cas est illustré à la figure 4.9.

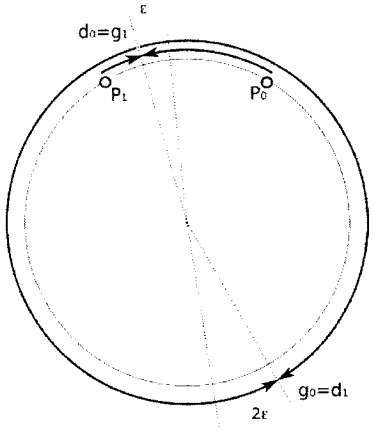

FIGURE 4.9 – Deux agents exécutant le programme de déplacement B avec départ en sens opposés

Notons qu'il est possible de déplacer le point de volte-face des agents de ϵ sans faire varier le coût d'exploration. Lorsque nous déplaçons le point de volte-face des agents de ϵ, nous notons que le point de rencontre des agents se déplace quant à lui de $2 \times \epsilon$.

$$\begin{aligned} agent\ 0 &: [g_0 - \epsilon, d_0 - 2\epsilon] \\ agent\ 1 &: [g_1 - 2\epsilon, d_1 - \epsilon] \end{aligned}$$

Si nous continuons de déplacer le point de volte-face (dans une direction ou dans l'autre), l'une des deux conditions suivantes se concrétisera éventuellement :

- Le point de volte-face coïncidera avec la position initiale d'un agent (cas 2-A.)
- Le point de rencontre coïncidera avec la position initiale d'un agent (cas 2-B.)

Dans le « cas 2-A » (voir figure 4.10), nous constatons que l'un des agents ne fait plus volte-face, et donc exécute le programme de déplacement A. Cette solution est optimale, mais vient contredire l'hypothèse de départ pour le cas 2-A.

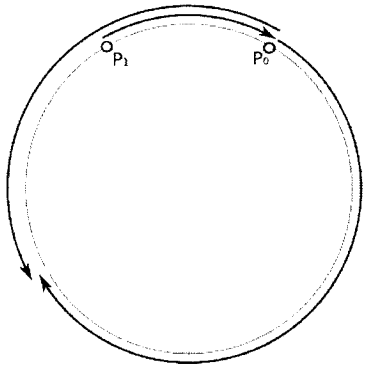

FIGURE 4.10 – Deux agents exécutant le programme de déplacement B avec un départ en sens opposés - Cas 2-A

Dans le cas 2-B on peut remplacer le parcours de l'agent 1 par le programme de déplacement A, réduisant ainsi de moitié son coût d'opération. Cette solution n'est donc pas optimale, ce qui contredit l'hypothèse de départ.

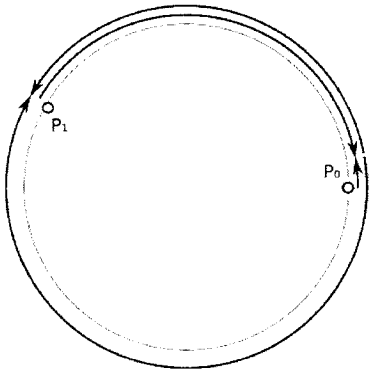

FIGURE 4.11 – Deux agents exécutant le programme de déplacement B avec un départ en sens opposés - Cas 2-B

Puisque les cas 1, 2-A et 2-B contredisent l'hypothèse de départ, nous concluons que ce dernier est invalide. Cette « preuve par contradiction » confirme le lemme 4.1.

□

Théorème 4.4.1. *Le coût d'exploration optimal $|F|$ d'un anneau par deux agents mobiles est donné par l'équation suivante :*

$$|F| = \begin{cases} \pi + \frac{d}{2} & pour\ d \in [0, \frac{2\pi}{5}] \\ \frac{4\pi - d}{3} & pour\ d \in [\frac{2\pi}{5}, \pi] \end{cases} \tag{4.7}$$

Démonstration. Supposons que nous avons un parcours optimal simple P qui respecte le lemme 4.1. Par symétrie nous pouvons supposer que c'est l'agent 0 qui emploie le programme de déplacement A. Évidemment, l'agent 0 se déplace dans la direction opposée à la direction de la plus courte distance vers l'agent 1.

Supposons que dans le parcours P, l'agent 1 termine au point x (évidemment sans visiter le point p_0 sur son parcours.) Nous dénoterons le segment d'opération de l'agent 0 ainsi : $a = [p_1, x]$. Il y a deux cas possibles :

Cas 1 : $d < |a|$

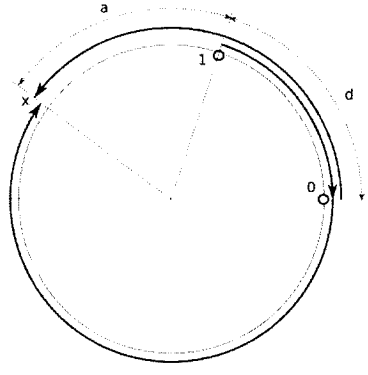

FIGURE 4.12 – Coût d'exploration optimal pour deux agents, cas 1

Tel qu'illustré à la figure 4.12, l'agent 1 se déplace de p_1 à p_0, fait volte-face, et se déplace ensuite jusqu'au point x. Le coût de ce déplacement est donc exprimé ainsi : $2d + |a|$. L'agent 0, quant à lui, se déplace afin de couvrir le reste de l'anneau, soit : $2\pi - d - |a|$. Les deux agents terminent leur déplacement en même temps. Nous pouvons donc affirmer :

$$2d + |a| = 2\pi - d - |a| \tag{4.8}$$

$$|a| = \pi - \frac{3d}{2} \tag{4.9}$$

Ce qui nous permet de déterminer le coût d'exploration des agents :

$$|F| = 2\pi - d - |a| = 2\pi - d - \pi + \frac{3d}{2} = \pi + \frac{d}{2} \tag{4.10}$$

Le cas 1 se produit lorsque $d < |a|$, c'est-à-dire $d < \pi - \frac{3d}{2}$, donc $d < \frac{2\pi}{5}$.

Cas 2 : $d \geq |a|$

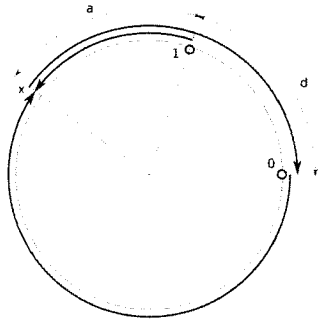

FIGURE 4.13 – Coût d'exploration optimal pour deux agents, cas 2

Dans le second cas, illustré à la figure 4.13, l'agent 1 atteint d'abord le point x avant de faire volte-face et de joindre le point p_0. Le coût d'un tel déplacement est de $2|a| + d$. Quant à lui, l'agent 0 couvre encore une fois le reste de l'anneau, soit : $2\pi - d - |a|$. Les deux agents terminent toujours leur déplacement en même temps, ce qui nous permet d'établir :

$$2|a| + d = 2\pi - d - |a| \tag{4.11}$$

$$|a| = \frac{2(\pi - d)}{3} \tag{4.12}$$

Le coût du parcours est donc déterminé ainsi :

$$|F| = 2|a| + d = \frac{4(\pi - d)}{3} + d = \frac{4\pi - d}{3} \tag{4.13}$$

En combinant les équations 4.10 et 4.13, nous obtenons l'équation 4.7 énoncée plus haut.

□

Observation 4.1. *Il est intéressant d'observer que dans le cas des deux agents sur un anneau, dans la solution optimale, les deux agents terminent leur parcours respectif en*

même temps. Ceci n'est pas le cas pour deux agents placés sur un segment de droite, comme le démontre Stec [21]. Par exemple, pour un segment $[0, 2\pi]$ *avec deux agents situés initialement aux positions* $\frac{\pi}{3}$ *et* $\frac{2\pi}{3}$ *(voir figure 4.14) dans la solution optimale, le sous-segment* $[\frac{2\pi}{3}, 2\pi]$ *doit être exploré par l'agent 1, qui termine son parcours au temps* $2\pi - \frac{2\pi}{3} = \frac{4\pi}{3}$. *L'agent* 0 *explore le reste du segment* $[0, 2\pi]$ *en temps* π *tel qu'illustré à la figure 4.14. Pour ce faire, il peut joindre le point* $\frac{2\pi}{3}$, *faire volte-face, et se déplacer jusqu'au point* 0.

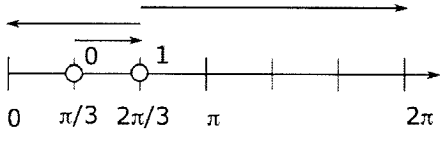

FIGURE 4.14 – Deux agents sur un segment de droite

Chapitre 5

Exploration de l'anneau par n agents

5.1 Introduction

Rappelons le théorème démontré par Stec [21]. Ce théorème nous sera utile plus tard.

Théorème 5.1.1. *(Stec [21], théorème 3) Il existe un algorithme qui trouve en temps $O(n^2)$ le plus petit temps nécessaire à un ensemble de n agents pour explorer un segment de droite, et la stratégie de déplacement employée par chaque agent pour ce faire.*

Considérons n agents mobiles $0, 1, \ldots, n-1$ aux positions initiales $\{p_0, p_1, \ldots, p_{n-1}\}$ sur l'anneau A. Les agents sont triés de sorte que $0 \leq p_i < p_j < 2\pi$, lorsque $i < j$. Pour tout agent i de l'intervalle $i = 0, 1, \ldots, n-1$, on appelle l'agent $(i+1) \mod n$ son *voisin horaire* et l'agent $(i-1) \mod n$ son *voisin antihoraire*. De la même manière, lorsque nous parlons du *prochain* agent ou de l'agent *précédent* sur l'anneau A, nous sous-entendons un ordre cyclique en sens horaire.

Nous noterons $< i, j >$, pour $0 \leq i, j \leq n-1$, une séquence d'agents consécutifs en sens horaire sur l'anneau A débutant avec l'agent i et se terminant avec l'agent j.

Remarque : Comme stipulé dans les préliminaires, le segment d'anneau $[a, b]$ est considéré dans le sens horaire à partir du point a jusqu'au point b. La longueur des segments d'anneau sont alors toujours considéré *modulo* 2π.

De façon similaire, les opérations sur les numéros des agents se font toujours *modulo* n. Par conséquent, le $i^{\text{ème}}$ voisin de l'agent j, dans le sens horaire, est l'agent $(j+i) \mod n$. Pour simplifier le texte, l'opération *modulo* ne sera pas répétée lorsque nous ferons référence à un agent $(j+i)$.

L'observation 5.1 est la généralisation de l'observation 3.2.

Observation 5.1. *En cherchant le coût de la stratégie optimale permettant d'explorer l'anneau, il est suffisant de ne considérer que les trajectoires des agents de sorte que les agents ne dépassent jamais la position initiale d'un autre agent, et qu'aucun agent ne pénètre à l'intérieur d'un segment visité par un autre agent. Par conséquent, lorsque nous mentionnons le sous-segment $[g_i, d_i]$ exploré par l'agent i, nous assumerons toujours que $g_{i+1} = d_i$, $p_{i-1} \leq g_i \leq p_i$ et $p_i \leq d_i \leq p_{i+1}$ pour $i = 0, 1, \ldots, n-1$.*

Nous dénotons par $[g_i, d_i]$ le segment d'opération de l'agent i (g_i pour l'extrémité gauche et d_i pour l'extrémité droite.)

Définition 5.1. *On dit qu'un parcours d'exploration $F = (f_0, f_1, ...f_{n-1})$ est* simple s'il respecte les conditions suivantes :

- Chaque agent i explore un segment d'anneau $[g_i, d_i]$ contenant sa position initiale, c'est-à-dire
$$p_i \in [g_i, d_i] \tag{5.1}$$
- Le parcours de chaque agent consiste en deux portions : l'agent visite d'abord l'extrémité de son segment d'opération la plus rapprochée de sa position initiale, pour ensuite parcourir tout son segment d'opération et visiter l'autre extrémité. Dans le cas où $p_i = g_i$ ou $p_i = d_i$, la première portion consiste en un déplacement nul.
- Les segments d'opération sont d'intérieurs disjoints et couvrent l'anneau au complet, c'est-à-dire
$$d_i = g_{(i+1)} \tag{5.2}$$

Conséquemment, en cherchant une exploration optimale de l'anneau A, il est suffisant de limiter la recherche d'un parcours optimal qui est un parcours simple.

Dans la suite de cette section, nous introduirons les définitions de certaines fonctions qui serviront à construire l'algorithme d'exploration optimal ainsi qu'à prouver son exactitude et sa complexité.

Soit $0 \leq x < 2\pi$ un point sur l'anneau A et $t > 0$ un intervalle de temps. Considérons i le premier agent en sens horaire à partir du point x. Par $F_1^+(x,t)$ nous définissons la longueur maximale d'un segment, en sens horaire, débutant au point x, que l'agent i est en mesure d'explorer en temps t. Pour un intervalle de temps fixe t^*, nous notons $F_1^+(x,t^*)$ la fonction à variable unique x. Grâce à l'observation 3.3, il est facile de constater que $F_1^+(x,t^*)$ peut être obtenue par la formule suivante :

$$F_1^+(x,t^*) = \begin{cases} 0 & \text{si } t^* < p_i - x \\ \frac{p_i+t^*-x}{2} & \text{si } t^* \leq p_i - x < 3t^* \\ x - p_i + t^* & \text{si } 3t^* \leq p_i - x \end{cases}$$

Effectivement, la condition $t^* \leq p_i - x < 3t^*$ signifie que l'extrémité gauche du segment explorable au temps t^* (le point x) est plus près de la position initiale p_i de l'agent que son extrémité droite. Par conséquent, la longueur $\frac{p_i+t^*-x}{2}$ du segment correspond à l'agent se déplaçant d'abord de sa position initiale p_i à l'extrémité droite x suivi par le mouvement vers son extrémité gauche $x + \frac{p_i+t^*-x}{2}$. De façon similaire, la condition $3t^* \leq p_i - x$ signifie que l'extrémité droite du segment maximal est plus rapprochée du point p_i que son extrémité gauche. La longueur $x - p_i + t^*$ correspond donc au mouvement de l'agent vers l'extrémité droite, suivi par son déplacement vers l'extrémité gauche (point x).

Comme nous l'avons noté à l'Observation 5.1, nous pouvons limiter le segment d'opération d'un agent i à la position initiale de l'agent suivant p_i. Conséquemment, nous allons modifier la fonction $F_1^+(x,t^*)$ comme suit :

$$F_1^+(x,t^*) = \begin{cases} 0 & \text{si } t^* < p_i - x \\ \max(\frac{p_i+t^*-x}{2}, p_{i+1} - x) & \text{si } t^* \leq p_i - x < 3t^* \\ \max(x - p_i + t^*, p_{i+1} - x) & \text{si } 3t^* \leq p_i - x \end{cases} \tag{5.3}$$

Cette équation nous permet de faire l'observation suivante :

Observation 5.2. *$F_1^+(x,t^*)$ est une fonction linéaire par morceaux avec au plus 3 points de non-linéarité (voir Figure 5.1).*

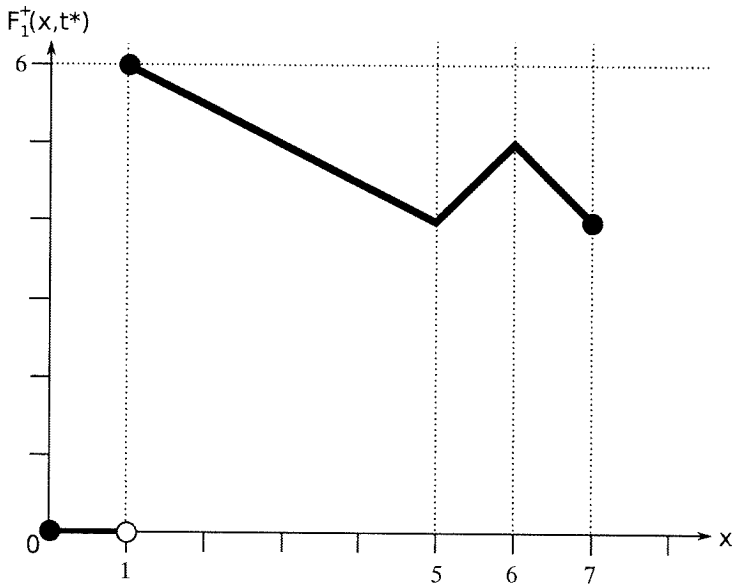

FIGURE 5.1 – Exemple de la fonction $F_1^+(x, t^*)$, une fonction linéaire par morceau. Les valeurs de cet exemple sont $t^* = 6$, $p_0 = 7$ et $p_1 = 15$.

Nous définissons $F_j^+(x, t^*)$ la fonction décrivant la longueur maximale d'un segment explorable en un temps t^* par le sous-ensemble de j agents consécutifs $i, i+1, \ldots, i+j-1$, en sens horaire débutant au point x. Comme dans la définition de la fonction $F_1^+(x, t^*)$, nous assumons que l'agent i est le premier agent dans le sens horaire dont la position initiale p_i suit le point x ($p_i \geq x$).

Nous avons le lemme suivant :

Lemme 5.1. *La valeur de $F_j^+(x, t^*)$ est définie par un algorithme glouton obtenu par la formule récursive suivante :*

$$\begin{aligned} &(1) \quad x_0 = x \\ &(2) \quad x_{m+1} = \min(x_i + F_1^+(x, t^*), p_{i+1} - x_0) \\ &(3) \quad F_j^+(x, t^*) = x_j - x_0 \end{aligned} \tag{5.4}$$

Démonstration. Nous prouvons par induction sur q que $[x_0, x_q]$ est le plus long segment explorable par les agents $i, i+1, \ldots, i+q-1$, pour $x_q \leq p_{i+q}$.

Étant donné la formule (2) de l'équation 5.4 et la définition de la fonction F_i^+, $[x_0, x_1]$ est le plus grand segment explorable par l'agent i pour $x_1 \leq p_{i+1}$, ce qui prouve la base de l'induction.

Supposons l'hypothèse d'induction selon laquelle $[x_0, x_q]$ est le plus grand segment explorable par les agents $i, i+1, \ldots, i+q-1$ pour $x_q \leq p_{i+q}$. Si $t^* < p_{q+1} - x_q$ alors ni l'agent $q+1$, ni aucun agent subséquent ne peuvent étendre le segment d'exploration $[x_0, x_q]$. Supposons alors que $t^* \geq p_{q+1} - x_q$. Par la définition de la fonction F_1^+, le plus grand segment $[x_q, y]$ explorable par l'agent $q+1$ est donné par $y = x_q + F_1^+(x, t^*)$. Puisque nous ne permettons pas à l'agent $q+1$ de dépasser le point p_{q+2} (la position de l'agent suivant) nous obtenons formule (2) de l'équation 5.4. Ceci complète la preuve par induction.

Par conséquent, la longueur du plus grand segment explorable par les agents $i, i+1, \ldots, i+j-1$ est donnée par la formule de la ligne (3) de l'équation 5.4. □

Lemme 5.2. *La fonction $F_j^+(x, t^*)$ est une fonction de x, linéaire par morceaux, avec au plus $3j$ points de non-linéarité.*

Démonstration. Selon le Lemme 5.1, la fonction $F_j^+(x, t^*)$ est obtenue par la composition (j fois) de fonctions linéaires par morceaux. Puisque chacune des fonctions F_i^+ utilisées dans cette composition, par l'Observation 5.2, a au plus trois points de non-linéarité, le nombre de points de non-linéarité de la fonction F_j^+ est limité à $3j$. □

Dénotons par $F^+(x, t^*)$ la longueur maximale d'un segment d'anneau débutant au point x, qui est explorable par un parcours d'exploration simple par l'ensemble des agents. Observons que $F^+(x, t^*) = F_j(x, t^*)$ pour certains $j \geq 0$. Dans de telles circonstances, nous dirons que j est le nombre maximal d'agents explorant en temps t^* à partir du point x. Si $F^+(x, t^*) = 0$, c'est-à-dire si le premier agent dans le sens horaire ne peut pas atteindre le point x en temps t^*, nous dénotons $F^+(x, t^*) = F_0^+(x, t^*)$.

5.2 Parcours d'explorations directs

Nous démontrerons maintenant qu'il existe un programme de déplacement résultant d'une exploration optimale dans lequel le segment d'opération d'un des agents est situé uniquement d'un côté de la position initiale de cet agent ou, en d'autres mots, que cet agent ne se déplace que dans une seule direction. Nous nommerons un tel parcours *parcours direct.* Nous prouverons d'abord le lemme suivant :

Lemme 5.3. *Considérons une valeur de x^* fixe et la fonction $F_j^+(x^*, t^*)$. Considérons aussi l'exploration d'un segment d'anneau $[x^*, x^* + F_j^+(x^*, t^*)]$ par l'algorithme glouton du*

Lemme 5.1. Supposons qu'un des agents, disons l'agent q, se déplace uniquement en sens horaire à partir de sa position initiale p_q et que chaque agent $q+1, q+2, \ldots, j$ se déplace dans deux directions depuis sa position initiale. Alors les agents $q, q+1, \ldots, j$ explorent en temps t^* le plus long segment possible débutant au p_q (segment $[p_q, p_q+F_{j-q+1}(x^*, t^*)]$) et le temps minimal pour explorer le segment $[p_q, p_q + F_{j-q+1}(x^*, t^*)]$ est t^*.

Démonstration. Prenons $m = q, q+1, \ldots, j$ et prouvons la thèse du lemme par induction sur m. Il est évident que pour $m = q$, l'agent q explore en temps t^* le sous-segment $[p_q, p_q + t^*]$, c'est-à-dire le plus grand segment débutant au point p_q. Supposons maintenant notre hypothèse d'induction selon laquelle les agents $q, q+1, \ldots, m$ explorent en temps t^* un segment $[p_q, y]$ qui est le plus long segment débutant au point p_q. Par la construction du mouvement de l'agent $m+1$ (voir la définition de F_1^+ utilisée dans le Lemme 5.1), le robot $m+1$ se déplace d'abord de sa position initiale p_{m+1} vers y si $|p_{m+1} - y| \leq \frac{t^*}{3}$. Sinon, l'agent $m+1$ se déplace d'abord dans la direction horaire de la distance maximale lui permettant d'atteindre le point y en temps t^*. Ceci optimise le point maximum exploré en sens horaire par l'agent $m+1$, ce qui conclut la preuve par induction. □

Lemme 5.4. *Prenons un point $x^* \in A$ et le temps t^*. Soit i le premier agent tel que $p_i \geq x^*$. Supposons que $F^+(x, t^*) = F_j^+(x, t^*)$ et que le parcours des agents $i, i+1, \ldots, i+j-1$ n'est pas direct. Supposons que x^* est tel que la position initiale p_q de l'agent q est exactement au centre de son segment d'opération $[x_{q-i}, x_{q-i+1}]$. Supposons que, dans le voisinage gauche du point x^* nous avons $F_q^+(x, t^*) = a_1 x + b_1$ et que, dans le voisinage droite de x^*, nous avons $F_q^+(x, t^*) = a_2 x + b_2$ pour certaines constantes a_1, b_1, a_2, b_2. Alors $a_2 > a_1$.*

Démonstration. Comme le parcours n'est pas direct, chaque agent est dans un état 1 ou 2 (c'est-à-dire, il fait demi-tour). Observons que dans ce cas, pour un changement quelconque (positif ou négatif) de l'argument x, l'extrémité droite du segment d'opération de chaque agent $i, i+1, \ldots, i+j-1$ change aussi (dans la direction positive ou négative, respectivement).

Supposons alors que pour un changement positif Δx de l'argument x, la valeur de d_{q-1} (l'extrémité droite du segment d'opération du robot $q-1$) augmente par $a\Delta x$ et la valeur de d_q augmente par $a_1 \Delta x$. Si l'agent q exécute la stratégie 1 (voir Figure 5.2) nous avons :

$$t^* = 2(d_q - p_q) + (p_q - d_{q-1}) = 2a_1\Delta x + 2(d_q - p_q) + (p_q - (d_{q-1} + a\Delta x)) \quad (5.5)$$

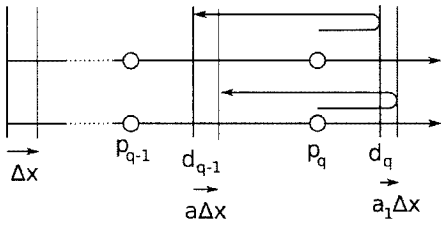

FIGURE 5.2 – $F_q^+(x, t^*)$ lorsque l'agent i exécute la stratégie 1

L'équation 5.5 implique $a_1 = \frac{a}{2}$.
Cependant, quand l'agent q exécute la stratégie 2 (voir Figure 5.2), nous avons :

$$t^* = (d_q - p_q) + 2(p_q - d_{q-1}) = (a_2\Delta x + d_q - p_q) + 2(p_q - (d_{q-1} + a\Delta x)) \quad (5.6)$$

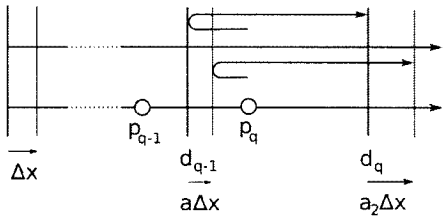

FIGURE 5.3 – $F_q^+(x, t^*)$ lorsque l'agent i exécute la stratégie 2

L'équation 5.6 implique $a_2 = 2a$.
Par conséquent, puisque $a > 0$, nous avons $a_2 > a_1$, ce qui termine la preuve. □

Nous sommes maintenant à faire la preuve du Lemme suivant :

Lemme 5.5. *Il existe un argument $\tilde{x}$ permettant de maximiser la valeur de $F_j^+(\tilde{x}, t^*)$ de sorte que l'un des agents $i, i+1, \ldots, i+q-1$ explore le segment $[\tilde{x}, \tilde{x} + F_j^+(\tilde{x}, t^*)]$ en temps t^* en ne se déplaçant que dans une direction, à partir de sa position initiale.*

Démonstration. Notons $M = F_j^+(\tilde{x}, t^*)$. Si la fonction $F_j^+(x, t^*)$ est maximisée sur un intervalle de valeurs d'intérieur non vide, alors nous assumons que la valeur de $\tilde{x}$ choisie est le point le plus à droite de cet intervalle. Selon le Lemme 5.2, la fonction $F_j^+(x, t^*)$ est linéaire par morceaux, ce qui implique que $\tilde{x}$ est un des points de non-linéarité.

Considérons un *algorithme glouton* résultant en l'exploration optimale du segment $[\tilde{x}, \tilde{x} + F_j^+(\tilde{x}, t^*)]$ par les agents $i, i+1, \ldots, i+j-1$ en temps t^*. Le Lemme 5.1 nous permet d'assumer que cet algorithme génère une série de points $x_0, x_1, \ldots, x_j$ de sorte que chaque agent m (pour $i \leq m \leq i+j-1$) explore le segment d'opération $[x_{m-i}, x_{m-i+1}]$. Supposons, par contradiction à notre Lemme, que chaque agent m explore un segment de chaque côté de sa position initiale p_m, c'est-à-dire $x_{m-i} < p_m < x_{m-i+1}$. Dans ce cas, le programme de déplacement de l'agent n'est pas *direct*. Par notre supposition de départ, le point $\tilde{x}$ doit être un point de non-linéarité de la fonction $F_j^+(x, t^*)$. Comme notre parcours n'est pas direct, un tel point de non-linéarité peut être obtenu seulement lorsque le segment d'opération $[x_{q-i}, x_{q-i+1}]$ d'un agent q et tel que la position initiale p_q est équidistante aux limites du segment $[x_{q-i}, x_{q-i+1}]$, c'est-à-dire $|p_q - x_{q-i}| = |x_{q-i+1} - p_q| = 3t^*$. Par contre, le Lemme 5.4 affirme que la pente de la fonction linéaire $F_q^+(x, t^*)$ est plus petite dans le voisinage gauche du point $\tilde{x}$ que dans le voisinage droit. Nous observons alors que dans un tel cas, $F_q^+(x, t^*)$ n'est pas maximisée (dans le voisinage de $\tilde{x}$), ce qui implique en retour que $F_j^+(x, t^*)$ n'est pas un maximum. Ceci conclut la preuve. □

Théorème 5.2.1. *Soit un anneau A et un ensemble d'agents placés sur A. En $O(n^3)$ temps nous pouvons trouver le temps minimal nécessaire pour explorer l'anneau par cette collection d'agents.*

Démonstration. Par l'Observation 5.1, les agents d'une solution optimale ont des segments d'intérieurs disjoints. En coupant l'anneau en un point de rencontre de deux segments d'opérations consécutifs, on obtient l'exploration d'un segment. Par le Lemme 5.5, le temps d'exploration de ce segment peut être minimisé dans une exploration où un des agents (supposons l'agent i) explore un segment de l'anneau débutant à sa position d'origine p_i. Pour trouver cette exploration, il est suffisant de prendre en considération $2n$ configurations où l'anneau A est coupé dans une des n positions d'origine des agents, et que l'agent à cette position est placé sur une des deux extrémités du segment résultant. Par le Théorème 5.1.1, il nous suffit de répéter $2n$ fois l'algorithme (présenté par Stec, s'exécutant en temps $O(n^2)$) pour trouver une exploration avec un temps d'exploration minimal. Un tel algorithme s'exécute alors en temps $O(n^3)$. □

5.3 Algorithme d'exploration en temps $O(n^2)$

Dans cette section nous montrerons qu'il est possible de construire un algorithme plus efficace que celui résultant du théorème 5.2.1.

5.3.1 Présentation intuitive de l'algorithme

Pour illustrer de manière intuitive notre algorithme, considérons un processus durant lequel chaque agent obtient la même quantité de temps pour son travail. Cette quantité de temps croit de façon continue. Notre algorithme calcule le comportement de chaque agent pour chaque valeur de temps $t > 0$, jusqu'au moment où le temps t atteint la valeur t^* permettant l'exploration de l'anneau entier par l'ensemble des agents.

On suppose que pour chaque valeur de t, les agents sont partagés en séquences (maximales) d'agents consécutifs, chaque séquence explorant un segment continu de l'anneau. Pour chacune de ces séquences d'agents $< i, j >$, les agents explorent en temps t un segment de l'anneau se terminant au point $p_j + t$ (c'est-à-dire que l'agent j se déplace uniquement en sens horaire à partir de sa position initiale). En plus, le segment $[x, p_j + t]$ exploré par les agents $< i, j >$ est de longueur maximale.

Soit $S_i(t) = [g_i(t), d_i(t)]$ le segment de l'anneau A exploré par l'agent i si tous les agents disposent d'un temps t pour l'exploration. Nous décrirons comment chaque segment $S_i(t)$ change selon notre algorithme au fur et à mesure que la valeur de t augmente. L'observation 5.1 nous permet de supposer que $p_{i-1} \leq g_i(t) \leq p_i \leq d_i(t) \leq p_{i+1}$. Pour de petites valeurs de t, chaque segment $S_i(t)$ débute au point p_i et continue le plus loin possible en sens horaire jusqu'à la position $d_i(t)$, c'est-à-dire, $S_i(t) = [p_i, p_i + t]$. Lorsque la valeur de $d_i(t)$ atteint p_{i+1} (la position de départ du prochain agent $i+1$) les segments $S_i(t)$ et $S_{i+1}(t)$ se fusionnent. À partir de ce moment, et si $S_{i+1}(t)$ et $S_{i+2}(t)$ ne se sont pas encore fusionnés, la valeur de $d_{i+1}(t)$ continue d'augmenter à la vitesse maximale, de sorte que $d_{i+1}(t) = p_{i+1} + t$, mais la valeur de $g_i(t)$ commence à décroitre (le temps supplémentaire alloué à l'agent i est utilisé par ce dernier pour couvrir la portion de l'anneau situé avant le point p_i).

Chaque segment $S_i(t) = [g_i(t), d_i(t)]$ change en fonction du temps t. Au départ, $g_i(t)$ demeure constant, alors que $d_i(t)$ augmente en sens horaire. Ceci correspond à l'agent i se déplaçant en sens horaire uniquement, à partir de sa position initiale. Nous disons alors que l'agent i est dans l'état 0. À un moment t_1, le segment $S_i(t_1)$ fusionne avec le segment $S_{i+1}(t_1)$ (c'est-à-dire $d_i(t_1) = g_{i+1}(t_1)$) et l'agent i passe en état 1. Dans l'état 1,

l'agent explore l'anneau A d'abord en sens antihoraire de p_i à $g_i(t)$, puis en sens horaire de $g_i(t)$ à $d_i(t) = g_{i+1}(t)$.

Chaque valeur de t pendant laquelle l'agent i est dans l'état 1 est caractérisé par $g_i(t) - p_i \leq d_i(t) - p_i$, de sorte que la stratégie optimale pour couvrir le segment $S_i(t)$ consiste à explorer d'abord l'extrémité gauche, qui est la plus rapprochée. Pendant que l'agent i est dans l'état 1, la valeur de $g_i(t)$ diminue en fonction de t et la valeur de $d_i(t)$ peut elle aussi diminuer, ou rester constante ($d_i(t)$ peut rester constante pour un certain temps, puis commencer à diminuer). Durant ce processus, à un point dans le temps t_2, il se peut que nous nous retrouvions dans une situation telle que $g_i(t_2) - p_i = d_i(t_2) - p_i$. À ce moment, l'agent i passe à l'état 2, dans lequel l'exploration optimale du segment $S_i(t)$ consiste à explorer d'abord en sens horaire de p_i à $d_i(t)$, puis en sens antihoraire de $d_i(t)$ à $g_i(t)$.

Lorsque l'agent i est dans l'état 2, la valeur de $g_i(t)$ continue de diminuer en fonction de t et la valeur de $d_i(t)$ demeure constante ou diminue elle aussi. Éventuellement, à un point dans le temps t_3, il se peut que $d_i(t_3) = p_i$, à partir duquel le déplacement de l'agent i se fera uniquement en sens antihoraire, de p_i à $g_i(t)$.

Nous appelons les valeurs de temps t_0, t_1, t_2, t_3 des *temps de changement d'état*. Notons que, alors que chaque agent change d'état de la manière décrite plus haut, certains agents demeureront dans l'état 0, 1 ou 2 au temps t^* où l'anneau sera exploré dans sa totalité (c'est-à-dire la plus petite valeur de t^* pour laquelle $d_i(t^*) = g_{i+1}(t^*)$, pour tous $i = 0, 1, \ldots, k-1$). Il est aussi important de noter que, à un temps t^*, il se peut que $S_i(t) = [p_{i-1}, d_i]$ après quoi la valeur de $g_i(t)$ demeurera constante à p_{i-1} et que l'agent i ne puisse plus utiliser de temps additionnel. Observons qu'une telle exploration est toujours *directe*.

Lorsque le temps t augmente, les segments explorés par les agents changent et se fusionnent de sorte qu'à chaque moment t, l'ensemble des agents est divisé en sous-ensembles d'agents consécutifs qui explorent un segment (continue) de l'anneau.

Par conséquent, pour chaque agent i et temps t, il est possible de définir un intervalle $[G_i(t), D_i(t)]$ qui est le segment de l'anneau A exploré en temps t par une séquence maximale contenant l'agent i. Les valeurs de $G_i(t)$ et $D_i(t)$ sont déterminées par les

formules suivantes :

(1) $D_i(t) =$ $\max_j : [p_i, p_j + t]$ est explorable par la séquence $< i, j >$
(2) $G_i(t) =$ $x : [x, D_i(t)]$ est le plus grand segment de l'anneau explorable par une séquence $< i', j >$ tel que $i' \leq i \leq j$

Le Lemme 5.2 nous conduit à l'observation suivante :

Observation 5.3. *Les fonctions $d_i(t), g_i(t)$ pour tous $i = 0, 1, \ldots, k-1$ sont continues et, par morceaux, linéaires. Les points de non-linéarité de ces fonctions sont les temps de changement d'état. Il existe un total de $O(n)$ des temps de changement d'état.*

L'idée principale derrière notre algorithme permettant de déterminer la stratégie optimale pour explorer l'anneau consiste à calculer progressivement chaque temps de changement d'état. Puisque les fonctions $g_i(t)$ et $d_i(t)$ (pour $i = 0, 1, \ldots, k-1$) sont linéaires entre deux temps de changement d'état, nous pouvons les décrire ainsi :

$$g_i(t) = g_{it}t + g'_{it}; \text{ et } d_i(t) = d_{it}t + d'_{it};$$

Les valeurs de $g_{it}, g'_{it}, d_{it}, d'_{it}$ demeurent constante dans chaque intervalle de temps, et peuvent changer lors des temps de changement d'état.

À chaque moment de temps t, il existe des séquences de segments consécutifs $S_i(t), S_{i+1}(t), \ldots, S_j(t)$ ayant fusionné, et des paires de segments consécutifs $S_j(t)$ et $S_{j+1}(t)$ entre lesquels il y a des *trous* (c'est-à-dire $d_j(t) \neq g_{j+1}(t)$). Ceci nous amène à faire l'observation suivante :

Observation 5.4. *Soit i et j une paire d'agents telle que $S_i(t), S_{i+1}(t), \ldots, S_j(t)$ est une séquence maximale de segments consécutifs fusionnés au temps t (c'est-à-dire que nous avons $d_k(t) = g_{k+1}(t)$ pour $k = i, i+1, \ldots, j-1$, mais $d_{k-1}(t) \neq g_k(t)$ et $d_j(t) \neq g_{j+1}(t)$). Alors $d_j(t)$ est le point le plus éloigné en sens horaire explorable par les agents $i, i+1, \ldots, j$, et $[g_i(t), d_j(t)]$ est le plus grand segment de l'anneau A se terminant au point $d_j(t)$ explorable par ces agents. De plus, $[g_i(t), d_j(t)] = [G_k(t), D_k(t)]$ pour chaque $i \leq k \leq j$.*

Notre algorithme réalise le processus présenté ci-dessus dans le sens horaire, en calculant la valeur finale de la variable T^+ qui représente la disposition du dernier *trou* entre les segments $g_j(t)$ et $d_{j+1}(t)$ pour $j = 0, \ldots, n-1$.

Ensuite, l'algorithme réalise le processus symétrique dans le sens antihoraire, calculant le temps T^-. Finalement, l'algorithme retourne $min(T^+, T^-)$.

Nous donnerons maintenant le pseudo-code de haut niveau représentant notre algorithme.

Algorithme ExploreRing($p_0, p_1, \ldots, p_{n-1}$)
1. $T^+ = 0$;
2. **for each** $i = 0$ **to** $n - 1$ **do**
3. $\quad g_i(t) = p_i$; $d_i(t) = p_i + t$;
4. **while** il y a un agent dans l'état 0 **do**
5. $\quad T^+ =$ la plus petite valeur de temps t à laquelle un agent change d'état;
6. $\quad$ **for each** $i = 0$ **to** $n - 1$ **do**
7. $\quad\quad$ Recalculer $g_i(t)$ et $d_i(t)$ pour l'intervalle débutant à T^+;
8. Répéter les étapes 1-7 en direction antihoraire afin de déterminer T^-;
9. $T_{ER} = \min(T^+, T^-)$;
10. **return** T_{ER};

Observons que les valeurs $g_i(T_{ER})$ et $d_i(T_{ER})$ pour $i = 0, \ldots, n-1$ obtenues dans l'algorithme ExploreRing permettent directement de générer les mouvements exploratoires pour chaque agent.

Théorème 5.3.1. *L'algorithme ExploreRing retourne le temps optimal dans lequel l'anneau A peut être exploré par un ensemble d'agents donné. L'algorithme possède une complexité de* $O(n^2)$.

Démonstration. Selon le Lemme 5.5, il existe un point p_i (la position initiale de l'agent i) où il est possible de *couper* l'anneau et exécuter l'algorithme d'exploration d'un segment de droite avec l'agent i placé au début du segment, de sorte que le temps T_{OPT} retourné par un tel algorithme est optimal. Nous nommerons cet algorithme A_{OPT}. Supposons, par symétrie, que l'exécution de cet algorithme correspond à l'agent i se déplaçant en sens horaire sur l'anneau. Considérons la portion de l'algorithme ExploreRing explorant l'anneau en sens horaire.

L'algorithme se termine lorsque le temps T_{ER} de l'exploration atteint un temps où le dernier trou entre les groupes d'agents se referme. Supposons que ce dernier trou est le trou entre les segments d'opération des agents j et $j + 1$. Au temps T_{ER}, l'agent j, qui se déplace uniquement dans une direction, atteint le point de l'anneau qui est la limite du segment d'opération de l'agent $j + 1$.

Supposons, par contradiction, que $T_{ER} > T_{OPT}$.

Observons d'abord que dans la solution en temps T_{ER}, générée par l'algorithme ExploreRing, l'agent i ne peut pas aller uniquement dans la direction horaire, c'est-à-dire $g_i(T_{ER}) < p_i$. Effectivement, comme l'agent i était dans l'état 0 juste avant le temps T_{ER}, il était dans l'état 0 au temps T_{OPT}. Par l'Observation 5.4 le segment $[G_i(T_{OPT}), D_i(T_{OPT})]$ est le plus long segment explorable par les agents en temps T_{OPT} se terminant au point $D_i(T_{OPT}) = p_i + T_{OPT}$. Comme l'algorithme ExploreRing ne s'est pas terminé en temps T_{OPT} alors le segment $[G_i(T_{OPT}), D_i(T_{OPT})]$ ne couvre pas la totalité de l'anneau. Ceci est une contradiction puisque A_{OPT} explore en temps T_{OPT} le segment continu se terminant en $p_i + T_{OPT}$ qui couvre l'anneau au complet.

L'observation ci-dessus nous permet de conclure que $j \neq i$. S'il y a plusieurs agents qui se déplacent uniquement dans la direction horaire dans la solution produite par l'algorithme ExploreRing, nous dénotons par j le dernier tel agent précédant l'agent i. Considérons les mouvements de l'ensemble C d'agents consécutifs $C = \{j, j+1, \ldots, i-1\}$. Observons que si l'un des agents $j+1, j+2 \ldots, i-1$ est forcé d'aller uniquement dans la direction antihoraire (disons l'agent k) alors le segment $[p_{j+1}, p_{k+1}]$ n'est pas explorable en temps plus court que T_{ER}. Conséquemment, tous les agents $j+1, j+2 \ldots, i-1$ sont dans l'état 1 ou 2 (ils font volte-face durant leur parcours respectif.) Dénotons $y = g_i(T_{ER})$ (rappelons que $y < p_i$). Par Lemme 5.3, T_{ER} est le temps optimal requis pour explorer le segment $S' = [p_j, y]$.

Considérons maintenant le segment S'' exploré par l'algorithme A_{OPT} par le même groupe d'agents $C = j, j+1, \ldots, i-1$ en temps T_{OPT}. Observons que, alors que l'agent i se déplace seulement en sens horaire selon A_{OPT}, $S'' = [z, p_i]$ où le point z précède le point p_j en sens horaire sur l'anneau. Donc $S' \subset S''$. Il s'agit d'une contradiction puisque le même ensemble d'agents C explore S' en un temps plus court que T_{OPT}, contrairement au Lemme 5.3. Ceci complète la preuve de l'exactitude de l'algorithme ExploreRing.

La complexité de l'algorithme ExploreRing est déterminée par la complexité totale de la ligne 7. Comme chaque agent peut changer son état trois fois, la boucle *while* de la ligne 4 est exécutée $O(n)$ fois. Une fois que nous avons déterminé le prochain changement d'état d'un agent à la ligne 5 (disons l'agent i^*), les coefficients des fonctions linéaires $g_{i*}(t)$ et $d_{i^*}(t)$ peuvent être recalculées en temps constant à la ligne 7. De la même façon, on recalcule en temps constant les coefficients $g_j(t)$ et $d_j(t)$ pour chaque agent j tel que $p_j \in [G_i(T^+), D_i(T^+)]$. Comme la boucle *for* de la ligne 6 s'exécute $O(n)$ fois, la complexité totale de la ligne 7 est de $O(n^2)$. Ceci complète la preuve. □

5.3.2 Pseudo-code

Dans cette section, nous donnerons les pseudo-codes un peu plus détaillés réalisant l'algorithme ExploreRing. L'implantation de ces algorithmes dans le langage de programmation Java se trouve en Annexe A.

5.3.2.1 Vue d'ensemble de l'algorithme

La fonction *ExploUnidirectionnel* permet de trouver le programme de déplacement optimal des agents, en supposant que l'agent q (qui n'effectue un déplacement que dans une seule direction) exécute son déplacement dans la direction donnée (horaire ou antihoraire). L'algorithme consiste à évaluer les vitesses de déplacement des frontières des segments d'opération de chaque agent à chaque point de non-linéarité, et d'établir quel sera le prochain point de non-linéarité. La boucle est répétée tant qu'au moins un agent opère selon la stratégie 0. Au moment où le dernier agent transitionne depuis la stratégie 0, le dernier « trou » est bouché et la fonction est en possession de la solution.

Object function ExploUnidirectionnel(**Object** ring, **direction** dir)
1. **real** $t = 0$, **integer** $numberOfAgentsInStateZero = ring \rightarrow k$, **Object** *solution*
2. **while** $numberOfAgentsInStateZero > 0$ **then**
3. computeStartAndEndBoundaryPositionAndSpeed($ring, solution, t$)
4. determineNextAgentStateChange($ring, solution, t$)
5. **if** $stateOfAgentChangingState = 0$ **then**
6. $numberOfAgentsInStateZero - -$
7. $t = timeOfNextAgentStateChange$
8. **return** solution

La fonction *Explo* procède à l'exploration de l'anneau en supposant que l'agent q effectue son déplacement d'abord en sens horaire, puis en sens antihoraire. La fonction compare alors le résultat afin de déterminer quelle est la solution optimale.

Object function Explo(**Object** ring)
1. **Object** $S_{clockwise}$ = ExploUnidirectionnel($ring, clockwise$)
2. **Object** $S_{counterclockwise}$ = ExploUnidirectionnel($ring, counterclockwise$)
3. **if** $S_{clockwise} \rightarrow time < S_{counterclockwise} \rightarrow time$ **then**
4. **return** $S_{clockwise}$

5. else
6. **return** $S_{counterclockwise}$

5.3.2.2 Algorithme détaillé

Les fonctions suivantes sont utilisées afin d'implanter l'algorithme. Pour des raisons de simplicité, l'algorithme ne s'attarde qu'à la recherche lorsque l'agent q se déplace, dans la solution optimale, dans le sens horaire. Plutôt que de rendre l'algorithme bidirectionnel, on fera plutôt une réflexion de la configuration des agents sur l'anneau, après quoi le même algorithme sera employé.

La fonction récursive *StartBoundarySpeed* détermine la vitesse de la frontière gauche g_i de l'agent i.

real function StartBoundarySpeed(**integer** i);
1. **if** $state_i = 0$ **then**
2. **return** 0
3. **else if** $state_i = 1$ **then**
4. **return** 0.5+ StartBoundarySpeed$(i+1)$ /2
5. **else if** $state_i = 2$ **then**
6. **return** 1+ StartBoundarySpeed$(i+1)$ $*2$
7. **else if** $state_i = 3$ **then**
8. **return** 0

La fonction *EndBoundarySpeed* détermine la vitesse de la frontière droite d_i de l'agent i. Notons que cette fonction fait appel à la fonction récursive *StartBoundarySpeed*.

real function EndBoundarySpeed(**integer** i);
1. **if** $state_i = 0$ **then**
2. **return** 1
3. **else**
4. **return** StartBoundarySpeed$(i+1)$

La fonction *StateChange* permet de déterminer le prochain temps de non-linéarité de la fonction d'exploration de l'agent i suivant le temps t, c'est-à-dire la valeur de temps à laquelle l'agent passera d'une stratégie à la suivante.

[real, integer] function StateChange(**real** t, **integer** i) ;

1. $state := 0$
2. **if** $e_i(t) < s_{i+1}(t)$ **then**
3. $t := \min_x : e_i(x) = p_{i+1}$
4. $state := 0$
5. **else if** $|p_i - s_i(t)| < |e_i(t) - p_i|$ **then**
6. $t := \min_x : |p_i - s_i(x)| = |e_i(x) - p_i|$
7. $state := 1$
8. **else if** $p_i < s_{i+1}(t)$ **then**
9. $t := \min_x : p_i = s_{i+1}(x)$
10. $state := 3$
11. **else**
12. $t := \infty$
13. $state := 2$
14. **return** $[t, state]$

La fonction *DetermineNextStateChange* détermine, parmi tous les agents sur l'anneau, lequel sera le prochain à faire un changement de stratégie, de même que le temps auquel ce changement de stratégie aura lieu.

[integer, integer, real] function DetermineNextStateChange(**real** *periodBeginning*) ;

1. $nextAgent := 0, nextState := 0, nextChange := \infty$
2. **for** $i := 0$, i++, **while** $i < n$
3. $[t, state] :=$ StateChange($periodBeginning, i$)
4. **if** $t < nextChange$ **then**
5. $nextAgent := i$
6. $nextState := state$
7. $nextChange := t$
8. **return** $[nextAgent, nextState, nextChange]$

La fonction *ApplyStateChange* met à jour la position et la vitesse des frontières gauches et droites de chaque agent lors d'un point de non-linéarité. À priori, il pourrait sembler que cette fonction possède une complexité de $O(n^2)$, puisque les fonctions $StartBoundarySpeed(i)$ et $EndBoundarySpeed(i)$ sont récursives. Par contre, puisque les résultats de $StartBoundarySpeed(i)$ et $EndBoundarySpeed(i)$ seront toujours les

mêmes à un point de non-linéarité donné, il n'est pas nécessaire de recalculer chacune de ces vitesses k fois. La fonction *ApplyStateChange* possède donc une complexité de $O(n)$.

boolean function ApplyStateChange(**integer** *changingAgent*, **real** t) ;
1. **for** $i := 0$, i++, **while** $i < n$
2. $s_i(t) \leftarrow$ StartBoundarySpeed(i)
3. $e_i(t) \leftarrow$ EndBoundarySpeed(i)

La fonction *ExploComplete* détermine si l'exploration de l'anneau est complétée.

boolean function ExploComplete() ;
1. **return** $numberOfAgentsInStateZero == 0$

L'algorithme *OptExpTime* détermine la solution optimale pour une configuration d'agent $0, 1, \ldots, k$, lorsque l'agent q effectue son déplacement dans le sens horaire. La boucle principale (lignes 1 à 5) est effectuée au plus $3k$ fois.

Algorithm OptExpTime($p_0, p_1, \ldots, p_{k-1}$) ;
1. $i := 0, state := 0, t_i := 0$;
2. **while not** ExploComplete()
3. [i,state,t] := DetermineNextStateChange()
4. $state_i = state$
5. ApplyStateChange(i, t)

Chapitre 6

Implantation de l'algorithme

Nous avons implanté l'algorithme d'exploration ainsi qu'un logiciel de visualisation de son exécution pour les fins pédagogiques. Cependant, au début de mon projet de maîtrise, la preuve de correction de cet algorithme n'était pas claire. Il fallait alors confirmer cette hypothèse par un ensemble d'expérimentations. En conséquence, nous avons programmé un ensemble de simulations pour mettre en compétition notre algorithme d'exploration avec un algorithme glouton.

Le langage de programmation Java a été sélectionné pour sa popularité et la familiarité de l'auteur vis-à-vis ce dernier. La librairie de tests unitaires *JUnit* a été utilisée. Le logiciel de contrôle de version *Git* nous a permis de conserver l'historique d'évolution de l'algorithme.

6.1 Compétition

Dès le départ, il a été pratique de mettre en compétition l'algorithme *glouton* décrit au chapitre 5 et l'algorithme optimal proposé. L'algorithme glouton implanté, inspiré des travaux de Stec [21], exécute une recherche binaire de manière à obtenir un résultat approximatif. Dans leurs versions finales, les deux algorithmes ont donné des résultats équivalents pour 50 000 configurations d'agents générées aléatoirement (durée approximative du test : 10 minutes).

6.2 Difficultés

L'emploi de nombres irrationnels tels que π introduit des erreurs d'approximations lors des calculs, même en utilisant des nombres réels exprimés sur 64 bits (type *double* en Java). Il a donc été nécessaire d'introduire un facteur de précision ϵ aux assertions exprimées dans les tests unitaires. Si l'emploi de la classe *java.math.BigDecimal* nous aurait permis de minimiser ϵ, il n'a pas été jugé nécessaire dans le cadre de ces travaux.

6.3 Logiciel de visualisation

La Figure 6.1 montre le logiciel de visualisation de la solution développé en parallèle. Il s'agit de la solution pour un problème comportant 6 agents répartis sur l'anneau. Les barres verticales représentent les points de non-linéarité décrits au chapitre 5.

Un curseur permet de visualiser la progression de l'algorithme.

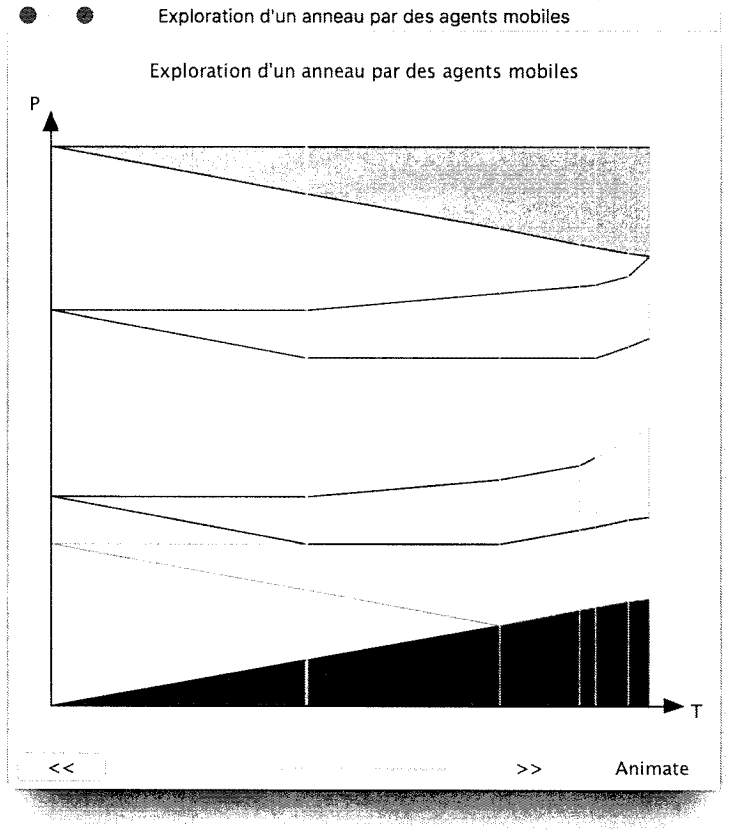

FIGURE 6.1 – Interface de visualisation de la solution

Il est possible de visualiser l'animation et de paramétrer le problème à résoudre grâce à l'*Applet Java* disponible en ligne à l'adresse suivante : http://www.alexandrelemieux.ca/explo/

Chapitre 7

Conclusion

Nous avons proposé une solution au problème d'exploration d'un anneau par un ensemble d'agents. Les agents sont placés dans des positions initiales données au départ. Notre algorithme trouve le temps minimal nécessaire pour l'exploration. Le Lemme 5.5 nous a permis d'obtenir un algorithme fonctionnant en temps $O(n^3)$ en utilisant une solution proposée par Stec [21] pour l'exploration d'un segment (voir théorème 5.1.1). Cependant, nous avons proposé un algorithme indépendant qui fonctionne en temps $O(n^2)$.

Notre algorithme a été implanté en Java. La solution affichée par notre programme permet de visualiser les concepts développés dans notre travail, ce qui constitue ses valeurs pédagogiques. L'animation de notre algorithme a été mise en ligne.

Il reste quelques questions ouvertes. Il semble que notre solution peut être étendue pour les agents ayant une vitesse d'exploration différente de la vitesse de parcours sans exploration (voir le problème de *Beachcombers* [8]). Un problème plus difficile est de considérer des agents ayant des vitesses distinctes. Un autre problème ouvert est de prouver la borne inférieure pour l'algorithme d'exploration de l'anneau dans le modèle étudié dans notre mémoire.

Bibliographie

[1] Anaya, J., Chalopin, J., Czyzowicz, J., Labourel, A., Pelc, A., and Vaxès, Y. Convergecast and broadcast by power-aware mobile agents. *Algorithmica 74*, 1 (2016), 117–155.

[2] Arkin, E. M., Hassin, R., and Levin, A. Approximations for minimum and min-max vehicle routing problems. *Journal of Algorithms 59*, 1 (2006), 1–18.

[3] Baeza-Yates, R., and Schott, R. Parallel searching in the plane. *Computational Geometry 5*, 3 (1995), 143–154.

[4] Bender, M. A., Fernández, A., Ron, D., Sahai, A., and Vadhan, S. The power of a pebble : Exploring and mapping directed graphs. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (1998), ACM, pp. 269–278.

[5] Bonato, A., Chiniforooshan, E., and Prałat, P. Cops and robbers from a distance. *Theoretical Computer Science 411*, 43 (2010), 3834–3844.

[6] Chalopin, J., Jacob, R., Mihalák, M., and Widmayer, P. Data delivery by energy-constrained mobile agents on a line. In *International Colloquium on Automata, Languages, and Programming* (2014), Springer, pp. 423–434.

[7] Chrobak, M., Gąsieniec, L., Gorry, T., and Martin, R. Group search on the line. In *International Conference on Current Trends in Theory and Practice of Informatics* (2015), Springer, pp. 164–176.

[8] Czyzowicz, J., Gasieniec, L., Georgiou, K., Kranakis, E., and MacQuarrie, F. The beachcombers' problem : walking and searching with mobile robots. *Theoretical Computer Science 608* (2015), 201–218.

[9] Czyzowicz, J., Gasieniec, L., Kosowski, A., and Kranakis, E. Boundary patrolling by mobile agents with distinct maximal speeds. In *European Symposium on Algorithms* (2011), Springer, pp. 701–712.

[10] CZYZOWICZ, J., ILCINKAS, D., LABOUREL, A., AND PELC, A. Worst-case optimal exploration of terrains with obstacles. *Inf. Comput. 225* (2013), 16–28.

[11] CZYZOWICZ, J., KOSOWSKI, A., AND PELC, A. How to meet when you forget : log-space rendezvous in arbitrary graphs. *Distributed Computing 25*, 2 (2012), 165–178.

[12] CZYZOWICZ, J., PELC, A., AND ROY, M. Tree exploration by a swarm of mobile agents. In *OPODIS* (2012), Springer, pp. 121–134.

[13] DOBREV, S., FLOCCHINI, P., PRENCIPE, G., AND SANTORO, N. Searching for a black hole in arbitrary networks : Optimal mobile agents protocols. *Distributed Computing 19*, 1 (2006).

[14] FLOCCHINI, P., ILCINKAS, D., PELC, A., AND SANTORO, N. How many oblivious robots can explore a line. *Information processing letters 111*, 20 (2011), 1027–1031.

[15] FRAIGNIAUD, P., AND PELC, A. Delays induce an exponential memory gap for rendezvous in trees. *ACM Transactions on Algorithms (TALG) 9*, 2 (2013), 17.

[16] GORRY, T. *Navigation Problems for Autonomous Robots in Distributed Environments.* PhD thesis, The University of Liverpool, 2015.

[17] LESSARD, F. Exploration optimale d'un segment de droite par deux agents mobiles. Master's thesis, Université du Québec en Outaouais, 2013.

[18] PELC, A. Deterministic rendezvous in networks : A comprehensive survey. *Networks 59*, 3 (2012), 331–347.

[19] RUSSELL, S., AND NORVIG, P. *Artificial Intelligence : A Modern Approach, 2nd edition.* Prentice-Hall, 2003.

[20] SKIENA, S. S. *The algorithm design manual, 2nd edition.* Springer Science & Business Media, 2010.

[21] STEC, A. L'exploration optimale d'un segment par un ensemble de robots mobiles. Master's thesis, Université du Québec en Outaouais, 2015.

Annexe A

Code source

A.1 Agent.java

```
package net.fortrel.circle2;

public class Agent {
    private String name;
    private double position;
    private double origin;
    private Circle circle;

    public Agent() {
    }

    public Agent(String name) {
        this.name = name;
    }

    public double getPosition() {
        return position;
    }

    public void setPosition(double position) {
        this.position = position;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Circle getCircle() {
        return circle;
    }

    public void setCircle(Circle circle) {
        this.circle = circle;
    }

    public int getIndex() {
        return circle.indexOf(this);
    }

    public Agent getNext() {
        int index = getIndex() + 1;
        index = index % circle.getAgents().size();
        return circle.getAgent(index);
    }

    public Agent getPrevious() {
        int index = getIndex() - 1;
        while (index < 0) {
            index += circle.getAgents().size();
        }
        return circle.getAgent(index);
```

```
    }

    public double getOrigin() {
        return origin;
    }

    public void setOrigin(double origin) {
        this.origin = origin;
    }

    public boolean equals(Object o) {
        if (o.getClass().equals(Agent.class) == false) {
            return false;
        }

        Agent other = (Agent)o;
        if (getName().equals(other.getName()) == false) {
            return false;
        }
        if (Math.abs(getOrigin()-other.getOrigin()) > State.MAX_BOUND_DIFF) {
            return false;
        }
        return true;
    }
}
```

A.2 AgentComparator.java

```
package net.fortrel.circle2;

import java.util.Comparator;

public class AgentComparator implements Comparator<Agent> {

    public int compare(Agent a, Agent b) {
        double diff = a.getPosition() - b.getPosition();
        if (diff > 0) {
            return 1;
        }
        else if (diff < 0) {
            return -1;
        }
        else {
            return 0;
        }
    }

}
```

A.3 BinarySearchAdapter.java

```
package net.fortrel.circle2;

import net.fortrel.robots.RingSegmentSearch;
import net.fortrel.robots.Robot;
import net.fortrel.robots.Solution;

public class BinarySearchAdapter {
    public static double solve(Circle c) {
        RingSegmentSearch segment = new RingSegmentSearch();

        segment.setLength(2*Math.PI);
        for (Agent anAgent : c.getAgents()) {
            Robot r = new Robot();
            r.setPosition(anAgent.getOrigin());
            r.setSearchSpeed(1);
            r.setWalkSpeed(1);
            segment.addRobot(r);
        }

        Solution solu = segment.solve();
        return solu.getTimeRequired();
    }
}
```

A.4 Boundary.java

```
package net.fortrel.circle2;

public class Boundary {
    private double position;
    private double timeOfLastKnownPosition;

    private double speed;
    private boolean reached = false;

    private Boundary neighbor;

    private static void log(String message) {
        //System.out.println(message);
    }

    public Boundary(double position, double speed, boolean reached) {
        this.position = position;
        this.speed = speed;
        this.reached = reached;
    }

    public Boundary() {
    }

    /**
     * Extrapolate the boundary position at a given time based on the
     * last know location and the current time.
     */
    public double getPosition(double currentTime) {
        double timeDifference = currentTime - getTimeOfLastKnownPosition();
        if (timeDifference != 0) {
            if (getSpeed() == 0) {
                // Ok we don't move, there is no prediction to be made.
            }
            else {
                return Circle.toPositivePosition(position + (timeDifference * getSpeed()));
            }
        }
        return position;
    }

    /**
     * Set the "Last know position". We can derive the "current position" from there
     * according to the speed and current time.
     */
    public void setPosition(double lastKnownPosition, double time) {
        this.position = Circle.toPositivePosition(lastKnownPosition);
        this.timeOfLastKnownPosition = time;
    }

    public void updatePosition(
            double newCurrentTime) {
        double timeAdded = newCurrentTime - timeOfLastKnownPosition;
        double newPosition = Circle.toPositivePosition(position + (timeAdded * getSpeed()));

        double predictedPosition = getPosition(newCurrentTime);
        if (Circle.areApproximatelyTheSame(newPosition, predictedPosition) == false) {
            throw new IllegalStateException("Prediction doesn't match.");
        }

        position = newPosition;
        timeOfLastKnownPosition = newCurrentTime;
    }

    public double getSpeed() {
        return speed;
    }

    public void setSpeed(double speed, double currentTime) {
        if (this.speed == speed) {
            // Speed didn't change. No need to adjust anything.
            return;
        }

        if (timeOfLastKnownPosition != currentTime) {
            if (timeOfLastKnownPosition > currentTime) {
                throw new IllegalStateException("Change cannot occur prior to the time of last know position.");
            }

            double timeSpent = currentTime - timeOfLastKnownPosition;
            double distanceCoveredWhenSpeedChanges = timeSpent * this.speed;
            log("Previous speed was: " + this.speed);
            log("Distance covered when the speed changes: " + distanceCoveredWhenSpeedChanges);

            this.position = Circle.toPositivePosition(this.position + (distanceCoveredWhenSpeedChanges));
            this.timeOfLastKnownPosition = currentTime;
        }
```

```
        this.speed = speed;
    }

    public boolean isReached() {
        return reached;
    }

    public void setReached(boolean reached) {
        this.reached = reached;
    }

    public double getTimeOfLastKnownPosition() {
        return timeOfLastKnownPosition;
    }

    public void setTimeOfLastKnownPosition(double timeOfLastKnownPosition) {
        this.timeOfLastKnownPosition = timeOfLastKnownPosition;
    }

    @Override
    public String toString() {
        return "Boundary: Last position: " + position + " at " + timeOfLastKnownPosition
                + ", speed: " + getSpeed();
    }

    public Boundary getNeighbor() {
        return neighbor;
    }

    public void setNeighbor(Boundary neighbor) {
        this.neighbor = neighbor;
    }
}
```

A.5 Circle.java

```
package net.fortrel.circle2;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.Writer;
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.List;
import java.util.Properties;
import java.util.Random;

public class Circle {
    private static final Comparator<Agent> AGENT_COMPARATOR = new AgentComparator();

    private List<Agent> agents = new LinkedList<Agent>();

    /**
     * Default constructor.
     */
    public Circle() {
    }

    public Circle(File aFile) throws IOException {
        this(loadPropertiesFile(aFile));
    }

    public Circle(Properties prop) {
        int i=1;
        while(true) {
            String name = prop.getProperty("agent"+i+".name");
            if (name == null) {
                break;
            }
            double origin = Double.parseDouble(prop.getProperty("agent"+i+".origin"));

            Agent anAgent = new Agent();
            anAgent.setName(name);
            anAgent.setPosition(origin);
            anAgent.setOrigin(origin);

            addAgent(anAgent);
```

```
            i++;
        }
    }

    public Circle(Circle circle) {
        this(circle.toProperties());
    }

    private static void log(String str) {
        // System.out.println(str);
    }

    private static Properties loadPropertiesFile(File aFile) throws IOException {
        InputStream input = new FileInputStream(aFile);
        Properties prop = new Properties();
        try {
            prop.load(input);
            return prop;
        }
        finally {
            input.close();
        }
    }

    @Override
    public boolean equals(Object o) {
        if (o.getClass().equals(Circle.class) == false) {
            return false;
        }

        Circle other = (Circle)o;
        if (agents.size() != other.getAgents().size()) {
            return false;
        }
        for (int i=0; i<agents.size(); i++) {
            Agent anAgent = agents.get(i);
            Agent anotherAgent = other.getAgent(i);
            if (anAgent.equals(anotherAgent) == false) {
                return false;
            }
        }
        return true;
    }

    @Override
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Circle(").append(agents.size()).append(")");
        return buffer.toString();
    }

    public void addAgent(Agent toAdd) {
        // Set origin.
        toAdd.setOrigin(toAdd.getPosition());

        toAdd.setCircle(this);
        agents.add(toAdd);
        Collections.sort(agents, AGENT_COMPARATOR);
    }

    public List<Agent> getAgents() {
        return agents;
    }

    public void setAgents(List<Agent> agents) {
        this.agents = agents;
    }

    public Agent getAgent(int index) {
        return agents.get(index);
    }

    public Agent getAgent(String str) {
        for (Agent anAgent : agents) {
            if (anAgent.getName().equals(str)) {
                return anAgent;
            }
        }
        return null;
    }

    public int indexOf(Agent a) {
        for (int i=0; i<agents.size(); i++) {
            if (agents.get(i).equals(a)) {
                return i;
            }
        }
```

```
    return -1;
}

public static double distanceFrom(double from, double to, Direction dir) {
    double distance;
    switch(dir) {
    case Clockwise:
        distance = from - to;
        if (distance < 0) {
            distance += (2*Math.PI);
        }
        return eliminateRoundingErrorsForFullCircleWhenShouldBeZero(distance);

    case CounterClockwise:
        distance = to - from;
        if (distance < 0) {
            distance += (2*Math.PI);
        }
        return eliminateRoundingErrorsForFullCircleWhenShouldBeZero(distance);
    default:
        throw new IllegalStateException();
    }
}

private static double eliminateRoundingErrorsForFullCircleWhenShouldBeZero(double distance) {
    if (((2*Math.PI) - distance) < State.MAX_BOUND_DIFF ) {
        return 0;
    }
    return distance;
}

public static double absoluteDistanceBetween(double a, double b) {
    a = toPositivePosition(a);
    b = toPositivePosition(b);
    double distance;
    if (a >= b) {
        distance = distanceFrom(a, b, Direction.Clockwise);
    }
    else {
        distance = distanceFrom(a, b, Direction.CounterClockwise);
    }
    if (distance > Math.PI) {

    }

    return distance;
}

public static double toPositivePosition(double pos) {
    while(pos < 0) {
        pos += (2*Math.PI);
    }

    while(pos > (2*Math.PI)) {
        pos -= (2*Math.PI);
    }

    return pos;
}

public void mirror() {
    for (Agent anAgent : agents) {
        double pos = toPositivePosition(anAgent.getPosition());
        double newPos = toPositivePosition(-pos);
        anAgent.setOrigin(newPos);
        anAgent.setPosition(newPos);
    }

    Collections.sort(agents, new AgentComparator());
}

public void rotate(double angleRad) {
    for (Agent anAgent : agents) {
        double pos = toPositivePosition(anAgent.getPosition());
        pos += angleRad;
        double newPos = toPositivePosition(pos);
        anAgent.setOrigin(newPos);
        anAgent.setPosition(newPos);
    }

    Collections.sort(agents, new AgentComparator());
}

public static Circle createRandomProblem() {
    Random rand = new Random();
    int numberOfRobots = rand.nextInt(9)+2;
    return createRandomProblem(numberOfRobots);
```

```
    }

    public static Circle createRandomProblem(int numberOfRobots) {
        Random rand = new Random();
        Circle segment = new Circle();

        for (int i=0; i<numberOfRobots; i++) {
            Agent a = new Agent(String.valueOf((char)(i+'A')));
            double position = rand.nextDouble()*(2*Math.PI);

            a.setPosition(position);
            segment.addAgent(a);
        }

        return segment;
    }

    public static class AgentComparator implements Comparator<Agent> {

        public int compare(Agent a, Agent b) {
            double diffAsDouble = Circle.toPositivePosition(a.getPosition())
                    - Circle.toPositivePosition(b.getPosition());

            if (diffAsDouble > 0) {
                return 1;
            }
            else if (diffAsDouble < 0) {
                return -1;
            }
            else {
                return 0;
            }
        }}

    public Properties toProperties() {
        Properties prop = new Properties();
        for (int i=0; i<agents.size(); i++) {
            Agent anAgent = agents.get(i);
            prop.setProperty("agent"+(i+1)+".name", anAgent.getName());
            prop.setProperty("agent"+(i+1)+".origin", Double.toString(anAgent.getOrigin()));
            prop.setProperty("agent"+(i+1)+".origin.deg", Double.toString(toDeg(anAgent.getOrigin())));
        }
        //log(prop);
        return prop;
    }

    public static double toDeg(double rad) {
        return (rad/(2*Math.PI))*360;
    }

    public void save(File destination) throws IOException {
        Properties toSave = toProperties();
        Writer writer = new FileWriter(destination);
        try {
            toSave.store(writer, null);
        }
        finally {
            writer.close();
        }
    }

    public static boolean areApproximatelyTheSame(double valueA,
            double valueB) {
        boolean theSame = Math.abs(valueA - valueB) < State.MAX_BOUND_DIFF;
        if (!theSame) {
            log(valueA + " != " + valueB);

            // Rotate 90 deg and re-test;
            valueA = toPositivePosition(valueA + Math.PI/2);
            valueB = toPositivePosition(valueB + Math.PI/2);

            theSame = Math.abs(valueA - valueB) < State.MAX_BOUND_DIFF;
            if (!theSame) {
                log("After rotating by 90 deg, values are still different: "
                        + valueA + " != " + valueB);
            }

        }
        return theSame;
    }
}
```

A.6 CircleSolver.java

```
package net.fortrel.circle2;

import java.awt.Graphics2D;
import java.awt.geom.Point2D;
import java.awt.image.BufferedImage;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;
import java.io.UnsupportedEncodingException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

import javax.imageio.ImageIO;

import net.fortrel.circle2.states.State1;

public class CircleSolver {
    private Circle circle;
    private double timeElapsed = 0;
    private boolean mirror = false;
    private Boundary negativeBoundaries[];
    private Boundary positiveBoundaries[];
    private State states[];
    private StateChange predictions[];
    private int numberOfAgentsLeftInState1 = 0;
    private boolean boundaryValidation = true;
    private String label;

    // The following paths and triggerrers are not required
    // by the algorithm per say, but by the widget displaying
    // the evolution of the solution graphically.
    private Map<String, List<Point2D.Double>> negativeBoundariesPaths
            = new HashMap<String, List<Point2D.Double>>();
    private Map<String, List<Point2D.Double>> positiveBoundariesPaths
            = new HashMap<String, List<Point2D.Double>>();
    private List<Agent> eventTrigerrers = new LinkedList<Agent>();

    public Circle getCircle() {
        return circle;
    }

    public void init(Circle circle) {
        this.circle = circle;
        this.timeElapsed = 0;
        this.numberOfAgentsLeftInState1 = circle.getAgents().size();

        states = new State[circle.getAgents().size()];
        positiveBoundaries = new Boundary[circle.getAgents().size()];
        negativeBoundaries = new Boundary[circle.getAgents().size()];

        determineInitialBoundaries();
        determineInitialStateChangePredictions();
    }

    public List<Agent> getEventTrigerrers() {
        return eventTrigerrers;
    }

    public void setNegativeBoundary(Agent a, Boundary b) {
        negativeBoundaries[a.getIndex()] = b;
    }

    public void setPositiveBoundary(Agent a, Boundary b) {
        positiveBoundaries[a.getIndex()] = b;
    }

    /**
     * Performance: This is O(n)
     */
    private void determineInitialBoundaries() {
        for (Agent anAgent : circle.getAgents()) {
            Agent previous = anAgent.getPrevious();
            Boundary negativeBoundary = new Boundary();
            negativeBoundary.setPosition(previous.getPosition(), 0);
            setNegativeBoundary(anAgent, negativeBoundary);

            Agent next = anAgent.getNext();
            Boundary positiveBoundary = new Boundary();
            positiveBoundary.setPosition(next.getPosition(), 0);
            setPositiveBoundary(anAgent, positiveBoundary);

            State initialState = new State1(this, anAgent);
            states[anAgent.getIndex()] = initialState;
```

```
        }

        for (Agent anAgent : circle.getAgents()) {
            Agent nextAgent = anAgent.getNext();

            Boundary nextNegative = getNegativeBoundary(nextAgent);
            Boundary currentPositive = getPositiveBoundary(anAgent);

            nextNegative.setNeighbor(currentPositive);
            currentPositive.setNeighbor(nextNegative);
        }

        for (Agent anAgent : circle.getAgents()) {
            State initialState = getState(anAgent);
            initialState.setBoundarySpeeds();
        }
    }

    public double getTimeElapsed() {
        return timeElapsed;
    }

    public void setTimeElapsed(double timeElapsed) {
        this.timeElapsed = timeElapsed;
    }

    public Boundary getNegativeBoundary(Agent anAgent) {
        return negativeBoundaries[anAgent.getIndex()];
    }

    public Boundary getPositiveBoundary(Agent anAgent) {
        return positiveBoundaries[anAgent.getIndex()];
    }

    public State getState(Agent agent) {
        return states[agent.getIndex()];
    }

    public State getState(String agentName) {
        return states[circle.getAgent(agentName).getIndex()];
    }

    private void determineInitialStateChangePredictions() {
        predictions = new StateChange[circle.getAgents().size()];

        Agent anAgent = circle.getAgent(0);
        for (int i=0; i<circle.getAgents().size(); i++) {
            State currentState = getState(anAgent);

            // currentState.getTransitionTime(...) is O(1)
            double nextState = currentState.getTransitionTime();

            predictions[anAgent.getIndex()] = new StateChange(anAgent,
                    getTimeElapsed(), nextState);

            anAgent = anAgent.getNext();
        }
    }

    /**
     * Determine which agent will be the next one to change state
     * as we add more time to solve the problem.
     *
     * Performance: This is O(1)
     *
     * @return  The next state change to occur. This contains the
     *          agent changing state and in how much time this
     *          state change will occur.
     */
    public StateChange getNextStateChange() {
        StateChange nextStateChange = null;

        for (StateChange aStateChange : predictions) {
            if (nextStateChange == null) {
                nextStateChange = aStateChange;
                continue;
            }

            if (aStateChange.getAbsoluteTime() < nextStateChange.getAbsoluteTime()) {
                nextStateChange = aStateChange;
            }
        }
        return nextStateChange;
    }

    /**
     * Performance: O(n)
```

```
 */
public void apply(StateChange stateChange) {
    double nextTimeElapsed = stateChange.getAbsoluteTime();

    State agentState = getState(stateChange.getAgent());
    eventTrigerrers.add(stateChange.getAgent());

    // Completion of the state of the next transition.
    // Every implementation if O(1)
    agentState.complete();

    setTimeElapsed(nextTimeElapsed);

    // isSolved() is O(1)
    if (!isSolved()) {
        State nextState = agentState.nextState();
        if (nextState == null) {
            throw new NullPointerException("Failed to determine next state.");
        }
        states[stateChange.getAgent().getIndex()] = nextState;
    }

    // This is O(n)
    updatePredictionsAfterApplyingFirst();

    // This detects a problem with many random scenarios.
    // Only for debugging and double-verification;
    // this is a no-op when the algorithm runs
    // in normal mode.
    redoBoundaries();
}

public void updateConnectedBoundary(Agent anAgent) {
    Boundary currentPosBound = getPositiveBoundary(anAgent);

    if (currentPosBound.isReached()) {
        Boundary nextNegativeBound = currentPosBound.getNeighbor();
        currentPosBound.setSpeed(nextNegativeBound.getSpeed(), getTimeElapsed());
    }
}

/**
 * Update the change prediction of the agents. The more "connected"
 * the explored segments become, the longer this method takes to
 * execute.
 *
 * Performance: O(n)
 */
private void updatePredictionsAfterApplyingFirst() {
    log("Before:");
    log("=======");
    log(dumpPredictions());
    log("=======");

    // This is O(n)
    StateChange winner = getNextStateChange();
    log("Applying " + winner);
    Agent winningAgent = winner.getAgent();

    // This will loop at most n times.
    // an O(n)
    Agent anAgent = winningAgent;
    for (int i=0; i<circle.getAgents().size(); i++) {
        getState(anAgent).setBoundarySpeeds();

        double nextTransitionTime = getState(anAgent).getTransitionTime();

        StateChange updatedPrediction = new StateChange(
                anAgent, getTimeElapsed(), nextTransitionTime);
        // This is O(1)
        updatePrediction(updatedPrediction);

        // This is O(1)
        updateConnectedBoundary(anAgent.getPrevious());

        Boundary negBoundary = getNegativeBoundary(anAgent);
        if (negBoundary.isReached() == false) {
            log("Negative boundary is not reach.."
                    + "No need to redo the predictions past agent "
                    + anAgent.getName());
            break;
        }

        anAgent = anAgent.getNext();
    }
}
```

```
/**
 * Saves the prediction of the next state change of a given agent.
 *
 * Performance: O(1)
 *
 * @param newPrediction  The state change to save. It contains a reference
 *                       to the agent, and the time of the change.
 */
private void updatePrediction(StateChange newPrediction) {
    log("Updating prediction: " + newPrediction);
    predictions[newPrediction.getAgent().getIndex()] = newPrediction;
}

public String dumpPredictions() {
    return dumpPredictions(predictions);
}

public String dumpPredictions(StateChange changes[]) {
    StringBuffer buffer = new StringBuffer();
    for (StateChange aState : changes) {
        buffer.append(aState.toString(true, 11)).append("\n");
    }
    return buffer.toString().trim();
}

public Agent getAgent(String agentName) {
    return getCircle().getAgent(agentName);
}

public void dumpState() {
    ByteArrayOutputStream bOut = new ByteArrayOutputStream();
    PrintStream stream = new PrintStream(bOut);
    dumpState(stream);
    stream.flush();
    String asString = bOut.toString();
    log(asString);
}

public void dumpState(PrintStream out) {
    out.println("Time Elapsed: " + getTimeElapsed());
    out.println("Number of agents: " + circle.getAgents().size());
    for (Agent anAgent : circle.getAgents()) {
        out.println("- Agent " + anAgent.getName()
                + " (" + anAgent.getIndex() + ") State: "
                + getState(anAgent).getName());
        out.println("   Origin: " + anAgent.getOrigin()
                + " (" + Circle.toDeg(anAgent.getOrigin()) + " deg.)");
        //out.println("   Current position: " + anAgent.getPosition());
        Boundary positive = getPositiveBoundary(anAgent);
        out.println("   Positive boundary: " + positive.getPosition(getTimeElapsed())
                + " (moving at " + positive.getSpeed() + ", reached? "
                + positive.isReached() + ")");
        Boundary negative = getNegativeBoundary(anAgent);
        out.println("   Negative boundary: " + negative.getPosition(getTimeElapsed())
                + " (moving at " + negative.getSpeed() + ", reached? "
                + negative.isReached() + ")");
    }
}

@Override
public String toString() {
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    PrintStream stream = new PrintStream(output);
    dumpState(stream);
    stream.flush();
    try {
        return output.toString("utf8");
    }
    catch (UnsupportedEncodingException e) {
        throw new IllegalStateException(e);
    }
}

/**
 * Performance: O(1)
 * @return
 */
public boolean isSolved() {
    return numberOfAgentsLeftInState1 == 0;
}

protected void appendBoundaryPositions() {
    for (Agent anAgent : circle.getAgents()) {
        Boundary positiveBoundary = getPositiveBoundary(anAgent).getNeighbor();
        Boundary negativeBoundary = getNegativeBoundary(anAgent).getNeighbor();

        Point2D.Double positiveBoundaryPoint
```

```
                = new Point2D.Double(timeElapsed,
                        positiveBoundary.getPosition(timeElapsed));
        Point2D.Double negativeBoundaryPoint
                = new Point2D.Double(timeElapsed,
                        negativeBoundary.getPosition(timeElapsed));

        List<Point2D.Double> positiveBoundaryPath
                = positiveBoundariesPaths.get(anAgent.getName());
        if (positiveBoundaryPath == null) {
            positiveBoundaryPath = new LinkedList<Point2D.Double>();
            positiveBoundariesPaths.put(anAgent.getName(), positiveBoundaryPath);
        }
        List<Point2D.Double> negativeBoundaryPath
                = negativeBoundariesPaths.get(anAgent.getName());
        if (negativeBoundaryPath == null) {
            negativeBoundaryPath = new LinkedList<Point2D.Double>();
            negativeBoundariesPaths.put(anAgent.getName(), negativeBoundaryPath);
        }

        positiveBoundaryPath.add(positiveBoundaryPoint);
        negativeBoundaryPath.add(negativeBoundaryPoint);
    }
}

public List<Point2D.Double> getPositiveBoundaryPath(String agentName) {
    return positiveBoundariesPaths.get(agentName);
}

public List<Point2D.Double> getNegativeBoundaryPath(String agentName) {
    return negativeBoundariesPaths.get(agentName);
}

public CircleSolver solve() {
    return solve(true);
}

/**
 * Performance: O(n^2)
 */
public CircleSolver solve(boolean alsoDoMirror) {
    init(circle);
    int roundCounter = 1;
    appendBoundaryPositions();
    // Each agent can change state 3 times (state 1, 2 and 3) but state
    // 2 and 3 may become "grid-locked" so that amounts for 5 transitions
    // per agent.
    // Each round calls getNextStateChange() and apply(...) which
    // are both O(n).
    // This would mean O(5n)*O(2n) which is O(n^2)
    while(isSolved() == false) {
        // O(n)
        StateChange nextStateChange = getNextStateChange();
        log("Round " + roundCounter + ", Agent " + nextStateChange.getAgent().getName()
                + " is the next agent to change state (at "
                + nextStateChange.getAbsoluteTime() + ")");

        // O(n)
        apply(nextStateChange);
        roundCounter++;
        appendBoundaryPositions();
    }

    // If we have to do a mirror analysis, this will be a one-time
    // recursive call.
    if (alsoDoMirror) {
        Circle mirrorCircle = new Circle(circle);
        mirrorCircle.mirror(); // This calls Collections.sort
        CircleSolver mirrorSolution = new CircleSolver();
        mirrorSolution.init(mirrorCircle);
        mirrorSolution.setMirror(true);
        mirrorSolution.setBoundaryValidation(isBoundaryValidation());
        mirrorSolution.solve(false);

        if (getTimeElapsed() < mirrorSolution.getTimeElapsed()) {
            return this;
        }
        else {
            return mirrorSolution;
        }
    }
    else {
        return this;
    }
}

public void redoBoundaries() {
    if (isBoundaryValidation() == false) {
```

```
            log("Boundary validation disabled.");
            return;
        }

        Agent anAgent = getFirstAgentInState1();
        for (int i=0; i<circle.getAgents().size(); i++) {
            getState(anAgent).assertAccurateBoundaries();
            //log("Agent " + anAgent.getName() + " boundaries are valid.");
            assertValidBoundary(anAgent);

            anAgent = anAgent.getNext();
        }
    }

    /**
     * Performance: This is O(n)
     * @return
     */
    private Agent getFirstAgentInState1() {
        for (Agent anAgent: circle.getAgents()) {
            if (getState(anAgent) instanceof State1) {
                return anAgent;
            }
        }
        throw new IllegalStateException("Cannot find an agent in state 1.");
    }

    private void assertValidBoundary(Agent anAgent) {
        if (getPositiveBoundary(anAgent).isReached()
                && getNegativeBoundary(anAgent.getNext()).isReached()) {
            // Both marked as reached.
            double posBound = getPositiveBoundary(anAgent).getPosition(getTimeElapsed());
            double nextNegBound = getNegativeBoundary(anAgent.getNext()).getPosition(getTimeElapsed());

            if (Circle.absoluteDistanceBetween(posBound, nextNegBound) > State.MAX_BOUND_DIFF) {
                log("Absolute distance: "
                        + Circle.absoluteDistanceBetween(posBound, nextNegBound));
                log("Boundary between agent " + anAgent.getName() + " and "
                        + anAgent.getNext().getName() + ":");
                log(" is " + posBound);
                log(" is " + nextNegBound);
                dumpState();
                throw new IllegalStateException("Boundary between agent " + anAgent.getName()
                        + " and " + anAgent.getNext().getName()
                        + " is inconsistent. (Delta = "
                        + Circle.absoluteDistanceBetween(posBound, nextNegBound) + ")");
            }
        }
        else if (!getPositiveBoundary(anAgent).isReached()
                && !getNegativeBoundary(anAgent.getNext()).isReached()) {
            // Both marked as not reached.
        }
        else {
            throw new IllegalStateException("Boundary between agent "
                    + anAgent.getName() + " and " + anAgent.getNext().getName()
                    + " is not marked as reached on both sides.");
        }
    }

    public Boundary getNegativeBoundary(String agentName) {
        return getNegativeBoundary(getAgent(agentName));
    }

    public Boundary getPositiveBoundary(String agentName) {
        return getPositiveBoundary(getAgent(agentName));
    }

    public double getTransitionTime(String agentName) {
        return getState(agentName).getTransitionTime();
    }

    public boolean isBoundaryValidation() {
        return boundaryValidation;
    }

    public void setBoundaryValidation(boolean boundaryValidation) {
        this.boundaryValidation = boundaryValidation;
    }

    public boolean isMirror() {
        return mirror;
    }

    public void setMirror(boolean b) {
        mirror = b;
    }
```

```
    public void decrementNumberOfAgentsLeftInState1() {
        if (numberOfAgentsLeftInState1 <= 0) {
            throw new IllegalStateException(
                    "Number of agents left in state 1 cannot be decremented below 0.");
        }
        numberOfAgentsLeftInState1--;
    }

    public void toImage(File imageFile, int width, int height) throws IOException {
        SolutionPainter painter = new SolutionPainter(this, width, height);
        BufferedImage bImg = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
        Graphics2D cg = bImg.createGraphics();
        painter.paint(cg);

        if (ImageIO.write(bImg, "png", imageFile) == false) {
            throw new IllegalArgumentException("Failed to write image.");
        }
    }

    public List<Double> getEventTimes() {
        List<Double> times = new ArrayList<Double>();

        for (Point2D.Double aPoint : getNegativeBoundaryPath(getCircle().getAgent(0).getName())) {
            times.add(aPoint.x);
        }

        return times;
    }

    private static void log(String str) {
        // System.out.println(str);
    }

    public String getLabel() {
        return label;
    }

    public void setLabel(String label) {
        this.label = label;
    }
}
```

A.7 Direction.java

```
package net.fortrel.circle2;

public enum Direction {
    Clockwise, CounterClockwise;
}
```

A.8 Element.java

```
package net.fortrel.circle2;

public interface Element {
    public double getPosition(double time);
}
```

A.9 SolutionGraphWidget.java

```
package net.fortrel.circle2;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Window;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Collections;
```

```
import java.util.LinkedList;
import java.util.List;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComponent;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JSeparator;
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import net.fortrel.circle2.test.BoundaryTest;

public class SolutionGraphWidget extends JComponent {
    private static final long serialVersionUID = 632654219160926545 7L;

    private CircleSolver solution;
    private double visiblePortion; // 0 - 1.0

    public static void main(String args[]) {
        JFrame frame = new DemoFrame(getDefaultSolution());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    public static class DemoFrame extends JFrame {
        private static final long serialVersionUID = 4205510572675666878L;

        private final int MAX_SLIDER_VALUE = 1000;
        private SolutionGraphWidget widget;
        private JSlider slider;
        private JButton playButton;
        private JButton previousEventButton;
        private JButton nextEventButton;
        private JLabel resultLabel;

        public DemoFrame(CircleSolver solution) {
            setTitle("Exploration d'un anneau par des agents mobiles");

            widget = new SolutionGraphWidget();
            widget.setSolution(solution);
            getContentPane().add(widget, BorderLayout.CENTER);

            slider = new JSlider(JSlider.HORIZONTAL, 0, MAX_SLIDER_VALUE, MAX_SLIDER_VALUE);
            slider.addChangeListener(new SliderChangeListener());

            previousEventButton = new JButton("<<");
            previousEventButton.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    onPreviousEvent();
                }
            });

            nextEventButton = new JButton(">>");
            nextEventButton.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    onNextEvent();
                }
            });

            JPanel sliderPanel = new JPanel(new BorderLayout());
            sliderPanel.add(previousEventButton, BorderLayout.WEST);
            sliderPanel.add(slider, BorderLayout.CENTER);
            sliderPanel.add(nextEventButton, BorderLayout.EAST);

            playButton = new JButton("Animer");
            playButton.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    setControlWidgetsEnabled(false);

                    AnimateThread thread = new AnimateThread();
                    thread.start();
                }
            });
```

```
        resultLabel = new JLabel();
        updateResultLabel();

        JPanel bottomPane = new JPanel(new BorderLayout());
        bottomPane.add(sliderPanel, BorderLayout.CENTER);
        bottomPane.add(playButton, BorderLayout.EAST);
        bottomPane.add(resultLabel, BorderLayout.NORTH);

        getContentPane().add(bottomPane, BorderLayout.SOUTH);

        addWindowListener(new RepaintOnStartup());

        setJMenuBar(createMenu());
    }

    protected void updateResultLabel() {
        resultLabel.setText(
                "Anneau␣explorÃ©␣en␣temps␣T=" +
                Double.toString(getSolution().getTimeElapsed()) + ".");
    }

    public Circle getProblem() {
        return getSolution().getCircle();
    }

    private JMenuBar createMenu() {
        JMenuBar bar = new JMenuBar();

        JMenu editMenu = new JMenu("Ã‰dition");
        bar.add(editMenu);

        for (int i=0; i<PREDEFINED_PROBLEMS.size(); i++) {
            final int problemIndex = i;
            JMenuItem predefinedProblem = new JMenuItem("ProblÃ¨me␣" + (i+1) + "␣-␣" + PREDEFINED_PROBLEMS.get(i
            predefinedProblem.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    onEditProblem(problemIndex);
                }});
            editMenu.add(predefinedProblem);
        }

        editMenu.add(new JSeparator());

        JMenuItem editProblem = new JMenuItem("Ã‰diter␣le␣problÃ¨me...");
        editMenu.add(editProblem);
        editProblem.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                onEditProblem();
            }});

        return bar;
    }

    protected void onEditProblem(int predefinedProblemIndex) {
        widget.setSolution(PREDEFINED_PROBLEMS.get(predefinedProblemIndex));
        updateResultLabel();
    }

    protected void onEditProblem() {
        EditProblemDialog dialog = new EditProblemDialog(this);
        dialog.setProblem(getProblem());
        dialog.pack();
        dialog.setVisible(true);

        int op = dialog.getUserOperation();
        if (op == JOptionPane.OK_OPTION) {
            CircleSolver solution = dialog.getProblem();
            widget.setSolution(solution);
            updateResultLabel();
        }
    }

    public CircleSolver getSolution() {
        return widget.getSolution();
    }

    protected class SliderChangeListener implements ChangeListener {
        @Override
        public void stateChanged(ChangeEvent e) {
            widget.setVisiblePortion(slider.getValue() / (double)MAX_SLIDER_VALUE);
        }
    }

    protected class RepaintOnStartup extends WindowAdapter {
```

```
    @Override
    public void windowOpened(WindowEvent e) {
        slider.setValue(MAX_SLIDER_VALUE);
        widget.setVisiblePortion(1.0);
        widget.repaint();
    }
}

protected class AnimateThread extends Thread {
    public void run() {
        final int steps = MAX_SLIDER_VALUE;
        for (int i=0; i<steps; i++) {
            slider.setValue(i);

            try {
                Thread.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
                break;
            }
        }

        slider.setValue(MAX_SLIDER_VALUE);
        setControlWidgetsEnabled(true);
    }
}

protected void setControlWidgetsEnabled(boolean b) {
    slider.setEnabled(b);
    playButton.setEnabled(b);
    previousEventButton.setEnabled(b);
    nextEventButton.setEnabled(b);
}

protected void onPreviousEvent() {
    Double previousEvent = getPreviousEvent();
    if (previousEvent == null) {
        return;
    }

    double ratio = getEventRatio(previousEvent);
    int newSliderValue = (int)(ratio * MAX_SLIDER_VALUE);
    slider.setValue(newSliderValue);
}

protected void onNextEvent() {
    Double nextEvent = getNextEvent();
    if (nextEvent == null) {
        return;
    }

    double ratio = getEventRatio(nextEvent);
    int newSliderValue = (int)(ratio * MAX_SLIDER_VALUE);
    slider.setValue(newSliderValue);
}

private double getEventRatio(double time) {
    double maxTime = getSolution().getTimeElapsed();
    return time / maxTime;
}

private Double getPreviousEvent() {
    double sliderRatio = getCurrentSliderRatio();
    double maxTime = getSolution().getTimeElapsed();
    double currentTime = maxTime*sliderRatio;

    List<Double> times = getSolution().getEventTimes();
    Collections.reverse(times);
    for (double aTime : times) {
        if (aTime < currentTime) {
            return aTime;
        }
    }

    return null;
}

private Double getNextEvent() {
    double sliderRatio = getCurrentSliderRatio();
    double maxTime = getSolution().getTimeElapsed();
    double currentTime = maxTime*sliderRatio;

    List<Double> times = getSolution().getEventTimes();
    for (double aTime : times) {
        if (aTime > currentTime) {
            // This time is after the current time.
            // But because of rounding errors, it might be the current one...
```

```
                int newSliderValue = (int)(getEventRatio(aTime)*MAX_SLIDER_VALUE);
                int currentSliderValue = slider.getValue();
                if (newSliderValue <= currentSliderValue) {
                    // This would not move the slider forward...
                    continue;
                }
                return aTime;
            }
        }

        return null;
    }

    private double getCurrentSliderRatio() {
        return slider.getValue() / (double)MAX_SLIDER_VALUE;
    }
}

@SuppressWarnings("serial")
protected static class EditProblemDialog extends JDialog {
    private static final int MAX_AGENTS = 10;
    private int userOperation;
    private JCheckBox agentEnabled[] = new JCheckBox[MAX_AGENTS];
    private JSlider agentPosition[] = new JSlider[MAX_AGENTS];

    public EditProblemDialog(Window parent) {
        super(parent, "Ã‰dition du problÃ¨me");
        setModal(true);

        GridBagLayout bag = new GridBagLayout();
        JPanel center = new JPanel(bag);

        for (int i=0; i<MAX_AGENTS; i++) {
            final int index = i;
            agentEnabled[i] = new JCheckBox();
            agentEnabled[i].addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    onToggleCheckbox(index);
                }
            });
            GridBagConstraints c = new GridBagConstraints();
            c.gridx = 0;
            c.gridy = i;
            c.gridheight = 1;
            c.gridwidth = 1;
            center.add(agentEnabled[i], c);

            agentPosition[i] = new JSlider(0,1000);
            c = new GridBagConstraints();
            c.gridx = 1;
            c.gridy = i;
            c.gridheight = 1;
            c.gridwidth = GridBagConstraints.REMAINDER;
            center.add(agentPosition[i], c);
        }

        JPanel bottom = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                onOk();
            }
        });
        JButton cancelButton = new JButton("Annuler");
        cancelButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                onCancel();
            }
        });
        bottom.add(okButton);
        bottom.add(cancelButton);

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(center, BorderLayout.CENTER);
        getContentPane().add(bottom, BorderLayout.SOUTH);
    }

    protected void onToggleCheckbox(int i) {
        if (agentEnabled[i].isSelected()) {
            // Enabling.
            agentPosition[i].setEnabled(true);
        }
        else {
```

```
            // Disabling.
            agentPosition[i].setEnabled(false);
        }
    }

    public void setProblem(Circle c) {
        for (int i=0; i<c.getAgents().size(); i++) {
            agentEnabled[i].setSelected(true);

            double positionOnRing = c.getAgent(i).getOrigin();
            double positionOnSlider = (positionOnRing/(2*Math.PI))*1000;

            agentPosition[i].setValue((int)positionOnSlider);
            agentPosition[i].setEnabled(true);
        }
        for (int i=c.getAgents().size(); i<MAX_AGENTS; i++) {
            agentEnabled[i].setSelected(false);
            agentPosition[i].setValue(0);
            agentPosition[i].setEnabled(false);
        }
    }

    protected void onOk() {
        if (getProblem() == null) {
            return;
        }

        userOperation = JOptionPane.OK_OPTION;
        dispose();
    }

    protected void onCancel() {
        userOperation = JOptionPane.CANCEL_OPTION;
        dispose();
    }

    public int getUserOperation() {
        return userOperation;
    }

    public CircleSolver getProblem() {
        Circle circle = new Circle();
        for (int i=0; i<MAX_AGENTS; i++) {
            if (agentEnabled[i].isSelected() == false) {
                continue;
            }

            Agent anAgent = new Agent(String.valueOf((char)('A'+i)));
            double position = agentPosition[i].getValue();
            position = (position / 1000) * (2*Math.PI);
            anAgent.setPosition(position);

            circle.addAgent(anAgent);
        }

        if (circle.getAgents().size() <= 2) {
            JOptionPane.showMessageDialog(this, "Vous devez configurer au moins trois agents.", "Nombre d'agents
            return null;
        }

        CircleSolver solver = new CircleSolver();
        solver.init(circle);
        solver.setBoundaryValidation(false);
        try {
            return solver.solve();
        }
        catch(Exception e) {
            e.printStackTrace();
            JOptionPane.showMessageDialog(this,
                    "Une erreur inattendue est survenue. Veuillez essayer une autre combinaison.",
                    "Oups!", JOptionPane.ERROR_MESSAGE);
            return null;
        }
    }
}

private static java.util.List<CircleSolver> PREDEFINED_PROBLEMS;
static {
    PREDEFINED_PROBLEMS = new LinkedList<CircleSolver>();

    CircleSolver solu;
    BoundaryTest test = new BoundaryTest();

    test.setupCase01();
    solu = test.getSolution(true);
    solu.setLabel("Deux agents rapprochÃ©s");
```

```
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase02();
        solu = test.getSolution(true);
        solu.setLabel("Deux agents antipodaux");
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase03ModifiedForFinalPresentation();
        solu = test.getSolution(true);
        solu.setLabel("Trois agents A (dÃ©faut)");
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase06();
        solu = test.getSolution(false);
        solu.setLabel("Trois agents B+");
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase06();
        solu = test.getSolution(false);
        Circle mirror = new Circle(solu.getCircle());
        mirror.mirror();
        solu = new CircleSolver();
        solu.init(mirror);
        solu.setMirror(true);
        solu = solu.solve(false);
        solu.setLabel("Trois agents B-");
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase03();
        solu = test.getSolution(true);
        solu.setLabel("Trois agents C");
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase04();
        solu = test.getSolution(true);
        solu.setLabel("Quatre agents");
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase05();
        solu = test.getSolution(true);
        solu.setLabel("Quatre agents (deux agents dÃ©butant au mÃªme point)");
        PREDEFINED_PROBLEMS.add(solu);

        test.setupCase07();
        solu = test.getSolution(true);
        solu.setLabel("Six agents");
        PREDEFINED_PROBLEMS.add(solu);
    }

    public static CircleSolver getDefaultSolution() {
        return PREDEFINED_PROBLEMS.get(2);
    }

    @Override
    public Dimension getPreferredSize() {
        return new Dimension(500,500);
    }

    public void paint(Graphics g) {
        SolutionPainter painter = new SolutionPainter(getSolution(),
                getSize().width,
                getSize().height);
        painter.setVisiblePortion(visiblePortion);
        painter.paint((Graphics2D)g);
    }

    public CircleSolver getSolution() {
        return solution;
    }

    public void setSolution(CircleSolver solution) {
        this.solution = solution;
        repaint();
    }

    public double getVisiblePortion() {
        return visiblePortion;
    }

    public void setVisiblePortion(double visiblePortion) {
        this.visiblePortion = visiblePortion;
        repaint();
    }
}
```

A.10 SolutionPainter.java

```
package net.fortrel.circle2;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Insets;
import java.awt.Point;
import java.awt.Polygon;
import java.awt.RenderingHints;
import java.awt.geom.Point2D;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class SolutionPainter {
    private CircleSolver solution;
    private int width;
    private int height;

    private Insets axisInsets = new Insets(80,30,30,50);
    private int arrowWidth = 11;
    private int arrowLength = 15;
    private int axisOverflow = 10;

    private double visiblePortion = 1.0; // 0-1

    public SolutionPainter(CircleSolver solution, int width, int height) {
        this.solution = solution;
        this.width = width;
        this.height = height;
    }

    public void paint(Graphics2D g) {
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

        g.setColor(Color.white);
        g.fillRect(0, 0, width, height);

        g.setStroke(new BasicStroke(1));

        for (Agent anAgent : solution.getCircle().getAgents()) {
            List<Point2D.Double> positiveBoundaryPath = solution.getPositiveBoundaryPath(anAgent.getName());
            List<Point2D.Double> negativeBoundaryPath = solution.getNegativeBoundaryPath(anAgent.getName());

            for (Polygon aShape : getAreaBetweenPath(anAgent, positiveBoundaryPath, negativeBoundaryPath)) {
                g.setColor(getFillColor(anAgent));
                g.fill(aShape);

                g.setColor(getBoundaryColor(anAgent));
                g.drawPolygon(aShape);
            }
        }

        hideNonVisiblePortion(g);

        drawEventTimes(g);

        drawAxis(g);

        drawTitle(g);
    }

    protected void drawTitle(Graphics2D g) {
        g.setColor(Color.BLACK);

        int width = g.getFontMetrics().stringWidth(getTitle());
        int height = g.getFontMetrics().getHeight();

        int x = (this.width - width) / 2;
        int y = (axisInsets.top - height) /2;

        g.drawString(getTitle(), x, y);
    }

    public String getTitle() {
        return "Exploration d'un anneau par des agents mobiles";
    }

    protected void hideNonVisiblePortion(Graphics2D g) {
        // Now, hide the portion not visible.
        int eventHorizonPixel = (int)((width-(axisInsets.left+axisInsets.right))
                * getVisiblePortion())+axisInsets.left;
        g.setColor(Color.white);
```

```
        g.fillRect(eventHorizonPixel, 0, width, height);
    }

    protected void drawEventTimes(Graphics2D g) {
        List<Double> events = solution.getEventTimes();

        g.setColor(Color.lightGray);
        for (Double d : events) {
            int x = toPixelX(d);
            int y1 = toPixelY(0);
            int y2 = toPixelY(2*Math.PI);
            g.drawLine(x, y1, x, y2);
        }
    }

    protected String pathToString(List<Point2D.Double> path) {
        StringBuffer buffer = new StringBuffer();
        for (Point2D.Double aPoint : path) {
            if (buffer.length() > 0) {
                buffer.append(",␣");
            }
            buffer.append(aPoint.toString());
        }
        return buffer.toString();
    }

    protected Color getFillColor(Agent anAgent) {
        Color fillColors[] = new Color[] {
            // Pink
            new Color(255, 153, 153),
            // Light blue
            new Color(0, 153, 255),
            // Light green
            new Color(204, 255, 204),
            // Light grey
            new Color(204, 204, 204),
            // Light yellow
            new Color(255, 255, 153),
            // Light purple
            new Color(255, 204, 255)
        };

        return fillColors[anAgent.getIndex() % fillColors.length];
    }

    protected Color getBoundaryColor(Agent anAgent) {
        Color boundaryColors[] = new Color[] {
            Color.red,
            Color.blue,
            Color.green,
            Color.darkGray,
            // Dark yellow (ocre)
            new Color(255, 204, 0),
            // Dark purple
            new Color(153, 0, 204)
        };

        return boundaryColors[anAgent.getIndex() % boundaryColors.length];
    }

    protected Polygon[] getAreaBetweenPath(Agent anAgent, List<Point2D.Double> upperBound,
            List<Point2D.Double> lowerBound) {
        List<Polygon> polygons = new LinkedList<Polygon>();
        polygons.add(getMiddlePolygon(anAgent, upperBound, lowerBound));

        Polygon aboveOrigin = getUpperBoundOverOriginPolygon(upperBound);
        if (aboveOrigin != null) {
            polygons.add(aboveOrigin);
        }

        Polygon belowOrigin = getLowerBoundBelowOriginPolygon(lowerBound);
        if (belowOrigin != null) {
            polygons.add(belowOrigin);
        }

        return polygons.toArray(new Polygon[polygons.size()]);
    }

    private Polygon createPolygonBetweenBoundaries(
            List<Point2D.Double> upperBound,
            List<Point2D.Double> lowerBound) {
        Polygon poly = new Polygon();

        for (Point2D.Double aPoint : upperBound) {
            Point pix = toPixel(aPoint);
            poly.addPoint(pix.x, pix.y);
        }
```

```
        List<Point2D.Double> reverse = new LinkedList<Point2D.Double>();
        reverse.addAll(lowerBound);
        Collections.reverse(reverse);

        for (Point2D.Double aPoint : reverse) {
            Point pix = toPixel(aPoint);
            poly.addPoint(pix.x, pix.y);
        }

        return poly;
    }

    private Polygon getLowerBoundBelowOriginPolygon(List<Point2D.Double> lowerBound) {
        if (lowerBoundaryCrossesOrigin(lowerBound) == false) {
            return null;
        }

        //System.out.println("Lower bound crosses origin!");

        double lastTime = lowerBound.get(lowerBound.size()-1).getX();
        double timeOfCrossing = getTimeWhenLowerBoundaryCrossesOrigin(lowerBound);

        List<Point2D.Double> upperBound = new LinkedList<Point2D.Double>();
        List<Point2D.Double> lowerBoundBelowOrigin = new LinkedList<Point2D.Double>();

        upperBound.add(new Point2D.Double(timeOfCrossing, 2*Math.PI));
        upperBound.add(new Point2D.Double(lastTime, 2*Math.PI));

        lowerBoundBelowOrigin.add(new Point2D.Double(timeOfCrossing, 2*Math.PI));

        for (Point2D.Double aPoint : lowerBound) {
            if (aPoint.x > timeOfCrossing) {
                lowerBoundBelowOrigin.add(aPoint);
            }
        }

        return createPolygonBetweenBoundaries(upperBound, lowerBoundBelowOrigin);
    }

    private Polygon getUpperBoundOverOriginPolygon(List<Point2D.Double> upperBound) {
        if (upperBoundaryCrossesOrigin(upperBound) == false) {
            return null;
        }

        List<Point2D.Double> middleUpperBound = new LinkedList<Point2D.Double>();
        List<Point2D.Double> middleLowerBound = new LinkedList<Point2D.Double>();

        double lastTime = upperBound.get(upperBound.size()-1).getX();
        double timeOfCrossing = getTimeWhenUpperBoundaryCrossesOrigin(upperBound);

        middleUpperBound.add(new Point2D.Double(timeOfCrossing, 0));
        System.out.println(middleUpperBound.get(0));
        middleLowerBound.add(new Point2D.Double(timeOfCrossing, 0));
        for (Point2D.Double aPoint : upperBound) {
            if (aPoint.x <= timeOfCrossing) {
                continue;
            }
            middleUpperBound.add(aPoint);
            System.out.println(aPoint);
        }
        middleLowerBound.add(new Point2D.Double(lastTime, 0));

        return createPolygonBetweenBoundaries(middleUpperBound, middleLowerBound);
    }

    private Polygon getMiddlePolygon(Agent anAgent,
            List<Point2D.Double> upperBound,
            List<Point2D.Double> lowerBound) {
        List<Point2D.Double> middleUpperBound = new LinkedList<Point2D.Double>();
        List<Point2D.Double> middleLowerBound = new LinkedList<Point2D.Double>();

        double lastTime = upperBound.get(upperBound.size()-1).getX();

        if (upperBoundaryCrossesOrigin(upperBound)) {
            double timeOfCrossing = getTimeWhenUpperBoundaryCrossesOrigin(upperBound);

            for (Point2D.Double anUpperBoundPoint : upperBound) {
                if (anUpperBoundPoint.x <= timeOfCrossing) {
                    middleUpperBound.add(anUpperBoundPoint);
                }
                else {
                    middleUpperBound.add(new Point2D.Double(timeOfCrossing, (2*Math.PI)));
                    middleUpperBound.add(new Point2D.Double(lastTime, (2*Math.PI)));
                    break;
                }
            }
```

```
            // Clone the lower bound.
            middleLowerBound.addAll(lowerBound);

            return createPolygonBetweenBoundaries(middleUpperBound, middleLowerBound);
        }

        if (lowerBoundaryCrossesOrigin(lowerBound)) {
            middleUpperBound.addAll(upperBound);

            double timeOfCrossing = getTimeWhenLowerBoundaryCrossesOrigin(lowerBound);
            //System.out.println("Time of crossing: " + timeOfCrossing);
            for (Point2D.Double aPoint : lowerBound) {
                if (aPoint.x <= timeOfCrossing) {
                    middleLowerBound.add(aPoint);
                }
                else {
                    middleLowerBound.add(new Point2D.Double(timeOfCrossing, 0));
                    middleLowerBound.add(new Point2D.Double(lastTime, 0));
                    break;
                }
            }

            return createPolygonBetweenBoundaries(middleUpperBound, middleLowerBound);
        }

        // Both the upper and lower boundary does not
        // cross the origin.
        middleUpperBound.addAll(upperBound);
        middleLowerBound.addAll(lowerBound);

        return createPolygonBetweenBoundaries(middleUpperBound, middleLowerBound);
    }

    protected boolean upperBoundaryCrossesOrigin(List<Point2D.Double> path) {
        double origin = path.get(0).y;
        for (int i=0; i<path.size(); i++) {
            Point2D.Double up = path.get(i);

            if (up.y < origin) {
                return true;
            }
        }
        return false;
    }

    protected boolean lowerBoundaryCrossesOrigin(List<Point2D.Double> path) {
        double origin = path.get(0).y;
        for (int i=0; i<path.size(); i++) {
            Point2D.Double up = path.get(i);

            if (up.y > origin) {
                return true;
            }
        }
        return false;
    }

    protected double getTimeWhenUpperBoundaryCrossesOrigin(List<Point2D.Double> path) {
        double origin = path.get(0).y;
        Point2D.Double previousPoint = null;
        for (int i=0; i<path.size(); i++) {
            Point2D.Double up = path.get(i);

            if (up.y < origin) {
                //System.out.println("Crossed origin!");

                double previousTime = previousPoint.getX();
                double previousPosition = previousPoint.getY();
                double thisTime = up.getX();
                double thisPosition = up.getY();
                thisPosition += (2 * Math.PI);
                double slope = (thisPosition - previousPosition) / (thisTime - previousTime);
                //System.out.println(slope);

                double distanceLeft = (2*Math.PI) - previousPosition;
                double timeLeft = distanceLeft / slope;
                return previousTime + timeLeft;
            }

            previousPoint = up;
        }
        throw new IllegalStateException("Upper boundary did not cross origin.");
    }

    protected double getTimeWhenLowerBoundaryCrossesOrigin(List<Point2D.Double> path) {
        double origin = path.get(0).y;
```

```
        Point2D.Double previousPoint = null;
        for (int i=0; i<path.size(); i++) {
            Point2D.Double up = path.get(i);

            if (up.y > origin) {
                //System.out.println("Crossed origin!");

                double previousTime = previousPoint.getX();
                double previousPosition = previousPoint.getY();
                double thisTime = up.getX();
                double thisPosition = up.getY();
                thisPosition -= (2 * Math.PI);
                double slope = (thisPosition - previousPosition) / (thisTime - previousTime);
                //System.out.println(slope);

                double distanceLeft = previousPosition;
                double timeLeft = distanceLeft / slope;
                return previousTime + timeLeft;
            }

            previousPoint = up;
        }
        throw new IllegalStateException("Lower boundary did not cross origin.");
    }

    protected void drawPath(Graphics2D g, List<Point2D.Double> path) {
        for (int i=1; i<path.size(); i++) {
            Point2D.Double from = path.get(i-1);
            Point2D.Double to = path.get(i);

            Point pixFrom = toPixel(from);
            Point pixTo = toPixel(to);

            g.drawLine(pixFrom.x, pixFrom.y, pixTo.x, pixTo.y);
        }
    }

    protected Point toPixel(Point2D.Double inDomain) {
        return new Point(toPixelX(inDomain.getX()), toPixelY(inDomain.getY()));
    }

    protected int toPixelY(double inDomain) {
        double totalCirc = 2 * Math.PI;
        double yRatio = inDomain / totalCirc;

        int graphAreaHeight = height - (axisInsets.top+axisInsets.bottom);

        int y = (int)(yRatio * graphAreaHeight);

        y = (graphAreaHeight - y) + axisInsets.top;

        return y;
    }

    protected int toPixelX(double inDomain) {
        double totalTime = solution.getTimeElapsed();
        double xRatio = inDomain / totalTime;

        int graphAreaWidth = width - (axisInsets.left+axisInsets.right);

        int x = (int)(xRatio * graphAreaWidth);

        x += axisInsets.left;

        return x;
    }

    protected void drawAxis(Graphics2D g) {
        g.setColor(Color.black);
        g.drawLine(axisInsets.left, height-axisInsets.bottom,
                (width-axisInsets.right)+axisOverflow, height-axisInsets.bottom);
        g.drawLine(axisInsets.left, axisInsets.top-axisOverflow,
                axisInsets.left, height-axisInsets.bottom);

        Polygon rightArrow = new Polygon();
        rightArrow.addPoint((width-axisInsets.right)+axisOverflow, height-(axisInsets.bottom+arrowWidth/2)-1);
        rightArrow.addPoint((width-axisInsets.right)+arrowLength+axisOverflow, height-axisInsets.bottom);
        rightArrow.addPoint((width-axisInsets.right)+axisOverflow, ((height-axisInsets.bottom)+arrowWidth/2));
        g.fillPolygon(rightArrow);

        Polygon upArrow = new Polygon();
        upArrow.addPoint((axisInsets.left-arrowWidth/2)-1, axisInsets.top-axisOverflow);
        upArrow.addPoint(axisInsets.left, axisInsets.top-arrowLength-axisOverflow);
        upArrow.addPoint((axisInsets.left+arrowWidth/2)+1, axisInsets.top-axisOverflow);
        g.fillPolygon(upArrow);

        String rightText = "T";
```

```
        int width = g.getFontMetrics().stringWidth(rightText);
        int height = g.getFontMetrics().getHeight();
        int x = this.width - axisInsets.right + arrowLength + axisOverflow;
        int y = this.height - axisInsets.bottom;
        x += width/2;
        y += height/2;
        g.drawString(rightText, x, y);

        String leftText = "P";
        x = axisInsets.left/2;
        y = axisInsets.top - arrowLength - axisOverflow;
        g.drawString(leftText, x, y);

    }

    public void setVisiblePortion(double visiblePortion) {
        this.visiblePortion = visiblePortion;
    }

    public double getVisiblePortion() {
        return visiblePortion;
    }
}
```

A.11 State.java

```
package net.fortrel.circle2;

public interface State {
    public static final double MAX_BOUND_DIFF = 0.00001;

    public Agent getAgent();
    public String getName();
    public double getTransitionTime();
    public State nextState();
    public void complete();
    public void setBoundarySpeeds();
    public double getGroundCovered();
    public void assertAccurateBoundaries();
    public boolean isGridLocked();
}
```

A.12 StateChange.java

```
package net.fortrel.circle2;

public class StateChange implements Comparable<StateChange> {
    private Agent agent;
    private double baseTime;
    private double time;

    public StateChange(Agent anAgent, double baseTime, double time) {
        this.agent = anAgent;
        this.baseTime = baseTime;
        this.time = time;
    }

    public Agent getAgent() {
        return agent;
    }
    public double getTime() {
        return time;
    }
    public double getBaseTime() {
        return baseTime;
    }
    public double getAbsoluteTime() {
        return getBaseTime() + getTime();
    }

    public int compareTo(StateChange o) {
        double diff = getAbsoluteTime() - o.getAbsoluteTime();
        if (diff > 0) {
            return 1;
        }
        else if (diff < 0) {
            return -1;
        }
        else {
```

```
            return 0;
        }
    }

    @Override
    public String toString() {
        return toString(false, -1);
    }

    public String toString(boolean absoluteTimeOnly, int precision) {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Agent ").append(getAgent().getName());

        if (absoluteTimeOnly) {
            buffer.append(" at ");
            buffer.append(trimToPrecision(getAbsoluteTime(), precision));
        }
        else {
            buffer.append(" in ");
            buffer.append(trimToPrecision(getTime(), precision));
            buffer.append(" (at ");
            buffer.append(trimToPrecision(getAbsoluteTime(), precision));
            buffer.append(")");
        }

        return buffer.toString();
    }

    private static String trimToPrecision(double d, int precision) {
        if (d == Double.MAX_VALUE) {
            return "MAX";
        }
        String asString = String.valueOf(d);
        if (precision <= 0) {
            return asString;
        }
        else {
            if (asString.length() < precision) {
                precision = asString.length();
            }
            return asString.substring(0, precision);
        }
    }

    public boolean equals(Object o) {
        if (o.getClass().equals(getClass()) == false) {
            return false;
        }
        StateChange other = (StateChange)o;
        if (other.getAgent().equals(getAgent()) == false) {
            return false;
        }
        if (Math.abs(other.getAbsoluteTime() - getAbsoluteTime()) > State.MAX_BOUND_DIFF) {
            return false;
        }
        return true;
    }
}
```

A.13 State1.java

```
package net.fortrel.circle2.states;

import net.fortrel.circle2.Agent;
import net.fortrel.circle2.Boundary;
import net.fortrel.circle2.Circle;
import net.fortrel.circle2.CircleSolver;
import net.fortrel.circle2.Direction;
import net.fortrel.circle2.State;

/**
 * Go clockwise only.
 */
public class State1 implements State {
    private Agent agent;
    private CircleSolver solver;

    public State1(CircleSolver parent, Agent anAgent) {
        this.agent = anAgent;
        this.solver = parent;
    }

    public Agent getAgent() {
```

```
    return agent;
}

public void setAgent(Agent agent) {
    this.agent = agent;
}

public String getName() {
    return "1";
}

private static void log(String str) {
    //System.out.println(str);
}

/**
 * Performance: O(1)
 */
@Override
public double getTransitionTime() {
    Boundary negBoundary = solver.getNegativeBoundary(agent);

    double position = Circle.toPositivePosition(agent.getOrigin() - solver.getTimeElapsed());

    double distance = Circle.distanceFrom(
            position,
            negBoundary.getPosition(solver.getTimeElapsed()),
            Direction.Clockwise);
    //log("Current agent's position: " + agent.getPosition());
    //log("Negative boundary: " + negBoundary.getPosition());
    //log("Ground to cover: " + distance);

    if (distance < 0) {
        throw new IllegalStateException("Distance to cover cannot be negative. (" + distance + ")");
    }

    //log("Negative boundary moving at " + negBoundary.getSpeed());
    //log("Positive boundary moving at " + posBoundary.getSpeed());

    double relativeSpeed = 1 + negBoundary.getSpeed();
    //log("Relative speed to negative boundary " + relativeSpeed);
    double transitionTime = distance / relativeSpeed;
    //log("Will hit negative boundary in " + transitionTime + " sec.");
    return transitionTime;
}

public State nextState() {
    return new State2(solver, agent);
}

@Override
public void setBoundarySpeeds() {
    Agent previous = getAgent().getPrevious();
    solver.getPositiveBoundary(previous).setSpeed(-1.0, solver.getTimeElapsed());
}

@Override
public void complete() {
    solver.getNegativeBoundary(getAgent()).setReached(true);
    solver.getPositiveBoundary(getAgent().getPrevious()).setReached(true);
    solver.decrementNumberOfAgentsLeftInState1();
}

@Override
public double getGroundCovered() {
    return solver.getTimeElapsed();
}

/**
 * Performance: O(1)
 */
@Override
public void assertAccurateBoundaries() {
    double originalPositiveBound = Circle.toPositivePosition(
            solver.getNegativeBoundary(agent.getNext()).getPosition(solver.getTimeElapsed()));
    double expectedPositibeBound = Circle.toPositivePosition(agent.getOrigin());

    if (!Circle.areApproximatelyTheSame(originalPositiveBound, expectedPositibeBound)) {
        throw new IllegalStateException("State 1 positive boundary calculated differs.");
    }

    if (solver.getPositiveBoundary(agent).isReached()) {
        Boundary nextNegativeBound = solver.getNegativeBoundary(agent.getNext());

        if (Math.abs(originalPositiveBound - nextNegativeBound.getPosition(solver.getTimeElapsed()))
                > MAX_BOUND_DIFF) {
            throw new IllegalStateException("State 1 positive boundary calculated differs.");
```

```
            }
        }

        double originalNegativeBound = Circle.toPositivePosition(
                solver.getPositiveBoundary(agent.getPrevious()).getPosition(solver.getTimeElapsed()));
        double expectedNegativeBound = Circle.toPositivePosition(
                agent.getOrigin() - solver.getTimeElapsed());

        // Found a problem!
        if (!Circle.areApproximatelyTheSame(originalNegativeBound, expectedNegativeBound)) {
            //solver.dumpState();
            log("Agent " + agent.getName());
            log("Original negative boundary: " + originalNegativeBound);
            log("Re-calculated negative boundary: " + expectedNegativeBound);
            log("Next agent's origin: " + agent.getNext().getOrigin());
            log("Delta:" + Math.abs(originalNegativeBound - expectedNegativeBound));
            throw new IllegalStateException("Agent " + agent.getName()
                    + " state 1 negative boundary calculated differs.");
        }

        if (solver.getNegativeBoundary(agent).isReached()) {
            Boundary previousPositiveBound = solver.getPositiveBoundary(agent.getPrevious());

            if (Math.round(originalNegativeBound - previousPositiveBound.getPosition(solver.getTimeElapsed()))
                    > MAX_BOUND_DIFF) {
                throw new IllegalStateException("State 1 negative boundary calculated differs.");
            }
        }
    }

    @Override
    public boolean isGridLocked() {
        // State 1 is never grid-locked.
        return false;
    }
}
```

A.14 State2.java

```
package net.fortrel.circle2.states;

import net.fortrel.circle2.Agent;
import net.fortrel.circle2.Boundary;
import net.fortrel.circle2.Circle;
import net.fortrel.circle2.CircleSolver;
import net.fortrel.circle2.Direction;
import net.fortrel.circle2.State;

public class State2 implements State {
    private static final boolean logsEnabled = false;

    private Agent agent;
    private CircleSolver solver;
    private boolean willLockInState2OnFollowingTransition = false;
    private boolean lockedInState2 = false;

    public State2(CircleSolver parent, Agent agent) {
        this.agent = agent;
        this.solver = parent;
    }

    public Agent getAgent() {
        return agent;
    }

    public String getName() {
        return "2";
    }

    private static void log(String str) {
        if (logsEnabled) {
            System.out.println(str);
        }
    }

    /**
     * Performance: O(1)
     */
    @Override
    public double getTransitionTime() {
        willLockInState2OnFollowingTransition = false;

        Boundary negBoundary = solver.getNegativeBoundary(agent);
```

```
        Boundary posBoundary = solver.getPositiveBoundary(agent);
        double distanceCoveredOnce = Circle.distanceFrom(agent.getOrigin(),
                negBoundary.getPosition(solver.getTimeElapsed()), Direction.Clockwise);
        double distanceCoveredTwice = (solver.getTimeElapsed() - distanceCoveredOnce)/2;

        if (logsEnabled) {
            double distanceToNegBound = Circle.distanceFrom(agent.getPosition(),
                    negBoundary.getPosition(solver.getTimeElapsed()), Direction.Clockwise);
            double distanceToPosBound = Circle.distanceFrom(agent.getPosition(),
                    posBoundary.getPosition(solver.getTimeElapsed()), Direction.CounterClockwise);
            double distanceFromOriginToPosBound = Circle.distanceFrom(agent.getOrigin(),
                    posBoundary.getPosition(solver.getTimeElapsed()), Direction.CounterClockwise);

            log("== Agent " + agent.getName() + " ==");
            log("Between " + negBoundary.getPosition(solver.getTimeElapsed()) + " and "
                    + posBoundary.getPosition(solver.getTimeElapsed()));
            log("Negative boundary moving at " + negBoundary.getSpeed());
            log("Distance to negative boundary: " + distanceToNegBound);
            log("Distance to positive boundary: " + distanceToPosBound);
            log("Distance from origin to positive boundary: " + distanceFromOriginToPosBound);
            log("Distance to negative boundary: " + distanceCoveredOnce);
            log("Time elapsed: " + solver.getTimeElapsed());
            log("Distance already covered twice: " + distanceCoveredTwice);
        }

        if (distanceCoveredTwice < -State.MAX_BOUND_DIFF) {
            throw new IllegalStateException("Distance covered twice cannot have a negative value. ("
                    + distanceCoveredTwice + ")");
        }

        if (isGridLocked()) {
            log("Grid locked!");
            return Double.MAX_VALUE;
        }

        double timeTillSwitchToS3 =
                (2*(distanceCoveredOnce - distanceCoveredTwice))
                / ((3*negBoundary.getSpeed())+1);

        double exploredAreaPositiveBoundarySpeed = (negBoundary.getSpeed()/2)+0.5;
        double nextAgentOrigin = agent.getNext().getOrigin();
        double exploredAreaCurrentBoundary = agent.getPosition() + distanceCoveredTwice;
        double distanceToNextAgentOrigin = Circle.distanceFrom(agent.getOrigin(),
                nextAgentOrigin, Direction.CounterClockwise);
        double distanceLeftToReachNextAgentOrigin = distanceToNextAgentOrigin - distanceCoveredTwice;
        double timeToReachNextAgentOrigin = distanceLeftToReachNextAgentOrigin
                / exploredAreaPositiveBoundarySpeed;

        if (logsEnabled) {
            log(">>> Positive boundary at " + nextAgentOrigin);
            log(">>> Own positive boundary speed: " + exploredAreaPositiveBoundarySpeed);
            log(">>> Already covering " + distanceCoveredTwice + " toward next agent (reached "
                    + exploredAreaCurrentBoundary + ").");
            log(">>> Distance left to reach next agent's origin: " + distanceLeftToReachNextAgentOrigin);
            log(">>> Would reach the next agent's origin in " + timeToReachNextAgentOrigin);
            log(">>> Would switch to State 3 in " + timeTillSwitchToS3);
        }

        if (timeToReachNextAgentOrigin < State.MAX_BOUND_DIFF) {
            return Double.MAX_VALUE;
        }

        if (timeToReachNextAgentOrigin < timeTillSwitchToS3) {
            log(">>> Will reach next agent's origin before switching to State 3.");
            willLockInState2OnFollowingTransition = true;
            return timeToReachNextAgentOrigin;
        }
        else {
            log(">>> Will switch to State 3 before reaching the next agent's origin.");
            return timeTillSwitchToS3;
        }
    }

    public State nextState() {
        if (lockedInState2) {
            return this;
        }
        return new State3(solver, getAgent());
    }

    @Override
    public void complete() {
        if (willLockInState2OnFollowingTransition) {
            lockedInState2 = true;
        }

        // Nothing to do.
```

```
        log("Agent " + agent.getName() + " goes to state 3.");
    }

    /**
     * Performance: O(1)
     */
    @Override
    public void setBoundarySpeeds() {
        double ownNegBoundSpeed = solver.getNegativeBoundary(agent).getSpeed();
        //log("Own's (" + getAgent().getName() + ") negative boundary speed: " + ownNegBoundSpeed);

        Agent previous = getAgent().getPrevious();
        solver.getPositiveBoundary(previous).setSpeed(ownNegBoundSpeed, solver.getTimeElapsed());

        Agent next = getAgent().getNext();

        if (lockedInState2) {
            solver.getNegativeBoundary(next).setSpeed(0, solver.getTimeElapsed());
            solver.getPositiveBoundary(agent).setSpeed(0, solver.getTimeElapsed());
            return;
        }

        double nextAgentNegativeBoundarySpeed;
        if (isGridLocked()) {
            nextAgentNegativeBoundarySpeed = 0;
        }
        else {
            nextAgentNegativeBoundarySpeed = 0.5 + (ownNegBoundSpeed/2);
        }

        solver.getNegativeBoundary(next).setSpeed(nextAgentNegativeBoundarySpeed, solver.getTimeElapsed());
    }

    /**
     * Performance: O(1)
     */
    public double getGroundCoveredClockwise(CircleSolver solver) {
        Boundary currentAgentNegBound = solver.getNegativeBoundary(getAgent());
        return Circle.distanceFrom(getAgent().getOrigin(), currentAgentNegBound.getPosition(
                solver.getTimeElapsed()), Direction.Clockwise);
    }

    /**
     * Performance: O(1)
     */
    public double getGroundCoveredCounterClockwise(CircleSolver solver) {
        double distanceCoveredOnce = getGroundCoveredClockwise(solver);
        log("We have " + distanceCoveredOnce + " to cover to the negative bound.");
        double timeLeft = solver.getTimeElapsed() - distanceCoveredOnce;
        double distanceCoveredTwice = timeLeft/2;

        Boundary currentAgentPosBound = solver.getPositiveBoundary(getAgent());
        double distanceMaxToCoverTwice = Circle.distanceFrom(agent.getOrigin(),
                currentAgentPosBound.getPosition(solver.getTimeElapsed()), Direction.CounterClockwise);
        log("Should not go back more than " + distanceMaxToCoverTwice + " counter-clockwise.");

        if (distanceCoveredTwice >= distanceMaxToCoverTwice) {
            log("Grid-locked!");
            return distanceMaxToCoverTwice;
        }

        return distanceCoveredTwice;
    }

    /**
     * Performance: O(1)
     */
    @Override
    public boolean isGridLocked() {
        double distanceCoveredOnce = getGroundCoveredClockwise(solver);
        //log("We have " + distanceCoveredOnce + " to cover to the negative bound.");
        double timeLeft = solver.getTimeElapsed() - distanceCoveredOnce;
        double distanceCoveredTwice = timeLeft/2;

        //log("Agent's origin: " + agent.getOrigin());
        //log("Next agent's origin: " + agent.getNext().getOrigin());

        double distanceMaxToCoverTwice = Circle.distanceFrom(
                agent.getOrigin(),
                agent.getNext().getOrigin(),
                Direction.CounterClockwise);
        //log("Should not go back more than " + distanceMaxToCoverTwice + " counter-clockwise.");
        //log("Distance covered counter-clockwise: " + distanceCoveredTwice);

        return (distanceCoveredTwice >= distanceMaxToCoverTwice);
    }
```

```
    @Override
    public double getGroundCovered() {
        return getGroundCoveredClockwise(solver) + getGroundCoveredCounterClockwise(solver);
    }

    /**
     * Performance: O(1)
     */
    @Override
    public void assertAccurateBoundaries() {
        if (solver.getNegativeBoundary(agent).isReached() == false) {
            throw new IllegalStateException("Cannot be in state 2 if the negative boundary is not reached.");
        }
        if (Circle.absoluteDistanceBetween(
                solver.getNegativeBoundary(agent).getPosition(solver.getTimeElapsed()),
                solver.getPositiveBoundary(agent.getPrevious()).getPosition(solver.getTimeElapsed()))
                        > MAX_BOUND_DIFF) {
            throw new IllegalStateException("Negative boundary is inconsistent.");
        }

        double actualPositiveBoundPosition = solver.getNegativeBoundary(
                agent.getNext()).getPosition(solver.getTimeElapsed());

        if (lockedInState2) {
            //log("Agent " + agent.getName() + " is locked in State 2."
            //        + "Expecting positive boundary to be the origin of the next agent, or beyound that.");
            //log("  Next agent's origin: " + agent.getNext().getOrigin());
            //log("  Actual positive bound: " + actualPositiveBoundPosition);
            double delta = agent.getNext().getOrigin() - actualPositiveBoundPosition;
            //log("  Delta: " + delta);
            if (delta <= State.MAX_BOUND_DIFF) {
                // Good!
                return;
            }
            else {
                //solver.dumpState();
                throw new IllegalStateException("Negative boundary is inconsistent.");
            }
        }

        double negBound = solver.getNegativeBoundary(agent).getPosition(solver.getTimeElapsed());
        double origin = agent.getOrigin();
        double distanceCoveredOnce = Circle.distanceFrom(origin, negBound, Direction.Clockwise);
        //log("Agent " + agent.getName() + " in state 2; distance covered once: " + distanceCoveredOnce);
        double timeLeft = solver.getTimeElapsed() - distanceCoveredOnce;
        //log("Agent " + agent.getName() + " in state 2; time left: " + timeLeft);
        double distanceCoveredTwice = timeLeft/2;
        double expectedPositiveBoundPosition =
                Circle.toPositivePosition(origin + distanceCoveredTwice);
        //log("Agent " + agent.getName() + " in state 2; expected positive position is: "
        //        + expectedPositiveBoundPosition);
        //log("Agent " + agent.getName() + " in state 2; actual positive position is: "
        //        + actualPositiveBoundPosition);

        if (Circle.areApproximatelyTheSame(actualPositiveBoundPosition,
                expectedPositiveBoundPosition) == false) {
            solver.dumpState();
            throw new IllegalStateException("State 2 positive boundary (agent " + agent.getName()
                    + ") calculated differs :( (" + actualPositiveBoundPosition
                    + " != " + expectedPositiveBoundPosition + ")");
        }

        double distanceToNextAgentOrigin = Circle.distanceFrom(origin,
                agent.getNext().getOrigin(), Direction.CounterClockwise);
        if (distanceCoveredTwice - distanceToNextAgentOrigin > State.MAX_BOUND_DIFF) {
            solver.dumpState();
            //log("Problem with agent " + agent.getName());
            //log("Distance to cover to reach next agent's origin: " + distanceToNextAgentOrigin);
            //log("Distance covered in counter-clockwise direction: " + distanceCoveredTwice);
            throw new IllegalStateException("State 2 positibe boundary (agent " + agent.getName()
                    + ") cannot be set past agent " + agent.getNext().getName() + " origin.");
        }
    }
}
```

A.15 State3.java

```
package net.fortrel.circle2.states;

import net.fortrel.circle2.Agent;
import net.fortrel.circle2.Boundary;
import net.fortrel.circle2.Circle;
import net.fortrel.circle2.CircleSolver;
```

```
import net.fortrel.circle2.Direction;
import net.fortrel.circle2.State;

public class State3 implements State {
    private Agent agent;
    private CircleSolver solver;

    public State3(CircleSolver parent, Agent agent) {
        this.agent = agent;
        this.solver = parent;
    }

    public Agent getAgent() {
        return agent;
    }

    public String getName() {
        return "3";
    }

    private static void log(String str) {
        // System.out.println(str);
    }

    /**
     * Performance: O(1)
     */
    @Override
    public double getTransitionTime() {
        // The 3rd state is the last state; once an agent gets into this
        // state, it never moves to another state.
        // But...
        // Predictions will change if we reach the next agent's origin.

        log("Agent " + agent.getName() + " is in state 3.");

        Boundary negBoundary = solver.getNegativeBoundary(agent);
        Boundary posBoundary = solver.getPositiveBoundary(agent);

        double groundCoveredTwice = Circle.distanceFrom(agent.getOrigin(),
                negBoundary.getPosition(solver.getTimeElapsed()), Direction.Clockwise);
        double timeLeft = solver.getTimeElapsed() - (groundCoveredTwice*2);
        double groundCoveredOnce = timeLeft;
        double distanceToPosBound = Circle.distanceFrom(agent.getOrigin(),
                posBoundary.getPosition(solver.getTimeElapsed()), Direction.CounterClockwise);
        double remainingGroundToCover = distanceToPosBound - groundCoveredOnce;

        log("Ground covered twice: " + groundCoveredTwice);
        log("Ground covered once: " + groundCoveredOnce);
        log("Ground to cover to positive bound: " + distanceToPosBound);
        log("Remaining ground to cover to positive bound: " + remainingGroundToCover);
        log("Speed of negative boundary: " + negBoundary.getSpeed());
        log("Speed of positive boundary: " + posBoundary.getSpeed());

        double collisionSpeed = 1 + (2*negBoundary.getSpeed()); // - posBoundary.getSpeed();
        log("Collision speed: " + collisionSpeed);

        double distanceToNextAgentOrigin = Circle.distanceFrom(agent.getOrigin(),
                agent.getNext().getOrigin(), Direction.CounterClockwise);
        log("Distance to next agent's origin: " + distanceToNextAgentOrigin);
        double distanceLeftToNextAgentOrigin = distanceToNextAgentOrigin - groundCoveredOnce;
        log("Distance left to reach the next agent's origin: " + distanceLeftToNextAgentOrigin);

        if (distanceLeftToNextAgentOrigin < State.MAX_BOUND_DIFF) {
            log("Agent is grid-locked.");
            return Double.MAX_VALUE;
        }

        double timeToGridLock = distanceLeftToNextAgentOrigin / collisionSpeed;
        log("Time left to reach the next agent's origin: " + timeToGridLock);
        log("Absolute time to reach the next agent's origin: " + (timeToGridLock+solver.getTimeElapsed()));

        return timeToGridLock;
    }

    /**
     * We remain in this state.
     * But we have reached the boundary...
     */
    public State nextState() {
        return this;
    }

    @Override
    public void complete() {
        solver.getPositiveBoundary(getAgent()).setReached(true);
        solver.getNegativeBoundary(getAgent().getNext()).setReached(true);
```

```
}

/**
 * Performance: O(1)
 */
@Override
public void setBoundarySpeeds() {
    Agent previousAgent = getAgent().getPrevious();
    Agent nextAgent = getAgent().getNext();

    double negBoundarySpeed =
            solver.getNegativeBoundary(getAgent()).getSpeed();
    log(agent.getName() + "> Neg boundary speed: " + negBoundarySpeed);

    solver.getPositiveBoundary(previousAgent).setSpeed(0, solver.getTimeElapsed());
    double nextAgentNegativeBoundarySpeed = 1 + (2*negBoundarySpeed);
    if (isGridLocked()) {
        log(agent.getName() + "> Grid-locked. Neg boundary speed: " + negBoundarySpeed);
        nextAgentNegativeBoundarySpeed = 0;
    }
    else {
        log(agent.getName() + "> is not grid-locked.");
    }
    log("Next agent's negative boundary moving at " + nextAgentNegativeBoundarySpeed);

    solver.getNegativeBoundary(nextAgent).setSpeed(nextAgentNegativeBoundarySpeed,
            solver.getTimeElapsed());
}

/**
 * Performance: O(1)
 */
@Override
public boolean isGridLocked() {
    Boundary currentAgentNegBound = solver.getNegativeBoundary(getAgent());
    double distanceCoveredTwice = Circle.distanceFrom(getAgent().getOrigin(),
            currentAgentNegBound.getPosition(solver.getTimeElapsed()),
            Direction.Clockwise);

    log("Distance to negative boundary (from " + getAgent().getOrigin()
            + " to " + currentAgentNegBound.getPosition(solver.getTimeElapsed())
            + ") is " + distanceCoveredTwice);

    double timeLeft = solver.getTimeElapsed() - (2*distanceCoveredTwice);

    double distanceToNextAgentOrigin = Circle.distanceFrom(
            getAgent().getOrigin(),
            getAgent().getNext().getOrigin(),
            Direction.CounterClockwise);

    log("Distance to next agent's origin: " + distanceToNextAgentOrigin);
    log("Time left: " + timeLeft);

    if (timeLeft < 0) {
        throw new IllegalStateException("Time left is: " + timeLeft);
    }

    if (distanceToNextAgentOrigin - timeLeft <= State.MAX_BOUND_DIFF) {
        return true;
    }
    else {
        return false;
    }
}

/**
 * Performance: O(1)
 */
@Override
public double getGroundCovered() {
    Boundary currentAgentNegBound = solver.getNegativeBoundary(getAgent());

    if (isGridLocked()) {
        double entireSpace = Circle.distanceFrom(
                currentAgentNegBound.getPosition(solver.getTimeElapsed()),
                agent.getNext().getOrigin(),
                Direction.CounterClockwise);
        log("Space between " + agent.getPrevious().getName() + " and "
                + agent.getNext().getName() + " is: " + entireSpace);
        return entireSpace;
    }

    double distanceCoveredTwice = Circle.distanceFrom(getAgent().getOrigin(),
            currentAgentNegBound.getPosition(solver.getTimeElapsed()), Direction.Clockwise);
    double timeLeft = solver.getTimeElapsed() - (2*distanceCoveredTwice);
    double distanceCoveredOnce = timeLeft;
```

```
        log("Ground covered to the negative boundary: " + distanceCoveredTwice);
        log("Ground covered in counter-clockwise direction: " + distanceCoveredOnce);
        log("Total ground covered: " + (distanceCoveredOnce + distanceCoveredTwice));

        return distanceCoveredOnce + distanceCoveredTwice;
    }

    public double getGroundCoveredClockwise(CircleSolver solver) {
        Boundary currentAgentNegBound = solver.getNegativeBoundary(getAgent());
        return Circle.distanceFrom(getAgent().getOrigin(),
                currentAgentNegBound.getPosition(solver.getTimeElapsed()), Direction.Clockwise);
    }

    public double getGroundCoveredCounterClockwise(CircleSolver solver) {
        Boundary currentAgentNegBound = solver.getNegativeBoundary(getAgent());
        double distanceCoveredTwice = Circle.distanceFrom(getAgent().getOrigin(),
                currentAgentNegBound.getPosition(solver.getTimeElapsed()), Direction.Clockwise);
        double timeLeft = solver.getTimeElapsed() - (2*distanceCoveredTwice);
        return timeLeft;
    }

    /**
     * Performance: O(1)
     */
    @Override
    public void assertAccurateBoundaries() {
        if (solver.getNegativeBoundary(agent).isReached() == false) {
            throw new IllegalStateException("Cannot be in state 3 if the negative boundary is not reached.");
        }
        if (Circle.absoluteDistanceBetween(
                solver.getNegativeBoundary(agent).getPosition(solver.getTimeElapsed()),
                solver.getPositiveBoundary(agent.getPrevious()).getPosition(solver.getTimeElapsed()))
                        > MAX_BOUND_DIFF) {
            throw new IllegalStateException("Negative boundary is inconsistent.");
        }

        double negBound = solver.getNegativeBoundary(agent).getPosition(solver.getTimeElapsed());
        double origin = agent.getOrigin();
        double distanceCoveredTwice = Circle.distanceFrom(origin, negBound, Direction.Clockwise);
        log("Agent " + agent.getName() + " in state 3; time elapsed: " + solver.getTimeElapsed());
        log("Agent " + agent.getName() + " in state 3; distance covered twice: " + distanceCoveredTwice);
        double timeLeft = solver.getTimeElapsed() - (2 * distanceCoveredTwice);
        log("Agent " + agent.getName() + " in state 3; time left: " + timeLeft);
        double distanceCoveredOnce = timeLeft;
        double expectedPositiveBoundPosition =
                Circle.toPositivePosition(origin + distanceCoveredOnce);
        log("Agent " + agent.getName() + " in state 3; expected positive position is: "
                + expectedPositiveBoundPosition);
        double actualPositiveBoundPosition = solver.getNegativeBoundary(
                agent.getNext()).getPosition(solver.getTimeElapsed());
        log("Agent " + agent.getName() + " in state 3; actual positive position is: "
                + actualPositiveBoundPosition);

        double distanceToNextAgentOrigin = Circle.distanceFrom(agent.getOrigin(),
                agent.getNext().getOrigin(), Direction.CounterClockwise);
        log("Distance to next agent's origin: " + distanceToNextAgentOrigin);
        log("Distance covered once: " + distanceCoveredOnce);
        if (distanceToNextAgentOrigin < distanceCoveredOnce) {
            log("Grid locked!");
            expectedPositiveBoundPosition = agent.getNext().getOrigin();
        }

        double distanceCW = Circle.distanceFrom(actualPositiveBoundPosition,
                expectedPositiveBoundPosition, Direction.Clockwise);
        double distanceCCW = Circle.distanceFrom(actualPositiveBoundPosition,
                expectedPositiveBoundPosition, Direction.CounterClockwise);
        log("Actual boundary:           " + actualPositiveBoundPosition);
        log("Expected boundary:         " + expectedPositiveBoundPosition);
        log("Distance clock-wise:       " + distanceCW);
        log("Distance counter clock-wise: " + distanceCCW);

        if (Circle.areApproximatelyTheSame(actualPositiveBoundPosition, expectedPositiveBoundPosition)) {
            log("Are the same!");
        }
        else {
            log("Are different!");
            throw new IllegalStateException("State 3 positive boundary calculated differs.");
        }
    }
}
```