Chapitre 5

Algorithmique distribuée : systèmes d'agents mobiles

Arnaud Labourel

Un système distribué est un environnement dans lequel plusieurs processus collaborent pour réaliser un objectif commun ; dans un tel système, les différents processus ne peuvent communiquer directement qu'avec un nombre limité d'autres processus. On cherche à déterminer quels sont les comportements globaux qui peuvent être obtenus dans ces systèmes où les actions des processus n'ont qu'un impact local. Le cœur scientifique de l'algorithmique distribuée est donc l'exploration des relations entre local et global.

Dans ce chapitre, on va s'intéresser aux aspects fondamentaux de l'algorithmique distribuée pour les systèmes d'agents mobiles. Dans ces systèmes, les nœuds du réseau sont inactifs et ce sont des processus mobiles (nommés agents) se déplaçant sur le réseau qui ont la charge d'exécuter l'algorithme. On présentera deux problèmes fondamentaux de l'algorithmique distribuée pour ces systèmes : l'exploration et le rendez-vous.

5.1 Préliminaires

Dans cette section, on va considérer des systèmes distribués composés d'agents mobiles se déplaçant au sein d'un réseau. Les algorithmes distribués pour ces systèmes consistent à décrire les actions à effectuer (essentiellement des déplacements et des calculs pour déterminer les prochains déplacements) par les agents au sein d'un réseau. La principale difficulté lors de la conception de tels algorithmes est d'arriver à garantir l'accomplissement d'une tâche globale alors que le comportement de chaque agent est décrit seulement à partir de leur vision locale. Les deux tâches qu'on va considérer dans ce chapitre sont celles de l'exploration (faire visiter à un ou plusieurs agents tous les nœuds du réseau) et celle du rendez-vous (regrouper plusieurs agents initialement dispersés).

Le réseau est représenté par un graphe fini et connexe. Afin que le graphe soit navigable par les agents, le réseau est muni de numéro de ports. Formellement, cela veut dire que chaque demi-arête est étiquetée par un entier et que l'ensemble des demi-arêtes incidentes à un nœud utilise les entiers allant de 0 jusqu'au degré du nœud moins 1 (voir Figure 5.1).

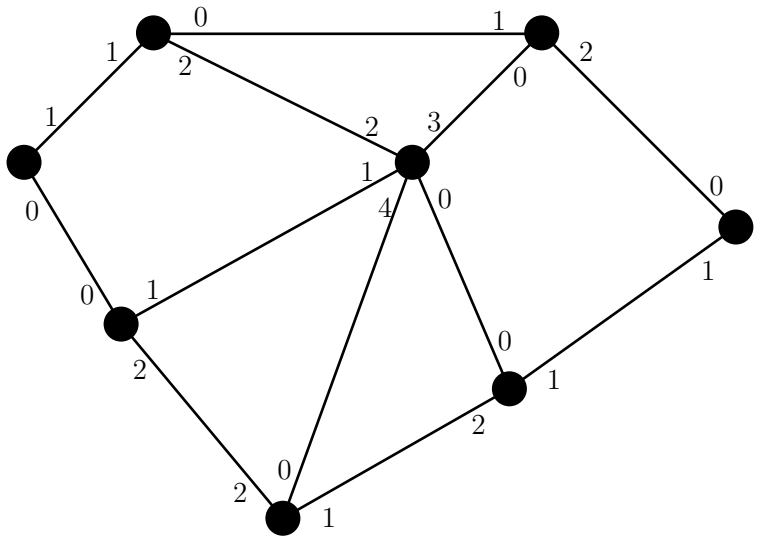

FIGURE *5.1 – Exemple de graphe représentant un réseau.*

Chaque agent mobile commence son exécution sur un des nœuds du graphe et a la possibilité de se déplacer le long des arêtes du graphe. Une étape de calcul pour un agent mobile consiste à calculer son prochain déplacement (numéro de port à emprunter à partir de son nœud courant) et à effectuer le déplacement calculé. Dans cette section, on supposera que chaque agent exécute le même algorithme déterministe.

Un agent mobile n'a généralement qu'une vision locale du graphe, il ne voit donc que les informations présentes sur le nœud courant, le numéro

de port par lequel il est arrivé sur le nœud et l'ensemble des numéros de ports des demi-arêtes incidentes (et donc le degré du nœud courant). L'agent doit calculer de manière déterministe chacun de ses déplacement à partir de ces informations locales plus les informations stockées dans sa mémoire persistante, c'est-à-dire la mémoire conservée après chacun de ses déplacements.

Il existe de nombreuses hypothèses possibles pour les systèmes d'agents mobiles :

— Les nœuds peuvent disposer d'identifiants unique ou non. Un graphe sans identifiant sur ses nœuds sera dit *anonyme*.
— Les nœuds peuvent disposer d'une capacité de stockage. Un agent présent sur un nœud pourra écrire n'importe quelle information sous forme binaire qui sera lisible par n'importe quel agent visitant le nœud.
— L'agent peut commencer l'algorithme avec des informations sur la configuration (borne sur le nombre de nœud du graphe, identifiant de l'agent dans le cas d'un système utilisant plusieurs agents, ...).

On peut mesurer l'efficacité d'algorithmes utilisant des agents mobiles à l'aide des critères de complexité suivants :

— le nombre de mouvements : le nombre total de mouvements effectués par les agents
— la mémoire : taille en nombre de bits de la mémoire persistante de l'agent, c'est-à-dire la quantité de mémoire que l'agent conserve après chaque déplacement
— la capacité de stockage de chaque nœud : taille en nombre de bits des informations que peuvent écrire les agents sur un nœud.

La complexité sera exprimée en fonctions des paramètres classiques de graphes :

— Le nombre de nœuds du graphe noté n
— Le degré maximal d'un nœud du graphe noté Δ
— Le nombre d'arête du graphe noté m.

5.2 Problème de l'exploration

Un des problèmes fondamentaux des systèmes à agents mobiles est l'exploration. En effet, être en mesure d'explorer le graphe, c'est-à-dire de visiter tous les nœuds du graphe, est souvent nécessaire pour accomplir la plupart des tâches importantes pour les agents mobiles comme cartographier le réseau, trouver un nœud avec une propriété particulière, rencontrer

un autre agent, ... Il existe de nombreuses variantes du problème de l'exploration. On ne traitera dans ce chapitre que des trois variantes suivantes :

- sans terminaison : l'agent doit explorer tous les nœuds du graphes.
- avec terminaison : l'agent doit s'arrêter une fois qu'il a visité tous les nœuds.
- perpétuelle : l'agent doit explorer le graphe continuellement et chaque nœud doit être visité infiniment souvent.

Si vous êtes intéressé par le problème d'exploration et souhaitez en savoir davantage, nous vous conseillons de consulter le chapitre dédié à ce problème [1] dans le livre *Distributed Computing by Mobile Entities*.

5.2.1 Méthodes classiques d'exploration de graphes

Parcours en profondeur

Une des techniques les plus basiques pour l'exploration consiste à effectuer un parcours en profondeur tel que décrit par le pseudo-code ci-dessous.

Algorithme 5.1 : Parcours en profondeur pour l'exploration

```
si le nœud courant v est incident à au moins une demi-arête non-explorée
alors
    Se déplacer via la demi-arête inexplorée ayant le plus petit
     numéro de port;
    si le nœud atteint a déjà été visité alors
        Se déplacer vers le nœud précédent v;
sinon
    si le nœud courant v n'est pas le nœud de départ alors
        Se déplacer via l'arête utilisée pour atteindre pour la
         première fois v;
```

Pour que le parcours en profondeur soit réalisable, l'agent doit être capable de :

- différencier les nœuds visités : présence d'identifiant unique sur chaque nœud ou bien une capacité de stockage en $O(\log n)$
- mémoriser les arêtes non-explorées et l'arête utilisée lors la première visite de chaque nœud : mémoire de l'agent en $O(n\Delta)$ ou capacité de stockage des nœuds en Δ.

Si ces conditions sont vérifiées, il est clair que l'algorithme termine après avoir visité tous les nœuds du graphe car il simule un parcours en profondeur du graphe. On peut remarquer que chaque demi-arête

n'est utilisée qu'une fois. La complexité en nombre de mouvements de l'algorithme est donc de $2m$.

Algorithme de la main droite contre le mur

Une autre manière naïve d'explorer un graphe est d'utiliser la règle de la main droite contre le mur. Cette technique intuitive permet de résoudre un labyrinthe simplement connexe [1] et consiste à suivre le mur de droite avec sa main lorsqu'il y a un embranchement. La technique peut donc se résumer ainsi : à chaque embranchement prendre le chemin suivant celui par lequel on est arrivé dans le sens inverse des aiguilles d'une montre. Dans le cas de l'exploration de graphe, l'agent doit donc se déplacer par le numéro de port suivant celui par lequel il est arrivé. La numéro de port suivant i sera $i + 1$ modulo le degré du nœud.

L'algorithme de de la Main Droite contre le Mur (MDM) permet d'explorer perpétuellement les arbres. Afin de démontrer cela, on va considérer l'arbre enraciné dans la position de départ de l'agent et on va le dessiner dans le plan de sorte à ce que l'ordre des numéro de ports soit croissant dans le sens inverse des aiguilles d'une montre (voir figure 5.2 pour un exemple).

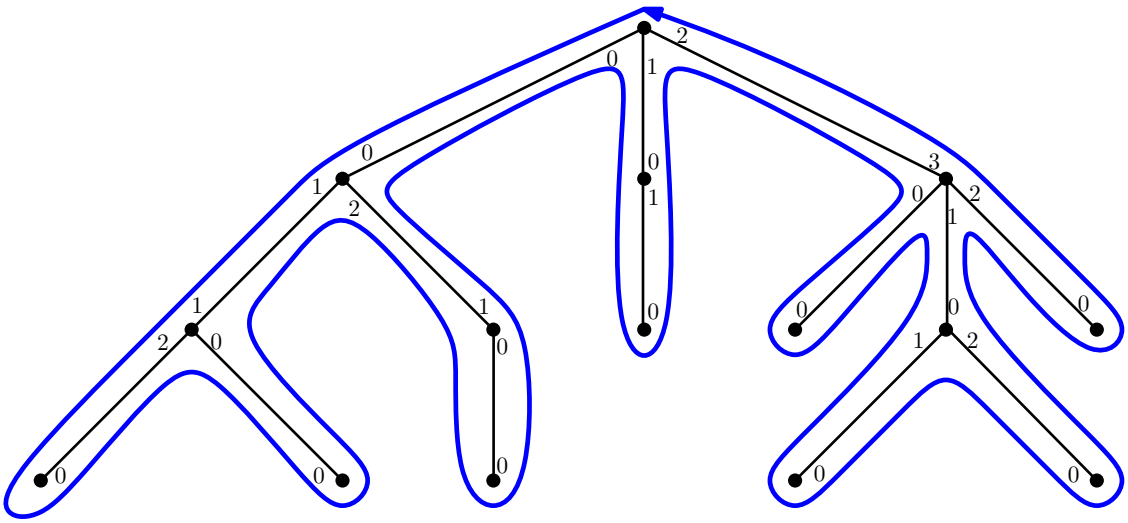

FIGURE 5.2 – *Exemple de parcours de MDM sur un arbre.*

Dans ce cas, l'algorithme MDM va faire un tour de l'arbre et donc explorer tous les nœuds de l'arbre. Cela se prouve facilement par induc-

1. En topologie, la notion de simple connexité raffine celle de connexité : un espace simplement connexe est sans trou ni poignée en plus d'être connexe. Dans notre cas, cela signifie qu'on considère des labyrinthes connexes et sans cycles et donc en termes de graphes des arbres.

tion sur la profondeur des sous-arbres d'un nœud. Pour un sous-arbre de profondeur 1 qui correspond à une feuille, l'agent va aller à la feuille puis revenir. L'agent a donc exploré le sous arbre et est revenu à son point de départ. Pour un sous-arbre de profondeur n, l'agent va aller au fils du nœud correspondant à l'arête pour ensuite explorer les sous-arbres du fils dans l'ordre des numéro de port. Par hypothèse d'induction, ces sous-arbres sont explorés par l'agent qui revient ensuite par la même arête car ces sous-arbres sont de profondeur au plus $n-1$. Une fois que tous les sous-arbres du fils sont explorés, l'agent va remonter par l'arête reliant le père au fils. Le sous-arbre induit par le fils est donc aussi entièrement exploré dans ce cas. Par le principe d'induction, l'algorithme MDM réalise donc l'exploration de l'arbre. Si on n'arrête jamais l'algorithme, l'exploration est perpétuelle car il réalise une infinité de tours de l'arbre. Contrairement à la méthode précédente, il ne nécessite pas d'avoir du stockage sur les nœuds, des identifiants ou bien de la mémoire sur l'agent. Il peut aussi permettre l'exploration avec terminaison des arbres. Pour cela, il lui suffit :

- de connaître une borne k sur le nombre d'arêtes de l'arbre et d'avoir une mémoire en $\log k$, l'exploration se terminant après $2k$ déplacements,
- ou bien assez de mémoire pour se souvenir de la carte de l'arbre,
- ou bien d'être capable de marquer le nœud de départ, l'exploration se terminant lorsqu'il revient sur le nœud de départ via le numéro de port égal au degré moins un.

Algorithme 5.2 : Parcours MDM pour les arbres

Entrée : Une borne k du nombre d'arête de l'arbre
$p :=$ numéro de port d'arrivée (initialement égal à 0);
$d :=$ degré du nœud courant;
Se déplacer via la demi-arête ayant le numéro de port $(p+1)$ mod d;
i := i+1 (initialement égal à 0);
si $i \geqslant 2k$ **alors**
  s'arrêter;

Séquences d'exploration universelles

L'idée derrière l'algorithme MDM peut être généralisée à l'aide de Séquences d'Exploration Universelles (SEU). Intuitivement, l'idée est d'avoir une séquence d'entiers qui va définir la valeur à ajouter au numéro de port d'arrivée pour obtenir le numéro de port de départ (au lieu de la valeur 1 ajoutée dans tous les cas par l'algorithme MDM). Soit $S = (s_1, s_2, \ldots, s_t)$

une séquence finie de t entiers. Pour tout nœud u d'un graphe G, on définit le chemin P_u suivant $(v_0, e_1, v_1, e_2, v_2 \dots e_t, v_t)$ tel que :

— $v_0 = u$
— Pour tout $1 \leqslant i \leqslant t$, l'arête $e_i = \{v_{i-1}, v_i\}$ correspond à la demi-arête incidente à v_{i-1} ayant le numéro de port $p = (a + s_i) \mod \Delta(v_i)$ avec a le numéro de port de l'arête e_{i-1} dans v_i ($a = 0$ pour $i = 0$) et $\Delta(v_i)$ le degré de v_i.

La séquence S est une SEU pour le graphe G si pour tout nœud u de G, le chemin P_u contient tous les nœuds de G. Une séquence S est une *SEU* pour une famille de graphes si c'est une SEU pour tous les graphes de la famille.

Théorème 5.2.1 (Reingold [5])**.** *Il existe une SEU de longueur polynomiale en n pour la famille des graphes connexes d'au plus n nœuds. Cette séquence peut être générée avec une mémoire en $O(\log(n))$.*

Le résultat de Reingold nous permet d'obtenir un algorithme d'exploration qu'on nommera RUXS. Le pseudo-code de l'algorithme est quasi-identique à celui de MDM sauf que maintenant on se déplace via le numéro de port $(p + s_i) \mod d$ avec $S = (s_1, s_2, \dots,)$ obtenue en concaténant des SEU pour les graphes de toutes les tailles (par ordre croissant).

— L'algorithme RUXS permet d'explorer les graphes anonymes sans capacité de stockage sur les nœuds.
— Il est nécessaire de connaître la taille du graphe (ou une borne sur celle-ci) pour avoir la terminaison.
— L'algorithme nécessite un mémoire en $\log(n)$.

5.2.2 Explorations efficaces en temps

N'importe quel algorithme d'exploration doit visiter toutes les arêtes du graphe. La complexité en temps de l'algorithme de parcours en profondeur est donc asymptotiquement optimale (en $2m$ alors que la borne inférieure est en m). Il est possible d'améliorer cette borne pour les graphes ayant $\Omega(n)$ arêtes avec un algorithme ayant une complexité en temps en $m + O(n)$ dans le pire des cas. Cet algorithme est une variante optimisée de l'algorithme de parcours en profondeur dans laquelle le nombre de retours en arrière de l'agent est diminué.

Comme indiqué préalablement, l'algorithme de parcours en profondeur fonctionne dans le cas où les nœuds du graphes ont des identifiants. Dans ce cas, une mémoire pour l'agent en $O(n \log n)$ permet de se souvenir de tous les nœuds visités et aucune capacité de stockage sur les nœud n'est nécessaire.

Dans le cas de graphes sans identifiants sur les nœuds, il est possible de faire un parcours en profondeur si chaque nœud peut être marqué comme visité (capacité de stockage en $O(1)$). Dans ce cas, l'agent peut réaliser le parcours en profondeur en gardant en mémoire à chaque étape l'arbre engendré par le parcours avec les numéros de port des arêtes. Un tel arbre peut être stocké avec une mémoire en $O(n \log n)$ ce qui nous donne le résultat suivant basé sur l'algorithme classique de parcours en profondeur [7].

Théorème 5.2.2. *Il existe un algorithme d'exploration avec terminaison pour les graphes anonymes (sans identifiants sur les nœuds) utilisant $O(1)$ bits de stockage par nœud et une mémoire de $O(n \log n)$ pour l'agent.*

5.2.3 Explorations efficaces en stockage

Comme nous l'avons vu précédemment, dans le cas d'un graphe avec identifiants sur les nœuds, il est possible de réaliser l'exploration sans capacité de stockage sur les nœuds. Dans le cas d'un graphe anonyme, l'exploration est possible avec une capacité de stockage en $O(1)$. Le théorème suivant nous indique qu'une capacité de stockage est nécessaire pour réaliser l'exploration avec terminaison dans ce cas.

Théorème 5.2.3. *Il n'existe pas d'algorithme d'exploration avec terminaison pour les graphes anonymes sans capacité de stockage sur les nœuds.*

Démonstration. On considère deux cycles C_1 et C_2 ayant respectivement $n_1 = 3$ et n_2 nœuds. Chaque arête de ces deux cycles sera numérotées avec les numéros de ports 1 et 0. Le numéro de port 0 correspond à un mouvement dans le sens inverse des aiguilles d'une montre alors que le numéro de port 1 correspond à un mouvement dans le sens des aiguilles d'une montre. On peut remarquer que chacun des nœuds des graphes sont vus comme identiques par un agent. Supposons qu'il existe un algorithme d'exploration avec terminaison. On considère l'exécution de cet algorithme sur le cycle C_1. L'algorithme termine après un certain nombre de déplacements de l'agent qu'on notera t. Fixons maintenant la taille de C_2 à $2t + 2$. Il est impossible pour un agent mobile de différencier les nœuds de C_1 des nœuds de C_2, l'exécution de l'algorithme sur C_2 s'arrêtera donc après t déplacements de l'agent. Le nœud à distance $t + 1$ du nœud de départ dans C_2 ne sera donc pas visité. On obtient donc une contradiction et il n'existe pas d'algorithme d'exploration avec terminaison. ■

On peut observer que l'impossibilité de l'exploration vient du fait que l'agent n'a aucune connaissance préalable sur le graphe à explorer. Si

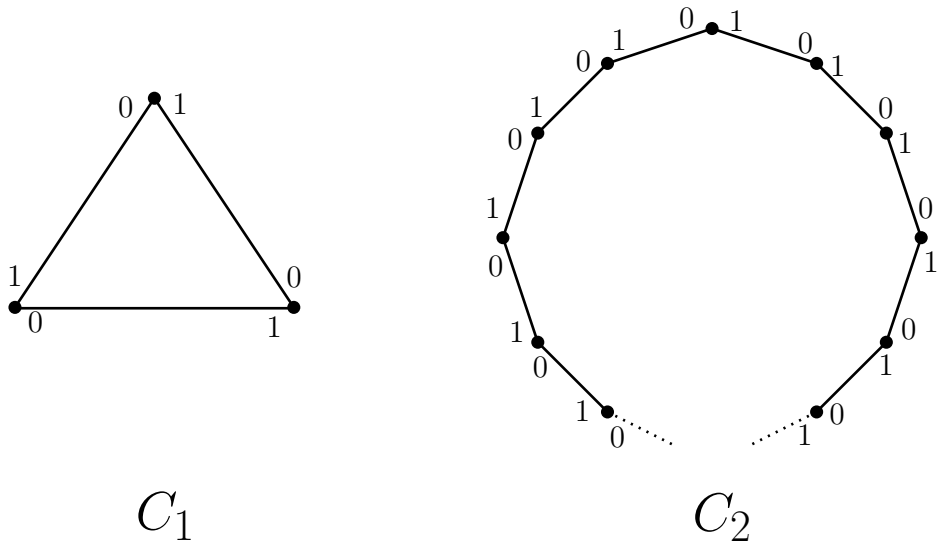

FIGURE 5.3 – *Cycles C_1 et C_2.*

l'agent connaît la taille du graphe, il est possible d'utiliser l'algorithme RUXS pour obtenir le résultat suivant.

Théorème 5.2.4. *Il existe un algorithme d'exploration avec terminaison pour les graphes anonymes utilisant* 0 *bits de stockage par nœud si l'agent connaît la taille n du graphe. Cet algorithme a une complexité polynomiale.*

5.3 Problème du rendez-vous

Un algorithme de rendez-vous permet de regrouper des agents mobiles initialement dispersés dans un réseau sur un unique nœud du réseau. Le problème du rendez-vous est très étudié pour les systèmes à agents mobiles puisqu'il nécessite de briser totalement les symétries initiales. De plus, de nombreuses tâches distribuées requièrent des agents qu'ils se coordonnent et donc se rencontrent car les communications se font de manière locale. Le rendez-vous est donc très souvent une tâche préalable nécessaire à l'accomplissement de nombreuses tâches distribuées.

Dans cette partie, nous ne traiterons que des algorithmes de rendez-vous synchrones. Si vous êtes intéressé par le problème du rendez-vous et souhaitez en savoir davantage sur les différents hypothèses possibles, nous vous conseillons de consulter l'article d'Andrzej Pelc qui donne une vue d'ensemble sur le domaine [4].

5.3.1 Préliminaires

On considérera uniquement dans cette partie le rendez-vous de deux agents mobiles. Deux agents seront donc présents dans un même graphe et auront pour tâche de se rencontrer : c'est-à-dire être sur le même nœud au même moment.

On considérera les hypothèses suivantes :

— Graphe anonyme : il n'y a pas d'identifiants sur les nœuds.

— Agents munis d'identifiants : les agents possèdent chacun un identifiant id unique sous la forme d'un entier. Chaque agent connaît son identifiant mais pas celui de l'autre agent. Par la suite on notera id_1 et id_2 les deux identifiants des agents.

— Système synchrone : l'exécution de l'algorithme par les agents est découpées en étapes se déroulant en même temps pour tous les agents. À chaque ronde chacun des agents choisit d'attendre ou bien de se déplacer par un numéro de port et tous les agents se déplacent (ou pas si l'agent a choisi d'attendre) en même temps.

— Aucun moyen de communication entre agents sauf s'ils se trouvent sur le même nœud. Les agents ne peuvent donc pas communiquer avant qu'ils aient réalisé le rendez-vous.

La première hypothèse permet de séparer le problème du rendez-vous du problème d'exploration. En effet dans un graphe avec identifiants sur les nœuds, il suffit que les agents explorent le graphe et attendent sur le nœud ayant le plus petit identifiant pour réaliser le rendez-vous.

La deuxième hypothèse permet de rendre le rendez-vous possible dans les cycles. En effet, si les deux agents n'ont pas d'identifiants, il doivent exécuter le même algorithme déterministe. Si on place les deux agents à distance $t + 1$ dans le cycle C_2 de la démonstration du théorème 5.2.3, ils vont se déplacer de la même manière pour n'importe quel algorithme déterministe et rester à tout moment à distance $t + 1$. Le rendez-vous est donc impossible dans ce cadre.

La quatrième hypothèse permet aussi de séparer le problème du rendez-vous du problème d'exploration. En effet, si les agents pouvaient communiquer, ils pourraient échanger leurs identifiants. L'agent ayant le plus petit identifiant attendrait sur son nœud de départ pendant que l'agent ayant le plus grand identifiant explorerait le graphe jusqu'à trouver l'autre agent.

5.3.2 Rendez-vous synchrone dans un cycle

Dans cette partie, on va considérer le rendez-vous dans un cycle. Par soucis de simplicité on considérera que la numérotation des ports est cohérente : le numéro de port 0 correspondant à un mouvement dans le sens inverse des aiguilles d'une montre (appelé gauche) alors que le numéro de port 1 correspond à un mouvement dans le sens des aiguilles d'une montre (appelé droite).

Il est facile de produire des algorithmes naïfs de rendez-vous dans un cycle. Si les agents connaissent la taille du cycle, on peut utiliser l'algorithme suivant : chaque agent fait id tours de l'anneau ($id \times n$ déplacement vers la droite) et s'arrête. Les deux agents se rencontre forcément car un des deux fait au minimum un tour de plus de l'anneau. La complexité en termes de nombre de déplacement de cet algorithme est en $O(n \min\{id_1, id_2\})$. Un autre algorithme naïf consisterait à utiliser la possibilité pour les agents de se déplacer à des vitesses différentes. Chaque agent se déplace vers la droite à vitesse $\frac{1}{id}$, c'est-à-dire se déplace puis attend $id - 1$ étapes. L'agent ayant le plus petit identifiant sera plus rapide et rattrapera donc l'autre agent.

Il est possible de concevoir un algorithme bien plus efficace que ces algorithmes naïfs.

Théorème 5.3.1 (Dessmark, Fraigniaud et Pelc [2])**.** *Il existe un algorithme de rendez-vous déterministe pour les cycles en temps $O(D \log(\min\{id_1, id_2\}))$ où D est la distance initiale entre les deux nœuds de départ des agents.*

Démonstration. L'algorithme de rendez-vous utilise deux algorithmes correspondant à deux cas différents :

- un algorithme Identifiants Proches (noté IP) dans le cas où les identifiants ont des valeurs proches : $\lfloor \log \log id_1 \rfloor = \lfloor \log \log id_2 \rfloor$
- un algorithme Identifiants Différents (noté ID) dans le cas où les identifiants ont des valeurs très différentes : $\lfloor \log \log id_1 \rfloor \neq \lfloor \log \log id_2 \rfloor$.

L'algorithme IP utilise un mot binaire id^* construit en prenant la représentation de l'identifiant id en binaire à laquelle on rajoute des 0 pour obtenir un mot de longueur égale à une puissance de 2. Par exemple, pour un identifiant égal à 15 on obtient le mot 1111 et pour 16 on obtient

00010000. On peut remarquer que la longueur $|id^*|$ du mot ainsi obtenu est égal à $2^{\lfloor \log\log id \rfloor + 1}$.

Algorithme 5.3 : Rendez-vous avec Identifiants Proches (IP)

```
pour i de 1 à +∞ faire
    pour j de 1 à |id*| faire
        si id*[j] = 1 alors
            se déplacer de 2^i pas vers la droite;
            se déplacer de 2^{i+1} pas vers la gauche;
            se déplacer de 2^i pas vers la droite;
        sinon
            attendre pendant 2^{i+2} étapes;
```

Pour utiliser l'algorithme IP, on suppose que $\lfloor \log\log id_1 \rfloor$ égale $\lfloor \log\log id_2 \rfloor$, et donc $|id_1^*| = |id_2^*|$. Puisqu'on est dans un modèle synchrone, les valeurs des indices i et j sont les mêmes pour les deux agents à tout moment. Puisque les identifiants sont différents, il existe un k tel que le k-ième bit de id_1^* est différent de celui de id_2^*. Pour $i = \lceil \log D \rceil$ et $j = k$, les deux agents se rencontrent car :

— un agent (celui ayant le bit à 1) atteint la position de départ de l'autre agent car il visite tous les nœuds à distance au plus $2^{\lceil \log D \rceil}$

— l'autre agent (celui ayant le bit à 0) ne bouge pas et reste sur sa position de départ.

La complexité de l'algorithme IP est en $O(D \log(\min\{id_1, id_2\}))$.

L'algorithme ID utilise un entier id^+ défini par :

$$id^+ = \begin{cases} 1 & \text{si } id = 1 \\ \lfloor \log\log id \rfloor + 2 & \text{autrement .} \end{cases}$$

Le pseudo-code

Algorithme 5.4 : Rendez-vous avec Identifiants Différents (ID)

```
pour i de 1 à +∞ faire
    pour j de 1 à i faire
        si j = i − id+ alors
            se déplacer de 2^j pas vers la droite;
            se déplacer de 2^{j+1} pas vers la gauche;
            se déplacer de 2^j pas vers la droite;
        sinon
            attendre pendant 2^{j+2} étapes;
```

Pour utiliser l'algorithme ID, on suppose que $\lfloor \log\log id_1 \rfloor \neq \lfloor \log\log id_2 \rfloor$. On peut supposer sans perte de généralités que $id_1^+ < id_2^+$. Durant la i-ème itération de la boucle, l'agent 1 (celui ayant l'identifiant id_1^+) parcourt tous les nœuds de l'anneau à distance $\leqslant 2^{i-id_1^+}$. Pour le plus petit s tel que $2^{s-id_1^+} \geqslant D$, l'agent 1 rencontre donc l'autre agent puisque celui-ci attend sur sa position de départ car $id_1^+ \neq id_2^+$. La complexité de l'algorithme ID est en $O(2^s) = O(2^{id_1^+ + \log D}) = O(D\log(\min\{id_1, id_2\}))$.

Maintenant il ne reste plus qu'à combiner les deux algorithmes en alternant leurs exécutions grâce à l'algorithme suivant.

Algorithme 5.5 : Rendez-vous dans les cycles

pour *i de 1 à* $+\infty$ **faire**
- Exécuter l'algorithme IP pendant 2^i étapes;
- Revenir sur la position de départ en t étapes;
- Attendre $2^i - t$ étapes;
- Exécuter l'algorithme ID pendant 2^i étapes;
- Revenir sur la position de départ en t étapes;
- Attendre $2^i - t$ étapes

Puisque le système est synchrone, les agents exécutent le même algorithme au même moment. Au moins un des deux algorithmes va réaliser le rendez-vous avec $c = O(D\log(\min\{id_1, id_2\}))$ déplacements. Les agents se rencontreront donc à l'itération numéro $i = \lceil \log c \rceil$ de la boucle. L'itération i a un coût inférieur ou égal à 4×2^i et la somme des coûts des itérations de 1 à i est inférieure ou égale à 8×2^i. Le coût total de l'algorithme est donc en $O(2^i) = O(D\log(\min\{id_1, id_2\}))$. ■

Cet algorithme de rendez-vous dans l'anneau est optimal à un facteur multiplicatif constant près.

Théorème 5.3.2 (Dessmark, Fraigniaud et Pelc [2])**.** *Tout algorithme de rendez-vous déterministe a une complexité en temps de* $\Omega(D\log(\min\{id_1, id_2\}))$ *dans l'anneau où* D *est la distance initiale entre les deux nœuds de départ des agents.*

Démonstration. Supposons par l'absurde qu'il existe un algorithme résolvant le rendez-vous pour $Dy/4$ mouvements avec $y \in \mathbb{N}$ pour tous identifiants $id_1, id_2 \leqslant 2^y$. Soit D un entier, on considère un cycle C de $kD/2$ nœuds avec k un entier quelconque. On partitionne le cycle C en k composantes connexes (appelées *tranches*) de $D/2$ nœuds. Pour tout identifiant id, on exécute l'algorithme pour un agent seul d'identifiant id dans le cycle. On observe la position de l'agent toutes les $D/2$ étapes et on construit lettre à lettre un mot de comportement dans l'alphabet $\{1, 0, -1\}$ en choisissant chaque lettre de la manière suivante :

— 0 : si l'agent est restée dans la même tranche
— 1 : s'il s'est déplacé dans la tranche de droite
— -1 : s'il s'est déplacé la tranche de gauche.

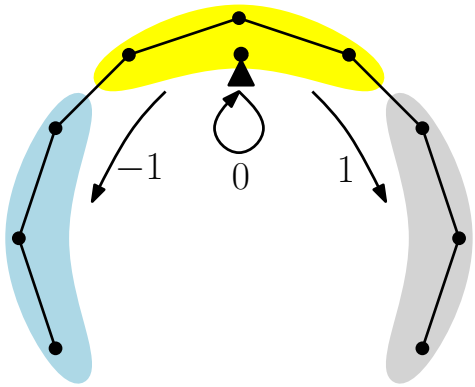

FIGURE 5.4 – *Code de comportement.*

Il y a $3^{y/2} < 2^y$ codes de comportement de longueur $y/2$. On peut trouver deux identifiants $id_1, id_2 \leqslant 2^y$ qui ont le même code de comportement de longueur $y/2$. Ces deux agents ne se rencontrent pas pendant leur $Dy/4$ premiers déplacements. On obtient donc une contradiction et un algorithme résolvant le rendez-vous avec une complexité en $Dy/4 = D\log(\min\{id_1, id_2\})/4$ n'existe pas. ■

5.3.3 Rendez-vous synchrone dans un arbre

Une autre classe de graphes important pour le problème de rendez-vous est la classe des arbres.

Théorème 5.3.3 (Dessmark, Fraigniaud, Pelc [2])**.** *Il existe un algorithme de rendez-vous synchrone dans les arbres avec une complexité en temps en* $O(n + \log(\min\{id_1, id_2\}))$.

Démonstration. L'idée de l'algorithme est d'explorer l'arbre afin de trouver le ou les centres de l'arbre. Un centre d'un arbre est un nœud qui minimise l'*excentricité*, c'est-à-dire la plus grande distance à un nœud de l'arbre. On va utiliser une propriété bien connue [3] qui est que pour tout arbre il existe :

— un centre unique qu'on appellera nœud central,
— ou bien deux centres adjacents qui forme une arête centrale.

On ramène donc le problème du rendez-vous dans les arbres au problème de rendez-vous dans l'arête grâce au pseudo-code suivant.

Algorithme 5.6 : Rendez-vous dans les arbres

```
Explorer l'arbre à l'aide de l'algorithme de parcours MDM;
si l'arbre possède un nœud central alors
    s'arrêter sur ce nœud;
sinon
    exécuter RV-arête sur l'arête centrale de l'arbre
```

Les deux agents ne vont pas nécessairement arriver sur l'arête au même moment. Il nous faut donc concevoir un algorithme fonctionnant même si un agent arrive en retard sur l'arête. Il nous est donc impossible d'utiliser l'algorithme de rendez-vous sur l'anneau du théorème 5.3.1 qui nécessite que les agents commencent l'exécution de l'algorithme au même moment. On va réaliser le rendez-vous dans une arête grâce au pseudo-code suivant.

Algorithme 5.7 : RV-arête pour le rendez-vous dans une arête

```
pour j de 1 à +∞ faire
    traverser l'arête;
    attendre une étape;
    pour i de 0 à ⌊log id⌋ − 1 faire
        si id[i] = 0 alors
            traverser l'arête deux fois;
        sinon
            attendre deux étapes;
```

Il nous reste donc à prouver que l'algorithme RV-arête est correct et a une complexité en $O(\log(\min\{id_1, id_2\}))$. L'algorithme doit fonctionner même si un des agents arrive après l'autre sur l'arête. On va prouver qu'à partir du moment où les deux agents sont présents sur l'arête et exécutent l'algorithme, le rendez-vous se produit en au plus $\log(\min\{id_1, id_2\}) + 6$ étapes. On note A_1 le premier agent arrivé sur l'arête et A_2 le deuxième agent. On note τ nombre d'étapes entre les arrivées des deux agents. On dit que l'agent exécute un mot binaire w si à l'étape i il se déplace si et seulement si $w[i] = 1$ (et donc $w[i] = 0$ s'il attend). L'algorithme prend l'identifiant de l'agent et en déduit une période en doublant chaque bit et en ajoutant 10 au début. Ensuite, le comportement de l'agent est obtenu en répétant à l'infini la période. On est obligé de considérer un comportement infini car l'autre agent peut arriver après n'importe quel nombre d'étapes. Par exemple, si l'agent a pour identifiant 101, sa période sera 10110011

et son comportement sera 10110011 10110011 10110011 ... On peut remarquer que la période correspondant à un identifiant id est de longueur $2\lfloor \log id \rfloor + 4$. L'idée est que les deux agents se rencontrent dès que leurs comportements diffèrent car si l'un bouge alors que l'autre attend, ils se rencontrent forcément. Le comportement 10 au début d'une période est là pour garantir que si les comportements ne sont pas synchronisés, c'est-à-dire que les périodes ne commencent pas au même moment, alors les agents se rencontrent forcément. Si les comportements sont synchronisés alors le rendez-vous se fera sur la partie du comportement correspondant à des bits différents dans les identifiants des deux agents.

Plus formellement, on fixe l'étape 0 comment étant la première étape pour laquelle les deux agents sont présents et on va considérer 4 cas suivant les valeur de τ (le nombre d'étapes entre l'arrivée de l'agent A_1 et celle de A_2) :

- **Cas 1 : τ est impair**
  L'agent A_2 exécute les bits 10 pour les deux premières étapes. L'agent A_1 doit donc exécuter 10 en même temps pour éviter le rendez-vous. L'agent A_1 reste donc dans ce cas immobile pour la troisième étape car il doit exécuter 1100 entre les étapes -1 et +2 comprises puisqu'il ne pouvait pas être au début de sa période. Le rendez-vous se produit à la troisième étape car l'identifiant de A_2 commence forcément par un 1 et il doit donc se déplacer car une période commence toujours par 1011.
- **Cas 2 : τ est pair et n'est pas divisible par** $2\lfloor \log id_1 \rfloor + 4$
  Dans ce cas, les comportements des deux agents ne sont pas synchronisée. L'agent A_2 exécute les bits 10 pour les deux premières étapes. L'agent A_1 exécute 11 ou 00 pour les deux premières étapes car ce n'est pas le début de sa période. Dans tous les cas, le rendez-vous se produit donc durant les deux premières étapes.
- **Cas 3 : τ est divisible par** $2\lfloor \log id_1 \rfloor + 4$ **et** $\lfloor \log id_1 \rfloor = \lfloor \log id_2 \rfloor$
  Dans ce cas les comportements des deux agents sont synchronisées. Les étiquettes de A_1 et A_2 diffèrent sur un bit de numéro b. À l'étape $2b + 1$ correspondant au premier bit de comportement du bit numéro b, les deux agents ont un comportement différent et le rendez-vous se produit.
- **Cas 4 : τ est divisible par** $2\lfloor \log id_1 \rfloor + 4$ **et** $\lfloor \log id_1 \rfloor \neq \lfloor \log id_2 \rfloor$
  Soit l le plus petit des identifiants des agents. L'agent ayant l'identifiant l a un comportement différent aux étapes $2\lfloor \log l \rfloor + 5$ et $2\lfloor \log l \rfloor + 6$ puis qu'il exécute le début de sa prochaine période. L'autre agent a le même comportement (11 ou 00) car il est encore

en train d'exécuter sa période. Le rendez-vous se produit donc au plus tard à l'étape $2\lfloor \log l \rfloor + 6$. ■

5.3.4 Rendez-vous synchrone dans les graphes généraux

En ce qui concerne le problème du rendez-vous, il y a peu de résultats connus pour des classes de graphes autres que les arbres et les cycles. Pour les graphes généraux, le meilleur résultat connu s'appuie sur les séquences UXS qu'on a vu précédemment pour le problème de l'exploration.

Théorème 5.3.4 (Ta-Shma et Zwick [6])**.** *Il existe un algorithme de rendez-vous synchrone pour tous les graphes ayant comme complexité* $\widetilde{O}(\Delta^2.n^3.\log l)$ [2] *avec* l *étant le plus petit des deux identifiants des agents.*

La meilleure borne inférieure connue est en $\Omega(n^2)$ [2] et savoir s'il est possible d'obtenir des bornes plus serrées reste un problème ouvert.

Bibliographie

[1] S. DAS : Graph explorations with mobile agents. *In* P. FLOCCHINI, G. PRENCIPE et N. SANTORO, éds : *Distributed Computing by Mobile Entities*. Springer, 2019.

[2] A. DESSMARK, P. FRAIGNIAUD, D. R. KOWALSKI et A. PELC : Deterministic rendezvous in graphs. *Algorithmica*, 46(1):69–96, 2006.

[3] C. JORDAN : Sur les assemblages de lignes. *J. Reine Angew. Math*, 70(185):81, 1869.

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

[5] O. REINGOLD : Undirected connectivity in log-space. *J. ACM*, 55(4):17, 2008.

[6] A. TA-SHMA et U. ZWICK : Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. *ACM Trans. Algorithms*, 10(3):12 :1–12 :15, mai 2014.

[7] R. TARJAN : Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

2. La notation $\widetilde{O}$ indique un comportement asymptotique à un polylogarithme près (exemple : $n(\log n)^2 = \widetilde{O}(n)$).

Table des matières