Faculté
des Sciences
Département des Sciences Mathématiques
Département d'Informatique

UMONS
Université de Mons

Objectif d'atteignabilité et équilibres de Nash dans les jeux sur graphe

Projet de Master
Réalisé par Aline GOEMINNE

Directeurs: Brihaye Thomas
Mélot Hadrien
Services : Service de Mathématiques effectives
Service d'Algorithmique

Année académique 2015–2016

Remerciements

Tout d'abord, je tiens à commencer ce document en remerciant toutes les personnes ayant contribué d'une quelconque manière à l'aboutissement de ce projet.

Je remercie Thomas Brihaye pour m'avoir permis de découvrir et d'apprécier ce domaine qu'est la théorie des jeux ainsi que pour sa bienveillance et ses encouragements constants. Je remercie Hadrien Mélot grâce à qui j'ai pu, au travers de ce projet, éveiller ma curiosité quant aux connections possibles entre deux disciplines qui me tiennent à coeur - les mathématiques et l'informatique. Je les remercie conjointement pour leur disponibilité ainsi que pour tous les bons conseils prodigués. Enfin, je remercie Véronique Bruyère pour ses remarques et conseils de rédaction ainsi que pour avoir patiemment répondu à mes questions.

A mes amis, aux fidèles du « salon bleu » , à tous ceux avec qui j'ai partagé mes doutes mais également des moments bien plus ludiques, je leur dis merci. Je remercie également ma famille, elle qui croit toujours en moi, pour m'avoir supportée et encouragée tout au long de la réalisation de ce projet. Je réserve un merci tout particulier à mes neveux, Antoine et Thomas, qui, sans le savoir, me rappellent constamment la valeur des joies quotidiennes.

A toutes ces personnes qui par leurs mots, attentions ou conseils m'ont permis de garder le cap, encore une fois merci.

Table des matières

1 Introduction

La théorie des jeux est une discipline mathématique visant à modéliser des situations où différents agents interagissent de manière stratégique. Elle rencontre de nombreuses applications dans divers domaines tels que l'économie, l'informatique et la biologie. A une situation, on associe un *jeu* pour lequel les *joueurs* représentent les agents et on s'interroge sur la meilleure façon dont doit agir chaque joueur, c'est-à-dire sur la *stratégie* qu'ils adoptent.

Chaque agent est indépendant et vise à atteindre son propre objectif. La théorie s'est d'abord penchée sur le cas des jeux à deux joueurs à *objectif qualitatif*. Pour ce type de jeux, un joueur tente d'atteindre son objectif alors que l'autre veut l'en empêcher. Cette représentation est utilisée pour modéliser la situation où un contrôleur lutte contre l'environnement pour parvenir à son but. Cette situation a ensuite été étendue à des *objectifs quantitatifs* où le contrôleur essaie de maximiser son gain tandis que l'environnement tente de le minimiser. Dans cette situation, on veut déterminer quelle est la *stratégie optimale* que doit choisir chaque joueur.

Cette modélisation n'est pas suffisante pour des situations plus complexes, avec plus de deux intervenants. C'est pourquoi, il est intéressant d'étudier les jeux multijoueurs à somme non nulle. Dans ce contexte, nous nous intéressons au concept de solution que sont les *équilibres de Nash*. Un équilibre de Nash est un profil de stratégies - *i.e.,* la donnée d'une stratégie pour chaque joueur - tel qu'aucun joueur n'a intérêt à dévier de sa stratégie si les autres joueurs se conforment à celle qui leur correspond dans ce profil. Nous sommes donc particulièrement intéressés par l'étude des équilibres de Nash pour des jeux multijoueurs à somme non nulle avec des objectifs quantitatifs que l'on peut représenter grâce à l'utilisation de *graphes* et où chaque joueur joue tour à tour.

Dans le cadre de notre travail, l'objectif quantitatif qui nous intéresse est celui d'*atteignabilité* : chaque joueur tente de minimiser le coût du chemin dans le graphe qui lui permet d'atteindre son objectif. Dans leur article [1], Brihaye *et al.* démontrent qu'il existe toujours un équilibre de Nash dans un tel jeu. Toutefois, quand il existe plusieurs équilibres de Nash, nous sommes intéressés par celui qui est le plus pertinent. Nous entendons par équilibre pertinent celui pour lequel un maximum de joueurs atteignent leur objectif et tel que la somme des coûts de chaque joueur ayant atteint son objectif est minimale.

Le but de notre travail est d'expliciter et d'implémenter un processus algorithmique rapide pour trouver un équilibre de Nash pertinent. Parcourir tous les chemins possibles du graphe, en extraire tous les équilibres de Nash

et puis seulement sélectionner le plus pertinent n'est pas une approche exploitable car elle est d'une complexité exponentielle. Nous avons donc testé plusieurs approches afin de guider notre exploration du graphe.

Notre travail se présente de la manière suivante : nous commençons par expliquer certaines notions générales de la théorie des jeux dont nous aurons besoin, nous nous attarderons sur la description des jeux à somme nulle et sur les jeux multijoueurs à somme non nulle (sections [2–3]). Ensuite, nous expliquerons ce qu'est, pour nous, un équilibre de Nash pertinent et nous énoncerons une propriété qui nous permettra de tester si un certain *outcome* du jeu (un chemin infini dans le graphe du jeu) correspond à celui d'un équilibre de Nash (section 4). Nous explicitons ensuite, quels procédés nous avons testés afin d'arriver à notre objectif . Nous terminons en expliquant l'implémentation effectuée de ces procédés ainsi qu'en exposant quelques résultats obtenus (section 5).

2 Concepts fondamentaux de théorie des jeux

Dans le cadre de notre projet, nous sommes intéressés par les jeux sur graphe où tous les joueurs ont un *objectif d'atteignabilité*. Dans cette section, nous abordons la notion de jeux sur graphe de manière générale ainsi que les concepts fondamentaux liés à la théorie des jeux.

Cette section est essentiellement inspirée de l'article de Brihaye *et al.* [1] ainsi que de la thèse de Julie De Pril [3].

2.1 Jeux sur graphe

Définition 2.1 (Arène)**.** Soit Π un ensemble (fini) de joueurs. On appelle *arène* le tuple suivant :
$\mathcal{A} = (\Pi, V, (V_i)_{i\in\Pi}, E)$ où :

- $G = (V, E)$ est un *graphe orienté* dont V est l'ensemble (fini) de ses sommets (*vertices*) et $E \subseteq V \times V$ est l'ensemble de ses arcs (*edges*). De plus, pour tout $v \in V$ il existe $v' \in V$ tel que $(v, v') \in E$ (*i.e.,* pour tout sommet dans le graphe, il existe un arc sortant de ce sommet).
- $(V_i)_{i\in\Pi}$ est une partition de l'ensemble des sommets du graphe G telle que V_i est l'ensemble des sommets du joueur i.

Définition 2.2 (Jeu sur graphe)**.** Un *jeu sur graphe*, noté $\mathcal{G}$ est la donnée d'une arène $\mathcal{A}$ et d'un *objectif* pour chaque joueur.

Dans le cadre de ce document, les objectifs sur lesquels nous portons notre attention sont les *objectifs d'atteignabilité*. Cette notion est abordée plus amplement dans la section 3.

Définition 2.3 (Jeu initialisé)**.** Dans certains types de jeux sur graphe, on considère que le jeu commence à partir d'un *sommet initial* donné. On le note communément v_0. Soit $\mathcal{G}$ un tel jeu, on note alors $(\mathcal{G}, v_0)$ le jeu ayant pour sommet initial v_0. On appelle $(\mathcal{G}, v_0)$ un *jeu initialisé*.

Notations. Tout au long de ce rapport nous utilisons les conventions suivantes :

- Soit $\mathcal{G}$ un jeu sur graphe, on note $(\mathcal{G}, v_0)$ le jeu ayant v_0 comme sommet initial .
- On note J_i le joueur i.
- Pour $i \in \Pi$, on note : $-i \equiv \Pi\backslash\{i\}$.

Nous pouvons maintenant expliquer comment se déroule une partie dans un jeu sur graphe.

Déroulement d'une partie

Nous pouvons imaginer le déroulement d'une partie d'un jeu sur graphe de la manière suivante : pour commencer un jeton est positionné sur un sommet v_0 du graphe (le sommet initial de la définition 2.3). Ensuite, comme ce sommet appartient à un certain V_i, le joueur i choisit une arête $(v_0, v_1) \in E$ et fait « glisser » le jeton le long de l'arc vers le sommet v_1. Ce sommet v_1 appartient à un certain V_j, c'est donc au joueur j de choisir une arête $(v_1, v_2) \in E$ du graphe et faire « glisser » le jeton le long de cette arête. Le jeu se poursuit de la sorte infiniment. Notons que cette procédure infinie est possible car nous imposons que tout noeud possède un arc sortant.

Les glissements successifs du jeton de noeud en noeud décrit un chemin infini dans le graphe du jeu et les choix effectués par chaque joueur déterminent leur stratégie. Définissons formellement ces notions ainsi que certains concepts qui y sont relatifs.

2.2 Stratégie

Notions de chemin et de jeu

Une *partie* $\rho \in V^\omega$(respectivement une *histoire* $h \in V^*$) dans $\mathcal{A}$ est un chemin infini (respectivement fini) à travers le graphe. Nous notons ε l'histoire vide, *Plays* l'ensemble des jeux dans $\mathcal{A}$ et *Hist* l'ensemble des histoires. Nous utilisons les notations suivantes $\rho = \rho_0\rho_1\rho_2\rho_3\ldots$ (où $\rho_0, \rho_1, \ldots \in V$) représente un jeu et de manière similaire, pour une histoire h, $h = h_0h_1h_2h_3...h_k$ (pour un certain $k \in \mathbb{N}$) où $h_0, h_1, \ldots \in V$.

Soit $h = h_0h_1\ldots h_k$ une histoire et soit $v \in V$ tel que $(h_k, v) \in E$ on note hv l'histoire $h_0h_1\ldots h_kv$. De même, étant donné une histoire $h = h_0h_1h_2h_3...h_k$ et un jeu $\rho = \rho_0\rho_1\rho_2\ldots$ tels que $(h_k, \rho_0) \in E$ on note $h\rho$ le jeu $h_0h_1\ldots h_k\rho_0\rho_1\rho_2\ldots$.

Etant donné une histoire $h = h_0h_1h_2h_3...h_k$, on définit une fonction *Last* (respectivement *First*) qui prend comme argument l'histoire h et qui retourne le dernier sommet h_k (respectivement le premier sommet h_0). Nous définissons l'ensemble des histoires telles que c'est au tour du joueur $i \in \Pi$ de prendre une décision $Hist_i = \{h \in Hist | Last(h) \in V_i\}$.

Remarque 2.1. Si un sommet initial v_0 a été fixé, alors tous les jeux (et toutes les histoires) commencent par le sommet v_0.

Définition 2.4 (Stratégie)**.** Une *stratégie* d'un joueur $i \in \Pi$ dans $\mathcal{A}$ est une fonction $\sigma_i : Hist_i \to V$ telle que à chaque histoire $h = h_0h_1h_2h_3...h_k$ pour laquelle $h_k \in V_i$ est associée un sommet $v \in V$. De plus, on a : $(Last(h), \sigma_i(h)) \in E$.

A la notion de stratégie, nous pouvons associer celle de partie consistante avec une stratégie. Une partie est dite consistante avec une stratégie d'un joueur si à chaque fois que c'est au tour de ce joueur de choisir l'action qu'il veut effectuer, ce choix est conforme à celui définit par sa stratégie. Nous définissons cela formellement :

Définition 2.5 (Partie consistante)**.** Une partie $\rho = \rho_0\rho_1 \dots$ est dite *consistante* avec une stratégie σ_i du joueur i si pour tout préfixe $p = \rho_0\rho_1 \dots \rho_k$ (pour un certain $k \in \mathbb{N}$) tel que $p \in Hist_i$ on a : $\sigma_i(p) = \rho_{k+1}$.

Notons que la notion de jeu consistant est facilement adaptable à la notion d'*histoire consistante*.

Notations. Tout au long de ce document nous utilisons les conventions suivantes :

- Un *profil de stratégies* $(\sigma_i)_{i\in\Pi}$ est un tuple tel que pour tout i, σ_i désigne la stratégie du joueur i.
- Soit $(\sigma_i)_{i\in\Pi}$ un profil de stratégies, pour un certain joueur $j \in \Pi$ on note $(\sigma_i)_{i\in\Pi} = (\sigma_j, \sigma_{-j})$.
- A un profil de stratégies $(\sigma_i)_{i\in\Pi}$ et à un sommet initial v_0 est associé un unique jeu ρ qui est consistant avec toutes les stratégies σ_i. Ce jeu est appelé *outcome* de σ_i et est noté $\langle(\sigma_i)_{i\in\Pi}\rangle_{v_0}$.
- On note Σ_i l'ensemble des stratégies de J_i.

Stratégie avec mémoire vs. stratégie sans mémoire

Lorsque l'on cherche des stratégies pour un joueur J_i, on distingue les *stratégies sans mémoire*, les *stratégies avec mémoire finie* et les *stratégies avec mémoire infinie*. Nous définissons ci-dessous ces différents concepts.

Définition 2.6 (Stratégie sans mémoire)**.** Une stratégie $\sigma_i \in \Sigma_i$ est une *stratégie sans mémoire* si le choix du prochain sommet dépend uniquement du sommet courant (*i.e.,* $\sigma_i : V_i \to V$).

Définition 2.7 (Stratégie à mémoire finie)**.** Une stratégie $\sigma_i \in \Sigma_i$ est une *stratégie à mémoire finie* si on peut lui associer un *automate de Mealy* $\mathcal{A} = (M, m_0, V, \delta, \nu)$ où :

- M est un ensemble fini non vide d'états de mémoire,
- $m_0 \in M$ est l'état initial de la mémoire,
- $\delta : M \times V \to M$ est la fonction de mise à jour de la mémoire,
- $\nu : M \times V_i \to V$ est la fonction de choix, telle que pour tout $m \in M$ et $v \in V_i$ $(v, \nu(m, v)) \in E$.

On peut étendre la fonction de mise à jour de la mémoire à une fonction $\delta^* : M \times Hist \to M$ définie par récurrence sur la longueur de $h \in Hist$ de la manière suivante :

$$\begin{cases} \sigma^*(m, \varepsilon) = m \\ \sigma^*(m, hv) = \sigma(\sigma^*(m, h), v) \quad \text{pour tout } m \in M \text{ et } hv \in Hist \end{cases}$$

La stratégie $\sigma_{\mathcal{A}_i}$ calculée par un automate fini $\mathcal{A}_i$ est définie par $\sigma_{\mathcal{A}_i}(hv) = \nu(\delta^*(m_0, h), v)$ pour tout $hv \in Hist_i$. Cela signifie qu'à chaque fois qu'une décision est prise, la mémoire est modifiée en conséquence et que chaque nouvelle décision est prise en fonction de la mémoire enregistrée jusque maintenant. Dès lors, on dit que σ_i est une stratégie à mémoire finie s'il existe un automate fini $\mathcal{A}_i$ tel que $\sigma_i = \sigma_{\mathcal{A}_i}$. De plus, on dit que σ_i a une mémoire de taille $|M|$.

Nous pouvons remarquer que le cas des stratégies sans mémoire est un cas particulier des stratégies à mémoire finie. En effet, dans ce cas l'automate de Mealy qui lui est associé est tel que $|M| = 1$.

Définition 2.8 (Stratégie à mémoire infinie)**.** Une stratégie $\sigma_i \in \Sigma_i$ est une *stratégie à mémoire infinie* si elle n'est ni sans mémoire ni à mémoire finie.

De même, nous pouvons définir ce que signifie la notion de mémoire pour un profil de stratégies.

Définition 2.9. On appelle $(\sigma_i)_{i \in \Pi}$ un profil de stratégies de mémoire m si pour tout $i \in \Pi$, la stratégie σ_i a une mémoire d'au plus m.

Mettons maintenant en lumière ces notions par un simple exemple.

Exemple 2.1. Soit le jeu illustré par la figure 1. L'arène de ce jeu est définie de la manière suivante :

- $\Pi = \{1, 2\}$;
- $V = \{v_0, v_1, v_2, v_3, v_4\}$;
- $V_1 = \{v_0, v_1, v_3, v_4\}$ et $V_2 = \{v_2\}$;

- $E = \{(v_0, v_1), (v_1, v_2), (v_1, v_0), (v_2, v_4), (v_3, v_4), (v_4, v_2), (v_2, v_3), (v_4, v_3), (v_3, v_0)\}$

Si nous fixons v_0 comme sommet initial nous avons que $v_0v_1v_0(v_2v_4)^\omega$ ou encore $(v_0v_1v_2v_3)^\omega$ sont des parties du jeu tandis que $v_0v_1v_0$ et $v_0v_1v_2v_3$ sont des histoires du jeu. De plus, $v_0v_1v_0$ appartient à $Hist_1$ et $v_0v_1v_2$ appartient à $Hist_2$.

Illustrons désormais le concept de stratégie sur cet exemple. Définissons une stratégie sans mémoire possible pour chacun des deux joueurs.

$$\sigma_1(v) = \begin{cases} v_1 & \text{si } v = v_0 \\ v_2 & \text{si } v = v_1 \\ v_3 & \text{si } v = v_4 \\ v_0 & \text{si } v = v_3 \end{cases} \qquad \text{et} \qquad \sigma_2(v) = v_3 \text{ si } v = v_2$$

Nous remarquons que le jeu $(v_0v_1v_2v_3)^\omega$ est consistant avec les stratégies σ_1 et σ_2 tandis que $v_0v_1v_0(v_2v_4)^\omega$ n'est pas consistant avec σ_1.

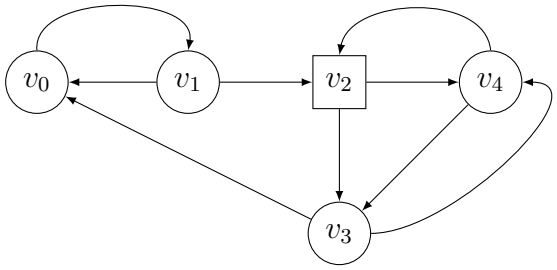


FIGURE 1 – Arène d'un jeu sur graphe

Nous donnons maintenant un exemple de machine de Mealy qui permet de calculer une stratégie à mémoire finie. Cet exemple est issu de la référence [3].

Exemple 2.2 (Automate de Mealy)**.** Nous considérons cette fois le jeu dont l'arène est représentée à la figure 2. Dans cet exemple, $\Pi = \{1, 2\}$, $V_1 = \{v_0, v_2, v_3\}$ (les noeuds de J_1 sont représentés par les ronds) et $V_2 = \{v_1\}$ (ce noeud est représenté par un carré).

Posons comme stratégie à mémoire finie pour le second joueur la stratégie suivante : $\sigma_2(v_1) = v_2$ et $\sigma_2(hv_1) = v_0$ pour tout $h \neq \varepsilon$. Nous pouvons construire un automate de Mealy $\mathcal{M}_{\sigma_2}$ qui calcule cette stratégie. Celui-ci est illustré à la figure 3.

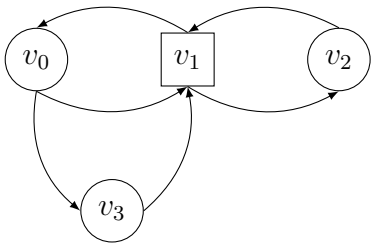


FIGURE 2 – Arène d'un jeu sur graphe

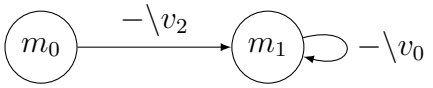


FIGURE 3 – Automate de Mealy, $\mathcal{M}_{\sigma_2}$

L'ensemble des états de mémoire est l'ensemble $M = \{m_0, m_1\}$ dont m_0 est l'état initial. La fonction de mise à jour de la mémoire $\delta : M \times V \to M$ est définie par $\delta(m_0, v) = m_1$ et $\delta(m_1, v) = m_1$ pour tout $v \in V$ et la fonction de choix $\nu : M \times V_2 \to V$ est donnée par $\nu(m_0, v_1) = v_2$ et $\nu(m_1, v_1) = v_0$. L'étiquette $-\backslash v'$ sur un arc (m, m') signifie que $\delta(m, v) = m'$ pour tout $v \in V$ et $\nu(m, v_1) = v'$.

Comme nous l'avons dit précédemment, lorsque l'on définit un jeu sur graphe nous devons préciser l'arène du jeu ainsi que les objectifs que chaque joueur doit atteindre. Dans la section suivante nous allons nous intéresser à un certain type d'objectif : l'objectif d'atteignabilité.

3 Jeux d'atteignabilité

Un jeu d'atteignabilité est un jeu sur graphe particulier. Chaque joueur possède un ensemble objectif (un sous-ensemble de sommets du graphe) qu'il souhaite atteindre. Le déroulement d'une partie d'un jeu d'atteignabilité se déroule comme un jeu sur graphe (cf. section 2.1) sauf que le but de chaque joueur est d'atteindre un élément de son ensemble objectif. On peut considérer les jeux d'atteignabilité selon deux points de vue : les jeux qualitatifs et les jeux quantitatifs. Nous développons dans cette section ces deux notions.

3.1 Jeux qualitatifs

Nous nous attardons dans un premier temps sur les *jeux d'atteignabilité à objectif qualitatif*. Ce sont des jeux sur graphe tels que chaque joueur tente d'atteindre un élément de son ensemble objectif. Nous nous intéressons alors à savoir si à partir d'un noeud du graphe un joueur peut toujours s'assurer d'atteindre un élément de son ensemble objectif et ce quelles que soient les stratégies adoptées par les autres joueurs. La réponse à cette question est alors binaire : soit oui, soit non. Dans le cas où celle-ci est positive, nous disons que ce joueur possède une *stratégie gagnante* et que ce noeud de départ est un *état gagnant*.

Définition 3.1 (Jeu d'atteignabilité à objectif qualitatif)**.** Un *jeu d'atteignabilité à objectif qualitatif* est un jeu sur graphe $\mathcal{G} = (\mathcal{A}, (Goal_i)_{i\in\Pi}, (\Omega_i)_{i\in\Pi})$ où :

- $\mathcal{A} = (\Pi, V, (V_i)_{i\in\Pi}, E)$ est l'arène d'un jeu sur graphe,
- Pour tout $i \in \Pi$, $Goal_i \subseteq V$ est l'ensemble des sommets de V que J_i essaie d'atteindre,
- Pour tout $i \in \Pi$, $\Omega_i = \{(u_j)_{j\in\mathbb{N}} \in V^\omega | \exists k \in \mathbb{N} \text{ tel que } u_k \in Goal_i\}$. C'est l'ensemble des jeux ρ sur $\mathcal{G}$ pour lesquels J_i atteint son objectif.

Définition 3.2 (Stratégie gagnante)**.** Soit $v \in V$, soit σ_i une stratégie du joueur i, on dit que σ_i est *gagnante pour* J_i à partir de v si

$$\forall \sigma_{-i},\ \langle \sigma_i, \sigma_{-i} \rangle_v \in \Omega_i.$$

Définition 3.3. Soit $\mathcal{G} = (\Pi, (V, E), (V_i)_{i\in\Pi}, (Goal_i)_{i\in\Pi}, (\Omega_i)_{i\in\Pi})$,
$W_i = \{v_j | v_j \in V \text{ et } \exists \sigma_i \text{ une stratégie gagnante pour } J_i \text{ à partir de } v_j\}$ est *l'ensemble des états gagnants* de J_i. C'est l'ensemble des sommets de $\mathcal{G}$ à partir desquels J_i est assuré d'atteindre son objectif.

Une fois le concept de jeu d'atteignabilité clairement établi, nous pouvons nous poser les questions suivantes : « Quels joueurs peuvent gagner le jeu ? » et « Quelle stratégie doivent adopter les joueurs pour atteindre leur objectif quelle que soit la stratégie jouée par les autres joueurs ? » .

Dans le cadre de ce travail nous abordons uniquement le cas des jeux qualitatifs à deux joueurs et à sommes nulles.

Jeux à deux joueurs et à somme nulle

Nous sommes intéressés par l'étude des jeux d'atteignabilité à objectif qualitatif dans le cadre des jeux à deux joueurs. Dans ce cadre, nous notons $\Pi = \{1, 2\}$. Nous supposons que $\Omega_2 = V^\omega \backslash \Omega_1$, on dit alors que le jeu est *à somme nulle*. Ceci signifie que dans le cas du jeu d'atteignabilité à deux joueurs le but de J_2 est d'empêcher J_1 d'atteindre son objectif. Nous allons expliciter une méthode permettant de déterminer à partir de quels sommets J_1 (respectivement J_2) est assuré de gagner le jeu (respectivement d'empêcher J_1 d'atteindre son objectif). Dans ce cas, nous posons F l'ensemble des sommets objectifs de J_1.

Nous commençons en énonçant et en prouvant quelques propriétés qui permettent d'élaborer un processus algorithmique afin de trouver les états gagnants de chacun de deux joueurs.

Propriété 3.1. Soit $\mathcal{G}$ un jeu, on a : $W_1 \cap W_2 = \emptyset$.

Preuve. Supposons au contraire que $W_1 \cap W_2 \neq \emptyset$. Cela signifie qu'il existe $s \in W_1$ tel que $s \in W_2$.
$s \in W_1$ si et seulement si il existe σ_1 une stratégie de J_1 telle que pour toute σ_2 stratégie de J_2 nous avons :$\langle \sigma_1, \sigma_2 \rangle_s \in \Omega_1$.
$s \in W_2$ si et seulement si il existe $\tilde{\sigma}_2$ une stratégie de J_2 telle que pour toute $\tilde{\sigma}_1$ stratégie de J_1 nous avons :$\langle \tilde{\sigma}_1, \tilde{\sigma}_2 \rangle_s \in \Omega_2$.
Dès lors, on obtient : $\langle \sigma_1, \tilde{\sigma}_2 \rangle_s \in \Omega_1 \cap \Omega_2$. Or $\Omega_1 \cap \Omega_2 = \emptyset$, ce qui amène la contradiction.

□

Cette première propriété signifie simplement qu'un noeud du jeu ne peut pas être un état gagnant pour les deux joueurs. En couplant cette propriété avec la définition suivante, nous constatons que dans le cas des jeux déterminés tels que $W_1 \cap W_2 = \emptyset$, les ensembles des états gagnants de J_1 et de J_2 forment une partition de V. De surcroit, si l'on connait un procédé permettant de déterminer W_1 alors on connait également W_2.

Définition 3.4 (Jeu déterminé)**.** Soit $\mathcal{G}$ un jeu, on dit que ce jeu est *déterminé* si et seulement si $W_1 = V \backslash W_2$.

Définition 3.5. Soit $X \subseteq V$.
Posons $Pre(X) = \{v \in V_1 | \exists v'((v, v') \in E) \wedge (v' \in X)\} \cup \{v \in V_2 | \forall v'((v, v') \in E) \Rightarrow (v' \in X)\}$. Définissons $(X_k)_{k\in\mathbb{N}}$ la suite de sous-ensembles de V suivante :

$$\begin{cases} X_0 = F \\ X_{k+1} = X_k \cup Pre(X_k) \end{cases}.$$

Propriété 3.2.
La suite $(X_k)_{k\in\mathbb{N}}$ est ultimement constante (*i.e.*, $\exists n \in \mathbb{N}\, \forall n \geqslant n X_k = X_n$).

Preuve. Premièrement, nous avons clairement que $\forall k \in \mathbb{N}, X_k \subseteq X_{k+1}$.
Deuxièmement, nous avons : $\forall k \in \mathbb{N}, |X_k| \leqslant |V|$.
Dès lors, vu que la suite $(X_k)_{k\in\mathbb{N}}$ est une suite croissante dont la cardinalité des ensembles est bornée par celle de V, elle est ultimement constante.

□

Définition 3.6. L'*attracteur de* F , noté $Attr(F)$, est défini de la manière suivante :$Attr(F) = \bigcup_{k\in\mathbb{N}} X_k$.

$Pre(X)$ est l'ensemble des noeuds à partir desquels J_1 est certain d'atteindre un élément de X en une étape (en empruntant un arc du graphe). Calculer X_k revient donc à déterminer à partir de quels états J_1 est assuré d'atteindre son objectif en au plus k étapes. Dès lors, comme $Attr(F)$ est la limite de la suite $(X_k)_{k\in\mathbb{N}}$, trouver $Attr(F)$ revient à trouver W_1. C'est le résultat que nous énonçons et démontrons ci-dessous.

Propriété 3.3.

$$W_1 = Attr(F) \tag{1}$$

$$W_2 = V \backslash Attr(F) \tag{2}$$

Preuve. Pour prouver (1) et (2) nous allons procéder en plusieurs étapes en prouvant chaque inclusion séparément. Nous commençons par montrer que $Attr(F) \subseteq W_1$ puis que $V \backslash Attr(F) \subseteq W_2$ et enfin nous concluons en prouvant que $W_1 \subseteq Attr(F)$ et $W_2 \subseteq V \backslash Attr(F)$. De ces quatre inclusions nous pouvons alors conclure les égalités recherchées.

$\mathbf{Attr(F)} \subseteq \mathbf{W_1}$: Soit $v \in Attr(F)$ alors par la propriété 3.2 on a : $Attr(F) = X_N$ pour un certain $N \in \mathbb{N}$. Montrons par récurrence sur n que dans ce cas, pour tout $n \in \mathbb{N}$ tel que $X_n \subseteq Attr(F)$ on peut construire une stratégie σ_1 pour J_1 telle que pour tout $\sigma_2 \in \Sigma_2$, $\langle \sigma_1, \sigma_2 \rangle_v \in \Omega_1$.

- ⋆ Pour $n = 0$: alors $v \in X_0 = F$ et l'objectif est atteint par J_1. σ_1 peut alors être définie de n'importe quelle manière puisque l'objectif est atteint.
- ⋆ Supposons que la propriété soit vérifiée pour tout $0 \leqslant n \leqslant k$ et montrons qu'elle est toujours satisfaite pour $n = k + 1 \leqslant N$.
  Soit $v \in X_{k+1} = X_k \cup Pre(X_k)$.
  Si $v \in X_k$ alors par hypothèse de récurrence, il existe σ_1 telle que pour tout σ_2, $\langle \sigma_1, \sigma_2 \rangle_v \in \Omega_1$.
  Si $v \in Pre(X_k) \backslash X_k$, alors si $v \in V_1$ par définition de $Pre(X_k)$ on sait qu'il existe $v' \in X_k$ tel que $(v, v') \in E$. De plus, comme $v' \in X_k$, par hypothèse de récurrence, on sait qu'il existe $\tilde{\sigma}_1$ telle que $\forall \sigma_2 \in \Sigma_2$ $\langle \sigma_1, \sigma_2 \rangle_{v'} \in \Omega_1$. Ainsi, on définit $\sigma_1(u) = \begin{cases} v' & \text{si } u = v \\ \tilde{\sigma_1}(u) & \text{sinon} \end{cases}$. Il en découle que $\forall \sigma_2 \in \Sigma_2 : \langle \sigma_1, \sigma_2 \rangle_v \in \Omega_1$. Tandis que si $v \in V_2$, par définition de $Pre(X_k)$, quelle que soit la stratégie σ_2 adoptée par J_2 nous sommes assurés que $\sigma_2(v) \in X_k$. Donc par hypothèse de récurrence, on sait qu'il existe $\tilde{\sigma}_1 \in \Sigma_1$ telle que pour toute stratégie $\tilde{\sigma_2} \in \Sigma_2$ on ait : $\langle \tilde{\sigma}_1, \tilde{\sigma}_2 \rangle_{\sigma_2(v)} \in \Omega_1$. Il suffit donc de prendre $\sigma_1 = \tilde{\sigma}_1$.

  Dès lors, le résultat : $\exists \sigma_1 \in \Sigma_1$ telle que $\forall \sigma_2 \in \Sigma_2$ on ait $\langle \sigma_1, \sigma_2 \rangle_v \in \Omega_1$ est assuré.

Par cette preuve par récurrence, l'assertion est bien vérifiée.

$\mathbf{V \backslash Attr(F)} \subseteq \mathbf{W_2}$: Soit $v \in V \backslash Attr(F)$. Une stratégie gagnante pour J_2 est une stratégie telle que à chaque tour de jeu le sommet s considéré soit dans l'ensemble $V \backslash Attr(F)$. En effet, puisque $F \subseteq Attr(F)$, en s'assurant de rester en dehors de l'attracteur, on est certain de ne pas atteindre un élément de F.

Si $v \in V_1$ alors cela signifie que $\forall v'$ tel que $(v, v') \in E$ on a $v' \in V \backslash Attr(F)$. Tandis que si $v \in V_2$ alors $\exists v'$ tel que $(v, v') \in E$ et $v' \in V \backslash Attr(F)$. On définit donc $\sigma_2(v) = v'$. Pour construire la stratégie gagnante σ_2 de J_2, on réitère cet argument.

$\mathbf{W_1} \subseteq \mathbf{Attr(F)}$: Supposons au contraire : $W_1 \not\subseteq Attr(F)$. Cela signifie qu'il existe $v \in W_1$ tel que $v \notin Attr(F)$. D'où $v \in V \backslash Attr(F)$ et comme

$V \backslash Attr(F) \subseteq W_2$, on a $v \in W_2$. Or par la propriété 3.1, $W_1 \cap W_2 = \emptyset$ et ici $v \in W_1$ et $v \in W_2$. Ce qui amène la contradiction.

$\mathbf{W_2 \subseteq V \backslash Attr(F)}$: La preuve est similaire à celle de $W_1 \subseteq Attr(F)$.

Ces quatre inclusions d'ensemble démontrent donc (1) et (2). □

Remarque 3.1. Cette propriété nous montre que les jeux d'atteignabilité à objectif qualitatif et à deux joueurs sont déterminés.

Illustrons maintenant le calcul de l'attracteur sur un simple exemple.

Exemple 3.1. Soit $\mathcal{G} = ((V, E), V_1, V_2, \Omega_1, \Omega_2, F)$ le jeu d'atteignabilité représenté par le graphe ci-dessous. On a : $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$, E est l'ensemble des arcs représentés sur le graphe, V_1 est représenté par les sommets de forme ronde, V_2 est représenté par les sommets de forme carrée et F est l'ensemble des sommets grisés.
Appliquons sur l'exemple ci-dessous, le principe de l'attracteur.

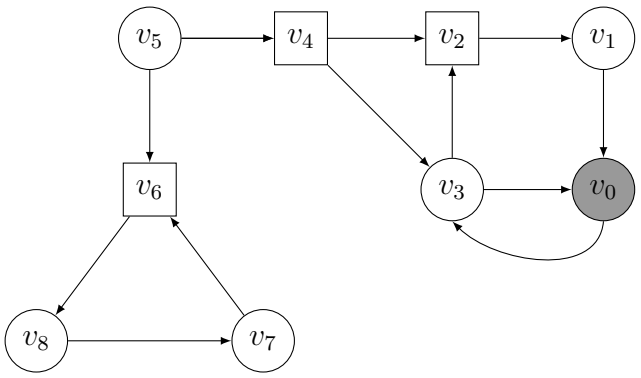


FIGURE 4 – $X_0 = \{v_0\}$ -Situation initiale, le seul état grisé est l'état objectif.

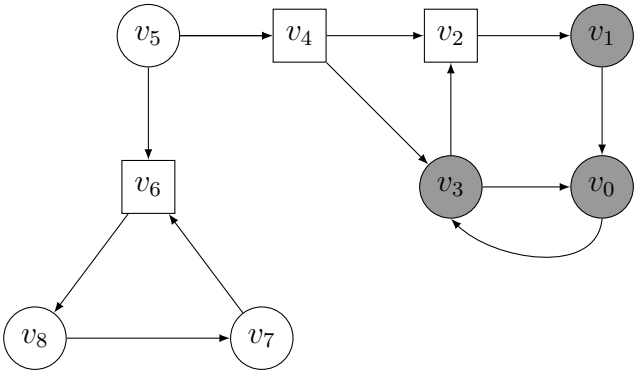


FIGURE 5 – $X_1 = \{v_0, v_1, v_3\}$ – Première étape, $v_1, v_3 \in Pre(X_0)$ car $v_1, v_3 \in V_1$ et il existe un arc entre v_1 et v_0 et entre v_3 et v_0.

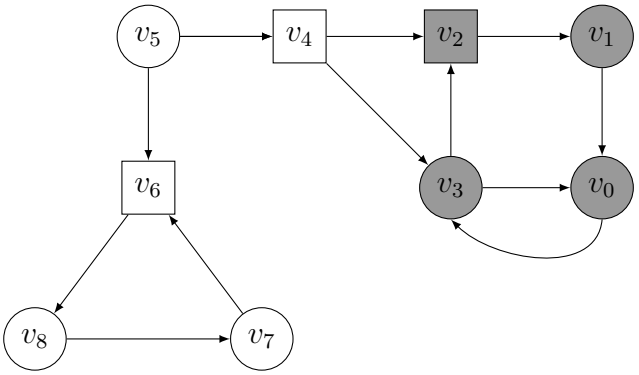


FIGURE 6 – $X_2 = \{v_0, v_1, v_2, v_3\}$ – Deuxième étape, $v_2 \in Pre(X_1)$ car $v_2 \in V_2$ et tous les arcs sortant de v_2 atteignent un état de X_1.

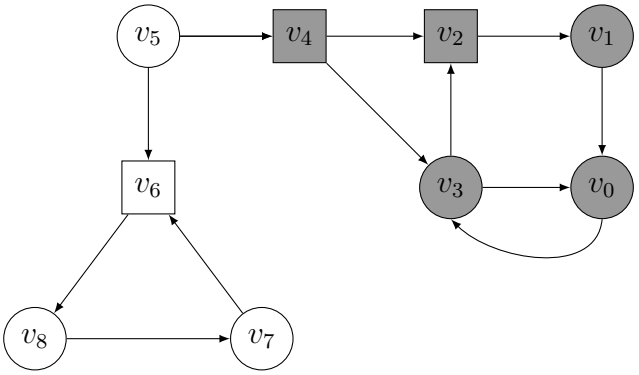


FIGURE 7 – $X_3 = \{v_0, v_1, v_2, v_3, v_4\}$ – Troisième étape, $v_4 \in Pre(X_2)$ car $v_4 \in V_2$ et tous les arcs sortants de v_2 atteignent un état de X_2 (v_2 et v_3).

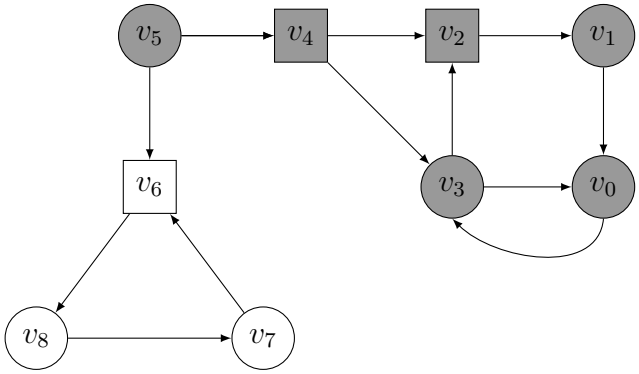


FIGURE 8 – $X_4 = Attr(F) = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ – Dernière étape, $v_5 \in Pre(X_3)$ car $v_4 \in V_1$ et il existe un arc sortant de v_5 vers $v_4 \in Pre(X_3)$.

Implémentation :

Au vu de la définition de l'attracteur d'un ensemble, l'implémentation de la résolution d'un jeu d'atteignabilité à deux joueurs avec objectif qualitatif en découle aisément. En effet, si nous possédons un algorithme permettant de calculer $Pre(X)$ pour tout sous-ensemble X de sommets du graphe, il suffit alors d'appeler plusieurs fois cet algorithme afin de générer la suite $(X_k)_{k\in\mathbb{N}}$ conformément à la définition 3.5 jusqu'à ce que celle-ci soit constante. Nous donnons ci-après le pseudo-code d'un algorithme permettant de résoudre ce type de jeu.

Soit $\mathcal{G} = ((V, E), V_1, V_2, F, \Omega_1, \Omega_2)$, nous supposons que V comprend n sommets numérotés de 0 à $n-1$ et que le graphe $G = (V, E)$ est encodé sur base de sa *liste d'adjacence* : adj - *i.e.,* pour chaque sommet indicé par k, on possède une liste des sommets v_l tels que $0 \leqslant l \leqslant n-1$ et $(v_k, v_l) \in E$.

Un premier algorithme (algorithme 1) calcule à partir d'un sous ensemble $X \subseteq V$ de V l'ensemble $Pre(X)$ comme défini en 3.5. Le principe de cet algorithme est le suivant : pour tout sommet $v \in V_1$ on teste s'il existe un arc sortant de v vers un sommet de X. Si c'est le cas, alors $v \in Pre(X)$. On traite ensuite tous les sommets $v \in V_2$. Pour qu'un tel sommet v appartienne à $Pre(X)$ il faut que tous les arcs sortants de v atteignent un sommet de X.

Notations. Pour l'écriture de cet algorithme nous adoptons les conventions de notation suivantes :

- On note *adj* la liste d'adjacence. Pour récupérer la liste des successeurs du noeud v_1, on note donc $adj[1]$ et pour récupérer le premier

successeur de v_1 on écrit donc $adj[1][0]$ (on suppose que les listes utilisées sont indicées à partir de 0).

- La notation $|adj[n]|$ désigne le nombre de successeurs que possède le noeud indicé par n (*i.e.,* la longueur de la liste $adj[n]$).

Un second algorithme (algorithme 2) permet de calculer les états gagnants de J_1 en utilisant les résultats 3.2 et 3.3. En effet, on y construit itérativement la suite $(X_k)_{k\in\mathbb{N}}$ jusqu'à ce qu'elle devienne ultimement constante (*i.e.,* jusqu'à ce que la cardinalité des ensembles X_k ne varie plus).

Complexité

Terminons en abordant brièvement la complexité de ces algorithmes. Posons n (respectivement m) le nombre de sommets du graphe (respectivement le nombre d'arcs).

Algorithme 1 Pour chaque noeud du graphe, on parcourt sa liste de successeurs. Dans le pire des cas, on doit parcourir toute cette liste et donc effectuer un test pour chaque successeur de chaque noeud. Chacun de ses tests et des opérations qui en découlent sont en $\mathcal{O}(1)$ et sont effectuées dans le pire cas m fois. La complexité de cet algorithme est donc en $\mathcal{O}(m)$.

Algorithme 2 Supposons que nous utilisons une structure de données telle que l'on puisse récupérer la taille des ensembles en $\mathcal{O}(1)$, alors la complexité de cet algorithme est en $\mathcal{O}(m.n)$. En effet, dans le pire cas, on part d'un ensemble F de cardinalité égale à 1 et on ne rajoute qu'un seul élément à la suite $(X_k)_{k\in\mathbb{N}}$ à chaque étape du calcul de $Pre(X_k)$ jusqu'à obtenir un ensemble de cardinalité n. Dès lors, dans ce cas, on fait appel $n-1$ fois à l'algorithme 1 et on obtient bien un algorithme en $\mathcal{O}(m.n)$.

Dans la section suivante nous ne nous contentons plus d'une réponse binaire à la question qui est de savoir si un joueur est assuré d'atteindre ou non son objectif. Nous désirons quantifier la réalisation de cet objectif.

Algorithme 1 PreX

Entrée(s): Un sous-ensemble X de sommets de V.
Sortie(s): Pre(X)

```
1: preX ← un ensemble vide
2: pour tout v_i ∈ V_1 faire
3:    ind ← 0
4:    existeArc ← faux
5:    tant que ¬existeArc et ind ⩽ |adj[i]| faire
6:       si adj[i][ind] ∈ X alors
7:          existeArc ← vrai
8:       sinon
9:          ind ← ind + 1
10:      fin si
11:   fin tant que
12:   si (existeArc = vrai) alors
13:      ajouter v_i à preX
14:   fin si
15: fin pour
16: pour tout v_i ∈ V_2 faire
17:   ind ← 0
18:   tousArcs ← vrai
19:   tant que tousArcs et ind ⩽ |adj[i]| faire
20:      si adj[i][ind] ∈ X alors
21:         ind ← ind + 1
22:      sinon
23:         tousArcs ← faux
24:      fin si
25:   fin tant que
26:   si (tousArcs = vrai) alors
27:      ajouter v_i à preX
28:   fin si
29: fin pour
30: retourner preX
```

Algorithme 2 Attr(F)

Entrée(s): F l'ensemble des états objectifs de J_1
Sortie(s): $Attr(F)$ l'ensemble des états gagnants de J_1

```
1: X_k ← F
2: tailleDiff ← vrai
3: tant que (tailleDiff = vrai) faire
4:    ancCard ← |X_k|
5:    preX_k ← PreX(X_k)
6:    Ajouter à X_k tous les éléments de preX_k
7:    nouvCard ← |X_k|
8:    si ancCard = nouvCard alors
9:       tailleDiff ← faux
10:   fin si
11: fin tant que
12: retourner X_k
```

3.2 Jeux quantitatifs

Contrairement aux jeux à objectif qualitatif pour lesquels l'objectif d'un joueur est d'assurer qu'une certaine propriété soit vérifiée, on associe aux *jeux à objectif quantitatif* une certaine valeur quantifiée. Le but d'un joueur est donc de *maximiser* ou de *minimiser* cette valeur afin que sa satisfaction soit maximale.

Nous introduisons cette section par un exemple, ensuite nous abordons les notions essentielles aux jeux quantitatifs en distinguant les *jeux multi-joueurs* et les *jeux à deux joueurs*.
Les définitions et les notions sont inspirées de l'article de Brihaye *at al.* [1].

Exemple introductif

Antoine et Thomas désirent se rendre à l'école à pied. Le chemin étant long, Antoine propose à Thomas un jeu. A chaque carrefour et à tour de rôle un des deux garçons choisit la route à emprunter. A chaque mètre parcouru, Thomas devra donner un bonbon à Antoine. L'objectif de Thomas est donc d'emprunter le plus court chemin jusque l'école afin de minimiser son coût. Tandis que l'objectif d'Antoine est d'emprunter le plus long chemin pour obtenir le plus de bonbons possible.

Au vu de cet exemple, il est clair que le modèle des jeux qualitatifs n'est pas suffisant pour modéliser cette situation. En effet, il ne permet pas

de caractériser le fait que plus Antoine obtiendra de bonbons, plus il sera satisfait et inversement pour Thomas. Pour ce faire, nous devons introduire les concepts de *fonction de gain* (ou *fonction de coût* en fonction du point de vue duquel on se place) ainsi qu'un nouveau concept de solution pour ces jeux appelés *jeu avec coût (cost games)* : les *équilibres de Nash*. Nous supposons que dans de tels jeux les joueurs sont *rationnels* c'est-à-dire qu'ils jouent de telle sorte à maximiser leur gain ou minimiser leur coût.

Un moyen de modéliser notre exemple est de considérer un jeu Min-Max à somme nulle. Nous commençons donc en abordant les notions de jeux Min-Max avec coût ainsi que de stratégies optimales et nous continuons en traitant le cas des jeux multijoueurs avec coût et le concept d'équilibre de Nash.

3.2.1 Jeux Min-Max avec coût

Définition 3.7 (Jeu Min-Max avec coût)**.**
Soit une arène $\mathcal{A} = (V, (V_{Min}, V_{Max}), E)$, un *jeu Min-Max avec coût* est un tuple $\mathcal{G} = (\mathcal{A}, Cost_{Min}, Gain_{Max})$, où

- $Cost_{Min} : Plays \to \mathbb{R} \cup \{+\infty, -\infty\}$ est la *fonction de coût* du joueur *Min*.
- $Gain_{Max} : Plays \to \mathbb{R} \cup \{+\infty, -\infty\}$ est la *fonction de gain* du joueur *Max*.

De plus, $Cost_{Min}(\rho) \geqslant Gain_{Max}(\rho)$ pour tout $\rho \in Plays$.

Remarque 3.2. Dans cette définition, on sous-entend que $\Pi = \{Min, Max\}$.

Pour chaque $\rho \in Plays$, $Cost_{Min}(\rho)$ représente le montant que *Min* doit payer quand la partie ρ est jouée et $Gain_{Max}(\rho)$ représente le gain que *Max* obtient quand la partie ρ est jouée. Le but de *Min* (resp. *Max*) est donc de **minimiser** (resp. **maximiser**) sa fonction de coût (resp. fonction de gain). Ce qui explique le choix des noms des joueurs.

Afin de pouvoir définir ces fonctions de coût ou de gain, il est souvent utile de définir une fonction de poids sur les arcs du graphe. En d'autres mots, à chaque arc du graphe est associée une valeur chiffrée.

Définition 3.8 (Fonction de poids)**.** A chaque arc d'un graphe $G = (V, E)$ on peut y associer un *poids* (*i.e.,* une valeur chiffrée). On associe donc à G une *fonction de poids* $w : E \to \mathbb{R}$. On dit alors que G est un graphe *pondéré*.

Remarque 3.3. Même si par définition, considérer une fonction de poids à valeurs réelles est correct, dans notre cas nous ne nous intéressons qu'à des fonctions de poids à valeurs dans $\mathbb{N}_0$.

La fonction de poids que l'on associe à un graphe dépend de la situation qui est modélisée. Voici quelques exemples de fonctions qui peuvent être envisagées :

Exemple 3.2. Si chaque $v \in V$ représente une ville sur une carte, alors on peut imaginer une fonction de poids représentant une des valeurs suivantes :

- le nombre de kilomètres entre deux villes,
- le temps pour aller d'une ville à l'autre,
- la consommation d'essence pour aller d'une ville à l'autre.

Définition 3.9 (Jeu à somme nulle)**.** Un jeu Min-Max avec coût est dit *à somme nulle* si $Gain_{Max} = Cost_{Min}$.

Définition 3.10 (Garantir le paiement)**.**

Soit $(\mathcal{G}, v_0)$ un jeu Min-Max avec coût initialisé,

On dit que le joueur *Max garantit le paiement* $d \in \mathbb{R}$ dans $(\mathcal{G}, v_0)$
ssi
$$\exists \sigma_1 \in \Sigma_{Max} \text{ tq } \forall \sigma_2 \in \Sigma_{Min}\ Gain_{Max}(\langle \sigma_1, \sigma_2 \rangle_{v_0}) \geqslant d$$

On dit que le joueur *Min garantit le paiement* $d \in \mathbb{R}$ dans $(\mathcal{G}, v_0)$
ssi
$$\exists \sigma_1 \in \Sigma_{Min} \text{ tq } \forall \sigma_2 \in \Sigma_{Max}\ Cost_{Min}(\langle \sigma_1, \sigma_2 \rangle_{v_0}) \leqslant d$$

Définition 3.11 (Valeur inférieure/supérieure)**.** Soit $\mathcal{G}$ un jeu Min-Max avec coût, on définit pour chaque sommet $v \in V$:

- *Valeur supérieure :* $\overline{Val}(v) = \inf\limits_{\sigma_1 \in \Sigma_{Min}} \sup\limits_{\sigma_2 \in \Sigma_{Max}} Cost_{Min}(\langle \sigma_1, \sigma_2 \rangle_v)$.
- *Valeur inférieure :* $\underline{Val}(v) = \sup\limits_{\sigma_2 \in \Sigma_{Max}} \inf\limits_{\sigma_1 \in \Sigma_{Min}} Gain_{Max}(\langle \sigma_1, \sigma_2 \rangle_v)$.

Remarque 3.4. La *valeur supérieure* $\overline{Val}(v)$ est la plus petite valeur dont J_{Min} garantit le paiement dans $(\mathcal{G}, v)$ et la *valeur inférieure* $\underline{Val}(v)$ est la plus grande valeur dont J_{Max} garantit le paiement dans $(\mathcal{G}, v)$.

Propriété 3.4. Pour tout $v \in V$, on a : $\underline{Val}(v) \leqslant \overline{Val}(v)$

Preuve. Par hypothèse, nous savons :

$$\forall v \in V \, \forall \sigma_1 \in \Sigma_{Min} \, \forall \sigma_2 \in \Sigma_{Max} \, Gain_{Max}(\langle \sigma_1, \sigma_2 \rangle_v) \leqslant Cost_{Min}(\langle \sigma_1, \sigma_2 \rangle_v)$$

Dès lors, l'assertion suivante est également vérifiée.

$$\forall v \in V \, \forall \sigma_1 \in \Sigma_{Min} \, \forall \sigma_2 \in \Sigma_{Max} \inf_{\tilde{\sigma_1} \in \Sigma_{Min}} Gain_{Max}(\langle \tilde{\sigma}_1, \sigma_2 \rangle_v) \leqslant Cost_{Min}(\langle \sigma_1, \sigma_2 \rangle_v)$$

En passant au suprémum sur les deux membres de l'inégalité, nous obtenons :

$$\forall v \in V \, \forall \sigma_1 \in \Sigma_{Min} \sup_{\sigma_2 \in \Sigma_{Max}} \inf_{\tilde{\sigma_1} \in \Sigma_{Min}} Gain_{Max}(\langle \tilde{\sigma}_1, \sigma_2 \rangle_v) \leqslant \sup_{\sigma_2 \in \Sigma_{Max}} Cost_{Min}(\langle \sigma_1, \sigma_2 \rangle_v)$$

De plus, en passant cette fois à l'infimum sur les deux membres de l'inégalité :

$$\forall v \in V \inf_{\sigma_1 \in \Sigma_{Min}} \sup_{\sigma_2 \in \Sigma_{Max}} \inf_{\tilde{\sigma_1} \in \Sigma_1} Gain_{Max}(\langle \tilde{\sigma}_1, \sigma_2 \rangle_v) \leqslant \inf_{\sigma_1 \in \Sigma_{Min}} \sup_{\sigma_2 \in \Sigma_{Max}} Cost_{Min}(\langle \sigma_1, \sigma_2 \rangle_v)$$

Mais le terme $sup_{\sigma_2 \in \Sigma_{Max}} \inf_{\tilde{\sigma_1} \in \Sigma_{Min}} Gain_{Max}(\langle \tilde{\sigma}_1, \sigma_2 \rangle_v)$ n'ayant pas de dépendance en σ_1, nous pouvons réécrire l'inégalité précédente de la manière suivante :

$$\forall v \in V \sup_{\sigma_2 \in \Sigma_{Max}} \inf_{\sigma_1 \in \Sigma_{Min}} Gain_{Max}(\langle \sigma_1, \sigma_2 \rangle_v) \leqslant \inf_{\sigma_1 \in \Sigma_{Min}} \sup_{\sigma_2 \in \Sigma_{Max}} Cost_{Min}(\langle \sigma_1, \sigma_2 \rangle_v)$$

Ce qui est bien ce qu'il fallait démontrer. □

Définition 3.12 (Jeu déterminé et valeur d'un jeu)**.** Soit $\mathcal{G}$ un jeu Min-Max avec coût, on dit que $\mathcal{G}$ est *déterminé* si pour tout $v \in V$, $\overline{Val}(v) = \underline{Val}(v)$. On dit alors que le jeu $\mathcal{G}$ a une *valeur* et pour tout $v \in V$ on note $Val(v) = \overline{Val}(v) = \underline{Val}(v)$.

Définition 3.13 (Stratégie ε-optimale)**.** Soit $\varepsilon > 0$,

- $\sigma_2^* \in \Sigma_{Max}$ est une *stratégie ε-optimale* ssi pour tout $v \in V$ on a :

$$\inf_{\sigma_1 \in \Sigma_{Min}} Gain_{Max}(\langle \sigma_1, \sigma_2^* \rangle_v) \geqslant \underline{Val}(v) - \varepsilon.$$

- $\sigma_1^* \in \Sigma_{Min}$ est une *stratégie ε-optimale* ssi pour tout $v \in V$ on a :
$$\sup_{\sigma_2 \in \Sigma_{Max}} Cost_{Min}(\langle \sigma_1^*, \sigma_2 \rangle_v) \leqslant \overline{Val}(v) + \varepsilon.$$

- Si $\varepsilon = 0$, on dit que la stratégie σ_i^* est *optimale*.

Dans le cadre de ce travail nous sommes particulièrement intéressés par un certain type de jeu Min-Max à somme nulle : les « reachability-price games ». Ces jeux sont des jeux à deux joueurs tels qu'un joueur tente d'atteindre son objectif le plus efficacement possible (*i.e.,* le poids du chemin pour atteindre son objectif doit être minimal) tandis que l'autre joueur désire l'en empêcher.

Définition 3.14 (Reachability-price game)**.** Soit $\mathcal{A} = (V, (V_{Min}, V_{Max}), E)$, soit $w : E \to \mathbb{R}$ une fonction de poids, un *"reachability-price game"* est un jeu Min-Max avec coût $\mathcal{G} = (\mathcal{A}, RP_{Min}, RP_{Max})$
avec un objectif donné $Goal \subseteq V$, où pour tout $\rho \in Plays$ tq $\rho = \rho_0\rho_1...$:

$$g := \begin{cases} \sum_{i=0}^{n-1} w(\rho_i, \rho_{i+1}) & \text{si } n \text{ est le plus petit indice tq } \rho_n \in Goal \\ +\infty & \text{sinon} \end{cases}$$

et $RP_{Min}(\rho) = RP_{Max}(\rho) = g$.
Ce jeu est un jeu à somme nulle et nous le notons : $\mathcal{G} = (\mathcal{A}, g, Goal)$.

Illustrons sur ce type de jeu, les notions définies précédemment.

Exemple 3.3. Soit $\mathcal{G} = (V, (V_{Min}, V_{Max}), E, g, Goal)$ le « *reachability-price game* » décrit par la figure 9 où les sommets contrôlés par J_{Min} sont les sommets ronds et les sommets contrôlés par J_{Max} sont les sommets carrés et $Goal = \{v_3\}$ et $V_{Min} = \{v_0, v_1, v_3, v_4\}$, $V_{Max} = \{v_2\}$ et où la fonction de poids w est représentée sur le graphe.

Dans un premier temps, montrons que $\mathcal{G}$ est déterminé. Nous avons : $\underline{Val}(v_0) = \overline{Val}(v_0) = 7$, $\underline{Val}(v_1) = \overline{Val}(v_1) = 6$, $\underline{Val}(v_2) = \overline{Val}(v_2) = 5$, $\underline{Val}(v_3) = \overline{Val}(v_3) = 0$, $\underline{Val}(v_4) = \overline{Val}(v_4) = 1$.

Ensuite, exhibons une stratégie optimale pour le joueur *Min* et une stratégie optimale pour le joueur *Max*.

Prenons la stratégie sans mémoire de J_{Min} définie comme suit :
$$\sigma_1(v) = \begin{cases} v_0 & \text{si } v = v_3 \\ v_1 & \text{si } v = v_0 \\ v_2 & \text{si } v = v_1 \\ v_3 & \text{si } v = v_4 \end{cases}.$$

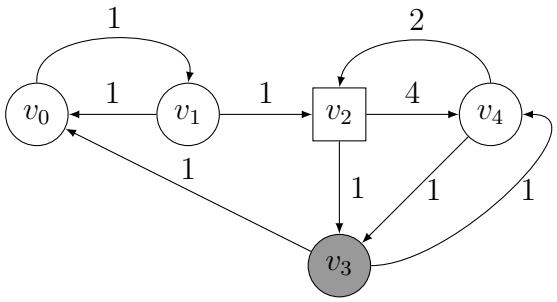


FIGURE 9 – reachability-price game

Pour J_{Max}, considérons la stratégie sans mémoire suivante :

$$\sigma_2(v) = v_4 \text{ si } v = v_2.$$

Nous savons que ce type de jeux est déterminé et possède des stratégies optimales sans mémoire.

Théorème 3.1 ([1])**.** Les « *reachability-price games* » tels que la fonction de poids associée au graphe est positive sont déterminés et ont des stratégies optimales sans mémoire.

De ce résultat découle une propriété sur la valeur des noeuds du graphe.

Propriété 3.5. Soit $\mathcal{A} = (V, (V_{Min}, V_{Max}), E)$ une arène, $\mathcal{G} = (\mathcal{A}, g, Goal)$ un « reachabilty-price game » et σ_1^* (resp. σ_2^*) une stratégie optimale pour J_{Min} (resp. J_{Max}) alors pour tout $v \in V$:

$$Val(v) = g(\langle \sigma_1^*, \sigma_2^* \rangle_v)$$

Preuve. Puisque σ_1^* et σ_2^* sont des stratégies optimales, nous savons que :

$$\forall v \in V \quad \sup_{\sigma_2 \in \Sigma_{Max}} g(\langle \sigma_1^*, \sigma_2 \rangle_v) \leqslant \overline{Val}(v) \tag{3}$$

$$\forall v \in V \quad \inf_{\sigma_1 \in \Sigma_{Min}} g(\langle \sigma_1, \sigma_2^* \rangle_v) \geqslant \underline{Val}(v) \tag{4}$$

Soit $v \in V$, comme le jeu est déterminé (*i.e.,* $\underline{Val}(v) = \overline{Val}(v)$), de (3) et (4) découle :

$$g(\langle \sigma_1^*, \sigma_2^* \rangle_v) \leqslant \sup_{\sigma_2 \in \Sigma_{Max}} g(\langle \sigma_1^*, \sigma_2^* \rangle_v) \leqslant Val(v) \tag{5}$$

et

$$Val(v) \leqslant \inf_{\sigma_1 \in \Sigma_{Min}} g(\langle \sigma_1, \sigma_2^* \rangle_v) \leqslant g(\langle \sigma_1^*, \sigma_2^* \rangle_v) \tag{6}$$

De (5) et (6), nous pouvons donc conclure :

$$Val(v) = g(\langle \sigma_1^*, \sigma_2^* \rangle_v)$$

□

3.2.2 Algorithme de Dijkstra pour les jeux Min-Max

Au vu du résultat 3.1 précédent, comme dans le cas des jeux d'atteignabilité à objectif qualitatif, nous nous interrogeons quant à la façon d'implémenter un algorithme pour résoudre les « *reachability-price games* ». L'objectif de cet algorithme est de trouver une stratégie optimale pour chaque joueur ainsi que la valeur associée à chaque sommet du graphe. L'idée est la suivante : le joueur *Min* a pour but d'emprunter un **plus court chemin** possible allant d'un noeud initial v_0 vers un noeud $v \in Goal$. On trouve dans la littérature différents algorithmes qui résolvent les problèmes de plus court chemin dans les graphes orientés. Toutefois, il n'y a pas de deuxième joueur qui agit de manière **antagoniste** face au joueur *Min* qui entre en compte dans ces algorithmes. C'est pourquoi, nous avons tenté une adaptation de l'algorithme de *Dijkstra* (que nous rappelons dans l'annexe A (p.74)). Après avoir pris connaissance du travail de Simon Olbregts [5], une comparaison et une refactorisation de notre algorithme basées sur le pseudo-code qu'il a lui-même proposé ont été effectuées.

De manière similaire à l'algorithme de Dijkstra, la fonction de coût associée au graphe doit être de la forme $w : E \to \mathbb{R}^+$. Le but de l'algorithme est de calculer le paiement minimum que le joueur *Min* peut garantir quelle que soit la stratégie adoptée par le joueur *Max*. Du fait que le joueur *Max* joue de manière antagoniste par rapport au joueur *Min*, à chaque fois que c'est au joueur *Max* de prendre une décision, il voudra maximiser son gain. Pour ce faire, on aura besoin de connaître le chemin le plus coûteux allant du sommet du joueur *Max* en cours de traitement vers un certain état objectif $o \in Goal$. Dès lors, contrairement à l'algorithme classique de Dijkstra qui part d'une source, l'algorithme s'exécutera à rebours à partir des états $o \in Goal$. Ci-dessous, nous reprenons les idées essentielles de l'algorithme proposé ainsi que son pseudo-code.

Idées de l'algorithme

- A tout sommet $v \in V$, on associe une valeur d qui représente l'estimation de la valeur $Val(v)$ (qui existe par le théorème 3.1). Cette valeur est mise à jour en cours d'exécution de l'algorithme de sorte qu'à la fin de celle-ci, on ait pour tout $v \in V$ $Val(v) = d$. Comme pour l'algorithme de Dijkstra, on initialise la valeur des sommets à $+\infty$ sauf pour les sommets objectifs $o \in Goal$ pour lesquels on initialise la valeur à 0.
- De plus, pour tout sommet $v \in V$, on associe un *tas* S qui est mis à jour en cours d'exécution. En effet, si $v \in V_{Max}$, à chaque fois qu'un des successeurs s de v est relaxé (*i.e.,* dont on a fini le traitement), on ajoute un couple (val, s) au tas. La valeur val est calculée de la manière suivante : $val = w(v, s) + s.d$. La preuve d'exactitude montre qu'en fait, $s.d = Val(s)$ car le noeud s a été complètement traité lorsque l'on rajoute (val, s) dans le tas correspondant au noeud v. Dans le cas où $v \in V_{Min}$, seule la valeur val minimale donnée par le calcul explicité précédemment est importante. On ne garde donc pas trace de toutes les valeurs calculées à chaque fois qu'un successeur est relaxé mais uniquement de la minimale.
- Pour tout sommet $v \in V_{Max}$, on associe le nombre de successeurs que ce sommet possède. On note ce nombre : $nbrSucc$. De plus, si l'on désire reconstituer la stratégie optimale pour le joueur *Max*, on stocke dans une structure de données adéquate la liste des successeurs déjà testés pour la recherche du chemin le plus long.
- On utilise un *tas* Q qui permet de stocker les sommets classés par leur valeur d. Sur ce tas, on peut effectuer les opérations suivantes : insertion d'un élément, extraction d'un élément ayant la clef de la plus petite valeur, lecture de l'élément ayant la clef de la plus petite valeur, test de la présence ou non d'élément dans Q, augmentation ou diminution de la valeur de la clef associée à un sommet. A l'initialisation de l'algorithme, les sommets présents dans Q sont tous ceux présents dans V.
- On maintient $T \subseteq V$ un ensemble de sommets qui vérifient la propriété suivante : pour tout sommet $v \in V$ $Val(v) = d$. Il s'agit donc de l'ensemble des sommets dont le traitement est terminé. A l'initialisation de l'algorithme $T = \emptyset$.
- Tant que $Q \neq \emptyset$, l'algorithme procède de la manière suivante :

  (a) On regarde la valeur du minimum de Q et le sommet s associé.

(b) Si cette valeur est $+\infty$, alors cela signifie qu'on a fini de traiter tous les sommets qui pouvaient atteindre un sommet objectif. On ajoute donc tous les sommets restants de Q dans T.

(c) S'il s'agit d'un sommet du joueur *Min* ou d'un sommet objectif o, alors on extrait le minimum de Q et on l'ajoute à T, on *relaxe* tous les arcs entrants de s. Il en va de même si le sommet est un sommet de joueur *Max* et que la valeur de $nbrSucc$ est égale à 1. En effet, cela signifie qu'il n'a pas le choix du chemin à emprunter.

(d) S'il s'agit d'un sommet du joueur *Max* tel que la valeur de $nbrSucc$ n'est pas égale à 1 alors, alors *Max* a plusieurs choix de chemin. Pour maximiser son gain, il ne choisira pas de passer par le successeur s tel que la valeur de $w(v,s) + Val(s)$ est la plus petite. On retire donc la plus petite valeur de S, on met à jour la clef de s dans Q et on décrémente $nbrSucc$. Un tel arc (v,s) est alors considéré comme « bloqué ». Ainsi, on sait qu'il y a un successeur de moins à traiter et on ajoute ce successeur à l'ensemble des successeurs déjà traités. En effet, on désire que la dernière valeur restante dans $s.S$ soit celle qui assure la maximisation du gain de *Max*.

- La *relaxation* des arcs entrants de s consiste en la méthode suivante : pour tout $(p,s) \in E$, on lit le minimum de $s.S$ (qui est en fait $Val(s)$), on calcule la nouvelle estimation de la valeur du sommet p. Si $p \in V_{Max}$, on insère cette valeur associée à s dans $p.S$. De plus dans tous les cas, si cette nouvelle valeur est plus petite que $p.d$, on met à jour $p.d$ dans Q et si $p \in V_{Min}$, on met à jour la valeur de la racine du tas $p.S$.

Pseudo-code

Soient $\mathcal{A} = (V, (V_{Min}, V_{Max}), E)$ une arène et $\mathcal{G} = (\mathcal{A}, g, Goal)$ un « *reachability-price game* ». Nous utilisons les notations suivantes :

- Q et S sont les tas décrits ci-dessus.
- $s.S$ correspond au tas S du sommet s.
- T est le sous-ensemble de V décrit ci-dessus.
- Pour tout $v \in V$, $Pred(v)$ est l'ensemble des prédécesseurs de v.
- Pour tout $v \in V$, $v.d$ est l'estimation de $Val(v)$.
- Pour tout $v \in V_{Max}$, $v.nbrSucc$ est le nombre de successeurs de v qu'il reste à tester, $v.t$ est l'ensemble des successeurs de v déjà testés.

L'algorithme en lui même est explicité par l'algorithme 3 et les algorithmes 4,5,6,7 sont appelés au sein de l'algorithme 3. Comme pour l'algorithme de Dijkstra, pour une meilleure complexité les files de priorité sont supposées implémentées par une structure de données telle que chaque opération EXTRAIRE-MIN() ainsi que chaque insertion et suppression d'un élément sont en $\mathcal{O}(\log_2 V)$ ainsi que chaque DÉCRÉMENTERCLEF($Clef(v), nouvVal$). De plus, LIRE-MIN() doit être implémenté en $\mathcal{O}(1)$. Ici, nous avons choisi d'utiliser des tas.

Algorithme 3 DIJKSTRAMINMAX(G,w,Goal)

ENTRÉE(S): $G = (V, E)$ un graphe orienté pondéré où E est représenté par la liste des prédécesseurs de chaque noeud, $w : E \to \mathbb{R}^+$ une fonction de poids, *Goal* l'ensemble des sommets objectifs.

SORTIE(S): / **EFFET(S) DE BORD :** Calcule pour chaque noeud la valeur de ce noeud.

```
1: Q ←INITIALISER-Q(G, Goal)
2: S ←INITIALISER-S(G, Goal)
3: T ← ∅
4: tant que Q ≠ ∅ faire
5:    s ← Q.LIRE-MIN()
6:    (val, succ) ← s.S.LIRE-MIN()
7:    si val = +∞ alors
8:       u ← Q.EXTRAIRE-MIN()
9:       T.INSÉRER(u)
10:   sinon
11:      si (s ∈ V_Min ∪ {Goal}) ou (s.nbrSucc = 1) alors
12:         s ← Q.EXTRAIRE-MIN()
13:         T.INSÉRER(s)
14:         pour tout p ∈ Pred(s) faire
15:            RELAXER(p, s, w)
16:         fin pour
17:      sinon
18:         BLOQUER-MAX(s)
19:      fin si
20:   fin si
21: fin tant que
```

Algorithme 4 INITIALISER-S($G, Goal$)

ENTRÉE(S): G un graphe orienté pondéré et l'ensemble des objectifs $Goal$.
SORTIE(S): / **EFFET(S) DE BORD :** initialise les files de priorité de tous les sommets du joueur Max ainsi que le champ correspondant à la plus petite valeur calculée pour le joueur Min.

1: **pour tout** $o \in Goal$ **faire**
2: $S \leftarrow$ nouveau tas
3: $o.S \leftarrow S$
4: $o.S$.INSÉRER$(0, NULL)$
5: **fin pour**
6: **pour tout** $v \in G.V_1 \backslash Goal$ **faire**
7: $v.S \leftarrow (+\infty, NULL)$
8: **fin pour**
9: **pour tout** $v \in G.V_2 \backslash Goal$ **faire**
10: $S \leftarrow$ nouveau tas
11: $v.S \leftarrow S$
12: $v.S$.INSÉRER$(+\infty, NULL)$
13: **fin pour**

Algorithme 5 INITIALISER-Q($G, Goal$)

ENTRÉE(S): G un graphe orienté pondéré, l'ensemble des états objectifs $Goal$.
SORTIE(S): Un tas Q qui comprend tous les sommets de V classés par leur valeur d.

1: $Q \leftarrow$ nouveau tas
2: **pour tout** $v \in G.V \backslash Goal$ **faire**
3: $v.d \leftarrow +\infty$
4: Q.INSÉRER(v)
5: **fin pour**
6: **pour tout** $o \in Goal$ **faire**
7: $o.d \leftarrow 0$
8: Q.INSÉRER(o)
9: **fin pour**
10: **retourner** Q

Algorithme 6 RELAXER(p, s, w)

ENTRÉE(S): deux sommets p et s, une fonction de poids $w : E \to \mathbb{R}^+$.

SORTIE(S): / **EFFET(S) DE BORD :** Ajoute à p.S une valeur pour p et son successeur dans le cas où p est un noeud du joueur *Max* et met à jour la valeur de la racine sinon.

```
1: (sVal, succ) ← LIRE-MIN(s.S)
2: pVal ← w(p, s) + sVal
3: old ← p.S.LIRE-MIN()
4: si pVal < old alors
5:    Q.DÉCRÉMENTER-CLEF(p, pVal)
6:    si p ∈ V_Min alors
7:       p.S ← (pVal, s)
8:    fin si
9: fin si
10: si p ∈ V_Max alors
11:    p.S.INSÉRER((pVal, s))
12: fin si
```

Algorithme 7 BLOQUER-MAX(s)

ENTRÉE(S): un sommet s appartenant au joueur *Max* tel que $s.nbrSucc \neq 1$.

SORTIE(S): / **EFFET(S) DE BORD :** traite le sommet s en supprimant la plus petite valeur de s.S et met à jour la liste t de successeurs de s déjà testés.

```
1: (val, succ) ← s.S.EXTRAIRE-MIN()
2: (nouvVal, nouvSucc) ← s.S.LIRE-MIN()
3: Q.INCRÉMENTER-CLEF(s, nouvVal)
4: s.nbrSucc ← s.nbrSucc − 1
5: s.t.INSÉRER(succ)
```

On remarque que l'algorithme DijkstraMinMax est un algorithme à effet de bord mais ne renvoie aucune valeur. Pour récupérer la valeur de chaque sommet s, ainsi qu'une stratégie optimale pour J_{Min} et une strategie optimale pour J_{max}, il suffit de récupérer la racine de $s.S$ qui comprend $Val(s)$ ainsi que le successeur qu'il faut emprunter pour obtenir cette valeur. On obtient alors les algorithmes RécupérerStratégies (qui récupère une stratégie optimale pour chaque joueur) et RécupérerValeurs qui récupère la valeur de chaque noeud).

Remarque 3.5. Dans le cas où $Val(v) = +\infty$, nous n'avons pas de successeur à notre disposition dans la file de priorité. Comme les fonctions de stratégie sont des fonctions totales, il faut toutefois définir $\sigma_i(v)$.

$1^{er} cas$: Si $v \in V_{Min}$ alors cela signifie que pour tout $s \in V$ tq $(v, s) \in E$, on a : $Val(s) = +\infty$. En effet, s'il existe $s' \in V$ tq $(v, s') \in E$ et $Val(v') < +\infty$ alors, J_{Min} a tout intérêt à jouer $\sigma_{Min}(v) = s'$.

$2^{eme} cas$: Si $v \in V_{Max}$ alors, cela signifie qu'il existe $s \in V$ tq $(v, s) \in E$ tq : $Val(s) = +\infty$. Sinon, pour tout $s' \in V$ tq $(v, s') \in E$ et $Val(v') < +\infty$ alors, $Val(v) \neq +\infty$.

Algorithme 8 RécupérerStratégies(G)

```
1: Afficher("Stratégie optimale pour $J_{Max}$")
2: pour tout $v \in V_{Max}$ faire
3:    si $Val(v) = +\infty$ alors
4:       Choisir $s \in Succ(v) \backslash v.t$
5:       Afficher( $v$" → " $s$)
6:    sinon
7:       $(val, succ) \leftarrow v.S$.Lire-Min()
8:       Afficher( $v$" → " $succ$)
9:    fin si
10: fin pour
11: Afficher("Stratégie optimale pour $J_{Min}$")
12: pour tout $v \in V_{Min}$ faire
13:    si $Val(v) = +\infty$ alors
14:       Choisir $s \in Succ(v)$
15:       Afficher( $v$" → " $s$)
16:    sinon
17:       $(val, succ) \leftarrow v.S$
18:       Afficher( $v$"→" $succ$)
19:    fin si
20: fin pour
```

Algorithme 9 RécupérerValeurs(G)

```
1: pour tout $v \in V_2$ faire
2:    $(val, succ) \leftarrow v.S$.Lire-Min()
3:    Afficher( "La valeur de" $v$ "est " $val$)
4: fin pour
5: pour tout $v \in V_1$ faire
6:    $(val, succ) \leftarrow v.S$
7:    Afficher( "La valeur de" $v$ "est " $val$)
8: fin pour
```

Exemple 3.4. Soient $\mathcal{A} = (V, (V_{Min}, V_{Max}), E)$, $w : E \to \mathbb{N}_0$ une fonction de poids et $\mathcal{G} = (\mathcal{A}, g, Goal)$ le « *reachability-price game* » associé, l'arène et la fonction de poids sont représentées sur le graphe de la figure 10 et V_{Min} (resp. V_{Max}) est représenté par les sommets ronds (resp. les sommets carrés). $Goal = \{v_0\}$. Pour les figures [9-15] , les états grisés représentent les états entièrement traités (*i.e.,* les états dans T) et à l'intérieur de ceux-ci se trouve la valeur associée à l'état. De plus, le tableau 1 (p. 33) reprend pour chaque étape et chaque noeud le contenu du champ S (une file de priorité pour le joueur Max et un 2-uplet pour le joueur Min).

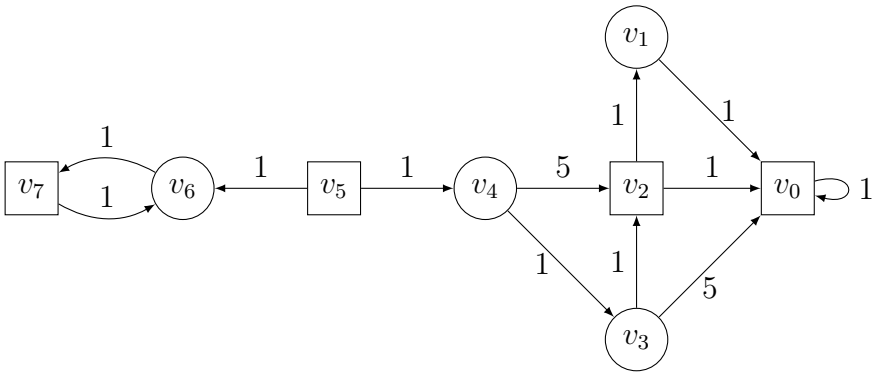


FIGURE 10 – Arène du « reachability-price game »

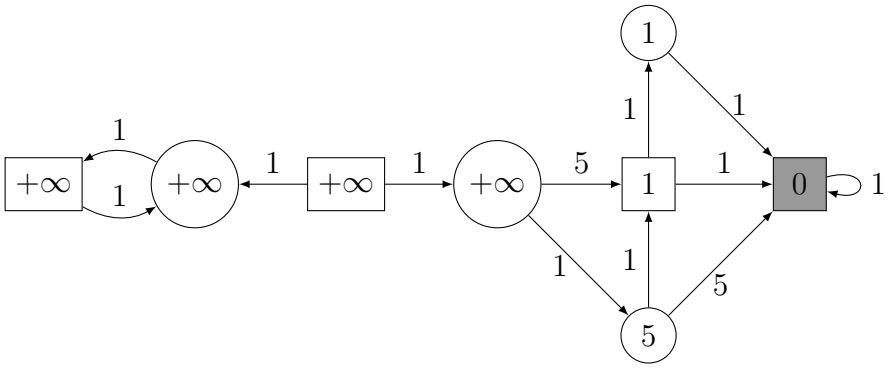


FIGURE 11 – Première étape – Traitement du noeud v_0. On relaxe :(v_1, v_0), (v_2, v_0), (v_3, v_0) et (v_{0z}, v_0). On grise v_0.

TABLE 1 – Données stockées dans le champ S pour chaque sommet et chaque étape de l'algorithme

	Sommets			
	v_0	v_1	v_2	v_3
Etape 1 :	$(0, null)$	$(1, v_0)$	$(+\infty, null), (1, v_0)$	$(5, v_0)$
Etape 2 :	$(0, null)$	$(1, v_0)$	$(+\infty, null)$	$(5, v_0)$
Etape 3 :	$(0, null)$	$(1, v_0)$	$(+\infty, null), (2, v_1)$	$(5, v_0)$
Etape 4 :	$(0, null)$	$(1, v_0)$	$(+\infty, null), (2, v_1)$	$(3, v_2)$
Etape 5 :	$(0, null)$	$(1, v_0)$	$(+\infty, null), (2, v_1)$	$(3, v_2)$
Etape 6 :	$(0, null)$	$(1, v_0)$	$(+\infty, null), (2, v_1)$	$(3, v_2)$
Etape 7 :	$(0, null)$	$(1, v_0)$	$(+\infty, null), (2, v_1)$	$(3, v_2)$
	v_4	v_5	v_6	v_7
Etape 1 :	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$
Etape 2 :	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$
Etape 3 :	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$
Etape 4 :	$(7, v_2)$	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$
Etape 5 :	$(4, v_3)$	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$
Etape 6 :	$(4, v_3)$	$(+\infty, null), (5, v_4)$	$(+\infty, null)$	$(+\infty, null)$
Etape 7 :	$(4, v_3)$	$(+\infty, null)$	$(+\infty, null)$	$(+\infty, null)$

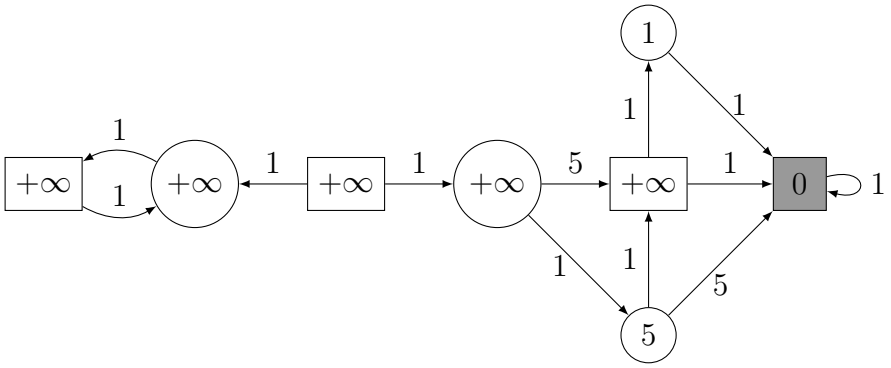


FIGURE 12 – Deuxième étape – Traitement du noeud v_2. $nbrSucc = 2$, on n'a donc pas encore testé tous les chemins possibles à partir de ce noeud. Décrémentation de $nbrSucc$, retrait de la plus petit valeur de S et ajout de v_0 dans les successeurs déjà testés.

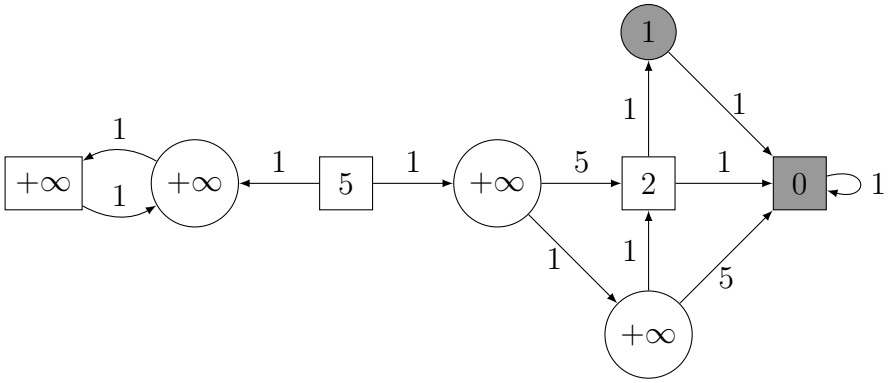


FIGURE 13 – Troisième étape – Traitement du noeud v_1. On relaxe (v_2, v_1). On grise v_1.

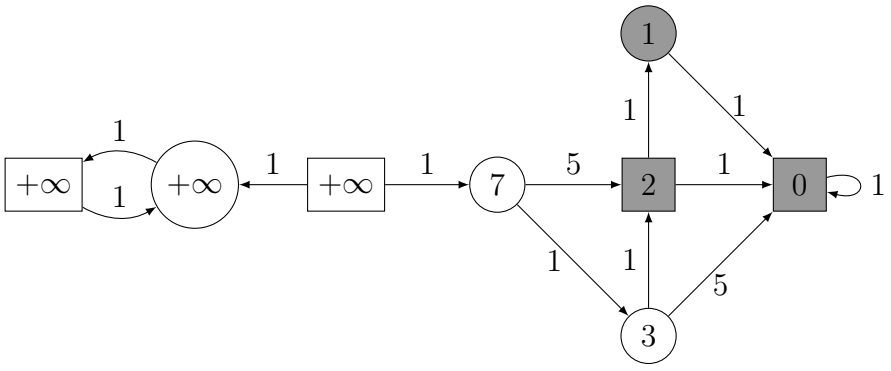


FIGURE 14 – Quatrième étape – Traitement du noeud v_2. $nbrSucc = 1$. On relaxe (v_4, v_2) et (v_3, v_2). On grise v_2.

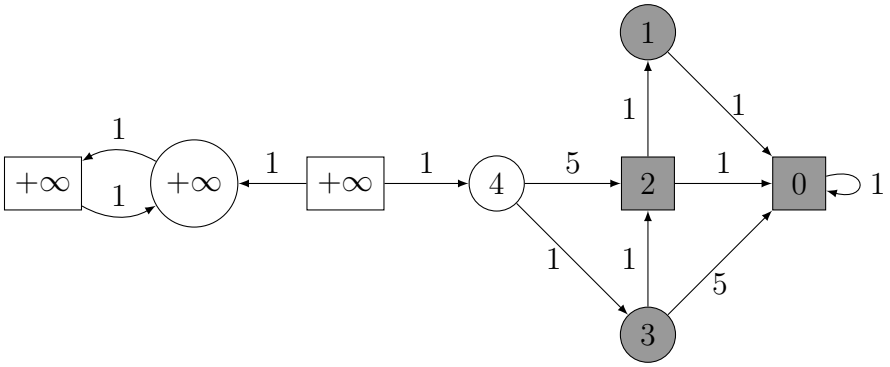


FIGURE 15 – Cinquième étape – Traitement du noeud v_3. On relaxe (v_4, v_3). On grise v_3.

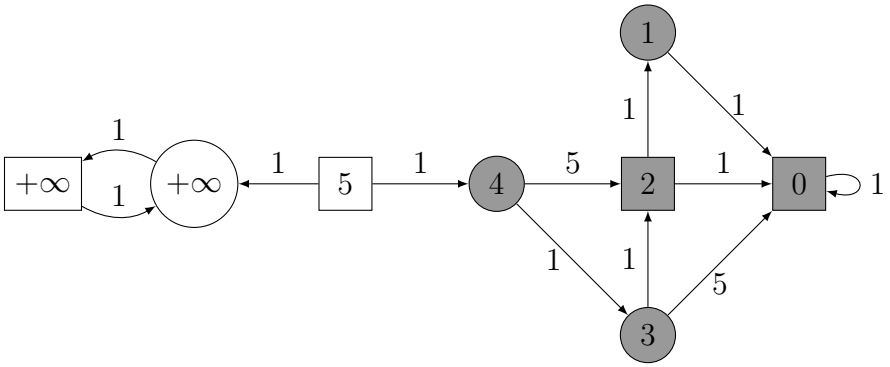


FIGURE 16 – Sixième étape – Traitement du noeud v_4. On relaxe (v_5, v_4). On grise v_4.

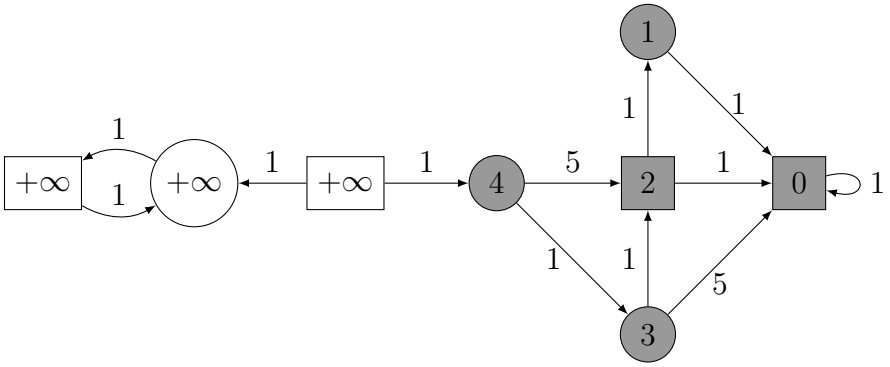


FIGURE 17 – Dernière étape – Traitement du noeud v_4. $nbrSucc = 2$.On retire la plus petite valeur de S.

Exactitude de l'algorithme DijkstraMinMax

Notre but maintenant est de montrer que l'algorithme DijkstraMinMax est exact. C'est-à-dire, que nous voulons montrer qu'à la fin de l'exécution de cet algorithme, nous connaissons $Val(v)$ pour tout $v \in V$ ainsi qu'une stratégie optimale pour le joueur Min et pour le joueur Max. Pour ce faire, nous nous inspirons de la preuve effectuée par Simon Olbregts dans le cadre de son projet de 1er Master en sciences informatiques [5]. Pour plus de clarté et de concision, nous avons besoin d'introduire quelques notations :

- Soient x et y des noeuds du graphe, $x \rightsquigarrow y$ représente un chemin x à y qui suit des stratégies optimales pour le joueur Min et le joueur Max (*i.e.,*il s'agit d'un *chemin optimal*). Par abus de notation, pour $x \rightsquigarrow y = v_0...v_n$ on définit $g(x \rightsquigarrow y) = \sum_{i=0}^{n-1} w(v_i, v_{i+1})$.
- T_i est l'ensemble des noeuds traités après la i$^{\text{ème}}$ itération. Un noeud est considéré comme traité quand il est enlevé de Q.
- E_i est l'ensemble des arcs du graphe G duquel on a enlevé tous ceux qu'on ne choisit pas d'emprunter car ils sortent d'un noeud Max ayant été sélectionné lors d'une itération $j \leqslant i$ et qu'il existe un meilleur choix pour le joueur Max à partir de ce noeud. Un tel arc est appelé un *arc bloqué*.
- $\rho_i(v) := \min\limits_{v' \in T_i \,|\, (v,v') \in E_i} \{w(v, v') + Val(v')\}$

Nous allons donc montrer que les invariants suivants sont exacts :

1. $\forall v \in T_i$ on connait la valeur $Val(v)$ et la stratégie à adopter en v.
2. $\forall v \in V \backslash T_i$, on connait $\rho_i(v)$.

A. Avant la première itération

Nous avons : $T_i = \emptyset$ et $E_i = E$ à l'initialisation de l'algorithme. Donc pour tout $v \in V$, nous connaissons $\rho_i(v)$. En effet, si v est un état objectif ($v \in Goal$) alors $\rho_i(v) = 0$ sinon, $\rho_i(v) = +\infty$.

B. En cours d'exécution

Au préalable, afin de prouver que le premier invariant est correct, nous allons établir la véracité de plusieurs propriétés dont voici les énoncés ainsi que leur preuve.

Propriété 3.6. Pour tout $v \in V_{Min}$, $Val(v) = \min_{(v,v') \in E}\{w(v, v') + Val(v')\}$

Preuve. Soit $v \in V_{Min}$, pour prouver cette propriété nous allons procéder en deux étapes. Une première étape montre le résultat suivant :

$$\exists v' \in V \text{ tq } (v, v') \in E \text{ et } Val(v) = w(v, v') + Val(v').$$

La seconde étape prouve notre propriété à l'aide de la première étape.

Première étape

Par la propriété 3.5, nous savons que $Val(v) = g(\langle \sigma_1^*, \sigma_2^* \rangle_v)$. Ce que nous pouvons réécrire :

$$\begin{aligned} Val(v) &= g(v.\langle \sigma_1^*, \sigma_2^* \rangle_{\sigma_1^*(v)}) \\ &= w(v, \sigma_1^*(v)) + g(\langle \sigma_1^*, \sigma_2^* \rangle_{\sigma_1^*(v)}) \\ &= w(v, \sigma_1^*(v)) + Val(\sigma_1^*(v)) \end{aligned}$$

En prenant, $v' = \sigma_1^*(v)$ nous retrouvons bien le résultat attendu.

Deuxième étape

Montrons que le v' tel que le minimum est atteint correspond au v' que nous avons exhibé à l'étape précédente. Procédons par une preuve par l'absurde. En supposant le contraire, il existe $u \in V$ tel que $(v, u) \in E$ et

$$\begin{aligned} Val(v) &= w(v, v') + Val(v') \\ &> w(v, u) + Val(u) \\ &= w(v, u) + g(\langle \sigma_1^*, \sigma_2^* \rangle_u) \end{aligned} \tag{1}$$

Si on considère la stratégie du joueur Min suivante :

$$\lambda_1(w) = \begin{cases} u & \text{si } w = v \\ \sigma_1^*(w) & \text{sinon} \end{cases}$$

alors nous pouvons réécrire l'équation (1). Nous obtenons :

$$g(\langle \sigma_1^*, \sigma_2^* \rangle_v) > g(\langle \lambda_1, \sigma_2^* \rangle_v).$$

Ce qui contredit le fait que σ_1^* soit une stratégie optimale pour le joueur Min.

□

Nous pouvons établir un résultat similaire dans le cas où $v \in V_{Max}$. La preuve étant similaire, elle n'est pas rédigée dans le présent document.

Propriété 3.7. Pour tout $v \in V_{Max}, Val(v) = \max_{(v,v')\in E}\{w(v,v')+Val(v')\}$

Propriété 3.8. Soit i une certaine étape de l'algorithme et $v \in V\backslash T_i$ tel que v se trouve sur un chemin optimal et que le successeur de v sur ce chemin, notons-le u, ait déjà été traité (*i.e.,* $u \in T_i$) alors :

$$v \in V_{Min} \Rightarrow \rho_i(v) = Val(v).$$

Preuve.
Comme nous avons pris v et u sur un chemin optimal cela signifie qu'il existe $(\sigma_1^*, \sigma_2^*) \in \Sigma_{Min} \times \Sigma_{Max}$ des stratégies optimales telles que pour un certain $t \in V$ on ait : $\langle \sigma_1^*, \sigma_2^* \rangle_t = t \dots vu \dots$. Nous voulons montrer que si on connait $Val(u)$ alors on a : $Val(v) = \rho_i(v)$. Nous savons, vu que σ_1^* et σ_2^* sont des stratégies optimales, que :

$$\begin{aligned} Val(v) &= g(\langle \sigma_1^*, \sigma_2^* \rangle_v) \\ &= w(v,u) + g(\langle \sigma_1^*, \sigma_2^* \rangle_u) \\ &= w(v,u) + Val(u) \end{aligned}$$

De plus, la propriété 3.6 établit :

$$Val(v) = \min_{(v,v')\in E}\{w(v,v') + Val(v')\}.$$

Donc u réalise ce minimum.
Nous pouvons maintenant conclure. En effet,

$$\begin{aligned} \rho_i(v) &= \min_{v'\in T_i | (v,v')\in E_i}\{w(v,v') + Val(v')\} && \text{par définition} \\ &= \min_{(v,v')\in E}\{w(v,v') + Val(v')\} && u \in T_i \text{ et } (v,u) \in E_i \text{ car aucun arc issu d'un noeud Min n'est bloqué} \\ &= Val(v). \end{aligned}$$

□

Propriété 3.9. Soit i une certaine étape de l'algorithme et $v \in V\backslash T_i$ tel que v se trouve sur un chemin optimal et que le successeur de v sur ce chemin, notons-le u, ait déjà été traité (*i.e.,* $u \in T_i$) alors :

$$v \in V_{Max} \Rightarrow \rho_i(v) \leqslant Val(v).$$

Preuve.
Comme nous avons pris v et u sur un chemin optimal cela signifie qu'il existe $(\sigma_1^*, \sigma_2^*) \in \Sigma_{Min} \times \Sigma_{Max}$ des stratégies optimales telles que pour un certain $t \in V$ on ait : $\langle \sigma_1^*, \sigma_2^* \rangle_t = t \dots vu \dots$. Nous voulons montrer que si on connait $Val(u)$ alors on a : $Val(v) \geqslant \rho_i(v)$.

Supposons au contraire que $\rho_i(v) > Val(v)$. Nous avons alors :

$$\forall t \in T_i \text{ tq } (v,t) \in E_i \; w(v,t) + Val(t) > g(\langle \sigma_1^*, \sigma_2^* \rangle_v). \tag{2}$$

De cette relation nous pouvons conclure. En effet, cela signifie que depuis le noeud v le joueur Max possède une déviation profitable qui est d'emprunter un des arcs (v,t) de (2). Ceci contredit le fait que σ_2^* est une stratégie optimale.

□

Maintenant que ces propriétés sont établies, reprenons la preuve d'exactitude de l'algorithme. Supposons que les invariants sont vérifiés jusqu'à l'étape i et montrons qu'ils le sont toujours pour l'étape $i+1$. Commençons par le montrer pour le premier invariant.

Soit u le noeud lu dans Q (ligne 5 de `DikjstraMinMax`). Comme il s'agit d'un minimum nous avons : $\forall v \in V \backslash T_i$, $\rho_i(u) \leqslant \rho_i(v)$. Plusieurs cas sont alors possibles.

1. $\rho_i(u) = +\infty$: Dans ce cas pour tous les noeuds $v \in V \backslash T_i$ (*i.e.,* ceux restants dans Q) $\rho_i(v) = +\infty$ car $\rho_i(u) = \min_{v \in V \backslash T_i}\{\rho_i(v)\}$. S'il s'agit d'un noeud du joueur Max cela signifie qu'il peut choisir un arc tel qu'après cet arc il est assuré que le joueur Min ne pourra pas atteindre un de ses objectifs. S'il s'agit d'un noeud du joueur Min cela signifie qu'à partir de ce noeud il ne pourra jamais atteindre un de ses états objectifs. La stratégie optimale consiste donc à choisir n'importe quel arc dans ceux restants dans E_i.
2. $\rho_i(u) < +\infty$ et $u \in Goal$: Comme il s'agit d'un noeud objectif, on doit avoir que $Val(u) = 0$ ce qui est bien le cas car lors de l'initialisation de l'algorithme la valeur associée à chaque état objectif est 0 et elle ne peut pas être modifiée. En effet, vu que le poids des arcs est positif, on ne peut pas décrémenter la valeur associée à un noeud objectif dans Q. La décrémentation de cette valeur se fait uniquement dans la méthode Bloquer-Max or, cette méthode n'est jamais appelée pour un noeud objectif.
3. $\rho_i(u) < +\infty$ et $u \in V_{Min} \backslash Goal$: Nous devons vérifier que nous pouvons calculer $Val(u)$ et donc que l'on pouvait bien rajouter u aux éléments traités. Nous devons donc nous assurer que $Val(u) = \rho_i(u)$.

Supposons au contraire que $Val(u) \neq \rho_i(u)$. Nous savons qu'il existe $(\sigma_1^*, \sigma_2^*) \in \Sigma_1 \times \Sigma_2$ des stratégies optimales. Dès lors on a : $Val(u) = g(\langle \sigma_1^*, \sigma_2^* \rangle_u)$. Soit σ_1 la stratégie du joueur Min définie de la manière suivante :

$$\sigma_1(v) = \begin{cases} \underset{v' \in T_i, (v,v') \in E_i}{\arg\min} \{w(v,v') + Val(v')\} & \text{si } v = u \\ \sigma_1^*(v) & \text{sinon} \end{cases}$$

Dès lors, nous avons :

$$Val(u) = \inf_{\tau_1 \in \Sigma_{Min}} g(\langle \tau_1, \sigma_2^* \rangle_u) \leqslant g(\langle \sigma_1, \sigma_2^* \rangle_u) = \rho_i(u)$$

Mais comme nous avons supposé que $Val(u) \neq \rho_i(u)$, il en découle :

$$g(\langle \sigma_1^*, \sigma_2^* \rangle_u) < \rho_i(u). \tag{3}$$

De plus, nous pouvons écrire : $\langle \sigma_1^*, \sigma_2^* \rangle_u = u \dots xy \dots$ où $x \in V \backslash T_i$ et $y \in T_i$.
L'équation (3) peut alors se réécrire :

$$g(u \rightsquigarrow x) + g(\langle \sigma_1^*, \sigma_2^* \rangle_x) < \rho_i(u).$$

Comme les poids sont positifs, l'inégalité suivante est également vérifiée :

$$g(\langle \sigma_1^*, \sigma_2^* \rangle_x) < \rho_i(u). \tag{4}$$

Une étude de l'équation (4) nous permet d'exhiber la contradiction souhaitée.

(a) **Si $\mathbf{x} \in \mathbf{V_{Min}}$** : Par la propriété 3.8,
$\rho_i(x) = Val(x) = g(\langle \sigma_1^*, \sigma_2^* \rangle_x) < \rho_i(u)$. Or, on a choisi u de telle manière que $\rho_i(u)$ soit minimal.

(b) **Si $\mathbf{x} \in \mathbf{V_{Max}}$** : Par la propriété 3.9,
$\rho_i(x) \leqslant Val(x) = g(\langle \sigma_1^*, \sigma_2^* \rangle_x) < \rho_i(u)$. Ce qui contredit également le choix de u.

(c) **Si x n'existe pas** : Par la propriété 3.8, $\rho_i(u) = Val(u) = +\infty$. Ce qu'on avait supposé faux.

4. $\rho_i(u) < +\infty$ et $u \in V_{Max} \backslash Goal$: Remarquons que dans ce cas, pour qu'un sommet soit ajouté à l'ensemble des sommets traités, il faut qu'il ne reste plus qu'un arc « non-bloqué » sortant du sommet u.

Les autres arcs ont déjà été traités et n'ont pas été retenus pour la construction d'un chemin optimal. Notons cet arc (u, u') . On alors :

$$\rho_i(u) = w(u, u') + Val(u') \tag{5}$$

Nous voulons montrer l'égalité suivante :

$$\rho_i(u) = Val(u).$$

Or, nous savons :

$$Val(u) = \sup_{\tau_2 \in \Sigma_2} g(\langle \sigma_1^*, \tau_2 \rangle_u) = \max_{v \in V | (u,v) \in E} \{w(u, v) + Val(v)\}.$$

Nous allons donc montrer :

$$\rho_i(u) = \max_{v \in V | (u,v) \in E} \{w(u, v) + Val(v)\}.$$

Pour ce faire il nous suffit de montrer que pour tout sommet $v \in V$ l'inégalité suivante est respectée.

$$w(u, v) + Val(v) \leqslant \rho_i(u) = w(u, u') + Val(u') \tag{6}$$

Soit $v \in V$ tel que $v \neq u'$ et $(u, v) \in E$, l'arc (u, v) a été « bloqué » lors d'une certaine itération $j < i$ dans ce cas nous connaissons les trois relations suivantes :

$$\rho_j(u) = w(u, v) + Val(v) \tag{7}$$

$$\rho_j(u) = \min_{v' \in T_j | (v,v') \in E_j} \{w(u, v') + Val(v')\}$$

$$\rho_j(u) \leqslant \rho_i(u)$$

Nous pouvons donc déjà conclure que si $u' \in T_j$ alors l'inégalité (6) est vérifiée. Intéressons-nous maintenant au cas où $u' \in V \backslash T_j$. Nous savons qu'il existe $(\sigma_1^*, \sigma_2^*) \in \Sigma_1 \times \Sigma_2$ telles que $Val(u') = g(\langle \sigma_1^*, \sigma_2^* \rangle_{u'})$. Soit $x \in \langle \sigma_1^*, \sigma_2^* \rangle_{u'}$ un noeud sur le chemin optimal partant de u' tel que $x \in V \backslash T_j$ et le successeur de x sur ce chemin optimal appartient à T_j. Alors :

$$Val(u') = g(\langle \sigma_1^*, \sigma_2^* \rangle_{u'}) = g(u' \rightsquigarrow x) + g(\langle \sigma_1^*, \sigma_2^* \rangle_x)$$

Dès lors comme les poids sur les arcs sont positifs :

$$Val(u') \geqslant g(\langle \sigma_1^*, \sigma_2^* \rangle_x)$$

Plusieurs cas sont alors à traiter :

(a) **Si $\mathbf{x \in V_{Min}}$** : Par la propriété 3.8, nous savons :

$$\rho_j(x) = Val(x) = g(\langle \sigma_1^*, \sigma_2^* \rangle_x) \leqslant Val(u'). \tag{8}$$

Nous pouvons dérouler la suite d'inégalités suivante qui nous permet de conclure :

$$\begin{aligned}
\rho_i(u) &= w(u,u') + Val(u') && \text{par (5)} \\
&\geqslant Val(u') && \text{le poids des arcs est positif} \\
&\geqslant \rho_j(x) && \text{par (8)} \\
&\geqslant \rho_j(u) && \text{à l'étape } j \text{ on traite } u \text{ donc } \rho_j(u) \text{ minimal.} \\
&= w(u,v) + Val(v) && \text{par (7)}
\end{aligned}$$

Ce qui prouve bien (6).

(b) **Si $\mathbf{x \in V_{Max}}$** : Par la propriété 3.9,
$\rho_j(x) \leqslant Val(x) = g(\langle \sigma_1^*, \sigma_2^* \rangle_x) \leqslant Val(u')$. Nous concluons par le même raisonnement que dans le cas précédant.

(c) **Si x n'existe pas** : Par la propriété 3.8,
$\rho_j(u') = Val(u') = +\infty$ et le raisonnement précédent est toujours valable.

S'il restait plusieurs successeurs possibles, alors dans ce cas, $T_{i+1} = T_i$ et le premier invariant est vérifié. Nous avons ainsi terminé la preuve concernant le premier invariant.

Pour le second invariant nous pouvons remarquer que pour tout $v \in V$ la valeur de $\rho_i(v)$ est susceptible d'être modifiée dans deux cas : lorsqu'un noeud est complètement traité et qu'une relaxation a lieu ou lorsque qu'un arc est bloqué. Dans un cas comme dans l'autre, soit on décrémente la valeur si nécessaire (dans la relaxation) soit on l'incrémente (lorsque l'on « bloque » un arc). Le second invariant est donc également vérifié.

Complexité de l'algorithme DijkstraMinMax

Posons $n = |V|$ et $m = |E|$.

Premièrement, nous rappelons les complexités dans le pire des cas des opérations sur les tas.

Opération	Complexité dans le pire cas
LIRE-MIN()	$\mathcal{O}(1)$
EXTRAIRE-MIN()	$\mathcal{O}(\log_2 n)$
INSÉRER(item)	$\mathcal{O}(\log_2 n)$
AUGMENTER-CLEF($clef, valeur$)	$\mathcal{O}(\log_2 n)$
DÉCRÉMENTER-CLEF($clef, valeur$)	$\mathcal{O}(\log_2 n)$

Deuxièmement, avant de nous intéresser à la complexité de l'algorithme principal 3 nous établissons la complexité des algorithmes secondaires.

1. INITIALISER-S($G, Goal$) : Pour chaque noeud on instancie un tas en $\mathcal{O}(1)$ et on initialise la racine de celui-ci en $\mathcal{O}(1)$. Comme il y a n noeuds, l'algorithme est en $\mathcal{O}(n)$.
2. INITIALISER-Q($G, Goal$) : On commence par instancier un tas Q en $\mathcal{O}(1)$. Ensuite, on rajoute une valeur pour chacun des noeuds dans Q. Rappelons que la complexité de l'insertion dans un tas est $\mathcal{O}$(hauteur du tas), dans ce cas, comme les valeurs sont rajoutées une à une, la complexité est en $\mathcal{O}(n)$.
3. RELAXER(p, s, w) : Les instructions [1–3] sont effectuées en $\mathcal{O}(1)$ tandis que celles [4–9] sont en $\mathcal{O}(\log_2 n)$ à cause de l'instruction DÉCRÉMENTER-CLEF. Enfin, les instructions [10–12] ont une complexité en $\mathcal{O}(\log_2 n)$. Dès lors, la complexité de cet algorithme secondaire est en $\mathcal{O}(\log_2 n)$.
4. BLOQUER-MAX(s) : Les instructions 1 et 3 sont en $\mathcal{O}(\log_2 n)$ tandis que les autres sont en $\mathcal{O}(1)$. Dès lors, la complexité de cet algorithme est en $\mathcal{O}(\log_2 n)$.

Nous pouvons maintenant clôturer l'analyse de l'algorithme DIJKSTRA-MINMAX en démontrant que sa complexité est en $\mathcal{O}((m+n)\log_2 n)$. Pour se faire une analyse minutieuse des instructions [4–21] est nécessaire. En effet, plutôt que de calculer la complexité du corps de la boucle « tant que » et ensuite le nombre de fois où l'on rentre dans cette boucle, nous préférons une analyse globale. Nous nous intéressons donc, pour chaque instruction, au nombre de fois où celle-ci est exécutée dans le pire cas et ce sur l'entièreté de l'exécution de l'algorithme.

- Pour les lignes 5 et 6 sont exécutées dans le pire cas n fois. Donc, la complexité globale de ces deux lignes est en $\mathcal{O}(n)$.
- Pour la condition « si » [7–9], la complexité globale est également en $\mathcal{O}(n \log_2 n)$.
- Pour la condition « sinon » [10–20], il est important de remarquer qu'un arc ne peut être relaxé qu'une seule fois. De ce fait, la complexité globale de l'instruction RELAXER(p, s, w) est en $\mathcal{O}(m \log_2 n)$.

Nous pouvons ainsi affirmer que le bloc d'instructions [11–16] est en $\mathcal{O}((m+n)\log_2 n)$ tandis que celui [17–19] est en $\mathcal{O}((m+n)\log_2 n)$. Ce qui nous fait une complexité globale pour toute la condition en $\mathcal{O}((m+n)\log_2 n)$

Comme l'initialisation des tas est en $\mathcal{O}(n)$, nous pouvons enfin conclure que l'algorithme DijkstraMinMax a une complexité en $\mathcal{O}((m+n)\log_2 n)$.

3.2.3 Jeux multijoueurs avec coût

Pour finir, nous nous intéressons à des jeux où chaque joueur possède une fonction de coût qui représente pour chaque partie du jeu le montant qu'il doit payer lorsque cette partie est jouée. Le but de chaque joueur est donc de **minimiser** sa fonction de coût.

De manière formelle, nous définissons les *jeux multijoueurs avec coût.*

Définition 3.15 (Jeu multijoueur avec coût)**.** Soit $\mathcal{A} = (\Pi, V, (V_i)_{i\in\Pi}, E)$ une arène, un *jeu multijoueur avec coût* est un tuple $\mathcal{G} = (\mathcal{A}, (Cost_i)_{i\in\Pi})$ où

- $\mathcal{A} = (\Pi, V, (V_i)_{i\in\Pi}, E)$ est l'arène d'un jeu sur graphe.
- $Cost_i : Plays \to \mathbb{R} \cup \{+\infty, -\infty\}$ est la *fonction de coût* de J_i.

Dans le cadre de ce projet, nous nous intéressons aux jeux sur graphe tels que l'objectif des joueurs est un objectif quantitatif. De plus, nous souhaitons que l'objectif des joueurs soit atteint le plus rapidement possible. Les fonctions de coût qui nous intéressent sont donc les suivantes :

Exemple 3.5 (Fonctions de coût)**.** Pour tout $\rho = \rho_0\rho_1\rho_2\ldots$ où $\rho \in Plays$ on définit :

1. $Cost_i(\rho) = \begin{cases} \min\{i|\rho_i \in Goal_i\} & \text{si } \exists i \text{ tq } \rho_i \in Goal_i \\ +\infty & \text{sinon} \end{cases}$

2. $\varphi_i(\rho) = \begin{cases} \sum_{i=0}^{n-1} w(\rho_i, \rho_{i+1}) & \text{si } n \text{ est le plus petit indice} \\ & \text{tq } \rho_n \in Goal_i \\ +\infty & \text{sinon} \end{cases}$

Remarque 3.6. L'exemple 1 est un cas particulier de l'exemple 2 si l'on prend $w(\rho_i, \rho_{i+1}) = 1$ pour tout i.

Nous pouvons maintenant définir les jeux qui nous intéressent particulièrement et sur lesquels nous portons maintenant notre attention.

Définition 3.16 (Jeu d'atteignabilité multijoueur à objectif quantitatif)**.** Un *jeu d'atteignabilité multijoueur à objectif quantitatif* est un jeu multijoueur avec coût $\mathcal{G} = (\Pi, V, (V_i)_{i\in\Pi}, E, (Cost_i)_{i\in\Pi})$ tel que pour tout joueur $i \in \Pi$ $Cost_i = \varphi_i$ pour un certain $Goal_i \subseteq V$.
On note ces jeux $\mathcal{G} = (\mathcal{A}, (\varphi_i)_{i\in\Pi}, (Goal_i)_{i\in\Pi})$.

Exemple 3.6 (Jeu d'atteignabilité multijoueur à objectif quantitatif)**.**
Soit un jeu $\mathcal{G} = (\Pi, V, (V_1, V_2), E, (\varphi_1, \varphi_2), (Goal_1, Goal_2))$ tel que $\Pi = \{1, 2\}$, $V_1 = \{v_0, v_1, v_3, v_4\}$, $V_2 = \{v_2\}$, $Goal_1 = \{v_3\}$, $Goal_2 = \{v_0\}$. Cet exemple est illustré par la figure 18.

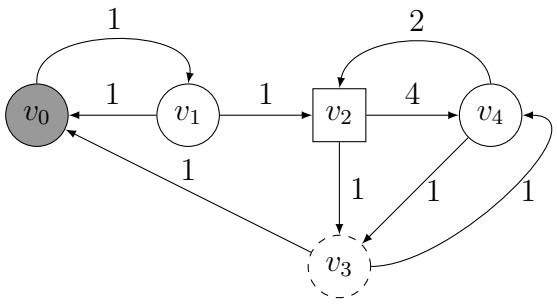


FIGURE 18 – Jeu d'atteignabilité multijoueur à objectif quantitatif

Pour les jeux d'atteignabilité avec coût et à objectif quantitatif, différents concepts de solutions sont étudiés. L'un de ceux-ci et qui est celui que nous abordons est l'*équilibre de Nash*[1]. Etudier les équilibres de Nash permet de prendre en compte le comportement rationnel et égoïste des joueurs. En d'autres termes, ils préfèrent gagner plutôt que perdre et ils ne se préoccupent pas des objectifs des autres joueurs. En connaissant les stratégies des autres joueurs, un tel joueur choisit donc la stratégie qui lui est personnellement la plus favorable.

Définition 3.17 (Equilibre de Nash)**.** Soit $(\mathcal{G}, v_0)$ un *jeu multijoueur avec coût et initialisé*, un profil de stratégie $(\sigma_i)_{i\in\Pi}$ est un *équilibre de Nash* dans $(\mathcal{G}, v_0)$ si, pour chaque joueur $j \in \Pi$ et pour chaque stratégie $\tilde{\sigma}_j$ du joueur j, on a :

$$Cost_j(\langle(\sigma_i)_{i\in\Pi}\rangle_{v_0}) \leqslant Cost_j(\langle\tilde{\sigma}_j, \sigma_{-j}\rangle_{v_0})$$

En fait, $(\sigma_i)_{i\in\Pi}$ est un équilibre de Nash si aucun joueur n'a d'intérêt à dévier de sa stratégie si tous les autres joueurs s'en tiennent à la leur. On dit alors qu'aucun joueur ne possède de *déviation profitable*.

1. D'autres concepts de solutions sont abordés dans la référence [3].

Définition 3.18 (Déviation profitable)**.** Soit $(\mathcal{G}, v_0)$ un *jeu multijoueur avec coût et initialisé*, soit $(\sigma_i)_{i\in\Pi}$ un profil de stratégie, $\tilde{\sigma_j}$ est une *déviation profitable* pour le joueur j relativement à $(\sigma_i)_{i\in\Pi}$ si :

$$Cost_j(\langle(\sigma_i)_{i\in\Pi}\rangle_{v_0}) > Cost_j(\langle\tilde{\sigma}_j, \sigma_{-j}\rangle_{v_0})$$

Exemple 3.7. Considérons le jeu décrit dans l'exemple 3.6 et donnons un exemple d'équilibre de Nash de ce jeu.

$$\text{Soit } \sigma_1(v) = \begin{cases} v_1 & \text{si } v = v_0 \\ v_2 & \text{si } v = v_1 \\ v_0 & \text{si } v = v_3 \\ v_3 & \text{si } v = v_4 \end{cases} \qquad \text{et} \qquad \sigma_2(v) = v_3 \text{ si } v = v_2$$

(rem : ces deux stratégies sont des stratégies sans mémoire), (σ_1, σ_2) est un équilibre de Nash de $(\mathcal{G}, v_0)$. De plus, soit $\rho = \langle\sigma_1, \sigma_2\rangle_{v_0}$ nous avons : $\varphi_1(\rho) = 3$ et $\varphi_2(\rho) = 0$.
Montrons que J_1 ne possède pas de déviation profitable. Nous devrions pour cela le montrer pour toutes les stratégies possibles pour J_1. Nous nous contentons de le montrer pour les stratégies sans mémoire.

$$\text{Considérons } \tilde{\sigma_1}(v) = \begin{cases} v_1 & \text{si } v = v_0 \\ v_0 & \text{si } v = v_1 \\ v_0 & \text{si } v = v_3 \\ v_3 & \text{si } v = v_4 \end{cases}.$$

Dans ce cas, nous avons : $\varphi_1(\langle\tilde{\sigma_1}, \sigma_2\rangle_{v_0}) = +\infty$. C'est en fait le cas pour toutes les déviations $\tilde{\sigma_1}$ telles que $\tilde{\sigma_1}(v_1) = v_0$. Ce ne sont donc pas des déviations profitables.

$$\text{Considérons maintenant } \tilde{\sigma_1}(v) = \begin{cases} v_1 & \text{si } v = v_0 \\ v_2 & \text{si } v = v_1 \\ v_0 & \text{si } v = v_3 \\ v_2 & \text{si } v = v_4 \end{cases}.$$

On a alors $\varphi_1(\langle\tilde{\sigma_1}, \sigma_2\rangle_{v_0}) = 3 = \varphi_1(\langle\sigma_1, \sigma_2\rangle_{v_0})$. Ce n'est donc pas non plus une déviation profitable. Les déviations ci-dessous ont également un coût de 3 et ne sont pas des déviations profitables.

$$\tilde{\sigma_1}(v) = \begin{cases} v_1 & \text{si } v = v_0 \\ v_2 & \text{si } v = v_1 \\ v_4 & \text{si } v = v_3 \\ v_3 & \text{si } v = v_4 \end{cases}. \qquad \tilde{\sigma_1}(v) = \begin{cases} v_1 & \text{si } v = v_0 \\ v_2 & \text{si } v = v_1 \\ v_4 & \text{si } v = v_3 \\ v_2 & \text{si } v = v_4 \end{cases}.$$

Nous avons ainsi testé toutes les déviations sans mémoire pour le J_1.

De plus, comme le jeu est initialisé en v_0 et qu'il s'agit de l'objectif du second joueur, celui-ci a un coût de 0 quelle que soit la stratégie qu'il adopte. J_2 n'a donc pas non plus de déviation profitable. Nous pouvons dès lors conclure que (σ_1, σ_2) est un équilibre Nash.

Dans [1], le théorème suivant est énoncé et prouvé :

Théorème 3.2. Soient $\mathcal{A} = (\Pi, V, (V_i)_{i\in\Pi}, E)$ et $\mathcal{G} = (\mathcal{A}, (\varphi_i)_{i\in\Pi}, (Goal_i)_{i\in\Pi})$ un jeu d'atteignabilité multijoueur à objectif quantitatif, si la fonction de poids associée au graphe du jeu est une fonction positive alors, il existe un équilibre de Nash dans tout jeu initialisé $(\mathcal{G}, v_0)$ avec $v_0 \in V$. De plus, cet équilibre possède une mémoire d'au plus $|V| + |\Pi|$.

Ce théorème signifie que dans de tels jeux, il existe un équilibre de Nash $(\sigma_i)_{i\in\Pi}$ tel que pour tout $i \in \Pi$ la taille de la mémoire pour représenter σ_i est d'au plus $|\Pi| + |V|$. L'idée de la preuve est de considérer pour chaque joueur $i \in \Pi$ le jeu Min-Max $\mathcal{G}^i$ associé tel que les autres joueurs jouent en coalition contre le joueur i. Du jeu $\mathcal{G}^i$, est extraite une stratégie optimale σ_i^* pour chaque joueur et une stratégie de punition pour chaque joueur $j \in \Pi\backslash\{i\}$. L'équilibre de Nash construit est le suivant : chaque joueur suit sa stratégie optimale σ_i^* et dès qu'un joueur j dévie de sa stratégie optimale, les autres joueurs jouent selon la stratégie de punition issue du jeu Min-Max $\mathcal{G}^j$. Dès lors, chaque joueur i a besoin de retenir sa stratégie optimale, ainsi que ses stratégies de punition issues des jeux $\mathcal{G}^j$ où $j \neq i$ et la position en laquelle un joueur a dévié pour la première fois. Ceci explique respectivement le terme $|\Pi|$ et $|V|$ de la taille de mémoire nécessaire.

4 Recherche d'équilibres de Nash pertinents

Les prérequis sur les jeux d'atteignabilité ayant été clairement expliqués, nous nous intéressons à la recherche - de manière algorithmique - d'équilibres de Nash « pertinents ». Dans un premier temps, nous définissons clairement l'objectif que nous désirons atteindre ainsi que ce signifie pour nous la notion d'équilibre de Nash pertinent. Ensuite, nous établissons quelques remarques et propriétés qui nous sont utiles afin de mener à bien notre raisonnement. Enfin, nous expliquons de quelle manière nous désirons mettre en oeuvre des procédés permettant de résoudre cette problématique.

4.1 Définition du problème et des équilibres pertinents

Tout d'abord, soit $\mathcal{G} = (\{1,2\}, V, (V_1, V_2), E, (Cost_1, Cost_2))$, considérons le jeu $(\mathcal{G}, v_1)$ où :

- Pour tout $\rho = \rho_0\rho_1\rho_2\ldots$ où $\rho \in Plays$:
  $Cost_i(\rho) = \begin{cases} \min\{i | \rho_i \in Goal_i\} & \text{si } \exists i \text{ tq } \rho_i \in Goal_i \\ +\infty & \text{sinon} \end{cases}$,
- $Goal_1 = \{v_3\}$ et $Goal_2 = \{v_0\}$,
- V_1 (resp. V_2) est représenté par les noeuds ronds (resp. carrés) du graphe de la figure 19.

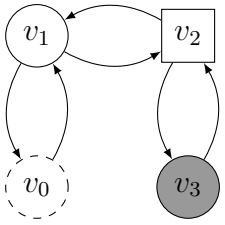


FIGURE 19 – Jeu d'atteignabilité avec coût

Soit $\sigma_1(v) = \begin{cases} v_2 & \text{si } v = v_1 \\ v_1 & \text{si } v = v_0 \\ v_2 & \text{si } v = v_3 \end{cases}$

et soit $\sigma_2(v) = v_1$ pour $v = v_2$ alors, (σ_1, σ_2) est un équilibre de Nash du jeu $(\mathcal{G}, v_1)$ dont l'outcome est $(v_1v_2)^\omega$. Nous remarquons qu'avec cet équilibre de Nash aucun des deux joueurs n'atteint son objectif. Nous pouvons également observer que si les deux joueurs coopéraient, ils pourraient tous deux minimiser leur coût. En effet, si les deux joueurs suivaient un profil de

stratégie ayant comme outcome $\rho = v_1v_0v_1(v_2v_3)^\omega$ nous aurions $Cost_1(\rho) = 4$ et $Cost_2(\rho) = 1$.

La question que nous nous posons alors est la suivante : « Etant donné un jeu d'atteignabilité à objectifs quantitatifs et multijoueur, nous souhaitons trouver de manière rapide un équilibre de Nash pertinent ». Nous avons donc dû nous interroger au sens à donner au concept d'équilibre de Nash pertinent.

Au vu de l'exemple ci-dessus, nous avons mis en lumière le fait que pour certains équilibres de Nash, aucun joueur ne voyait son objectif atteint. Dès lors, nous pouvons établir qu'un équilibre de Nash pertinent vérifie :

- Si on est dans un type de jeu tel qu'il existe (au moins) un équilibre de Nash tel que tous les joueurs voient leur objectif atteint alors, on souhaite que si ρ est l'outcome de l'équilibre de Nash trouvé on ait que $\sum_{i\in\Pi} \varphi_i(\rho)$ soit **minimale**.
- S'il n'est pas certain qu'il existe un équilibre de Nash du type de celui décrit ci-dessus alors on désire **maximiser** le nombre de joueurs qui atteignent leur objectif, notons cet ensemble de joueurs $Visit(\rho)$. De plus, on souhaite **minimiser** $\sum_{i\in\text{Visit}(\rho)} \varphi_i(\rho)$.

4.2 Caractérisation de l'outcome d'un équilibre de Nash

Une des premières questions que nous nous sommes alors posée est la suivante :

Question 1. Soit $G = (V, E)$ un graphe orienté fortement connexe[2] qui représente l'arène d'un jeu d'atteignabilité multijoueur avec coût : $(\mathcal{G}, v_0)$ (pour un certain $v_0 \in V$). Existe-t'il un équilibre de Nash tel que chaque joueur atteigne son objectif ?

Nous n'avons pas encore de réponse claire à cette question. Toutefois, pour un jeu à n joueurs, nous avons une bonne intuition quant à la manière de passer d'un équilibre de Nash où $n-1$ joueurs atteignent leur objectif à un équilibre de Nash où n joueurs atteignent leur objectif. En effet, à partir du dernier état objectif atteint, si j est le joueur qui n'a pas atteint son objectif, il suffit que tous les joueurs s'allient afin d'atteindre un objectif du joueur j. **Cette question est dès lors toujours une question ouverte.**

Nous remarquons que si nous répondons positivement à cette question alors, nous rentrons dans les conditions explicitées dans le premier point de la section précédente.

2. En théorie des graphes, un graphe $G = (V, E)$ est dit fortement connexe si pour tout u et v dans V, il existe un chemin de u à v

Maintenant, nous nous demandons s'il existe un processus algorithmique qui permet de déterminer si un outcome particulier correspond à l'outcome d'un équilibre de Nash. Dans le cas échéant, nous désirons expliciter ce procédé.

Dans la suite de cette section, nous répondons positivement à cette question. Nous énonçons d'abord une propriété (propriété 4.1) qui nous permet de déterminer une condition nécessaire et suffisante pour qu'un outcome soit l'outcome d'un certain équilibre de Nash. Toutefois, pour pouvoir effectuer une procédure algorithmique sur les outcomes, il faut que nous trouvions un moyen de représenter ceux-ci car nous travaillons avec des mots infinis. Nous nous convainquons donc par la suite que nous pouvons nous restreindre à l'étude d'équilibres de Nash dont l'outcome est de la forme $\alpha\beta^\omega$ où α et β sont des mots finis.

Avant de formuler notre propriété, nous devons aborder quelques notions qui nous sont nécessaires.

Définition 4.1. Soient $\mathcal{A} = (\Pi, V, (V_i)_{i\in\Pi}, E)$ une arène,
et $\mathcal{G} = (\mathcal{A}, (\varphi_i)_{i\in\Pi}, (Goal_i)_{i\in\Pi})$ un jeu d'atteignabilité où $|\Pi| \geqslant 2$ à objectif quantitatif. Pour tout joueur $i \in \Pi$, nous pouvons y associer un jeu à somme nulle de type « reachability-price game » noté $\mathcal{G}_i$. On définit ce jeu de la manière suivante : $\mathcal{G}_i = (\mathcal{A}_i, g, Goal)$ où :

- $\mathcal{A}_i = (\{i, -i\}, V, (V_i, V\backslash V_i), E)$
- $g = \varphi_i$
- $Goal = Goal_i$

De plus, pour tout $v \in V$, $Val_i(v)$ est la valeur du jeu $\mathcal{G}_i$ pour tout noeud $v \in V$.

En d'autres mots, $\mathcal{G}_i$ correspond au jeu où le joueur i (joueur Min) joue contre la coalition $\Pi\backslash\{i\}$ (joueur Max). Cela signifie que le joueur i tente d'atteindre son objectif le plus rapidement possible tandis que tous les autres joueurs veulent l'en empêcher (ou tout du moins maximiser son gain). Nous avons vu précédemment qu'un tel jeu est déterminé et que les deux joueurs possèdent une stratégie optimale (σ_i^* et σ_{-i}^*) telles que :

$$\inf_{\sigma_{i\in\Sigma_{Min}}} \varphi_i(\langle\sigma_i, \sigma_{-i}^*\rangle_v) = Val_i(v) = \sup_{\sigma_{-i}\in\Sigma_{Max}} \varphi_i(\langle\sigma_i^*, \sigma_{-i}\rangle_v).$$

De plus, de la stratégie optimale σ_{-i}^* nous pouvons dériver une stratégie pour tout joueur $j \neq i$ que nous notons $\sigma_{j,i}$.

Nous sommes maintenant aptes à énoncer notre propriété. La preuve de celle-ci a été effectuée par nos soins, mais nous faisons remarquer qu'une

preuve similaire dans le cas des jeux concurrents à informations parfaites a déjà été effectuée par Haddad [4].

Propriété 4.1. Soient $|\Pi| = n \geqslant 2$, $\mathcal{A} = (\Pi, V, (V_i)_{i\in\Pi}, E)$ une arène et $\mathcal{G} = (\mathcal{A}, (\varphi_i)_{i\in\Pi}, (Goal_i)_{i\in\Pi})$ un jeu d'atteignabilité à n joueurs à objectif quantitatif, on considère $(\mathcal{G}, v_0)$ le jeu initialisé pour un certain $v_0 \in V$.
Soit $\rho = v_0 v_1 ... \in Plays$, posons $(x_i)_{i\in\Pi} = (\varphi_i(\rho))_{i\in\Pi}$ le profil de paiements associé à la partie ρ. Nous définissons pour $v_k \in \rho : \varepsilon_k := \sum_{n=0}^{k-1} w(v_n, v_{n+1})$ où w est la fonction de poids associée à $G = (V, E)$.

Il existe un profil de stratégies $(\sigma_i)_{i\in\Pi} \in \prod_{i\in\Pi} \Sigma_i$ qui est un équilibre de Nash dans $(\mathcal{G}, v_0)$ et tel que $\langle (\sigma_i)_{i\in\Pi} \rangle_{v_0} = \rho$

si et seulement si

$$\forall k \in \mathbb{N}, \forall j \in \Pi,\ Val_j(v_k) + \varepsilon_k \geqslant x_j \text{ si } v_k \in V_j.$$

Preuve. Nous allons montrer les deux implications :

($\Downarrow$) Supposons au contraire qu'il existe $k \in \mathbb{N}$ et $j \in \Pi$ tels que $Val_j(v_k) + \varepsilon_k < x_j$,

$$i.e., Val_j(v_k) < x_j - \varepsilon_k = \varphi_j(\langle (\sigma_i)_{i\in\Pi} \rangle_{v_k}) \tag{1}$$

où $\varphi_j(\langle (\sigma_i)_{i\in\Pi} \rangle_{v_k})$ est le coût de la partie pour le joueur j si elle avait commencé en v_k. De plus, on a :

$$\begin{aligned} Val_j(v_k) &= \sup_{\tau_{-j}\in\Sigma_{Max}} g(\langle \sigma_j^*, \tau_{-j} \rangle_{v_k}) \\ &\geqslant g(\langle \sigma_j^*, \sigma_{-j} \rangle_{v_k}) = \varphi_j(\langle \sigma_j^*, \sigma_{-j} \rangle_{v_k}) \end{aligned} \tag{2}$$

où σ_j^* est la stratégie optimale du joueur j associée à $\mathcal{G}_j$ et σ_{-j} dans l'expression $g(\langle \sigma_j^*, \sigma_{-j} \rangle_{v_k})$ est un abus de notation désignant la stratégie où la coalition $\Pi \backslash \{j\}$ suit chacune des stratégies σ_i pour tout $i \neq j$.

Dès lors, (1) et (2) nous donnent :

$$\varphi_j(\langle \sigma_j^*, \sigma_{-j} \rangle_{v_k}) < \varphi_j(\langle (\sigma_i)_{i\in\Pi} \rangle_{v_k}) \tag{3}$$

La relation (3) signifie qu'à partir du noeud v_k, le joueur j ferait mieux de suivre la stratégie σ_j^*. Il s'agit donc d'une déviation profitable pour le joueur j par rapport au profil de stratégies $(\sigma_i)_{i\in\Pi}$. Cela implique que $(\sigma_i)_{i\in\Pi}$ n'est pas un équilibre de Nash. Nous avons donc la contradiction attendue.

(⇑) Soit $(\tau_i)_{i\in\Pi}$ un profil de stratégies qui permet d'obtenir l'outcome ρ de paiement $(x_i)_{i\in\Pi}$. A partir de $(\tau_i)_{i\in\Pi}$ nous désirons construire un équilibre de Nash ayant le même outcome (et donc le même profil de coût). L'idée est la suivante : dans un premier temps tous les joueurs suivent leur stratégie conformément au profil $(\tau_i)_{i\in\Pi}$. Si un des joueurs, notons le i, dévie de sa stratégie alors les autres joueurs se réunissent en une coalition $\Pi\backslash\{i\}$ et jouent en suivant leur stratégie de punition dans $\mathcal{G}_i$ (*i.e.,* pour tout j $\neq$ i, le joueur j suit la stratégie $\sigma^*_{j,i}$).

Comme dans le papier « Multiplayer Cost Games With Simple Nash Equilibria » [1], nous définissons une fonction de punition $P : Hist \to \Pi \cup \{\perp\}$ qui permet de définir quel est le premier joueur à avoir dévié du profil de stratégies initial $(\tau_i)_{i\in\Pi}$. Cette fonction est telle que $P(h) = \perp$ si aucun joueur n'a dévié le long de l'histoire h et $P(h) = i$ pour un certain $i \in \Pi$ si le joueur i a dévié le long de l'histoire h. Nous pouvons donc définir la fonction P par récurrence sur la longueur des histoires : pour v_0, le noeud initial, $P(v_0) = \perp$ et pour $h \in Hist$ et $v \in V$ on a :

$$P(hv) = \begin{cases} \perp & \text{si } P(h) = \perp \text{ et } hv \text{ est un préfixe de } \rho \\ i & \text{si } P(h) = \perp,\ hv \text{ n'est pas un préfixe de } \rho \\ & \text{et } Last(h) \in V_i \\ P(h) & \text{sinon } (i.e., P(h) \neq \perp) \end{cases}$$

Nous pouvons maintenant définir notre équilibre de Nash potentiel dans $\mathcal{G}$. Pour tout $i \in \Pi$ et tout $h \in Hist$ tels que $Last(h) \in V_i$:

$$\sigma_i(h) = \begin{cases} \tau_i(h) & \text{si } P(h) = \perp \text{ ou } i \\ \sigma^*_{i,P(h)}(h) & \text{sinon} \end{cases}$$

Nous devons désormais montrer que le profil de stratégies $(\sigma_i)_{i\in\Pi}$ ainsi défini est un équilibre de Nash d'outcome ρ.
Il est clair que $\langle(\sigma_i)_{i\in\Pi}\rangle_{v_0} = \rho$.
Montrons maintenant qu'il s'agit bien d'un équilibre de Nash.
Supposons au contraire que ce ne soit pas le cas. Cela signifie qu'il existe une déviation profitable (notons-la $\tilde{\sigma}_j$) pour un certain joueur

$j \in \Pi$.
Soit $\tilde{\rho} = \langle \tilde{\sigma}_j, (\sigma_i)_{i \in \Pi \setminus \{j\}} \rangle_{v_0}$ l'outcome tel que le joueur j joue sa déviation profitable et où les autres joueurs jouent conformément à leur ancienne stratégie. Puisque $\tilde{\sigma}_j$ est une déviation profitable nous avons :

$$\varphi_j(\tilde{\rho}) < \varphi_j(\rho) \tag{4}$$

De plus, comme ρ et $\tilde{\rho}$ commencent tous les deux à partir du noeud v_0, ils possèdent un préfixe commun. En d'autres termes, il existe une histoire $hv \in Hist$ telle que :

$$\rho = h.\langle (\sigma_i)_{i \in \Pi} \rangle_v \text{ et } \tilde{\rho} = h.\langle \tilde{\sigma_j}, (\sigma)_{i \in \Pi \setminus \{j\}} \rangle_v$$

S'il en existe plusieurs, nous en choisissons une de longueur maximale. Au vu de la définition de σ_i, nous pouvons réécrire :

$$\rho = h.\langle (\tau_i)_{i \in \Pi} \rangle_v \text{ et } \tilde{\rho} = h.\langle \tilde{\sigma_j}, (\sigma^*_{i,j})_{i \in \Pi \setminus \{j\}} \rangle_v$$

En effet, le joueur j dévie en v, donc à partir de v tout joueur $i \neq j$ joue sa stratégie de punition. De plus, nous avons les relations suivantes :

$$\begin{aligned} Val_j(v) &= \inf_{\mu_j \in \Sigma_{Min}} \varphi_j(\langle \mu_j, \sigma^*_{-j} \rangle_v) \\ &\leqslant \varphi_j(\langle \tilde{\sigma}_j, \sigma^*_{-j} \rangle_v) \\ &= \varphi_j(\langle \tilde{\sigma}_j, (\sigma^*_{i,j})_{i \in \Pi \setminus \{j\}} \rangle_v). \end{aligned} \tag{5}$$

Supposons $h = v_0 \dots v_k$ pour un certain $k \in \mathbb{N}$. Alors,

$$\varphi_j(\tilde{\rho}) = \varepsilon_k + \varphi(\langle \tilde{\sigma}_j, (\sigma^*_{i,j})_{i \in \Pi \setminus \{j\}} \rangle_v) \tag{6}$$

Dès lors, (5) et (6) nous donnent :

$$Val_j(v) \leqslant \varphi_j(\tilde{\rho}) - \varepsilon_k \tag{7}$$

Donc par (4) et (7) nous avons :

$$Val_j(v) \leqslant \varphi_j(\tilde{\rho}) - \varepsilon_k < \varphi_j(\rho) - \varepsilon_k = x_j - \varepsilon_k$$

Ce qui contredit l'hypothèse et conclut notre preuve.

□

Dans sa thèse [3], Julie De Pril explicite une procédure afin de construire à partir d'un équilibre de Nash un équilibre de Nash du même *type* tel que toutes les stratégies sont à mémoire finie. Le type d'un profil de stratégies est l'ensemble des joueurs qui ont visité leur objectif en suivant cet équilibre. Si le profil de stratégies est $(\sigma_i)_{i \in \Pi}$ alors on note le type de ce profil $\text{Type}((\sigma_i)_{i \in \Pi})$. Le théorème suivant est donc énoncé :

Théorème 4.1. Etant donné un équilibre de Nash dans un jeu multijoueur initialisé à objectif quantitatif, il existe un équilibre de Nash du même type.

Ce procédé consiste en deux étapes. Etant donné un équilibre de Nash, on commence par construire un second équilibre de Nash du même type duquel on a supprimé les cycles inutiles. Un cycle inutile est un cycle tel que lors de son parcourt aucun joueur qui n'avait pas encore visité son objectif n'atteint pas celui-ci mais qu'ensuite un nouvel objectif est atteint. Ensuite, on construit un équilibre de Nash $(\sigma_i)_{i\in\Pi}$, toujours avec le même type, tel qu'à partir de l'outcome $\langle(\sigma_i)_{i\in\Pi}\rangle_{v_0}$ on puisse identifier un préfixe $\alpha\beta$ pour lequel on peut répéter infiniment β. De plus, si nous définissons la notation $\text{Visit}(\alpha)$ comme étant l'ensemble des joueurs qui ont vu leur objectif atteint lors du parcourt de α, nous retrouvons dès lors le résultat suivant :

Propriété 4.2. Soit $(\sigma_i)_{i\in\Pi}$ un équilibre de Nash dans le jeu d'atteignabilité multijoueur à objectifs quantitatifs et initialisé $(\mathcal{G}, v_0)$, il existe un équilibre de Nash $(\tau_i)_{i\in\Pi}$ avec le même type et tel que $\langle(\tau_i)_{i\in\Pi}\rangle_{v_0} = \alpha\beta^\omega$, où $\text{Visit}(\alpha) = \text{Type}((\sigma_i)_{i\in\Pi})$ et $|\alpha\beta| < (\Pi + 1) \cdot |V|$.

Remarquons toutefois que dans le cadre de ces preuves, la fonction de poids w est la fonction telle que pour tout $e \in E$, $w(e) = 1$. Néanmoins, les preuves s'adaptent si la fonction de poids est de la forme $w : E \to \mathbb{N}_0$. En effet, s'il existe un arc (v, v') tel que $w(v, v') = c$ pour un certain $c \in \mathbb{N}_0$, il suffit de rajouter autant d'arc de poids 1 qu'il est nécessaire pour que la somme de ces poids vaille c (cf. figure 20 avec $c = 4$). Remarquons que si le graphe d'origine présente $|V|$ noeud, le graphe transformé, dont l'ensemble des noeuds est V', possède $|V'| = |V| + \sum_{e\in E}(w(e)-1)$ noeuds. Il semble donc raisonnable que nous nous contentions de considérer des outcomes de longueur $(\Pi + 1) \cdot |V'|$.

Une étude attentive de la preuve effectuée pour la propriété 4.2 permettrait peut-être d'exhiber une meilleure borne sur la longueur maximale des outcomes testés.

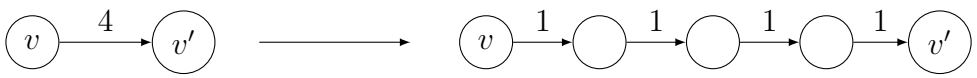


FIGURE 20 – Transformation d'une arête de poids différent de 1.

Nous pouvons donc conclure qu'il est correct de se restreindre à le recherche d'équilibres dont l'outcome est de cette forme. En effet, vu qu'ils possèdent le même type, cela n'influence pas notre intention de maximiser

le nombre de joueurs qui atteignent leur objectif. De plus, puisque les poids sur les arcs sont tous des poids positifs, la suppression des cycles lors de la première étape de la procédure ne fait que diminuer le coût des joueurs pour cet équilibre si un cycle est présent. Cette modification est à notre avantage. En effet, cela diminue la somme des coûts des joueurs qui est la valeur que nous désirons minimiser pour trouver un équilibre pertinent.

Nous avons désormais à notre disposition :

- Une manière de tester si un outcome correspond à l'outcome d'un équilibre de Nash.
- Un résultat permettant d'affirmer que nous pouvons nous contenter d'examiner les équilibres de Nash dont l'outcome est de la forme $\alpha\beta^\omega$ où α et β sont des éléments de V^+. Nous pouvons donc travailler uniquement à partir de chemins de longueur au plus $(\Pi + 1) \cdot |V'|$. De là, nous pouvons appliquer le point précédent sur $\alpha\beta$ et comme ce mot est un mot fini, un algorithme peut effectuer cette tâche.
- Un algorithme (`DijkstraMinMax`) qui permet des récupérer les valeurs de chaque noeud pour tous les jeux où un joueur joue contre la coalition des autres joueurs. Ces valeurs permettent de vérifier si la propriété du point précédent est respectée.

Il nous reste donc à déterminer un algorithme qui nous permet de trouver rapidement un équilibre de Nash pertinent. La phase d'implémentation de ce projet ayant commencé relativement tard les méthodes testées restent sans doute un peu naïves. Avant d'aborder ces dernières avec plus de précisions, nous expliquons certains types d'algorithmes d'exploration desquels nous nous sommes inspirés.

4.3 Algorithmes d'exploration

Cette section se base sur le livre de Russel et Norvig [6] ainsi que sur les notes du cours d'Intelligence Artificielle enseigné par Hadrien Mélot à l'université de Mons (notes de l'année 2014–2015).

Lorsque l'on recherche une solution à un problème plusieurs méthodes sont applicables en fonction de la nature du problème. Nous nous sommes penchés sur une résolution utilisant des méthodes d'exploration. C'est-à-dire qu'à un problème donné est associé un *espace d'états*. Ces états sont parcourus jusqu'à atteindre l'objectif attendu, celui-ci correspond à la solution du problème.

Dans notre cas, l'objectif que nous désirons atteindre est de trouver un

chemin dans le graphe du jeu qui corresponde à l'outcome d'un équilibre de Nash. La pertinence de cet équilibre est jaugée via les critères évoqués dans la section 4.1. Le but ultime est donc d'atteindre l'équilibre le plus pertinent.

Différents types d'exploration sont possibles. Deux grandes familles sont à évoquer :

- Algorithme d'exploration non informée : seule la définition du problème est connue (*blind search*).
- Algorithme d'exploration informée : la recherche est guidée grâce à des données supplémentaires (*heuristic search*).

Exemple 4.1. Un voyageur qui se situe dans une ville A souhaite rejoindre une ville B. Pour ce faire, il aimerait emprunter le chemin le plus court pour relier A et B. Une exploration non informée testerait les différentes villes voisines de la ville A et ainsi de suite jusqu'à atteindre la ville B. Une approche plus éclairée serait donc d'utiliser la distance à vol d'oiseau séparant une ville C de la ville B afin de pouvoir choisir à chaque fois quelle ville voisine C visiter en priorité. Il s'agit alors d'une exploration informée.

L'objectif à atteindre étant établi, le problème doit maintenant être défini. Les cinq composants suivants sont essentiels :

L'état initial est l'état à partir duquel l'exploration commence. Dans notre cas, puis que nous travaillons avec des jeux initialisés par un certain sommet v_0, l'état initial est le chemin v_0.

Les actions possibles à partir d'un état donné afin de pouvoir continuer l'exploration. Ici, si le dernier noeud du chemin courant est le noeud v_i, les actions possibles sont de choisir un noeud parmi ceux de $Succ(v_i) = \{v \mid (v_i, v) \in E\}$.

Le modèle de transition associe, à partir d'un état courant et d'une action, l'état successeur. Si on est à l'état $h = h_1...h_k$ et que l'action choisie est de sélectionner le noeud v, alors l'état successeur est l'état hv.

L'état initial, les actions possibles ainsi que le modèle de transition définissent *l'espace d'états* qui est l'ensemble de tous les états possibles associés au problème.

Un test détermine si un état correspond à un état objectif. Dans le cas présent, ce test consiste soit à vérifier si la longueur du chemin correspond à la longueur maximale de l'outcome d'un équilibre de Nash et dans le cas échéant de tester si ce chemin en est bien un équilibre de Nash via le critère de la propriété 4.1, soit tous les joueurs ont atteint leur objectif et nous vérifions via ce même critère si ce chemin est un équilibre de Nash.

Une fonction de coût modélise la qualité du chemin courant. Une solution optimale est telle que le coût associé à l'état objectif solution est minimal.

Maintenant que le modèle est explicité, nous expliquons de quelle manière la recherche est effectuée. Tous les algorithmes d'exploration utilisent cette structure. La différence entre ces différents algorithmes est la manière dont les états successifs sont sélectionnés, cette notion est nommée *stratégie d'exploration*.

La recherche se fait en parcourant un arbre dont la racine est l'état initial, les branches les actions possibles et les noeuds les états. A chaque fois qu'un état est sélectionné, tous ses successeurs sont générés et sont ajoutés à l'ensemble des états qui attendent d'être visités. Cet ensemble est appelé *frontière*, tandis que ce processus de génération des successeurs d'un état est appelé *expansion*. Tant qu'un état objectif n'est pas atteint et que la frontière contient des éléments, un nouvel état est sélectionné dans celle-ci et ses successeurs y sont ajoutés. Cette procédure est appelée *tree search*.

L'inconvénient de cette approche est que l'arbre généré peut-être infini. En effet, supposons qu'il existe un arc (v_0, v_1) et un arc (v_1, v_0) dans le graphe du jeu, alors le chemin $(v_0v_1)^\omega$ peut être généré et le processus pourrait ne jamais s'arrêter. Dans notre cas, puisque nous nous restreignons à la recherche de chemin d'une certaine longueur maximale, ce comportement n'est pas dérangeant. En effet, une fois la longueur maximale du chemin atteinte, nous pouvons couper la recherche à partir de ce noeud. De plus, empêcher l'apparition de cycle n'est pas envisageable car même pour la recherche d'équilibre de Nash pertinent, des cycles peuvent être nécessaires (cf. exemple de la section 4.1 pour lequel un outcome d'équilibre de Nash pertinent est $\rho = v_1v_0v_1(v_2v_3)^\omega$).

Comme nous l'avons déjà précisé, les différents algorithmes d'exploration différent entre eux par leur stratégie d'exploration. La structure de donnée utilisée pour représenter la frontière influence donc l'ordre de traitement des noeuds de l'arbre d'exploration. Les trois structures suivantes sont utilisées : les piles (le dernier élément ajouté est le premier retiré), les files (la premier élément ajouté est le premier élément retiré) et les files de priorités (les éléments sont triés selon une certaine préférence).

L'utilisation d'une file permet, par exemple, une exploration en largeur de l'arbre - appelée *breadth-first search*. Cette exploration visite en premier les noeuds de l'arbre qui sont les moins profonds. Tandis qu'une file assure prioritairement l'expansion des noeuds les plus profonds. Cette exploration est appelée *depth-first search*. Ces deux approches sont des exemples d'ex-

plorations non informées. Elles ne sont, en général, pas optimales car elles retournent le premier état objectif trouvé. De plus, leur complexité en temps est exponentielle. Si d est distance minimale entre l'état initial et l'état objectif, b est le nombre maximum d'enfants d'un noeud dans l'arbre d'exploration et m est la profondeur maximale de l'espace d'état, alors breadth-first search a une complexité en temps en $\mathcal{O}(b^d)$ tandis que le le depth-first search est en $\mathcal{O}(b^m)$.

Ces deux approches étant assez naïves et non exploitables en pratique, nous aimerions guider notre recherche afin que celle-ci trouve le plus rapidement possible une solution de bonne qualité. Les stratégies d'exploration informées sont utilisées dans ce but et ce type d'exploration est appélé *best-first search*.

Dans ce genre de stratégie, les noeuds de la frontière sont ordonnés grâce à une *fonction d'évaluation* f qui permet d'estimer à quel point ce noeud est souhaitable. De plus, pour la plupart des algorithmes de type best-first search, une *fonction heuristique* h est utilisée dans l'expression de la fonction f. En fait, $h(n)$ estime le plus petit coût nécessaire pour rejoindre l'état objectif le plus proche à partir du noeud n.

Terminons en expliquant le cas particulier de l'algorithme A^* ainsi que la manière dont nous l'avons utilisé afin de trouver une solution à notre problème.

4.3.1 Algorithme A*

L'algorithme A^* est un exemple d'algorithme best-first search qui est caractérisé par sa fonction d'évaluation. Dans ce cas, f est défini de la manière suivante :

$$f(n) = g(n) + h(n)$$

où

- $g(n)$ est le coût nécessaire pour aller du noeud initial au noeud n.
- $h(n)$ est l'estimation du coût minimum nécessaire pour aller du noeud n à un état objectif.
- $f(n)$ est donc l'estimation du coût minimum pour pour aller du noeud initial à un noeud objectif et ce en passant par le noeud n.

Pour notre part, la fonction que nous désirons minimiser est la suivante :

$$F(\rho) = \sum_{i \in \text{Visit}(\rho)} \varphi_i(\rho) + |\Pi \setminus \text{Visit}(\rho)| \cdot p$$

où nous avons :

- $\rho = v_0\rho_1...\rho_l$ où v_0 est le noeud initial du jeu et l est la longueur maximale des chemins à tester.
- p est le poids maximal que peut atteindre un chemin tel que sa longueur est la longueur maximale à tester.
  Dans le cadre de nos tests, nous fixons cette valeur à : $((\Pi + 1) \cdot (|V| + \sum_{e \in E}(w(e) - 1))) \cdot \max_{e \in E} w(e)$.
  Une étude plus approfondie du problème permettrait-elle également de trouver une borne plus fine ? Améliorerait-elle vraiment les résultats obtenus ?

Nous utilisons donc pour g et h les fonctions suivantes :

$$g(n) = \sum_{i \in \text{Visit}(\rho)} \varphi_i(\rho) + |\Pi \setminus \text{Visit}(\rho)| \cdot \varepsilon_k$$

$$h(n) = \sum_{i \notin \text{Visit}(\rho)} \min\{p - \varepsilon_k,\, c_i\}$$

où nous avons :

- ρ est le chemin $\rho = h_0h_1...h_k$ (où $h_0 = v_0$ le noeud initial du jeu) stocké dans le noeud n.
- $\varepsilon_k = \sum_{j=0}^{k-1} w(h_j, h_{j+1})$.
- c_i est le poids du plus court chemin pour rejoindre un objectif du joueur i à partir du sommet h_k.

Nous savons que sous certaines conditions l'algorithme A^* est optimal. Est-il possible dans ce cas d'arriver à une telle conclusion ? L'est-il sous certaines conditions ? Quelles sont les performances réelles de cette approche ? Quelles en sont les limites, les types de graphes de jeu sur lesquels elle est applicable ?

Toutes les notions théoriques qui nous sont nécessaires pour l'implémentation d'un procédé visant à rechercher un équilibre de Nash pertinent sont maintenant expliquées. Nous pouvons maintenant expliquer la manière dont nous les avons mises en œuvre.

5 Implémentation et résultats

La phase d'implémentation de notre projet ayant été entreprise tardivement, il ne s'agit pas de la plus aboutie et de nombreuses autres pistes de recherche sont encore à explorer. Nous exposons toutefois dans cette section celles que nous avons empruntées ainsi que les résultats obtenus.

5.1 Implémentation

Notre implémentation a été réalisée en Python et se découpe en différents modules et classes.

- **Module MinHeap** permet de représenter la structure de tas nécessaire à l'exécution de l'algorithme DijkstraMinMax.
- **Module DijkstraMinMax** regroupe toutes les méthodes nécessaires à l'algorithme DijkstraMinMax ainsi que des méthodes pour récupérer les résultats de ce dernier.
- **Module GraphGame** comprend :
  - ⋆ **Classe Graph** pour représenter un graphe ainsi que différentes fonctions pour effectuer des manipulations sur ce dernier.
  - ⋆ **Classe Vertex** pour représenter un sommet du graphe.
  - ⋆ **Classe ReachabilityGame** qui modélise la structure de jeu d'atteignabilité. Elle regroupe, entre autre, les méthodes nécessaires pour calculer le coût d'un chemin, déterminer si un chemin correspond à l'outcome d'un équilibre de Nash ainsi que les différents algorithmes d'exploration testés.
- **Les modules DijkstraMinMaxTest, ReachabilityGameTest et TestMinHeap** qui nous ont permis de tester nos algorithmes au fur et à mesure de leur implémentation.
- **Main** qui permet de tester nos algorithmes sur des graphes générés aléatoirement.

Bien que la plupart de nos algorithmes sont utilisables dans des cas plus généraux, nous avons fixé certaines conditions sur les jeux considérés.

- Jeux à deux joueurs,
- Un seul objectif par joueur (ces deux objectifs étant différents),
- Le graphe du jeu est un graphe complet (excepté l'arc d'un noeud vers lui-même),
- La fonction de poids considérées est de la forme $w : E \to [1, 100] \cap \mathbb{N}$.

Comme nous l'avons déjà évoqué, plusieurs approches ont été testées afin de trouver un équilibre de Nash pertinent. Nous les expliquons brièvement ci-après.

Méthode aléatoire

De loin la plus naïve, la méthode aléatoire génère de manière aléatoire à partir du sommet initial un certain nombre de chemins (ce nombre est paramétrable) dans le graphe du jeu et ne retient que ceux qui correspondent à l'outcome d'un équilibre de Nash. La longueur de ce chemin est la longueur maximale des chemins à tester. A partir de ce résultat, une méthode a été implémentée afin de ne retenir que l'équilibre le plus pertinent.

Breadth-first search

Nous avons également à notre disposition un algorithme breadth-first search qui a été un peu modifié. En effet, au lieu de retourner directement le premier objectif rencontré lors du parcours de l'arbre, il est possible de les stocker et de tous les retourner. Nous pouvons ensuite, comme pour la méthode précédente retenir uniquement l'équilibre le plus pertinent. Comme le parcours de l'entièreté de l'arbre d'exploration peut prendre beaucoup de temps (complexité exponentielle), il est possible de préciser une limite de temps pour l'exécution de l'algorithme. Une fois cette limite dépassée, ce dernier renvoie tous les équilibres trouvés durant ce laps de temps.

Best-first search

Une implémentation de la méthode d'exploration best-first search a également été mise en œuvre. Nous l'avons toutefois améliorée au vu de l'utilisation que nous en faisons. En effet, dans certains cas, il est possible de couper des branches de l'arbre d'exploration :

- Si le chemin dans le noeud en cours de traitement a atteint la longueur maximale des chemins à tester, il est inutile de pratiquer l'expansion à partir de ce noeud.
- Si lors de l'expansion d'un noeud, et donc de l'ajout d'un sommet v_i du graphe du jeu au chemin en cours de construction, le noeud v_i ajouté correspond à un sommet objectif d'un joueur j n'ayant pas encore atteint son objectif, alors on peut appliquer le critère de la propriété 4.1. En effet, si cette propriété n'est pas respectée pour le joueur j alors il ne sert à rien de continuer l'exploration à partir de ce chemin.

Comme pour l'approche précédente, une limite sur le temps d'exécution de l'algorithme peut être paramétrée. De plus, n'importe quelle fonction d'évaluation peut être passée en paramètre afin d'ordonner les noeuds dans la frontière. Dans notre cas, après plusieurs tentatives de fonction d'évalua-

tion peu convaincantes, nous avons effectué nos tests à partir de la fonction d'évaluation correspondant à l'algorithme A^* (cf. section 4.3.1).

Best-first search initialisé

Au lieu de commencer l'exploration à partir du chemin composé uniquement du noeud initial v_0, nous avons tenté de guider au mieux le début de la recherche. Pour ce faire, nous procédons de la manière suivante : si o_1 (resp. o_2) est le noeud objectif de J_1 (resp. J_2), on construit dans un premier temps le chemin le plus court de v_0 à o_1, si ce chemin n'enfreint pas le critère pour être un équilibre de Nash, alors on lance le best-first search à partir du chemin $v_0...o_1$. Si par contre le chemin ainsi constitué enfreint le critère ou si la recherche prend trop temps, on recommence à partir du plus court chemin entre v_0 et o_2.

5.2 Résultats

Nous exposons dans cette section les résultats obtenus sur les quelques tests que nous avons effectués.

Tests sur des graphes complets générés aléatoirement

Afin de tester l'efficacité de l'algorithme A^* que nous avons implémenté, nous avons généré des graphes de manière aléatoire. Pour tout $n \in \mathbb{N}$ tel que $5 \leqslant n \leqslant 20$, 100 graphes dont les poids sur les arcs sont des entiers entre 1 et 100 ont été créés. L'algorithme a alors été exécuté sur ces instances et nous avons comptabilisé le nombre d'équilibres de Nash qui sont trouvés en permettant un temps de 30 secondes pour chaque exécution de l'algorithme. Les résultats de ces tests [3] sont illustrés par le tableau 2 et par la figure 21.

3. Tous les tests repris dans ce rapport ont été effectués sur un MacBook Pro, 2,8 GHz Intel Core i5

TABLE 2 – Données résultant des tests sur A^*

Nombre de sommets	EN trouvés	EN manqués	Temps total d'exécution (s)	Temps moyen pour trouver un EN(s)
5	100	0	0,4	0,004
6	100	0	2,37	0,0237
7	99	1	32,45	0,0247
8	98	2	63,08	0,0314
9	100	0	19,76	0,1976
10	99	1	46,22	0,1638
11	95	5	158,19	0,0862
12	99	1	33,86	0,039
13	98	2	135,71	0,7726
14	93	7	243,86	0,3641
15	98	2	80,08	0,2049
16	96	4	144,58	0,256
17	94	6	248,38	0,7274
18	93	7	258,74	0,5241
19	95	5	298,07	1,5586
20	92	8	331,21	0,9914

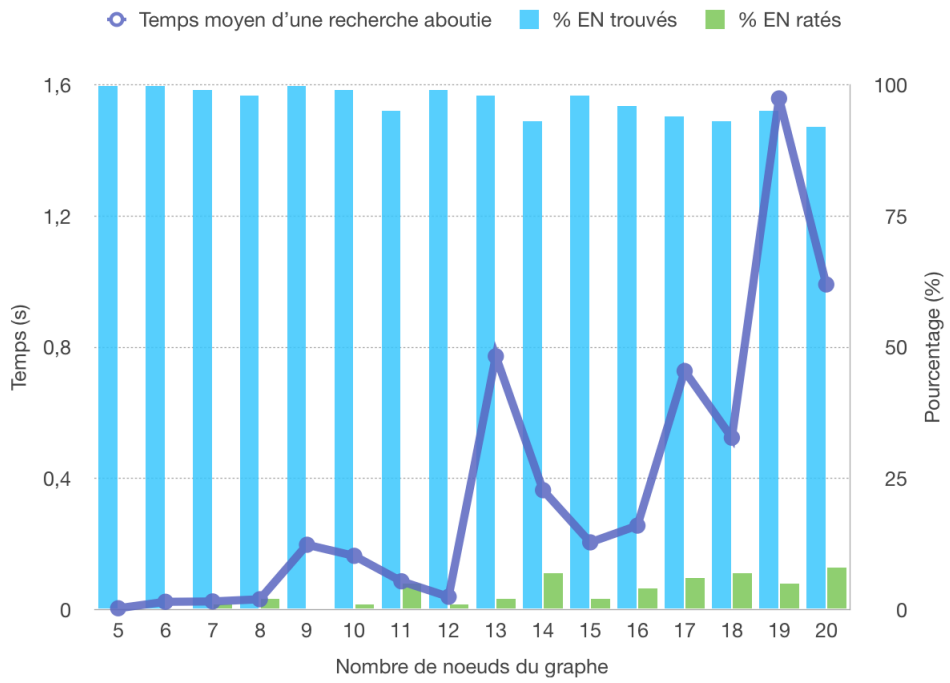


FIGURE 21 – Graphique illustrant les résultats des tests sur A^*

Tests sur des graphes non complets

Bien que notre objectif était de tester nos algorithmes sur des graphes complets, nous les avons également essayé sur les quelques exemples qui jalonnent notre travail.

Premier exemple

Commençons par nous intéresser par l'exemple 3.6 de la page 45. Les résultats retournés en appliquant la méthode aléatoire et l'algorithme A^* sont les suivants :

```
A_star : [v1, v2, v3, v0]
Est un EN?  True
Information sur l'outcome :
Cout pour chaque joueur:  {1: 2, 2: 3}
Joueurs ayant atteint leur objectif  [1, 2]

Random :  [v1, v0, v1, v2, v3, v0, v1, v2,
v4, v3, v4, v2, v3, v4, v3, v0, v1, v0, v1,
v0, v1, v2, v3, v0, v1, v2, v3]
Est un EN?  True
Information sur l'outcome :
Cout pour chaque joueur:  {1: 4, 2: 1}
Joueurs ayant atteint leur objectif  [1, 2]
```

Nous remarquons que pour les deux outcomes trouvés les deux joueurs atteignent leur objectif et que la somme de leur coût est de 5. Ils sont donc équivalents du point de vue de notre test de pertinence. De plus, si la méthode random est lancée une nouvelle fois, le même outcome que pour l'algorithme A^* est retourné.

Deuxième exemple

Pour l'exemple illustré dans la section 4.1 les résultats sont ceux repris ci-dessous.

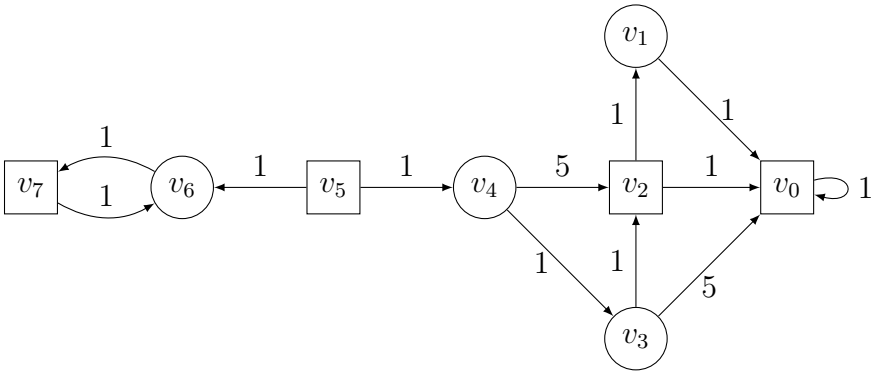
```
A_star : [v0, v2, v0, v1, v3]
Est un EN?  True
Information sur l'outcome :
Cout pour chaque joueur:  {1: 4, 2: 1}
Joueurs ayant atteint leur objectif  [1, 2]

Random :[v0, v2, v0, v1, v3, v1, v3,
```

```
v1, v3, v1, v0, v1]
Information sur l'outcome :
Cout pour chaque joueur:  {1: 4, 2: 1}
Joueurs ayant atteint leur objectif  [1, 2]
```

Les deux méthodes mènent au même résultat qui est de surcroît le résultat auquel nous nous attendions.

Troisième exemple

Enfin, à partir de l'arène définie dans l'exemple 3.4 de la page 32, on définit $Goal_1 = \{v_6\}$ et $Goal_2 = \{v_0\}$.



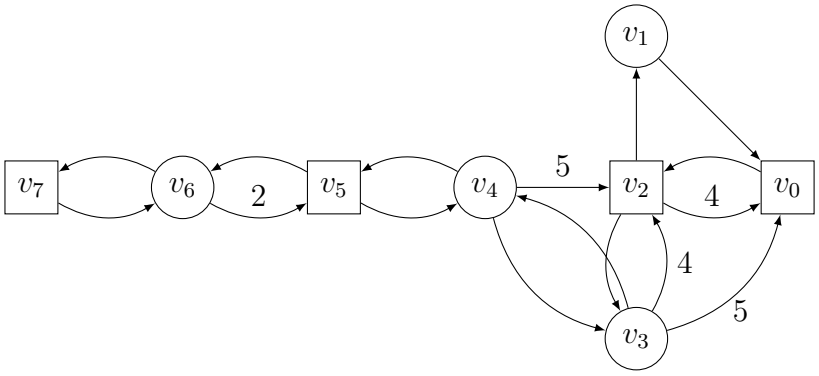
Voici les résultats obtenus :

```
A_star :
[v3, v2, v0, v0, v0, v0, v0, v0, v0, v0, v0,
v0, v0, v0, v0, v0, v0, v0, v0, v0, v0, v0 ,v0, v0,
v0, v0, v0 , v0, v0, v0, v0, v0 ,v0, v0, v0, v0, v0,
v0, v0, v0, v0, v0 , v0, v0, v0, v0, v0, v0, v0, v0]
Est un EN?  True
Information sur l'outcome :
Cout pour chaque joueur:  {1: inf, 2: 2}
Joueurs ayant atteint leur objectif  [2]

Random :
[v3, v2, v0, v0, v0, v0, v0, v0, v0, v0, v0,
v0, v0, v0, v0, v0, v0, v0, v0, v0, v0, v0 ,v0, v0,
v0, v0, v0 , v0, v0, v0, v0, v0 ,v0, v0, v0, v0, v0,
v0, v0, v0, v0, v0 , v0, v0, v0, v0, v0, v0, v0, v0]
Est un EN?  True
Information sur l'outcome :
```

```
Cout pour chaque joueur:  {1: inf, 2: 2}
Joueurs ayant atteint leur objectif  [2]
```

Puisque depuis le noeud v_0 la seule action possible est de boucler sur v_0, ce comportement est normal.

Rajoutons des noeuds à ce graphe afin que celui-ci soit un graphe fortement connexe. Afin de ne pas surcharger le graphe, les arcs qui ne possèdent pas d'étiquette sont des arcs de poids 1.



Les résultats obtenus sont les suivants :

```
A_star : [v3, v4, v5, v6, v5, v4, v3, v0]
Est un EN?  True
Information sur l'outcome :
Cout pour chaque joueur:  {1: 3, 2: 12}
Joueurs ayant atteint leur objectif  [1, 2]

Random :  [v3, v4, v5, v6, v5, v4, v3, v2, v1, v0,
v2, v3,v0,v2,v1,v0, v2, v0, v2, v1, v0, v2, v3, v4,
v5, v4, v5,v4, v3, v2, v1, v0, v2, v1, v0, v2, v0,
v2, v1, v0, v2, v1, v0, v2, v1, v0, v2, v0,v2, v0,
v2,v3, v2, v3, v2, v1, v0, v2, v1,v0, v2, v0, v2,
v1, v0, v2, v1, v0, v2]
Est un EN?  True
Information sur l'outcome :
Cout pour chaque joueur:  {1: 3, 2: 13}
Joueurs ayant atteint leur objectif  [1, 2]
```

Le résultat retourné par l'algorithme A^* est meilleur que celui de la méthode aléatoire. En effet, pour le premier la somme des coûts des joueurs est de 15 tandis que pour le second 16.

Comparaison des résultats

Afin de pouvoir comparer nos différentes méthodes, une série de tests est générée. Nous avons exécuté nos tests comme suit : nous générons aléatoirement 100 jeux qui respectent les conditions explicitées dans le section 5 et dont le graphe du jeu possède 5 sommets. Ensuite, les quatre méthodes auxquelles nous nous sommes intéressées sont exécutées sur chacun de ces jeux en permettant un temps d'exécution de 30 secondes pour chaque méthode et en générant 100 chemins différents pour la méthode aléatoire. Puisque nous voulons tester l'efficacité du best-first search avec l'utilisation de la fonction d'évaluation de type A^*, qui nous semble être l'approche la plus aboutie, chaque résultat obtenu est comparé avec celui de cette méthode. Nous avons alors cinq possibilités, le résultat d'A^* est :

- meilleur ;
- pire ;
- égal (le même outcome est retourné pour les deux méthodes) ;
- équivalent (un outcome différent est retourné pour les deux méthodes, mais le nombre de joueurs ayant atteint leur objectif ainsi que la somme des coûts pour ceux-ci sont égaux) ;
- échec pour les deux méthodes (une méthode échoue si elle n'a pas trouvé d'équilibre de Nash endéans le temps imparti).

En plus de cela, le temps nécessaire pour l'exécution des 100 recherches pour chaque méthode est calculé.
Les résultats de ces tests sont repris dans le tableau 3 et à la figure 22.

TABLE 3 – Comparaison des différentes méthodes

Méthode	EN trouvés	EN manqués	Temps moyen de recherche (s)
A^*	100	0	0,0028
Méthode aléatoire	97	3	1,135
Best-first search initialisé	95	5	6,6105
Breadth-first search	100	0	0,0054

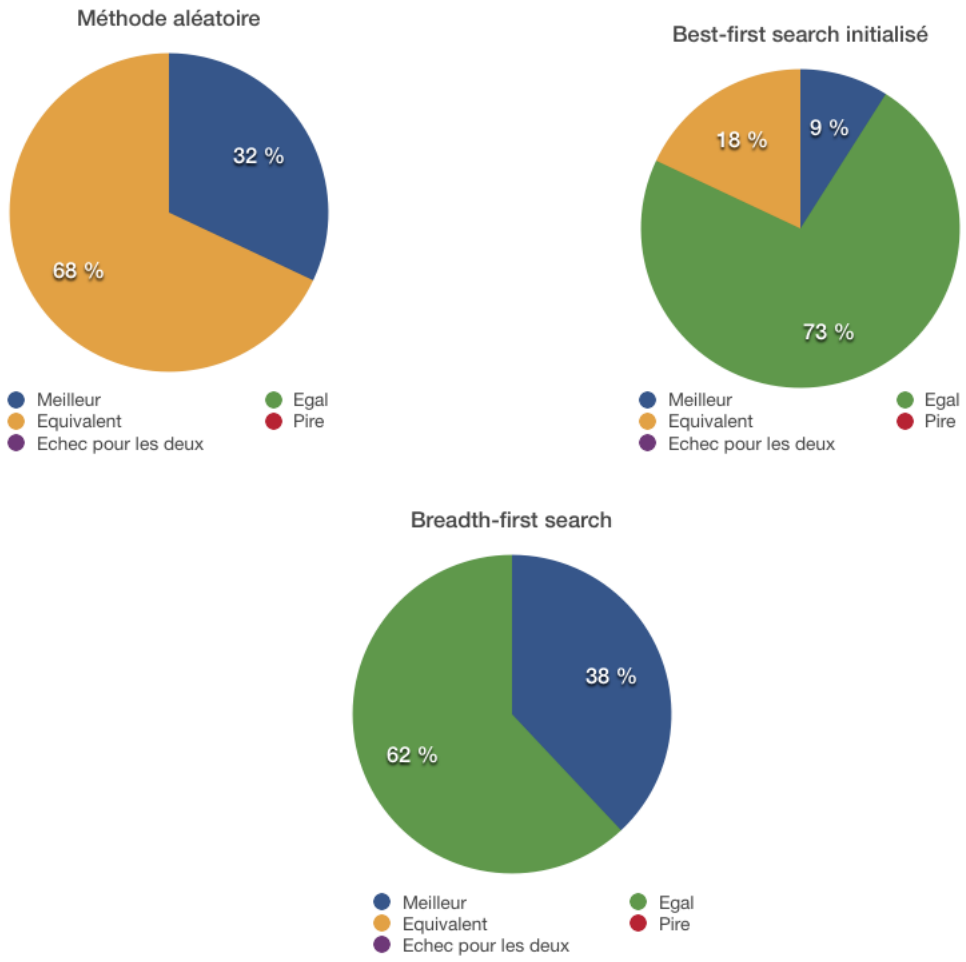


FIGURE 22 – Comparaison d'A^* avec les trois autres méthodes

Bien que peu de tests permettant de comparer les différentes approches aient été effectués, nous pouvons toutefois tirer quelques conclusions :

- La méthode best-first search en utilisant la fonction d'évaluation A^* est celle la plus aboutie. En effet, au vu des résultats sur les graphes complets, un équilibre est trouvé en moins de 30 secondes dans plus de 90% des tests. De plus, si l'on regarde les tests de comparaison avec les autres méthodes, cette méthode est meilleure aussi bien au point de vue du temps nécessaire pour trouver un équilibre de Nash, que pour la qualité de celui-ci.
- La méthode aléatoire trouve généralement un équilibre de Nash mais rien ne garantit l'optimalité de celui-ci. De plus, dans certains cas, elle peut ne retourner aucun résultat.
- La méthode de best-first search initialisé, qui est une des premières approches que nous avons testée, est celle qui nous garantissait les meilleurs résultats avant que nous ne changions la fonction d'évaluation de notre best-first search. Elle comporte divers inconvénients. Puisque l'on teste l'initialisation du chemin jusqu'au premier noeud initial, que l'on attend un certain laps de temps puis seulement que l'on relance la recherche, cette approche peut s'avérer coûteuse en temps. De plus, elle ne trouve pas toujours d'équilibre de Nash dans le temps imparti.
- Si la méthode breadth-first search est utilisée en renvoyant le premier équilibre trouvé alors, puisqu'elle parcourt les chemins par taille croissante, cette recherche n'est pas optimale. En effet, il se peut qu'un chemin d'une plus longue taille mais de poids moindre soit préférable à un chemin plus court. Il en va de même, si on récupère tous les équilibres trouvés pendant un certain laps de temps. De plus, pour cette dernière procédure, le temps d'exécution est toujours égal au temps d'exécution permis.

Remarquons que les temps de calcul repris dans les tableaux [2–3] sont les temps nécessaires pour que les algorithmes retournent une solution et non les temps CPU. Ces données sont calculées afin de nous faire une idée sur l'ordre de grandeur du temps nécessaire à l'exécution de nos méthodes. Toutefois, pour une étude plus fine, la réalisation de tests tenant compte du temps CPU sertait plus pertinente.

6 Conclusion

Le but de notre travail était de mettre au point un processus algorithmique permettant de trouver le plus rapidement possible un équilibre de Nash pour les jeux sur graphe multijoueurs avec coût et objectif d'atteignabilité.

Pour ce faire, un critère permettant de déterminer si un chemin infini dans le graphe du jeu correspond à l'outcome d'un équilibre de Nash a été prouvé par nos soins. Afin de pouvoir appliquer ce critère, la valeur de chaque noeud du jeu, et ce pour chaque jeu où un joueur joue contre la coalition des autres joueurs, est requise. Nous avons donc implémenté une variante de l'algorithme de Dijkstra afin de récupérer ces valeurs. Ensuite, nous nous sommes interrogés sur ce qu'était pour nous la notion d'équilibre de Nash pertinent. Une fois ce concept défini et après avoir réfléchi à la manière d'appliquer notre critère d'équilibre, un algorithme ne pouvant pas s'exécuter sur un chemin infini, une implémentation et une expérimentation de diverses méthodes de recherches ont été effectées. Nous avons retenu la méthode aléatoire, le breadth-first search, le best-first search associé à une fonction d'évaluation de type A^* et le best-first search initialisé. Enfin, nous avons réalisé quelques tests sur des arènes de jeux particuliers (graphe complet, 2 joueurs, un objectif différent par joueur et fonction de poids du graphe à valeurs dans $[1, 100] \cap \mathbb{N}$). Ces tests nous ont permis de nous convaincre que la méthode A^* était la plus aboutie.

Faiblesses et perspectives d'amélioration

Certaines questions pour lesquelles nous n'avons pu mener notre réflexion à son terme se sont présentées :

- Existence d'un équilibre de Nash où tous les joueurs visitent leur objectif pour les jeux dont le graphe est fortement connexe (question 1 p. 49) ;
- Choix des bornes pour les longueurs et les poids maximum des chemins à tester pour trouver les équilibres (p. 20 et p.60) ;
- Optimalité de la méthode A^* (p. 60).

De plus, comme la partie implémentation de notre travail visait essentiellement à tester nos approches, arriver à des résultats était l'objectif principal. C'est pourquoi, le code proposé gagnerait à être amélioré. En effet, nous gardons parfois en mémoire plusieurs fois les mêmes informations ou calculons des données ayant déjà été calculées au préalable. Un bon compromis temps d'exécution et allocation mémoire doit être étudié. De plus,

certains de nos algorithmes ne fonctionnent que si chaque joueur ne possède qu'un seul objectif et que les objectifs de tous les joueurs sont différents.

En outre, nous n'avons eu le temps de n'effectuer que peu de tests. D'autres tests, en se fiant cette fois sur le temps CPU, seraient donc bénéfiques afin de se convaincre de la performance de nos résultats sur un plus grand échantillon de tests. Il serait également intéressant d'étudier le comportement de nos approches sur d'autres types de jeux sur graphe avec coût et objectif d'atteignabilité (par exemple : plus de deux joueurs, graphe non complet, etc).

Enfin, d'autres types de méthodes que les méthodes d'exploration pourraient être envisageables. Nous pensons, par exemple, à l'application de métaheuristiques telles que le hill-climbing.

Bibliographie

[1] Brihaye, T., De Pril, J., and Schewe, S. Multiplayer cost games with simple nash equilibria. In *Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings* (2013), S. N. Artëmov and A. Nerode, Eds., vol. 7734 of *Lecture Notes in Computer Science*, Springer, pp. 59–73.

[2] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[3] De Pril, J. *Equilibria in Multiplayer Cost Games*. PhD thesis, UMons, 2013.

[4] Haddad, A. Characterising nash equilibria outcomes in fully informed concurrent games, 2016. Disponible à l'adresse http ://-web1.ulb.ac.be/di/verif/haddad/H16.pdf.

[5] Olbregts, S. Plus courts chemins dans un graphe pondéré, 2014-2015. Projet de 1er Master en sciences informatiques à l'Umons.

[6] Russel, S., and Norvig, P. *Artificial Intelligence : A Modern Approach*, 3rd ed. Pearson Education, 2010, ch. 3.

Annexe A : Algorithme de Dijkstra

Cette section se base sur le livre de Cormen *et al.* [2] (pages 658–662).

Soit $G = (V, E)$ un graphe orienté et pondéré, à partir d'un sommet s donné (appelé *source*) l'algorithme de Dijkstra permet de calculer les plus courts chemins à partir de s vers les autres sommets du graphe. Cet algorithme s'applique sur des graphes orientés pondérés tels que la fonction de poids w associée au graphe vérifie la propriété suivante : pour tout $(u, v) \in E$ on a que $w(u, v) \geqslant 0$.

Idées de l'algorithme :

- A tout sommet s' de V on associe une *valeur* d qui représente l'estimation du plus court chemin de s à s'. Cette valeur est mise à jour en court d'exécution de l'algorithme afin qu'à la fin de celle-ci d soit exactement le poids du plus court chemin de s à s'. On initialise l'algorithme de Dijkstra en mettant la valeur $+\infty$ à tous les sommets et 0 à s. En effet, le plus court chemin pour aller de s à s est de rester en s.
- On utilise une *file de priorité* Q (structure de données permettant de stocker des éléments en fonction de la valeur d'une clef) qui permet de stocker les sommets classés par leur valeur d. Sur cette file de priorité, on peut effectuer les opérations suivantes : insertion d'un élément, extraction d'un élément ayant la clef de la plus petite valeur, test de la présence ou non d'élément dans Q, augmentation ou diminution de la valeur de la clef associée à un sommet. A l'initialisation de l'algorithme les sommets présents dans Q sont tous ceux présents dans V.
- Afin de pouvoir retrouver un plus court chemin de s à un autre sommet s', chaque sommet stocke le *prédécesseur* qui a permis de constituer ce plus court chemin.
- On maintient $S \subseteq V$ un ensemble de sommets qui vérifient la propriété suivante : pour tout sommet $s' \in V$ le plus court chemin de s à s' a déjà été calculé. A l'initialisation de l'algorithme $V = \emptyset$.
- De manière répétée :
  (a) On sélectionne un sommet $u \in V \backslash S$ associé à l'approximation minimum du plus court chemin de s à u.
  (b) On ajoute u à S.

(c) On *relaxe* tous les arcs sortant de u.

- La *relaxation* des arcs sortant de u consiste à vérifier pour tout u' tq $(u, u') \in E$ qu'il n'existe pas un plus court chemin de s vers u' que celui potentiellement déjà calculé et tel que ce nouveau chemin est de la forme $s...uu'$. Si on trouve un tel nouveau chemin, alors on procède à la mise à jour du prédécesseur de u' (qui devient en fait u).

Pseudo-code

Maintenant que les grandes idées de l'algorithme ont été expliquées, retranscrivons le pseudo-code. Pour des questions de complexité, l'ensemble des arcs du graphe sont représentés sous la forme d'une *liste d'adjacence*[4] et la file de priorité est implémentée par un *tas (heap min)*. L'algorithme 10 représente le pseudo-code de l'algorithme proprement dit, comme explicité dans le livre de Cormen *et al.* [2]. Les algorithmes 11,12,13 décrivent les algorithmes qui sont appelés au sein de l'algorithme 10.

Algorithme 10 DIJKSTRA(G,W,S)

ENTRÉE(S): $G = (V, E)$ un graphe orienté pondéré où E est représenté par sa liste d'adjacence, $w : E \to \mathbb{R}^+$ une fonction de poids,s le sommet source.

SORTIE(S): / **EFFET(S) DE BORD :** Calcule un plus court chemin de s vers les autres sommets du graphe.

```
1: INITIALISER-SOURCE-UNIQUE(G, s)
2: S ← ∅
3: Q ←INITIALISER-TAS(G)
4: tant que Q ≠ ∅ faire
5:    u ← Q.EXTRAIRE-MIN()
6:    S ← S ∪ {u}
7:    pour tout v ∈ Adj[u] faire
8:       RELAXER(u, v, w)
9:    fin pour
10: fin tant que
```

La complexité de l'algorithme de Dijkstra dépend de la manière dont est implémentée la file de priorité. Si on implémente celle-ci de telle sorte que

4. Une liste d'adjacence,Adj, est définie comme telle : à chaque case d'un tableau est associé un sommet de V et à chacun de ces sommets est associée la liste de ses successeurs. $Adj[u]$ permet de récupérer la liste des successeurs du sommet u.

Algorithme 11 INITIALISER-SOURCE-UNIQUE(G, s)

ENTRÉE(S): G un graphe orienté pondéré
SORTIE(S): / **EFFET(S) DE BORD :** initialise les valeurs de tous les sommets.

```
1: pour tout v ∈ G.V faire
2:     v.d ← +∞
3:     v.pred ← NULL
4: fin pour
5: s.d = 0
```

Algorithme 12 INITIALISER-TAS(G)

ENTRÉE(S): G un graphe orienté pondéré
SORTIE(S): Un tas Q qui comprend tous les sommets de V classés par leur valeur d.

```
1: Q ← nouveau tas
2: pour tout v ∈ G.V faire
3:     Q.INSÉRER(v)
4: fin pour
5: retourner Q
```

Algorithme 13 RELAXER(u, v, w)

ENTRÉE(S): deux sommets u et v, une fonction de poids $w : E \to \mathbb{R}^+$.
SORTIE(S): / **EFFET(S) DE BORD :** Met potentiellement à jour la valeur de v et son prédécesseur.

```
1: nouvVal ← w(u, v) + u.d
2: si nouvVal < v.d alors
3:     Q.DÉCRÉMENTERCLEF(Clef(v), nouvVal)
4:     v.p ← u
5: fin si
```

chaque opération EXTRAIRE-MIN() est en $\mathcal{O}(\log V)$ ainsi que chaque DÉCRÉMENTERCLEF($Clef(v), nouvVal$) alors l'agorithme est en $\mathcal{O}((V + E) \log V)$. Les preuves d'exactitude et de complexité de l'agorithme Dijkstra ne sont pas abordées ici mais se trouvent dans le livre de Cormen *et al.* [2].