Réalisation d'un solveur pour le problème *Rush Hour*

Clément Legrand-Lixon
ENS Rennes
Rennes, France
clement.legrand-lixon@ens-rennes.fr

ABSTRACT

Il existe de nombreux solveurs dans la littérature qui permettent de résoudre de manière efficace des problèmes qu'ils soient d'optimisation, d'élimination de quantificateurs ou *NP-complets*. Cependant peu de solveurs s'intéressent à des problèmes d'autres classes de complexité, comme ceux de la classe *PSPACE* par exemple. Bien qu'il existe un solveur pour le problème *TQBF*, qui est *PSPACE-complet*, il n'est pas toujours évident de réduire un problème de *PSPACE* à ce dernier. On s'intéresse donc ici à la réalisation d'un solveur pour le problème *generalized rush hour* montré comme étant également *PSPACE-complet*. Le solveur présenté ici cherche à prédire les mouvements nécessaires pour aboutir à une solution. Pour réduire le temps de calcul lié aux nombreuses configurations conservées, on en supprime aléatoirement. On compare ensuite notre solveur à l'algorithme classique de la littérature résolvant le *generalized rush hour*.

KEYWORDS

Solveur, *PSPACE-complet*, *Generalized Rush Hour*, *Expérimentations*

Clément Legrand-Lixon. 2018. Réalisation d'un solveur pour le problème *Rush Hour*.

INTRODUCTION

De manière générale, lorsqu'on s'intéresse à un problème, on peut trouver de nombreuses méthodes afin de le résoudre. Un algorithme prenant en entrée une instance d'un tel problème et renvoyant un moyen (i.e. une suite d'exécutions à réaliser) pour résoudre cette instance est appelé un *solveur*. Les solveurs sont très largement répandus dans de nombreux domaines et permettent de résoudre certains problèmes classique. On peut ainsi trouver la méthode du simplexe pour les problèmes d'optimisation convexe, la décomposition cylindrique algébrique pour résoudre les problèmes d'élimination de quantificateurs, ou encore des solveurs *SAT* et *SMT*, permettant de résoudre les problèmes *NP-complets* du même nom.

En ce qui concerne la classe de complexité *PSPACE*, mis à part le problème *TQBF*, peu de problèmes de cette classe disposent d'un solveur. Pour agrandir cette bibliothèque de solveurs, il est donc légitime de s'intéresser au problème du *Generalized Rush Hour*, montré comme étant *PSPACE-complet* [2]. C'est également un problème intéressant à étudier car il est plus facile en général de réduire un jeu à un joueur au problème du *Generalized Rush Hour*, plutôt qu'au problème *TQBF*.

Le problème du *RushHour* consiste à faire sortir un véhicule spécifique (appelé *véhicule cible*) d'un parking contenant de nombreux autres véhicules, et bloquant la voie vers la sortie. Dans la littérature on trouve peu de résultats concernant ce problème. Ces derniers ne s'intéressent souvent qu'au cas où le parking forme un carré de côté 6, taille répandue dans le jeu du même nom, et effectuent une recherche exhaustive pour trouver la solution au problème. Mais il existe aussi des heuristiques performantes sur le problème [3].

On se propose à travers cet article de présenter un nouveau solveur permettant de résoudre le *Rush Hour*. Notre objectif est de réussir à prédire les mouvements à effectuer pour aboutir à une solution. Cela permet notamment de réduire l'arbre de la recherche exhaustive. Toutefois le nombre de configurations à étudier grandissait toujours trop vite (notamment dans le cas d'instances compliquées). Ainsi lorsqu'il y a trop de configurations, on décide de conserver chaque configuration avec probabilité *prob*. Nous essayons ensuite de comparer notre solveur à l'algorithme de recherche exhaustive sur des instances de différentes tailles et générées aléatoirement, puisqu'il n'existe pas vraiment de références dans ce domaine.

Ainsi, on commence par présenter le problème du *Generalized Rush Hour* dans la section 1, ainsi que les quelques résultats que l'on peut trouver dans la littérature. Puis la section 2 décrit le solveur réalisé pour résoudre le problème, en commençant par décrire un algorithme naïf, amélioré par la suite. Enfin, nous comparons dans la section 3 les résultats obtenus avec l'algorithme exhaustif et le solveur crée.

1 ÉTAT DE L'ART

1.1 Description du problème

Le problème du *Generalized Rush Hour* (que l'on notera *GRH* dans la suite), est un jeu de plateau à un joueur. Il consiste en un parking (*grille*) rectangulaire, où sont situés de nombreux véhicules, parmi lesquels un véhicule cible, et une sortie (sur le bord du plateau). Les véhicules sont de longueur 2 ou 3, et ont une certaine direction (horizontale ou verticale), qu'il n'est pas possible de changer au cours de la résolution. L'objectif

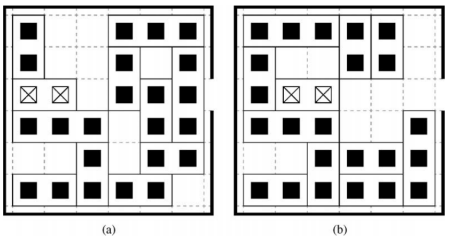

FIGURE 1: Une configuration initiale du *GRH* (a) et une configuration gagnante (b). Le véhicule cible est représenté avec des croix.

est alors de réaliser une suite de mouvements permettant de faire sortir le véhicule cible du parking.

L'article [2] montre que ce problème est PSPACE-complet, en procédant à une réduction à *TQBF*, et le définit de la manière suivante :

Definition 1. Une *instance* de *GRH* est un tuple $\langle w, h, x, y, n, \mathcal{C} \rangle$ tel que :
- $(w, h) \in \mathbb{N}^2$ sont respectivement la largeur et la hauteur de la grille ;
- $(x, y) \in \mathbb{N}^2$ sont les coordonnées de la sortie, qui doit se trouver sur le périmètre de la grille ;
- $n \in \mathbb{N}$ est le nombre de véhicules présents dans la grille (en ne comptant pas le véhicule cible) ;
- $\mathcal{C} = \{c_0, ..., c_n\}$ est un ensemble de $n+1$ tuples caractérisant chaque véhicule : $c_i = \langle x_i, y_i, o_i, s_i \rangle$ avec :
  - $(x_i, y_i) \in \mathbb{N}^2$ les coordonnées du véhicule ;
  - $o_i \in \{N, S, E, W\}$ l'orientation du véhicule ;
  - $s_i \in \{2, 3\}$ est la longueur du véhicule ;
  - c_0 correspond au véhicule cible.

  Il est important de préciser que $\mathcal{C}$ doit être cohérent, dans le sens où tous les véhicules doivent être intégralement placés dans la grille, et deux véhicules distincts ne peuvent être superposés.

Le même article définit également une solution au problème *GRH* :

Definition 2. Une *solution* du problème *GRH* consiste en une séquence de m mouvements, où chaque mouvement est décrit par un numéro de véhicule, i, une direction cohérente avec l'orientation du véhicule c_i et une distance (correspondant au nombre de cases parcourues par le véhicule). Chaque mouvement doit amener à une configuration $\mathcal{C}$ cohérente, et être cohérent avec la configuration précédant ce mouvement. De plus, pour qu'un véhicule puisse se déplacer d'une distance d, les configurations obtenues pour tout d', $0 \leq d' \leq d$ doivent être cohérentes (par conséquent la "téléportation" à travers les obstacles n'est pas autorisée). Si $d = 1$, le mouvement est appelé *élémentaire*.

La figure 1 représente une instance de taille 6 et sa configuration gagnante associée.

Il existe deux variantes du *GRH* : le *path GRH* où on s'intéresse à la liste des mouvements à effectuer pour obtenir une solution, et le *decisional GRH* où on ne s'intéresse qu'à l'existence d'une solution. On ne s'intéressera dans la suite qu'à la variante *path GRH* (ce qui permet aussi de résoudre *decisional GRH*, mais certainement de manière moins efficace). Aussi on ne s'intéressera dans la suite qu'à des mouvements élémentaires, car ils sont plus simples à modéliser.

On définit également formellement ce que sont les configurations accessibles depuis une configuration c_{init} :

$$Conf_{accessibles}(g_{init}) = \{g \mid \exists\ move, \\ g = apply\_move(g_{init}, move)\}$$

La fonction apply_move applique un mouvement élémentaire passé en argument à la configuration donnée et renvoie la nouvelle configuration obtenue.

Une configuration est dite *atteignable* depuis une certaine configuration c_{init} si cette configuration peut être obtenue en réalisant une suite de mouvements élémentaires depuis la configuration c_{init}.

On commence par définir l'ensemble des configurations atteignables en au plus i mouvements élémentaires de la manière suivante :

$$Conf^0_{atteignables}(g_{init}) = \{g_{init}\}$$

$$Conf^i_{atteignables}(g_{init}) = \cup_{g \in Conf^{i-1}_{atteignables}(g_{init})} Conf_{accessibles}(g)$$

On en déduit alors l'ensemble des configurations atteignables :

$$Conf_{atteignables}(g_{init}) = \cup_{i \in \mathbb{N}} Conf^i_{att}(g_{init})$$

1.2 Résolution classique

Les méthodes de résolutions proposées dans la littérature se résument généralement à une recherche exhaustive en effectuant un parcours en largeur.

Cette recherche exhaustive se fait de la manière suivante : depuis la configuration actuelle, on calcule l'ensemble des configurations accessibles. De ce fait, si on peut réaliser 2 mouvements élémentaires dans la configuration actuelle, on obtient 2 nouvelles configurations à étudier (si ce ne sont pas des configurations qui ont déjà été vues). On recommence alors pour chaque nouvelle configuration obtenue, jusqu'à trouver une solution, ou atteindre la profondeur de recherche (i.e. le nombre maximal de déplacements autorisés) choisie. Sans cette limite l'algorithme peut ne pas terminer, surtout s'il n'y a pas de solutions.

D'autres méthodes de résolution utilisent des heuristiques variés, dont certaines sont décrites dans l'article [3] et bien qu'elles semblent être efficaces elles n'ont pas pu être reproduites ici.

La plupart du temps les auteurs se restreignent aux instances proposées dans les jeux (i.e. la grille correspond

à un carré de taille 6). Le temps de calcul requis est toutefois trop long (la complexité en temps est exponentielle en la taille de l'entrée) pour de plus grandes configurations, et cela se ressent d'autant plus si la solution possède un grand nombre de déplacements. Cette solution peut d'ailleurs être de taille exponentielle en la taille de l'entrée.

1.3 Autres travaux

La littérature s'est aussi intéressée au calcul de la configuration la plus difficile (au sens où sa solution nécessite un grand nombre de déplacements) du *Rush Hour* dans sa version classique (i.e. dans le cas où on a une grille carrée de taille 6), ce que l'on notera par la suite *Classical Rush Hour* (*CRH*).

Pour déterminer cette configuration, les articles [4] et [1] ont essayé de classifier les différentes instances du problèmes en plusieurs groupes. Deux instances font partie d'un même groupe si en partant de l'une des configurations on peut atteindre l'autre en effectuant une suite de mouvements (i.e. si l'une est atteignable depuis l'autre). Ils ont ensuite utilisé des structures abstraites pour représenter ces données (par le biais de *Reduced Ordered Binary Decision Diagram* (*ROBDD*) notamment).

Cette instance est néanmoins très utile car elle permet de tester l'efficacité des algorithmes proposées, ainsi que leur précision (la solution renvoyée est-elle optimale ? autrement dit minimise-t-elle le nombre de déplacements à effectuer ?) On utilisera cette instance par la suite pour comparer l'efficacité des différents algorithmes.

2 PRÉSENTATION DU SOLVEUR

2.1 Construction du solveur

Si la recherche exhaustive permet assez facilement de résoudre le problème dans le cas d'une grille carrée de taille 6, cela devient assez compliqué pour des grilles de grandes tailles, notamment à cause du nombre de configurations atteignables. Nous décrivons plusieurs idées pour réduire le nombre de configurations à explorer.

Première idée. Une première idée pour réduire cet ensemble de possibilités a été de déplacer uniquement les véhicules qui allaient permettre de débloquer le véhicule cible. S'il n'était pas possible de choisir un tel véhicule alors on choisissait de bouger un véhicule qui allait en libérer un autre. Aussi dès que c'était possible on faisait avancer le véhicule cible vers la sortie. Toutefois cette solution s'est avérée beaucoup trop simpliste dans le sens où, il faut parfois de nombreux mouvements avant de pouvoir déplacer le véhicule cible vers la sortie.

Deuxième idée. Une deuxième idée était de déterminer l'ensemble des véhicules dont dépendait le véhicule cible. Cet ensemble se construisait de la manière suivant : on ajoute d'abord l'ensemble des véhicules qui bloquent actuellement le véhicule cible (cet ensemble contient au plus deux éléments).

$$dependance_0(c) = \{c' | c' \text{ bloque } c\}$$

S'il n'y en a pas, alors l'ensemble est vide.

Les ensembles suivants sont alors obtenus en prenant chaque véhicule de l'ensemble actuel et en ajoutant chaque nouveau véhicule (i.e. qui n'appartient à aucun ensemble précédent) qui les bloquent.

$$dependance_i(c) = \{c_i \mid \exists c_{i'} \in dependance_{i-1}(c), c_i \text{ bloque } c_{i'}\}$$

On s'arrête lorsqu'il n'y a plus de nouveaux véhicules à ajouter.

$$dependance(c) = \cup_{i \in \mathbb{N}} dependance_i(c)$$

Une fois cet ensemble construit, on cherchait parmi les mouvements possibles ceux qui permettaient de déplacer l'un des véhicules de l'ensemble. Cette méthode a été plus efficace que la première, mais une configuration avec un cycle pouvait poser problème, et empêcher l'algorithme de terminer.

Troisième idée. C'est pourquoi nous avons employé cette troisième méthode, sur laquelle est basée notre solveur. Dans les méthodes précédentes, le véhicule cible jouait un rôle privilégié, mais on s'est rendu compte que selon l'objectif secondaire que l'on veut atteindre (e.g. pour bouger le véhicule c_i alors il faut bouger le véhicule $c_{i'}$), il était plus intéressant de privilégier le véhicule concerné par l'objectif actuel. Ici notre objectif principal (amener le véhicule cible à la sortie) est découpé en plusieurs sous objectifs. Ces nouveaux objectifs sont déterminés de la manière suivante :

- Nous cherchons actuellement à atteindre l'objectif i $obj_i = (c, p)$, avec c le véhicule à bouger et p la position dans la grille qu'il doit atteindre. C'est l'objectif i au sens où, l'on doit actuellement atteindre i objectifs pour pouvoir faire avancer notre véhicule cible ;
- Nous calculons ensuite l'ensemble $dependance_0(c)$, contenant au plus un véhicule (en effet p contient la direction dans laquelle doit bouger le véhicule pour atteindre l'objectif) ;
- Si $dependance_0(c)$ est vide, alors on avance c dans la bonne direction, on regarde si l'objectif est atteint, si oui alors cet objectif est supprimé et on retourne à l'objectif $i-1$. Sinon on recommence à l'étape précédente ;
- Si $dependance_0(c)$ est non vide, il contient un seul véhicule c'. Selon la taille de c' et la position de c dans la grille, on détermine au plus deux emplacements p_1 et p_2 que doit atteindre c' pour pouvoir nous rapprocher de notre objectif obj_i. Cela crée alors deux nouveaux objectifs (c', p_1) et (c', p_2). Il faut alors vérifier que ces objectifs ne sont pas déjà parmi les objectifs $obj_1, ..., obji - 1$, et que leur réalisation ne demande de réaliser l'un des i objectifs actuels. S'il n'est pas possible de trouver au moins un nouvel objectif, c'est l'objectif

était impossible à atteindre, on supprime donc la liste d'objectif associée.

On reprend alors les différentes étapes depuis le début, tant qu'on a des objectifs à accomplir.

L'un des avantages de cette méthode est de ne créer qu'au plus deux nouveaux objectifs à chaque étape, permettant d'explorer beaucoup moins de configurations qu'avec une recherche exhaustive classique. Néanmoins sur les configurations les plus difficiles à résoudre, il reste toujours trop de configurations à étudier. Nous verrons donc un peu plus tard, une amélioration de cette méthode pour accélérer la recherche.

Cette méthode est illustrée dans la section suivante.

2.2 Exemple

Nous allons illustrer la troisième méthode décrite précédemment au travers d'un exemple simple. La grille de départ est représenté par le tableau suivant.

4	4	5	5	5
0	0	0	0	0
0	1	1	2	0
0	0	0	2	0
0	3	3	3	0

La sortie est placée en $(5,3)$ en face du véhicule rouge (le véhicule cible) après le véhicule jaune. Notre objectif initial est donc $(1,(5,3))$. Comme $dependance_0(1) = \{2\}$, on a à priori (vu la position de 1, la taille de 2 et la taille de la grille) deux nouveaux objectifs : $(2,(4,4))$ et $(2,(4,1))$ (on prend en compte la case la plus en bas (resp à gauche) du véhicule s'il est vertical (resp horizontal) pour atteindre un objectif).

Considérons l'objectif $(2,(4,4))$. Aucun véhicule ne bloque 2, on peut donc le faire avancer. Ensuite on trouve que le véhicule 5 bloque le véhicule 2. On essaie alors de trouver un nouvel objectif pour le véhicule 5. On ne va pas en trouver car le véhicule 4 bloque le 5 et qu'il est aussi horizontal. Ainsi l'objectif $(2,(4,4))$ est impossible. Cet objectif est alors supprimé.

Considérons alors l'objectif $(2,(4,1))$. Le véhicule 3 bloque le 2. On trouve alors comme nouvel objectif $(3,(1,1))$. Cet objectif peut directement être atteint car aucun véhicule ne bloque 3.

L'objectif $(3,(1,1))$ ayant été réalisé, on s'intéresse de nouveau à l'objectif $(2,(4,1))$. Cet objectif est réalisable immédiatement, puisque 3 ne bloque plus 2. Puisque cet objectif est réalisé, on s'intéresse maintenant à notre objectif principal $(1,(5,3))$. Le véhicule n'est plus bloqué jusqu'à la sortie, on peut donc le faire avancer jusqu'à atteindre la sortie.

Nous n'avons plus d'objectif à atteindre, c'est donc que nous avons résolu le problème.

2.3 Amélioration

Bien que chaque objectif à atteindre donne accès à au plus deux nouveaux objectifs (et donc au plus deux nouvelles configurations), on peut rapidement se retrouver avec un très gros nombre de configurations à étudier, et cela peut considérablement allonger le temps de calcul. D'autant plus qu'à partir d'un certain nombre de déplacements, certaines configurations à étudier peuvent être redondantes (cela peut être une ancienne configuration qui n'a plus d'intérêts, ou une configuration qui a déjà été étudié). Ainsi pour réduire le nombre de configurations à étudier, on décide que lorsqu'on a dépassé un certain nombre de configurations *maxConf*, on garde une configuration avec probabilité *prob*.

2.4 Implémentation du solveur

l'algorithme 1 décrit en pseudo-code le fonctionnement du solveur dans les grandes lignes. Il fonctionne de la manière décrite dans la troisième idée de la section 2.1.

On implémente le solveur en python. Cela permet une gestion des structures plus simple, bien que plus lente, ainsi qu'une phase de débogage plus rapide.

La grille est gérée comme une matrice à h lignes et w colonnes. On représente un véhicule dans cette grille par son identifiant. L'identifiant 1 est réservé au véhicule cible. Un emplacement libre est représenté par un 0. Par convention, lorsqu'un véhicule horizontal est placé devant le véhicule cible, il est impossible de trouver une solution. Pour repérer chaque véhicule dans la grille, on dispose d'une table qui associe à chaque id, s'il se trouve sur une ligne ou une colonne, le numéro de cette ligne ou colonne, et sa taille (2 ou 3).

Pour plus de précision le code est fourni dans le fichier solveur.py.

Nous allons à présent nous intéresser aux performances de l'algorithme, à l'influence des paramètres $maxConf$ et *prob* sur le temps d'exécution et la précision du résultat obtenu. Puis nous le comparerons à un algorithme trouvé sur internet.

3 EXPÉRIMENTATIONS

On décide de ne s'intéresser qu'à des instances où la grille a une forme carrée (i.e. $w = h$). On considère uniquement des instances où la sortie est située sur la même ligne que le véhicule cible (sinon il n'y a pas de solution). On génère des instances de taille w aléatoirement.

3.1 Influence des paramètres $maxConf$ et $prob$

Pour évaluer l'influence des paramètres $maxConf$ (qui limite le nombre maximal de configurations à conserver simultanément) et *prob* (qui permet de décider si on garde une configuration avec probabilité *prob*) on exécute le solveur

Algorithm 1: SOLVEURRH résout le problème du *Rush Hour* donné en entré

Input: Une grille g représentant le problème à résoudre, $depth$ la profondeur de recherche maximale, $maxConf$ est le nombre maximal de configurations que l'on peut conserver simultanément, $prob$ est la probabilité de conserver une configuration donnée

Output: Une liste de mouvements permettant de résoudre le problème et -1 si le problème est impossible ou si l'algorithme n'a pas trouvé de solution

```
objInit ← définir l'objectif initial
listObj ← [objInit]
allObj ← [(g, listObj, 0)]
currentDepth ← 0
while currentDepth < depth do
    (grid, listObj, cpt, sol) ← premier élément de allObj
    (car, position) ← dernier élément de listObj
    calculer dependance_0(car)
    if dependance_0(car) est vide then
        grid ← avancer car vers position
        sol ← ajouter le mouvement à sol
        cpt ← cpt + 1
        currentDepth ← cpt
        if position est atteinte then
            supprimer le dernier élément de listObj
            if listObj est vide then
                return sol
            if |allObj| > maxConf then
                allObj ← conserver chaque configuration
                  avec probabilité prob
        else
            nextCar ← dependance_0(car)[0]
            nextObj ← déterminer les nouveaux
              objectifs
            nextObj ← supprimer de nextObj les
              objectifs infaisables et redondants par
              rapport à listObj
            allObj ← supprimer le premier élément de
              allObj
            if nextObj est non vide then
                for obj in nextObj do
                    nextListObj ← ajouter obj à la fin de
                      listObj
                    allObj ← ajouter
                      (grid, nextListObj, cpt, sol) à la fin
                      de allObj
```

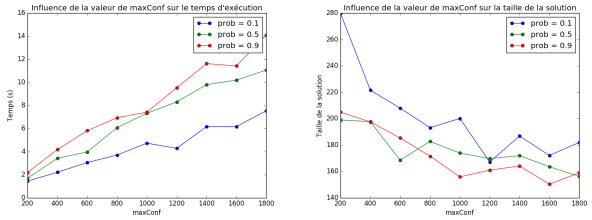

FIGURE 2: Influence du paramètre $maxConf$.

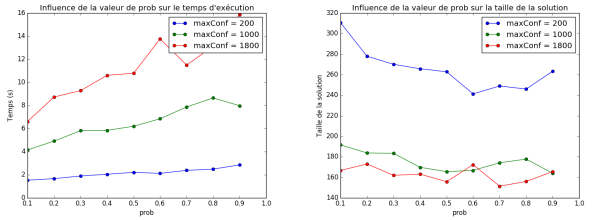

FIGURE 3: Influence du paramètre $prob$.

sur l'instance la plus difficile en taille 6. Cette instance a une solution de longueur minimale 93.

On représente d'abord l'influence de $maxConf$ pour quelques valeurs de $prob$, et on regarde le temps d'exécution ainsi que la taille de la solution renvoyée. Notre solveur étant aléatoire, on prend la moyenne sur 10 exécutions. Les courbes sont présentes en figure 2.

On fait de même pour le paramètre $prob$, avec quelques valeurs de $maxConf$. Les courbes sont présentes en figure 3.

On remarque avec les résultats obtenus qu'on ne peut pas minimiser à la fois le temps d'exécution et la taille de la solution obtenue. Il faut donc faire un compromis entre le temps d'exécution du solveur et la qualité de la solution de la obtenue. On décide pour cela de prendre $maxConf = 1000$ et $prob = \frac{1}{2}$, qui permettent d'avoir un bon équilibre entre temps de résolution et solution obtenue.

3.2 Comparaison

Les instances utilisées dans l'article [3] ne sont plus disponibles, il est donc difficile de réellement comparer notre algorithme à ceux présentés dans cet algorithme. Néanmoins, les résultats obtenus par notre solveur sont moins bons que ceux présentés dans l'algorithme. De plus notre solveur ne parvient pas à retrouver la solution de longueur minimale pour l'instance difficile de taille 6.

Il est cependant clair, que l'algorithme proposé est bien plus efficace qu'une recherche exhaustive.

CONCLUSION

Dans cet article, nous avons proposé un solveur pour résoudre le problème du *Generalized Rush Hour*. Le solveur proposé est bien plus efficace qu'une recherche exhaustive pour trouver une solution. Néanmoins il ne semble pas surpasser les autres algorithmes utilisées pour résoudre ce problème. Le manque de *benchmarks* pour ce problème ne permet pas non plus de donner réellement une idée de son efficacité.

Il pourrait être intéressant d'ajouter certaines heuristiques efficaces présentes dans [3], pour faire des choix plus intéressants au niveau des configurations à éliminer (plutôt que de ne les conserver qu'avec une probabilité $\frac{1}{2}$ par exemple). Aussi l'algorithme crée n'a pas été totalement optimisé, et il se peut qu'il y ait plusieurs calculs inutiles à certains endroits qui fassent perdre du temps.

Pour poursuivre ce travail, il pourrait aussi être intéressant de réaliser quelques *benchmarks* pour pouvoir situer notre solveur, par rapport aux algorithmes existants.

RÉFÉRENCES

[1] Sébastien Collette, Jean-François Raskin, and Frédéric Servais. 2006. On the symbolic computation of the hardest configurations of the Rush Hour game. In *International Conference on Computers and Games*. Springer, 220–233.

[2] Gary William Flake and Eric B Baum. 2002. Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science* 270, 1-2 (2002), 895–911.

[3] Ami Hauptman, Achiya Elyasaf, Moshe Sipper, and Assaf Karmon. 2009. GP-Rush : using genetic programming to evolve solvers for the Rush Hour puzzle. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, 955–962.

[4] Jelle van Assema. [n. d.]. On the Hardness of 6x6 Rush Hour. ([n. d.]).