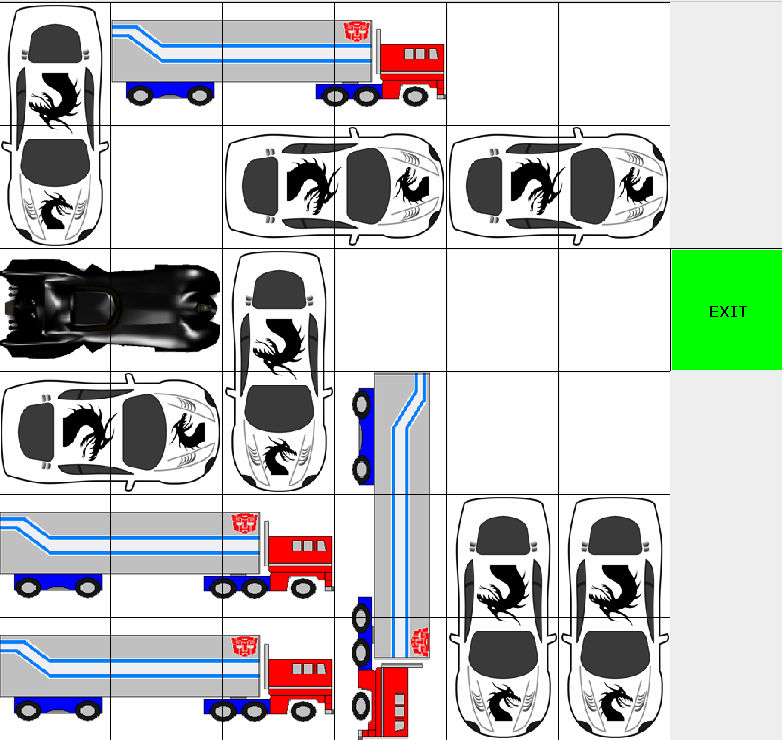
01/12/2016

Projet de MOGPL

Résolution d’un casse-tête Rush Hour



Loïc LAFONTAINE

Laxime LAVASTE

Table des matières

[Introduction 3](#_Toc468415455)

[I. Représentation d’un RushHour 5](#_Toc468415457)

[II. Résolution par programmation linéaire en variables binaires 6](#_Toc468415458)

[III. Résolution par l’algorithme de Dijkstra 10](#_Toc468415459)

[1. Résolution de Rush Hour Cases 10](#_Toc468415460)

[2. Résolution de Rush Hour Mouvements 11](#_Toc468415461)

[IV. Expérimentations numériques 12](#_Toc468415462)

[Conclusion 17](#_Toc468415463)

# Introduction

Dans le cadre de l’enseignement de l’UE MOGPL, nous avons créé un outil de résolution d’un problème de puzzle de Rush Hour.

Le jeu Rush Hour simule une congestion automobile dans un parking à l’heure de pointe (appelée « *rush hour* » en anglais, d’où son nom) ; le but du jeu est d’extraire le véhicule rouge d’une grille dans laquelle plusieurs autres véhicules bloquent la sortie. Il faut pour cela les déplacer, mais ils sont suffisamment entrecroisés et il y a suffisamment de contraintes de déplacement pour que la solution ne soit pas triviale.[[1]](#footnote-1)

Pour résoudre le problème, deux méthodes ont été implémentées, la méthode de résolution par l’algorithme de Dijkstra, ainsi qu’une méthode par programmation linéaire à l’aide du solver Gurobi.

Nous avons décidé d’implémenté ces deux méthodes en Java, puisque c’est un langage efficace pour créer des ensembles d’objets. Python, bien que plus maniable possède une vitesse d’exécution moins bonne. Enfin, nous maitrisons la création d’IHM en Java via la bibliothèque graphique Swing.

Q1 Différence entre les solutions optimales RHC et RHM

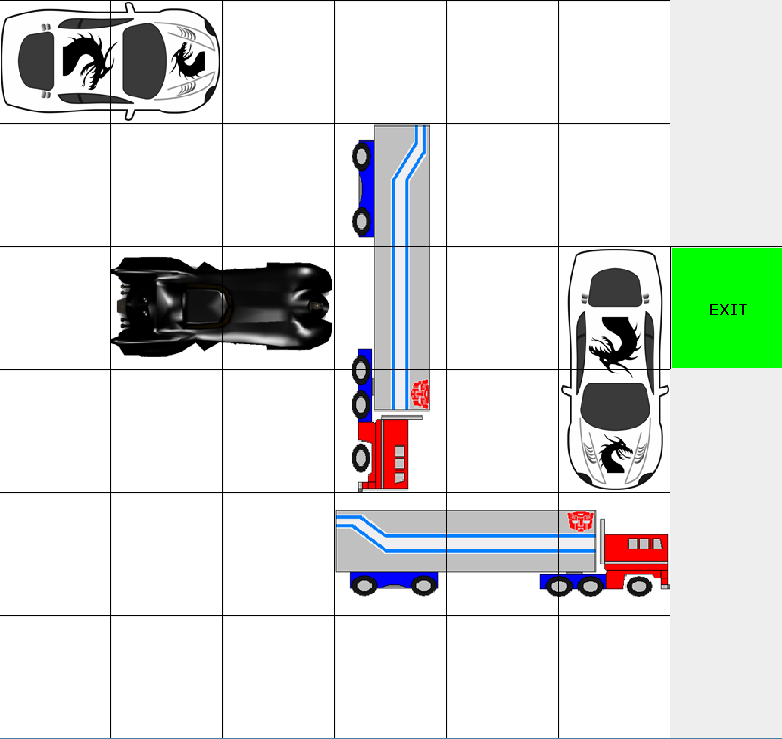


Figure 1RushHour à résoudre

Pour résoudre RHM, on déplace la voiture à droite tout en haut, puis on met le camion de droite tout à gauche, ensuite on baisse le camion du milieu tout en bas. Ensuite notre voiture peut sortir. Nous avons résolu ce Rush Hour en quatre mouvements avec un déplacements de dix cases. Pour résoudre RHC, on bouge d’une case le camion en bas vers la droite, puis la voiture de droite une case vers le bas. On déplace ensuite le camion de deux cases vers la gauche. Puis, l’autre camion deux cases vers le bas. Ensuite notre voiture peut sortir. Nous avons résolu ce Rush Hour en cinq mouvements avec un déplacements de neuf cases. Les deux fonctions renvoient une configuration et une valeur de retour différente, la solution optimale d’un RHM ne correspond donc pas à la solution optimale d’un RHC.

# Représentation d’un RushHour

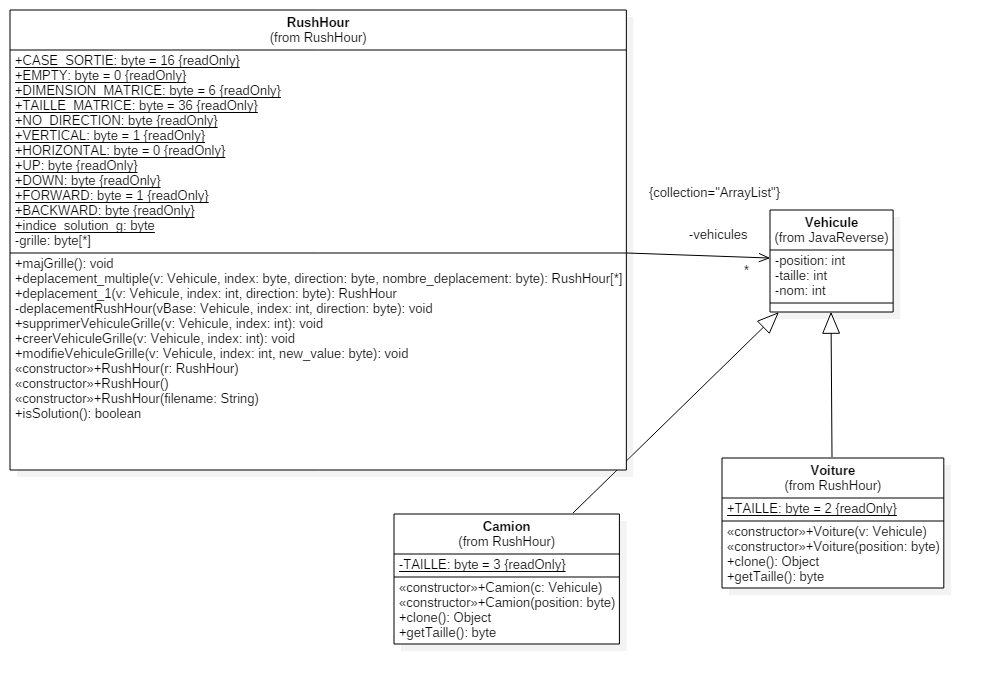


Figure 2 Représentation du Rush Hour dans notre code

Un tableau de 36 bytes permet de représenter notre grille de jeu et les positions des véhicules ainsi que les espaces vides. On transforme les noms de véhicules « c1 », « t1 » etc. en un nombre. Les positions des véhicules sont donc juste codées par la représentation d’un entier.

De plus, on possède une ArrayList contenant les véhicules de la résolution en cours, pour pouvoir connaitre leur taille, la position du marqueur ainsi que son orientation.

# Résolution par programmation linéaire en variables binaires

correspond à la position du marqueur du véhicule , au tour .

correspond aux positions occupées par un véhicule au marqueur au tour .

correspond à un mouvement d’un véhicule , au tour du marqueur vers le marqueur .

le nombre de mouvements possible pour que la voiture rouge atteigne l’arrivée.

Le but de cet exercice est de programmer un programme de résolution linéaire d’un problème de RushHour. Nous allons devoir générer de nombreuses contraintes pour résoudre ce problème.

Question 4 : Déterminer les fonctions objectives

Plusieurs chemins sont possibles pour atteindre la victoire, d’un coût plus ou moins important. Nous devons créer deux formules qui minimisent le coût de déplacement pour RHM ainsi que pour RHC.

On cherche, dans RHM à minimiser le nombre de mouvements. Or, un mouvement correspond à une valeur de . Nous devons donc minimiser cette fonction objective.

contient toutes les positions entre et inclus. Donc si on soustrait à sa taille, on retrouve bien le nombre de cases de déplacement entre la position et la position .

Question 5 : Contraintes additionnelles

Nous avons déjà quatre contraintes données par le sujet.

≤ 1 -

a. La voiture rouge est positionnée devant la sortie au terme du dernier mouvement.

Avec i, l’identifiant de la voiture rouge. En informatique, la 17e case d’un tableau est au 16e élément, donc c’est le 16e élément de notre grille.

b. Au plus un véhicule est déplacé par tour.

La somme des pour un k donné, corresponds au nombre de mouvements effectués pour un tour. On ajoute donc comme contrainte, pour chaque k, la somme de dois être comprise entre .

c. La position du marqueur d’un véhicule i est bien mise à jour si le véhicule i se déplace.

Dans cette formule, nous avons six cas différents, selon la valeur de .

Pour alors, puisqu’il n’y peut pas y avoir de mouvements de .

Pour , on peut avoir un mouvement de , mais aucun de l’intérieur.

De plus, dans notre implémentation, nous avons ajouté des contraintes pour initialiser le problème pour et .

Question 6 : Sous Modèle binaire

En prenant toutes les variables binaires, on admet certaines irréalisables dans une instance de RushHour. Par exemple, la voiture rouge (indexé 1 pour cet exemple) ne peut pas se déplacer verticalement : donc est impossible peu importe le k. Cette variable ne sera pas ajoutée au modèle.

De la même manière, on peut fixer les positions possibles des véhicules et de leur marqueur. En effet, une voiture aura 5 positions de marqueur possible (4 pour un camion) et chaque véhicule peut occuper 2 ou 3 cases (selon la taille du véhicule) parmi les 6 constituant sa grille ou sa colonne, définissable par l’orientation du véhicule.

En supprimant toutes les variables impossibles, irréalisables, on obtient un sous-ensemble de l’ensemble de variables binaires de départ.

NB : Nous aurions pu améliorer la vitesse d’exécution de Gurobi en supprimant aussi les où (pour éviter la prise en compte d’un déplacement sur la même position).

Nous aurions aussi pu tester si 2 véhicules sont sur la même ligne ou sur la même colonne, et possédant la même orientation, ce qui réduit ainsi les cases pouvant être occupées.

# Résolution par l’algorithme de Dijkstra

Cette partie consiste à créer un ensemble de graphes qui représente toute les configurations possibles en bougeant les différents véhicules. Ensuite, nous devons appliquer Dijkstra pour trouver la solution de RHC et RHM.

Nous avons utilisé l’algorithme de Breadth-first search pour générer toutes les configurations de graphes possibles à partir du graphe de départ. Nous utilisons une structure de liste adjacente pour stocker nos arêtes entre chaque graphe différent.

## 1. Résolution de Rush Hour Cases

Question 8 : Trouver la solution

Le poids d’une arrête entre deux sommets représentent le nombre de cases d’un mouvement entre deux configurations différentes. En appliquant l’algorithme de Dijkstra, nous obtenons le plus court chemin pour arriver à une configuration de victoire et donc un chemin avec le moins de cases de déplacements.

À chaque itération de l’algorithme, on marque le sommet (non marqué) qu’on peut atteindre avec la plus petite distance.

Ainsi, la condition d’arrêt pour Dijkstra est le marquage d’un sommet correspondant à une configuration-but, c’est-à-dire quand la voiture rouge est devant la sortie.

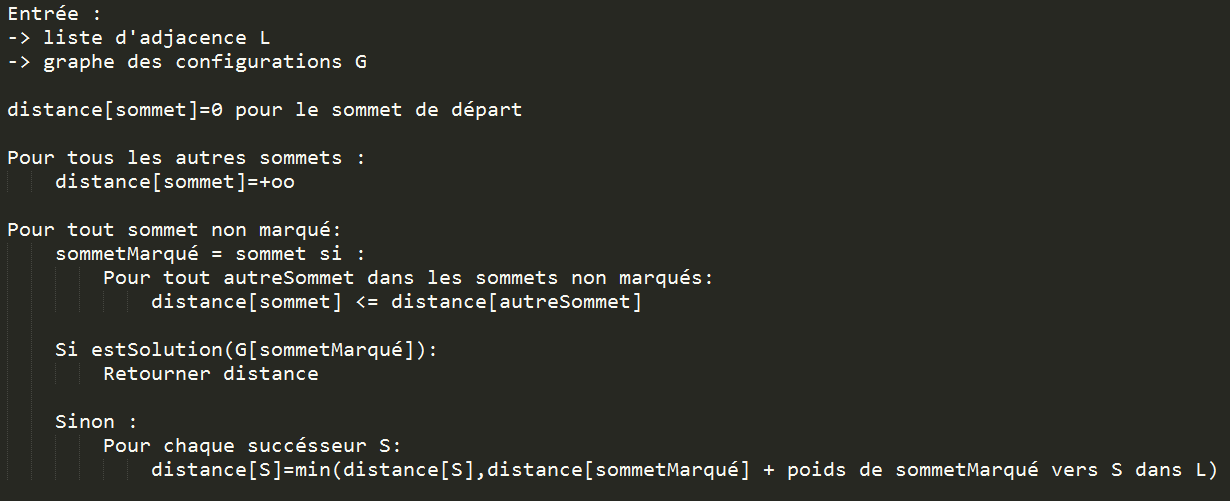


Figure 3 Pseudo code de Dijkstra

Question 9 : Afficher la solution

En stockant les prédécesseurs de chaque sommet lors de l’exécution de l’algorithme de Dijkstra, on est capable de recréer la séquence de déplacements. En effet, un sommet équivaut à une configuration. Il suffit alors de partir de la solution et de remonter les prédécesseurs jusqu’à retrouver la configuration de départ. On a au final la séquence de déplacements de la solution vers la configuration initiale, il suffit alors de l’inverser pour avoir cette séquence dans le bon ordre.

NB : Il aurait été plus difficile d’utiliser un tableau de successeurs vu qu’un sommet du sous-graphe à la fin de l’algorithme peut avoir plusieurs successeurs alors qu’il n’a qu’un seul prédécesseur.

## Résolution de Rush Hour Mouvements

Question 10

S’il existe, une arrête entre deux sommets du graphe des configurations, il existe alors un déplacement (d’un véhicule) permettant de passer de l’un à l’autre. Dans un problème RHM, on cherche à minimiser le nombre de ces déplacements. Lorsque l’on déplace un véhicule de 1,2,3, etc. cases, on n’effectue au final qu’un seul déplacement, le poids de chaque arête est donc fixé à 1. Nous pouvons ensuite appliquer Dijkstra pour obtenir un chemin de poids minimal (et donc effectuant le moins de mouvement) de la configuration de départ vers un sommet représentant une configuration-but.

Question 11

La question n’étant pas explicite, on peut la comprendre différemment. S’il s’agit de compter le nombre total de configurations réalisables : cela correspond à la taille de la structure correspondant au graphe des configurations. Si on veut calculer cette taille par l’algorithme de Dijkstra, il faut incrémenter un compteur à chaque nouveau sommet marqué. La condition d’arrêt est donc cette fois-ci le marquage de tous les sommets. Cependant s’il s’agit de compter le nombre de configurations possibles pour un nombre M de mouvements donné : on peut fixer les poids des arêtes à 1 pour parcourir les niveaux de profondeur un à un. On s’arrête quand le sommet marqué est de profondeur M+1 sans compter ce sommet.

# Expérimentations numériques



Figure 4Résolution grâce à l'algorithme de Dijkstra en microseconde



Figure 5 Résolution d'un RushHour via Gurobi

On peut conclure que notre implémentation via Dijkstra est beaucoup plus efficace que celle de gurobi. Avec Dijkstra, il nous faut au maximum deux secondes dans le pire des cas (jam14) pour résoudre un RushHour, alors que pour Gurobi, même des cas triviaux dépassent la durée limite de 120 secondes. De plus, certains sont considérés comme infaisables à cause d’un N trop petit, comme jam10. Par conséquent, il vaut mieux utiliser notre algorithme de Djikstra. Nous allons donc spécifiquement nous intéresser à notre implémentation la plus efficace.

On peut observer que le temps d’exécution pour RHM ainsi que pour RHC augmente plus on a de sommets. La difficulté pour résoudre le RushHour via Dijkstra est donc proportionnelle au nombre de sommet.

On peut observer que le nombre de véhicules n’influence pas sur la difficulté pour résoudre le problème dans Dijkstra.

# Conclusion

1. https://fr.wikipedia.org/wiki/Rush\_hour\_(casse-t%C3%AAte) [↑](#footnote-ref-1)