Projet - MOGPL
Résolution d'un casse-tête : Rush Hour^{TM}

1. Introduction

On s'intéresse dans ce projet à la résolution d'un casse-tête intitulé *Rush Hour*. Il est possible d'y jouer en ligne, par exemple via le lien :

`http://www.mah-jongg.ch/rushhour/rushhour.swf`

Le but du jeu est de faire sortir une voiture rouge (unique) d'un embouteillage sur une grille 6×6. Sur cette grille sont positionnés des voitures (deux cases de long) et des camions (trois cases de long) horizontalement ou verticalement. Les véhicules ne peuvent pas tourner, autrement dit ils se déplacent uniquement sur leur rangée de départ horizontale ou verticale. Une configuration de départ du jeu est décrite par une carte comme celle représentée sur la figure de gauche ci-dessous. On positionne alors les véhicules (15 au maximum) sur la grille comme indiqué sur la carte. La voiture rouge (pointée par la flèche) est toujours positionnée horizontalement sur la troisième ligne, la sortie étant située à l'extrémité droite de cette ligne.

Une solution pour cette configuration de départ consiste à déplacer la voiture verte d'une case vers la droite, le camion violet d'une case vers le haut, la voiture orange d'une case vers le haut, le camion vert de deux cases vers la gauche, la voiture turquoise de trois cases vers la gauche, le camion bleu de deux cases vers le bas, le camion jaune de trois cases vers le bas, et enfin la voiture rouge de trois cases vers la droite. On considère le jeu résolu dès que la voiture rouge est positionnée sur les deux cases devant la sortie. Toute configuration où la voiture rouge est ainsi positionnée est appelée *configuration-but*. Le but du jeu est de passer d'une configuration de départ à une configuration-but par une séquence de déplacements de véhicules.

Deux fonctions objectifs seront considérées :

(a) minimiser le nombre de mouvements : un mouvement consiste à déplacer un véhicule d'un nombre quelconque de cases ; par exemple, la solution décrite ci-dessus comporte 8 mouvements ;

(b) minimiser le nombre total de cases de l'ensemble des mouvements : c'est la version pondérée de la fonction objectif précédente, puisque chaque mouvement coûte le nombre de cases de déplacement ; par exemple, il y a 16 cases de déplacement au total dans la solution décrite plus haut.

Pour l'objectif (a) on parlera du problème RHM (Rush Hour Mouvements), et pour l'objectif (b) du problème RHC (Rush Hour Cases).

Question 1. Montrer que, pour une même configuration de départ, une solution optimale pour RHC n'est pas nécessairement optimale pour RHM. Pour ce faire, on pourra utiliser une configuration de départ où la voiture rouge est positionnée comme dans l'exemple de l'introduction, et qui comporte en tout trois voitures (en comptant la rouge) et deux camions.

Les problèmes RHM et RHC sont difficiles d'un point de vue computationnel. Le projet consiste à développer une ou deux méthodes de résolution, l'une fondée sur la programmation linéaire en variables binaires, l'autre sur les graphes, et à développer une interface graphique (qui peut être minimaliste) pour décrire les solutions trouvées. Afin de faciliter le débogage, on commencera par développer l'interface graphique.

2. Configurations de départ et affichage

Il y a 40 configurations de départ de difficulté croissante fournies avec la version physique du jeu. Elles sont stockées sous la forme de fichiers textes à télécharger sur le site web de MOGPL :

`http://www-poleia.lip6.fr/~perny/ANDROIDE1617/MOGPL/`

Les fichiers sont classés en quatre niveaux : débutant (`jam1.txt` à `jam10.txt`), intermédiaire (`jam11.txt` à `jam20.txt`), avancé (`jam21.txt` à `jam30.txt`) et expert (`jam31.txt` à `jam40.txt`). Le fichier `jam1.txt` par exemple se présente sous la forme suivante :

```
6 6
c1 c1 0 0 0 t1
t2 0 0 t3 0 t1
t2 g g t3 0 t1
t2 0 0 t3 0 0
c2 0 0 0 c3 c3
c2 0 t4 t4 t4 0
```

La première ligne du fichier indique le nombre de lignes et de colonnes de la grille. Précisons que toutes les grilles considérées dans ce projet seront

de taille 6×6. Néanmoins, la possibilité de changer la taille de la grille peut aider au débogage du code. Chaque ligne suivante dans le fichier correspond à une ligne de la grille. Le caractère `0` signifie une case libre, tandis qu'une case occupée comporte le code du véhicule qui l'occupe. Le caractère `g` indique les cases occupées par la voiture rouge. Les codes des autres voitures débutent par le caractère `c` (pour *car*), et les codes des camions débutent par le caractère `t` (pour *truck*). Par exemple, dans `jam1.txt`, la voiture `c1` occupe les deux premières cases à gauche de la première ligne, et le camion `t1` occupe les trois premières cases en haut de la dernière colonne.

Remarque : pour aider au débogage, trois configurations "tests" particulièrement simples sont également fournies (`test1.txt` à `test3.txt`).

Question 2. Coder une fonction de lecture d'un fichier d'entrée, qui génère une structure de données représentant la configuration de départ. La structure choisie devra permettre d'accéder en $\Theta(1)$ à n'importe quelle case de la grille (ainsi qu'à l'identifiant du véhicule qui l'occupe si la case est occupée).

Question 3. Coder une fonction d'affichage qui prend en entrée la structure de données représentant une configuration quelconque et qui affiche la grille. L'affichage peut se faire dans la console tant qu'il reste lisible.

Les parties 3 et 4 peuvent être traitées indépendamment. Il n'est pas obligatoire d'aborder les deux parties, mais il faut en traiter au moins une. Toutefois pour pouvoir dépasser la note de 15 il est nécessaire d'avoir traité les deux parties afin de comparer les deux méthodes.

3. Résolution par programmation linéaire en variables binaires

Cette section est consacrée à une méthode de résolution de Rush Hour fondée sur la programmation linéaire en variables binaires. Afin d'identifier la position d'un véhicule, on considérera la position de son *marqueur*, qui correspond au numéro de la case la plus en haut à gauche occupée par ce véhicule. La numérotation des cases est indiquée sur la grille ci-dessous. Par exemple, si une voiture (resp. un camion) occupe les cases 14 et 15 (resp. 16, 22 et 28), alors son marqueur est positionné en 14 (resp. 16).

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

SORTIE

Pour pouvoir formuler les problèmes RHC et RHM comme des programmes mathématiques, il est nécessaire de s'imposer un nombre maximum

autorisé N de mouvements de véhicules (pour avoir un nombre fini de variables) pour résoudre le casse-tête. Ce nombre N est fixé par l'utilisateur avant de lancer la résolution. Les valeurs de N à utiliser pour les expérimentations numériques sont indiquées dans la section 5.

Les variables figurant dans le programme mathématique comportent toutes au moins trois indices : l'identifiant i du véhicule concerné (il peut être pratique de coder l'identifiant comme une chaîne de caractères, correspondant au nom du véhicule), l'indice $j \in \{1, \ldots, 36\}$ de la position concernée dans la grille, et enfin l'indice $k \in \{0, \ldots, N\}$ du tour de jeu (les variables indicées par $k = 0$ correspondent à la configuration de départ du jeu). Il y a trois types de variables (toutes binaires) :

$x_{i,j,k} = 1$ si le marqueur du véhicule i est en position j au terme du $k^{\text{ème}}$ mouvement, 0 sinon ;

$z_{i,j,k} = 1$ si le véhicule i occupe la position j (notons qu'une voiture occupe deux positions, et un camion trois positions) au terme du $k^{\text{ème}}$ mouvement, 0 sinon ;

$y_{i,j,l,k} = 1$ si le marqueur du véhicule i est passé de la position j à la position l lors du $k^{\text{ème}}$ mouvement, 0 sinon.

Afin d'exprimer les différentes contraintes du problème, on fera appel aux notations suivantes :

v_i : longueur (en nombre de cases) du véhicule i ;

$m_{i,j}$: ensemble des positions occupées par le véhicule i quand le marqueur i est en position j ;

$p_{j,l}$: ensemble des positions comprises entre les positions j et l.

Les contraintes permettant de caractériser les positions occupées par les différents véhicules sont les suivantes :

$$v_i x_{i,j,k} \leq \sum_{m \in m_{i,j}} z_{i,m,k} \qquad \forall i, j, k \qquad (1)$$

$$\sum_i z_{i,j,k} \leq 1 \qquad \forall j, k \qquad (2)$$

$$\sum_j z_{i,j,k} = v_i \qquad \forall i, k \qquad (3)$$

$$y_{i,j,l,k} \leq 1 - \sum_{i' \neq i} z_{i',p,k-1} \qquad \forall i, j, l, k \quad \forall p \in p_{j,l} \qquad (4)$$

En s'appuyant sur la position j du marqueur du véhicule i, la contrainte (1) assure que $z_{i,m,k} = 1$ pour toute case m occupée par i au terme du $k^{\text{ème}}$ mouvement. La contrainte (2) assure qu'il n'y ait pas plusieurs véhicules occupant une même case j au terme du $k^{\text{ème}}$ mouvement, et la contrainte (3)

renforce la formulation en établissant que seules v_i cases sont occupées par le véhicule i dans sa rangée. Enfin, la contrainte (4) assure qu'un véhicule i ne peut se déplacer d'une case j à une case l lors du $k^{\text{ème}}$ mouvement ($y_{i,j,l,k} = 1$) que si les cases comprises entre j et l sont libres.

Question 4. Donner l'expression de la fonction objectif :

- **a.** si l'on souhaite résoudre RHM (minimiser le nombre de mouvements) ;
- **b.** si l'on souhaite résoudre RHC (minimiser le nombre total de cases déplacement).

Question 5. Donner les contraintes, impliquant les variables $x_{i,j,k}$ et $y_{i,j,l,k}$, qui assurent que :

- **a.** la voiture rouge est positionnée devant la sortie au terme du dernier mouvement ;
- **b.** au plus *un* véhicule est déplacé par tour ;
- **c.** la position du marqueur d'un véhicule i est bien mise à jour si le véhicule i se déplace.

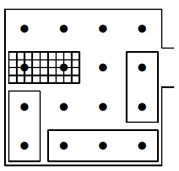
Question 6. Expliquer pourquoi seul un sous-ensemble de toutes les variables binaires potentielles doivent être introduites dans le modèle.

Question 7. Coder la fonction qui fait appel un solveur de programmation linéaire en variables binaires (Gurobi) pour résoudre Rush Hour. La solution devra être affichée sous la forme d'une séquence de déplacements à réaliser afin de résoudre le casse-tête, et non pas sous la forme d'une liste de variables binaires avec leurs valeurs (ce qui serait rapidement fastidieux, et difficilement intelligible pour un non-initié !). L'affichage en mode texte dans la console est accepté.

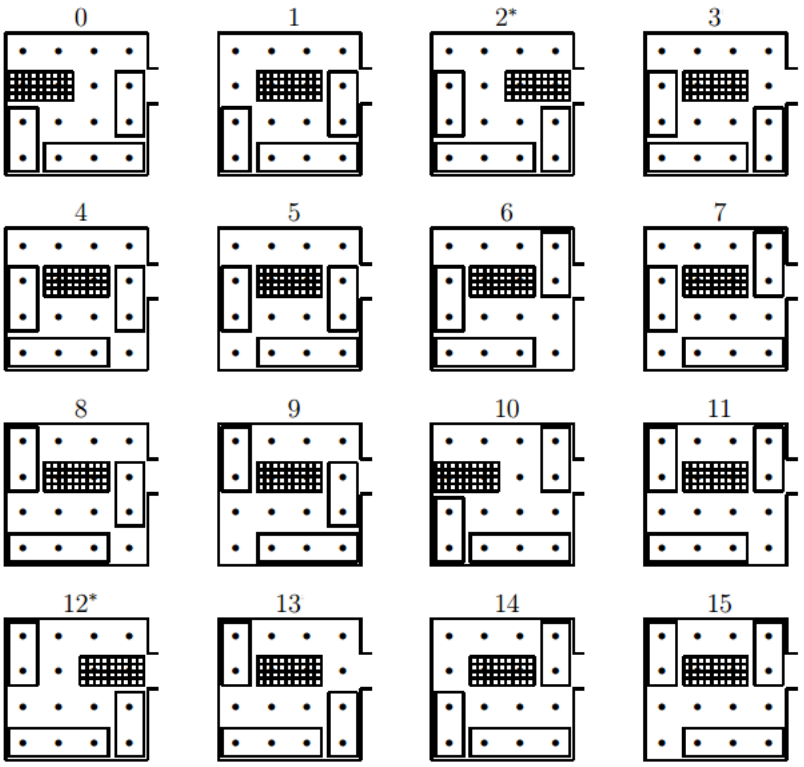
4. Résolution par l'algorithme de Dijkstra

Une autre méthode de résolution de Rush Hour consiste à appliquer l'algorithme de Dijkstra dans le *graphe des configurations* du jeu. Le graphe des configurations est un graphe non-orienté où chaque sommet correspond à une configuration réalisable (c'est-à-dire sans chevauchement de véhicules, et atteignable depuis la configuration de départ) et où il y a une arête entre deux configurations si il est possible de passer de l'une à l'autre en déplaçant *un* véhicule d'une ou plusieurs cases.

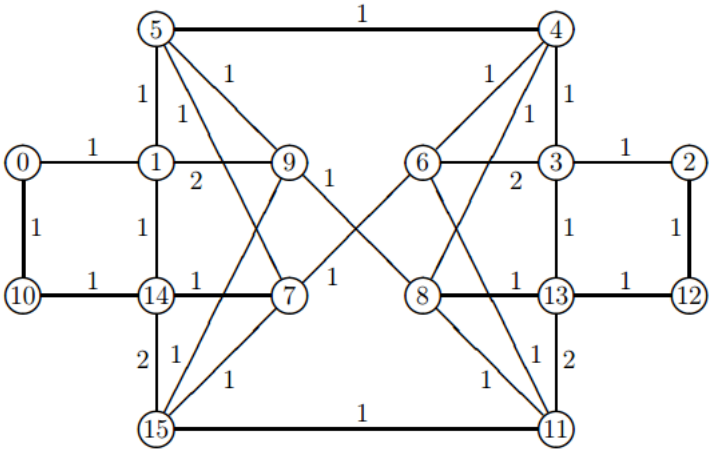
A titre d'illustration, considérons la configuration de départ suivante (sur une grille volontairement réduite afin que la taille du graphe des configurations reste contenue), où le rectangle hachuré représente la voiture rouge :



Pour cette configuration de départ, l'ensemble des 16 configurations réalisables est listé ci-dessous (les deux configuration-buts sont marquées par une astérisque) :



Le graphe des configurations est représenté ci-dessous. La valeur d'une arête correspond au nombre de cases de déplacement pour passer d'une configuration à l'autre.



Si l'objectif est de minimiser le nombre total de cases de déplacement (RHC), la résolution du casse-tête proposé consiste alors à appliquer l'algorithme de Dijkstra pour trouver une plus courte chaîne de la configuration de départ à une configuration-but. Sur l'exemple, il s'agit de trouver une plus courte chaîne du sommet 0 vers un des sommets 2 ou 12.

Notons qu'en pratique le graphe n'est pas donné explicitement (on ne dispose pas de la liste des configurations et des arêtes) mais qu'il est défini en compréhension : étant donnée une configuration, on est capable de déterminer les configurations voisines.

Afin de résoudre RHC et RHM par l'algorithme de Dijkstra, il sera donc nécessaire d'implémenter une fonction déterminant les configurations voisines d'une configuration (en recensant les déplacements possibles de véhicules). Pour chaque nouveau sommet du graphe des configurations ainsi découvert, on sera amené à réaliser une copie de la grille de jeu et à la modifier pour simuler le déplacement d'un véhicule (d'une ou plusieurs cases vers la gauche, la droite, le haut ou le bas).

Question 8. Coder une méthode de résolution de RHC fondée sur l'utilisation de l'algorithme de Dijkstra dans le graphe des configurations. Quelle est la condition d'arrêt de l'algorithme ?

Question 9. Coder une fonction qui permet d'afficher la séquence de déplacements à réaliser afin de résoudre le casse-tête depuis une configuration de départ donnée en entrée (l'affichage en mode texte dans la console est accepté), ainsi que la valeur correspondante de la fonction objectif.

Question 10. Expliquer comment adapter la méthode de la question 8 si l'on souhaite résoudre RHM (minimiser le nombre de mouvements). Implanter cette fonctionnalité dans le code.

Question 11. Expliquer comment adapter la méthode la question 8 si l'on souhaite compter le nombre de configurations réalisables (en un nombre quelconque de mouvements de véhicules) depuis la configuration de départ. Implanter cette fonctionnalité dans le code.

5. Expérimentations numériques

Tester la (les) méthode(s) de résolution que vous avez codée(s) sur les 40 configurations de départ fournies sur le site web du module. Chaque exécution devra tourner au plus deux minutes, au terme desquelles sera retournée la meilleure solution trouvée jusqu'alors si une solution a été trouvée, et aucune solution dans le cas contraire. Pour la méthode fondée sur la programmation linéaire en variables binaires, on posera $N = 14$ pour les configurations de niveau *débutant*, $N = 25$ pour les configurations de niveau *intermédiaire*, $N = 31$ pour les configurations de niveau *avancé*, et $N = 50$ pour les configurations de niveau *expert*.

Question 12. Fournir les temps de résolution obtenus pour chaque méthode implantée et chaque instance ou groupe d'instances, sous la forme d'un tableau ou d'une courbe. Pour la méthode fondée sur l'algorithme de Dijkstra, il pourra être intéressant également d'étudier le comportement en fonction du nombre de configurations réalisables depuis la configuration de départ.

6. Organisation et dates

Le choix du langage de programmation est libre. Pour la section 3, le solveur à employer est Gurobi (`http://www.gurobi.com`), qui est disponible gratuitement pour les étudiants (prévoir un éventuel délai d'un jour ou deux pour obtenir une licence étudiant auprès de Gurobi). Ce solveur est de plus installé dans certaines salles machines.

Les projets doivent s'effectuer en **binôme** (étudiants du même groupe).

Les projets doivent être rendus le **1er décembre 2016** au plus tard **par mail** (sujet : [MOGPL-GRx] Nom1-Nom2, où x est le numéro de groupe et Nom1 et Nom2 sont les noms de famille des étudiants composant le binôme) à l'une des adresses suivantes selon votre groupe :

Groupe 1 (lundi)	hung.nguyen@lip6.fr
Groupe 2 (jeudi)	olivier.spanjaard@lip6.fr
Groupe 3 (jeudi)	safia.kedad-sidhoum@lip6.fr
Groupe 4 (vendredi)	patrice.perny@lip6.fr

Votre livraison sera constituée d'une archive `zip` qui comportera les sources du programme, un fichier README détaillant comment compiler et exécuter le programme, et un rapport (un fichier au format `pdf`) avec les réponses aux différentes questions (pour les questions portant sur le code, une présentation *synthétique* du code développé devra être proposée, via un pseudo-code ou un extrait de code). Le plan du rapport suivra le plan du sujet.

Les séances de TD de la **semaine du 5 décembre** seront consacrées aux **soutenances** du projet, et se dérouleront en salle machine. Il ne sera pas nécessaire de préparer des transparents. Prévoir par contre de pouvoir tester le code facilement sur une ou plusieurs nouvelles instances qui vous seront fournies lors de la soutenance au format de fichier détaillé dans la section 2.

Il est **strictement interdit** de communiquer son rapport ou son code à un autre binôme, ou de récupérer du code sur le web. Les contrevenants s'exposent à des sanctions.