INF421 Programming Project

Solver of Rush Hour

Celine Hajjar, Alya Zeinaty

Dimanche 2 février 2020

Table des matières

1 Contexte

Rush Hour est à l'origine un casse-tête de déplacement, dont le but est de faire sortir un véhicule par une sortie désignée sur un plateau de jeu. Dans ce projet, nous avons simulé le Puzzle de Rush Hour pour ensuite proposer des idées de solutions.

2 Configuration du jeu

Un plateau de RushHour est représenté informatiquement par un fichier texte. Or il va sans dire qu'un tel fichier est difficilement manipulable. Il s'agit donc dans cette partie de mettre en place les structures de données adaptées à un jeu de RushHour.

2.1 Lecture du fichier

Afin de représenter un jeu de RushHour, nous avons créé une classe *Game*, dont les arguments sont explicités dans la bibliothèque.

Notre algorithme de lecture de fichier se structure en 3 parties.

Dans un premier temps, on initialise les arguments *size, numberOfVehicles, cars* à partir du fichier texte.

Dans un second temps, on utilise la fonction *updateDisplayConstructor()* qui met à jour la matrice display d'un jeu *game* à partir de son set de véhicules, en vérifiant qu'il est valide. Le principe est le suivant : *display* est une matrice carrée de taille *size* initialisée à 0. On remplit les cases occupées par chaque véhicule dans la matrice avec son étiquette entière. Si la case que l'on souhaite remplir est déjà occupée par un autre véhicule, on arrête l'algorithme et on attribue à *valid* la valeur *false*. Si le set de véhicules est valide, l'algorithme termine et on attribue à *valid* la valeur *true*.

Enfin, dans un dernier temps, on définit la sortie en face du véhicule 1, grâce à *defineExit()*.

La complexité d'un tel algorithme est O(size^2).

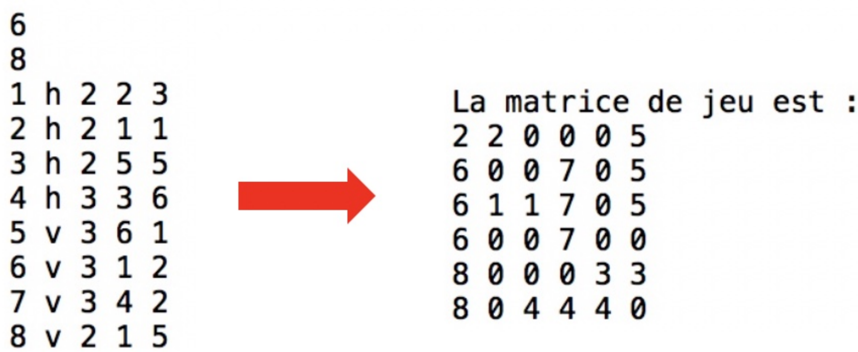


Figure 1 – Transformation du fichier en matrice

2.2 Affichage de l'état courant

Afin d'afficher le jeu, on utilise la fonction *printdisplay()*.

Cette fonction renvoie "The game is not valid" si l'argument *valid* est égal à *false*, et renvoie la matrice *display* du jeu sinon. (fig 1)

La complexité d'une telle fonction est O(size^2).

3 Résolution du problème

3.1 Algorithme *Brute-Force*

La première étape de la résolution du problème consiste en l'implémentation d'un algorithme de force brute. L'idée est de parcourir, à partir d'une configuration initiale donnée, toutes les séquences de mouvements admissibles. L'algorithme termine dès que l'état courant est un état final, c'est-à-dire dès que le véhicule 1 se trouve juste devant la sortie.

Le choix de la structure de données Pour stocker les états déjà explorés, nous avons choisi une structure de HashMap. En effet, une HashMap présente une complexité très faible (O(1)) pour les opérations de base, comme l'ajout d'un élément ou la recherche d'une clé. Étant donné qu'à chaque itération de notre programme, nous effectuons une recherche dans la liste des états explorés, dont le nombre est très important, une HashMap semble plus adaptée à notre problème qu'un tableau ou qu'une LinkedList.

Pseudocode L'ensemble des états *Game* du jeu est représenté sous forme de graphe. Les sommets de ce graphe sont les états *Game* du jeu. Deux états ou sommets sont liés si on peut passer de l'un à l'autre par un seul mouvement. La liste des états accessibles en un mouvement à partir d'un état donné est déterminée par la fonction *getNext()* de la classe *Game*. Ainsi, pour parcourir tous les états d'un jeu, il suffit d'implémenter l'algorithme de parcours d'un graphe en largeur :

Algorithm 1: Algorithme Brute-Force

```
Input : L'état initial d'un jeu
Output : Un état final et le nombre d'étapes requises pour y arriver
queue := Liste Chaînée vide ;
visited := HashMap (État, booléen) vide ;
nb :=0 ;
ajouter l'état initial à queue ;
while queue non vide do
    g :=queue.poll() //enlever le premier élément ;
    incrémenter nb de 1;
    if g état final then
        imprimer nb;
        imprimer matrice display de g;
        break;
    end
    next := g.getNext() //renvoie une liste chaînée des voisins de g ;
    for e dans next do
        if e pas dans visited then
            ajouter e à la fin de queue;
            ajouter (e, true) dans visited;
        end
    end
end
```

Implémentation du code La principale différence entre le pseudocode et le code implémenté est que les clés de la HashMap ne sont pas les états *Game* eux-mêmes, mais une représentation en chaîne de caractères de leur matrice *display*, afin de pouvoir comparer deux *Game* plus facilement.

Complexité La complexité dans le pire cas d'un tel algorithme est O(nombre de sommets+nombre d'arêtes)=O($numberOfVehicles^{size^2}$).

3.2 Reconstruction de la solution

Afin de reconstruire la solution, on a modifié les structures de données utilisées. Notamment, on a créé une nouvelle classe *Gamove*, qui représente un couple (Game, Move). La HashMap *visited* devient donc une HashMap <Game

(en chaîne de caractères), Gamove>.

Ainsi, dans l'algorithme de force brute, au lieu d'ajouter l'élément $(e, true)$ à *visited*, on y ajoute $(e, Gamove(g, m))$, où g représente l'état parent de e et m le Move qui permet de passer de g à e.

De même, la méthode *getNext()* rend alors une liste chaînée de *Gamove* (e, m) , où e est un état fils de l'état courant g et m le Move qui permet d'accéder de g à e.

Finalement, pour reconstruire la solution, on remonte la HashMap *visited* de l'état final jusqu'à l'état initial, et on retient à chaque fois le *Move* concerné.

Complexité La complexité de cet algorithme est la même que celle de l'algorithme Brute Force.

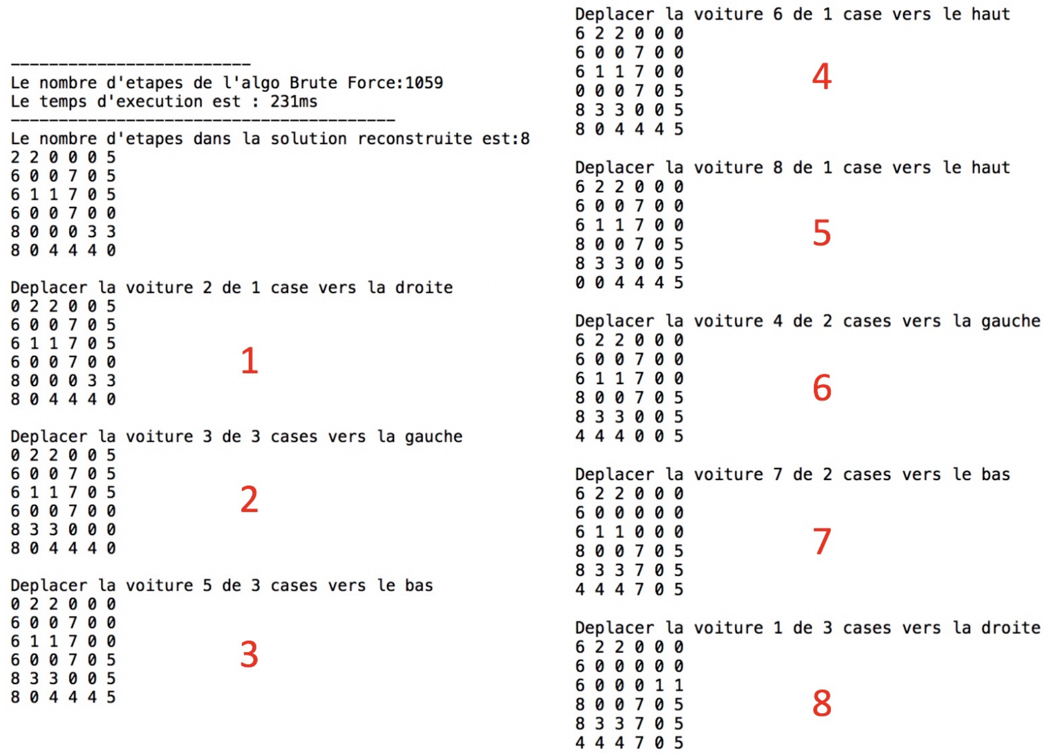
3.3 Résultats

L'exécution de l'algorithme de recherche et de la reconstruction de la solution minimale a terminé en 231 ms (fig 3)

Pour exécuter le code, il faut aller dans la classe *Solve* et préciser l'emplacement du fichier qui représente l'état initial. Ceci se fait dans le constructeur du nouveau jeu dans la fonction *main*. (fig 2)

```java
public static void main(String[] args) {
    Game game=new Game ("/Users/celinehajjar/Desktop/2A/INF421/Projet_RushHour/RushHour/ExRushHour/GameP01.txt");
    long startTime = System.currentTimeMillis();
    solve(game);
    long endTime=System.currentTimeMillis();
    long time=endTime-startTime;
    System.out.println("Temps d'execution: "+time +" ms");
}
```

FIGURE 2 – Méthode *main* de la classe *Solve*



```
-------------------------
Le nombre d'etapes de l'algo Brute Force:1059
Le temps d'execution est : 231ms
----------------------------------------
Le nombre d'etapes dans la solution reconstruite est:8
2 2 0 0 0 5
6 0 0 7 0 5
6 1 1 7 0 5
6 0 0 7 0 0
8 0 0 0 3 3
8 0 4 4 4 0

Deplacer la voiture 2 de 1 case vers la droite
0 2 2 0 0 5
6 0 0 7 0 5
6 1 1 7 0 5
6 0 0 7 0 0          1
8 0 0 0 3 3
8 0 4 4 4 0

Deplacer la voiture 3 de 3 cases vers la gauche
0 2 2 0 0 5
6 0 0 7 0 5
6 1 1 7 0 5          2
6 0 0 7 0 0
8 3 3 0 0 0
8 0 4 4 4 0

Deplacer la voiture 5 de 3 cases vers le bas
0 2 2 0 0 0
6 0 0 7 0 0
6 1 1 7 0 0
6 0 0 7 0 5          3
8 3 3 0 0 5
8 0 4 4 4 5

Deplacer la voiture 6 de 1 case vers le haut
6 2 2 0 0 0
6 0 0 7 0 0
6 1 1 7 0 0          4
0 0 0 7 0 5
8 3 3 0 0 5
8 0 4 4 4 5

Deplacer la voiture 8 de 1 case vers le haut
6 2 2 0 0 0
6 0 0 7 0 0
6 1 1 7 0 0          5
8 0 0 7 0 5
8 3 3 0 0 5
0 0 4 4 4 5

Deplacer la voiture 4 de 2 cases vers la gauche
6 2 2 0 0 0
6 0 0 7 0 0
6 1 1 7 0 0
8 0 0 7 0 5          6
8 3 3 0 0 5
4 4 4 0 0 5

Deplacer la voiture 7 de 2 cases vers le bas
6 2 2 0 0 0
6 0 0 0 0 0
6 1 1 0 0 0          7
8 0 0 7 0 5
8 3 3 7 0 5
4 4 4 7 0 5

Deplacer la voiture 1 de 3 cases vers la droite
6 2 2 0 0 0
6 0 0 0 0 0
6 0 0 0 1 1
8 0 0 7 0 5          8
8 3 3 7 0 5
4 4 4 7 0 5
```

FIGURE 3 – Résultats de l'implémentation de l'agorithme Brute Force

4 Heuristiques

Afin de réduire le temps d'exécution de notre algorithme de force brute, nous nous sommes intéressées, comme le suggérait l'énoncé, aux heuristiques. Une heuristique est une fonction qui à chaque état *game* du jeu associe une valeur entière $h(game)$, qui représente une borne inférieure d'une solution qui part de *game*. Ainsi, plus la valeur $h(game)$ est basse, plus l'état *game* est supposé proche de l'état final. Pour un état final *final*, on a nécessairement $h(final) = 0$.

4.1 Pseudocode

Le pseudocode 3.1. doit donc être modifié pour prendre en compte les heuristiques de chaque état du jeu. Il s'agit de traiter en priorité les états de *queue* qui ont l'heuristique la plus basse. Par conséquent, au lieu de ranger les états à visiter dans une liste chaînée comme précédemment, nous allons les ranger dans une queue de priorité, dont le comparateur sera une heuristique. Ainsi, le pseudocode devient :

Algorithm 2: Algorithme Heuristique

```
Input :  L'état initial d'un jeu
Output :  Un état final et le nombre d'étapes requises pour y arriver
queue := PriorityQueue vide ;
visited := HashMap (État, booléen) vide ;
nb :=0 ;
ajouter l'état initial à queue ;
while queue non vide do
    g :=queue.poll() //enlever l'élément d'heuristique minimale ;
    incrémenter nb de 1;
    if g état final then
        imprimer nb;
        imprimer matrice display de g;
        break;
    end
    next := g.getNext() //renvoie une liste chaînée des voisins de g ;
    for e dans next do
        if e pas dans visited then
            ajouter e dans queue;
            ajouter (e, true) dans visited;
        end
    end
end
```

Correction Il paraît évident que si l'algorithme termine, il renvoie bien un état final. En effet, on parcourt tous les états du graphe jusqu'à ce que l'on trouve l'état final, ou que tous les états aient été visités.

Il reste maintenant à prouver que l'algorithme termine.

Terminaison Afin de prouver la terminaison de notre algorithme, il suffit de prouver que la boucle *while* se termine. En d'autres termes, il suffit de prouver que soit l'on trouve un état final, soit *queue* finit nécessairement par être vide.

Supposons que l'on ne trouve pas d'état final.

Etant donné que chaque itération de la boucle supprime un élément de *queue*, il suffit de montrer que le nombre d'éléments *ajoutQueue* ajoutés à *queue* est fini.

Tous les sommets qui sont ajoutés à *queue* sont ajoutés à *visited*. Ainsi, $cardinal$(visited)$\geqslant$ $cardinal$(ajoutQueue).

Or un sommet n'est jamais supprimé de *visited*. Ainsi, $cardinal$(visited) est croissant, majoré par le nombre total de sommets $cardinal$(G).

D'où $cardinal$(ajoutQueue) $\leqslant$ $cardinal$(G).

Ainsi, il y aura au plus $cardinal$(G) itérations de la boucle, puisque chaque itération supprime un sommet de *queue*.

L'algorithme termine.

Remarque On considère dans les paragraphes suivants plusieurs heuristiques différentes. Afin de changer l'heuristique utilisée par notre algorithme, il faut décommenter la section d'intérêt dans la classe *GameComparator*, et mettre l'heuristique précédente en commentaire.

4.2 Heuristique zéro

La première heuristique testée est celle qui associe à chaque état l'entier 0. L'algorithme Heuristique dans cas là revient à un algorithme de parcours en profondeur.

Les résultats de l'algorithme avec l'heuristique 0 sont exposés en figure 4.

```
-------------------------
Le nombre d'etapes dans l'algorithme pour l'heuristique 0 est :492
-----------------------
Temps d'execution: 181 ms
```

FIGURE 4 – Résultats de l'algorithme Brute Force avec l'heuristique 0

4.3 Heuristique de blocage

Consistance Soient deux sommets s et s' du graphe des sommets. Soit h l'heuristique qui renvoie le nombre de véhicules entre le véhicule 1 et la sortie. Arbitrairement, soit $h(s) \leqslant h(s')$. Chaque véhicule qui bloquait la voie en s' et qui ne la bloque plus en s a dû être déplacé, ce qui coûte au moins autant de mouvements que de véhicules qui ne bloquent plus la voie. Si $k_{s,s'}$ représente le nombre de mouvements nécessaires pour passer de s' à s, alors on a bien $k_{s,s'} \geqslant h(s') - h(s)$. D'où $h(s') \leqslant h(s) + k_{s,s'}$. L'heuristique de blocage est consistante.

Résultats Les résultats de l'algorithme avec cette heuristique sont exposés en figure 5. Le temps d'exécution est divisé par plus de 4 !

```
-------------------------
Le nombre d'etapes dans l'algorithme pour la nouvelle heuristique est :73
-----------------------
Temps d'execution: 64 ms
```

FIGURE 5 – Résultats de l'algorithme Brute Force avec l'heuristique de blocage

4.4 Nouvelle heuristique

Nous avons ensuite implémenté une nouvelle heuristique *CustomHeuristic*, qui est en réalité une amélioration de l'heuristique de blocage. Ainsi, après avoir compté les véhicules qui bloquent la voie du véhicule 1, on compte en plus le nombre de véhicules qui empêchent complètement les premiers véhicules bloquant de bouger.

Consistance La preuve de la consistance de cette heuristique ressemble fortement à celle de l'heuristique de blocage. Soient deux sommets s et s' du graphe des sommets. Soit h notre nouvelle heuristique. Arbitrairement, soit $h(s) \leqslant h(s')$. Si $k_{s,s'}$ représente le nombre de mouvements nécessaires pour passer de s' à s, alors on a bien $k_{s,s'} \geqslant h(s') - h(s)$. En effet, il a fallu au moins $h(s') - h(s)$ mouvements pour passer de s' à s, puisque chaque véhicule qui bloque le véhicule 1 ou les véhicules bloquant le véhicule 1 nécessite au moins un mouvement pour cesser de bloquer. D'où $h(s') \leqslant h(s) + k_{s,s'}$. La nouvelle heuristique est consistante.

Résultats Les résultats de l'algorithme avec cette heuristique sont exposés en figure 6. Bien que le temps de calcul de la nouvelle fonction heuristique soit plus important que pour l'heuristique de blocage, notre nouvelle heuristique reste pourtant efficace. En effet, le temps d'exécution avec cette heuristique est comparable à celui avec l'heuristique de blocage. Selon les cas, l'une ou l'autre est plus efficace.

```
-------------------------
Le nombre d'etapes dans l'algorithme pour la nouvelle heuristique est :63
-----------------------
Temps d'execution: 44 ms
```

FIGURE 6 – Résultats de l'algorithme Brute Force avec la nouvelle heuristique

A Annexe : Liste des classes et des méthodes

A.1 Game

Fields
boolean valid
int size
int numberOfVehicles
Vehicle [] cars
int [] [] display
Couple exit

Methods
Game(int, int, Couple, Vehicle[], int[][]) *//Constructor*
Game(String) *//Constructor*
isGoal() *//checks if the state is a final state*
equals(Game) *//checks if two states are equal*
defineExit()
updateDisplayConstructor()
printdisplay()
allMoves()
printAllMoves(LinkedList<Move>)
updateDisplay(Vehicle[], int)
updateState(Game, Move)
getNext(Game)
clonecars(Vehicle[])
clonedisplay(int[][])
main(String[])

A.2 Vehicle

Fields
int name
String orientation
Couple coordinates
int sizeOfVehicle

Methods
Vehicle(String[]) *//Constructor*
Vehicle(int, String, Couple, int) *//Constructor*

A.3 Couple

Fields

int abs
int ord

Methods

Couple(int, int) *//Constructor*
printCouple(Couple)

A.4 Move

Fields

int nameOfVehicle
Couple newCoordinates
int moveLength
boolean forward
String orientation

Methods

Move(int, Couple, int, boolean, String) *//Constructor*
toString()

A.5 Gamove

Fields

Game game
Move move

Methods

Gamove(Game, Move) *//Constructor*

A.6 Solve

Methods

solve(Game)
printSolution(HashMap¡String, Gamove¿, Game, Game)
main(String[])

A.7 Heuristic

Abstract method
getValue(Game)

A.8 SolveH

Methods
solve(Game)
printSolution(HashMap¡String, Gamove¿, Game, Game)
main(String[])