
RushHour est un jeu dont le but est de libérer une voiture d'un parking. Les voitures évoluent dans une grille 6x6, soit verticalement soit horizontalement. Chaque voiture occupe 2 ou 3 cases. Les voitures ne peuvent quitter le parking, à l'exception de la voiture rouge. Le but du jeu est de faire sortir la voiture rouge. L'image qui suit donne un exemple d'une configuration initiale.

L'objectif de ce TP est d'écrire un programme qui trouve la solution de ce problème en un minimum de mouvement.

Si vous voulez vous familiariser avec ce jeu ou essayer une solution, vous pouvez consulter le lien suivant : `http://www.thinkfun.com/mathcounts/play-rush-hour`.

Pour le TD il faudra compléter les deux classes fournies en JAVA (vous pouvez les réécrire dans le langage de votre choix). Les classes sont disponibles sur moodle.

1. Représentation du problème(15pts)

Adoptez la représentation suivante du problème. Les six lignes sont numérotées de haut en bas, de 0 à 5 et les 6 colonnes de gauche à droite, de 0 à 5. La classe `RushHour` contient les quatre champs suivants:

`nbcars` : le nombre de voitures ;

`color` : un vecteur donnant la couleur de chaque voiture ;

`horiz` : un vecteur binaire indiquant pour chaque voiture si elle bouge horizontalement ;

`len` : un vecteur indiquant la longueur de chaque voiture (2 ou 3) ;

`moveon` : un vecteur indiquant la ligne (ou la colonne) où se trouve la voiture (ligne ou colonne où elle peut bouger).

Par convention la voiture rouge a pour index 0 et est de longueur 2. Et la sortie du parking est comme sur la figure

L'état du parking au début et à tout moment du jeu est représenté par un objet de la classe `State` qui contient les champs suivants :

`pos` : un vecteur indiquant la position de chaque véhicule (la colonne de gauche pour les voitures qui bougent horizontalement et la première ligne pour les autres);

`c,d` et `prev` : indiquant que l'état actuel a été obtenu à partir de l'état `prev` en déplaçant la voiture `c` de `d` cases (`d`=+1 ou `d`=-1: -1 indique un déplacement vers la gauche ou vers le haut; +1 l'inverse)

Complétez la méthode `success` de la classe `State` qui indique si un état est final (si la voiture rouge peut sortir en 1 coup)

Complétez le constructeur `State(State* s, int c, int d)` de la classe `State` construit à partir de l'état `s` en déplaçant la voiture `c` de `d` cases (d=+1 ou -1). Nous supposons que le déplacement est possible. **Attention** : Il est nécessaire de construire un nouveau vecteur `pos` dans le nouvel état, parce que nous voulons garder les états précédents.

Vous pouvez tester votre code avec la fonction suivante :

```
static void test1() {
int[] positioning = { 1, 0, 1, 4, 2, 4, 0, 1 };
State s0 = new State(positioning);
System.out.println(!s0.success());
boolean b = !s0.success();
State s = new State(s0, 1, 1);

System.out.println(s.prev == s0);
b = b && s.prev == s0;
System.out.println(s0.pos[1] + " " + s.pos[1]);

s = new State(s, 6, 1);
s = new State(s, 1, -1);
s = new State(s, 6, -1);

System.out.println(s.equals(s0));
b = b & s.equals(s0);
s = new State(s0, 1, 1);
```

```
s = new State(s, 2, -1);
s = new State(s, 3, -1);
s = new State(s, 4, -1);
s = new State(s, 4, -1);
s = new State(s, 5, -1);
s = new State(s, 5, -1);
s = new State(s, 5, -1);
s = new State(s, 6, 1);
s = new State(s, 6, 1);
s = new State(s, 6, 1);
s = new State(s, 7, 1);
s = new State(s, 7, 1);
s = new State(s, 0, 1);
s = new State(s, 0, 1);
s = new State(s, 0, 1);
b = b & s.success();
System.out.println(s.success());
if (!b)
System.err.println("mauvais résultat");
}
```

dont le résultat devrait être :

```
true
true
0 1
true
true
```

2. Mouvement possibles(15pts)

A partir de maintenant, nous allons utiliser la classe `RushHour`.

Dans cette question nous allons compléter la méthode : `list<State*> moves(State* s)` qui détermine l'ensemble des états accessibles à partir de l'état `s` en un unique mouvement (d'une voiture).

Nous voulons utiliser une matrice 6x6 `boolean free[6][6]` indiquant les cases du parking qui sont libres. Complétez la méthode `initFree(State* s)`qui initialise la matrice `free` en fonction de l'état donné `s` avec la convention que `true` signifie que la cases est libre. On Adoptera la convention que `free[i][j]` représente la case (i, j) du parking.

Attention : toutes les cases de la matrice `free` doivent être initialisées avec `initFree` parce que nous allons réutiliser la matrice.

Utilisez la fonction suivante pour tester votre code (la fonction implémentée dans la classe `RushHour`):

```
static void test2() {
boolean[][] res = { { false, false, true, true, true, false }, { false, true, true, fals
{ false, false, false, false, true, false }, { false, true, true, false, true, true },
{ false, true, true, true, false, false }, { false, true, false, false, false, true } };
RushHour rh = new RushHour();
rh.nbcars = 8;
rh.horiz = new boolean[] { true, true, false, false, true, true, false, false };
rh.len = new int[] { 2, 2, 3, 2, 3, 2, 3, 3 };
rh.moveon = new int[] { 2, 0, 0, 0, 5, 4, 5, 3 };
State s = new State(new int[] { 1, 0, 1, 4, 2, 4, 0, 1 });
rh.initFree(s);
boolean b = true;
for (int i = 0; i < 6; i++) {
for (int j = 0; j < 6; j++) {
System.out.print(rh.free[i][j] + "\t");
b = b && (rh.free[i][j] == res[i][j]);
}
System.out.println();
}
if (b)
System.out.println("résultat correct");
else
System.err.println("mauvais résultat");
}
```

vous devriez obtenir le résultat suivant :

```
false false true true true false
false true true false true false
false false false false true false
false true true false true true
false true true true false false
false true false false false true
```

Complétez la méthode `moves` qui renvoie l'ensemble d'états qui peuvent être atteints à partir de l'état `s`. Cet ensemble est représenté par une `list<State*>`; l'ordre dans la liste n'est pas important.

Utilisez la fonction suivante pour tester votre code

```
static void test3() {
RushHour rh = new RushHour();
```

```
rh.nbcars = 12;
rh.horiz = new boolean[] { true, false, true, false, false, true, false, true, false, tr
rh.len = new int[] { 2, 2, 3, 2, 3, 2, 2, 2, 2, 2, 2, 3 };
rh.moveon = new int[] { 2, 2, 0, 0, 3, 1, 1, 3, 0, 4, 5, 5 };
State s = new State(new int[] { 1, 0, 3, 1, 1, 4, 3, 4, 4, 2, 4, 1 });
State s2 = new State(new int[] { 1, 0, 3, 1, 1, 4, 3, 4, 4, 2, 4, 2 });
System.out.println(rh.moves(s).size());
System.out.println(rh.moves(s2).size());
}
```

vous devriez obtenir le résultat suivant : (il y a 5 mouvements possibles à partir de l'état `s` qui représente l'état de l'image au début du TP, et 6 à partir de l'état où la voiture bleu à bougé d'une case vers la droite):

```
5
6
```

3. A la recherche d'une solution(15pts)

Nous allons maintenant chercher une solution. Il y a deux idées principales:

- Il faut est nécessaire de mémoriser les états que l'on a déjà trouvé et pour cela on va utiliser un ensemble hash (Il s'agit d'utiliser une variable locale `visted` de type `hash_set<State*>`: consultez le lien suivant pour savoir comment l'utiliser `http://www.sgi.com/tech/stl/hash_set.html`)
- Nous sommes intéresses par la solution la plus rapide, il nous faut énumérer tous les états pour les ordonner en ordre croissant en terme de nombre de mouvements. On va donc utiliser une liste (FIFO).

On peut représenter un ensemble d'état par un arbre dont la racine est l'état initial et les nœuds correspondent aux états `s`. La relation est `s` *est fils de* `p`, où `s` et `p` sont deux états, est définie par : `p` peut être obtenu à partir de `s` en un mouvement.

On cherche à faire une recherche en largeur sur cet arbre, on peut donc représenter les états par une liste (FIFO). La liste contient au début l'état initial, tant que la liste n'est pas vide on extrait le premier état, s'il est final on termine l'algorithme, sinon on ajoute ses fils qui n'ont pas encore été visités à la fin de la liste. On ajoute ensuite les fils à la liste `visited`.

Complétez la méthode `State* solve(State* s)` en implémentant cet algorithme. La méthode doit renvoyer un état final (la voiture rouge doit être juste devant la sortie). Utilisez la fonction suivante pour tester votre code:

```
static void test4() {
RushHour rh = new RushHour();
rh.nbcars = 12;
rh.color = new String[] { "rouge", "vert clair", "jaune", "orange", "violet clair", "ble
"violet", "vert", "noir", "beige", "bleu" };
rh.horiz = new boolean[] { true, false, true, false, false, true, false, true, false, tr
rh.len = new int[] { 2, 2, 3, 2, 3, 2, 2, 2, 2, 2, 2, 3 };
rh.moveon = new int[] { 2, 2, 0, 0, 3, 1, 1, 3, 0, 4, 5, 5 };
State s = new State(new int[] { 1, 0, 3, 1, 1, 4, 3, 4, 4, 2, 4, 1 });
int n = 0;
for (s = rh.solve(s); s.prev != null; s = s.prev)
n++;
System.out.println(n);
}
```

vous devriez obtenir le résultat suivant : (il y a 46 mouvement à faire pour résoudre le problème de la figure de départ)

```
46
```

4. Récupérons une solution(15pts)

Complétez la méthode `void printSolution(State* s)` qui affiche une solution à partir d'un état final `s`. La solution peut être obtenue en utilisant le champ `prev`. Il faut afficher cette solution dans le bon ordre.

Cette méthode doit aussi afficher le nombre total de mouvements. On peut utiliser pour cela le champ `nbMoves`.

Utilisez la fonction suivante pour tester votre code: Ce code représente la figure en début de TP

```
static void solve22() {
RushHour rh = new RushHour();
rh.nbcars = 12;
rh.color = new String[] { "rouge", "vert clair", "jaune", "orange", "violet clair", "ble
"violet", "vert", "noir", "beige", "bleu" };
rh.horiz = new boolean[] { true, false, true, false, false, true, false, true, false, tr
rh.len = new int[] { 2, 2, 3, 2, 3, 2, 2, 2, 2, 2, 2, 3 };
rh.moveon = new int[] { 2, 2, 0, 0, 3, 1, 1, 3, 0, 4, 5, 5 };
State s = new State(new int[] { 1, 0, 3, 1, 1, 4, 3, 4, 4, 2, 4, 1 });
s = rh.solve(s);
rh.printSolution(s);
}
```

vous devriez obtenir le résultat suivant :

```
46 mouvements
voiture bleu vers la droite
voiture noir vers la droite
voiture vert vers le haut
voiture rose vers le bas
voiture orange vers le haut
voiture rouge vers la gauche
voiture vert clair vers le bas
voiture jaune vers la gauche
voiture jaune vers la gauche
voiture vert clair vers le bas
voiture vert clair vers le bas
voiture rouge vers la droite
voiture orange vers le bas
voiture jaune vers la gauche
voiture violet clair vers le haut
voiture violet vers la gauche
voiture beige vers le haut
voiture bleu vers la droite
voiture beige vers le haut
voiture noir vers la droite
voiture vert clair vers le bas
voiture violet vers la gauche
voiture violet vers la gauche
voiture violet clair vers le bas
voiture violet clair vers le bas
voiture bleu ciel vers la gauche
voiture beige vers le haut
voiture beige vers le haut
voiture bleu ciel vers la gauche
voiture bleu ciel vers la gauche
voiture violet clair vers le haut
voiture violet clair vers le haut
voiture violet vers la droite
voiture violet vers la droite
voiture violet vers la droite
voiture rose vers le haut
voiture violet clair vers le bas
voiture violet clair vers le bas
voiture verte clair vers le haut
voiture bleu vers la gauche
voiture bleu vers la gauche
```

```
voiture violet clair vers le bas
voiture rouge vers la droite
voiture rouge vers la droite
voiture rouge vers la droite
```

Note: Votre solution peut être différente de celle présentée ici, cela dépend de la méthode de construction de la liste dans la méthode `moves`. Cependant votre solution doit avoir le même nombre de mouvements (46).

Voici deux autres problèmes à résoudre : le premier doit être résolu en 16 mouvement; le second en 81.

```
static void solve1() {
RushHour rh = new RushHour();
rh.nbcars = 8;
rh.color = new String[] { "rouge", "vert clair", "violet", "orange", "vert", "bleu ciel"
rh.horiz = new boolean[] { true, true, false, false, true, true, false, false };
rh.len = new int[] { 2, 2, 3, 2, 3, 2, 3, 3 };
rh.moveon = new int[] { 2, 0, 0, 0, 5, 4, 5, 3 };
State s = new State(new int[] { 1, 0, 1, 4, 2, 4, 0, 1 });
s = rh.solveAstar(s);
rh.printSolution(s);
}

static void solve40() {
RushHour rh = new RushHour();
rh.nbcars = 13;
rh.color = new String[] { "rouge", "jaune", "vert clair", "orange", "bleu clair", "rose"
"bleu", "violet", "vert", "noir", "beige", "jaune clair" };
rh.horiz = new boolean[] { true, false, true, false, false, false, false, true, false, f
true };
rh.len = new int[] { 2, 3, 2, 2, 2, 2, 3, 3, 2, 2, 2, 2, 2 };
rh.moveon = new int[] { 2, 0, 0, 4, 1, 2, 5, 3, 3, 2, 4, 5, 5 };
State s = new State(new int[] { 3, 0, 1, 0, 1, 1, 1, 0, 3, 4, 4, 0, 3 });
s = rh.solveAstar(s);
rh.printSolution(s);
}
```

5. Autre Approche(30

La méthode proposée précédemment utilise une recherche en largeur pour trouver la solution. Une solution plus rapide peut être mise en place avec un algorithme A*.

A* algorithme

L'idée principale de l'algorithme A* est d'insérer de *l'intelligence artificielle* à la méthode de résolution. Ceci peut être fait en utilisant une fonction d'estimation qui permet d'évaluer la distance entre un état et la solution.

Pour cela au lieu d'utiliser une approche *first-in first-out (premier entré, premier sorti)* lors de la visite des états, l'algorithme A* défini une valeur de priorité pour chaque état, à partir de la fonction d'estimation, et commence par visiter ceux qui ont une plus forte priorité. Pour cela on utilise une structure de données bien plus appropriée : la file de priorité (`http://www.cplusplus.com/reference/queue/priority_queue/`).

En plus de la fonction d'estimation, A* doit aussi tenir compte d'une autre information importante : le coût actuel de l'état. Ce coût est tout simplement dans notre cas la distance entre l'état actuel et l'état initial. Ces deux éléments mis ensembles permettent de calculer la fonction d'estimation heuristique $\boldsymbol{f}$ d'un état, comme la somme du coût actuel de l'état $\boldsymbol{c}$ et de l'estimation du coût restant $\boldsymbol{h}$.

$$f(s) = c(s) + h(s)$$

La priorité est inversement proportionnelle à la fonction d'estimation heuristique.

Mise en place de l'algorithme

Il nous faut donc calculer le nombre de mouvements nécessaires pour atteindre l'état actuel. Pour le calculer, créez une variable `nbMoves` dans la classe `State` qui enregistre le nombre de mouvements effectués pour arriver à cet état. Cette variable commence à 0 et doit être incrémentée à chaque fois que l'on crée un nouvel état en utilisant le constructeur `State(State* s, int c, int d)`.

1. La variable `f` dans l'état State représente le coût associé à un état. Elle est utilisée par la file de priorité pour déterminer quel est le prochain élément à sortir. Complétez tout d'abord ce coût par le nombre de mouvements pour atteindre l'état.

2. Maintenant complétez la méthode `SolveAstar(State s)` pour résoudre le problème de la même façon qu'au début du TD, mai en utilisant la file de priorité à la place d'une liste ordinaire.

3. Modifiez le code des fonctions `solve1,solve22,solve40` pour résoudre les problèmes à l'aide de l'algorithme A*. Vérifiez que les solutions sont obtenues en autant de mouvements qu'avec le premier algorithme.

Modifiez les fonctions `SolveAstar(State s)` et `Solve(State s)` de façon à compter et afficher le nombre d'états visités lors de la résolution.

La fonction d'estimation qui estime le nombre de coups restants joue un rôle important dans l'efficacité de A*. Il est donc important de créer une bonne fonction pour résoudre le problème. Plus cette fonction est efficace, moins d'états inutiles seront visités. Cette fonction doit par contre toujours renvoyer un minorant du nombre de coups restant. On va analyser deux fonctions différentes dans ce TP.

Fonction 1: à partir de la distance entre la voiture Rouge et la sortie.

1. Complétez dans la classe `State`, la fonction `int estimee1()` qui renvoie la distance entre la voiture rouge et la sortie.
2. Modifiez le champ `f` fourni de la classe `State` pour prendre en compte cette heuristique.
3. Utiliser les exemples fournis (`solve22(), solve1() et solve40` pour tester cet algorithme et le comparer au précédent.

Fonction 2: à partir de la distance de la voiture rouge à la sortie et de nombre de voitures qui lui barre le passage.

1. Complétez dans la classe `State`, la fonction `int estimee2()` qui renvoie la distance entre la voiture rouge et la sortie plus le nombre de voitures entre la voiture rouge et la sortie, utilisez le champ `rh` fourni de la classe RushHour pour accéder à la position des voitures dans la classe State.

   Attention : Ne pas utiliser la matrice `free` pour cela, cela conduirait à des bugs dans le programme (interférences avec la méthode `move`)
2. Modifiez le champ `f` de la classe `State` pour prendre en compte cette heuristique.
3. Utiliser les exemples fournis (`solve22(), solve1() et solve40`) pour tester cet algorithme et le comparer aux précédents. Comparez le nombre d'état visités par chacun des algorithmes pour le problème `solve22()` et concluez.

Modifiez le code de façon à calculer le temps mis par chacun des algorithmes pour trouver la solution. Comparez le temps mis par chaque algorithme puis le ratio temps/(nb états visités). Conclure.

Bonus : Proposez une autre heuristique au problème et la tester.

6. Directives de remise

Le travail sera réalisé en équipe de deux ou trois. Vous remettrez un fichiers .zip par personne, nommés TD1_nom_prenom_matricule.zip. Vous devez également remettre un fichier pdf contenant une explication de vos implémentations ainsi que les réponses aux deux questions. Tout devra être remis avant le **18 février à 23h55**. Tout travail en retard sera pénalisé d'une valeur de 10% par jour de retard.

Barème :

partie 1 :	15pts
partie 2 :	15pts
partie 3 :	15pts
partie 4 :	15pts
partie 5 :	30pts
rapport :	10pts

*Ce TP a été imaginé par Steve Oudot et Jean-Christophe Filliâtre (Ecole polytechnique, France) et complété par Pierre Hulot et Rodrigo Randel