INF421

RIOULT Antonin et GADONNEIX Julien

Devoir maison – 01/02/2023

L'ensemble des codes écrits lors de ce projet est fourni en annexe.

Contexte et aperçu général du problème

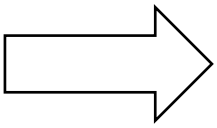
Nous nous intéressons au jeu RushHour, puzzle avec des véhicules horizontaux et verticaux sur une grille, dont l'objectif est de libérer la voiture rouge comme ci-contre. Nous allons ici chercher à implémenter une recherche de solution, avec comme objectif de trouver la solution la plus rapide (nombre de coups minimal). Nous allons dans un premier temps nous intéresser à un algorithme de force brute, puis nous nous intéresserons à l'utilisation d'heuristiques afin d'améliorer le temps d'exécution de celui-ci. La classe contenant le code des différentes parties est précisé à côté du nom des questions.

Figure 1 – Grille de RushHour

Question 1 : (Code dans CheckFile et IncorrectFile)

La première étape est de vérifier que le fichier texte donné correspond à une entrée valide. Informatiquement, nous avons décidé de représenter un état du jeu sous la forme d'une matrice. Cette structure a l'avantage d'être relativement simple à représenter et à implémenter et permet de facilement décrire l'évolution du jeu. Les 0 correspondent aux cases vides. Chaque voiture possède un id (numéro entier) différent qui apparaît sur chacune des cases où la voiture se trouve. Afin de vérifier la validité du fichier texte, nous commençons par créer une matrice de la taille de la grille remplie uniquement de 0. Nous lisons ensuite le fichier texte en positionnant petit à petit les id des voitures dans la matrice. Si l'on remarque un chevauchement, le fichier est incorrect et une exception *IncorrectFile* est levée :

```
6
8
1 h 2 2 3
2 h 2 1 1
3 h 2 5 5
4 h 3 3 6
5 v 3 6 1
6 v 3 1 2
7 v 3 4 2
8 v 2 1 5
```



```
220005
600705
611705
600700
800033
804440
```

Figure 2 – Du fichier texte initial à la représentation matricielle

Question 2 : (Utilise ColorCellRenderer, ModeleStatique et JTableBasiqueAvecModeleStatique)

Afin de représenter graphiquement un état du jeu, nous avons choisi d'associer une couleur à chaque id de la matrice (0=blanc, 1=rouge, etc…) puis d'afficher les couleurs dans un tableau. Pour implémenter cela nous avons utilisé les classes JFrame et CellRenderer puis nous nous sommes inspiré de versions déjà existantes :

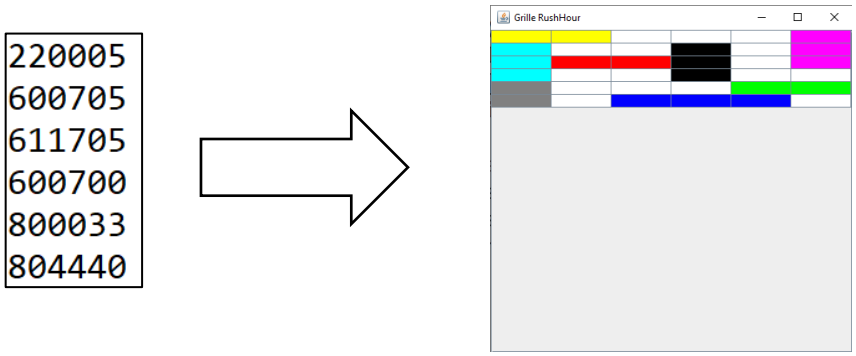


Figure 3 – De la représentation matricielle à l'affichage graphique

Question 3 :

Afin de trouver une solution optimale à la grille donnée, nous allons d'abord implémenter un algorithme de force brute. Nous nous sommes inspirés de l'algorithme « Breadth First Search » de parcours en largeur. Ne connaissant pas à l'avance les coups à réaliser, nous nous éloignons de la grille initiale en réalisant progressivement tous les coups possibles. Nous analysons d'abord tous les états distants d'un seul coup, puis ceux distants de deux coups, puis de trois, etc… Ainsi, lorsque nous trouvons une grille solution (la voiture rouge est collée au bord droit), nous sommes sûrs que c'est une solution optimale.

Pour éviter de former des boucles (une voiture bougeant en avant puis en arrière puis en avant), nous stockons les états visités. Concernant l'implémentation du pseudocode en Java, nous avons choisi une table de hachage afin de stocker les états déjà visités puisqu'il s'agit de la meilleure structure de données permettant de stocker des objets lorsque l'on veut ajouter et rechercher des éléments efficacement. Nous avons implémenté les méthodes *hashcode()* et *equals()* (par *@Override*) afin de permettre à la recherche dans la table de hachage de s'intéresser au contenu de la grille et non au pointeur de l'objet.

Concernant la file de priorité nous avons d'abord choisi une liste chaînée dans laquelle nous ajoutons les états par la fin, pour être sûr de les traiter dans l'ordre de leur distance à la grille initiale. Dans un second temps nous avons utilisé une file de priorité en implémentant le comparateur correspondant afin d'obtenir des résultats comparables aux algorithmes utilisant des heuristiques que nous présentons par la suite. Voici le pseudocode de notre algorithme :

```
Entrée : Une matrice M décrivant l'état initial du jeu.
Sortie : Une matrice décrivant l'état du jeu à la fin d'une solution optimale.
Algorithme : Créer une table de hachage H qui contiendra les états déjà visités.
Créer une file de priorité Q qui contiendra les prochains coups possibles. La
priorité est établie sur la distance (nombre de coups) à la grille initiale.
Q.ajouter(M)
Créer une matrice "Current"
Tant que (Q n'est pas vide) :
      Current=Q.retirer()
      Si(H ne contient pas Current) :
            H.ajouter(Current)
            Si(Current.estSolution()) :
                  Renvoyer (Current)
            Sinon :
                  Créer une liste vide L destinée à contenir des matrices
                  distantes d'un coup de Current
                  L=Current.prochainsCoupsPossibles()
                  Q.ajouter(L) #N'ajouter que les états qui ne sont pas dans H

      Sinon :
            Continuer la boucle tant que.
Renvoyer (erreur, aucune solution trouvée).
```

Questions 4,5 et 6 : (Code dans NextMoves, BruteForce, BruteForcetest et GridCounter)

Nous devons implémenter une méthode établissant la liste de tous les états distants d'un seul coup à partir d'une grille donnée. Pour cela, nous itérons sur les voitures :

1. Nous recherchons la voiture sur la grille et obtenons sa position, sa longueur et son orientation.
2. Nous regardons, dans sa direction, le nombre de cases vides se trouvant à l'avant et à l'arrière.
3. Nous en déduisons tous les mouvements possibles pour cette voiture.

Pour le cas de la voiture noire, cela produit les grilles suivantes :

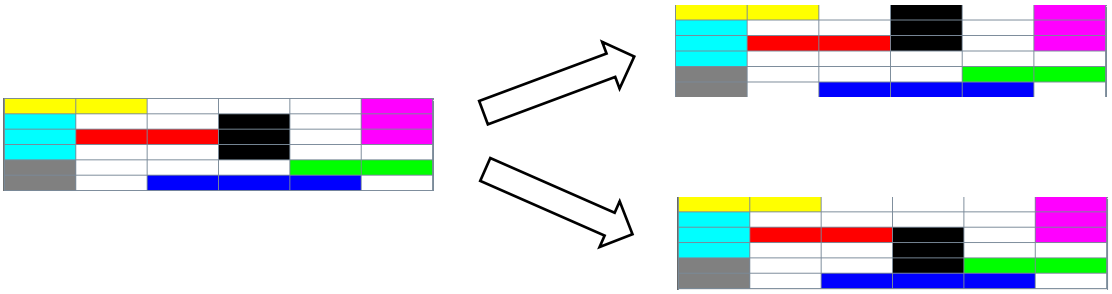


Figure 4 – Exemple de grilles suivantes possibles

Nous comparons les performances de nos différents algorithmes à la fin de ce rapport.

Question 7 : (Code dans PrintSolution)

Notre objectif est maintenant d'afficher les différentes étapes menant à la solution. Pour cela nous avons créé une nouvelle classe « GridCounter » représentant nos états du jeu. Un objet de cette classe est constitué de la grille en elle-même, d'un compteur afin de connaître sa distance à la grille initiale ainsi que d'un pointeur vers la grille précédente. Ainsi, afin de retracer le chemin suivi, il suffit de remonter le chemin à partir de la solution. Pour le cas de la grille suivie depuis le début :

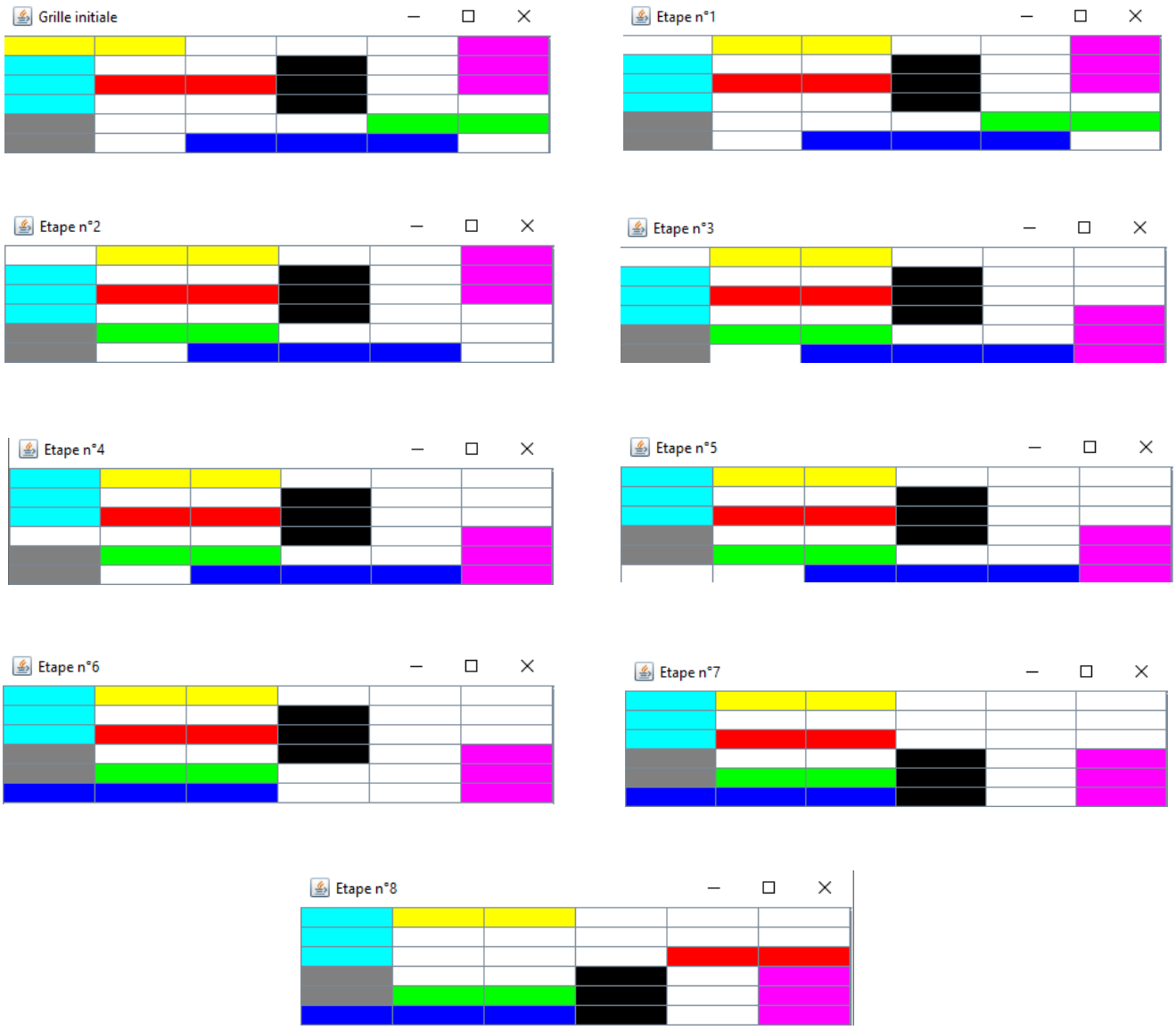


Figure 5 – Exemple de résolution

Question 8 :

Pour améliorer la complexité en temps (et en nombre de grilles visitées), nous implémentons l'usage d'heuristique dans notre algorithme. Nous utilisons maintenant une file de priorité dont l'ordre est établi par la valeur de la fonction *g* sur une grille :

g : *grille → nombre de coup depuis la grille initiale + heuristique (grille)*

La fonction *g* est sensée approcher la longueur de chemin le plus court passant par la grille considérée. C'est une borne inférieure, qui est d'autant plus pertinente que l'heuristique est performante.

De plus, à partir d'une grille donnée, les grilles suivantes (générées par *NextMoves*) ne peuvent pas prendre une valeur de *g* plus faible :

Soit *current* une grille donnée et *next* une grille distante d'un coup de current.

- g(current) = count(current) + h(current)

Alors, g(next) = (count(current) + 1) + h(next). Or, h étant consistante :

- $h(current) - 1 \leq h(next) \leq h(current) + 1$

Ainsi, $g(current) \leq g(next)$ et g ne peut ainsi pas diminuer le long d'une suite de coups.

Or, la priorité de file étant établie sur *g*, une fois une solution trouvée (g(solution) = nombre de coups de la solution), nous ne pourrons pas trouver de solution avec une valeur de *g* plus faible donc cette solution est optimale. L'algorithme est donc correct.

```
Entrée : Une matrice M décrivant l'état initial du jeu, une heuristique h.
Sortie : Une matrice décrivant l'état du jeu à la fin d'une solution optimale.
Algorithme : Créer une table de hachage H qui contiendra les états déjà visités.

Créer une file de priorité Q qui contiendra les prochains coups possibles. La
priorité est établie sur la valeur de la fonction g (nombre de coups depuis la
grille initiale + valeur de h).
Q.ajouter(M)
Créer une matrice "Current"
Tant que (Q n'est pas vide) :
      Current=Q.retirer()
      Si(Current n'est pas dans H OU s'il y est avec une valeur de g plus grande) :
      #Cette deuxième condition permet d'explorer le même état s'il est rencontré
      une seconde fois, après un nombre de coups plus faibles (et donc un chemin
      plus court car l'heuristique peut être mauvaise)
            H.ajouter(Current)
            Si(Current.estSolution()) :
                  Rencoyer(Current).
            Sinon :
                  Créer une liste vide L destinée à contenir des matrices distantes
                  d'un coup de Current
                  L=Current.prochainsCoupsPossibles()
                  Q.ajouter(L) #N'ajouter que les états qui ne sont pas dans H ou
                  bien qui ont une valeur de g plus faible que l'état similaire
                  déjà visité
Renvoyer (erreur, aucune solution trouvée).
```

Lors de l'implémentation de la file de priorité, nous avons modifié l'objet « GridCounter » en lui rajoutant un champ *g* et nous avons également modifié le comparateur lié à cette classe afin que la priorité se fasse sur *g*.

L'heuristique constante égale à 0 revient à trier les grilles dans la file selon leur distance à la grille de départ. Ainsi, cela revient à l'algorithme de force brute établi à la question 3. Cela sera confirmé par l'analyse de performances en fin de rapport.

Question 9 :

Soient *s*, *s'* deux grilles données et $k_{s,s'}$ le nombre de coups les séparant.

On note $\Delta = | h(s) - h(s') |$ la différence du nombre de voitures entre la voiture rouge et la sortie entre les états *s* et *s'*. Ces voitures ont donc bougé, le nombre de coup est ainsi au moins égal à Δ :

$$\Delta = | h(s) - h(s') | \leq k_{s,s'}$$

Ainsi, l'heuristique proposée est consistante.

Question 10 : (Code dans HeuristicSolving et HeuristicGridCounter)

Les performances de l'algorithme avec cette heuristique sont légèrement meilleures que celui de force brute. Les résultats obtenus (nombre d'états du jeu visités avant de trouver la solution et temps d'exécution) sont comparés dans la dernière partie.

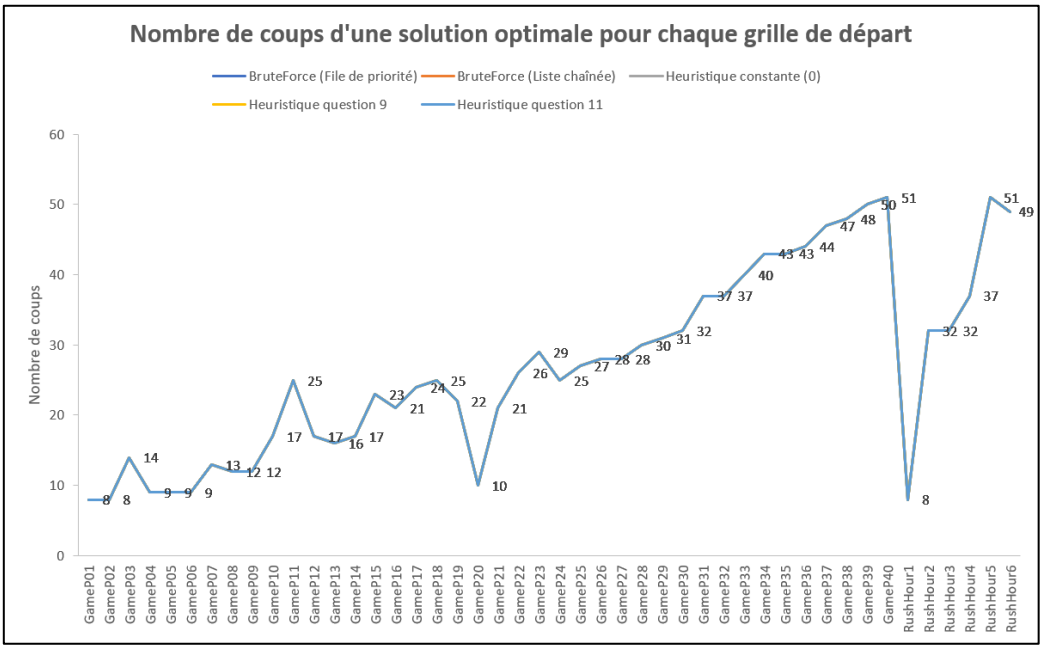
Question 11 : (Code dans HeuristicSolving et PairListBoolean)

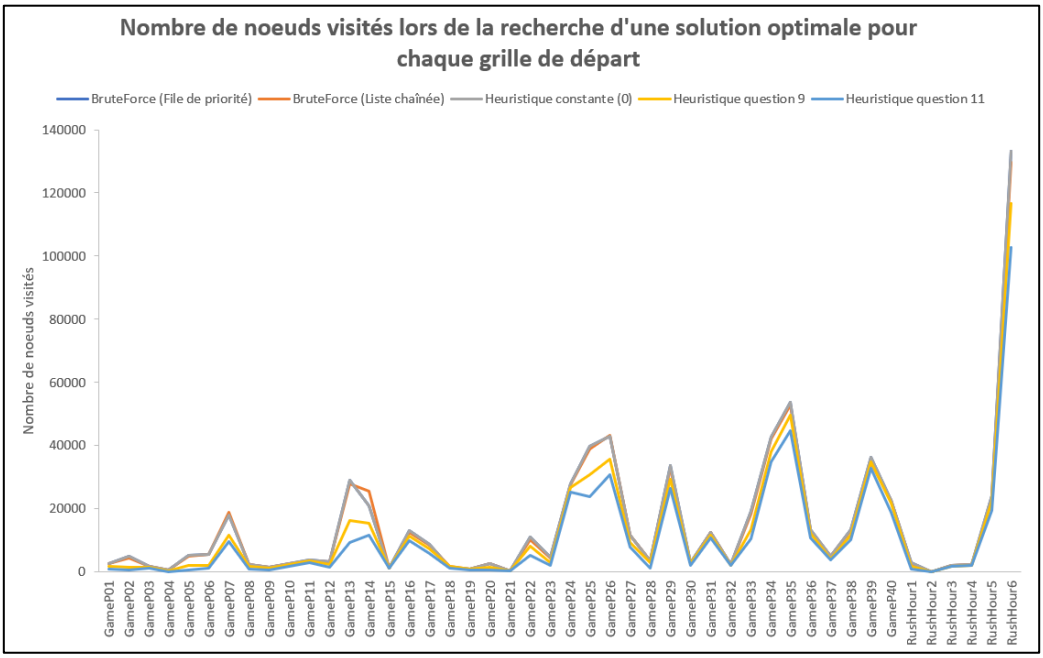
Afin d'augmenter encore les performances de notre algorithme, nous cherchons à utiliser une heuristique plus performante. Notre idée est de d'améliorer l'heuristique utilisée en question 9 et de rajouter le nombre de voiture bloquant les voitures se situant entre la voiture rouge et la sortie. En effet, afin de bouger les voitures situées entre la voiture rouge et la sortie il est nécessaire de bouger également les véhicules bloquant ces voitures. Grâce à une preuve similaire à celle présentée en question 9, l'heuristique est consistante. Cette heuristique est la troisième implémentée dans la classe *HeuristicSolving*.

Il est possible de pousser cette démarche afin de prendre en compte les voitures bloquant les voitures qui bloquent les voitures, …… de manière récursive en vérifiant que l'on évite les cycles. Cela permettrait d'obtenir une heuristique encore plus performante, mais il est possible que les répercussions sur le temps de calcul soient néfastes. Cependant, nous n'avons pas eu le temps de le vérifier.

Comparaison des performances des différents algorithmes présentés :

Ci-dessous sont présents les graphiques détaillant les performances des différents algorithmes sur trois critères : le nombre de coups d'une solution optimale, le nombre de grilles visitées avant d'y parvenir ainsi que le temps d'exécution. Nous remarquons que le nombre de coups pour arriver à une solution est similaire pour chaque algorithme : chaque algorithme trouve une solution optimale. Ensuite, le nombre de nœuds visités est minimisé lors de l'emploi d'heuristiques. Cependant, le calcul de l'heuristique pour chaque nœud visité peut avoir un impact contraire sur le temps d'exécution.





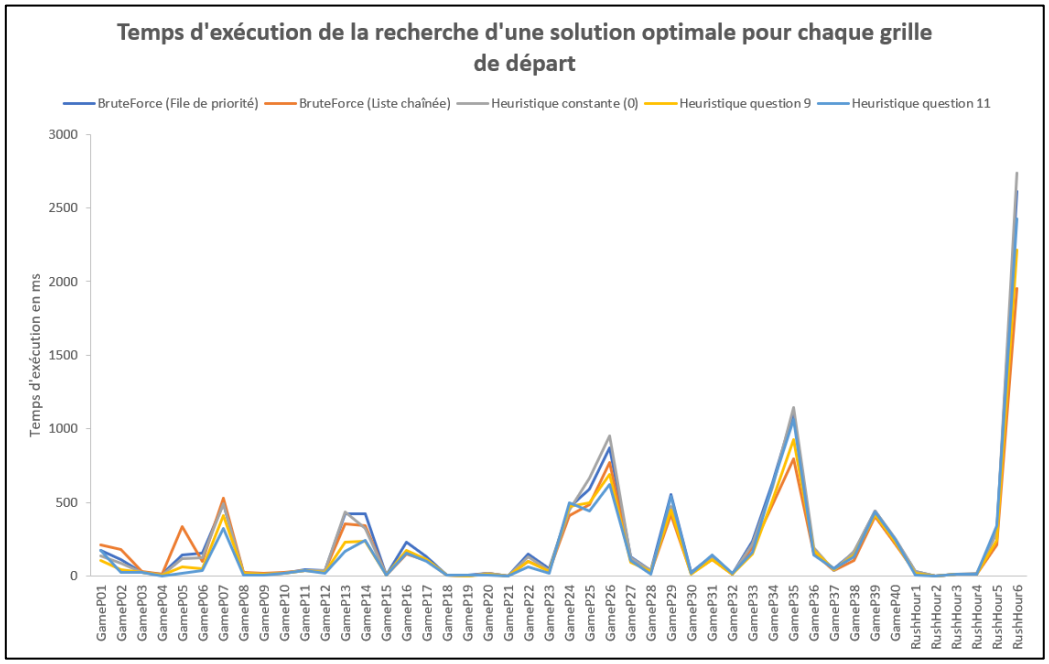


Figure 6 – Comparaison des performances