HERBEAUX Raphaël,
ATTARIAN Keyvan

16 Février 2025

RAPPORT DE PROJET D'INFORMATIQUE

Unblock me

Table des matières

1 Introduction

L'objectif de ce projet est de trouver la meilleure solution au jeu *Unblock Me* (ou *Rush Hour*). Aussi, nous cherchons à déterminer comment amener la voiture rouge vers la sortie en le moins d'étape possible. La voiture rouge est bloquée par des voitures verticales et horizontales, les voitures verticales ne pouvant se déplacer que verticalement et les voitures horizontales horizontalement. Celles-ci ne peuvent bien sûr pas se traverser, et peuvent, en une étape, se déplacer d'autant de cases que le joueur le souhaite, tant que les règles énoncées ci-dessus sont respectées.

La première difficulté est de bien représenter nos données, c'est-à-dire la position de chaque voiture. Ensuite, la question se reformule en un simple parcours de graphe, à l'intérieur duquel nous cherchons le plus court chemin d'un sommet a vers un ensemble de sommet $S = \{m : m \text{ est une solution du jeu}\}$

2 Mise en place du jeu

2.1 Structure de données

Pour représenter chaque situation du jeu, nous avons décidé d'utiliser deux classes, une classe `Car` et une classe `Game`.

Un objet de la classe `Game` représentera une situation donnée du jeu. Nous utilisons une liste d'objets de type `Car`, représentant chacun une voiture. Nous gardons l'invariant suivant :

```
cars[k].id = k+1
```

Pour plus de simplicité, nous construisons aussi une tableau numpy en 2D, représentant la grille de jeu. Chaque voiture y est alors représentée par son identifiant.

2.2 Initialisation

Pour construire un jeu à partir du fichier texte, il suffit de le lire ligne à ligne et d'ajouter une par une les voitures. Ensuite, nous utilisons la fonction `updateGrid` qui met à jour la grille pour qu'elle corresponde à la liste des voitures. Si nous rencontrons une contradiction lors de la construction (c'est à dire que l'on veut mettre une voiture sur une case déjà occupée), nous renvoyons `False`, et `True` dans le cas ou tout se passe bien.

2.3 Affichage

Pour l'affichage, nous utilisons la fonction `imshow` de la bibliothèque `matplotlib`. Nous construisons une `colormap` personalisée en associant à chaque nombre de $[|0, \texttt{nbrCar}|]$ une couleur. L'affichage se fait par l'appel d'une methode `show` de l'objet jeu.

3 Calcul de la longueur de la meilleure solution

3.1 Algorithme & pseudo-code : BFS

On peut voir ce problème comme celui du parcours d'un graphe : chaque état du jeu est un sommet, et le sommet a est relié au sommet b si on peut aller de a à b en un mouvement.

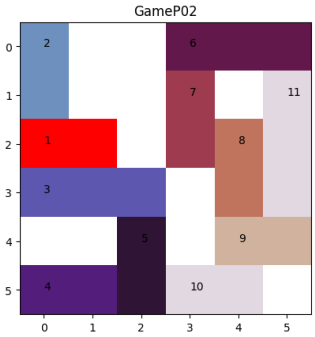


Figure 1 – Représentation de l'état inital du jeu P02

Nous implémentons donc un algorithme type BFS (*Breadth-First Search*, parcours en largeur du graphe).Voilà le pseudo-code d'un tel algorithme :

```
BFS(start):
    queue = créer_file()
    visited = créer_ensemble()

    enfiler(queue, (start,0))
    ajouter(visited, start)

    tant que non_vide(queue):
        move, profondeur = défiler(queue)
        si move est final:
            retourner profondeur
        pour chaque voisin dans voisins(move):
            si voisin n'est pas dans visited:
                ajouter(visited, voisin)
                enfiler(queue, (voisin,profondeur+1)
```

Le principe de l'algorithme du parcours en largeur d'un graphe est d'utiliser une file d'attente, et d'enfiler un après l'autre les voisin visités. On visite donc le graphe des états prondeur par prondeur, et dès que l'on tombe sur un état final, on sait qu'il est de profondeur minimale.

3.2 Structures de données

Il faut donc se doter de deux structures de données : la file d'attente, et un ensemble. La file d'attente est directement implémentée en python grace au module `collections` et par sa classe `deque`. L'ajout et le retrait se font en $\mathcal{O}(1)$. Si nous n'avions pas utilisé `deque`, nous aurions pu utiliser une liste doublement chainée.

Nous souhaitons utiliser `set` comme structure de données pour `visited`. Cette structure de données est l'implémentation python d'une table de hashage. Cependant, il faut y stocker des états de jeu, c'est-à-dire des instance de `Game`, qui ne sont a priori pas des données hashable. Nous dotons donc la classe `Car` d'une méthode `__hash__()`, qui hashe une voiture en renvoyant le hash du tuple (id,position,longueur,x,y). En suite, nous dotons `Game` d'une méthode `__hash__()`

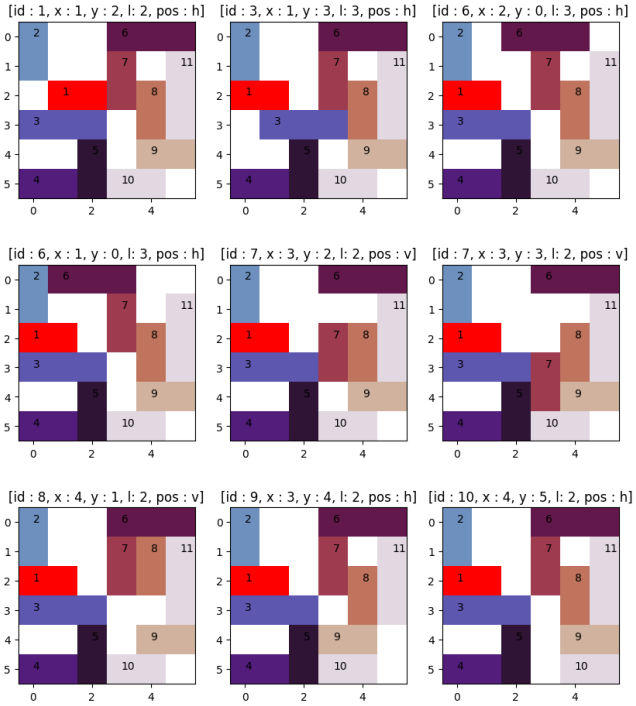


Figure 2 – L'ensemble des voisins de l'état décrit en figure 1

qui renvoie le hash du tuple associé au tableau contenant chaque voitures. Grace à ces implémentation, on peut tester si un état à déjà été visité en $\mathcal{O}(1)$.

3.3 Determiner les voisins d'un état

Il ne reste plus qu'à se doter d'une fonction donnant l'ensemble des états accessibles depuis un état donné. La fonction `possibleMove` parcourt chaque voiture et vérifie les cases vides autour de celle-ci pour déterminer les mouvements possibles. Si la voiture est horizontale, elle vérifie les mouvements vers la gauche et la droite. Si elle est verticale, elle vérifie les mouvements vers le haut et le bas. Les mouvements valides sont ajoutés à une liste, qui est ensuite retournée. Les mouvements sont ajoutés grâce à une méthode `copy` de la classe `Game`. Celle ci construit un nouvel état avec les mêmes voitures aux mêmes positions, à l'exception de celle passée en argument.

Comme on considère que la taille de chaque voiture est toujours 2 ou 3, et donc que l'ajout d'une voiture à la grille se fait en $\mathcal{O}(1)$, la fonction `updateGrid` s'execute en $\mathcal{O}(|\text{voitures}|)$. Aussi, la copie se fait en $\mathcal{O}(|\text{voitures}|)$. On copie une fois par voisin, et donc la fonction `possibleMove` s'exécute en $\mathcal{O}(|\text{voitures}| \times |\text{voisins}|)$. Or il y a au plus $n = \texttt{size}$ voisins possibles par voiture. Aussi, la complexité de la recherche de voisins est en $\mathcal{O}(n|\text{voitures}|^2)$.

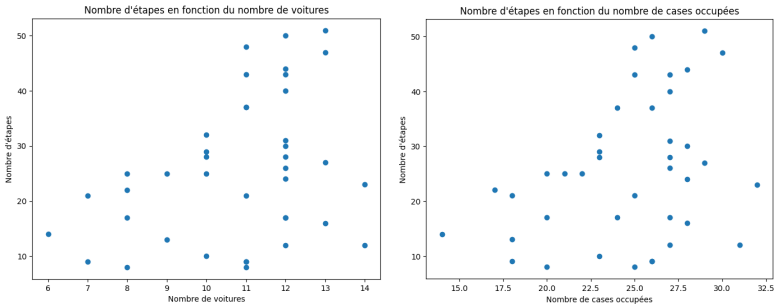
Un exemple est donnée en figure 2.

3.4 Temps d'execution et complexité

Pour tester si un état est final, il suffit de regarder la position de la voiture rouge. Ceci se fait en $\mathcal{O}(1)$. Notre algorithme visite chaque sommet une fois exactement, et teste en temps constant s'il est final. Chaque voisin est ajouté en $O(|\text{voitures}|)$, et ceci se fait au plus $|\text{arêtes}|$. Par ailleurs,

pour chaque sommet, on sait qu'il y a au plus $\mathcal{O}(n|\text{voitures}|)$ voisins. L'initialisation se faisant en $O(|\text{voitures}|)$, la complexité de notre algorithme est :

$$\mathcal{O}(|\text{sommets}| + |\text{arêtes}|n|\text{voitures}|) = \mathcal{O}(|\text{états possibles}| + n|\text{voitures}|^2)$$



(a) Nombre d'étapes en fonction du nombre de voitures

(b) Nombre d'étapes en fonction du nombre de cases occupées

Figure 3 – Analyse des étapes en fonction de différents paramètres

3.5 Reconstruction de la solution

Pour reconstruire la solution, on ajoute un attribut `move` à chaque copie, pointant vers l'état copié. Ceci permet de construire étape après étape, en partant de la fin, notre solution. Un exemple est donné en figure 4.

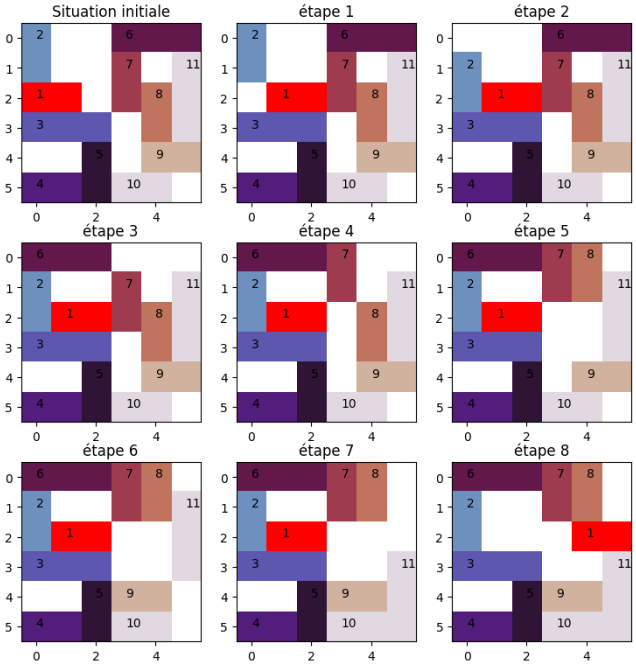


Figure 4 – Solution pour la situation initiale décrite dans GameP02.txt

4 Approche basée sur les heuristiques

4.1 Algorithme A*

On propose d'implémenter l'algorithme A*, qui permet de trouver le plus court chemin entre deux sommets (ou deux ensembles de sommets comme ici) en utilisant une heuristique. L'idée est la même que celle du BFS, mais au lieu d'utiliser une file classique, nous utilisons une file de priorité. Nous proposons le pseudo-code suivant :

```
A*(start,h):
    visited = dictionaire{}
    toVisit = filePriorité()
    toVisit.push((start,0),h(start))

    tant que toVisit est non vide:

        currentState, profondeur = toVisit.pop()
        si currentState déjà visité et que visited[currentState] <= profondeur:
            passer

        sinon, si currentState est final:
            retourner currentState, profondeur

        sinon:
            visited[currentState] = profondeur

            pour chaque voisin dans voisins(currentState):
                cout = profondeur + 1
                heuristique = cout + h(voisin)

                si le voisin n'est pas dans visited ou que visited[currentState] > cout:
                    toVisit.push((voisin,profondeur),heuristique)
    retourner une erreur
```

4.2 Démonstration de la correction de A*

Donons nous une heuristique h consistante. Montrons qu'àlors `A*(start,h)` renvoie le plus court chemin vers un état final.

Complétude Premièrement, notons que si l'algoritme termine avec une erreur, c'est une fois qu'il a visité toute la composante connectée de `start`. Aussi, s'il existe une solution à notre jeu, alors `A*(start, h)` en renverra toujours une.

Optimalité Par ailleurs, notre algorithme respecte l'invariant suivant : la variable `heuristique` d'un sommet s est toujours inférieure à $d(\texttt{start}, \{f : f\text{est final}\})$. En effet, soit f final. Comme h est admissible, $h(s) \leq d(s, f)$ pour tout s. On a donc

$$\forall s, \texttt{heuristique}(s) \leq d(\texttt{start}, s) + d(s, f) = d(\texttt{start}, f)$$

Aussi, si l'algorithme propose un sommet P de `start` à f_0 final, on a

$$\forall s \in P, \texttt{heuristique}(s) \leq d(\texttt{start}, \{f : f \text{est final}\})$$

En particulier,

$$\texttt{heuristique}(f_0) = d(\texttt{start}, f_0) \leq d(\texttt{start}, \{f : f \text{est final}\})$$

car on a toujours $h(f_0) = 0$. L'inégalité est donc saturée, et P est optimal.

4.3 Premières heuristiques

Si $h = 0$, alors h est bien consistante, et `A*` revient à un simple BFS.

Notons maintenant $h(s) =$ le nombre de voitures entre la voiture rouge et la sortie. Alors h est consistante. En effet, la voiture rouge ne peut que se déplacer horizontalement, et doit aller de sa position à la sortie, sur le bord droit de l'aire de jeu. Comme elle ne peut pas traverser les voitures qui sont entre elle et la sortie, il faudra au moins $h(s)$ étapes pour déplacer ces voitures.

4.4 Implémentation et résultats

On utilise un dictionaire python pour enregistrer le cout (réel) de chaque sommet visité. Ceci est possible grâce aux fonction de hash déjà impémentées.

Pour la file de priorité, nous utilisons le module `heapq` qui implémente une structure de *tas*, qui est l'implémentation classique d'une file de priorité. Notons que cette structure nécéssite qu'un ordre existe sur nos objets, et renvoie le plus petit élément selon cet ordre. Aussi, nous enfilons un tuple `(heuristique, profondeur, état)`, et utilisons l'odre lexicographique déjà implémenté sur les tuples. Il faut cependant donner un ordre sur les états, et pour cela nous définissons une méthode `__lt__(self,other)` qui renvoie un booléen quelconque.

En théorie, la complexité ne diminue pas : dans le pire des cas, A* est aussi mauvais qu'un BFS. Cependant, sur les 40 exemples donnés, on obtient une diminution de 13% de nombre de sommets visités.

4.5 Autres heuristiques

Une des premières idées que nous avons eu pour trouver de nouvelles heuristiques adaptées à ce problème fut de prendre en compte les voitures horizontales dans notre calcul. En effet, l'heuristique précédente n'utilise que la position des voitures verticales, nous pensions donc aspirer à plus de précision en considérant plus de voitures.

La première heuristique à laquelle nous avons pensé s'inspire de la précédente. Pour un état donné, on commence par ajouter 1 pour chacune des voitures verticales positionnées entre la voiture rouge et la sortie. Puis, pour chacune des voitures verticales qui bloquent la rouge, on observe si elle est bloquée par des voitures horizontales qui l'empêchent de libérer le chemin devant la voiture rouge en un seul déplacement. Si c'est le cas, on ajoute 1 de plus à l'heuristique dans cet état. Nous pensions, en implémentant cette heuristique, que si une telle voiture verticale était bloquée de la sorte, il aurait fallu au moins deux déplacements pour la décaler hors du chemin de la voiture rouge, ce qui aurait garanti la consistance de l'heuristique.

Cependant, nous n'avions pas pensé au fait qu'il pouvait exister des situations où lorsque l'on déplaçait une voiture horizontale une unique fois, elle pouvait libérer le chemin de deux voitures

verticales. Si ces deux dernières bloquaient la voiture rouge auparavant, la valeur de notre heuristique pourrait donc décroître de deux entre deux états séparés d'un unique déplacement. Ainsi, cela montre que cette heuristique n'est pas consistante.

Nous l'avons quand même implémentée, elle est efficace en nombre d'étapes effectuées pour arriver à une solution mais on constate effectivement sa non consistance puisqu'elle ne renvoie pas nécessairement le chemin le plus court.

Pour pallier ce problème de non consistance, nous avons décider de conserver une idée très similaire en ajustant simplement la valeur de la pénalité que nous ajoutions lorsque qu'une voiture verticale, étant sur le chemin de la rouge, était bloquée par une voiture horizontale. On observe la voiture horizontale la plus longue qui bloque la voiture verticale et on ajoute l'inverse de sa longueur à l'heuristique dans l'état considéré. En effet, cette voiture horizontale peut bloquer au maximum un nombre de voitures verticales égal à sa longueur L, ainsi, en la déplaçant, on libère au maximum un nombre de voitures égal à L et l'heuristique ne peut qu'augmenter ou diminuer de 1 au plus. L'autre voiture qui bloque (si elle existe, cela peut aussi être le mur) est au moins plus petite, si on la déplace on libère moins de voitures verticales au maximum donc l'heuristique ne varie que de 1 au maximum.

Ainsi, à chaque déplacement, l'heuristique peut ne pas changer de valeur, augmenter ou diminuer de 1, ce qui prouve qu'elle est consistante.

Les résultats que l'on obtient cependant sont bien moins satisfaisants qu'avec notre heuristique précédente comptant le nombre de voitures verticales entre la voiture rouge et la sortie. On obtient effectivement le chemin le plus court mais en un nombre d'étapes à chaque fois très légèrement inférieur à l'algorithme n'utilisant pas d'heuristique. Sur les 40 exemples donnés, on obtient une diminution de 5% du nombre d'étapes effectuées avant de trouver la solution.