IA41
Rush Hour

Adrien Burgun, Jean-Mael Legrand, Erwann Le Guilly, Mohamed Jani

Décembre 2021

Résumé

À l'occasion de la matière IA41 (« Intelligence artificielle : concepts fondamentaux et langages dédiés »), nous avons choisi comme sujet de projet le jeu « Rush Hour » (aujourd'hui vendu par *ThinkFun*).

Dans ce document, nous verrons d'abords la modélisation, puis une implémentation algorithmique. Ensuite, nous verrons les résultats de l'implémentation en Python ainsi que les optimisations à cette implémentation, qui ont été faites en Rust, un langage bas-niveau.

L'ordre de ce document diffère de l'ordre d'implémentation de ce jeu : nous avons d'abords implémenté une solution en Rust, un langage bas-niveau, et optimisé celle-ci, puis nous avons fait une solution en Python, accompagnée d'une interface graphique.

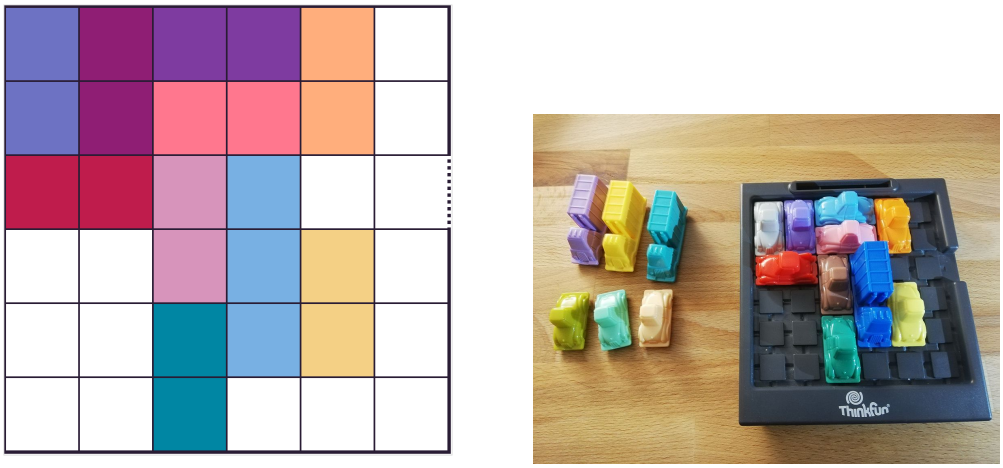
Le code pour ce projet est disponible sur github, à l'adresse suivante : https://github.com/adri326/ia41-project

Table des matières

1 Introduction

L'objectif de ce projet est de créer un programme qui va être capable de jouer à « Rush Hour » à partir de niveaux créés à l'avance.

Rush Hour se joue sur une grille de 6x6 cases, avec des véhicules, qui sont soit longs de 2 cases (représentant des voitures), soit longs de 3 cases (représentant des camions). Les véhicules peuvent avancer et reculer, mais ne peuvent ni tourner, ni passer à travers un autre véhicule, ni s'arrêter entre deux cases, ni sortir du plateau. Le but est de faire sortir la voiture rouge : une ouverture est présente à sa hauteur sur le plateau, mais l'accès à celle-ci est bloqué par d'autres véhicules. Pour y arriver, il faut déplacer ces véhicules un à un jusqu'à ce que la voiture atteigne la sortie, que nous définissons comme étant le bord droit du terrain.



(a) Affichage dans la version python (b) Mise en place sur le plateau physique

FIGURE 1 – Une position d'exemple (position numéro 3), affichée avec python et reproduite sur le plateau physique

2 Modélisation

La résolution de ce jeu est modélisée en tant que problème de décision séquentielle.

Chaque état du plateau peut être encodé de deux manières :

— Une matrice de taille 6×6, contenant un indice unique à chaque véhicule : $\mathbb{N}_{6\times 6}$
— Une liste de positions, directions et longueurs pour chaque véhicule $\mathbb{P} = ([0;6[\times [0;6[\times \{H, V\} \times [2;3])^*$

Il est possible de travailler sur le premier encodage, mais nous avons préféré utiliser le deuxième encodage au sein de l'algorithme pour sa simplicité d'implémentation. Les positions sont indiquées au programme sous un format texte équivalent au premier encodage, qui est ensuite traduit avant d'être donné à l'algorithme de résolution dans le deuxième encodage.

2.1 Fonction de transition

À partir d'une position donnée, nous pouvons trouver la liste des positions suivantes avec l'algorithme `next_states` (Algorithme 1). Cet algorithme utilise la fonction `case_vide` ; son implémentation est simple, car il suffit d'itérer sur $\mathbb{P}$ et de vérifier qu'aucune voiture ne se trouve à la position (x, y) recherchée (implémentée en Python et donné avec l'Algorithme 2). Cependant, il est possible de grandement accélérer cette fonction en maintenant un tableau des cases occupées : cette accélération est à la base de l'implémentation en Rust et elle permet notamment de réduire la complexité de `next_states` de $\mathcal{O}(n^2)$ à $\mathcal{O}(n)$.

Il est souvent nécessaire de bouger une voiture sur plusieurs cases : nous pouvons facilement modifier la fonction `next_states` pour donner les positions résultant d'un mouvement sur plusieurs cases. Afin d'obtenir une solution idéale, il faut cependant s'assurer que pour $0 < i < j$, la position p pour laquelle le véhicule $v \in p$ se déplace de i case apparaît avant la position où v se déplace de j cases dans la liste `résultat`.

Algorithme 1 : next_states(position)

```
   Données : position: ℙ
   Résultat : résultat: ℙ*
 1 Début next_states(position) : ℙ*
 2 |  résultat ⟵ [ ]
 3 |  Pour v dans position faire
 4 |  |  Si v→dir = H alors
   |  |  |  // Si le véhicule est à l'horizontale:
 5 |  |  |  Si case_vide(position, v→x − 1, v→y) alors
   |  |  |  |  // Si on peut reculer de 1 case vers la gauche
 6 |  |  |  |  (v→x) ⟵ v→x − 1
 7 |  |  |  |  résultat ⟵ résultat + [position ∪ v]
 8 |  |  |  FinSi
 9 |  |  |  Si case_vide(position, v→x + v→length, v→y) alors
   |  |  |  |  // Si on peut avancer de 1 case vers la droite
10 |  |  |  |  (v→x) ⟵ v→x + 1
11 |  |  |  |  résultat ⟵ résultat + [position ∪ v]
12 |  |  |  FinSi
13 |  |  Sinon
   |  |  |  // Même chose pour un véhicule à la verticale:
14 |  |  |  Si case_vide(position, v→x, v→y − 1) alors
15 |  |  |  |  (v→y) ⟵ v→y − 1
16 |  |  |  |  résultat ⟵ résultat + [position ∪ v]
17 |  |  |  FinSi
18 |  |  |  Si case_vide(position, v→x, v→y + v→length) alors
19 |  |  |  |  (v→y) ⟵ v→y + 1
20 |  |  |  |  résultat ⟵ résultat + [position ∪ v]
21 |  |  |  FinSi
22 |  |  FinSi
23 |  Fin
24 Fin
```

Algorithme 2 : case_vide(position, x, y)

Données :
- position: $\mathbb{P}$
- x: $[0; 6[$
- y: $[0; 6[$

Résultat : $\mathbb{B}$

Classe : $\mathcal{O}(len(\text{position}))$

```
1 Début case_vide(position, x, y) : 𝔹
2 |  Pour v dans position faire
3 |  |  Si v recouvre (x, y) alors
4 |  |  |  Retourner faux
5 |  |  FinSi
6 |  Fin
7 |  Retourner vrai
8 Fin
```

2.2 Algorithme de résolution

Les différentes positions accessibles depuis une position initiales forment un graphe non-dirigé, où chaque noeud est une position et chaque arrête un mouvement de véhicule. Nous pouvons alors utiliser un algorithme d'exploration de graphe, avec pour seule difficulté la cyclicité du graphe lorsque deux véhicules peuvent se déplacer indépendemment.

L'algorithme suivant est un algorithme d'exploration en largeur (Breadth-First Search). L'opération « `next est un état final` » revient à regarder si la voiture rouge (à sortir) a atteint le bord droit.

L'opération « `next` $\in$ `closed` » peut être implémentée avec une recherche linéaire dans `closed` (opération de complexité $\mathcal{O}(n * m)$), mais il est également possible d'utiliser un HashSet, amenant la complexité à $\mathcal{O}(Time[=_{\mathbb{P}}] + Time[\texttt{hash}]) = \mathcal{O}(n + Time[\texttt{hash}])$ pour une fonction `hash` optimale.

Nous proposons dans l'implémentation Python une fonction `hash` basique, définie comme $\texttt{hash}(pos) = \texttt{reduce}(\texttt{map}(pos, \lambda v \ . \ \texttt{hash}(v)), \text{XOR})$ (de complexité $\mathcal{O}(n)$. La fonction `hash` dans l'implémentation Rust fait la discussion d'une partie ultérieure.

Algorithme 3 : BFS(position)

```
   Données : position: ℙ
1  Début BFS(position)
2  |  open ⟵ [(position, [ ]]
3  |  closed ⟵ [ ]
4  |  Tant que (p, chemin) ⟵ pop_front(open) faire
5  |  |  Pour next dans next_states(p) faire
6  |  |  |  Si next est un état final alors
7  |  |  |  |  Retourner chemin + [next]
8  |  |  |  Sinon si non(next ∈ closed) alors
9  |  |  |  |  push_back(open, (next, chemin + next)
10 |  |  |  |  closed ⟵ closed + [next]
11 |  |  |  FinSi
12 |  |  Fin
13 |  Fin
14 |  Retourner none
15 Fin
```

Dans la position donnée en introduction (Figure 1), l'algorithme BFS éxecute 2298 itérations avant de trouver la solution. L'implémentation Rust trouve la solution en 3.6 ± 0.4 ms (AMD Ryzen 7 2700X). Pour la même position, l'implémentation Rust prend 21ms avec l'algorithme Iterative Deepening Depth-First Search (IDDFS).

Listing 1 – Solution proposée par l'algorithme BFS

```
23118
23998
0046
  467
  567
  5

7: 4,3 -> 4,4
8: 4,0 -> 4,2
9: 2,1 -> 3,1
6: 3,2 -> 3,3
1: 2,0 -> 3,0
4: 2,2 -> 2,0
0: 0,2 -> 2,2
3: 1,0 -> 1,3
```

```
2: 0,0 -> 0,3
0: 2,2 -> 0,2
4: 2,0 -> 2,2
9: 3,1 -> 0,1
1: 3,0 -> 0,0
8: 4,2 -> 4,0
4: 2,2 -> 2,0
0: 0,2 -> 4,2

114 8
994 8
   00
23 6
23567
 567
```

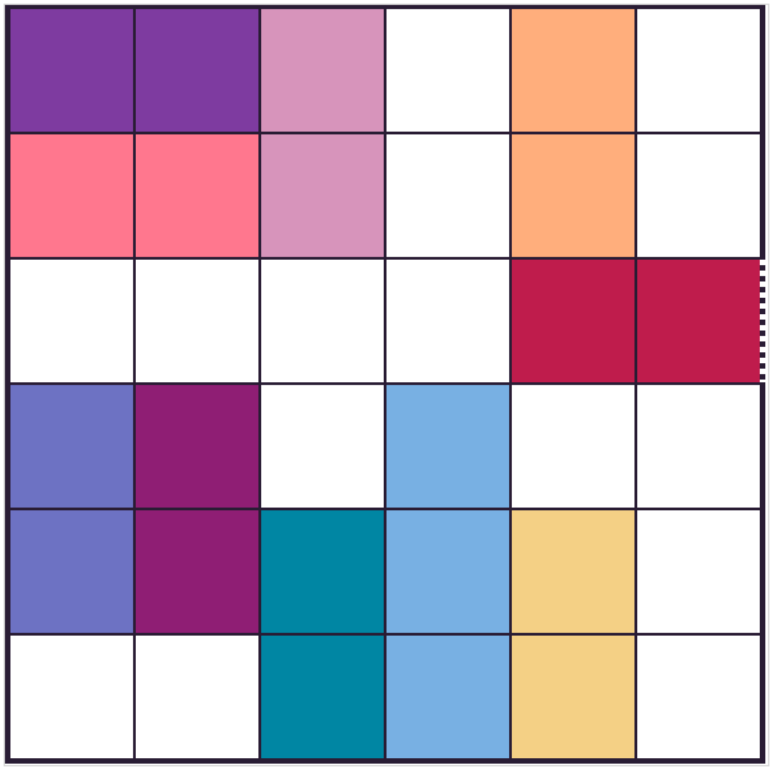


FIGURE 2 – Position finale

3 Organisation du projet

Le projet est séparé en différents fichiers, tous stockés dans une même repository en ligne. Ceci nous a permis de facilement gérer les versions de ces fichiers et la synchronisation du travail entre les membres du groupe.

Chaque fichier de code gère un aspect du projet :

— `board.py` contient la classe `Board`, qui stocke un état du plateau. Le fichier contient aussi différentes méthodes utiles à l'implémentation de l'algorithme de résolution
— `solve.py` contient la fonction `bfs`, qui résout une position avec l'algorithme BFS (Algorithme 3)
— `parse.py` contient la fonction `parse`, qui lit une chaîne de caractère et la transforme en instance de `Board`
— `canvas.py` contient uniquement une classe pour gérer la taille de la fenêtre avec le canvas de TKinter
— `render.py` contient le code pour afficher un état dans la fenêtre
— `main.py` appelle les fonctions des autres fichiers et initialise la fenêtre et la structure de donnée
— Le dossier `boards` contient un jeu de données, sous format texte
— Le dossier `rust` contient une implémentation alternative, en langage Rust

4 Optimisation et implémentation en Rust

Une seconde implémentation, faite dans le langage Rust (proche du C++ mais simplifiant la gestion de la mémoire), est proposée avec le code source de ce projet. Le but de cette deuxième implémentation était de mesurer la vitesse du programme lorsque celui-ci est écrit dans un langage dédié pour ceci, ainsi que de mesurer l'impact de différentes optimisations.

4.1 Stockage d'une carte des cases occupées

Une première optimisation revient à mélanger les deux modes de représentation ($\mathbb{N}_{6\times6}$ et $\mathbb{P}$), afin d'utiliser le plus rapide des deux pour les différentes

opérations. On construit alors un tableau des cases occupées et non occupées. Vu que ce tableau ne contient qu'un bit d'information par case et que le terrain contient 36 cases, il est possible de le stocker dans un mot de 64 bits :

$$\text{position}\rightarrow\text{bitmap} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \longleftrightarrow 0\text{x11c70f7df} \tag{1}$$

La fonction `case_vide` peut alors être grandement accélérée avec ce tableau :

Algorithme 4 : case_vide'(position, x, y)

Données :
- position: $\mathbb{P}$
- x: $[0; 6[$
- y: $[0; 6[$

Résultat : $\mathbb{B}$

Classe : $\mathcal{O}(1)$

```
1 Début case_vide'(position, x, y) : B
2 |  Retourner position→bitmap[x + 6 * y]
3 Fin
```

L'opération de mise à jour de la position d'un véhicule requiers de modifier ce tableau : il suffit de mettre les cases où se trouvait le véhicule à 0 puis de mettre les cases où le véhicule se trouve à nouveau à 1.

4.2 Utilisation de la carte dans le HashSet

La fonction `hash` proposée plus tôt a une complexité de $\mathcal{O}(n)$, mais il est possible de réduire cette complexité à $\mathcal{O}(1)$ si on utilise la bitmap proposée dans la dernière optimisation. Les fonctions $\texttt{hash}_1(pos) = \texttt{hash}(\text{pos}\rightarrow\text{bitmap})$ et $\texttt{hash}_2(pos) = \text{pos}\rightarrow\text{bitmap}$ ont une telle complexité. $\texttt{hash}_2$ est un peu plus rapide que $\texttt{hash}_1$ mais la probabilité de collisions au sein du HashSet est plus faible chez $\texttt{hash}_1$.

Il est possible de modifier le HashSet pour n'y stoquer que les hash des positions, réduisant ainsi la complexité de l'opération « `next` $\in$ `closed` » à $\mathcal{O}(1)$, mais introduisant une probabilité que l'opération donne un faux positif, comme dans la figure suivante :

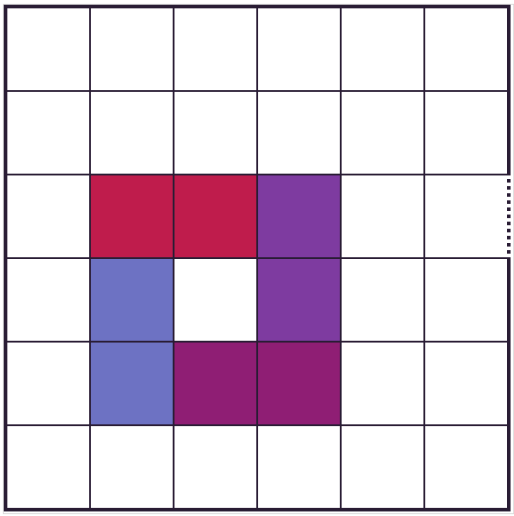


FIGURE 3 – Position où `next` $\in$ `closed` peut donner un faux positif

4.3 Optimisation de l'allocation de la liste de mouvements

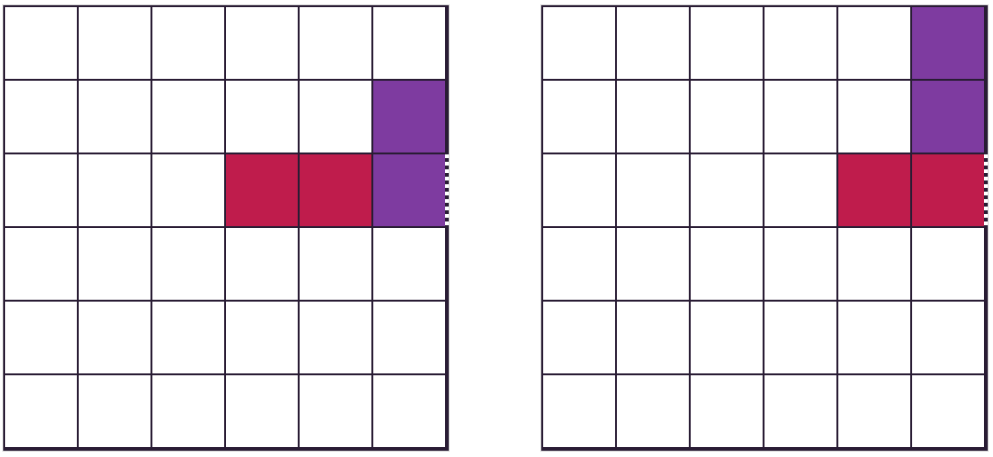
Vu que chaque état dans la liste `open` partage la majorité des mouvements avec d'autres états de cette liste, il est possible de réduire le nombre d'allocations en utilisant du reference counting et en ne stockant que le mouvement fait :

```
struct rc_node {
    int count; // Incrementation a chaque copie du pointeur et
       decrementation a chaque destruction
    rc_linkedlist* valeur; // free() lorsque count = 0
};

struct rc_linkedlist {
    mouvement valeur;
    rc_node* next;
};
```

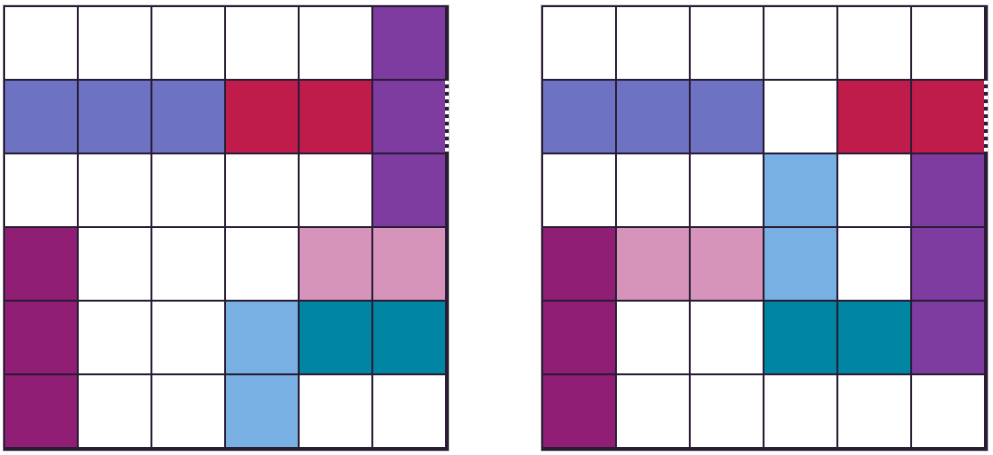
5 Jeux d'essais

Voici quelques exemples de positions initiales, ainsi que la solution proposée par le programme :



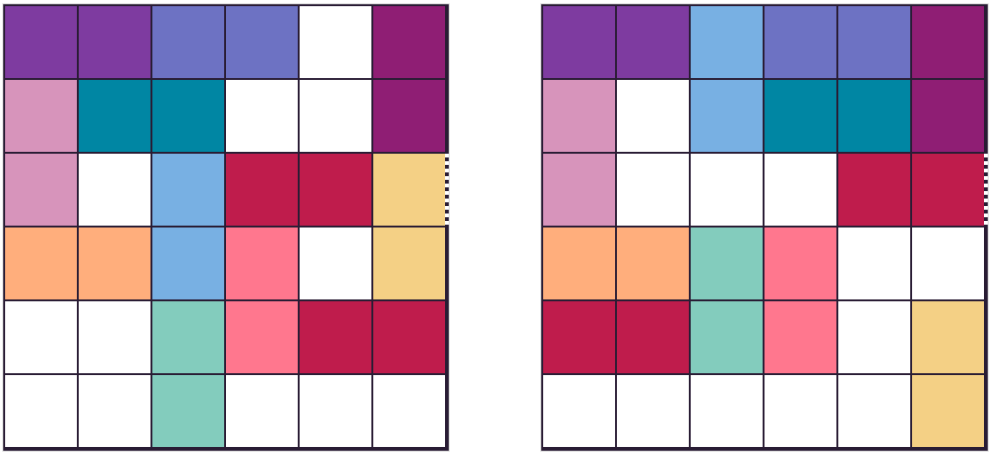
(a) Position initiale (b) Position finale

FIGURE 4 – Exemple "trivial" : seuls deux mouvements sont nécessaires. 5 noeuds sont analysés en 2.3 ± 0.07 μs



(a) Position initiale (b) Position finale

FIGURE 5 – Exemple "simple" : 5 mouvements sont nécessaires.
46 noeuds sont analysés en 39.5 ± 0.2 μs



(a) Position initiale (b) Position finale

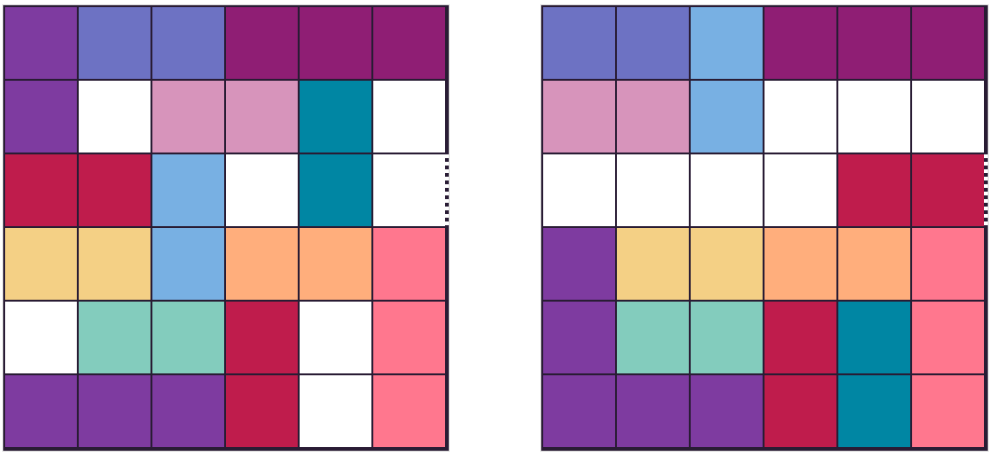
FIGURE 6 – Exemple "complexe" : 21 mouvements sont nécessaires (donnés dans la Figure 7. 3396 noeuds sont analysés en 4300 ± 100 μs

```
1122 3
455  3
4 6007
8869 7
  a9bb
  a

5: 1,1 -> 3,1
2: 2,0 -> 3,0
6: 2,2 -> 2,0
0: 3,2 -> 1,2
9: 3,3 -> 3,2
11: 4,4 -> 3,4
7: 5,2 -> 5,4
3: 5,0 -> 5,2
5: 3,1 -> 4,1
2: 3,0 -> 4,0
9: 3,2 -> 3,0
0: 1,2 -> 3,2
10: 2,4 -> 2,2
11: 3,4 -> 0,4
10: 2,2 -> 2,3
0: 3,2 -> 1,2
9: 3,0 -> 3,3
5: 4,1 -> 3,1
2: 4,0 -> 3,0
3: 5,2 -> 5,0
0: 1,2 -> 4,2

116223
4 6553
4   00
88a9
bba9 7
     7
```

FIGURE 7 – Série optimale de mouvements nécessaires pour résoudre l'exemple "complexe"



(a) Position initiale (b) Position finale

FIGURE 8 – Exemple "expert" : 44 mouvements sont nécessaires (donnés dans la Figure 2). 3396 noeuds sont analysés en 3610 $\pm$ 200 μs

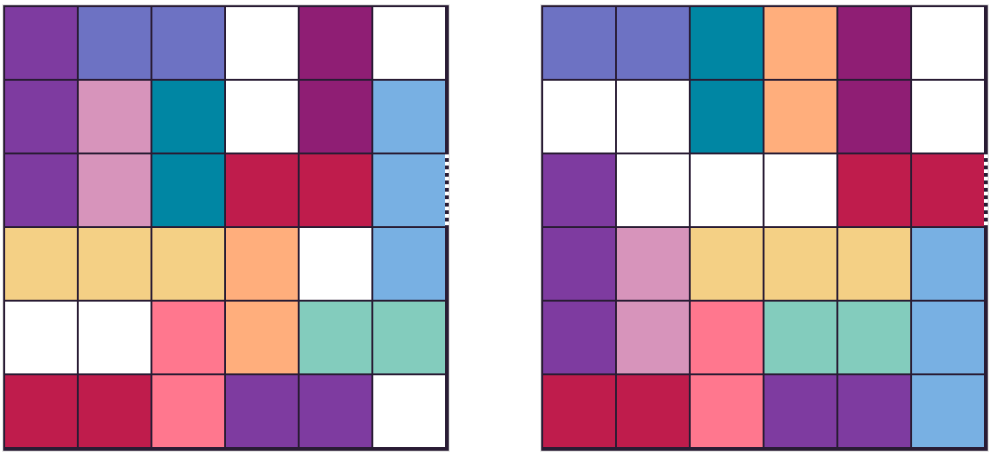
Listing 2 – Série optimale de mouvements nécessaires pour résoudre l'exemple "expert"

```
122333
1 445
006 5
776889
 aab 9
cccb 9

10: 1,4 -> 0,4
6: 2,2 -> 2,3
4: 2,1 -> 1,1
0: 0,2 -> 1,2
1: 0,0 -> 0,1
2: 1,0 -> 0,0
3: 3,0 -> 2,0
9: 5,3 -> 5,0
8: 3,3 -> 4,3
11: 3,4 -> 3,1
12: 0,5 -> 3,5
8: 4,3 -> 3,3
9: 5,0 -> 5,1
3: 2,0 -> 3,0
```

```
2: 0,0 -> 1,0
1: 0,1 -> 0,0
0: 1,2 -> 0,2
6: 2,3 -> 2,2
10: 0,4 -> 3,4
6: 2,2 -> 2,4
7: 0,3 -> 1,3
0: 0,2 -> 1,2
1: 0,0 -> 0,4
7: 1,3 -> 0,3
4: 1,1 -> 0,1
2: 1,0 -> 0,0
3: 3,0 -> 2,0
9: 5,1 -> 5,0
0: 1,2 -> 0,2
6: 2,4 -> 2,1
12: 3,5 -> 1,5
10: 3,4 -> 1,4
8: 3,3 -> 2,3
5: 4,1 -> 4,4
8: 2,3 -> 4,3
7: 0,3 -> 1,3
1: 0,4 -> 0,3
12: 1,5 -> 0,5
11: 3,1 -> 3,4
8: 4,3 -> 3,3
9: 5,0 -> 5,3
3: 2,0 -> 3,0
6: 2,1 -> 2,0
0: 0,2 -> 4,2

226333
446
   00
177889
1aab59
cccb59
```



(a) Position initiale (b) Position finale

FIGURE 9 – Exemple "maximal" : 51 mouvements sont nécessaires, position découverte par Michael Fogleman. 2810 noeuds sont analysés en 3720 ± 300 μs

6 Améliorations possibles

L'interface graphique du projet pourrait grandement bénéficier de fonctionnalités en plus : il aurait été utile de pouvoir déplacer ou placer des voitures directement dans la fenêtre, au lieu de devoir modifier un fichier texte.

Il aurait été également intéressant de permettre l'inter-opérabilité entre les deux implémentations faites, ou de directement programmer l'interface graphique en Rust, afin de bénéficier de la vitesse de calcul de ce dernier. L'implémentation Python prend le plus souvent moins d'une seconde pour résoudre une position, donc le temps d'attente n'est pas insoutenable.

Avec un algorithme qui prend un maximum de 5ms à résoudre une position, il aurait été facile de faire tourner celui-ci sur toutes les positions possibles, afin de voir lesquelles peuvent se transposer vers d'autres et lesquelles peuvent être résolues ou non. C'est une tâche qui est évidemment possible, après avoir lu l'article de Michael Fogleman, et des problèmes intéressants de synchronisation, de parallélisation et de génération de positions s'y présentent.

Le jeu de Rush Hour peut lui-même être étendu à des pièces non-rectangulaires : on peut imaginer un Rush Hour avec uniquement des tetrominos ou des pentaminos, par exemple. Notre solution se limite à des pièces rectangulaires, mais celles-cis peuvent être de taille arbitraire et le terrain peut être lui aussi de taille arbitraire.

7 Conclusion et retour d'expérience

7.1 Adrien Burgun

J'ai été personellement satisfait du projet, l'implémentation dans les deux langages a été sans grandes difficultés. J'ai trouvé l'optimisation de la solution en Rust intéressante, et j'ai pu y appliquer les techniques d'analyse de code apprises sur d'autres projets.

En revanche, j'ai eu beaucoup de mal à utiliser la librairie TKinter : la majorité de sa documentation se trouvait sur le site effbot.org, qui n'est désormais plus en ligne. J'ai entendu que son propriétaire, Fredrik Lundh, est

décédé fin 2021 - je ne saurais lier l'état du site à son décès, mais j'en suis embêté et triste.

7.2 Erwann Le Guilly

7.3 Jean-Maël Le Grand

7.4 Mohamed Jani

8 Annexe : listing du code

8.1 Algorithme BFS

Listing 3 – Algorithme BFS dans le fichier solve.py

```
def bfs(initial):
    open = [(initial, [])] # Liste des positions  explorer ainsi que
        du chemin parcouru jusqu' celles-cis
    closed = set() # Liste des positions dj explores
    # Limitation de Python: on ne peut pas stocker d'objets dans un
        set(), donc on stocke uniquement le hash.
    # Ceci peut mener  des faux positifs, mais aucun faux positif
        faussant la solution n'a t trouv
    closed.add(hash(initial))
    n_scanned = 0 # Compteur des noeuds explors

    while len(open) > 0: # Tant qu'il y a des positions  explorer
        n_scanned += 1
        (current, path) = open.pop(0)
        for (child, move) in current.next_states(): # Pour toutes les
            positions rsultante d'un mouvement...
            hashed = hash(child)
            if not (hashed in closed): # Si la position n'a pas dj
                te  explore
                if child.solved(): # Si la position est la solution,
                    retourner la suite de coups
                    print(n_scanned, "nodes scanned")
                    return path + [move]
                closed.add(hashed) # Sinon, l'ajouter  closed
                open.append((child, path + [move])) # et l'ajouter
                    open

    return None
```

8.2 Fonctions next_states et move

Listing 4 – Fonction next_states et move dans board.py

```
# Returns an instance of Board where the ‘index‘-th car was moved to
    ‘(x, y)‘
def move(self, index, x, y):
    res = Board(self.width, self.height) # On cre une nouvelle
        instance de Board
    res.exit_y = self.exit_y
    for n in range(0, len(self.cars)): # Les voitures sont clones,
        if n == index: #  l’exception de la voiture ‘index‘, qui
            reoit une nouvelle position
            res.cars.append((x, y, self.cars[n][2], self.cars[n][3]))
        else:
            res.cars.append(self.cars[n])

    return res

# Returns a list of child states and the movements to get to these
def next_states(self):
    res = [] # La liste  retourner, contenant des paires de positions
        et de mouvements
    for n in range(0, len(self.cars)): # Pour chaque voiture dans
        self.cars
        car = self.cars[n]
        if car[3]: # Si la voiture est horizontale
            for x in range(0, car[0]): # Pour x dans [0; car.x[
                if self.get_car(car[0] - x - 1, car[1]) == None: # Si
                    l’espace est vide:
                    # Ajouter la nouvelle position ainsi que le
                        mouvement fait  res
                    res.append((self.move(n, car[0] - x - 1, car[1]),
                        (n, car[0] - x - 1, car[1])))
                else: # Sinon, s’arrter
                    break
            for x in range(car[0], self.width - car[2]): # Pour x dans
                [car.x; width - car.length[
                if self.get_car(x + car[2], car[1]) == None:
                    res.append((self.move(n, x + 1, car[1]), (n, x + 1,
                        car[1])))
                else:
                    break
        else: # Si la voiture est verticale
            for y in range(0, car[1]): # Pour y dans [0; car.y[
                if self.get_car(car[0], car[1] - y - 1) == None:
```

```
                res.append((self.move(n, car[0], car[1] - y - 1),
                    (n, car[0], car[1] - y - 1)))
            else:
                break
        for y in range(car[1], self.height - car[2]): # Pour y
            dans [car.y; height - car.length[
            if self.get_car(car[0], y + car[2]) == None:
                res.append((self.move(n, car[0], y + 1), (n,
                    car[0], y + 1)))
            else:
                break
    return res
```