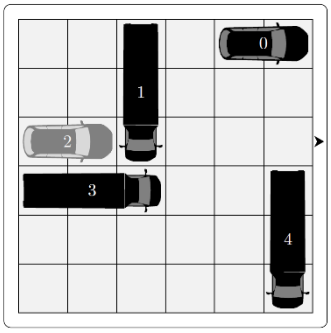
TD : HACHAGE – RUSH HOUR

(TD inspiré de Centrale-Supélec MPI 2025)

Le problème posé dans le sujet était d'écrire un programme permettant de résoudre le jeu Rush Hour. Le jeu de plateau Rush Hour simule un embouteillage dans un parking à l'heure de pointe. Dans ce jeu à un joueur, on doit extraire le véhicule gris d'une grille dans laquelle plusieurs autres véhicules noirs bloquent la sortie. Il est possible de déplacer chaque véhicule, d'une ou plusieurs cases, vers l'avant ou l'arrière. Les véhicules sont soit des voitures, soit des camions. Les voitures occupent 2 cases tandis que les camions occupent 3 cases. La sortie est repérée par une flèche qui se situe sur le côté droit de la grille. La figure ci-contre illustre une position de départ du jeu.



Pour résoudre ce casse-tête, on commence par déplacer la voiture 0 d'une case en arrière. On se retrouve alors dans la configuration de la figure 2.a. Puis on déplace le camion 4 de 3 cases en arrière, le camion 3 de 3 cases en avant et le camion 1 de 3 cases en avant. On se retrouve alors dans la configuration de la figure 2.b qui nous permet de déplacer la voiture grise numérotée 2 de 3 cases vers l'avant et nous retrouver dans la configuration de la figure 2.c.

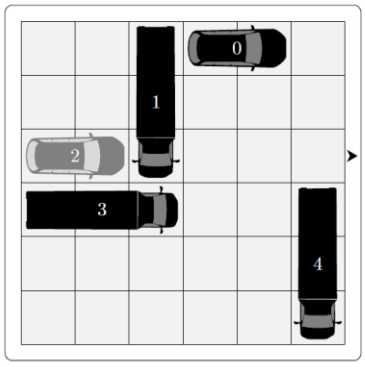


Figure 2.a

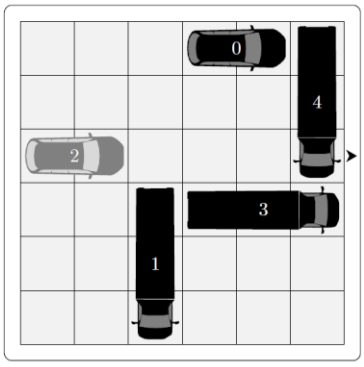


Figure 2.b

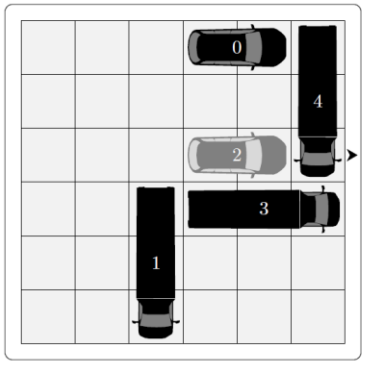


Figure 2.c

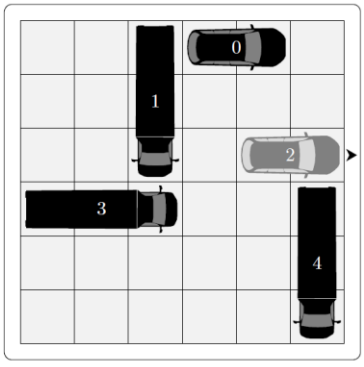


Figure 2.d

On déplace alors le camion 1 de 3 cases vers l'arrière, puis le camion 3 de 3 cases vers l'arrière et le camion 4 de 3 cases vers l'avant. On avance enfin la voiture grise numérotée 2 d'une case. Elle se retrouve devant la sortie indiquée par la flèche sur la droite de la grille. Cette configuration gagnante est indiquée à la figure 2.d.

I) RETOUR SUR LA PREMIÈRE PARTIE DU SUJET : MODÉLISATION DU JEU

La première partie du sujet portait sur l'implémentation de :

- la modélisation d'un état du jeu (ensemble de véhicules sur une grille),
- des actions permettant de :
  - créer/libérer des états,
  - tester leur égalité,
  - construire le plateau à partir d'un état,
  - lister et représenter les déplacements possibles.

Vous utiliserez les fichiers « TD6. Rush_Hour_Sujet.py » et son fichier bibliothèque associé « Lib_Rush_Hour.py » (les deux fichiers doivent être dans le même répertoire).

Les représentations utilisées dans ce TD sont les suivantes :

- La position de chaque **véhicule** sur la grille est représentée sous forme d'un dictionnaire :

```
{
    "i": ...,          # ligne de la case la plus en haut à gauche
    "j": ...,          # colonne de cette case
    "longueur": 2 (voiture) ou 3 (camion),
    "direction": "H" (Horizontal) ou "V" (Vertical)
}
```

- Un **état** est un dictionnaire qui définit la configuration des véhicules sur le plateau :

```
{
    "vehicules": [ ... ],   # liste de véhicules
    "distance": 0           # nombre de coups depuis l'état initial
}
```

- Un **déplacement** de la voiture d'indice `k` de pas cases dans le sens des `i` (ou des `j`) croissant est représenté par un dictionnaire :

```
{
    "k": indice_du_véhicule, (k=0 pour la voiture grise)
    "pas": entier (positif ou négatif)
}
```

  Par exemple, dans la solution du casse-tête donné en introduction, le premier mouvement de la voiture 0 d'une case vers la gauche aura une valeur de k = 0 et de pas = -1. Le second mouvement du camion 4 de 3 cases vers le haut aura une valeur de k = 4 et de pas = -3.

- Un **plateau** est une liste de listes `plateau[i][j]` contenant `-1` (vide) ou l'indice d'un véhicule (1, 2, 3…).

Les fonctions implémentées dans la bibliothèque « Lib_Rush_Hour.py » sont les suivantes :

- `nouvel_etat()` : crée et retourne un état vide.
- `ajoute_vehicule(etat,i,j,longueur,direction)` : ajoute un nouveau véhicule à un état.
- `affiche_etat(etat,indice_gris=0)` : affiche l'état sous forme de grille texte ("G" pour la voiture grise d'indice k=0, "." pour les cases vides, l'indice du véhicule – "1", "2", "3", … – pour les autres véhicules).

- `liste_deplacements_valides(etat)` : calcule tous les déplacements possibles à partir d'un état. Retourne une liste de déplacements, chaque déplacement étant un dictionnaire.
- `applique_deplacement(etat, dep)` : applique un déplacement à un état, retourne le nouvel état qui correspond à l'état après le déplacement.
- `etat_vers_plateau(etat)` : transforme un état en un plateau de taille MxM (avec ici M = 6).

1. Écrire la fonction `initialise()` qui crée un nouvel état et ajoute les véhicules afin de construire le plateau donné en introduction.

```
Vérifier : >>> initialise()
           {'vehicules': [{'i': 2, 'j': 0, 'longueur': 2,
           'direction': 'H'}, {'i': 0, 'j': 4, 'longueur': 2,
           'direction': 'H'}, {'i': 0, 'j': 2, 'longueur': 3,
           'direction': 'V'}, {'i': 3, 'j': 0, 'longueur': 3,
           'direction': 'H'}, {'i': 3, 'j': 5, 'longueur': 3,
           'direction': 'V'}], 'distance': 0}
           >>> affiche_etat(etat)

              0 1 2 3 4 5
             +------------+
           0 | . . 2 . 1 1 |
           1 | . . 2 . . . |
           2 | G G 2 . . . |--> sortie
           3 | 3 3 3 . . 4 |
           4 | . . . . . 4 |
           5 | . . . . . 4 |
             +------------+
```

2. Lister les déplacements valides à partir de cet état initial ;

3. Appliquer le premier déplacement valide retourné et afficher le nouvel état obtenu après ce déplacement ;

4. Afficher le plateau correspondant à cet état.

```
Vérifier :     >>> affiche_etat(etat)

                 0 1 2 3 4 5
               +------------+
             0 | . . 2 1 1 . |
             1 | . . 2 . . . |
             2 | G G 2 . . . |--> sortie
             3 | 3 3 3 . . 4 |
             4 | . . . . . 4 |
             5 | . . . . . 4 |
               +------------+
```

II) TABLE DE HACHAGE

Nous cherchons maintenant à résoudre un problème lié à l'exploration des configurations du jeu Rush Hour : comment éviter de revisiter plusieurs fois les mêmes états sans effectuer des comparaisons longues et coûteuses ?

Pour cela, nous allons associer à chaque état un nombre entier appelé valeur de hachage, obtenu à partir de la disposition des véhicules sur le plateau. Cette valeur servira de « signature » de l'état et nous permettra de le ranger rapidement dans une table de hachage.

Nous construirons ensuite une structure de données capable de mémoriser les états déjà rencontrés et de répondre efficacement à la question : « cet état a-t-il déjà été vu ? ».

Nous allons implémenter une structure d'ensemble à l'aide d'une table de hachage en adressage ouvert. Les éléments de cet ensemble sont de type `etat`. Nous allons commencer par définir une fonction de hachage. Pour tout état e, on définit

$$\text{hash}(e) = \sum_{\substack{0 \le i < M \\ 0 \le j < M}} d_{i,j} 3^{iM+j}$$

… où pour tout i, j $\in$ [0,M[:

$$d_{i,j} = \begin{cases} 0 & \text{si la case (i,j) est inocupée} \\ 1 & \text{si la case (i,j) est occupée par un véhicule horizontal} \\ 2 & \text{si la case (i,j) est occupée par un véhicule vertical} \end{cases}$$

Rappel : on dispose de la fonction `etat_vers_plateau(etat)` qui étant donné un état, renvoie un plateau `p` dont les éléments `p[i][j]` sont égaux à $d_{i,j}$.

1. Écrire une fonction `e_hash(etat)` prenant en entrée un état e et renvoyant hash(e).

```
Vérifier :        >>> e_hash(etat)
                  175454050611577982
```

Deux états sont égaux s'ils ont le même nombre de véhicules n et que, pour tout indice k $\in$ [0, n[, les véhicules d'indice k ont les mêmes caractéristiques de position et caractéristiques (longueur, direction). Deux états peuvent être égaux en ayant des valeurs différentes associées au champ `distance`.

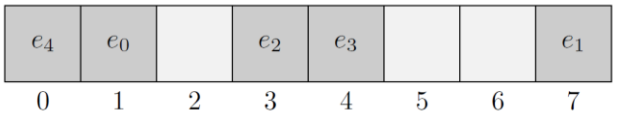
2. Montrer que si e_1 et e_2 sont deux états accessibles à partir d'un même état initial e et que hash(e_1) = hash(e_2), alors $e_1 = e_2$.

Pour stocker les états dans notre table de hachage, nous allons utiliser un tableau possédant un nombre de cases appelé capacité de la table de hachage. Cette capacité sera de la forme 2^p. La taille de la table de hachage est le nombre d'états qu'elle contient.

Lorsqu'on souhaite ajouter un état e dans notre table de hachage, on commencera par calculer :

$$h = \text{hash}(e) \bmod 2^p$$

Si la case d'indice h est vide, on placera e dans cette case. Sinon, on cherchera la première case vide en partant de l'indice h et en avançant de un en un de manière circulaire dans le tableau, jusqu'à trouver une case vide. Par exemple, si l'on part d'une table de hachage vide de capacité 8, et qu'on ajoute successivement les états e_0, e_1, e_2, e_3 et e_4 ayant des hash respectifs de $h_0 = 1$, $h_1 = 7$, $h_2 = 3$, $h_3 = 3$ et $h_4 = 7$, on obtiendra la table de hachage suivante :



Lorsque la table de hachage est trop peuplée, on double sa capacité et on réinsère les éléments de l'ancienne table de hachage dans la nouvelle.

On utilisera le dictionnaire suivant pour représenter notre table de hachage :

```
{
    "lst_etats": [ ... ],   # liste des états dans la table
    "taille": entier        # taille de la table de hachage
    "capacite": entier      # capacité de la table de hachage
}
```

La liste `lst_etats` dans le dictionnaire contient les différents états présents dans la table de hachage. Nous utiliserons la valeur `-1` pour représenter les cases vides de la table.

3. Créer la fonction `t_nouveau()` qui crée et retourne une table vide ayant une capacité égale à 1.

   Tester :
   ```
   >>> t_nouveau()
      {'lst_etats': [-1], 'taille': 0, 'capacite': 1}
   ```

4. Créer la fonction `t_contient(t, e)` déterminant si l'état e est présent dans la table de hachage t. Vous n'utiliserez pas de recherche de type `in list` en Python. L'objectif est de faire une recherche très rapide à l'aide de la table de hachage. N'oubliez pas le chaînage linéaire.

   Tester :
   ```
   >>> etat = initialise()
   >>> tbl=t_nouveau()
   >>> t_contient(tbl,etat)
   False
   >>> tbl['lst_etats'][0]=etat
   >>> t_contient(tbl,etat)
   True
   >>> etat = applique_deplacement(etat,
   liste_deplacements_valides(etat)[0])
   >>> t_contient(tbl,etat)
   False
   ```

5. Créer la fonction `t_augmente_capacite(t)` qui prend en entrée une table de hachage t de capacité $c \geq 1$ et qui augmente sa capacité à 2c.
   Attention : si la capacité change, l'indice où enregistrer les états va également changer et donc les états ne sont plus enregistrés au bon endroit… Il faut donc les recopier au bon endroit après avoir changé la capacité.

   Tester :
```
>>> tbl=t_nouveau()
>>> tbl['lst_etats'][0]=etat
>>> tbl['taille']=1
>>> t_augmente_capacite(tbl)
>>> tbl
{'lst_etats': [{'vehicules': [{'i': 2, 'j': 0,
'longueur': 2, 'direction': 'H'}, {'i': 0, 'j': 4,
'longueur': 2, 'direction': 'H'}, {'i': 0, 'j': 2,
'longueur': 3, 'direction': 'V'}, {'i': 3, 'j': 0,
'longueur': 3, 'direction': 'H'}, {'i': 3, 'j': 5,
'longueur': 3, 'direction': 'V'}], 'distance': 0}, -1],
'taille': 1, 'capacite': 2}
```

6. Créer la fonction `t_ajoute(t, e)` qui ajoute un état e à la table de hachage. Si m est la taille de la table et c sa capacité, on fera attention à ce que l'on ait toujours $m \leq 2c/3$.

   Vérifier :
```
>>> etat = initialise()
>>> tbl=t_nouveau()
>>> t_ajoute(tbl,etat)
>>> etat = applique_deplacement(etat,
liste_deplacements_valides(etat)[0])
>>> t_ajoute(tbl,etat)
>>> tbl
{'lst_etats': [{'vehicules': [{'i': 2, 'j': 0, 'longueur': 2,
'direction': 'H'}, {'i': 0, 'j': 4, 'longueur': 2, 'direction': 'H'},
{'i': 0, 'j': 2, 'longueur': 3, 'direction': 'V'}, {'i': 3, 'j': 0,
'longueur': 3, 'direction': 'H'}, {'i': 3, 'j': 5, 'longueur': 3,
'direction': 'V'}], 'distance': 0}, {'vehicules': [{'i': 2, 'j': 0,
'longueur': 2, 'direction': 'H'}, {'i': 0, 'j': 3, 'longueur': 2,
'direction': 'H'}, {'i': 0, 'j': 2, 'longueur': 3, 'direction': 'V'},
{'i': 3, 'j': 0, 'longueur': 3, 'direction': 'H'}, {'i': 3, 'j': 5,
'longueur': 3, 'direction': 'V'}], 'distance': 1}], 'taille': 2,
'capacite': 2}
```

7. Expliquer en quoi ce serait une mauvaise idée de remplacer le 3^{iM+j} par 4^{iM+j} dans la définition de hash(e).

III) RECHERCHE D'UNE SOLUTION OPTIMALE AVEC COÛT CONSTANT

On se place dans le cas où chaque déplacement engendre un coût de 1. Ainsi, la valeur 'distance' de l'état sera incrémentée de 1 à chaque déplacement.

Dans l'exemple ci-dessous, à partir de l'état initial, on demande un déplacement valide de 2 pas, mais on peut observer que la distance résultante est de 1 :

```
>>> etat = initialise()
>>> etat['distance']
0
>>> liste_deplacements_valides(etat)
[{'k': 1, 'pas': -1}, {'k': 3, 'pas': 1}, {'k': 3, 'pas': 2}, {'k': 4, 'pas': -1}, {'k': 4, 'pas': -2}]
>>> etat = applique_deplacement(etat, liste_deplacements_valides(etat)[2])
>>> etat['distance']
1
```

On modélise chaque état du jeu comme un sommet d'un graphe, et chaque mouvement possible comme une arête. Comme tous les mouvements ont le même coût et que le nombre de configurations est fini, on peut appliquer un BFS à partir de l'état initial : le premier état gagnant rencontré donnera un nombre minimal de coups.

III.1. Rappel sur l'algorithme BFS sur un graphe statique (connu à l'avance)

Dans l'algorithme BFS appliqué à un graphe statique, on connaît à l'avance tous les sommets et toutes les arêtes (via une matrice, une paire de listes ou un dictionnaire). On utilise une file pour stocker les sommets visités et une liste pour enregistrer s'ils ont été visités ou non (ou on ajoute cette information au dictionnaire). Chaque indice de cette liste correspond à un sommet du graphe : `visites[0 … n-1] = True / False` (avec un graphe à n sommets)

Dans notre cas, les sommets étant les états du jeu, l'algorithme BFS « statique » (si on connaissait à l'avance tous les états possibles du jeu) pour trouver la distance minimale dans notre cas serait le suivant :

Algorithme BFS sur un graphe statique

```
Créer une file et y enfiler l'état initial puis le marquer comme visité.
Tant que la file n'est pas vide :
    Défiler un état
    Si l'état est gagnant :
        Retourner la distance (BFS terminé)
    Sinon :
        Pour chaque état voisin connu de cet état :
            Si cet état voisin n'a pas été visité :
                Marquer cet état voisin comme visité
                Ajouter cet état voisin à la file
Si la file est vide sans avoir trouvé d'état gagnant, retourner -1
```

III.2. Recherche de la distance minimale avec BFS sur le graphe dynamique

Dans notre cas, on ne peut pas connaître tous les états possibles à l'avance (il y en a énormément) et on ne peut donc pas stocker le graphe complet en mémoire. Il va donc être construit de manière dynamique.

On ne peut donc pas indexer une liste des états visités, et il nous faut une structure pour savoir si tel ou tel état a déjà été visité. C'est là que la table de hachage va nous être utile : elle servira à mémoriser tous les états déjà rencontrés pour ne jamais les revisiter. La file quant à elle sera utilisée pour gérer l'ordre dans lequel on explore les configurations (0 coup, puis 1 coup, puis 2, etc.).

Avec la table de hachage, on teste « déjà vu ? » en temps quasi constant, et chaque configuration n'est traitée qu'une seule fois, ce qui rend le BFS faisable en pratique.

Utiliser une table de hachage plutôt qu'une liste pour stocker les états déjà vus permet de tester en temps quasi constant si un état a déjà été vu (au lieu de parcourir une longue liste en O(n)).

En Python, pour une file (queue), on utilise en général `collections.deque` :

```
from collections import deque

file = deque()          # Création d'une nouvelle file
file.append(objet)      # Ajout d'un objet dans la file
objet = file.popleft()  # Sortir un objet de la file
```

L'algorithme à implémenter est donc le suivant :

Algorithme BFS avec table de hachage

```
Construire l'état initial (distance = 0)
Créer une file et y enfiler l'état initial
Créer une table de hachage et ajouter l'état initial à la table

Tant que la file n'est pas vide :
    Défiler un état (le prochain à explorer)
    Si l'état est gagnant :
        Retourner la distance (BFS terminé)
    Sinon :
        Générer les déplacements valides à partir de l'état courant
        Pour chaque déplacement valide :
            Appliquer le déplacement valide sur l'état courant
            Si l'état retourné n'est pas dans la table de hachage :
                Ajouter l'état retourné à la table de hachage
                Enfiler l'état retourné dans la file
Si la file devient vide sans avoir trouvé d'état gagnant, renvoyer -1.
```

On se place dans le cas où la voiture grise est la seule voiture horizontale sur la ligne de sortie S, et l'état est gagnant quand cette voiture touche le bord droit de la grille. Les constantes M et S sont déjà définies dans la bibliothèque `Lib_Rush_Hour.py`.

1. Écrire la fonction `e_gagnant(e)` qui retourne `True` si l'état `e` est gagnant, et retourne `False` dans le cas contraire.

   Tester :
   ```
   >>> etat=initialise()
   >>> e_gagnant(etat)
   False
   >>> e_gagnant(etat_final_correction)
   True
   ```

2. Écrire la fonction `BFS(e)` qui, à partir d'un état initial `e`, renvoie la distance minimale pour arriver à la sortie.

   Tester :
   ```
   >>> etat=initialise()
   >>> distance_min, etat_final = BFS(etat)
   >>> distance_min
   9
   >>> affiche_etat(etat_final)
      0 1 2 3 4 5
     +------------+
   0 | . . 2 1 1 . |
   1 | . . 2 . . . |
   2 | . . 2 . G G |--> sortie
   3 | . . 3 3 3 4 |
   4 | . . . . . 4 |
   5 | . . . . . 4 |
     +------------+
   ```

III.3. Extrapolation du chemin optimal avec BFS

On voudrait maintenant récupérer les déplacements optimaux à effectuer pour sortir le véhicule gris.

Pour cela, il suffit d'ajouter un dictionnaire qui enregistre, pour chaque nouvel état découvert, son état parent (d'où il vient) et par quel déplacement le nouvel état a été obtenu.

Ensuite, on peut remonter le chemin à l'envers jusqu'à l'état initial. Enfin, il faut ensuite renverser la liste.

1. Modifier votre fonction `BFS(e)` afin qu'elle enregistre dans un dictionnaire les informations nécessaires à la reconstruction du chemin optimum (hash des états et déplacements effectués) et qu'elle retourne la distance minimale, l'état final et le dictionnaire de l'historique. On pourra par exemple utiliser un dictionnaire de la forme :

   ```
   historique = {
       "etat_decouvert": [...],   # liste des hash des états découverts
       "etat_parent": [...]       # liste des hash des états parents
       "deplacement": [...]       # liste des déplacements effectués
        }
   ```

2. Écrire la fonction `reconstruction(historique, etat_initial, etat_final)` permettant de renvoyer la liste des déplacements optimums.

```
>>> etat=etat_initial

>>> affiche_etat(etat_initial)
   0 1 2 3 4 5
  +------------+
0 | . . 2 . 1 1 |
1 | . . 2 . . . |
2 | G G 2 . . . |--> sortie
3 | 3 3 3 . . 4 |
4 | . . . . . 4 |
5 | . . . . . 4 |
  +------------+

>>> liste_dep=reconstruction(chemin,etat_initial,etat_final)
>>> etat = applique_deplacement(etat,liste_dep[0])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 . |
1 | . . 2 . . . |
2 | G G 2 . . . |--> sortie
3 | 3 3 3 . . 4 |
4 | . . . . . 4 |
5 | . . . . . 4 |
  +------------+

>>> etat = applique_deplacement(etat,liste_dep[1])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 4 |
1 | . . 2 . . 4 |
2 | G G 2 . . 4 |--> sortie
3 | 3 3 3 . . . |
4 | . . . . . . |
5 | . . . . . . |
  +------------+
```

```
>>> etat = applique_deplacement(etat,liste_dep[2])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 4 |
1 | . . 2 . . 4 |
2 | G G 2 . . 4 |--> sortie
3 | . . . 3 3 3 |
4 | . . . . . . |
5 | . . . . . . |
  +------------+

>>> etat = applique_deplacement(etat,liste_dep[3])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . . 1 1 4 |
1 | . . . . . 4 |
2 | G G . . . 4 |--> sortie
3 | . . 2 3 3 3 |
4 | . . 2 . . . |
5 | . . 2 . . . |
  +------------+

>>> etat = applique_deplacement(etat,liste_dep[4])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . . 1 1 4 |
1 | . . . . . 4 |
2 | . . . G G 4 |--> sortie
3 | . . 2 3 3 3 |
4 | . . 2 . . . |
5 | . . 2 . . . |
  +------------+
```

```
>>> etat = applique_deplacement(etat,liste_dep[5])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 4 |
1 | . . 2 . . 4 |
2 | . . 2 G G 4 |--> sortie
3 | . . . 3 3 3 |
4 | . . . . . . |
5 | . . . . . . |
  +------------+

>>> etat = applique_deplacement(etat,liste_dep[6])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 4 |
1 | . . 2 . . 4 |
2 | . . 2 G G 4 |--> sortie
3 | . . 3 3 3 . |
4 | . . . . . . |
5 | . . . . . . |
  +------------+

>>> etat = applique_deplacement(etat,liste_dep[7])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 . |
1 | . . 2 . . . |
2 | . . 2 G G . |--> sortie
3 | . . 3 3 3 4 |
4 | . . . . . 4 |
5 | . . . . . 4 |
  +------------+

>>> etat = applique_deplacement(etat,liste_dep[8])

>>> affiche_etat(etat)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 . |
1 | . . 2 . . . |
2 | . . 2 . G G |--> sortie
3 | . . 3 3 3 4 |
4 | . . . . . 4 |
5 | . . . . . 4 |
  +------------+
```

IV) RECHERCHE D'UNE SOLUTION OPTIMALE AVEC COÛT VARIABLE

On se place maintenant dans le cas où le coût d'un déplacement est égal au nombre de cases parcourues (par exemple, « +3 cases parcourues » engendre un coût de 3, « -2 cases parcourues » engendre un coût de 2 – on ne tient pas compte du signe).

Certaines solutions peuvent avoir plus de coups joués mais moins de cases déplacées, ou l'inverse. On cherche alors le chemin de coût total minimal, pas forcément celui avec le moins de coups joués.

On utilise maintenant la fonction `applique_deplacement_c_var()` qui comptabilise les coûts variables :

```
>>> etat = initialise()
>>> etat['distance']
0
>>> liste_deplacements_valides(etat)
[{'k': 1, 'pas': -1}, {'k': 3, 'pas': 1}, {'k': 3, 'pas': 2}, {'k': 4, 'pas': -1}, {'k': 4, 'pas': -2}]
>>> etat = applique_deplacement_c_var(etat, liste_deplacements_valides(etat)[4])
>>> etat['distance']
2
```

Le graphe des états devient pondéré (avec des poids positifs). Dans ce cas, l'algorithme BFS n'est plus correct : il faut utiliser Dijkstra.

IV.1. Rappel sur l'algorithme Dijkstra sur un graphe statique (connu à l'avance)

Dans l'algorithme Dijkstra appliqué à un graphe statique, on connaît à l'avance tous les sommets et toutes les arêtes (via une matrice, une paire de listes ou un dictionnaire). On utilise une liste pour enregistrer la meilleure distance connue depuis la source et une liste pour enregistrer les sommets visités. Chaque indice de ces listes correspond à un sommet du graphe : `dist[0..n-1]` et `visite[0..n-1]`, où n est le nombre de sommets du graphe.

Dans notre cas, l'algorithme de Dijkstra « statique » (si on connaissait à l'avance tous les états possibles du jeu) pour trouver la distance minimale serait le suivant :

Algorithme de Dijkstra sur un graphe statique

```
Initialiser toutes les distances à l'infini, sauf celle de l'état initial, à 0

Tant qu'il existe un état non visité atteignable :
    Récupérer l'état u qui a la plus petite distance parmi tous les états non visités
    Si l'état est gagnant :
        Retourner la distance (Dijkstra terminé)
    Sinon :
        Marquer cet état comme visité
        Pour tous les voisins v connus de l'état u :
            Si dist[u] + ℓ(u, v) < dist[v] :
                Mettre à jour la distance de v : dist[v] =  dist[u] + ℓ(u, v)
Si tous les états ont été visités sans trouver d'état gagnant, retourner -1
```

IV.2. Recherche de la distance minimale avec Dijkstra sur le graphe dynamique

De nouveau, dans notre cas on ne peut pas connaître tous les états possibles à l'avance et on ne peut donc pas stocker le graphe complet en mémoire. Il va donc être construit de manière dynamique.

On ne peut donc pas indexer une liste des distances et des états visités et on va générer les voisins à la volée à partir d'un état en appliquant les déplacements valides. Il nous faut une structure pour savoir quelle est la meilleure distance connue pour un état précis.

C'est là que la table de hachage va jouer un rôle : la clé sera l'état voisin généré à la volée, et la valeur enregistrée contiendra la distance la plus petite depuis la source jusqu'à cet état généré.

À chaque fois qu'un état voisin sera généré :

- On vérifie si cet état est déjà dans la table
- S'il n'est pas dans la table :
    - On l'insère dans la table et sa distance est enregistrée
- Sinon :
    - On calcule la nouvelle distance avec celle stockée :
        - Si la nouvelle distance est plus petite, on la met à jour
        - Sinon, on ignore ce nouveau chemin

L'algorithme à implémenter est donc le suivant :

Algorithme de Dijkstra avec table de hachage

```
Construire l'état initial e0 (distance = 0)
Créer une table de hachage et ajouter l'état initial à la table (distance = 0)
Créer une liste d'états visités et y insérer l'état initial

Tant que la liste des états visités n'est pas vide :
    Trouver l'état u qui a la plus petite distance parmi ceux de la liste des états visités
    Retirer l'état u de la liste
    Si l'état u est gagnant :
        Retourner la distance de cet état (Dijkstra terminé)
    Sinon :
        Pour chaque déplacement valide de l'état u :
            Construire l'état voisin v de u en suivant ce déplacement
            Si v n'est pas dans la table de hachage :
                Ajouter v dans la table (la distance (e0, v) est enregistrée)
                Ajouter v dans la liste des états visités
            Sinon :
                Si dist(e0, v) < dist[v] enregistrée dans la table de hachage
                    Mettre à jour la distance de l'état v dans la table de hachage
Si tous les états ont été visités sans trouver d'état gagnant, retourner -1
```

Dans cet algorithme, on a besoin de comparer la distance d'un état voisin généré par un déplacement valide à partir d'un état u, avec une distance déjà enregistrée précédemment pour ce voisin dans la table de hachage. Il faut donc qu'on puisse récupérer l'indice de cet état dans la table de hachage.

1. Écrire la fonction `t_recherche_etat(table, e)`, qui retourne l'indice d'un état enregistré dans la table de hachage (attention à ne pas oublier le chaînage…).

2. Écrire la fonction Dijkstra(e) qui renvoie la distance minimale pour arriver à la sortie à partir de l'état initial donné en paramètre.

Tester :

```
>>> etat=initialise()
>>> distance_min, etat_final = Dijkstra(etat)
>>> distance_min
21
>>> affiche_etat(etat_final)
   0 1 2 3 4 5
  +------------+
0 | . . 2 1 1 . |
1 | . . 2 . . . |
2 | . . 2 . G G |--> sortie
3 | . . 3 3 3 4 |
4 | . . . . . 4 |
5 | . . . . . 4 |
  +------------+
```

IV.3. Extrapolation du chemin optimal avec Dijkstra

Comme pour le BFS, on voudrait maintenant récupérer les déplacements optimaux à effectuer pour sortir le véhicule gris.

Pour cela, il suffit d'ajouter un dictionnaire qui enregistre, pour chaque nouvel état découvert, son état parent (d'où il vient) et par quel déplacement le nouvel état a été obtenu.

Ensuite, on peut remonter le chemin à l'envers jusqu'à l'état initial. Enfin, il faut ensuite renverser la liste.

1. Modifier votre fonction `Dijkstra(e)` afin qu'elle enregistre dans un dictionnaire les informations nécessaires à la reconstruction du chemin optimum (hash des états et déplacements effectués) et qu'elle retourne la distance minimale, l'état final et le dictionnaire de l'historique. On pourra par exemple utiliser un dictionnaire de la forme :

```
historique = {
    "etat_decouvert": [...],   # liste des hash des états découverts
    "etat_parent": [...]       # liste des hash des états parents
    "deplacement": [...]       # liste des déplacements effectués
     }
```

2. Tester votre fonction en utilisant la fonction de reconstruction déjà implémentée dans le cadre du BFS :

```
>>> etat_initial=initialise()
>>> distance_min, etat_final, historique = Dijkstra_chemin(etat_initial)
>>> print(distance_min)
21
>>> liste_dep = reconstruction(historique,etat_initial,etat_final)
>>> print(liste_dep)
[{'k': 1, 'pas': -1}, {'k': 4, 'pas': -3}, {'k': 3, 'pas': 3}, {'k': 2,
'pas': 3}, {'k': 0, 'pas': 3}, {'k': 2, 'pas': -3}, {'k': 3, 'pas': -1},
{'k': 4, 'pas': 3}, {'k': 0, 'pas': 1}]
```

V) COMPARAISON BFS / DIJKSTRA

Dans le cas où chaque coup joué coûte 1 point, on utilise BFS. Dans le cas où les coups joués engendrent des coûts différents, il faut utiliser Dijkstra.

Il y a un couple d' « état initial / état final » intéressant pour comparer BFS et Dijkstra : un cas où les deux algorithmes ne donnent pas les mêmes déplacements, et un cas où le chemin avec le moins de coups n'est pas celui avec le plus petit coût total (en nombre de cases déplacées).

Pour le modèle exact du sujet (coût = nombre de cases, déplacements longs autorisés en un seul coup), trouver un petit exemple concret « état initial / état final » où BFS et Dijkstra

donnent une solution vraiment différente n'est pas trivial, et il est possible que dans beaucoup de cas simples ils coïncident.

On va donc apporter une légère modification du coût pour rendre un exemple concret. Le coût sera maintenant donné par la formule : coût = longueur_du_véhicule × |pas|.

1. Définir une fonction permettant d'initialiser un état initial dans la configuration suivante :
   - La sortie est à droite sur la ligne i = 2.
   - La voiture grise (0) doit aller jusqu'à la colonne 4–5 pour sortir.
   - Le camion (1) bloque la ligne de sortie.
   - La petite voiture (2) peut être déplacée soit vers la gauche, soit vers la droite.

```
>>> affiche_etat(etat_initial)
   0 1 2 3 4 5
  +------------+
0 | . . . 1 . . |
1 | . . . 1 . . |
2 | G G . 1 . . |--> sortie
3 | . . . 2 2 . |
4 | . . . . . . |
5 | . . . . . . |
  +------------+
```

2. Appliquer l'algorithme BFS (coût unitaire pour chaque déplacement) et vérifier que le chemin optimal trouvé est le suivant :

```
BFS avec reconstruction :
Coût équivalent : 2x2+3x3+4x2=21,
[{'k': 2, 'pas': -2}, {'k': 1, 'pas': 3}, {'k': 0, 'pas': 4}]
```

Sur cette configuration, BFS a trouvé un chemin optimal en 3 déplacements pour un coût total équivalent de 21 : déplacer la voiture 2 de 2 cases à gauche, puis le camion de trois cases vers le bas, puis la voiture grise vers la sortie.

La fonction permettant de générer les déplacements avec le nouveau coût proportionnel à la longueur du véhicule se nomme `applique_deplacement_c_var2(etat, dep)`.

3. Modifier votre algorithme Dijkstra afin de prendre en compte le nouveau coût et appliquer l'algorithme. On obtient le résultat suivant :

```
   0 1 2 3 4 5
  +------------+
0 | . . . . . . |
1 | . . . . . . |
2 | . . . . G G |--> sortie
3 | . . . 1 2 2 |
4 | . . . 1 . . |
5 | . . . 1 . . |
  +------------+       Coût : 19
[{'k': 2, 'pas': 1}, {'k': 1, 'pas': 3}, {'k': 0, 'pas': 4}]
```

Dijkstra trouve également un chemin en 3 déplacements, mais il choisit un autre premier déplacement pour la voiture 2 (un glissement plus court vers la droite), ce qui donne un coût total plus petit (moins de « coût camions × distance » et « voiture × distance »).

Voici une autre configuration de départ :

```
Initialisation du plateau pour la comparaison BFS/Dijkstra
   0 1 2 3 4 5
  +------------+
0 | . . 3 3 . . |
1 | . . . . . 2 |
2 | G G 4 . . 2 |--> sortie
3 | . . 4 . . 2 |
4 | . . . 1 1 1 |
5 | . . . . . . |
  +------------+
```

Voici le résultat donné par BFS (5 déplacements pour un coût total équivalent de 25) :

```
BFS avec reconstruction
Coût équivalent : 1x3+2x3+2x2+2x2+4x2 = 25
[{'k': 1, 'pas': -1}, {'k': 2, 'pas': 2}, {'k': 3, 'pas': -2}, {'k':
4, 'pas': -2}, {'k': 0, 'pas': 4}]
```

Et le résultat donné par Dijkstra (6 déplacements pour un coût total de 21) :

```
Coût : 21
   0 1 2 3 4 5
  +------------+
0 | . . 3 3 . . |
1 | . . . . . . |
2 | . . 4 . G G |--> sortie
3 | . . 4 . . 2 |
4 | . . 1 1 1 2 |
5 | . . . . . 2 |
  +------------+
[{'k': 4, 'pas': 1}, {'k': 0, 'pas': 3}, {'k': 4, 'pas': -1}, {'k':
1, 'pas': -1}, {'k': 2, 'pas': 2}, {'k': 0, 'pas': 1}]
```

Ce dernier exemple montre que :

- BFS choisit un chemin avec moins de déplacements (5) mais plus cher (25).
- Dijkstra choisit un chemin avec plus de déplacements (6) mais moins cher (21).

Ces exemples montrent que :

- BFS minimise le nombre de coups (toutes les arêtes de même poids),
- Dijkstra minimise la somme des coûts quand les déplacements n'ont pas tous le même poids (ici longueur × |pas|), et peut donc prendre plus de coups mais moins coûteux au total.