CONCOURS CENTRALE•SUPÉLEC

Informatique

MPI

2025

4 heures

Calculatrice autorisée

Rush Hour

L'épreuve est composée de deux problèmes indépendants. L'objectif du premier problème est d'écrire un programme permettant de résoudre le casse-tête de déplacements nommé *Rush Hour* dont le but est de faire sortir une voiture d'un parking. Dans le second problème, on souhaite connaitre le maximum de véhicules qu'il est possible de placer sur le terrain d'une *casse automobile*.

Les candidats devront répondre aux questions en utilisant le langage C dans le premier problème et le langage OCaml dans le second problème. Ils sont invités, lorsque c'est pertinent, à décrire succinctement le fonctionnement de leurs programmes, un dessin étant souvent plus parlant qu'un long discours. Le barème tient compte de la clarté et de la concision des programmes. Nous recommandons de choisir des noms de variables intelligibles et de structurer, si cela s'avère nécessaire, de longs codes par des fonctions auxiliaires dont on décrira le rôle.

On identifiera une même grandeur écrite dans deux polices différentes, en italique lorsque nous la considèrerons d'un point de vue mathématique (par exemple n), et en police à largeur fixe lorsque nous la considèrerons d'un point de vue informatique (par exemple `n`). Si $a, b \in \mathbb{Z}$, on note $[\![a, b[\![$ l'ensemble $\{k \in \mathbb{Z} \,|\, a \leqslant k < b\} = \{a, a+1, \ldots, b-1\}$. Un graphe non orienté est un couple $G = (S, A)$ où S est un ensemble fini non vide d'éléments appelés sommets et A un ensemble de parties de S à deux éléments, appelées arêtes. Si $x, y \in S$ sont deux sommets distincts de S, l'arête $\{x, y\}$ sera notée $x - y$. Un graphe orienté est un couple $G = (S, A)$ où S est un ensemble fini non vide d'éléments appelés sommets et A un ensemble d'éléments (x, y) de $S \times S$ tels que $x \neq y$, appelés arcs. Si $x, y \in S$ sont deux sommets distincts de S, l'arc (x, y) sera noté $x \to y$.

Par complexité en temps d'un algorithme, on entend nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de l'algorithme dans le pire des cas.

Partie A – 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 1 illustre une position de départ du jeu.

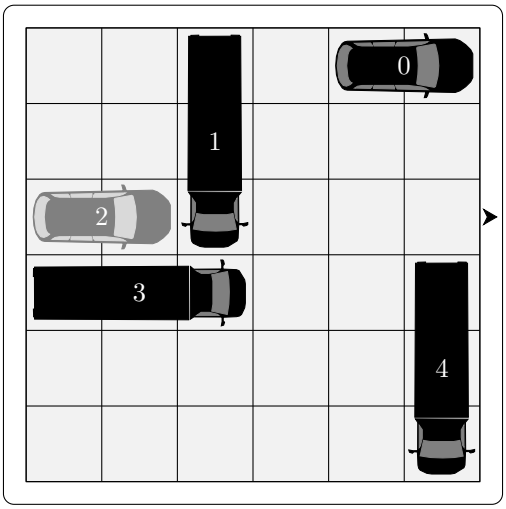


Figure 1.

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 sans la configuration de la figure 2.c.

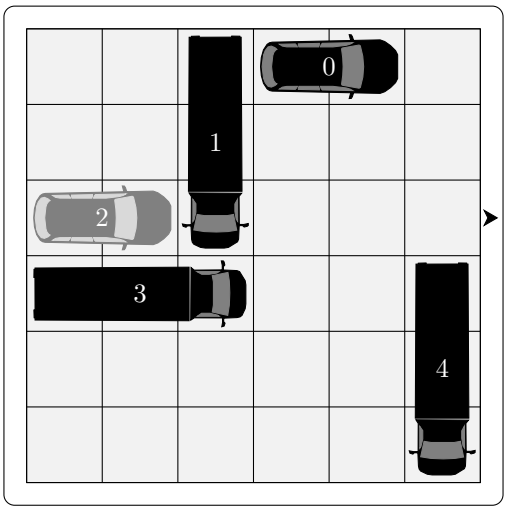


Figure 2.a

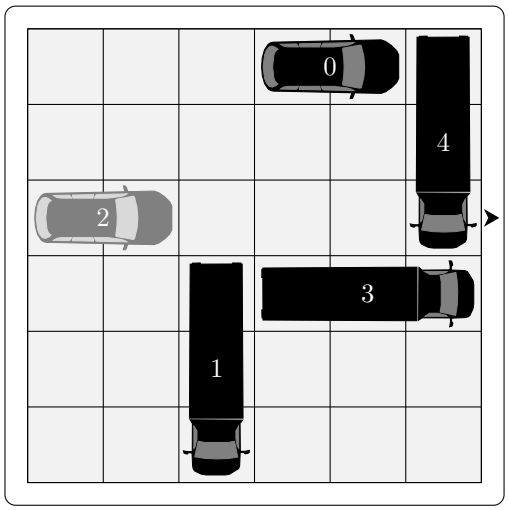


Figure 2.b

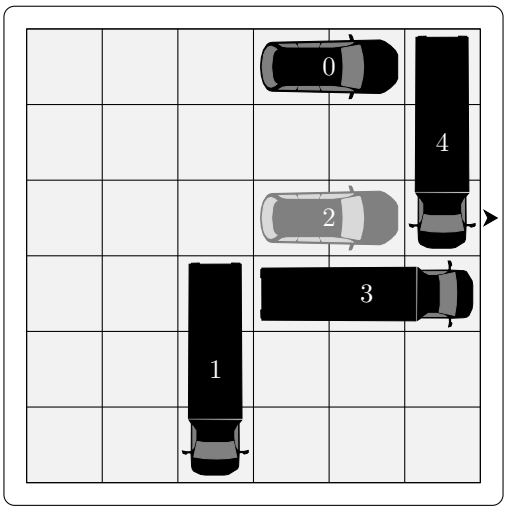


Figure 2.c

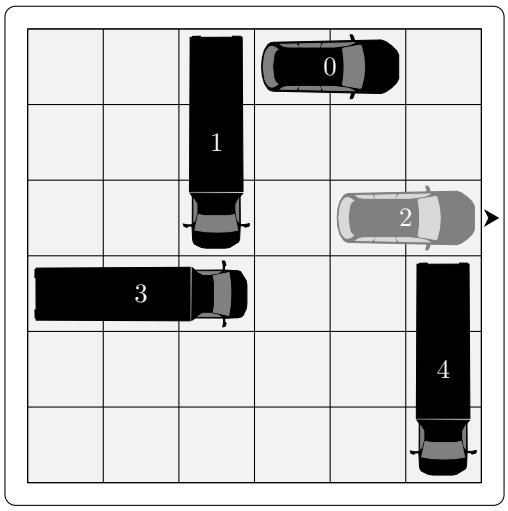


Figure 2.d

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.

Dans ce problème, nous supposerons que la grille du jeu est carrée et comporte $M \times M$ cases. Comme tous les véhicules occupent au moins 2 cases, le nombre de véhicules présents sur la grille est majoré par $(M \times M)/2$. Dans notre programme, cet entier est noté `MAX_V`. Les cases sont repérées par leurs coordonnées $(i, j) \in [\![0, M[\![^2$, l'indice i indiquant le numéro de la ligne et l'indice j indiquant le numéro de la colonne. La ligne sur laquelle se trouve la sortie est indiquée par S. Dans l'exemple de la figure 3 disponible page suivante, on a $M = 6$ et $S = 2$. Dans la suite du sujet, nous supposerons que la voiture grise est l'unique voiture placée horizontalement sur la ligne de la sortie. Comme c'est la seule voiture pouvant sortir de la grille, contrairement aux schémas présents dans ce sujet sur lesquels elle est en gris, elle ne fera l'objet d'aucun marquage particulier dans notre algorithme.

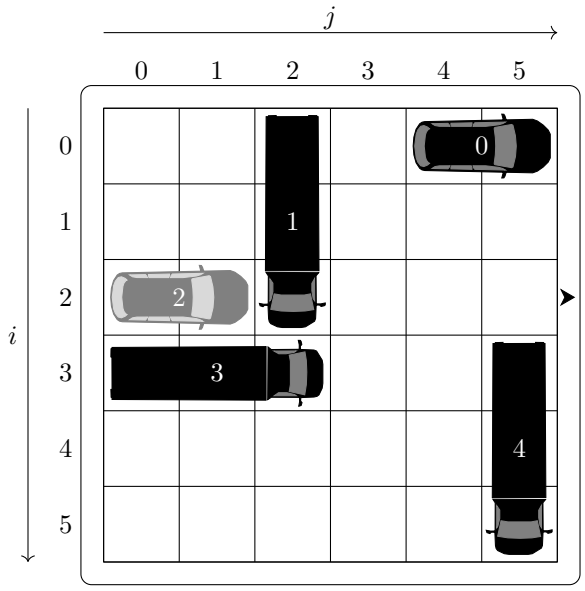


Figure 3.

Nous supposerons que les entêtes suivants ont été inclus et que les constantes ci-dessous sont définies au début du programme.

```c
#include <assert.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

#define M 6
#define MAX_V (M * M) / 2
#define S 2
```

Les directives **#define** présentes ici ont pour but de définir des synonymes pour les constantes entières utilisées dans le sujet. Chaque fois que l'identifiant `M` sera présent dans la suite, il sera directement substitué par la constante 6 par le compilateur.

I – Introduction

La position de chaque véhicule sur la grille est représentée par la structure de donnée suivante :

```c
struct vehicule_s {
    int i;
    int j;
    int longueur;
    char direction;
};
typedef struct vehicule_s vehicule;
```

Le couple (i, j) représente les coordonnées de la case occupée par le véhicule située la plus en haut à gauche sur la grille. Par exemple, sur la figure 3, les coordonnées du véhicule 0 sont $(0, 4)$ et celles du véhicule 1 sont $(0, 2)$. Le champ `longueur` indique la longueur, en nombre de cases, du véhicule : 2 pour une voiture et 3 pour un camion. Enfin, le champ `direction` indique la direction du véhicule : `'H'` si le véhicule est placé de manière horizontale, comme le véhicule 0 de la figure 3, et `'V'` si le véhicule est placé de manière verticale, comme le véhicule 1 de la figure 3.

La configuration des véhicules sur le plateau de jeu est représentée par la structure suivante :

```c
struct etat_s {
    vehicule v[MAX_V];
    int n;
    int distance;
};
typedef struct etat_s etat;
```

Le nombre de véhicules présents sur la grille est donné par le champ `n` et vérifie `n` $\leqslant$ `MAX_V`. Pour tout $k \in [\![0, n[\![$, le champ `v[k]` donne la position et les caractéristiques du véhicule d'indice k. Enfin, le champ `distance` n'est pas utilisé pour le moment et sera initialisé à 0.

Q1. Écrire une fonction `etat *e_nouveau(void)` renvoyant un état ne possédant aucune voiture sur son plateau, ainsi qu'une fonction `void e_libere(etat *e)` libérant la mémoire associée à cet état.

Q2. Écrire une fonction `bool e_egal(etat *e1, etat *e2)` déterminant si deux états sont égaux, c'est-à-dire possédant le même nombre de véhicules n et tels que pour tout $k \in [\![0, n[\![$ les véhicules d'indice k ont mêmes positions et mêmes caractéristiques. Deux états peuvent être égaux en ayant des valeurs différentes associées au champ `distance`.

Q3. Écrire une fonction `void e_ajoute_vehicule(int i, int j, int longueur, char direction, etat *e)` ajoutant à l'état e un véhicule à la position (i, j) dont la longueur et la direction sont données par les variables homonymes. On ne cherchera pas à vérifier que les cases sont libres, mais on utilisera une assertion pour vérifier que le nombre n de véhicules présents sur la grille reste inférieur ou égal à `MAX_V`.

Afin de faciliter la manipulation des états, nous allons définir une structure `plateau` représentant le plateau de jeu. Cette structure est définie comme suit :

```
const int vide = -1;

struct plateau_s {
    int tab[M][M];
};
typedef struct plateau_s plateau;
```

Q4. Écrire une fonction `plateau *p_etat(etat *e)` renvoyant un plateau dont les cases sont initialisées à la valeur `vide`, excepté les cases sur lesquelles se trouvent les véhicules de l'état e qui sont chacune initialisée avec l'indice du véhicule la recouvrant.

Pour chaque état, seul un nombre fini de déplacements est valide. Pour manipuler cette collection, nous allons utiliser une liste chainée, dont le maillon `deplacement` est défini comme ceci :

```
struct deplacement_s {
    int k;
    int pas;
    struct deplacement_s *suivant;
};
typedef struct deplacement_s deplacement;
```

Un tel maillon permet de représenter un déplacement de la voiture d'indice `k` de `pas` cases dans le sens des i (ou des j) croissants. Par exemple, dans la solution du casse-tête de la figure 1, 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`. Le champ `suivant` pointe vers le prochain maillon de la liste chainée. La liste vide est représentée par le pointeur `NULL`.

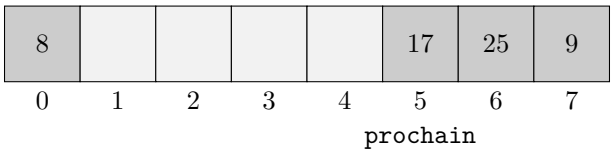
Q5. Écrire une fonction `deplacement *d_cons(int k, int pas, deplacement *lst)` ajoutant en tête de la liste chainée `lst`, un déplacement de la voiture d'indice `k` de `pas` cases.

Pour la suite du sujet, on suppose disposer d'une fonction `deplacement *e_deplacements(etat *e)` renvoyant la liste des déplacements valides dans l'état e.

Q6. Implémenter une fonction `void d_libere(deplacement *lst)` libérant la mémoire occupée par les différents éléments de la liste `lst`.

II – Implémentation d'une file

Dans cette partie, nous allons réaliser une file à l'aide d'un tableau circulaire. Sur l'exemple ci-dessous, nous avons une file d'entiers représentée par un tableau possédant 8 cases. On dit que la file a une capacité de 8 éléments.



Ce tableau représente la file $17 \leftarrow 25 \leftarrow 9 \leftarrow 8$. Le prochain élément à sortir de la file est 17 et le dernier élément à avoir été ajouté est 8. L'indice `prochain`, qui est égal à 5 sur cet exemple, indique l'indice du prochain élément qui sortira de la file. Cette file possède 4 éléments. On dit que sa taille est égale à 4.

La file que nous allons implémenter est une file d'états. Elle est représentée par la structure suivante :

```
struct file_s {
    etat **tab;
    int prochain;
    int taille;
    int capacite;
};
typedef struct file_s file;
```

Q7. Écrire une fonction `file *f_nouveau(void)` qui renvoie une file de taille et de capacité nulle. Écrire d'autre part une fonction `void f_libere(file *f)` libérant la mémoire associée à la file f. Cette fonction ne devra pas libérer la mémoire associée aux états présents dans la file.

Q8. Écrire une fonction `void f_change_capacite(file *f, int c)` prenant en entrée une file de taille m ainsi qu'un entier $c \geqslant m$. Cette fonction devra modifier la capacité de la file f afin qu'elle devienne égale à c.

Q9. Écrire les fonctions `void f_enfile(file *f, etat *e)` et `etat *f_defile(file *f)` permettant respectivement d'ajouter un nouvel élément e à l'entrée de la file f et d'enlever et récupérer un élément à la sortie de la file. Lorsqu'on souhaite ajouter un nouvel élément dans une file dont la capacité est égale à la taille, on multiplie cette capacité par 2 (sauf pour le cas particulier où la capacité est nulle où on la change en 1) avant d'ajouter notre nouvel élément.

On souhaite montrer que la complexité amortie des fonctions `f_enfile` et `f_defile` est en $\mathcal{O}(1)$, c'est-à-dire que si l'on part d'une file vide, la complexité d'une succession de n appels, chaque appel se faisant à l'une de ces fonctions, a une complexité en $\mathcal{O}(n)$. Pour les deux questions suivantes, et seulement pour ces deux questions, la complexité d'une opération sera calculée comme le nombre d'accès aux éléments du tableau `tab`, que ces accès soient en lecture ou en écriture. Nous allons utiliser pour cela la méthode du potentiel. On définit le potentiel Φ d'une file f par

$$\Phi(f) = |4 \cdot \text{taille}(f) - 2 \cdot \text{capacite}(f)|.$$

On note $f_0, f_1, \ldots, f_n$ les états successifs de la file. L'état f_0 est son état initial qui est une file de taille et de capacité nulle. Pour tout $k \in [\![1, n]\!]$, f_k est l'état de la file après la k-ième opération.

$$f_0 \xrightarrow{\text{op}_1} f_1 \xrightarrow{\text{op}_2} f_2 \ \cdots \ f_{n-1} \xrightarrow{\text{op}_n} f_n.$$

Pour tout $k \in [\![1, n]\!]$, on note C_k la complexité de l'opération op_k qui fait passer la file de l'état f_{k-1} à l'état f_k.

Q10. Montrer qu'il existe un entier $K \geqslant 0$ que l'on déterminera, tel que

$$\forall k \in [\![1, n]\!], \quad C_k + \Phi(f_k) - \Phi(f_{k-1}) \leqslant K.$$

Q11. En déduire que

$$\sum_{k=1}^{n} C_k \leqslant Kn$$

puis conclure.

III – Table de hachage

Dans cette partie, 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

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

où pour tout $i, j \in [\![0, M[\![$

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

On suppose disposer d'une fonction `plateau *p_hash(etat *e)` qui étant donné un état e, renvoie un plateau p dont les éléments `tab[i][j]` sont égaux à $d_{i,j}$.

Q12. Écrire une fonction `int64_t e_hash(etat *e)` prenant en entrée un état e et renvoyant $\text{hash}(e)$.

Q13. Montrer que si e_1 et e_2 sont deux états accessibles à partir d'un même état initial e et que $\text{hash}(e_1) = \text{hash}(e_2)$, 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 $h = \text{hash}(e) \mod 2^p$. 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.

e_4	e_0		e_2	e_3			e_1
0	1	2	3	4	5	6	7

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. Nous utiliserons la structure suivante pour représenter notre table de hachage.

```
struct tableh_s {
    etat **tab;
    int taille;
    int capacite;
};
typedef struct tableh_s tableh;
```

Le tableau `tab` contient les différents états présents dans la table de hachage. Nous utiliserons le pointeur `NULL` pour représenter les cases vides de la table. On suppose que l'on dispose d'une fonction `tableh *t_nouveau(void)` créant une table de hachage vide ayant une capacité de 1 et d'une fonction `void t_libere(tableh *t)` libérant la mémoire associée à la table de hachage t, ainsi que la mémoire associée à tous les états qu'elle contient.

Q14. Écrire une fonction `bool t_contient(tableh *t, etat *e)` déterminant si l'état e est présent dans la table de hachage t.

On suppose disposer d'une fonction `void t_augmente_capacite(tableh *t)` qui prend en entrée une table de hachage t de capacité $c \geqslant 1$ et qui augmente sa capacité à $2c$.

Q15. Écrire une fonction `void t_ajoute(tableh *t, etat *e)` qui ajoute un état e à la table de hachage t. Si m est la taille de la table et c sa capacité, on fera attention à ce que l'on ait toujours $m \leqslant 2c/3$.

Q16. Expliquer en quoi ce serait une mauvaise idée de remplacer le 3^{iM+j} par 4^{iM+j} dans la définition de $\text{hash}(e)$.

IV – Recherche d'une solution optimale

On supposera disposer dans cette partie d'une fonction `bool e_gagne(etat *e)` déterminant si la voiture grise est devant la sortie, ainsi que d'une fonction `etat *e_copie(etat *e)` effectuant une copie d'un état e.

Q17. Écrire une fonction **`int`** `resout(etat *e)` prenant en entrée un état e et renvoyant le nombre minimal de mouvements pour résoudre le jeu. La fonction devra renvoyer -1 dans le cas où il n'y a pas de solution. On souhaite de plus que la mémoire associée à l'état e soit libérée à la fin de l'appel à `resout`. On pourra utiliser le champ `distance` d'un état pour indiquer le nombre de coups minimal nécessaires pour atteindre cet état à partir de l'état initial.

Q18. Montrer que la fonction `resout` n'engendre aucun problème mémoire.

Partie B – La casse automobile

Dans cette partie, nous disposons d'un terrain sur lequel nous souhaitons entreposer le plus grand nombre de véhicules hors d'usage. Ce terrain est modélisé par un ensemble de cases du plan, extraites d'une grille régulière. La taille d'un terrain désigne le nombre de cases le constituant. Deux types de véhicules sont disponibles : les voitures, qui occupent 2 cases, et les camions, qui occupent 3 cases.

Le premier objectif de cette partie est d'implémenter un algorithme efficace, ayant une complexité polynomiale en la taille du terrain, permettant de déterminer le maximum de voitures qu'il est possible de disposer sur ce dernier. Cet algorithme nous permet notamment de décider s'il est possible de couvrir tout le terrain à l'aide de voitures. La figure 4 donne un exemple de couverture complète d'un terrain possédant 14 cases à l'aide de voitures.

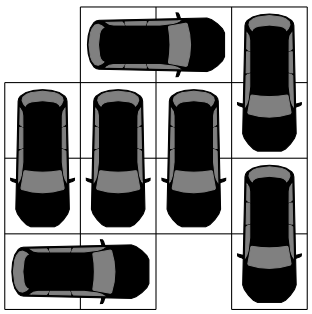


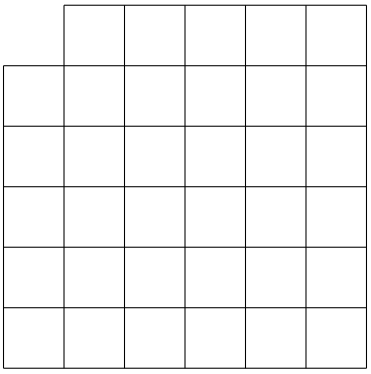
Figure 4.

Le second objectif de cette partie est de démontrer que, si l'on remplace les voitures, qui occupent 2 cases, par des camions, qui occupent 3 cases, le problème devient NP-complet.

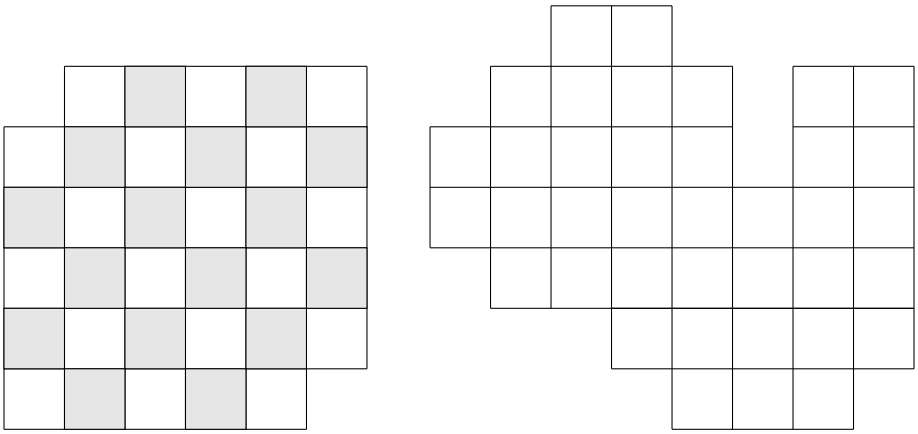
I – Remplir une casse avec le maximum de voitures

Dans cette partie B.I, tous les véhicules considérés sont des voitures et occupent 2 cases.

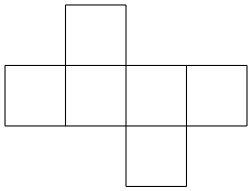
Q19. Prouver qu'il est impossible de couvrir tout le terrain suivant à l'aide de voitures. En déduire une condition nécessaire sur le terrain pour qu'il soit possible de le couvrir entièrement à l'aide de voitures.



Q20. Prouver qu'il est impossible de couvrir entièrement le terrain de gauche ci-dessous à l'aide de voitures. Ce terrain a une taille de 34 cases et la coloration de ses cases en blanc et en noir à la manière d'un échiquier n'a aucune incidence sur la disposition des voitures. Est-il possible de couvrir entièrement le terrain de droite ci-dessous ? On attend soit un exemple de couverture, soit une preuve de son impossibilité. En déduire une nouvelle condition nécessaire sur le terrain pour qu'il soit possible de le couvrir entièrement à l'aide de voitures.



Q21. En considérant le terrain ci-dessous, prouver que la condition nécessaire trouvée dans la question précédente n'est pas suffisante pour assurer l'existence d'une couverture complète à l'aide de voitures.



Dans la suite de cette partie, nous allons écrire une algorithme permettant de déterminer le plus grand nombre de voitures qu'il est possible de disposer sur un terrain, ainsi qu'une telle disposition. Pour cela, on définit le graphe non orienté $G = (S, A)$ dont l'ensemble des sommets S est constitué des cases du terrain, une arête $x - y$ appartenant à A si et seulement si x et y sont deux cases adjacentes, c'est-à-dire lorsque y est l'une des 4 cases au-dessus, en dessous, à gauche ou à droite de x. Nous choisissons arbitrairement de colorier une case du terrain en noir, puis de colorier le reste du terrain à la manière d'un échiquier en alternant les couleurs blanches et noires. On note N l'ensemble des sommets noirs et B l'ensemble des sommets blancs. Puisqu'une arête relie toujours deux sommets de couleurs différentes, $G = (N \sqcup B, A)$ est un graphe biparti. On numérote les r cases du terrain en commençant par les p cases noires, puis les q cases blanches. On a donc $r = p + q$, $N = [\![0, p[\![$ et $B = [\![p, r[\![$. En OCaml, ce graphe sera représenté par un tableau de listes d'adjacence `(g : int list array)` ainsi que l'entier `(p : int)`. Pour tout $x \in [\![0, r[\![$, `g.(x)` est la liste des voisins du sommet x.

On rappelle qu'un couplage de $G = (S, A)$ est une partie C de A telle que chaque sommet n'est l'extrémité que d'au plus une arête de C. Un sommet $x \in S$ est dit libre lorsqu'il n'est l'extrémité d'aucune arête de C. Deux sommets $x, y \in S$ sont dits conjoints lorsqu'ils sont les extrémités d'une même arête de C. En OCaml, un couplage sera représenté par le tableau `(conjoint : int option array)` défini par

- `conjoint.(x) = None` lorsque le sommet x est libre.
- `conjoint.(x) = Some y` et `conjoint.(y) = Some x` lorsque x et y sont conjoints.

Un chemin $x_0 - x_1 - \cdots - x_k$ de longueur k sera représenté en OCaml par la liste `[x0; ...; xk]` et sera représenté par la variable `(w : int list)`. Un chemin est dit élémentaire lorsque ses sommets sont deux à deux distincts. Un chemin est dit alternant pour C lorsqu'il est élémentaire et ses arêtes $x_i - x_{i+1}$ alternent entre des éléments de C et $A \setminus C$. On dit qu'il est augmentant lorsqu'il est alternant et que ses extrémités sont libres.

Si X et Y sont deux parties de A, on définit la différence symétrique $X \Delta Y = (X \cup Y) \setminus (X \cap Y)$. On admet que cette opération est associative et commutative. Si X est une partie de A et w est un chemin, $X \Delta w$ est défini en considérant le chemin w comme un ensemble d'arêtes.

Q22. Montrer que si C est un couplage et que w est un chemin augmentant pour C, alors $C\Delta w$ est un couplage.

Q23. Écrire une fonction `delta (conjoint : int option array) (w : int list) : unit` prenant en entrée un couplage C représentée par le tableau `conjoint` et un chemin w que l'on supposera augmentant et modifiant `conjoint` de manière à ce qu'il représente le couplage $C\Delta w$.

On définit le graphe orienté G_C possédant les mêmes sommets que G et possédant un arc orienté pour chaque arête $n - b$ de G où $n \in N$ et $b \in B$: si $n - b \in C$, cet arc est $b \to n$, sinon cet arc est $n \to b$. D'autre part, dans la suite de cette partie, on utilisera l'enregistrement

```
type couplage = {
  graphe : int list array;
  p : int;
  conjoint : int option array;
}
```

pour représenter un couplage ainsi que le graphe biparti auquel il est associé. Si `c` est une variable de type `couplage`, `c.graphe` permet d'accéder au tableau de listes d'adjacence du graphe, `c.p` permet d'accéder à l'entier p et `c.conjoint` permet d'accéder au tableau `conjoint`. On construit un couplage à l'aide de la syntaxe

```
{ graphe = g; p = p; conjoint = c }
```

Enfin, on note respectivement N_ℓ et B_ℓ l'ensemble des sommets libres de N et de B.

Q24. Montrer qu'un chemin $x_0 - x_1 - \cdots - x_k$ de G dont le premier élément x_0 est dans N_ℓ et le dernier élément x_k est dans B_ℓ est augmentant pour C si et seulement si $x_0 \to x_1 \to \cdots \to x_k$ est un chemin de G_C.

Q25. Écrire une fonction `oriente (c : couplage) : int list array` prenant en entrée un couplage C et renvoyant le tableau de listes d'adjacence représentant le graphe orienté G_C.

Pour tout $x \in S$, on note $\mathrm{d}(x)$ la plus petite longueur parmi les longueurs des chemins alternants reliant un élément $n \in N_\ell$ à x. Si un tel chemin n'existe pas, on pose par convention $\mathrm{d}(x) = +\infty$. Afin de manipuler efficacement cette distance, nous utiliserons par la suite l'enregistrement

```
type distance = {
  d : int array;
  dmin : int;
}
```

où `d` est un tableau tel que `d.(x) =` $\mathrm{d}(x)$ pour tout $x \in [\![0, r[\![$, la valeur $+\infty$ étant représenté par l'entier `max_int`, et `dmin` est la longueur $d_{\min}$ du plus court chemin augmentant pour C, avec par convention `dmin = max_int` lorsqu'un tel chemin n'existe pas.

Q26. Écrire une fonction `distance (c : couplage) : distance` prenant en entrée un couplage C et renvoyant l'enregistrement `(dist : distance)` associé à la distance d. On pourra utiliser le module `Queue` de la bibliothèque standard OCaml offrant une implémentation d'une file impérative et dont la signature est rappelée ci-dessous.

- `Queue.create () : 'a Queue.t`, création d'une file vide.
- `Queue.push (x : 'a) (f : 'a Queue.t) : unit`, enfiler un élément x dans la file f.
- `Queue.pop (f : 'a Queue.t) : 'a`, défiler et renvoyer le prochain élément en attente de la file f.
- `Queue.is_empty (f : 'a Queue.t) : bool`, renvoie `true` lorsque la file f est vide, `false` sinon.

Nous souhaitons construire le plus grand ensemble de chemins augmentants pour C tel que ces chemins n'aient aucun sommet en commun et soient tous de longueur $d_{\min}$. Un tel ensemble est appelé ensemble de chemins de Hopcroft-Karp. Pour construire cet ensemble, on commence par définir le graphe G_{HK} possédant les mêmes sommets que G_C, en ne conservant que les arcs $x \to y$ tels que $\mathrm{d}(y) = \mathrm{d}(x) + 1$ et $\mathrm{d}(y) \leqslant d_{\min}$. Ensuite, on initialise notre ensemble de chemins à l'ensemble vide. Puis, pour chaque sommet $x \in N_l$, on cherche un chemin reliant x à un sommet $y \in B_l$ dans G_{HK} à l'aide d'un parcours en profondeur. Tant qu'un tel chemin existe, on retire de G_{HK} les sommets empruntés par notre chemin, on ajoute ce chemin à notre ensemble de chemins et on réitère l'opération. Pour notre implémentation, on ne

cherchera pas à générer le graphe G_{HK}, mais on travaillera uniquement à partir du graphe G_C.

On souhaite donc écrire une fonction `ensemble_chemins (c : couplage) (dist : distance) : int list list` prenant en entrée un couplage C, ainsi que l'enregistrement `dist` calculé par la fonction `distance` et renvoyant un ensemble de chemins de Hopcroft-Karp sous forme de liste. Pour cela, on introduit l'exception

```
exception Chemin of int list
```

et on définit la fonction

```
let ensemble_hc (c : couplage) (dist : distance) : int list list =
  let r = Array.length c.graphe in
  let visite = Array.make r false in
  let go = oriente c in
  let ensemble = ref [] in
  let rec chemin (x : int) (w : int list) : unit =
    <code1>
  in
  <code2>
  !ensemble
```

Q27. Compléter la partie `<code1>` de la fonction `ensemble_hc` afin de définir la fonction `chemin (x : int) (w : int list) : unit`.

Cette fonction doit effectuer un parcours en profondeur dans G_{HK} à partir du sommet x en ayant déjà parcouru le chemin w codé à l'envers (le dernier sommet visité est en tête de liste), et renvoyer une exception `Chemin w` lorsqu'un chemin augmentant w a été trouvé.

Q28. Compléter la partie `<code2>` de la fonction `ensemble_hc`.

Dans la question suivante, on admet le lemme de Berge : un couplage dans un graphe est maximum pour l'inclusion si et seulement si il n'existe pas de chemin augmentant.

Q29. Définir la fonction `couplage_maximum (g : int list array) (p : int) : int option array` prenant en entrée un graphe biparti donné par son tableau de listes d'adjacence ainsi que par son nombre de sommets noirs et renvoyant un couplage maximum pour ce graphe.

II – Remplir une casse avec le maximum de camions

Dans cette partie, on souhaite montrer que le problème qui consiste, étant donné un terrain t ainsi qu'un entier n, de déterminer s'il est possible de placer au moins n camions sur le terrain t, est NP-complet. Dans la suite, on nomme ce problème 3CASSE.

Q30. Définir ce qu'est un problème de décision appartenant à la classe NP. Définir ce qu'est un problème NP-difficile, puis un problème NP-complet.

On définit les types

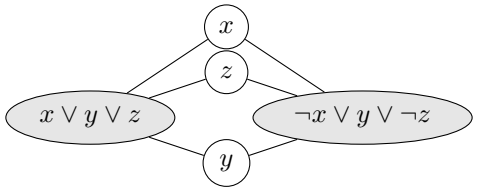
```
type terrain = bool array array
type direction = Horizontal | Vertical
type camion = {i : int; j : int; d : direction}
```

le type `terrain` représentant un terrain à l'aide d'un tableau bidimensionnel de booléens. Pour chaque case de cette grille, le booléen associé est égal à `true` si et seulement si elle fait partie du terrain. Le type `camion` représente quant à lui la position d'un camion sur le terrain, de manière similaire à ce qui a été décrit dans la partie A.I.

Q31. Écrire une fonction `est_valide (t : terrain) (n : int) (lst : camion list) : bool` prenant en entrée un terrain t, un entier n et une liste de positions de camions `lst` et renvoyant `true` si la liste possède n camions et qu'il est possible de les placer sur le terrain t comme spécifié. Si ce n'est pas le cas, la fonction renverra `false`.

Q32. En déduire que le problème 3CASSE est dans la classe NP.

On souhaite montrer que 3CASSE est NP-difficile. Pour cela, nous allons partir d'une variante du problème 3SAT, nommée Planar 3SAT et qui est également NP-complet. Tout comme pour 3SAT, les instances de ce problème de décision sont des formules logiques en forme normale conjonctive. Pour une formule φ on considère le graphe non orienté dont les sommets sont formés des variables et des clauses et dont les arêtes relient une variable à une clause si et seulement si la variable est présente dans la clause. Le problème Planar 3SAT est semblable à 3SAT mais restreint les formules à celles dont le graphe est planaire. Par exemple, la formule $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z)$ est planaire puisqu'on peut la représenter par le graphe suivant :



Étant donnée une formule φ de Planar 3SAT, nous allons construire une instance de 3CASSE. Pour cela, nous allons construire un terrain à l'aide de *gadgets*. L'essentiel de ces gadgets peut accueillir un nombre fixe de camions. Les gadgets représentant des clauses sont des exceptions et peuvent accueillir un camion en plus lorsque cette clause est vraie. Les gadgets sont reliés entre eux afin de créer des chemins, transportant des *signaux*. Chaque signal représente la valeur d'un littéral x ou de sa négation $\neg x$.

Concrètement, les gadgets sont reliés entre eux en superposant une de leur case, par laquelle sera transmis ce signal. Les gadgets sont agencés de manière à ce que tout camion placé sur le terrain se trouve entièrement sur un unique gadget. Par exemple, le terrain de la figure 5 est composé de 4 gadgets α, β, γ, δ reliés par les cases a, b, c et d. Les gadgets α et γ sont reliés par la case a qui appartient aux deux gadgets. Sur ce terrain, nous avons placé deux camions. Le camion 1 appartient au gadget γ. Bien qu'il empiète sur le gadget α, on ne considère pas qu'il appartienne à ce gadget. Le camion 2 appartient quant à lui au gadget δ.

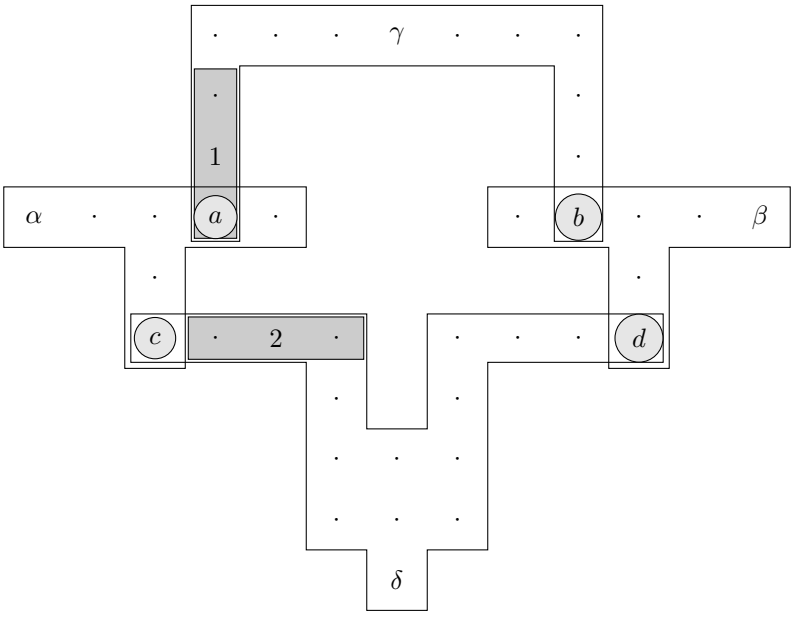
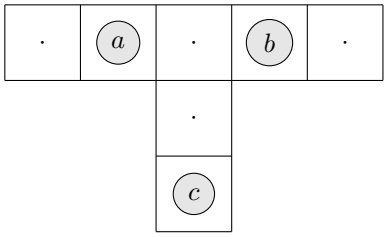


Figure 5.

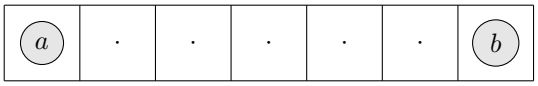
La disposition des camions sur le terrain va permettre de représenter les signaux. Si ε est un gadget et qu'un camion n'appartenant pas à ce gadget empiète sur la case e du gadget ε, on dit que le gadget ε possède une protrusion en e. On dit dans ce cas que, pour le gadget ε, le signal e est faux. Dans le cas contraire, on dit que ce signal est vrai. Par exemple, avec la disposition de camions ci-dessus, pour le gadget α, on dit que le signal a est faux et que le signal c est vrai.

Q33. Afin de représenter une clause $a \vee b \vee c$, nous allons utiliser le gadget suivant sur lequel il peut y avoir des protrusions sur les cases a, b, c.



Montrer que ce gadget permet seulement d'accueillir 0 ou 1 camion et que l'un des signaux a, b, c est vrai si et seulement si il est possible d'y accueillir un camion.

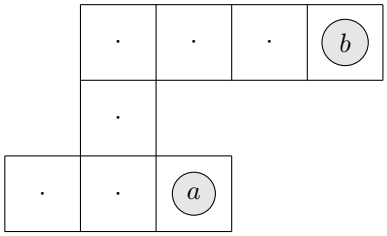
Q34. Afin de transmettre une valeur, nous allons utiliser le gadget suivant, sur lequel on souhaite placer deux camions.



Montrer que, si le signal a est faux, pour placer deux camions sur ce gadget, il est nécessaire de rendre le signal b faux pour le gadget suivant. Montrer que, si le signal a est vrai, il est possible de placer ces deux camions de manière à ce que le signal b soit vrai pour le gadget suivant.

L'idée du gadget précédent peut être généralisée afin d'obtenir un gadget permettant de transmettre un signal tout en se décalant d'un nombre de cases qui est un multiple de 3. Il est cependant nécessaire de transmettre des signaux tout en se décalant d'un nombre de cases quelconque.

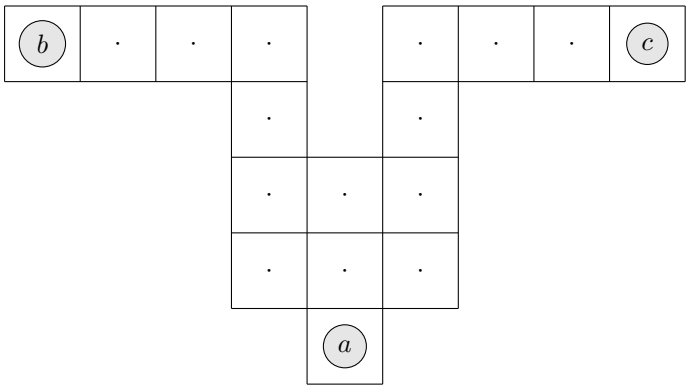
Q35. Afin de transmettre un signal tout en le décalant de deux cases, nous allons utiliser le gadget suivant, sur lequel on souhaite placer deux camions.



Montrer que, si le signal a est faux, pour placer deux camions sur ce gadget, il est nécessaire de rendre le signal b faux pour le gadget suivant. Montrer que, si le signal a est vrai, il est possible de placer ces deux camions de manière à ce que le signal b soit vrai pour le gadget suivant.

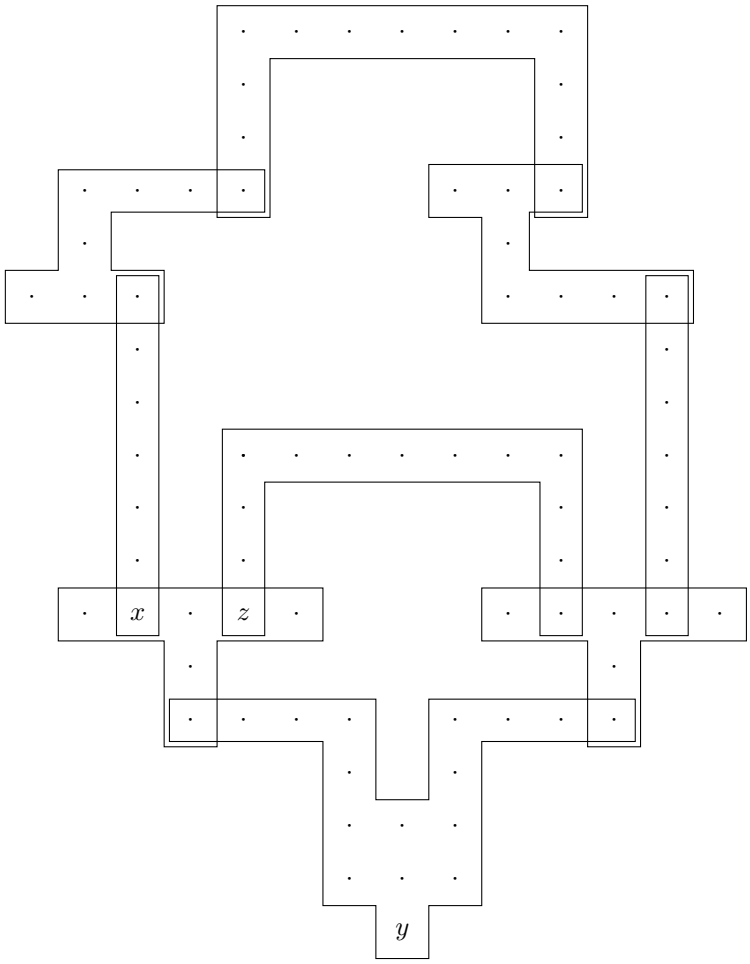
Un signal représentant un littéral x (ou sa négation $\neg x$) peut être utilisé dans plusieurs clauses. C'est pourquoi il est nécessaire de pouvoir le dupliquer.

Q36. Afin de diviser un signal en deux, nous allons utiliser le gadget suivant, sur lequel on souhaite placer cinq camions.



Montrer que, si le signal a est faux, pour placer cinq camions sur ce gadget, il est nécessaire de rendre les signaux b et c faux pour les gadgets suivants. Montrer que, si le signal a est vrai, il est possible de placer cinq camions de manière à ce que les signaux b et c soient vrais pour les gadgets suivants.

Q37. En utilisant les idées développées dans les questions 33 à 36 ainsi que le fait qu'il est possible de satisfaire la formule $\varphi = (x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z)$, déterminer le plus grand nombre de camions qu'il est possible de placer sur le terrain suivant. On justifiera la réponse en donnant une telle disposition ainsi que les valeurs des variables x, y et z associées.



Q38. Montrer que le problème 3CASSE est NP-complet.

◇ Fin ◇