Mines Informatique commune MP 2017 — Corrigé

Ce corrigé est proposé par Virgile Andreani (ENS Ulm) ; il a été relu par Cyril Ravat (Professeur en CPGE) et Julien Dumont (Professeur en CPGE).

Ce sujet aborde la modélisation d'un croisement routier, représenté par deux files de voitures qui s'intersectent.

- Dans la première partie, il pose les bases de la modélisation avec le cas d'une file unique.
- Dans la deuxième, il invite à programmer la dynamique de la file en présence ou non d'un obstacle sur la voie, à l'aide d'une opération élémentaire, fournie par une fonction déjà écrite, qui consiste à faire avancer toute la file d'un coup.
- Dans une troisième partie qui ne comporte que deux questions, on s'intéresse enfin au cas du croisement, en simulant conjointement deux files couplées par une case commune.
- La quatrième partie, courte elle aussi, prépare la suivante en introduisant la notion de transition entre configurations non immédiatement successives.
- L'avant-dernière partie, la moins facile de l'épreuve, est consacrée au codage d'un algorithme de recherche en largeur sur les états du système ce qui permet de déterminer si une configuration est atteignable depuis une autre.
- Enfin, l'épreuve se termine par trois questions utilisant les bases de données.

La difficulté de ce problème est progressive, avec de nombreuses questions très accessibles en début et milieu d'épreuve, suivies d'autres plus délicates en fin d'épreuve, notamment dans la cinquième partie. Il est accessible dès la première année (sauf la question 18). L'ensemble du programme est couvert à l'exception de la simulation numérique.

Indications

Partie II

9 La fonction `avancer` peut servir d'opération élémentaire dans la fonction demandée, il faut s'en servir. On peut faire appel à la concaténation de listes au moyen de l'opérateur `+`.

10 Noter que la case m est inoccupée. Que cela implique-t-il sur les voitures à gauche de m ?

11 Étant donné que les voitures immédiatement à gauche de m sont bloquées, où sont les voitures qui ont la possibilité d'avancer ?

Partie III

12 Bien réfléchir à l'ordre dans lequel appeler les trois fonctions précédemment définies `avancer_debut`, `avancer_debut_bloque` et `avancer_fin`.

Partie V

17 Noter que la liste est triée : il suffit donc de comparer chaque valeur à la précédente pour déterminer si elle est nouvelle ou non.

20 Chercher dans le code où apparaissent les variables en question et les opérations dans lesquelles elles sont impliquées pour déterminer leur type.

21 Quel est le principal critère de choix d'un algorithme ?

24 Démontrer que le nombre d'éléments uniques contenus dans la liste `espace` est croissant et borné.

I. Préliminaires

1 Soit une case est vide, soit elle contient une voiture. On peut donc encoder ces deux états au moyen d'une variable binaire qui vaut `True` si une voiture est présente à cet endroit et `False` sinon. Pour représenter n cases, on utilise une liste de n de ces booléens.

2 On commence par créer une liste vide, puis on remplit les cases qui doivent l'être :

```
A = 11*[False]
A[0] = True
A[2] = True
A[3] = True
A[10] = True
```

> Il vaut mieux éviter ici de construire la liste directement avec `A = [True, False, True, True, etc.]` pour réduire les risques d'erreur.

3 Vu la manière dont on a encodé les états des cases, la fonction `occupe` ne renvoie rien d'autre que la valeur booléenne de la case demandée :

```
def occupe(L, i):
    return L[i]
```

> Il faut prendre soin à partir d'ici d'utiliser uniquement cette fonction pour vérifier l'état des cases et éviter d'indexer directement les listes. En effet, si on décidait un jour de changer de représentation pour les états des cases, il suffirait d'adapter cette fonction une fois au lieu de chercher dans le code tous les endroits où l'on indexe une liste.

4 Chaque case pouvant être dans deux états différents, et ce de manière indépendante des autres cases,

> Le nombre de files différentes est le produit du nombre d'états possibles pour chacune des cases soit 2^n

.

5 Deux listes sont égales si et seulement si elles ont la même longueur et chacune de leurs cases égales deux à deux. Par conséquent :

```
def egal(L1, L2):
    if len(L1) != len(L2):
        return False
    for i in range(len(L1)):
        if L1[i] != L2[i]:
            return False
    return True
```

> On fait le test de longueur en premier pour s'assurer que les indices seront valides dans la boucle. À l'intérieur de celle-ci, on peut retourner `False` dès la première paire de valeurs différentes, inutile de poursuivre.
>
> Python permet de comparer directement deux listes avec l'opérateur `==`, mais écrire `L1 == L2` ici annule l'intérêt de la question et ne fait pas apparaître explicitement la complexité.

6 Si les listes sont de longueur différente, il n'y a pas de boucle et la fonction s'achève en temps constant. Le pire cas correspond à la situation où l'on itère la boucle un maximum de fois, c'est-à-dire quand les listes sont de même taille et que tous leurs éléments sont égaux deux à deux (sauf éventuellement les deux derniers). Dans ce cas,

La complexité est linéaire en la longueur commune des listes

7 Comme il était demandé à la question 5, cette fonction retourne un booléen.

II. Déplacement de voitures dans la file

8 Cette évolution consiste à faire avancer la file deux fois, d'abord sans ajouter de voiture au début, puis en en ajoutant une. On obtient donc `[True, False, True, False, True, True, False, False, False, False, False]`.

9 Cette étape partielle consiste à faire avancer la deuxième partie de la liste et à laisser la première inchangée. On peut réutiliser la fonction `avancer` pour la deuxième partie et recoller le résultat avec la première partie de la liste au moyen de l'opérateur de concaténation `+`.

```
def avancer_fin(L, m):
    return L[:m] + avancer(L[m:], False)
```

> Pour cette question comme pour les suivantes, l'énoncé incite à réutiliser la fonction `avancer`, qu'il définit mais n'utilise pas ailleurs. Une réponse acceptable sans utiliser cette fonction aurait pu être
>
> ```
> def avancer_fin(L, m):
>     return L[:m] + [False] + L[m:-1]
> ```
>
> ou encore, sans concaténation :
>
> ```
> def avancer_fin(L, m):
>     L1 = L[:m]
>     L1.append(False)
>     for i in range(m, len(L)-1):
>         L1.append(L[i])
>     return L1
> ```

10 La case m étant inoccupée, les voitures à gauche de m ne sont pas bloquées et peuvent toutes avancer. On peut donc là aussi réutiliser la fonction `avancer` sur la première partie de la liste et recoller le résultat avec la seconde partie.

```
def avancer_debut(L, b, m):
    return avancer(L[:m+1], b) + L[m+1:]
```

11 Les voitures bloquées immédiatement à la gauche de m ne bougent pas, de la même manière que les voitures à la droite de m. Toutes les voitures à la droite de la première case inoccupée à la gauche de m (si elle existe) voient donc leur position inchangée par `avancer_debut_bloque`, seules les voitures à la gauche de cette case pouvant avancer librement. Si toutes les cases à la gauche de m sont occupées, alors aucune voiture ne peut avancer. Par conséquent, il nous faut trouver l'indice l de la première case inoccupée à la gauche de m et appeler `avancer_debut(L, b, l)`.