Samedi 15 janvier PCSI

Devoir surveillé n° 2 – Correction

Problème 1 : Étude de trafic routier

Partie I. Préliminaires

1. On peut représenter une file de voitures par une liste de booléens, `True` représentant la présence d'une voiture et `False` représentant une absence de voiture. Par exemple `[True, False, True]` pourrait représenter une file de 3 places avec une voiture au début et à la fin, et un espace vide entre les deux.

2. On peut écrire, par exemple :

```
A = [True, False, True, True] + 6 * [False] + [True]
```

ou

```
A = [True, False, True, True, False, False, False, False, False, False, True]
```

pour représenter la situation de la Figure 1.

3. Par exemple :

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

Cette fonction renvoie `True` s'il y a `True` en position `i` dans la liste `L` et donc s'il y a une voiture en position `i` dans la file. Et inversement avec l'absence de voiture.

4. Pour une file de longueur n, on a 2 situations possibles seulement pour chaque place (une voiture ou pas de voiture). On a donc 2^n listes possibles.

5. Par exemple :

```
def egal(L1, L2):
    return L1 == L2
```

Cette fonction renvoie le booléen associé au test d'égalité entre les deux listes. Si elles diffèrent d'une seule position au moins, elle renverra `False`.

6. Le test d'égalité `L1 == L2` demande de comparer chaque élément de `L1` à l'élément correspondant de `L2`. La complexité, dans le pire des cas, est donc linéaire en la taille des listes.

7. La fonction `egal` renvoie un booléen.

8. On propose :

```
def nombre_occ(L):
    compteur = 0
    for x in L:
        if x:
            compteur += 1
    return compteur
```

9. On propose :

```
def max_succ(L):
    max = 0
    i = 0
    while i < len(L):
    # i est l'indice de parcours de la liste
        max_local = 0
        j = i
        while j < len(L) and occupe(L, j):
        # à partir de i on détermine la longueur
        # de la suite de places occupées
            max_local += 1
            j+= 1
        if max_local > max :
            max = max_local
        # Pour avancer dans la liste (indice i) :
        if max_local > 1 :
            i+= max_local
        else :
            i+= 1
    return max

# autre possibilité :
def max_succ(L):
    a = 0
    b = 0
    for i in range(len(L)):
        if L[i]:
            b+=1
        else:
            b=0
        if b > a:
            a = b
    return a
```

Parie II. Déplacement de voitures dans la file

10. On a :

```
A = [True, False, True, True, False, False, False, False, False, False, True]
```

L'appel `avancer(A,False)` renvoie la liste :

```
[False, True, False, True, True, False, False, False, False, False, False]
```

Et donc l'appel `avancer(avancer(A, False), True)` renvoie la liste :

```
[True, False, True, False, True, True, False, False, False, False, False].
```

11. Par exemple :

```
def avancer_fin(L, m):
    n = len(L)
    Lretour = []
    for i in range(m):
        Lretour.append(L[i])
    L2 = avancer(L[m:n], False)
    for j in range(n-m):
        Lretour.append(L2[j])
    return Lretour
```

ou

```
def avancer_fin(L, m):
    n = len(L)
    L1 = L[:] # Copie de L
    for i in range(m+1, n):
        L1[i] = L[i-1]
    L1[m] = False
    return L1
```

ou encore, en version courte :

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

12. Par exemple :

```
def avancer_debut(L, b, m) :
    n = len(L)
    Lretour = avancer(L[0:m+1], b)
    for i in range(m+1, n):
        Lretour.append(L[i])
    return Lretour
```

ou

```
def avancer_debut(L, b, m):
    n = len(L)
    L1 = L[:] # Copie de L
    for i in range(1, m+1):
        L1[i] = L[i-1]
    L1[0] = b
    return L1
```

ou, en version courte :

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

13. Par exemple :

```
def avancer_debut_bloque(L, b, m):
    for i in range(1 ,m):
        if not occupe(L, m-i):
            return avancer_debut(L, b, m-i)
    return L[:]
```

Partie III. Une étape de simulation à deux files

14. Par exemple :

```
def avancer_files(L1, b1, L2, b2):
    m = len(L1)//2
    R1 = avancer_fin(L1, m)
    R2 = avancer_fin(L2, m)
    if occupe(R1, m-1):
        R2 = avancer_debut_bloque(R2, b2, m)
    else:
        R2 = avancer_debut(R2, b2, m)
        R1 = avancer_debut(R1, b1, m)
    return[R1, R2]
```

ou, plus malin :

```
def avancer_files(L1, b1, L2, b2):
    m = len(L1)//2
    R2 = avancer_fin(L2, m)
    R1 = avancer(L1, b1) #Les voitures de L1 ne peuvent pas être bloquées
    if occupe(R1, m):
        R2 = avancer_debut_bloque(R2, b2, m)
    else:
        R2 = avancer_debut(R2, b2, m)
    return[R1, R2]
```

15. L'appel `avancer_files(D, False, E, False)` renvoie :

```
[[False, False, True, False, True],[False, True, False, True, False]]
```

Partie IV. Transitions

16. Considérons la situation suivante :

- la file `L1` est pleine,
- à chaque étape de la simulation, on ajoute une nouvelle voiture à la file `L1`,
- une voiture est sur la case `m-1` de la file `L2`.

Dans ce cas, la voiture sur la case `m-1` de la file `L2` est indéfiniment bloquée.

17. Examinons le temps nécessaire pour que les voitures de la file `L2` soient dans la position voulue :

- Les voitures de la file `L2` doivent laisser la priorité aux voitures de la file `L1`, elles restent donc immobiles jusqu'à l'étape 4 incluse.
- Les voitures de la file `L2` commencent à se déplacer à l'étape 5 et arrive dans la position voulue à l'étape 9.

Il n'est donc pas possible d'atteindre la configuration demandée en moins de 9 étapes. Il suffit d'ajouter des voitures à `L1` aux étapes 6, 7, 8 et 9 (et de n'ajouter aucune voiture à `L2`) pour obtenir la configuration souhaitée en 9 étapes. Conclusion : il faut 9 étapes.

18. Pour obtenir la configuration de droite en une étape à partir d'une configuration `C`, il faut que les voitures en position 6 de la configuration de droite soient, dans la configuration `C` toutes les deux en position 5 dans leur liste, donc toutes les deux sur le croisement, ce qui est impossible. Il est donc impossible de passer de la configuration de gauche à la configuration de droite.

Partie V. Atteignabilité

19. Par exemple :

```
def elim_double(L) :
    if L == []:
        return L[]
    R = [L[0]]
    n = len(L)
    for i in range(1, n):
        if L[i] > L[i-1]: # ou L[i] != R[-1] ou ...
        R.append(L[i])
    return R
```

20. L'appel `doublons([1, 1, 2, 2, 3, 3, 3, 5])` renvoie `[1, 2, 3, 5]`. `doublons` est une version récursive de `elim_double`.

21. Non, car elle ne compare que les valeurs consécutives d'une liste, elle ne peut pas supprimer un doublon de valeurs si les deux valeurs ne sont pas consécutives.

22. On a :

- `recherche` renvoie un booléen.
- `successeurs` renvoie une liste de listes de 2 listes de même longueur impaire correspondant à des files.
- `espace` pointe sur une liste de listes de 2 listes de même longueur impaire correspondant à des files.
- `but` pointe sur une liste de 2 listes de même longueur impaire correspondant à des files.

23. La recherche par `in1` a une complexité linéaire tandis que celle par `in2` est dichotomique, donc de complexité logarithmique. La deuxième est donc préférable.

24. Considérons k le nombre de configurations qui ne sont pas dans `espace`. On remarque que k est un variant de boucle (la suite des valeurs de k est une suite d'entiers naturels strictement décroissante), ainsi la boucle ne peut pas être infinie et termine. La suite des valeurs de k est strictement décroissante car la ligne 6 fait décroître k et la ligne 9 arrête la boucle si k n'a pas strictement décru (s'il n'a pas strictement décru, comme les doublons sont supprimés et les listes triées, il y aura égalité à la ligne 9).

25. Par exemple :

```
1  def recherche(but, init):
2      espace = [init]
3      if egal(init, but):
4          return 0
5      stop = False
6      nb_etapes = 0 # initialisation du nombre d'étapes
7      while not stop:
8          ancien = espace
9          espace = espace + successeurs(espace)
10         espace.sort() # permet de trier espace par ordre croissant
11         espace = elim_double(espace)
12         stop = egal(ancien,espace)
13         nb_etapes += 1 # on incrémente le nombre d'étapes
14         if but in espace:
15             return nb_etapes
16     return -1
```

On peut montrer la correction de cette fonction modifiée en utilisant l'invariant de boucle suivante : « `espace` contient toutes les configurations atteignables en `nb_etapes` étapes ou moins ». La fonction s'arrête à la première apparition de `but` dans `espace` et renvoie bien le nombre minimal d'étapes pour y arriver (on ne sort de la boucle qu'à la première apparition de `but` dans `espace` s'il y est) et `-1` s'il n'y apparaît pas.

26. On propose :

```
1  def avancer(L, b):
2      # L = une liste (on suppose que L contient au moins un élément...)
3      # b = un booléen
4      Lretour = L[:] # copie de L
5      n = len(L)
6      for i in range(1, n) :
7          Lretour[i] = L[i-1]
8      Lretour[0] = b
9      return Lretour
```