

Table des matières

I	Voyageur de commerce	2
1	Algorithme glouton	2
2	Force brute	3
3	Commentaires	4
II	Mots croisé	5
4	Questions préliminaires	5
5	Aide à la résolution	6
6	Les choses sérieuses commencent	8

Troisième devoir surveillé d'informatique

7 janvier 2022

Première partie

Voyageur de commerce

Soit $n \in \mathbb{N}$ et A_0, \dots, A_{n-1} n points du plan. M. X doit passer à chacun de ses points, dans un ordre quelconque, et cherche à minimiser la longueur totale de son trajet. On notera en outre P_0 sa position initiale.

Chaque point sera représenté en Python par un tableau de deux cases : les coordonnées du point dans un repère orthonormé fixé.

1 Algorithme glouton

M. X décide d'aller à chaque instant vers le point restant à visiter le plus proche de sa position actuelle.

1. Écrire une fonction `d_euc` prenant deux points et renvoyant la distance euclidienne entre ces deux points.
solution :

```
4 def d_euc(A, B):
5     xa, ya = A
6     xb, yb = B
7     return ((ya-xa)**2 + (yb-xb)**2)**.5
8 # rema : dans tout le problème on pourrait utiliser la distance au carré plutôt que la
   ↪ distance, pour éviter ces racines carrées.
```

2. Écrire une fonction `point_le_plus_proche` prenant en entrée un point P et un tableau de points t et renvoyant l'indice du point de t le plus proche de P .
solution :

```
12 def point_le_plus_proche(P, t):
13     dist = d_euc(P, t[0])
14     res = 0
15     for i in range(1, len(t)):
16         essai = d_euc(P, t[i])
17         if essai < dist:
18             dist = essai
19             res = i
20     return res
```

3. Écrire une procédure `supprime_ième` prenant un tableau t et un indice i d'icelui et supprimant l'élément i de t . Les opération Python sur les tableaux autorisées sont la suppression du dernier élément et la lecture et écriture d'une case.
solution :

```
24 def supprime_ième(t,i):
25     """
26     Effet : supprime 'l'élément 'd'indice i dans t, sans préserver 'l'ordre des autres
       ↪ éléments.
27     """
28     t[i]=t[-1]
29     t.pop()
```

4. Écrire une fonction `sans_le_ième` prenant un tableau t et un indice i d'icelui et renvoyant t privé de son i -ème élément.

solution :

```

1 '''
33 def sans_le_ième(t, i):
34     res = []
35     for j in range(len(t)):
36         if j!=i:
37             res.append(t[j])
38     return res

```

5. Écrire la fonction finale `voyageur_glouton`. Celle-ci prendra en entrée P_0 et le tableau $[A_0, \dots, A_{n-1}]$ et renverra un tableau contenant les points à visiter dans l'ordre où ils le seront par M. X.

solution :

```

1 '''
43 def voyageur_glouton(P0, à_visiter):
44     res = [P0]
45     P=P0 # position actuelle
46     while len(à_visiter) >0:
47         i = point_le_plus_proche(P, à_visiter)
48         P = à_visiter[i]
49         res.append(P)
50         supprime_ième(à_visiter, i)
51     return res

```

6. Trouver un exemple où cet algorithme glouton ne renvoie pas la solution la plus courte. *Indication :* On peut trouver un contre-exemple en prenant tous les points sur l'axe des abscisses.

solution : Par exemple : $P_0 = (3, 0)$, $A_0 = (0, 0)$, et $A_1 = (5, 0)$, $A_3 = (9, 0)$.

Avec l'algo glouton on va en A_1 puis A_2 puis A_0 , et la distance parcourue est de 15. Si on allait d'abord à A_0 , puis A_1 puis A_2 la distance serait de 12.

2 Force brute

M. Y décide de calculer tous les itinéraires possibles passant par les n points et de choisir le plus court. Un itinéraire sera un tableau contenant les n points A_0, \dots, A_{n-1} dans un certain ordre.

1. Écrire une fonction `longueur_iti` prenant un itinéraire ainsi que la position initiale P_0 et renvoyant la longueur totale du trajet (y compris la partie de la position initiale vers `iti[0]`).

solution :

```

1 '''
58 def longueur_iti(P0, iti):
59     P=P0 # position actuelle
60     res = 0
61     for M in iti:
62         res += d_euc(P, M)
63         P=M
64     return res

```

2. Combien y-a-t-il d'itinéraires possibles ? Combien d'additions seront effectuées par M. Y ?

solution : Le nombre d'itinéraires est le nombre d'ordres possibles sur l'ensemble des n points, c'est donc $n!$. Chaque calcul d'itinéraire nécessite n additions, donc au total M. X va effectuer $n \times n!$ additions.

3. Il reste à calculer toutes les permutations possibles. Compléter le programme ci-dessous. Vous pouvez rajouter autant de lignes que vous le souhaitez, et même créer des fonctions auxiliaires.

```

1 def permutations(t):
2     """
3     Entrée : un tableau t
4     Précondition : les éléments de t sont deux à deux distincts.
5     Sortie : le tableau de toutes les permutations de t.

```

```

6      """
7
8      if t == []:
9          return ...
10     else:
11         res=[]
12         for x in t:
13             # Calculons toutes les permutations qui se terminent par x
14             t_privé_de_x = ...
15             perms = permutations(t_privé_de_x)
16             # On rajoute x aux éléments de perms
17             ...
18             res.extend(...)
19         return res

```

solution :

```

1      """
71     def permutations(t):
72         """
73         Entrée : un tableau t
74         Sortie : le tableau de toutes les permutations de t.
75         """
76
77         if t == []:
78             return [[]]
79         else:
80             res=[]
81             for x in t:
82                 t_privé_de_x = [y for y in t if y!=x]
83                 perms = permutations(t_privé_de_x)
84                 for p in perms:
85                     p.append(x)
86                 res.extend(perms)
87             return res

```

4. Expliquer l'utilité de la précondition. Que se passerait-il si t contenait un élément en double ? (Cela peut dépendre de la manière dont vous avez programmé votre fonction.)

solution : S'il y avait un élément x en double dans t , au moment du calcul de $t_privé_de_x$, on enlèverait en fait les deux occurrences de x . Et dans les résultats finals renvoyés, il n'y aurait plus qu'un seul x .

5. Écrire enfin la fonction finale **voyageur_brut** prenant la position initiale et le tableau des points à visiter et renvoyant l'itinéraire le plus court passant par ces points.

solution :

```

1      """
92     def voyageur_brut(P0, à_visiter):
93         res = à_visiter
94         l_min = longueur_iti(P0, à_visiter)
95         for essai in permutations(à_visiter):
96             l_essai = longueur_iti(P0, essai)
97             if l_essai < l_min:
98                 l_min = l_essai
99                 res = essai
100         return res

```

3 Commentaires

Indiquer avantages et inconvénients des deux méthodes.

Pour la culture : aucune solution entièrement satisfaisante n'est connue pour ce problème. Précisément : on ne connaît aucun algorithme donnant la solution optimale en un nombre d'opération qui ne soit pas au moins une exponentielle en n .

Deuxième partie

Mots croisé

Une grille de mots croisés sera représentée par une matrice de chaînes de caractères. Une case non remplie contiendra le caractère ' ' (un espace). Une case noire (sur laquelle on ne peut inscrire aucune lettre) contiendra le caractère '#'. La figure 1 donne un exemple de grille de mots croisés et la représentation Python de celle-ci.

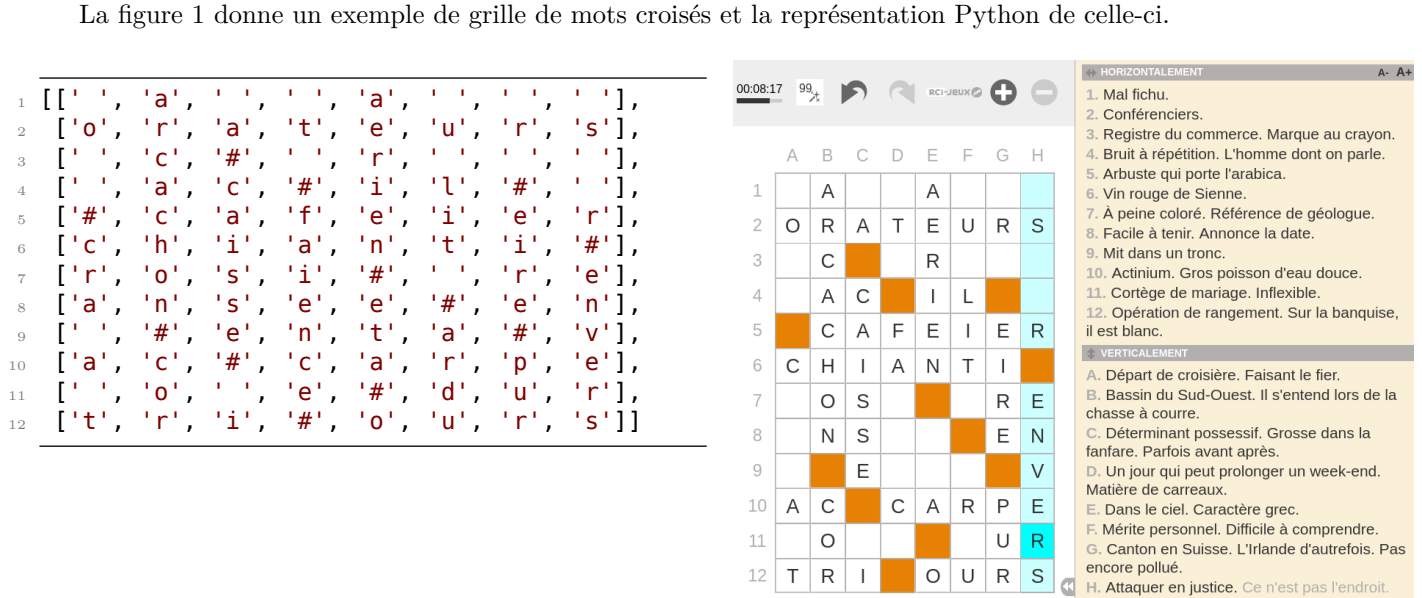


FIGURE 1 – Exemple de grille de mots croisés

4 Questions préliminaires

solution : Le fichier `mots_croisés.py` contient toutes les fonctions nécessaires à la résolution d'une grille de mots croisés, y compris celles permettant de récupérer le tableau `TOUS_LES_MOTS` contenant tous les mots du français sans accent.

Dans la fonction `mots_extraits`, indiquer l'emplacement où vous avez enregistré votre fichier `mots_français.txt` contenant ceux-ci.

1. Écrire en Python un prédicat indiquant si une grille est entièrement remplie, autrement dit si elle ne contient plus d'espace.
2. Écrire une fonction `mot_horizontal` prenant une grille `g` et deux indices `i` et `j` et renvoyant le mot horizontal inscrit dans la grille `g` qui démarre en case (i, j) . Il s'agit donc du mot contenant les lettres lues dans `g` en partant de la case (i, j) et en avançant vers la droite jusqu'à l'extrémité de la grille ou la prochaine case noire.

Plutôt que le mot lui-même, de type `str`, on pourra renvoyer le tableau de ses lettres.

Remarque : Dans cette question comme dans les suivantes, l'utilisation de la fonction `n'a` de sens que si la case (i, j) est le début d'un mot. On ne se préoccupera toutefois pas de vérifier ici si c'est effectivement le cas.

3. Écrire un prédicat `est_possible_h` prenant une grille `g`, deux indices `i` et `j` et une chaîne de caractères `mot` et indiquant si on peut inscrire le mot `mot` dans la grille `g` en partant de la case (i, j) en inscrivant le mot horizontalement (donc « vers la droite »). Il faudra vérifier qu'il y a le bon nombre de cases, qu'aucune de ces cases ne contient un '#' ou une lettre différente de celle qu'on souhaite inscrire, et enfin qu'il ne restera pas de case vide après le mot, autrement dit que la dernière lettre du mot sera dans la dernière case de la ligne `i` ou sera suivie d'un '#'.

solution :

```
40 def est_possible_h(g,i,j,m):
41
42     """ Indique si on peut placer le mot m à partir de la case (i,j) horizontalement. """
43     res = True
44     p = len(g[0])
```

```

45     for k in range(len(m)):
46         if j+k>=p or g[i][j+k]=="#" or (g[i][j+k] not in [" ",m[k]]):
47             res=False
48     k=len(m)
49     return res and (j+k==p or g[i][j+k]=="#")

```

Décrire brièvement comment modifier ce code pour obtenir un prédicat `est_possible_v` analogue.

solution :

```

53 def est_possible_v(g,i,j,m):
54     """ Indique si on peut placer le mot m à partir de la case (i,j) horizontalement. """
55     res = True
56     n = len(g)
57     for k in range(len(m)):
58         if i+k>=n or g[i+k][j]=="#" or (g[i+k][j] not in [" ",m[k]]):
59             res=False
60     k = len(m)
61     return res and (i+k==n or g[i+k][j]=="#")

```

4. Donner les spécifications de la fonction suivante. En particulier, quel est le sens de l'argument `d` ?

```

1 def est_possible (g,i,j,d,m):
2     return (d==0 and est_possible_h (g,i,j,m)) or (d==1 and est_possible_v (g,i,j,m))

```

solution :

- **Entrées :**
 - ◊ `g` une grille
 - ◊ `i` et `j` (entiers) indices d'une case de `g`
 - ◊ `d` $\in \{0,1\}$ indique une direction : 0 horizontalement, 1 verticalement
 - ◊ `m` (str) un mot
- **Sortie** le booléen « On peut écrire le mot `m` dans la grille `g`, en partant de la case (i,j) , dans la direction `d` ».

5. Écrire une procédure `écrit_h` prenant une grille `g`, deux indices `i` et `j` et une chaîne de caractères `mot` et inscrivant le mot `mot` dans la grille `g` horizontalement en partant de la case (i,j) . On supposera que c'est effectivement possible, autrement dit que le résultat de `est_possible_h(g,i,j,mot)` est `true`.

solution :

```

67 def écrit_h(g,i,j,mot):
68     for k in range(len(mot)):
69         g[i][j+k] = mot[k]

```

6. On suppose connu un tableau de mots contenant tous les mots nécessaires à la résolution de la grille. Ce tableau est placé en variable globale, et son nom est `TOUS_LES_MOTS`.

Écrire une fonction prenant une grille `g`, deux indices `i` et `j`, un entier `d` $\in \{0,1\}$ et renvoyant le tableau des mots qu'il est possible d'écrire dans `g` à partir de la case (i,j) dans la direction indiquée par `d`.

5 Aide à la résolution

Dans cette partie, nous fixons une grille `g`, dont nous notons `n` le nombre de lignes et `p` le nombre de colonnes. Tout au long de l'algorithme, nous maintiendrons une matrice `possibles`, de format $(n,p,2)$ telle que pour tout $(i,j,d) \in \llbracket 0,n \rrbracket \times \llbracket 0,p \rrbracket \times \llbracket 0,1 \rrbracket$, `possibles[i][j][d]` contiendra la liste des mots qu'il est possible d'écrire à partir de la case (i,j) dans la direction `d`.

1. *cours* : Écrire une fonction `nv_possibles` prenant en entrée `n` et `p` et renvoyant une matrice de format (n,p) , initialement remplie par des `[]`, `[]`.

solution :

```

152 def nv_possibles(n,p):
153     return [ [[]],[] ] for j in range(p) for u in range(n) ]

```

2. Écrire une fonction `init_possibles` prenant la grille `g` et renvoyant la matrice possibles telle que décrite ci-dessus.

solution :

```
157 def init_possibles(g, bavard=True):
158     n = len(g)
159     p = len(g[0])
160     à_remplir = cases_début(g)
161     possibles = nv_possibles(n,p)
162     print("Recherche des mots possibles")
163     for (i,j,d) in à_remplir:
164         if bavard : print(f" Case {i},{j}, direction {d}")
165         possibles[i][j][d] = [mot for mot in TOUS_LES_MOTS if est_possible(g,i,j,d,mot)]
166     print("Initialisation des mots possibles terminée.\n")
167     return possibles
```

3. Il est inutile de remplir toutes les cases de possibles : seules les cases situées au début d'un mot nous intéressent. Écrire une fonction `cases_début_h` renvoyant le tableau des couples (i,j) qui sont les coordonnées du début d'un mot horizontal.

solution :

```
124 def cases_début_h(g):
125     n=len(g)
126     p=len(g[0])
127     res=[]
128     for i in range(n):
129         for j in range(p-1): # p-1 car je ne veux pas de mots de une seule lettre
130             if g[i][j]!="#" and (j==0 or g[i][j-1]=="#") and g[i][j+1]!="#" and not
131                 ↪ mot_repli(g,i,j,0):
132                 res.append((i,j))
132     return res
```

Bonus : ignorer les mots d'une seule lettre.

4. Dans une optique gloutonne, quelle stratégie pourrait-on envisager pour choisir la prochaine case à remplir ? Toute réponse argumentée de manière cohérente sera acceptée.

solution : Je propose de choisir en priorité la cases où il y a le moins de possibilités, car c'est celle qui devrait être la plus facile.

5. En déduire une fonction `cases_début` renvoyant le tableau des triplets (i,j,d) tels que la case (i,j) est le début d'un mot dans la direction d .

solution : Je suppose qu'une fonction `cases_début_v` analogue à `cases_début_h` a été écrite.

```
147 def cases_début(g):
148     return [(i,j,0) for (i,j) in cases_début_h(g)] + [(i,j,1) for (i,j) in
    ↪ cases_début_v(g)]
```

6. Écrire une fonction `prochaine_case` prenant en entrée la tableau `à_remplir` tel que renvoyé par la fonction précédente, ainsi que la matrice `possibles` et renvoyant un triplet (i,j,d) pour lequel le nombre de possibilités est minimal. Par confort pour l'utilisateur, renvoyer (i,j,d) ainsi que la liste des possibilités.

Indication : Puisque `à_remplir` est un tableau de triplets, on peut le parcourir au moyen de la syntaxe suivante :

« `for (i, j, d) in à_remplir :` »

solution :

```
171 def prochaine_case( à_remplir, possibles):
172     """
173     Renvoie un triplet ( i,j, dir) indiquant une case où il y a le minimum de
    ↪ possibilités. dir indique la direction, 0 pour horizontal, et 1 pour vertical.
174     """
175     (i,j,d) = à_remplir[0]
176     mini = len(possibles[i][j][d])
177     res = (i,j,d)
178     for (i,j,d) in à_remplir :
179         if len(possibles[i][j][d]) < mini:
```

```

180         mini = len(possibles[i][j][d])
181         res= (i,j,d)
182     return res

```

6 Les choses sérieuses commencent

1. Écrire une procédure `màj` qui prend :

- la grille `g`;
- la matrice `possibles`;
- le tableau `à_remplir`;
- trois entiers `i, j, d` indiquant une position et une direction dans la grille;
- un mot `mot` dans `possibles[i][j][d]`;

et qui a les effets suivants :

- écrire `mot` dans `g` à l'emplacement (i, j, d) ;
- mettre à jour `possibles` en supprimant les mots qui ne sont plus possibles;
- supprimer (i, j, d) de `à_remplir`.

2. Modifier `màj` pour renvoyer en outre :

- Le tableau des cases $(i2, j2)$ de `g` où une nouvelle lettre a effectivement été écrite;
- Le tableau des quadruplets $(i2, j2, d2, m)$ tels que le mot `m` a été supprimé de `possibles[i2][j2][d2]`.

3. Écrire une procédure `annule_màj` prenant en entrée `g, i, j, d, possibles, à_remplir` ainsi que le résultat de `màj` tel que décrit à la question précédente et remettant `g, à_remplir`, et `possibles` dans l'état où ils étaient avant cet appel.

On peut alors écrire la fonction principale `résolution` qui va essayer de remplir la grille `g`. Cette fonction prendra en entrée `g, possibles`, et `à_remplir`. Elle renverra `True` si elle est parvenue à remplir `g` et `False` sinon.

Voici le principe proposé : Soit `i, j, d` le résultat de `prochaine_case(à_remplir, possibles)`. Pour tout mot dans `possibles[i][j][d]` :

- Essayer d'inscrire `mot` en (i, j, d) .
- Relancer `résolution`. Si la suite de la résolution fonctionne tout va bien. Sinon, annuler l'écriture de mot.

Si aucun mot n'a fonctionné renvoyer `False`.

solution :

```

194 import copy
195 def résolution(g, possibles=None, bavard=True):
196     n= len(g)
197     p=len(g[0])
198     if possibles==None :
199         possibles = init_possibles(g, bavard=bavard)
200     else:
201         possibles = copy.deepcopy(possibles)
202     à_remplir = cases_début(g)
203
204     def màj(i,j,d,mot):
205         """
206         Écrit mot en (i,j,d).
207         Met à jour possibles.
208         Supprime (i,j,d) de à_remplir.
209
210         Renvoie :
211         (La liste des (i2,j2) où une lettre a été écrite,
212          La liste des quadruplets (i2,j2,d2, m) de mots supprimés de possibles)
213         """
214
215         à_remplir.remove((i,j,d))
216

```



```

217 cases_écrites=[]
218 for l in range(len(mot)):
219     i2, j2 = avancé(i,j,d,l)
220     if g[i2][j2]==" ":
221         g[i2][j2]=mot[l]
222         cases_écrites.append((i2, j2))
223
224 mots_supprimés=[]
225 if d==0:
226     j_concernés = [j2 for (_,j2) in cases_écrites]
227 else:
228     i_concernés = [i2 for (i2,_) in cases_écrites]
229 for (i2,j2,d2) in à_remplir:
230     if d!=d2 and (d==0 and j2 in j_concernés or d==1 and i2 in i_concernés) :
231         nv_poss = []
232         for m in possibles[i2][j2][d2]:
233             if est_possible(g,i2,j2,d2,m):
234                 nv_poss.append(m)
235             else:
236                 mots_supprimés.append( (i2,j2,d2,m) )
237         possibles[i2][j2][d2] = nv_poss
238
239 return cases_écrites, mots_supprimés
240
241
242
243 def annule_changements(i,j,d,diff):
244     """ Prend les données renvoyées par māj, et remet g, à_remplir et possibles dans
245         ↪ l'état précédent cet appel à māj."""
246
247     for (i2,j2) in diff[0]:
248         g[i2][j2]=" "
249     for (i2,j2,d2,m) in diff[1]:
250         possibles[i2][j2][d2].append(m)
251     à_remplir.append((i,j,d))
252
253
254 #Fonction de backtracking pour objets mutables
255 print("Départ du backtracking")
256 def aux(prof):
257     """ prof indique la profondeur de l'appel actuel dans l'arbre des appels. Autrement
258         ↪ dit c'est le nombre de mots inscrits dans la grille. Cet argument ne sert que
259         ↪ pour l'affichage."""
260     if à_remplir == []:
261         return True
262     else:
263         (i,j,d) = prochaine_case(à_remplir, possibles)
264         if len(possibles[i][j][d]) == 0:
265             return False
266         else:
267             for mot in possibles[i][j][d] :
268                 # On essaie d'écrire mot en (i,j,d)
269                 if bavard : print(f{' '*prof}Écriture de {mot} en {i,j,d}")
270                 diff = māj(i,j,d,mot)
271                 if aux(prof+1):
272                     return True
273                 else:
274                     if bavard:print(f{' '*prof}Effaçage de {mot} en {i,j,d}")
275                     annule_changements(i,j,d,diff)
276             return False #aucun mot n'a fonctionné
277
278 return aux(0)

```

Ce type de méthode où on essaie une possibilité, on poursuit la résolution, et on efface si on rencontre une impossibilité s'appelle « backtracking ».