DS INFO N°2 (le 28/03/2023)

Avertissement :

La quatrième partie du Problème 2 ne fait pas partie du DS.

La troisième partie du Problème 1 est plus difficile.

PROBLÉME 1 (X-ENS 2022 MP-PC-PSI, 2h

Une phrase courte et claire prend moins de temps à écrire que des pensées confuses...
Utilisez du brouillon !

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve.
Le langage de programmation sera obligatoirement `Python`.

Complexité.

La complexité, ou le temps d'exécution, d'une fonction P est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que P a une complexité en $\mathcal{O}(\phi(n,m))$ lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de P est inférieure ou égale à $A \cdot \phi(n,m)$, pour tout $n \geqslant n_0$ et $m \geqslant m_0$.

Une fonction prenant une liste en argument sera dite de complexité linéaire si elle est de complexité $\mathcal{O}(n)$ où n désigne la longueur de la liste passée en argument.

Lorsqu'il est demandé de donner la complexité d'un programme, vous devrez justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Rappels concernant le langage Python.

L'utilisation de toute fonction Python sur les listes autre que celles mentionnées dans ce paragraphe est interdite.

Si `a` désigne une liste en Python de longueur n :

- `len(a)` renvoie la longueur de cette liste, c'est-à-dire le nombre d'éléments qu'elle contient ; la complexité de `len` est en $\mathcal{O}(1)$.
- `a[i]` désigne le i-ème élément de la liste, où l'indice i est compris entre 0 et `len(a) - 1` ; la complexité de cette opération est en $\mathcal{O}(1)$.
- `e in a` teste si e est élément de a ; la complexité de cette opération est en $\mathcal{O}(n)$.
- `a.append(e)` ajoute l'élément e à la fin de la liste a ; la complexité de cette opération est en $\mathcal{O}(1)$.
- `a.pop()` renvoie la valeur du dernier élément de la liste a et l'élimine de la liste a ; la complexité de cette opération est en $\mathcal{O}(1)$.
- `a.copy()` fait une copie de la liste a ; la complexité de cette opération est en $\mathcal{O}(n)$.
- Si f est une fonction (que l'on supposera de complexité $\mathcal{O}(1)$), la syntaxe `[f(x) for in a]` permet de créer une nouvelle liste, similaire à la liste b résultant de l'exécution du code suivant :

```
b = []
for x in a:
    b.append(f(x))
```

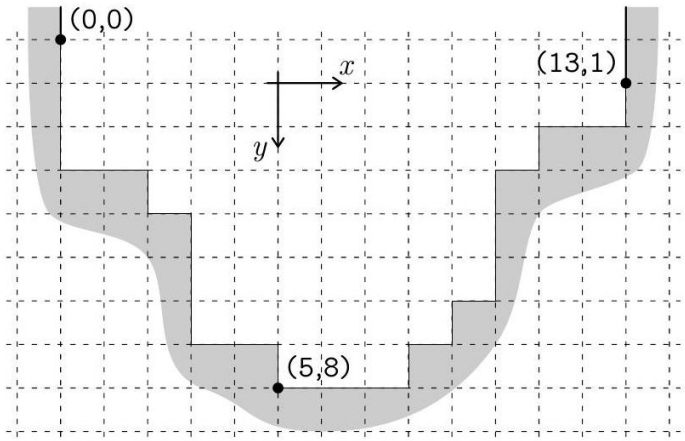
  La complexité de cette création de liste est en $\mathcal{O}(n)$.

On fera attention à éviter les effets de bord : sauf lorsque cela est explicitement demandé dans l'énoncé de la question, les fonctions proposées ne devront pas modifier les paramètres qui lui sont passés en argument.

Aucune justification d'un algorithme ou de sa complexité ne devrait excéder 10 lignes.

Le problème.

Nous allons déterminer le remplissage d'une grotte lors d'une inondation alimentée par une source d'eau localisée quelque part dans la grotte. La grotte considérée sera bi-dimensionnelle et décrite par le profil de son fond. On supposera définies quatre constantes globales `H = "H", B = "B" , G = "G"` et `D = "D"` représentant les quatre directions verticales (Haut et Bas) et horizontales (Gauche et Droite). Le profil de la grotte sera donné sous la forme d'une suite de pas horizontaux ou verticaux de longueur 1, encodée sous la forme d'une liste composée des constantes H, B, G et D, pour les quatre directions Haut, Bas, Gauche et Droite. L'origine du profil sera toujours le point $(0,0)$. On considérera toujours que le profil de la grotte se prolonge à gauche et à droite par deux murs verticaux infinis (B^∞ et H^∞). Dans tout le sujet, on supposera également que le profil contient toujours au moins un pas D vers la Droite. La figure 1 donne l'exemple d'une grotte et de son encodage par une liste :




```
[ B,B,B,D,D,B,D,B,B,B,D,D,B,D,
  D,D,H,D,H,D,H,H,H,D,H,D,D,H  ]
```

FIGURE 1 - UNE GROTTE ET SON PROFIL.

Partie I : Validité d'un profil

On dira qu'un profil est *sans rebroussement* s'il ne contient pas de pas qui revienne immédiatement sur le pas précédent, par exemple pas de ..., G, D , ...
Étant donné les conditions aux bords, le profil d'une grotte sans rebroussement ne commence pas par H et ne finit pas par B.

Question 1 Écrire une fonction `est_sans_rebroussement(g)` qui prend en argument la liste `g` décrivant un profil et renvoie `True` si et seulement si le profil est sans rebroussement, et `False` sinon.

Une *vallée* est une grotte dont le profil est sans rebroussement et commence par descendre en ne faisant que des pas vers le Bas ou la Droite, puis remonte en ne faisant que des pas vers le Haut ou la Droite jusqu'à son point d'arrivée (la direction Gauche est en particulier interdite). La grotte de la figure 1 est une vallée.

Question 2 Écrire une fonction `est_une_vallee(g)` qui prend en argument la liste `g` décrivant un profil et renvoie `True` si et seulement si le profil est une vallée, et `False` sinon.

On considère désormais que les axes des x et des y pointent respectivement vers la Droite et le Bas. On rappelle que le profil d'une grotte a pour origine la position $(0,0)$.

Question 3 Écrire une fonction `voisin (x, y, d)` qui prend en arguments deux entiers `x, y` et une direction `d` $\in$ `{H, B, G, D}`, et renvoie le couple de coordonnées du voisin du point `(x, y)` dans la direction `d`.

Question 4 Écrire une fonction `liste_des_points(g)` qui prend en argument la liste `g` décrivant un profil et renvoie la liste des coordonnées $[(x_0,y_0),\ldots,(x_n,y_n)]$ des points de l'origine à l'arrivée du profil.

On dira qu'un profil est *simple* s'il ne repasse pas par le même point.

Question 5 Écrire une fonction `est_simple(g)` qui prend en argument la liste `g` décrivant un profil et renvoie `True` si et seulement si le profil est simple. Expliciter sa complexité.

Partie II : Vallée

Dans cette partie, nous considérerons que les profils sont toujours de type "vallée".
Le *fond* d'une vallée est son point le plus à gauche parmi ses points les plus bas. Le fond de la vallée de la figure 2 page suivante a pour coordonnées $(5,8)$.

Question 6 Écrire une fonction `fond(v)` qui renvoie les coordonnées (x,y) du fond de la vallée encodée par la liste des directions `v`.

On considère à présent qu'au temps $t=0$, une source d'eau *située au fond de la vallée*, commence à couler avec un débit constant et à remplir la vallée. L'objectif de cette partie est de calculer quelle sera la hauteur de l'eau dans la vallée à chaque instant t. On considérera que le débit de la source est unitaire, c'est-à-dire d'une unité de surface (un carreau) par unité de temps. La figure 2 indique le niveau de l'eau à différentes dates t dans la vallée de la figure 1 :

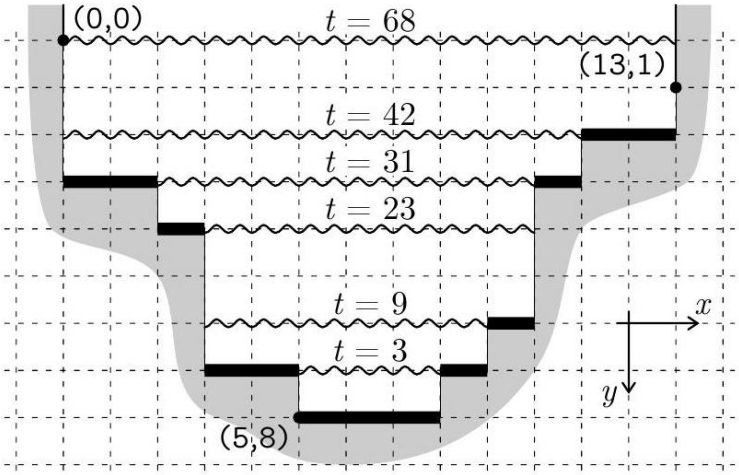


FIGURE 2 - REMPLISSAGE D'UNE VALLÉE.

On appelle *plateau* tout segment horizontal maximal du profil de la vallée. Un plateau est défini par le triplet (x_0,x_1,y) où $x_0<x_1$ sont les abscisses de ses deux extrémités et y est leur ordonnée. La vallée de la figure 2 possède exactement 8 plateaux, indiqués en gras sur la figure : `(0,2,3)`, `(2,3,4)`, `(3,5,7)`, `(5,8,8)`, `(8,9,7)`, `(9,10,6)`, `(10,11,3)`, `(11,13,2)`.

Question 7 Écrire une fonction `plateaux(v)` de complexité linéaire qui renvoie la liste des triplets correspondant aux plateaux de la vallée encodée par la liste `v`.

Remarquons que si l'on trie les plateaux d'une vallée du plus profond au moins profond (par y décroissants), on obtient une décomposition du volume intérieur de la vallée en rectangles. Ces rectangles sont délimités verticalement par les ordonnées consécutives des plateaux et horizontalement par les abscisses des extrémités des plateaux. La vitesse de montée des eaux est constante à l'intérieur de chaque rectangle et vaut exactement $\frac{1}{w}$ où w est la largeur du rectangle. L'eau met donc un temps hw à remplir chaque rectangle de taille $w\times h$. Dans le cas de la vallée illustrée ci-dessus la liste des tailles (w,h) des rectangles ainsi obtenus est, de bas en haut : `[(3,1), (6,1), (7,2), (8,1), (11,1), (13,-1)]` où la valeur -1 de la dernière hauteur signifie que ce dernier rectangle est de hauteur infinie.

Question 8 Écrire une fonction `decomposition_en_rectangles(v)` de complexité linéaire qui renvoie la liste des tailles des rectangles, triés de bas en haut, décomposant le volume intérieur d'une vallée encodée par la liste `v`. Justifier le bon fonctionnement de votre algorithme.

Question 9 Écrire une fonction `hauteur_de_l_eau(t,v)` qui pour tout nombre flottant $t\geqslant 0.0$, renvoie la hauteur de l'eau (mesurée depuis le fond) dans une vallée encodée par la liste `v`.

Partie III : Grottes à ciel ouvert

Une grotte est dite *à ciel ouvert* si son profil est simple et ne contient aucun pas vers la Gauche.
Nous dirons que le profil d'une grotte à ciel ouvert est *normalisé* si le point à la fin du profil est situé à la même profondeur que l'origine, 0, et si tous les autres points du profil sont à une profondeur au moins égale à 1 .
La figure 3 présente deux profils d'une même grotte à ciel ouvert : l'un normalisé (à droite) et l'autre non (à gauche).

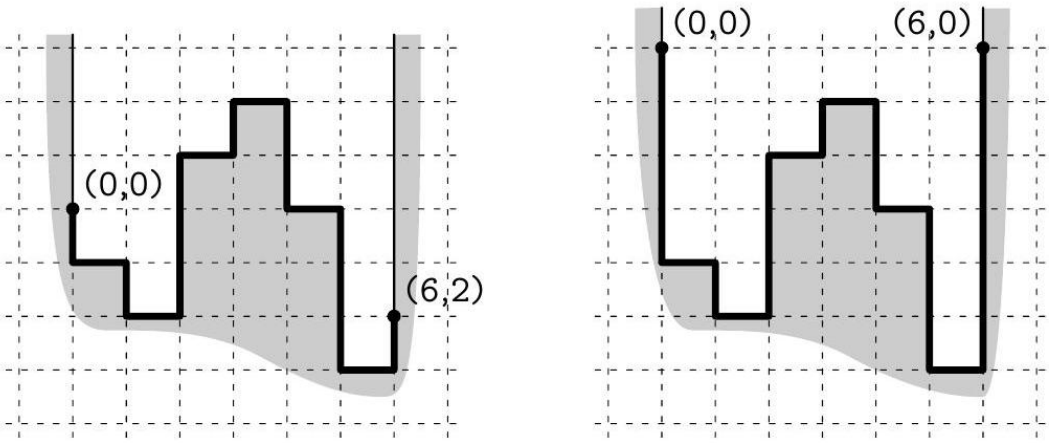


Figure 3 - Deux profils, non-normalisé (à gauche) et normalisé (à droite), d'une même grotte à ciel ouvert.

On suppose désormais que les profils seront tous à ciel ouvert et normalisés jusqu'à la fin de cette partie. Remarquons qu'un profil normalisé contient exactement le même nombre de pas B que de pas H. Cette propriété sera utile pour le bon déroulement des algorithmes ci-dessous.

Pour déterminer l'ordre de remplissage de la grotte, nous allons procéder comme précédemment en la découpant en rectangles, sauf que cette fois-ci, pour simplifier, tous les rectangles de la décomposition seront de hauteur 1 (sauf le dernier qui est de hauteur infinie).

Dans le cas d'une grotte à ciel ouvert, les rectangles qui se remplissent les uns après les autres ne sont plus les uns au-dessus des autres mais organisés hiérarchiquement : chaque rectangle qui n'est pas au fond de la grotte est le "parent" d'un ou plusieurs rectangles "enfants" au-dessous de lui que l'on liste de gauche à droite. La figure 4 ci-dessous montre la structure hiérarchique parent-enfant pour les 12 rectangles composant la grotte à ciel ouvert décrite par un profil normalisé :

```
[ B,B,B,D,H,D,B,B,D,H,H,H,D,B,B,D,H,D,B,B,D,H,D,B,D,H,H,D,B,B,D,H,H,H,H ]
```

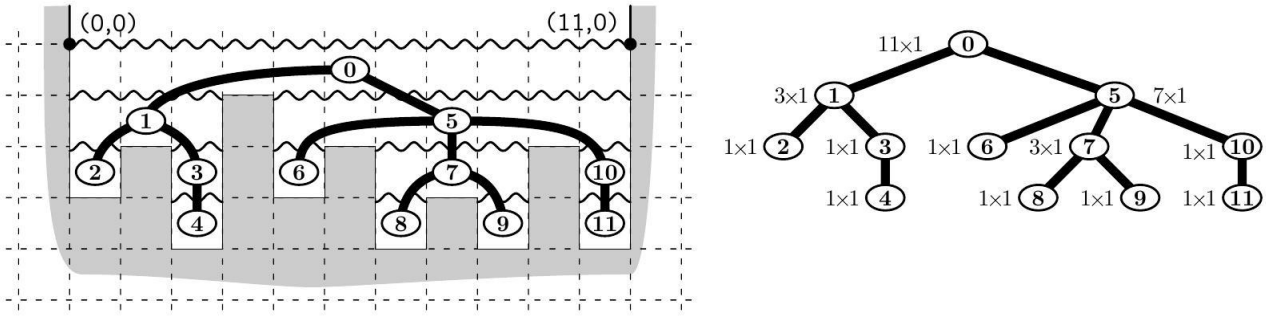


Figure 4 - La structure hiérarchique des rectangles.

Cette structure hiérarchique sera encodée par 4 listes `origine, largeur, parent, enfants`, de la façon suivante :

- les n rectangles seront numérotés de 0 à $n-1$;
- `origine[i]` contiendra le couple d'entiers correspondant aux coordonnées du coin inférieur gauche du rectangle n° i
- `largeur[i]` contiendra la largeur du rectangle n° i ;
- `parent[i]` contiendra le numéro du rectangle parent du rectangle n° i (ou -1 si c'est le rectangle au sommet de la hiérarchie) ;
- `enfants[i]` contiendra la liste des numéros de gauche à droite des rectangles enfants du rectangle n° i (cette liste sera vide, `[]`, si ce rectangle n'a pas d'enfants).

Voici les valeurs de ces quatre listes pour la grotte de la figure 4 :

```
origine = [(0,1),(0,2),(0,3),(2,3),(2,4),(4,2),(4,3),(6,3),(6,4),(8,4),(10,3),(10,4)]
largeur = [11, 3, 1, 1, 1, 7, 1, 3, 1, 1, 1, 1]
parent = [-1, 0, 1, 1, 3, 0, 5, 5, 7, 7, 5, 10]
enfants = [[1,5], [2,3], [], [4], [], [6,7,10], [], [8,9], [], [], [11], []]
```

Nous allons dans un premier temps construire cette structure hiérarchique, puis nous l'utiliserons pour calculer le niveau de l'eau dans les différentes parties en fonction de la position de la source et du temps.

Algorithme de décomposition en rectangles.

L'algorithme procède en parcourant le profil (normalisé) de la grotte une seule fois en partant de l'origine. Tout au long de l'algorithme, on maintient une liste `pile` qui contient les numéros des rectangles ouverts dont on connait l'origine mais pas encore la largeur et qui peuvent donc avoir des enfants :

- Au départ : toutes les listes `pile, origine, largeur, parent, enfants` sont vides.
- Tout au long de l'algorithme, on maintient les coordonnées `(x, y)` du point où nous en sommes sur le profil.
- À chaque fois que le pas du profil est `B` : on crée un nouveau rectangle dont on stocke l'origine dans `origine`, dont on met la largeur temporairement à -1 (car on ne la connaît pas encore), dont on initialise la liste des enfants à vide `[]`, et dont le parent est le numéro du rectangle au bout de la liste `pile` (ou -1 si `pile` est vide) ; on l'ajoute à la liste des enfants de son parent, puis on rajoute le numéro de ce rectangle nouvellement "ouvert" au bout de la liste `pile` des rectangles ouverts.
- À chaque fois que le pas du profil est `H` : on "ferme" le rectangle qui se trouve au bout de liste `pile` (qui contient les rectangles actuellement ouverts). Pour cela, on met à jour sa largeur en se basant sur la position actuelle et sur son origine stockée dans `origine` ; puis on retire son numéro de la liste `pile`.

La figure 5 page suivante exécute cet algorithme pas à pas sur un exemple, en montrant l'évolution des listes `pile, origine, largeur, parent, enfants` à chaque étape.

Le bon fonctionnement de cet algorithme est garanti par le fait que le profil est normalisé : à chaque ouverture d'un rectangle en suivant un pas `B` (correspondant à son bord gauche), correspond un pas `H` (son bord droit) pour sa fermeture.

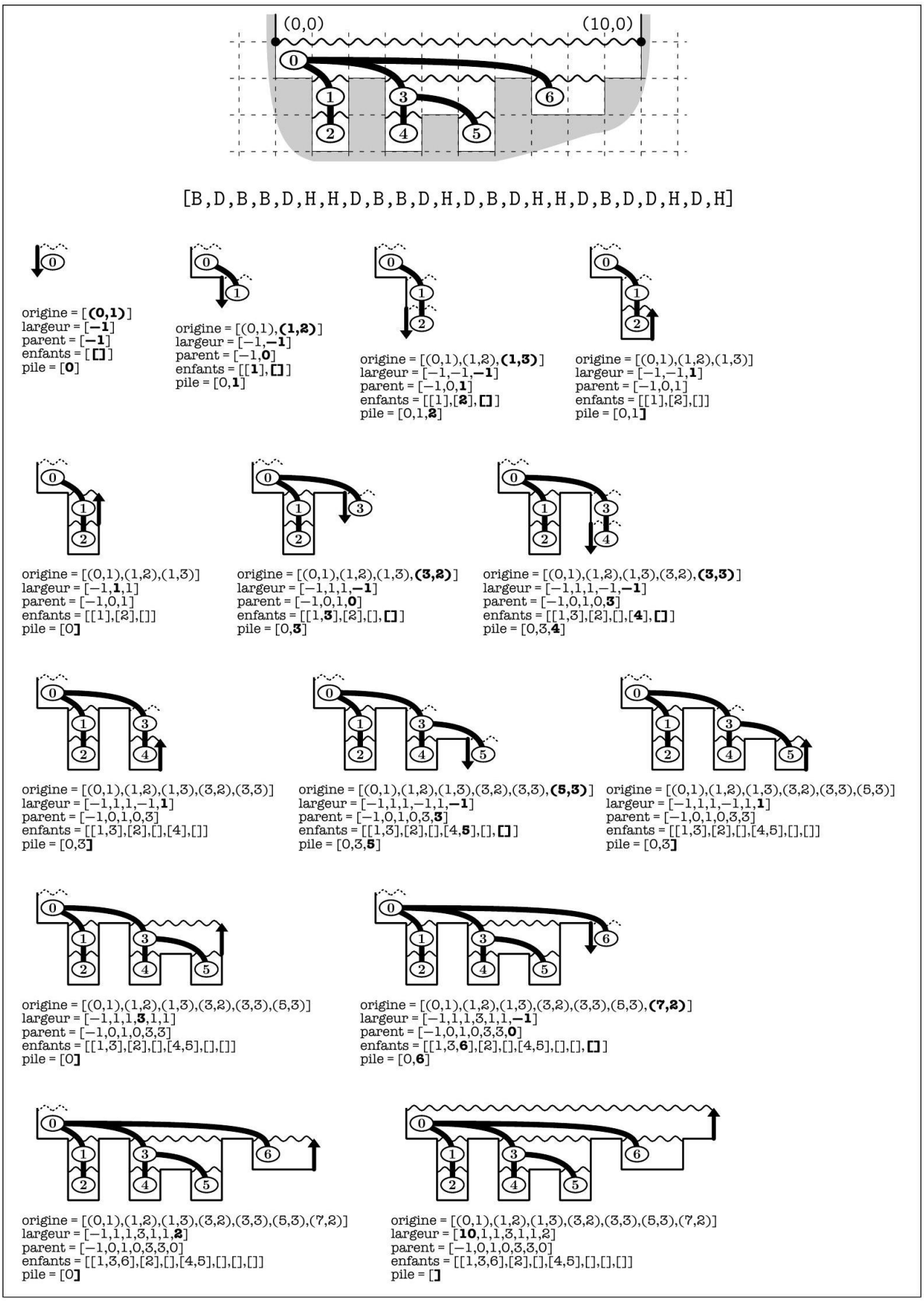


FIGURE 5 - UNE EXÉCUTION DE L'ALGORITHME DE DÉCOMPOSITION EN 7 RECTANGLES DE LA GROTTE À CIEL OUVERT EN HAUT : ON PEUT SUIVRE LES MODIFICATIONS EN GRAS DE CHACUNE DES LISTES PILE, ORIGINE, LARGEUR, PARENT ET ENFANTS À CHAQUE ÉTAPE DU PARCOURS DU PROFIL DE LA GROTTE.

Question 10 Écrire une fonction `hierarchie_rectangles(g)` qui renvoie le quadruplet des quatre listes `(origine, largeur, parent, enfants)` décrivant la hiérarchie de rectangles correspondant au profil normalisé `g`. Donner sa complexité.

Nous allons désormais exploiter cette structure hiérarchique pour calculer l'ordre de remplissage des rectangles de la grotte. Commençons par observer que cet ordre dépend de la position de la source. Sur la figure 6, la source (symbolisée par ▲) est placée soit à l'origine (figure de gauche), soit à l'origine du rectangle au milieu (figure de droite) :

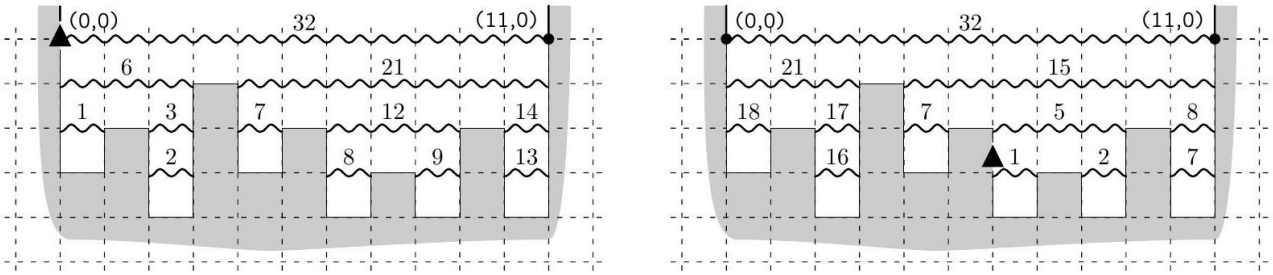


FIGURE 6 - LES DATES ET ORDRE DE REMPLISSAGE DÉPENDENT DE LA POSITION DE LA SOURCE (SYMBOLISÉE PAR ▲).

Les dates de remplissage des différents rectangles sont marquées au-dessus de leur bord supérieur. On constate que ces dates sont non seulement différentes, mais aussi que, lorsque la source est "au milieu" de la grotte, alors, plusieurs rectangles peuvent se remplir simultanément, comme c'est le cas des rectangles remplis entre $t = 5$ et $t = 7$ dans la figure de droite. Cette situation n'est cependant pas possible quand la source est située tout à gauche de la grotte, à la position $(0, 0)$ (admis).

Le cas de la source située à l'origine. On supposera que l'eau s'écoule instantanément verticalement et prend donc un temps nul à dévaler les pentes (comme cela a été supposé dans les deux chronologies de la figure 6). On se place dans le cas où la source est située à l'origine. On admet alors que l'eau remplit les rectangles de gauche à droite, un seul à la fois. On admettra également que chaque rectangle commencera à se remplir une fois que l'ensemble de ses rectangle-enfants seront remplis et que ceux-ci se remplissent l'un après l'autre de gauche à droite.

Question 11 Écrire une fonction `ordre_remplissage_depuis_origine(parent, enfants)` qui prend en entrée les deux listes `parent` et `enfants` décrivant la hiérarchie des rectangles et renvoie la liste des numéros des rectangles dans l'ordre dans lequel ils se remplissent. Donner sa complexité.

Question 12 Écrire une fonction `hauteurs_eau_depuis_origine(t, largeur, parent, enfants)` qui prend en entrée un flottant `t`, et les trois listes `largeur, parent, enfants`, décrivant la hiérarchie des rectangles d'une grotte à ciel ouvert, et renvoie une liste `hauteur` où `hauteur[i]` est la hauteur d'eau dans le rectangle n° i à l'instant t (la hauteur sera donc un flottant entre 0 et 1 sauf pour le dernier rectangle qui est infini et peut donc être rempli à une hauteur arbitrairement grande).
Expliciter sa complexité.

Le cas d'une source à une position arbitraire. Comme nous l'avons vu précédemment, lorsque la source est à une position arbitraire, il est possible que plusieurs rectangles se remplissent simultanément : quand un bassin est plein, l'eau s'écoule alors équitablement des deux côtés, comme illustré sur la figure 6 à droite entre les dates $t = 5$ et $t = 7$.

Question 13 Expliquez pourquoi jamais plus de deux rectangles ne se rempliront simultanément. *Votre réponse ne devrait pas excéder 5 lignes.*

Question 14 Écrire une fonction `volumes_totaux(largeur, parent, enfants)` qui prend en entrée les trois listes `largeur, parent, enfants` décrivant la hiérarchie des rectangles, et qui renvoie une liste `volume` telle que `volume[i]` est la somme des volumes des rectangles descendants du rectangle n° i, i inclus.

Question 15 Décrire un algorithme qui prend en entrée le numéro source du rectangle à l'origine duquel est située la source, les trois listes `largeur, parent, enfants` décrivant la hiérarchie des rectangles, et qui renvoie une liste `hauteur` telle que `hauteur[i]` est la hauteur d'eau présente dans le rectangle n° i à l'instant t. On pourra utiliser les procédures définies ci-dessus. On ne demande pas l'écriture d'un programme mais la présentation argumentée d'une solution algorithmique à ce problème.

FIN DU PROBLÈME 1

PROBLÉME 2 (extrait de MINES MP 2012, option Informatique

- *Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.*
- *Tout résultat fourni dans l'énoncé peut être utilisé pour les questions ultérieures même s'il n'a pas été démontré.*
- *Il ne faut pas hésiter à formuler les commentaires qui semblent pertinents même lorsque l'énoncé ne le demande pas explicitement.*

I. Préliminaires

Le langage de programmation sera obligatoirement `Python`.

Lorsque le candidat écrira une fonction ou une procédure, il pourra faire appel à une autre fonction ou procédure définie dans les questions précédentes. Enfin, si les paramètres d'une fonction ou d'une procédure à écrire sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction ou de cette procédure de tester si les hypothèses sont bien vérifiées.

Dans les énoncés du problème, un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain (par exemple `n`).

Un *graphe* G est défini par deux ensembles X et E. L'ensemble X est un ensemble fini d'éléments appelés *sommets*. L'ensemble E est un ensemble de paires de sommets. Un élément $\{x,y\}$ de E est appelé *arête* de G; x et y sont les *extrémités* de l'arête. L'*ordre* d'un graphe G est le nombre de sommets de G. Un graphe est représenté par un dessin où des cercles représentent les sommets et un trait joignant deux cercles représente une arête composée des deux sommets correspondant aux cercles. Si $\{x,y\}$ est une arête de G, on dit que x et y sont *voisins*. Le *degré* d'un sommet x est le nombre de voisins de x.

On dit que deux arêtes d'un graphe G sont *incidentes* si elles ont une extrémité en commun. On appelle *couplage dans* G un ensemble d'arêtes de G deux à deux non incidentes.

Un graphe G est dit *biparti* si on peut partitionner son ensemble de sommets X en deux sous-ensembles A et B $(A \neq \varnothing,\ B \neq \varnothing,\ A \cup B = X,\ A \cap B = \varnothing)$ de sorte que toute arête ait une extrémité dans A et une extrémité dans B. Si les ensembles A et B ont même cardinal, on dit qu'il s'agit d'un *graphe biparti équilibré*. Dans tout le problème, on ne considère que des graphes bipartis équilibrés. On note n le cardinal commun aux ensembles A et B ; l'ordre du graphe est donc égal à $2n$. On suppose que l'on a toujours $n \geqslant 1$. Les sommets de A sont numérotés de 0 à $n-1$ et nommés $0_A, 1_A, 2_A, \ldots, (n-1)_A$; les sommets de B sont numérotés de 0 à $n-1$ et nommés $0_B, 1_B, 2_B, \ldots, (n-1)_B$. Une arête de G est toujours écrite en mettant d'abord l'extrémité qui est dans A puis celle qui est dans B.

On représente les graphes bipartis équilibrés par des schémas comme on peut le voir dans la figure 1 avec le graphe G_0, en représentant les sommets de A à gauche et les sommets de B à droite.

Pour G_0, n vaut 4 et l'ordre de G_0 vaut 8.

Les arêtes de G_0 sont :
$\{0_A,0_B\}, \{0_A,1_B\}, \{0_A,2_B\}, \{1_A,3_B\}$
$\{2_A,0_B\}, \{2_A,1_B\}, \{2_A,2_B\}, \{2_A,3_B\}, \{3_A,3_B\}$.

Le sommet 0_A est de degré 3.

Le sommet 1_A est de degré 1.

Le sommet 2_A est de degré 4.

Le sommet 3_A est de degré 1.

Les sommets $0_B, 1_B$ et 2_B sont de degré 2.

Le sommet 3_B est de degré 3.

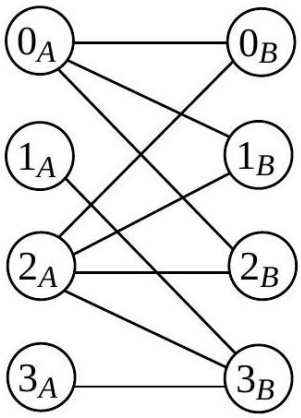


FIGURE 1 – Le graphe G_0

Dans le graphe G_0, les arêtes $\{0_A, 0_B\}$ et $\{2_A, 3_B\}$ étant non incidentes, elles forment un couplage, nommé C_0, dont les arêtes sont dessinées en gras ci-contre ; on dit alors que dans ce couplage :

- le sommet 0_A est couplé au sommet 0_B, et réciproquement ;
- le sommet 2_A est couplé au sommet 3_B, et réciproquement ;
- les sommets $1_A, 3_A, 1_B$ et 2_B sont non couplés.

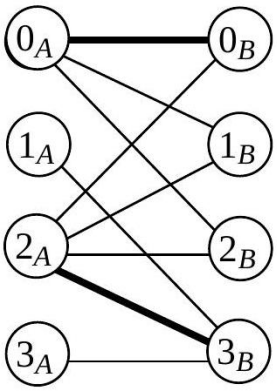


FIGURE 2 – Le graphe G_0 et le couplage C_0

Le cardinal d'un couplage est le nombre d'arêtes de celui-ci ; par exemple le cardinal de C_0 vaut 2.

Première partie : généralités

Question 1 Exhiber un couplage de cardinal 3 dans G_0, puis indiquer s'il existe dans G_0 un couplage de cardinal 4. Justifier la réponse.

Un graphe biparti équilibré d'ordre $2n$ est représenté par sa *matrice d'incidence* ; c'est une matrice carrée de dimension $n \times n$ dont les lignes correspondent aux éléments de A et les colonnes aux éléments de B. Les cases de cette matrice sont indicées par (i, j) avec $0 \leqslant i \leqslant n-1$ et $0 \leqslant j \leqslant n-1$ et contiennent des valeurs booléennes : la case d'indice (i, j) contient la valeur `True` si $\{i_A, j_B\}$ est une arête du graphe ; elle contient la valeur `False` dans le cas contraire. Le graphe G_0 ci-dessus est donc représenté par la matrice suivante :

i \ j	0	1	2	3
0	vrai	vrai	vrai	faux
1	faux	faux	faux	vrai
2	vrai	vrai	vrai	vrai
3	faux	faux	faux	vrai

et codé en `Python` par la liste de listes :

```
G0 = [  [ True, True, True, False],
        [False, False, False, True],
        [True, True, True, True],
        [False, False, False, True] ]
```

Un couplage est représenté par une liste de longueur n. Pour i vérifiant $0 \leqslant i \leqslant n-1$, si le sommet i_A est couplé avec le sommet j_B, la case d'indice i contient la valeur j ; si le sommet i_A n'est pas couplé, la case d'indice i contient une valeur égale à -1 . Le couplage C_0 de G_0, formé des arêtes $\{0_A, 0_B\}$ et $\{2_A, 3_B\}$, est ainsi représenté par la liste :

```
C0 = [0, -1, 3, -1]
```

Enfin, une arête a est codée par un couple `a` de deux entiers, avec `a[0]` dans A et `a[1]` dans B.

Question 2 Soit G un graphe biparti équilibré d'ordre $2n$. On considère une liste C d'entiers de longueur n et contenant dans ses cases indicées de 0 à $n-1$ soit la valeur -1 , soit une valeur comprise entre 0 et $n-1$. Il s'agit de savoir si cette liste C représente ou non un couplage dans G.
Écrire en `Python` une fonction `verifie` telle que :

– si `G` est une matrice codant le graphe G,
– si `C` est une liste codant C,

alors `verifie(G, C)` renvoie `True` si la liste C représente un couplage dans G et `False` sinon.

Indiquer la complexité de la fonction `verifie`.

Question 3 On considère une liste C, de longueur n, représentant un couplage d'un graphe G.

Écrire en `Python` une fonction `cardinal` telle que, si `C` est une liste représentant un couplage, alors `cardinal(C)` renvoie le cardinal de ce couplage.

Indiquer la complexité de la fonction `cardinal`.

Deuxième partie : un algorithme pour déterminer un couplage maximal

On dit qu'un couplage C dans un graphe G est *maximal* si toute arête de G n'appartenant pas à C est incidente à au moins une arête de C. Par exemple, le couplage C_0 de G_0 est maximal. Un couplage maximal de G n'est pas forcément de cardinal maximum parmi les couplages de G. On cherche à concevoir un algorithme qui détermine un couplage maximal dans un graphe biparti équilibré G.

L'algorithme, nommé *algo_approche*, est le suivant :

- on commence avec un couplage vide C ;
- tant que G possède au moins une arête :
  - on choisit une arête a de G telle que la somme des degrés des extrémités soit minimum ;
  - on ajoute l'arête a au couplage C ;
  - on retire de G l'arête a et toutes les arêtes incidentes à a.

On admettra que le résultat est, par construction, un couplage maximal.

Question 4 Appliquer, en détaillant les étapes, *algo_approche* au graphe G_0 (voir la figure 1).

On considère par la suite le graphe biparti équilibré G_1 d'ordre 12 représenté sur la figure 3 :

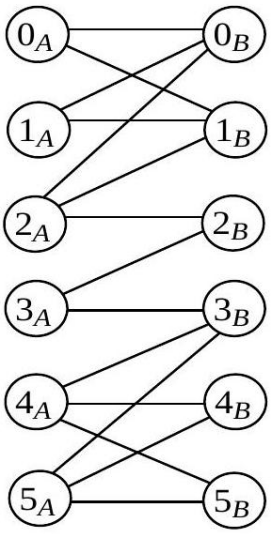


FIGURE 3 – Le graphe G_1

Question 5 On applique *algo_approche* au graphe G_1. Déterminer la première arête a_1 choisie par *algo_approche* ; tracer le graphe obtenu après suppression de a_1 et des arêtes incidentes à a_1. Montrer que le couplage obtenu par *algo_approche* est de cardinal au plus 5 et indiquer s'il est de cardinal maximum parmi les couplages de G_1.

Question 6 Soit G un graphe biparti équilibré d'ordre $2n$. Il s'agit d'écrire une fonction *arete_min* qui détermine une arête de G dont la somme des degrés des extrémités soit minimum.

Écrire en `Python` une fonction `arete_min(G)`, où `G` est une matrice représentant le graphe G, qui renvoie `False, None` si le graphe ne possède pas d'arête, et qui renvoie `True, a` sinon, où `a` est un couple représentant une arête qui réalise le minimum demandé.

Indiquer la complexité de cette fonction.

Question 7 Écrire en `Python` une fonction `supprimer` qui, étant donnés comme arguments une matrice `G` représentant le graphe G et un couple `a` représentant une arête a, modifie `G` pour que, après modifications, `G` représente le graphe obtenu à partir de G en supprimant a et toutes les arêtes incidentes à a.

Indiquer la complexité de la fonction `supprimer`.

Question 8 Écrire la fonction `algo_approche(G)`, où `G` est une matrice représentant le graphe G, qui, à partir d'une *copie* de `G`, applique l'algorithme *algo_approche* au graphe G et renvoie une liste représentant le couplage maximal obtenu.

Troisième partie : recherche exhaustive d'un couplage de cardinal maximum

Question 9 Soit G un graphe biparti équilibré d'ordre $2n$, représenté par sa matrice d'adjacence `G`.

Écrire une fonction `une_arete`, d'argument `G`, qui recherche une arête quelconque de G : si G n'a pas d'arête, `une_arete(G)` renvoie `False, None`, sinon la fonction renvoie `True, a`, où `a` est un couple représentant la première arête rencontrée.

Question 10 On cherche à établir un algorithme récursif, nommé *meilleur_couplage*, qui permette de déterminer un couplage de cardinal maximum dans un graphe biparti équilibré. Le principe est le suivant.

Si le graphe courant ne contient aucune arête, le cardinal maximum d'un couplage est 0 et aucun sommet n'est couplé.
Dans le cas contraire, l'algorithme considère une arête quelconque a du graphe courant et recherche successivement :

- un couplage de cardinal maximum parmi les couplages du graphe courant ne contenant pas a ;
- un couplage de cardinal maximum parmi les couplages du graphe courant contenant a.

L'algorithme déduit alors un couplage de cardinal maximum.

Écrire en `Python` une fonction récursive `meilleur_couplage`, telle que, étant donnée une matrice `G` représentant le graphe G, `meilleur_couplage(G)` renvoie une liste représentant un couplage de cardinal maximum dans G.

Quatrième partie : l'algorithme hongrois

On considère un graphe biparti équilibré G et un couplage C.
Une *chaîne* de G est une suite $x_0, x_1, \ldots, x_k, \ldots, x_p$ $(p \geqslant 0)$ de sommets distincts telle que, pour k compris entre 0 et $p-1$, $\{x_k, x_k+1\}$ est une arête de G. Le sommet x_0 s'appelle l'*origine* de la chaîne et le sommet x_p l'*extrémité* de la chaîne.

Une chaîne $x_0, x_1, \ldots, x_k, \ldots, x_p$ de G, avec $p \geqslant 0$, est dite *alternée relativement à* C si :

- pour tout indice k pair, x_k est dans A,
- pour tout indice k impair, x_k est dans B
- le sommet x_0 n'est pas couplé,
- pour tout entier i vérifiant $0 \leqslant 2i \leqslant p-1$, l'arête $\{x_{2i}, x_{2i+1}\}$ n'appartient pas à C,
- pour tout entier i vérifiant $2 \leqslant 2i \leqslant p$, l'arête $\{x_{2i-1}, x_{2i}\}$ appartient à C.

Autrement dit :

- l'origine de la chaîne est dans A et n'est pas couplée,
- la première arête de la chaîne n'est pas dans C, la deuxième est dans C, la troisième n'est pas dans C et ainsi de suite.

Une chaîne $x_0, x_1, \ldots, x_p$ alternée relativement au couplage C est dite *chaîne alternée augmentante relativement à* C si on a $p \geqslant 1$ et si de plus x_p n'est pas couplé, ce qui entraîne que x_p est dans B. Par exemple, dire qu'une chaîne $x_0, x_1, x_2, x_3, x_4, x_5$ constitue une chaîne alternée augmentante relativement à un couplage C signifie que :

- x_0, x_2 et x_4 sont des sommets de A, x_1, x_3 et x_5 sont des sommets de B ;
- $\{x_0, x_1\}, \{x_1, x_2\}, \{x_2, x_3\}, \{x_3, x_4\}, \{x_4, x_5\}$ sont des arêtes de G ;
- x_0 n'est pas couplé dans C, x_5 n'est pas couplé dans C ;
- x_1 est couplé avec x_2, x_2 n'est pas couplé avec x_3 et x_3 est couplé avec x_4.

Question 11

On considère le graphe G_1 et le couplage C_1 constitué des arêtes $\{0_A,0_B\}$, $\{1_A,1_B\}$, $\{3_A,2_B\}$, $\{4_A,3_B\}$, $\{5_A,5_B\}$ représentées sur la figure 4 en gras.

Après avoir indiqué le seul sommet de A qui puisse être l'origine d'une chaîne alternée augmentante relativement à C_1 et le seul sommet de B qui puisse être l'extrémité d'une chaîne alternée augmentante relativement à C_1, déterminer une chaîne alternée augmentante relativement à C_1.

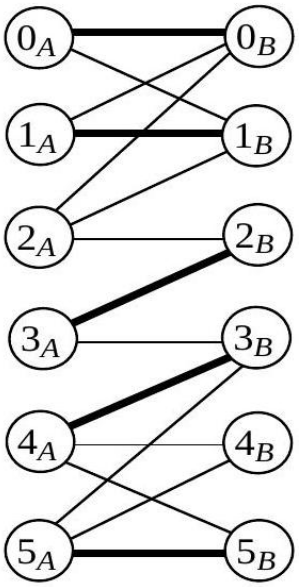


FIGURE 4 – Le graphe G_1 et le couplage C_1

Question 12 On considère un graphe biparti équilibré G et un couplage C dans G. Montrer que si il existe une chaîne alternée augmentante relativement à C, alors il existe dans G un couplage dont le cardinal est égal au cardinal de C augmenté de 1 .

On admet le théorème suivant[1] : *un couplage C d'un graphe biparti équilibré G est de cardinal maximum si et seulement s'il n'existe pas dans G de chaîne alternée augmentante relativement à C.*

Soit G un graphe biparti équilibré. L'*algorithme hongrois* s'appuie sur le théorème ci-dessus et détermine un couplage de cardinal maximum dans G. L'algorithme débute avec un couplage C de cardinal nul ; tant qu'il existe une chaîne augmentante relativement à C, l'algorithme modifie C pour incrémenter de 1 le cardinal du couplage en utilisant une telle chaîne augmentante.

La suite du problème a pour objectif de programmer cet algorithme. On commence par étudier la recherche d'une chaîne alternée augmentante.

On considère un graphe biparti équilibré G et un couplage C dans G. On va chercher s'il existe une chaîne alternée augmentante relativement à C. Pour cela, on recherche les sommets qui sont extrémités de chaînes alternées en procédant de proche en proche à partir des sommets de A non couplés.

Un sommet y est dit *atteint* si une chaîne alternée relativement à C et d'extrémité y est mise en évidence. Au départ, tous les sommets non couplés de A (un tel sommet est extrémité d'une chaîne alternée réduite à ce sommet) sont considérés comme atteints ; aucun autre sommet n'est considéré comme atteint. Un sommet y non encore atteint peut être atteint à partir d'un de ses voisins x déjà atteint si on a :

- soit y est dans B (et donc x est dans A) et l'arête $\{x,y\}$ n'est pas dans le couplage C ;
- soit y est dans A (et donc x est dans B) et l'arête $\{y,x\}$ est dans le couplage C.

On utilise des *marques attribuées aux sommets*. Ces marques sont des entiers initialisés à -1 pour tous les sommets. Lorsqu'un sommet y est atteint à partir d'un sommet x, la marque de y devient égale au numéro de x (on rappelle que le *numéro* d'un sommet i_A ou d'un sommet i_B vaut i) ; x est alors l'avant-dernier sommet dans la chaîne alternée $Ch(y)$ d'extrémité y mise en évidence. La chaîne $Ch(y)$ peut être retrouvée à l'envers, de proche en proche, grâce aux marques ; dans cette chaîne, seule l'origine porte une marque de valeur -1 .

Si un sommet non couplé y de B est atteint, $Ch(y)$ est une chaîne alternée augmentante relativement à C.

Dans le cas où simultanément :

- il n'y a plus de sommet non encore atteint qui puisse être atteint,
- aucun sommet non couplé de B n'est atteint,

on admet qu'il n'existe pas de chaîne alternée augmentante relativement à C.

1. Berge, 1957

Remarque : les valeurs des marques peuvent dépendre de l'ordre dans lequel on atteint les sommets, sans que cela n'ait d'importance pour la suite du problème.

Question 13

On considère le graphe G_1 et un couplage nommé C'_1 constitué des arêtes $\{0_A, 0_B\}, \{2_A, 1_B\}, \{3_A, 2_B\}, \{4_A, 4_B\}, \{5_A, 5_B\}$, en gras sur la figure 5.

Certains sommets ont été atteints ; sur la figure 5, les marques attribuées sont portées à côté des sommets, les sommets atteints sont encadrés.

Utiliser les marques pour reconstituer la chaîne alternée arrivant dans le sommet 3_B et correspondant aux marques. Indiquer s'il s'agit d'une chaîne alternée augmentante relativement à C'_1.

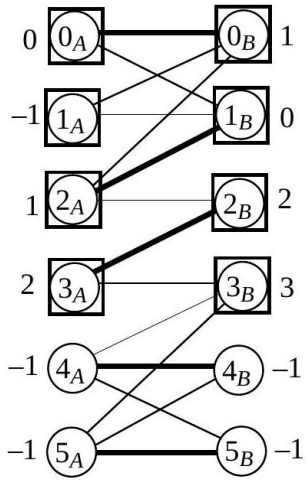


FIGURE 5 – Le graphe G_1 et le couplage C'_1

On considère quatre listes :

- Une liste nommée `C` codant un couplage C.
- Une liste nommée `R` (pour réciproque) de longueur n ; soit j vérifiant $0 \leqslant j \leqslant n-1$; si le sommet j_B n'est pas couplé dans C, la case d'indice j de la liste `R` contient la valeur -1 ; si le sommet j_B est couplé avec le sommet i_A, la case d'indice j de la liste `R` comporte la valeur i.
- Une liste nommée `mA` de longueur n servant à contenir les marques des sommets de A.
- Une liste nommée `mB` de longueur n servant à contenir les marques des sommets de B.

Question 14 On suppose qu'une chaîne $Ch(x_p) = x_0, x_1, \ldots, x_p$ alternée augmentante relativement à un couplage C a été déterminée et codée grâce aux marques. D'après la question **12**, il existe un couplage de cardinal égal à celui de C augmenté de 1 .

Écrire en `Python` une fonction `actualiser` qui reçoit en paramètres les quatre listes décrites ci-dessus ainsi que le numéro de x_p et transforme alors les listes `C` et `R` pour qu'elles correspondent à un couplage, obtenu à partir de C et de $Ch(x_p)$, dont le cardinal est celui du couplage C augmenté de 1.

Question 15

On considère le graphe G_2 ci-contre et un couplage, nommé C_2, constitué des arêtes $\{0_A, 0_B\}, \{2_A, 2_B\}, \{3_A, 3_B\}, \{4_A, 4_B\}$ en gras sur la figure 8.

Recopier la figure, encadrer tous les sommets qui peuvent être atteints et préciser à côté des sommets les marques obtenues.
Indiquer s'il existe dans G_2 une chaîne alternée augmentante relativement à C_2.

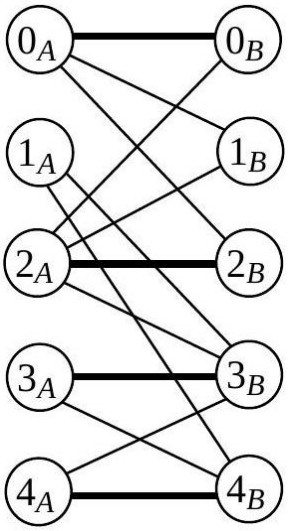


FIGURE 6 – Le graphe G_2 et le couplage C_2

Indication : on procédera de proche en proche à partir du seul sommet non couplé de A, c'est-à-dire à partir du sommet 1_A. L'ordre dans lequel on atteint les sommets n'a pas d'importance.

On suppose donnés un graphe biparti équilibré G et un couplage C dans G. Les chaînes alternées considérées dans la suite sont toujours des chaînes alternées relativement à C. Les questions **16** et **17** ont pour objectif de programmer un algorithme qui cherche une chaîne alternée augmentante en atteignant les sommets de proche en proche à partir des sommets non couplés de A.

Question 16 On suppose que certains sommets de G peuvent déjà avoir été atteints et les marques calculées en conséquence. On note x un sommet de A ou de B déjà atteint.
On définit deux nouvelles fonctions, les fonctions *chercherA* et *chercherB*.
Appliquée au sommet x, appelé *sommet de départ*, la fonction *chercherA* (si x est dans A) ou la fonction *chercherB* (si x est dans B) détermine des sommets non encore atteints qui peuvent être atteints, de voisin en voisin, récursivement, à partir de x, modifie les marques de ces sommets nouvellement atteints et s'arrête dans les cas suivants :

- un sommet y de B non couplé a été atteint ; elle renvoie alors le numéro du sommet y ;
- tous les sommets qui peuvent être atteints de voisin en voisin à partir de x l'ont été et aucun sommet non couplé de B n'est atteint ; elle renvoie alors la valeur -1.

Il s'agit d'écrire les deux fonctions *chercherA* et *chercherB* en utilisant une récursivité croisée, chacune des deux fonctions pouvant faire appel à l'autre.

Écrire en `Python` les deux fonctions `chercherA` et `chercherB` ; chacune de ces deux fonctions reçoit en paramètres :

- une matrice `G` représentant le graphe G ;
- les quatre listes décrites plus haut : `C, R, mA, mB` ;
- un entier codant le numéro du sommet de départ de la recherche.

Ces deux fonctions modifient les listes `mA` et `mB` conformément à la description ci-dessus. Elles renvoient le numéro d'un sommet atteint non couplé de B ou la valeur -1 selon le cas.

Question 17 On suppose donnés un graphe biparti équilibré G d'ordre $2n$ et un couplage C dans G. Tous les sommets de G possèdent une marque égale à -1 .
La fonction *chaine_alternee* cherche s'il existe une chaîne alternée augmentante en appliquant la fonction *chercherA* successivement à partir des sommets non couplés de A.

Écrire en `Python` la fonction :`chaine_alternee` telle que :

- si `G` est une matrice codant le graphe G,
- si `C, R, mA, mB` correspondent à la description donnée précédemment, toutes les cases de `mA` et `mB` étant initialisées à -1 ,

alors `chaine_alternee(G, C, R, mA, mB)` renvoie :

- -1 s'il n'existe pas de chaîne alternée augmentante,
- le numéro de l'extrémité d'une chaîne alternée augmentante dans le cas contraire.

De plus, la fonction modifie les listes `mA` et `mB` pour qu'elles contiennent les marques des sommets à la fin de l'exécution de la fonction.

Question 18 Dans cette question, on programme l'algorithme hongrois.

Écrire en `Python` la fonction `algorithme_hongrois` telle que, si `G` est une matrice codant un graphe biparti équilibré G, alors `algorithme_hongrois(G)` renvoie une liste codant le couplage obtenu par l'algorithme hongrois.