

TP Informatique évalué 01 - 2h

Télécharger le fichier TP01_AUTO.py depuis l'arborescence INFORMATIQUE/DS/TP01 EVALUE sur le site de la classe puis travailler directement sur ce fichier en le complétant.

Après avoir vérifié que le script s'exécute correctement, déposer le fichier sur le site de la classe (le nom du fichier n'importe pas).

⚠ Dans tout le script, on utilisera exclusivement la variable locale **noir** pour tester la noirceur d'une case et les variables locales **visite** et **non_visite** pour affecter ou tester l'état visité ou non d'un sommet dans un graphe.

Exercice 1

Dans ce problème, on considère une image constituée de pixels blancs et noirs. On la représente par une liste de listes de 0 et de 1 où les 0 désignent les pixels blancs et les 1 des pixels noirs. Un pixel est représenté par un tuple de ses coordonnées (x, y) avec x le numéro de ligne et y le numéro de colonnes.

```
>>> config
[[1,0,0,1,0,0],
 [0,0,1,0,1,0],
 [0,0,0,1,1,0],
 [0,0,0,0,1,1],
 [0,0,0,0,0,0]]
```

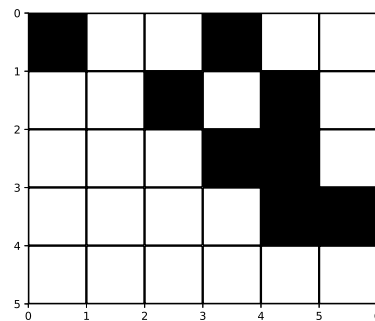


FIGURE 1 – Tâches de couleur noire

On considère que des pixels noirs appartiennent à une même tâche noire si on peut passer de l'un à l'autre par des transitions horizontales ou verticales. Dans l'exemple de la figure 1, les pixels de coordonnées $(1, 4)$, $(2, 3)$, $(2, 4)$, $(3, 4)$, $(3, 5)$ appartiennent à une même tâche. Les pixels de coordonnées $(0, 0)$, $(0, 3)$, $(1, 2)$ constituent des tâches indépendantes. Le graphe associé à cette image est le couple S, A où l'ensemble des sommets S est la liste des pixels noirs et l'ensemble des arêtes A est le dictionnaire où une clé désigne un pixel noir associé à sa valeur qui est la liste, éventuellement vide, des pixels noirs vers lesquels une transition horizontale ou verticale est possible.

1. Écrire une fonction `sommets(config)` d'argument `config` une liste de listes de 0 et 1 qui renvoie la liste des sommets du graphe associé à l'image représentée par `config`.

Par exemple, pour la figure 1, l'appel `sommets(config)` renvoie :

```
>>> sommets(config)
[(0, 0), (0, 3), (1, 2), (1, 4), (2, 3), (2, 4), (3, 4), (3, 5)]
```

Les nombres de lignes et colonnes devront être obtenus directement à partir de l'argument `config`. Pour tester la noirceur d'un pixel, on utilisera la variable locale `noir`. Par exemple pour tester, si le point (i, j) est noir, on écrira `config[i][j]==noir`.

2. Écrire une fonction `aretes(config)` d'argument `config` une liste de listes de 0 et 1 qui renvoie le dictionnaire des arêtes du graphe associé à l'image représentée par `config`.

Par exemple, pour la figure 1, l'appel `sommets(config)` renvoie :

```
>>> aretes(config)
{(0, 0): [], (0, 3): [], (1, 2): [],
 (1, 4): [(2, 4)], (2, 3): [(2, 4)],
 (2, 4): [(1, 4), (3, 4), (2, 3)],
 (3, 4): [(2, 4), (3, 5)], (3, 5): [(3, 4)]}
```

La fonction `aretes` fera appel à la fonction `sommets` (voir fichier `TP01_EVAL.py`). Les autres recommandations énoncées pour la première question valent aussi pour cette question. On sera vigilant à ne pas dégrader inutilement la complexité temporelle de `aretes`.

Exercice 2

On reprend les notations de l'exercice 1. On utilisera un dictionnaire `etat` où les pixels noirs seront des clés et leurs valeurs désignera un état visité ou non. Un pixel noir non visité sera à l'état 0 et un pixel noir visité sera à l'état 1.

1. Écrire une fonction `dfs_rec(S,A,s0,etat)` d'arguments `S`, `A` un graphe associé à une image, `s0` un sommet correspondant à un pixel noir non visité, `etat` un dictionnaire et qui réalise, depuis le sommet `s0`, le parcours de tous les pixels noirs de la tâche contenant `s0` de manière récursive : depuis le sommet `s0`, on balaie l'ensemble des sommets adjacents et quand l'un est non visité, on appelle récursivement `dfs_rec` depuis ce sommet (il s'agit d'un parcours en profondeur). L'état de chaque pixel noir visité voit son état modifié dans le dictionnaire `etat`. Par exemple, on reprend la configuration `config` de l'exercice 1. On initialise une variable `etat` pour avoir tous les pixels noirs non visités :

```
>>> etat
{(0, 0): 0, (0, 3): 0, (1, 2): 0, (1, 4): 0,
 (2, 3): 0, (2, 4): 0, (3, 4): 0, (3, 5): 0}
```

L'appel de `df_rec(S,A,(2,3),etat)` ne renvoie rien mais modifie la variable `etat` :

```
>>> dfs_rec(S,A,(2,3),etat)
>>> etat
{(0, 0): 0, (0, 3): 0, (1, 2): 0,
 (1, 4): 1, (2, 3): 1, (2, 4): 1, (3, 4): 1, (3, 5): 1}
```

Tous les pixels appartenant à la même tâche que le pixel $(2,3)$ ont été marqués comme visités (état égal à 1).

2. Écrire une fonction `nb_taches_dfs(S,A)` d'arguments `S`, `A` un graphe associé à une image qui renvoie le nombre de tâches de l'image. La fonction devra faire appel à la fonction `dfs_rec`. Par exemple, sur la configuration de l'exercice 1, l'appel de `nb_taches_dfs(S,A)` après génération du graphe renvoie :

```

>>> S,A=sommets(config),aretes(config)
>>> nb_taches_dfs(S,A)
4

```

Exercice 3

On reprend les notations des exercices 1 et 2.

1. Écrire une fonction `bfs(S,A,s0,etat)` d'arguments `S`, `A` un graphe associé à une image, `s0` un sommet correspondant à un pixel noir non visité, `etat` un dictionnaire et qui réalise, depuis le sommet `s0`, le parcours en largeur de tous les pixels noirs de la tâche contenant `s0` : on utilise une file et pour chaque sommet qui sort, on le considère visité et on fait entrer ses sommets adjacents dans la file. L'état de chaque pixel noir visité voit son état modifié dans le dictionnaire `etat`. On pourra reprendre l'exemple décrit dans l'exercice 2. On rappelle que pour une file `F` de type `deque`, les méthodes `append` et `popleft` servent respectivement à faire entrer et sortir de la file.
2. Écrire une fonction `nb_taches_bfs(S,A)` d'arguments `S`, `A` un graphe associé à une image qui renvoie le nombre de tâches de l'image. La fonction devra faire appel à la fonction `bfs`. On pourra reprendre l'exemple décrit dans l'exercice 2.

Exercice 4

On rappelle que l'algorithme de Dijkstra réalise la recherche de plus courts chemins dans un graphe orienté ou non valué positivement. On note `deb` le sommet source et `d[u]` la longueur du plus court chemin de `deb` à un sommet `u` découvert à un certain stade de l'algorithme. Le principe de l'algorithme est le suivant :

- on initialise $d[deb] = 0$ et $d[u] = \infty$ pour tout $u \in S \setminus \{deb\}$;
- tant que le sommet `fin` n'est pas visité :
 - on détermine un sommet `u` non visité tel que $d[u]$ est minimale ;
 - le sommet `u` est considéré comme « visité » ;
 - pour chaque sommet `s` non visité et successeur de `u`, on effectue l'opération de *relâchement* :

$$\text{si } d[s] > d[u] + \nu(u, s), \text{ alors } d[s] \leftarrow d[u] + \nu(u, s)$$

autrement dit : si pour aller à `s` depuis `deb`, il est plus court de passer par `u`, alors on le fait.

Pour son implémentation, on utilisera les dictionnaires suivants :

- `dist` : pour `s` un sommet, `dist[s]` est la valeur de $d[s]$;
- `predec` : pour `s` un sommet, `predec[s]` est son prédécesseur dans le plus court chemin depuis `s0` ;
- `djvu` : pour `s` un sommet, `djvu[s]` est un booléen qui dit si le sommet a été déjà vu ou pas.

Dans ce qui suit, on effectuera des recherches de plus courts chemins entre deux sommets en admettant que de tels chemins existent. On pourra tester les fonctions sur la configuration proposée en exemple dans le fichier.

1. Écrire une fonction `dijkstra_deb_fin(S,A,deb,fin)` implémentant l'algorithme de Dijkstra d'arguments `S` la liste des sommets, `A` le dictionnaire des arêtes valuées, `deb` le sommet

de départ, **fin** le sommet de fin et qui renvoie le dictionnaire **predec** du plus court chemin obtenu entre **deb** et **fin**.

2. Écrire une fonction **aretes_dijkstra** qui est une adaptation de la fonction **aretes** de l'exercice 1 et qui construit des arêtes avec valuation égale à 1 selon le schéma $((x,y),1)$ où **x** et **y** désignent des sommets du graphe.
3. Dans l'étape de minimisation de l'algorithme de Dijkstra, on choisit habituellement un sommet qui réalise le minimum mais on pourrait aussi s'efforcer de considérer tous les sommets réalisant le minimum afin de fournir éventuellement plusieurs plus courts chemins. Écrire une fonction **dijkstra_all_deb_fin**(**S,A,deb,fin**) qui renvoie la liste des dictionnaires **predec** de tous les plus courts chemins existants entre **deb** et **fin** que l'algorithme de Dijkstra peut identifier. On pourra utiliser une file (question très difficile ...).
4. Après obtention de tous les plus courts chemins accessibles par l'algorithme de Dijkstra, la liste résultat contient éventuellement des doublons de plus courts chemins entre **deb** et **fin** mais qui diffèrent sur d'autres portions dont on ne se préoccupe pas. Écrire une fonction **elim_doublon**(**L_sol**) d'argument **L_sol** la liste de tous les plus courts chemins fournie par l'algorithme de Dijkstra et qui renvoie une nouvelle liste sans doublon.