TP 16 : Algorithme de Dijkstra

Télécharger l'annexe sur le site web. Elle contient des exemples de graphes.

1 Graphes pondérés représentés par matrice d'adjacence : manipulations élémentaires

On rappelle qu'un graphe pondéré est représenté sous forme de matrice d'adjacence $M = (m_{i,j})_{0\leq i,j\leq n-1}$ de la façon suivante :

— on a $m_{i,i} = 0$ pour tout i ;

— pour $i \neq j$, s'il y a un arc entre i et j, alors $m_{i,j}$ est le poids de l'arc (noté $\omega(i,j)$ dans le cours), sinon $m_{i,j} = +\infty$.

En Python, une telle matrice se représente classiquement par une liste de listes. L'infini est représenté par le flottant infini, que l'on obtient via `float('inf')`. Dans l'annexe, la variable `inf` contient `float('inf')`.

Question 1. *Voisins d'un sommet.* Avec G représenté par matrice d'adjacence, les voisins d'un sommet i sont les sommets $j \neq i$ tels qu'il existe un arc de i à j (le coefficient dans la matrice d'adjacence n'est pas infini). Écrire une fonction `voisins(G, i)` renvoyant la liste des voisins du sommet `i`.

```
>>> voisins(Gex, 0) #Gex présent dans l'annexe
[1, 4]
>>> voisins(Gex, 3)
[0, 2]
```

Question 2. *Somme des poids des arcs.* Écrire une fonction `somme_poids(G)` faisant la somme des poids de tous les arcs du graphe passé en paramètre, représenté par matrice d'adjacence.

```
>>> somme_poids(Gex)
48
```

2 Algorithme de Dijkstra

On rappelle ci-dessous l'algorithme de Dijkstra pour le calcul des distances depuis un sommet source s vers tous les sommets du graphe.

Algorithme 1 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s
Sortie : Les poids minimaux $\delta(s,t)$ pour tout $t \in V$
$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;
$H \leftarrow \emptyset$; $F \leftarrow \{s\}$;
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F;
  pour *tout voisin* v *de* u **faire**
    si v *n'est ni dans* F *ni dans* H **alors**
      Ajouter v à F
    $d[v] \leftarrow \min(d[v], d[u] + \omega(u,v))$
  Ajouter u à H
Renvoyer d

Quelques explications :

— l'ensemble H représente les sommets déja traités : pour un sommt t de H, $d[t]$ contient effectivement $\delta(s,t)$, poids d'un chemin de plus petit poids de s à t ;

— l'ensemble F contient les sommets découverts (ils vérifient $\delta(s,t) \leq d[t] < +\infty$), néanmoins $d[t]$ peut encore diminuer.

Question 3. *Implémentation de l'algorithme de Dijsktra (matrice d'adjacence).* Cette question a pour but de vous aider à implémenter l'algorithme, comme dans le cours. Ne pas le regarder ! On suppose que le graphe G est donné par matrice d'adjacence, et l'implémentation diffère un peu du pseudo-code :

— l'ensemble H sera implémenté comme une liste de booléens `deja_traites` : `deja_traites[t]` vaut `True` si et seulement si `t` est dans H.

— l'ensemble $F \cup H$ consiste exactement en les sommets `t` vérifiant `d[t]` $< +\infty$: on utilise pas de structure particulière pour F.

1. Écrire une fonction `sommet_min(d, H)`, prenant en entrée : une liste `d` d'éléments dans $\mathbb{R}_+ \cup \{+\infty\}$, une liste de booléens de même taille `H`, et qui renvoie :

— -1 s'il n'existe pas d'indice `i` avec $0 \leq$ `i` $< n$ (où n est la taille commune de `d` et `H`) telle que `H[i]` est faux et `d[i]` $< +\infty$.

— sinon, un indice `i` tel que `d[i]` est minimal parmi les indices vérifiant `H[i]` faux.

Dans l'idée, on extrait de F le sommet vérifiant `d[i]` minimal, et on renvoie -1 s'il n'en existe pas.

2. En déduire une implémentation de l'algorithme `dijkstra(G, s)` prenant en entrée le graphe donné par matrice d'adjacence, et renvoyant la liste des poids des chemins de plus petits poids entre s et tout sommet du graphe. On pourra remplacer la boucle « Tant que » du pseudo-code par une boucle infinie, si la fonction précédente renvoie -1 c'est qu'on a terminé !

```
>>> dijkstra(Gex,0)
[0, 8, 9, 7, 5]
>>> dijkstra(Gex, 1)
[11, 0, 1, 4, 2]
```

Question 4. *Rajout des prédécesseurs et chemins de plus petits poids.* On peut modifier l'algorithme de Dijsktra pour calculer en même temps une liste P de prédecesseurs dans les chemins de plus petits poids depuis s. En pseudo code, il suffit de remplacer l'instruction $d[v] \leftarrow \min(d[v], d[u] + \omega(u,v))$ par un test : si $d[u] + \omega(u,v) < d[v]$, on remplace $d[v]$ par $d[u] + \omega(u,v)$ et on place u comme prédecesseur de v. Modifier l'algorithme pour qu'il renvoie une telle liste de prédecesseurs (on initialisera la liste avec des -1). L'algorithme `dijstra(G,s)` renverra le couple `(d, P)`.

```
>>> dijkstra(Gex,0)
([0, 8, 9, 7, 5], [-1, 4, 1, 4, 0])
>>> dijkstra(Gex, 1)
([11, 0, 1, 4, 2], [3, -1, 1, 4, 1])
```

Question 5. *Calcul effectif de chemins de plus petits poids -redite du TP sur le parcours en largeur-.* À partir de la liste P précédente, il est possible de calculer effectivement des chemins de poids minimaux : pour calculer le chemin de plus petit poids de s à v (avec v accessible depuis s), il suffit de remonter de prédecesseur en prédecesseur de v jusqu'à s grâce à la liste P : on obtient alors le chemin à l'envers. Écrire une fonction `chemin(P,s,v)` prenant en entrée la liste des prédecesseurs obtenue dans `dijkstra(G,s)`, les sommets s et v, et renvoyant (dans l'ordre !) un chemin de plus petit poids de s à v.

```
>>> chemin([3, -1, 1, 4, 1], 1, 3)
[1, 4, 3]
```

3 Application : chemin de plus petit poids dans un tableau d'entiers

On considère un tableau d'entiers positifs, comme celui ci-dessous. Dans un tel tableau, on peut se déplacer d'une case vers la gauche, la droite, le haut ou le bas. Le poids d'un chemin est la somme des entiers rencontrés sur le chemin. On se pose la question du poids minimal d'un chemin reliant la case en haut à gauche vers la case en bas à droite. Dans le tableau ci-dessous, ce poids est 2297, et un chemin correspondant est indiqué en gras.

131	673	**234**	**103**	**18**
201	**96**	**342**	965	**150**
630	803	746	**422**	**111**
537	699	497	**121**	956
805	732	524	**37**	**331**

On se propose d'écrire une fonction `sol_pb(T)` prenant en entrée un tel tableau d'entiers (qui sera supposé carré) et calculant le poids minimal d'un chemin de la case en haut à gauche à celle en bas à droite, et affichant le chemin via les indices de ses cases, comme ci-dessous :

```
>>> sol_pb(T5) # T est le tableau ci-dessus, représenté comme une liste de listes
Le poids minimal est :  2297
Un chemin est :  0-0 -> 1-0 -> 1-1 -> 1-2 -> 0-2 -> 0-3 -> 0-4 -> 1-4 -> 2-4 -> 2-3 -> 3-3 -> 4-3 -> 4-4
```

Le reste de la section est dédié à l'écriture de cette fonction.

3.1 Graphe associé au tableau d'entiers

À un tableau d'entiers $T = (t_{i,j})$ de taille $p \times p$ comme ci-dessus, on associe un graphe à p^2 sommets : chaque case du tableau est associé à un sommet. Il y a un arc entre deux sommets (i, j) et (i', j') si et seulement si ces cases sont voisines dans le tableau (on a $|i - i'| + |j - j'| = 1$), le poids de l'arc est la valeur $t_{i',j'}$: dans l'idée, se rendre sur la case (i', j') « coûte » $t_{i',j'}$.

On numérotera les sommets du graphe via les entiers de $[\![0, p^2 - 1]\!]$. Pour passer d'une case (i, j) à son numéro, on utilisera les deux fonctions ci-dessous :

```
def sommet(i,j,p):
    return p*i+j

def indices(a, p):
    return a//p, a%p
```

Par exemple, la case $(2, 3)$ du tableau de taille 5×5 sera numérotée $2 \times 5 + 3 = 13$:

```
>>> sommet(2,3,5)
13
>>> indices(13,5)
(2, 3)
```

Question 6. *Voisins d'une case.* Écrire une fonction `cases_voisines(p, i, j)` prenant en entrée un entier $p \geq 3$ et deux indices i et j vérifiant $0 \leq i, j < p$, et renvoyant les couples d'indices des cases voisines de (i, j) dans un tableau de taille $p \times p$: attention, une case située sur un bord (resp. dans un coin) ne possède que 3 (resp. 2) voisines.

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

On impose une complexité $O(1)$.

Question 7. *Graphe associé à un tableau.* À l'aide de la fonction précédente, écrire une fonction `graphe(T)` prenant en entrée un tableau d'entiers positifs de taille $p \times p$, et renvoyant la matrice d'adajcence du graphe associé (de taille $p^2 \times p^2$). Celle-ci est très creuse : il y a au plus 4 coefficients différents de $+\infty$ sur chaque ligne d'indice i, sans compter le zéro en case (i, i).

```
>>> graphe(T) #T est le tableau de taille 5x5 : le graphe associé a 25 sommets !
[[0, 673, inf, inf, inf, 201, inf, inf, ...
>>> T3 #de taille 3x3
[[45, 14, 8], [21, 101, 78], [11, 85, 47]]
>>> graphe(T3) #de taille 9x9
[[0, 14, inf, 21, inf, inf, inf, inf, inf], [45, 0, 8, inf, 101, inf, inf, inf, inf],
[inf, 14, 0, inf, inf, 78, inf, inf, inf], [45, inf, inf, 0, 101, inf, 11, inf, inf],
[inf, 14, inf, 21, 0, 78, inf, 85, inf], [inf, inf, 8, inf, 101, 0, inf, inf, 47],
[inf, inf, inf, 21, inf, inf, 0, 85, inf], [inf, inf, inf, inf, 101, inf, 11, 0, 47],
[inf, inf, inf, inf, inf, 78, inf, 85, 0]]
```

3.2 Résolution du problème

Question 8. En utilisant les fonctions précédentes, résoudre le problème ! *Indication :* il suffit d'appeler l'algorithme de Dijkstra sur le graphe obtenu, depuis le sommet 0 (associé à la case $(0,0)$). La case cible $(p-1,p-1)$ a pour numéro p^2-1. Pour le poids du chemin, ne pas oublier d'ajouter le coefficient en haut à gauche, qui n'est pas compté !

```
>>> sol_pb(T3)
Le poids minimal est :  192
Un chemin est :  0-0 -> 0-1 -> 0-2 -> 1-2 -> 2-2
```

4 Pour aller plus loin : Dijkstra par listes d'adjacence

Lorsque le graphe est très creux comme dans la section précédente, l'utilisation d'une matrice d'adjacence est problématique, à la fois du point de vue de la mémoire utilisée que de la complexité de l'algorithme de Dijkstra. Un graphe pondéré peut également s'implémenter par listes d'adjacence, on ne stocke pour un sommet i qu'une liste de couples $(j,\omega(i,j))$, où j est un voisin de i et $\omega(i,j)$ est le poids de l'arc (i,j). Pour implémenter l'algorithme de Dijkstra, il convient d'utiliser une structure efficace appelée *file de priorité* pour gérer l'ensemble F du pseudo-code.

4.1 Module `heapq` et file de priorité

Notion de file de priorité. Une file de priorité (abrégée FP)est une structure quoi doit gérer des couples (élément, priorité). Les opérations permises sont les suivantes :

- création d'une FP vide, test si une FP est vide ;
- ajout d'un couple (élément, priorité) dans une FP ;
- retrait du couple (élément, priorité) avec priorité le plus faible (on convient que plus l'élément priorité est faible, plus le couple est prioritaire).

Module `heapq`. Ce module permet de gérer une liste de sorte que les opérations d'insertion / suppression dans la liste ont toutes un coût $O(\log n)$ avec n la taille de la liste, et l'élément le plus petit est toujours à l'indice 0. On n'expliquera pas la structure (de *tas*) sous-jacente dans ce TP, seulement les opérations que l'on peut effectuer.

- `[]` : une liste vide est un tas vide. Dans la suite, `t` désigne un tas. On a importé le module comme `import heapq`.
- `heapq.heappush(t,x)` : rajoute `x` au tas ;
- `heapq.heappop(t)` : supprime le plus petit élément du tas et le renvoie.

Par exemple :

```
>>> t=[8, 4, 9, 12, 5, 7, 1, 3]
>>> heapq.heapify(t) #transformer t en tas
>>> t
[1, 3, 7, 4, 5, 8, 9, 12]
>>> heapq.heappop(t)
1
>>> t
[3, 4, 7, 12, 5, 8, 9]
>>> heapq.heappop(t)
3
>>> t
[4, 5, 7, 12, 9, 8]
>>> heapq.heappush(t, 6) ; t
[4, 5, 6, 12, 9, 8, 7]
>>>
>>> heapq.heappop(t)
4
>>> t
[5, 7, 6, 12, 9, 8]
```

File de priorité avec un tas. On peut implémenter une file de priorité en partant d'une liste vide, et en se restreignant aux opérations `heappop` et `heappush`. Par défaut, Python compare des couples en suivant l'ordre lexicographique (le premier élément d'abord). Il suffit donc de stocker les couples $(priorit, lment)$ dans la liste pour avoir une file de priorité.

4.2 Implémentation de l'algorithme de Dijkstra pour un graphe stocké par listes d'adjacence

Pour implémenter l'algorithme de Dijkstra, on peut utiliser une file de priorité pour gérer l'ensemble F. Indications :

— Il suffit donc de stocker les couples $(d[u], u)$ dans une file de priorité F (implémentée comme un tas). Lorsqu'on relâche l'arc $u \to v$ et que ce relâchement fait diminuer $d[v]$, on stockera $(d[v], v)$ dans F.

— on gardera une liste de booléens pour gérer l'ensemble H.

— un sommet u peut se trouver plusieurs fois dans la file : lorsqu'on sortira un couple de la file, on vérifiera que le sommet associé n'est pas déja dans H (ce serait inutile de traiter ce sommet). Si c'est le cas, il n'y a rien à faire !

Question 9. Écrire une fonction `dijkstra_creux(G,s)` reprenant cette idée, travaillant sur un graphe représenté par listes d'adjacence.

```
>>> dijkstra_creux(Gex_creux,0) #même graphe en version listes d'adjacence : même résultat !
([0, 8, 9, 7, 5], [-1, 4, 1, 4, 0])
```

4.3 Retour sur le problème du poids d'un chemin dans un tableau d'entiers

Question 10. *Graphe creux associé à un tableau.* Réécrire `graphe_creux(T)`, version de `graphe(T)` prenant en entrée un tableau d'entiers positifs de taille $p \times p$, et renvoyant le graphe associé à $p^2 \times p^2$ sommets, représenté par liste d'adjacence.

```
>>> graphe_creux(T3)
[[(3, 21), (1, 14)], [(4, 101), (0, 45), (2, 8)], [(5, 78), (1, 14)], [(0, 45), (6, 11), (4, 101)],
[(1, 14), (7, 85), (3, 21), (5, 78)], [(2, 8), (8, 47), (4, 101)], [(3, 21), (7, 85)],
[(4, 101), (6, 11), (8, 47)], [(5, 78), (7, 85)]]
```

Question 11. En déduire `sol_pb_creux(T)`, similaire à `sol_pb(T)` mais en utilisant un graphe représenté par listes d'adjacence. Sur le site web, vous trouverez dans le fichier `m80.py` un tableau `m80` de taille 80×80. Déterminez la somme minimale d'un chemin entre le coin en haut à gauche et le coin en bas à droite, et résolvez ainsi le problème 83 du site Project-Euler [1].

```
>>> sol_pb_creux(T5)
Le poids minimal est :  2297
Un chemin est :  0-0 -> 1-0 -> 1-1 -> 1-2 -> 0-2 -> 0-3 -> 0-4 -> 1-4 -> 2-4 -> 2-3 -> 3-3 -> 4-3 -> 4-4
>>> sol_pb_creux(m80)
.... à vous de trouver !
```

Remarque : sur un tableau de taille $p \times p$, le graphe associé possède p^2 sommets, l'algorithme de Dijkstra (version matrice d'adjacence) a donc une complexité $O(p^4)$. Comme ce graphe possède peu d'arcs ($O(p^2)$), on peut montrer que l'algorithme de Dijkstra version listes d'adjacence a une complexité $O(p^2 \log p)$. Il est possible de résoudre le problème par matrice d'adjacence, mais le temps de calcul est bien moindre avec la version par listes d'adjacence !

1. `https://projecteuler.net/problem=83`