Cours 5

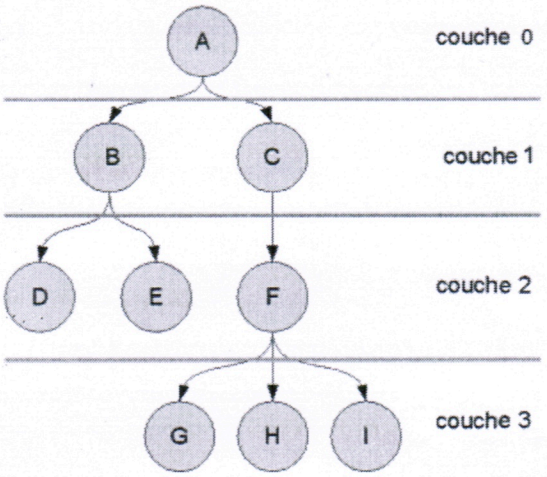
Chapitre 4 (suite) : Compléments sur les parcours de graphes

Objectifs concours

- Comprendre et savoir implémenter les algorithmes de graphes au programme de prépa, à savoir les parcours en largeur et en profondeur, ainsi que l'algorithme de Dijkstra.
- Même si ces algorithmes sont difficiles à implémenter ex nihilo, les concours peuvent considérer qu'il s'agit de questions de cours car figurant au programme d'ITC... attention donc !

1 - Parcours en largeur

On rappelle que dans un parcours en largeur, on explore les sommets d'un graphe G à partir un sommet source s par « couches », selon leur distance à s croissante : d'abord s, puis les voisins de s, puis les voisins de ces voisins, etc.



1.1 - Détails d'implémentation

À retenir : détails d'implémentation du parcours en largeur

- Comme pour le parcours en profondeur, pour éviter d'explorer plusieurs fois les mêmes sommets, on utilise un dictionnaire (dont les clés sont les sommets déjà vus et les valeurs associées arbitraires) ou une liste de booléens (si les sommets sont numérotés à partir de 0).
- Pour explorer les sommets dans le bon ordre, on stocke les sommets à traiter dans une liste ou de booléens.

Remarque : L'implémentation d'une telle structure n'est pas un attendu du programme. En Python, on peut par exemple la réaliser avec une liste via `append` (ajout) et `pop(0)` (extraction, voir cours de sup), ou avec une `deque` (« file à deux bouts »), qui permet d'ajouter et d'extraire des éléments aux deux extrémités en $\mathcal{O}(1)$ (voir par exemple le TP6). C'est cette seconde option qui est choisie dans le code page suivante.

1.2 - Application 1 : calcul des distances à la source s

Dans un graphe $G = (S, A)$, on rappelle que la *distance* entre deux sommets u et v est la longueur d'un plus court chemin de u vers v, et on la note $d(u, v)$. S'il n'existe aucun chemin de u vers v, on pose $d(u, v) = +\infty$. Comme le parcours en largeur explore le graphe par niveaux de profondeurs, on peut facilement l'adapter pour calculer les distances à la source, c'est-à-dire les $d(s, u)$ pour tout sommet $u \in S$.

À retenir : calcul des distances à la source

Si un sommet v a été vu **pour la première fois** lors de l'examen des voisins de u, alors $d(s,v) = d(s,u)+1$.

```
def parcoursLargeur(G, s):
    """
      Explore le graphe en largeur à partir du sommet s
      et renvoie les distances d(s,u) pour tout sommet u
      Entrées : graphe _G_ (listes d'adjacences), sommet _s_ (entier)
      Sortie : liste _dist_ des distances à s
    """
    n = len(G)
    dejaVu = [False] * n  # On retient pour chaque sommet s'il a déjà inséré dans F
    dist = [-1]*n # ou dist = [-1 for i in range(n)] .
    pere = [0]*n
    # La boucle (le parcours) démarrera du sommet s
    F = deque([s])
    dejaVu[s] = True
    dist[s] += 1
    pere[s] = 0
    # On continue tant qu'il reste un sommet à traiter
    while len(F) > 0:
        # On extrait le sommet de la file et on regarde ses voisins
        u = F.popleft()
        for v in G[u]:
            if not dejaVu[v]:
                F.append(v)
                dejaVu[v] = True
                dist[v] = dist[u] + 1
                pere[v] = u
    return dist pere
```

1.3 - Application 2 : plus courts chemins depuis la source s

Bien souvent, on s'intéresse davantage aux plus courts chemins qu'à leur longueur. On peut facilement les obtenir à partir du parcours en largeur, plus précisément en utilisant l'*arborescence* du parcours. En réalité, nous nous en sommes déjà servi (sans le dire) pour calculer les distances.

Arborescence d'un parcours : rappels et application

L'arborescence d'un parcours d'un graphe G est constituée des arêtes (ou arcs) (u,v) tels que le sommet v a été vu **pour la première fois** lors de l'examen de l'arête (u,v). On dit alors que u est le *père* de v. La donnée des pères de tous les sommets dans l'arborescence permet de reconstituer les plus courts chemins.

en rouge

Modifier `parcoursLargeur` pour renvoyer en plus l'arborescence sous la forme d'une liste `pere`, puis écrire la fonction `chemin(pere, s, t)` qui à partir de là renvoie une liste correspondant à un plus court chemin de `s` vers le sommet `t`.

```
def chemin(pere:list, s:int, t:int) -> list:
    pcc = [t]
    sommet = t
    while sommet != s:
        pcc.append(pere[sommet])
        sommet = pere[sommet]
    return pcc[::1]
```

2 - Rappels : complexité des parcours de graphe

On rappelle ici les grandes lignes permettant d'obtenir la complexité des algorithmes de parcours étudiés. On note $|S|$ et $|A|$ les nombres de sommets et d'arêtes/arcs de G.

À retenir : complexité des parcours (largeur ou profondeur)

- Les initialisations ont une complexité en $O(1)$
- Durant le parcours, chaque sommet est exploré 1 seule fois grâce à la liste deja Vu $\rightsquigarrow O(|S|)$
- Chaque arête est explorée 1 seule fois $\rightsquigarrow O(|A|)$
- Si les autres opérations sont toutes élémentaires, la complexité des parcours est donc en $O(|S|+|A|)$

Remarque : Si la file du parcours en largeur était gérée par une simple liste Python (`L.pop(0)` est en $\mathcal{O}$(`len(L)`)), ou si `G` était une matrice d'adjacence (parcours des voisins de u en $\mathcal{O}(n)$), la complexité obtenue ne serait pas optimale.

3 - Algorithme de Dijkstra

On cherche de nouveau à résoudre le problème des plus courts chemins à partir d'un sommet source s fixé, mais cette fois dans un graphe $G = (S, A)$ **pondéré**, c'est-à-dire dont les arêtes ont une longueur, plus communément appelée *poids*. On note $w : A \to \mathbb{R}$ la fonction de pondération associée.

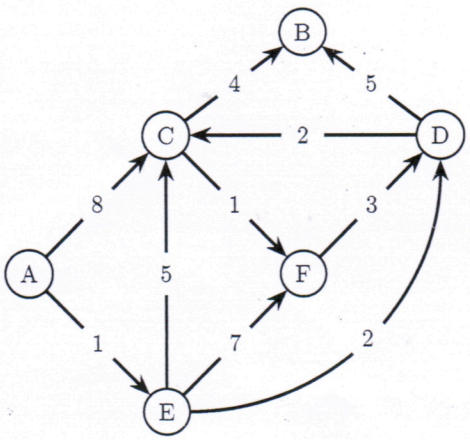
La longueur d'un chemin $\mathscr{C} = (u = u_0, u_1, \ldots, u_p = v)$ de u vers v est définie naturellement par la somme des longueurs des arêtes empruntées :

$$w(\mathscr{C}) = \sum_{i=0}^{p-1} w((u_i, u_{i+1}))$$

Algorithme de Dijkstra : objectif et courte présentation

- L'algorithme de Dijkstra (attention à l'orthographe !) permet de résoudre le problème des plus courts chemin dans un graphe **pondéré** dont les poids sont **positifs ou nuls**.
- Comme le parcours en largeur, l'ordre d'exploration des sommets suit leur distance à la source : des plus proches aux plus éloignés.

Exemple 1. Découverte et exécution de l'algorithme de Dijkstra sur le graphe ci-dessous, à partir du sommet A.



Commençons par explorer les voisins de A. On indiquera à la fin de chaque étape l'arborescence courante du parcours, stockant les meilleurs chemins actuels. Pour illustrer l'algorithme, la longueur d'une arête sur les schémas sera croissante selon son poids.

Le prochain sommet à explorer est . E . car . $w(A,E) < w(A,C)$ .....

.....

On illustre les ~~trois prochaines~~ étapes ci-dessous.

initialisation

	A	B	C	D	E	F
F	•	•	•	•	•	•
d	0	+∞	+∞	+∞	+∞	+∞

A

	A	B	C	D	E	F
F	X	•	•	•	•	•
d	0	+∞	8	+∞	1	+∞

E

	A	B	C	D	E	F
F	X	•	•	•	X	•
d	0	+∞	6	3	1	8

D

	A	B	C	D	E	F
F	X	•	•	X	X	•
d	0	8	5	3	1	8

C

	A	B	C	D	E	F
F	X	•	X	X	X	•
d	0	8	5	3	1	6

F

	A	B	C	D	E	F
F	X	•	X	X	X	X
d	0	8	5	3	1	6

B

	A	B	C	D	E	F
F	X	X	X	X	X	X
d	0	8	5	3	1	6

Algorithme de Dijkstra : principe détaillé

- Durant l'exécution de l'algorithme, on stocke l'ensemble des sommets qu'il reste à traiter, ainsi qu'une liste `d` contenant les **surestimations** actuelles des distances à la source $d(s,u)$ pour tout sommet u.
- Lors de l'exploration des voisins de u, on met à jour `d[v]` lorsqu'on vient de trouver un meilleur chemin entre s et v que le plus court chemin actuel.
- Comme les poids sont supposés tous positifs, on garantit ainsi qu'à chaque extraction, la distance estimée pour ce sommet est la bonne, c'est-à-dire `d[u]` $= d(s,u)$.

Pseudo-code :

`d[u]` $\leftarrow +\infty$ pour tout $u \in S$;
`d[s]` $\leftarrow 0$;
$F \leftarrow S$ (les sommets à traiter) ;] #initialisation.
tant que $F \neq \emptyset$ **faire** # tant qu'il reste des sommets à traiter.
  $u \leftarrow$ retirer de F un sommet v tel que $d[v]$ est minimal ;
  pour tous les *sommets v voisins de u* **faire**
    si . d[u] + poids(u,v) < d[v] ..... **alors**
      . d[v] = d[u] + poids(u,v) .....
    fin
  fin
fin
retourner `d`

(poids(u,v) = w(u,v))

Remarques :

- Pour montrer rigoureusement la correction de l'algorithme, on utilise l'. invariant de boucle suivant : « à la fin de chaque tour de boucle `while`, `d[u]` $= d(s,u)$ pour tous les sommets u **extraits** ».
- La complexité de l'algorithme de Dijkstra dépendra de : . * nombre de sommets et des poids des arcs : complexité maximale $O(n^2)$ ou $n = |S|$ .....

- Pour gérer l'ensemble F des sommets qu'il reste à traiter, on peut utiliser une liste de longueur n (n = nbre de sommets), initialisée à False.
- Si tous les poids sont égaux à 1, on peut simplement utiliser un parcours en largeur.
- On peut, comme pour le parcours en largeur, stocker et mettre à jour l'arborescence dans une liste `pere`. Le code de reconstruction de chemin est alors le même que celui écrit en fin de partie 1.

4 - Algorithme A^*, une variante de Dijkstra

Dans les utilisations pratiques d'un algorithme de plus courts chemins (par exemple, une recherche d'itinéraire via Google Maps), toutes les distances depuis la source ne nous intéressent pas : on souhaite surtout la distance minimale entre deux sommets du graphe s et t, et un plus court chemin associé (revoir les TP correspondants pour les détails).

L'idée de l'algorithme A^* est d'adapter l'algorithme de Dijkstra afin éviter des calculs inutiles, **en privilégiant l'examen de sommets dont ON PENSE qu'ils pourront donner le plus court chemin recherché.**

Soit $G = (S, A, w)$ un graphe pondéré. On suppose connue une fonction $f : S \to \mathbb{R}^+$ positive qui donne pour chaque nœud $u \in S$ une idée de la longueur d'un plus court chemin de u à t : c'est l'**heuristique**. L'algorithme fonctionne alors comme celui de Dijkstra, sauf que dans la boucle principale, on retire de l'ensemble des sommets non encore traités un sommet v tel que $d[v]+f(v)$ est minimal (on rappelle que $d[v]$ est la longueur du plus court chemin actuel, c'est-à-dire passant par les sommets déjà traités).

$d[u] \leftarrow +\infty$ pour tout $u \in S$;
$d[s] \leftarrow 0$;
$F \leftarrow S$ (les sommets à traiter) ;
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow$ retirer de F un sommet v tel que $d[v]+f(v)$ est minimal ;
  si $u = t$ **alors**
    retourner $d[t]$;
  fin
  pour tous les *sommets v voisins de u* **faire**
    $d[v] \leftarrow min(d[v], d[u] + \omega(u, v))$;
  fin
fin

Exemple 2. Dans un graphe constitué de points du plan dont certains sont reliés par des segments, une heuristique f possible serait la distance eudidienne entre s et le pt à traiter.
Voir ici et là pour un exemple de comparaison des deux algorithmes (Dijkstra et A^*) dans ce cas.

Remarque : En pratique, la qualité de la solution renvoyée va évidemment dépendre de l'heuristique f. Un critère intéressant est l'utilisation d'une heuristique dite *admissible*, c'est-à-dire qui ne surestime jamais la distance recherchée. On peut démontrer que dans le cas d'une heuristique admissible, l'algorithme A^* calcule bien la distance entre s à t. C'est notamment le cas dans l'exemple précédent (grâce à l'inégalité triangulaire). Trouver une heuristique admissible pour un graphe quelconque n'est cependant pas chose aisée...

Pour conclure : rappel des objectifs concours

Rappelons que l'algorithme de Dijkstra est, selon le programme officiel, potentiellement exigible aux concours... ainsi que l'algorithme A^*. Restituer un tel algorithme sans indication est une question de cours (très) difficile, il est plus probable de rencontrer plutôt des questions de compréhension fine de ces algorithmes. Nous vous conseillons donc de les connaître sur le bout des doigts (comme les parcours de graphe).