Informatique Tronc Commun
Devoir Surveillé 1

1 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet d'obtenir les distances et plus courts chemins dans un graphe orienté pondéré, d'un sommet source fixé à tous les sommets accessibles, comme l'algorithme de Dijkstra. Contrairement à ce dernier, il reste correct en présence de poids négatifs.

1.1 Relation de récurrence

On considère un graphe G ayant un ensemble $S = [\![0, n-1]\!]$ de n sommets et un ensemble A de m arcs pondérés, et on fixe s un sommet de départ. Pour tout sommet $u \in S$ et tout entier naturel k, on définit $\delta_{k,u}$ comme le poids minimal parmi les chemins formés d'au plus k arcs allant de s à u, en convenant que $\delta_{k,u}$ est infini s'il n'existe pas de tel chemin.

1. On considère dans cette question un graphe $G1$ défini par listes d'adjacences en Python

   ```
   G1 = [[(3,-2)], [(2,4)], [(0,1), (1,1), (3,4)], [(1,-2)]]
   ```

   où pour chaque sommet u, `G[u]` contient les couples de la forme `(v,w)`, où v est un successeur de u et w le poids de l'arc (u, v).

   (a) Représenter graphiquement le graphe $G1$ en faisant apparaître sous chaque arc son poids.

   (b) On fixe $s = 2$. Justifier précisément que $\delta_{1,3}$ vaut 4 et $\delta_{2,3}$ vaut -1.

   (c) Donner sans justification la valeur de $\delta_{3,1}$.

2. On revient au cas général. Justifier les identités suivantes, pour $v \in S$ et $k \in \mathbb{N}^*$:

   (a) $\delta_{0,s} = 0$

   (b) $\delta_{0,v} = +\infty$ si $v \neq s$

   (c) $\delta_{k,v} = \min\left(\delta_{k-1,v}, \min\limits_{(u,v)\in A}(\delta_{k-1,u} + w(u,v))\right)$, où $w(u,v)$ est le poids de l'arc (u,v).

3. Montrer que si G ne contient pas de cycle de poids strictement négatif, on a

$$\forall v \in S, \delta_{n-1,v} = \delta(s,v)$$

1.2 Première implémentation

La relation de récurrence obtenue nécessite de parcourir les prédécesseurs d'un sommet, plutôt que ses successeurs. Une représentation par listes d'adjacences n'est donc plus adaptée, et nous allons dans cette partie opter pour une représentation par matrice d'adjacence.

On convient que dans cette nouvelle représentation, la case de coordonnées (u, v) de la matrice contient $w(u, v)$, si l'arc (u, v) est dans le graphe, et $+\infty$ sinon.

1. Donner la matrice d'adjacence $M1$ du graphe $G1$.
2. Écrire une fonction `listes_vers_matrice` prenant en argument un tableau de listes d'adjacences et renvoyant la matrice d'adjacence correspondante. Cette fonction devra donc en particulier renvoyer $M1$ sur $G1$.
3. Écrire une fonction `bellman_ford` prenant en argument une matrice d'adjacence et un sommet s, et renvoyant la liste des $\delta(s, v)$ pour chaque v de S. On réalisera un calcul ascendant des identités de la partie précédente. On supposera, sans le vérifier, que le graphe ne contient pas de cycle de poids négatif. On pourra utiliser l'expression `float("inf")` pour représenter $+\infty$ en Python.

4. Écrire une variante `bellman_ford_chemins` de la fonction précédente renvoyant également une liste `P` telle que `P[u]` contient le prédécesseur de u dans un plus court chemin de s à u, ou -1 si un tel prédécesseur n'existe pas.
5. Écrire une fonction `chemin` prenant en argument une telle liste P et un sommet u et renvoyant un plus court chemin de s à u, sous forme de liste. La fonction pourra renvoyer n'importe quelle liste sur un sommet non accessible depuis s.
6. Déterminer les complexités spatiales et temporelles de la fonction `bellman_ford_chemins`.
7. Comparer la complexité temporelle obtenue à celle de l'algorithme de Dijkstra.
8. On admet que si le graphe contient un cycle de poids strictement négatif accessible depuis s, alors il existe $v \in S$ tel que $\delta_{n,v} < \delta_{n-1,v}$. Écrire une variante `bellman_ford_detection` de `bellman_ford_chemins` renvoyant `None` à la place du couple `D,P` si le graphe contient un tel cycle.

1.3 Implémentation optimisée

Pour optimiser l'algorithme de Bellman-Ford, on s'appuie sur les observations suivantes :
- Itérer sur chaque sommet v puis sur chaque prédécesseur u de v revient à itérer sur chaque arc (u, v) du graphe. En effet l'ordre dans lequel ces arcs sont relâchés n'a pas d'importance.
- Il n'est pas nécessaire de garder toutes les valeurs $\delta_{k,u}$ en mémoire, en effet seule la ligne précédente est nécessaire pour calculer une nouvelle ligne.
- On peut même se contenter de travailler sur une seule ligne. On va en effet parfois considérer une valeur $\delta_{k,u}$ à la place de $\delta_{k-1,u}$, mais ce n'est pas un problème car la première est en fait une meilleure approximation de $\delta(s, u)$ que la seconde. On se ramène donc au calcul d'une liste `D` vérifiant à la fin de chaque itération de la boucle principale $\delta(s, u) \leq$ `D[u]` $\leq \delta_{k,u}$.

D'après le premier point, il est donc plus efficace de revenir à une représentation par listes d'adjacences.

1. Écrire une fonction `Bellman_ford_opti` de même spécification que `bellman_ford_chemins`, mais prenant le graphe sous forme de listes d'adjacences, et mettant en oeuvre les optimisations ci-dessus.
2. Déterminer les complexités spatiales et temporelles de cette fonction (on pourra supposer $m \geq n$).
3. Démontrer que
$$\forall u \in S, \delta(s, u) \leq \texttt{D[u]} \leq \delta_{k-1,u}$$
est un invariant de la boucle principale, où `D` est la liste de distances mise à jour dans la version optimisée.

2 Représentation d'un graphe orienté pondéré en SQL

On représente un graphe orienté pondéré par deux tables SQL :
- une table `sommet`, d'attributs `id` (clé primaire) et `nom` ;
- une table `arc`, d'attributs `id` (clé primaire), `poids`, `id_sommet_depart` et `id_sommet_arrivee` (clés étrangères vers `sommet`).

1. Écrire une requête affichant le poids maximal d'un arc partant du sommet dont le nom est `"Alice"`.
2. Écrire une requête affichant le nom des sommets successeurs du sommet de nom `"Alice"`.
3. Écrire une requête affichant le nom des sommets de degré sortant égal à 2.
4. Écrire une requête affichant l'identifiant des arcs dont le poids est maximal parmi les arcs du graphe.