Informatique Tronc Commun
Devoir Surveillé 2
Corrigé

1 Algorithme de Bellman-Ford

1.1 Relation de récurrence

1. (a) Je ne vous fais pas un dessin.

   (b) Il existe un seul chemin allant de 2 à 3 avec au plus 1 arc : le chemin $(2,3)$, de poids 4, d'où $\delta_{1,3} = 4$.

   Il existe deux chemins allant de 2 à 3 avec au plus 2 arcs : le chemin $(2,3)$, de poids 4, et le chemin $(2,0,3)$, de poids -1, d'où $\delta_{2,3} = -1$.

   (c) $\delta_{3,1} = -3$.

2. (a) Il existe un seul chemin allant de s à s avec 0 arcs : le chemin (s), de poids 0. On a donc $\delta_{0,s} = 0$.

   (b) Il n'existe aucun chemin allant de s à v en empruntant 0 arc, donc $\delta_{0,v} = +\infty$.

   (c) Un chemin allant de s à v d'au plus k arcs est soit un chemin d'au plus $k-1$ arcs, soit un chemin d'au plus k arcs composé d'un chemin d'au plus $k-1$ arcs de s à un sommet u, et d'un arc de u à v (ces deux cas n'étant pas disjoints). Pour obtenir le chemin minimal, on prend donc le plus petit des chemins minimaux pour chacune de ces possibilités.

3. Soit c un chemin de s à v de poids $\delta(s,v)$ utilisant un nombre d'arcs minimal k. Ce chemin passe donc par $k+1$ sommets. Or tous ces sommets sont distincts, car sinon c contiendrait un cycle, de poids positif par hypothèse, et en supprimant ce cycle on contredirait l'hypothèse de minimalité du nombre d'arcs de c. On en déduit donc $k+1 \leq n$, ie $k \leq n-1$. c est donc un chemin formé d'au plus $n-1$ arcs, donc $\delta_{n-1,v} \leq \delta(s,v)$. Par ailleurs on a toujours par définition $\delta_{n-1,v} \geq \delta(s,v)$, d'où l'égalité demandée.

1.2 Première implémentation

1. $$\texttt{M1} = \begin{pmatrix} +\infty & +\infty & +\infty & -2 \\ +\infty & +\infty & 4 & +\infty \\ 1 & 1 & +\infty & 4 \\ +\infty & -2 & +\infty & +\infty \end{pmatrix}$$

2.
```
inf = float("inf")

def listes_vers_matrice(G):
    n = len(G)
    M = [[inf for _ in range(n)] for _ in range(n)]
    for u in range(n):
        for (v,w) in G[u]:
            M[u][v] = w
    return M
```

3.
```
def bellman_ford(M,s):
    n = len(M)
    d = [[inf for _ in range(n)] for _ in range(n)]
    d[0][s] = 0.
    for k in range(1,n):
        for v in range(n):
```

```
            d[k][v] = d[k-1][v]
            for u in range(n):
                new_d = d[k-1][u] + M[u][v]
                if new_d < d[k][v]:
                    d[k][v] = new_d
    return d[n-1]
```

4. ```
   def bellman_ford(M,s):
       n = len(M)
       d = [[inf for _ in range(n)] for _ in range(n)]
       d[0][s] = 0.
       P = [-1]*n  #ligne ajoutée
       for k in range(1,n):
           for v in range(n):
               d[k][v] = d[k-1][v]
               for u in range(n):
                   new_d = d[k-1][u] + M[u][v]
                   if new_d < d[k][v]:
                       d[k][v] = new_d
                       P[v] = u  #ligne ajoutée
       return d[n-1],P   #ligne modifiée
   ```

5. ```
   def chemin(P,u):
       if P[u] == -1:
           return [u]
       else:
           l = chemin(P,P[u])
           l.append(u)
           return l
   ```

6. On observe que la fonction `bellman_ford` est constituée de trois boucles imbriquées, chacune de longueur n et itérant par ailleurs des instructions en $O(1)$. Le coût total de ces trois boucles est donc en $O(n^3)$, qui domine le coût de création de `d` (en $O(n^2)$) et de `P` (en $O(n)$), d'où une complexité temporelle totale (dans le meilleur comme dans le pire cas) en $O(n^3)$.

   Le programme nécessite une quantité de mémoire en $O(n^2)$ pour `d`, d'où une complexité spatiale en $O(n^2)$.

7. La version de l'algorithme de Dijkstra vue en cours d'ITC a une complexité en $O(n^2)$, qui est donc meilleure que la complexité de cette version de Bellman-Ford (en gardant bien en tête que Dijkstra ne peut s'appliquer que dans le cas de poids positifs).

8. ```
   def bellman_ford(M,s):
       n = len(M)
       d = [[inf for _ in range(n)] for _ in range(n)]
       d[0][s] = 0.
       P = [-1]*n
       for k in range(1,n):
           for v in range(n):
               d[k][v] = d[k-1][v]
               for u in range(n):
                   new_d = d[k-1][u] + M[u][v]
                   if new_d < d[k][v]:
                       d[k][v] = new_d
                       P[v] = u
       #debut de la partie ajoutée
       for v in range(n):
           for u in range(n):
               new_d = d[n-1][u] + M[u][v]
   ```
   ```

```
            if new_d < d[n-1][v]:
                return None
    #fin de la partie ajoutée
    return d[n-1],P
```

9.
```
def bellman_ford_desc(M,s):
    dico = {}
    n = len(M)
    def aux(k,v):
        if k == 0:
            if v != s:
                return inf
            return 0
        if (k,v) in dico:
            return dico[(k,v)]
        min = aux(k-1, v)
        for u in range(n):
            if M[u][v] < inf:
                new_d = aux(k-1,u) + M[u][v]
                if new_d < min:
                    min = new_d
        dico[(k,v)] = min
        return min
    return [aux(n-1,u) for u in range(n)]
```

1.3 Implémentation optimisée

1.
```
def bellman_ford_opti(G,s):
    n = len(G)
    D = [inf for _ in range(n)]
    D[s] = 0.
    P = [-1]*n
    for k in range(1,n):
        for u in range(n):
            for (v,w) in G[u]:
                new_d = D[u] + w
                if new_d < D[v]:
                    D[v] = new_d
                    P[v] = u
    return D,P
```

2. Bien que cette fonction soit aussi constituée de trois boucles `for`, on observe que les deux boucles `for u` et `for (v,w)` parcourent exactement une fois l'ensemble des arcs du graphe, et sont donc ensemble de complexité $O(m)$ (en fait de complexité $O(max(n,m))$, puisqu'il faut aussi parcourir le tableau des listes d'adjacence, mais d'après l'hypothèse $n \leq m$ cela revient bien à $O(m)$). La complexité temporelle totale est donc en $O(nm)$.
   Le programme nécessite une quantité de mémoire en $O(n)$ pour `D` et en $O(n)$ pour `P`, d'où une complexité spatiale en $O(n)$.

3. - Au moment d'entrer dans la boucle, on a $k = 1$ et :
     — Si $u = s$, on a $\delta(s,u) = \texttt{D[u]} = \delta_{0,u} = 0$.
     — Si $u \neq s$, on a $\delta(s,u) \leq \texttt{D[u]} = \delta_{k-1,u} = +\infty$.
   - Considérons une itération de la boucle principale pour laquelle la proposition est vraie au début, et montrons qu'elle reste vraie à la fin.
     À chaque fois qu'un `D[v]` prend la valeur `D[u]` $+ w(u,v)$, on a par hypothèse (dont une récurrence immédiate sur l'ordre de relâchement des arcs), que $\delta(s,u) \leq$ `D[u]`, d'où, après cette affectation, `D[v]` $= \texttt{D[u]} + w(u,v) \geq \delta(s,u) + w(u,v) \geq \delta(s,v)$.

De plus, pour un sommet u, on observe que la valeur de `D[u]` ne peut que diminuer, donc on a toujours `D[u]` $\leq \delta_{k-1,u}$, puisque c'était vrai au début de l'itération.
À l'issue de tout les relâchements, on a par construction, pour un sommet `v` donné,

$$\texttt{D'[v]} = \min\left(\texttt{D[v]}, \min_{(u,v)\in A}(\texttt{D[u]} + w(u,v))\right)$$

où `D[v]` est le contenu de la case en début d'itération, `D'[v]` le contenu en fin d'itération, et les `D[u]` sont les valeurs de ces cases à un moment arbitraire de l'itération. D'après, le point précédent, on a donc

$$\texttt{D'[v]} \leq \min\left(\delta_{k-1,v}, \min_{(u,v)\in A}(\delta_{k-1,u} + w(u,v))\right) = \delta_{k,v}$$

La propriété reste donc vraie en fin d'itération.

- En conclusion (techniquement non demandée, mais ça ne coûte pas cher de terminer la preuve de correction), on en déduit que la proposition est un invariant de boucle et reste donc vraie en sortie de boucle, où k vaut n. Or d'après 1.1.3, $\forall u \in S, \delta(s,u) = \delta_{n-1,u}$, d'où $\forall u \in S, \texttt{D[u]} = \delta(s,u)$. Le programme renvoie donc bien les distances correctes.

2 Représentation d'un graphe orienté pondéré en SQL

1. ```
   SELECT max(poids)
   FROM arc
   JOIN sommet
   ON sommet.id = id_sommet_depart
   WHERE nom = "Alice"
   ```

2. ```
   SELECT sommet_arrivee.nom
   FROM arc
   JOIN sommet AS sommet_depart
   ON sommet_depart.id = id_sommet_depart
   JOIN sommet AS sommet_arrivee
   ON sommet_arrivee.id = id_sommet_arrivee
   WHERE sommet_depart.nom = "Alice"
   ```

3. ```
   SELECT nom
   FROM arc
   JOIN sommet
   ON sommet.id = id_sommet_depart
   GROUP BY sommet.id
   HAVING COUNT(*) = 2
   ```

4. ```
   SELECT id
   FROM arc
   WHERE poids = (SELECT MAX(poids) FROM arc)
   ```