

IFT436 - Devoir #3

Vous devez remettre vos réponses d'ici le **vendredi 16 octobre à 23h59**. Ce travail compte pour 6% des points de la session. Tout retard entraînera jusqu'à 33% de pénalité par jour.

Modalités. Considérez les points suivants.

- Le travail peut être fait seul ou en équipe de deux. Notez qu'en matière de charge de travail, je suppose que vous faites le devoir à deux.
- Le devoir doit être remis en format pdf via turnin. La provenance de votre pdf n'a pas d'importance — ce pdf peut provenir d'un export d'un fichier Word ou d'un fichier texte, d'un scan de vos écrits/dessins, etc.
Il n'y a pas de norme de présentation particulière. Écrivez lisiblement, et assurez-vous seulement que vos noms et CIP apparaissent clairement sur le devoir remis.
- Vous pouvez utiliser les techniques de preuves que vous désirez, les indices ne sont que des suggestions.
- Vous pouvez utiliser les résultats démontrés en classe et dans les solutionnaires d'exercices sans avoir à les démontrer. Par contre, si vous utilisez de nouveaux résultats, vous devez les démontrer.

Question 1 : graphes fortement connexes (40 points)

Soit $G = (V, E)$ un graphe **dirigé**. On dit que G est *fortement connexe* si on peut aller de n'importe quel sommet à n'importe quel autre sommet. Autrement dit, G est fortement connexe si pour toute paire de sommets $u, v \in V$, il existe un chemin orienté de u à v et un chemin orienté de v à u .

On a aussi les définitions suivantes. Pour un graphe dirigé $G = (V, E)$ et $u \in V$, on dénote par $r(G, u)$ les sommets qu'on peut atteindre à partir de u , i.e.

$$r(G, u) = \{v \in V : \text{il existe un chemin orienté de } u \text{ à } v \text{ dans } G\}$$

Notez que $u \in r(G, u)$. On dénote aussi G^T le graphe *transposé* de G obtenu en renversant chaque arête. C'est-à-dire, $G^T = (V, E^T)$ où $E^T = \{(v, u) : (u, v) \in E\}$.

- a. Démontrez qu'un graphe dirigé $G = (V, E)$ est fortement connexe si et seulement si, pour tout sommet $u \in V$, $r(G, u) = V$ et $r(G^T, u) = V$.

Solution.

Rappel: quand on veut montrer que p si et seulement si q , il faut montrer que $p \Rightarrow q$ et que $q \Rightarrow p$.

Solution

($\Rightarrow$) : supposons que G est fortement connexe, et soit $u \in V$. Par la définition de fortement connexe, on peut aller de u à n'importe quel sommet, et donc $r(G, u) = V$. De plus, toujours selon la définition de fortement connexe, pour tout $v \in V$, il y a un chemin orienté de v à u . Ceci se traduit par un chemin de u à v dans G^T , et puisque c'est vrai pour tout v , on a $r(G^T, u) = V$.

($\Leftarrow$) : supposons que $r(G, u) = r(G^T, u) = V$ pour tout $u \in V$. Ceci veut dire que tout sommet u peut se rendre à tout sommet v (car $v \in r(G, u)$), et donc G est fortement connexe.

-
- b. Démontrez que si vous choisissez un sommet u arbitrairement et que $r(G, u) = r(G^T, u) = V$, alors vous avez automatiquement $r(G, v) = r(G^T, v) = V$ pour tout sommet $v \in V$.

Solution.

Soit u tel que $r(G, u) = r(G^T, u) = V$, et soit $v \in V$ un autre sommet arbitraire. On veut montrer que pour tout sommet w , on a $w \in r(G, v)$ et $w \in r(G^T, v)$.

L'idée est simplement d'utiliser u comme intermédiaire pour montrer qu'il y a toujours un chemin de v à w et vice-versa.

Donc, soit $w \in V$ un sommet arbitraire. Puisque $v \in r(G^T, u)$, il existe dans G un chemin P_1 de v à u . De plus, puisque $w \in r(G, u)$, il existe dans G un chemin P_2 de u à w . En concaténant P_1 et P_2 , on voit qu'on peut se rendre de v à w dans G , donc $w \in r(G, v)$. De la même façon, il y a un chemin dans G de w à u , et un chemin de u à v , impliquant que $v \in r(G, w)$, et donc que $w \in r(G^T, v)$.

Puisque v et w sont arbitraires, ceci montre que $r(G, v) = r(G^T, v) = V$ pour tout $v \in V$.

- c. Donnez un algorithme qui décide si un graphe dirigé est fortement connexe. Donnez ensuite la complexité de votre algorithme, avec justifications. Vous aurez un maximum de points pour une complexité optimale.
-

Solution.

L'idée est de démarrer d'un u arbitraire et regarder si $r(G, u) = r(G^T, u) = V$. Pour calculer $r(G, u)$, il suffit d'une fouille en largeur (j'accepte qu'on appelle la fonction vue en classe). Il faut montrer comment construire G^T .

```

fonction fouilleEnLargeurOriente( $G = (V, E), u$ )
     $vis = u$ 
     $file = [u]$ 
    tant que  $\neg file.vide()$  faire
         $w = file.pop()$ 
        pour chaque voisin sortant  $z$  de  $w$  faire
            si  $z \notin vis$  alors
                 $vis.add(z)$ 
                 $file.append(z)$ 
        fin
    fin
    return  $vis$ 

fonction estFortementConnexe( $G = (V, E)$ )
     $u = \text{sommet arbitraire de } G$ 
     $rG = \text{fouilleEnLargeurOriente}(G, u)$ 
     $GT = (V, \emptyset)$ 
    pour  $(u, v) \in E$  faire
         $GT.ajouterArete(v, u)$ 
    fin
     $rGT = \text{fouilleEnLargeurOriente}(GT, u)$ 
    return  $(rG.size() == n \text{ et } rGT.size() == n)$ 

```

La complexité est $O(|V| + |E|)$ car les appels à la fouille en largeur prennent ce temps, de même que la construction de G^T .

- d. Soit G un graphe dirigé. Une *composante fortement connexe* de G est un sous-ensemble $X \subseteq V$ tel que $G[X]$ est fortement connexe, et ce de façon maximale, c'est-à-dire que $G[X \cup Z]$ n'est pas fortement connexe pour tout $Z \subseteq V \setminus X$ non-vide.

Donnez le pseudo-code d'un algorithme en temps polynomial qui reçoit en entrée un graphe dirigé G , et qui retourne tous les sous-ensembles de $V(G)$ formant une composante fortement connexe de G .

Votre algorithme n'a pas à être optimal, vous aurez tous vos points pour un algorithme en temps polynomial, *peu importe le polynôme*. Vous pouvez supposer qu'il existe une fonction *existe_chemin*(G, u, v) en temps $O(n + m)$ qui détermine s'il y a un chemin de u à v . Notez que vous pouvez répondre à cette question indépendamment des sous-questions précédentes.

Solution.

On passe les sommets un par un. Pour chaque $u \in V$, on trouve les sommets que u peut atteindre et qui peuvent atteindre u . Ceci forme la composante de u . Une fois trouvée on l'ajoute à la sortie. On s'assure de ne pas considérer un sommet déjà vu.

Il n'était pas demandé de montrer que ceci fonctionne, mais je recommande tout de même d'y penser.

```
fonction composantesFortementConnexes( $G = (V, E)$ )
    composantes = []
    deja_vu =  $\emptyset$  // sommets déjà dans une composante trouvée
    pour chaque  $u \in V$  faire
        si  $u \notin \text{deja\_place}$  alors
             $X = \{u\}$ 
            pour chaque  $v \in V \setminus \{u\}$  faire
                si  $\text{existe\_chemin}(G, u, v)$  et  $\text{existe\_chemin}(G, v, u)$  alors
                     $X.add(v)$ 
            fin
            pour chaque  $v \in X$  faire
                 $\text{deja\_vu.add}(v)$ 
            fin
            composantes.append( $X$ )
    fin
    return composantes
```

La boucle principale fait n tours, et chacun fait au plus $n - 1$ tours de la boucle interne, qui fait deux appels à *existe_chemin* en temps $O(n + m)$. Le reste est dominé par ceci (avec une table de hachage pour *deja_place*). Le temps total est donc $O(n^2(n + m))$, ce qui est polynomial.

Un temps linéaire est possible, mais c'est plus compliqué (vous trouverez une description sur wikipedia).

Question 2 : le problème du commis voyageur (40 points)

Dans le problème du commis voyageur, un vendeur doit parcourir n villes *exactement* une fois pour aller vendre son produit, puis doit revenir au point de départ. Le passage d'une ville u à une autre ville v engendre un coût $f(uv)$ et le but est de trouver la route de coût minimum.

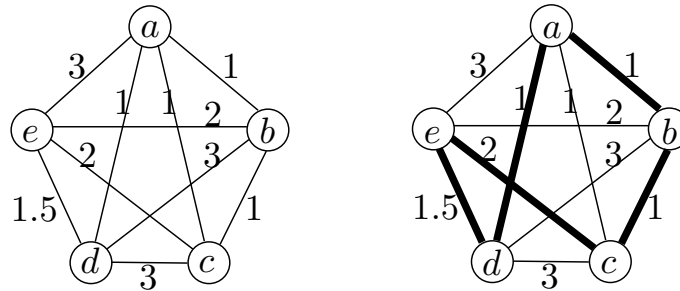


Figure 1: Un exemple d'instance du commis voyageur. À droite, le cycle (a, b, c, e, d, a) de coût total $1 + 1 + 2 + 1.5 + 1 = 6.5$.

Ceci peut se formaliser comme suit. On a un graphe complet $G = (V, E, f)$ non-orienté avec une valuation $f : E \rightarrow \mathbb{R}^{\geq 0}$ des arêtes. On cherche un cycle $C = (u_1, u_2, \dots, u_n, u_1)$ qui passe par tous les sommets tel que $\sum_{i=1}^{n-1} f(u_i u_{i+1}) + f(u_n u_1)$ est minimum parmi tous les cycles possibles.

- Considérez l'algorithme glouton suivant qui tente, en vain, de résoudre le problème du commis voyageur en choisissant toujours l'arête la plus prometteuse.

```

fonction commisGlouton( $G = (V, E, f)$ )
   $x$  = un sommet arbitraire de  $V$ 
   $C = (x)$  //  $C$  est le cycle que l'on construit
   $z = x$  //  $z$  est le dernier sommet de  $C$ 
  tant que  $C$  contient strictement moins de  $n$  éléments faire
     $best = null$ 
     $curmin = \infty$ 
    pour chaque sommet  $u$  qui n'est pas dans  $C$  faire
      si  $f(uz) < curmin$  alors
         $curmin = f(uz)$ 
         $best = u$ 
    fin
     $C.append(best)$ 
     $z = best$ 
  fin
   $C.append(x)$ 
  return  $C$ 

```

Démontrez que cet algorithme ne résout pas correctement le problème du commis voyageur.

[BONI 0 POINTS] Vous aurez 0 points bonis si vous montrez que l'algorithme est incorrect même si f satisfait *l'inégalité triangulaire* (qui est souvent satisfaite dans la vraie vie). L'inégalité triangulaire dit que pour tout triplet de sommets (u, v, w) , on a $f(uw) \leq f(uv) + f(vw)$.

Solution.

Voir <https://matheducators.stackexchange.com/questions/10692/counterexamples-to-the>

- b. Donnez un algorithme qui résout correctement le problème du commis voyageur. La complexité de votre algorithme ne sera pas évaluée: l'important est d'avoir un algorithme correct. Vous devez tout de même donner la complexité en temps de votre algorithme, avec justification.

Note: la meilleure complexité connue pour résoudre ce problème est de $O(n^2 2^n)$. Ceci est déjà un exploit, alors je recommande encore une fois de ne pas vous soucier de la complexité.

Solution.

```
fonction tsp( $G = (V, E, f)$ )
    mincout =  $\infty$ 
    pour chaque permutation  $P = (v_1, \dots, v_n)$  de  $v$  faire
        cout = 0
        minP = null
        pour  $i = 1..n - 1$  faire
            cout = cout +  $f(v_i v_{i+1})$ 
        fin
        cout = cout +  $f(v_n v_1)$ 
        si cout < mincout alors
            mincout = cout
            minP = P
    fin
    minP.append(minP.first) //juste pour boucler le cycle
    return minP
```

Le nombre de permutations énumérées est $n!$. Chaque permutation correspond

à un cycle, et son coût prend un temps $O(n)$ à calculer. La complexité est donc $O(n \cdot n!)$.

Il était possible d'atteindre $O(n(n-1)!) = O(n!)$ en fixant un sommet de départ arbitraire, puis en testant les permutations des $n-1$ autres sommets.

- c. Lorsque l'on construit un arbre couvrant minimum (acm), on n'a pas de contrôle sur les feuilles qui seront dans l'arbre résultant. Mais supposons qu'on peut le faire. Plus précisément, supposez qu'un programmeur vous donne accès au programme *acmFeuilles*($G = (V, E), X$). Le programme reçoit en argument un graphe G et un ensemble $X \subseteq V$, et retourne un arbre couvrant minimum A tel que les feuilles de A sont exactement X (si un tel A existe). Supposez que *acmFeuilles* s'exécute en temps $O(g(n))$ pour une certaine fonction g .

Montrez que vous pouvez résoudre le problème du commis voyageur en temps $O(n^2 \cdot g(n))$.

Solution.

On remarque qu'un arbre couvrant à deux feuille a et b est un chemin qui contient chaque sommet du graphe, qui commence en a et finit en b . On peut donc obtenir le coût minimum d'un cycle contenant l'arête ab en appelant *acmFeuille*($G, \{a, b\}$), puis en ajoutant $f(ab)$. On ne sait pas quelle arête ab fait partie de l'optimal, mais on peut toutes les tester avec $O(n^2)$ appels à *acmFeuilles*.

En fait, on peut optimiser à n appels (j'acceptais n^2 appels). Il suffit de fixer a et tester chacun de ses voisins possibles sur une solution. L'algorithme suivant fonctionne donc.

```

fonction tspacm( $G = (V, E, f)$ )
   $a$  = sommet arbitraire de  $V$ 
  mincout =  $\infty$ 
  minC = nil
  pour  $b \in V \setminus \{a\}$  faire
     $T = \text{acmFeuilles}(G, \{a, b\})$ 
    cout = cout( $T$ ) +  $f(ab)$ 
    si cout < mincout alors
      mincout = cout
       $T.\text{ajouterArete}(ab)$ 
      minC =  $T$ 
  fin
  return minC

```

On fait n tours de la boucle principale, et chaque tour prend un temps $g(n)$. La complexité est donc $O(n \cdot g(n))$.

Question 3: faire le plein optimalement (20 points)

On a un véhicule qui veut traverser un chemin $(s_1, \dots, s_n)$ avec un réservoir de capacité r , initialement plein. Le parcours de s_i à s_{i+1} nécessite $f(s_i, s_{i+1})$ en essence, et on ne doit jamais tomber en panne d'essence. Chaque fois que le véhicule arrive à un s_i , qui représente une station, il peut s'arrêter pour faire le plein ou bien continuer. Le but est de s'arrêter à un nombre minimum de stations pour passer de s_1 à s_n . Pour $i < j$, on définit $dist(s_i, s_j) = \sum_{k=i}^{j-1} f(s_k, s_{k+1})$.

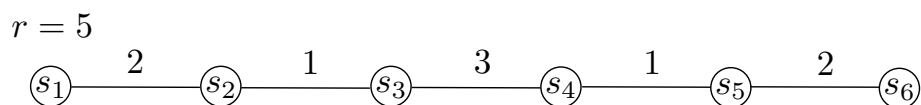


Figure 2: Un exemple d'instance du problème avec $r = 5$. Les arêtes sont étiquetées par les quantités d'essence requises. Une solution possible serait d'arrêter à s_2 et à s_4 pour faire le plein.

Ceci se formalise comme suit.

Entrée: une capacité r , une séquence $(s_1, \dots, s_n)$ et une fonction f qui attribue une valeur $0 \leq f(s_i, s_{i+1}) \leq r$ à chaque paire s_i, s_{i+1} , $i \in \{1, \dots, n-1\}$.

Sortie: une sous-séquence de stations $s_1 = s'_1 < s'_2 < \dots < s'_k = s_n$ contenant un nombre minimum d'éléments et telle que $dist(s'_i, s'_{i+1}) \leq r$ pour tout $i \in \{1, \dots, k-1\}$.

Considérez l'algorithme glouton suivant:

```

fonction minPlein( $r, (s_1, \dots, s_n), f$ )
   $S = [s_1]$ 
   $p = 1$  //  $p$  est la position de l'emplacement courant
  tant que  $p \neq n$  faire
     $prochain = \max\{i \in \{p+1, \dots, n\} : dist(s_p, s_i) \leq r\}$ 
     $p = prochain$ 
     $S.append(s_p)$ 
  fin
  return  $S$ 

```

Montrez que l'algorithme glouton *minPlein* retourne une solution comportant un nombre minimum de stations.

Solution.

Soit $S = [s'_1, \dots, s'_k]$ la sortie de notre algorithme, et soit $S^* = [s_1^*, \dots, s_h^*]$ une solution qui minimise le nombre de stations visitées. Considérons i comme le plus petit indice tel que $s'_i \neq s_i^*$ (si i n'existe pas, alors $S = S^*$ est optimal). Notons que $i > 1$ car $s_1 = s'_1 = s_1^*$. Puisque notre algorithme glouton va le plus loin possible après la station $s'_{i-1} = s_{i-1}^*$, on a que s_i^* vient avant s'_i . Si, dans S^* , on remplace s_i^*

par s'_i , on aura bien sûr assez d'essence pour aller de s_{i-1}^* à s'_i car c'est ce que fait notre algorithme. On sait aussi qu'on aura assez d'essence pour passer de s'_i à s_{i+1}^* car nous sommes plus près de cette station.

À partir d'une solution optimale contenant les $i - 1$ premiers éléments en commun avec S , nous venons de produire une solution optimale contenant les i premiers éléments en commun avec S . En répétant l'argument, on obtiendra S tout en maintenant l'optimalité de S .
