Analyse d'Algorithmes et Programmation

Contrôle Continu 2

Christina Boura et Yann Rotella
{christina.boura, yann.rotella}@uvsq.fr

5 mai 2021

1 Trouver une paire d'éléments ayant une somme donnée

On s'intéresse au problème suivant. Étant donnés une liste L de longueur n contenant des nombres entiers distincts, et un entier s, trouver (s'il existe) et renvoyer la première paire d'éléments (a, b) de la liste L qui satisfait $a + b = s$.

Exemple On suppose que $L = [5, 1, 3, 2, 7, 4]$ et $s = 4$. L'algorithme doit renvoyer $(a, b) = (1, 3)$.

(a) Décrire un algorithme naïf, parcourant toutes les possibilités de manière exhaustive, qui permet de résoudre ce problème.

(b) Décrire un algorithme amélioré dont la première étape est de trier le tableau L.

(c) Proposer une solution au problème en utilisant une table de hachage.

Pour chacun des algorithmes proposés, donner et expliquer brièvement la complexité en temps correspondante. Les algorithmes peuvent être donnés sous forme de pseudo-code ou bien être décrits dans la langue courante de manière claire.

(4.5 points)

2 Recherche de chaîne de caractères

L'algorithme naïf suivant permet de rechercher un motif P de longueur m dans un texte T de longueur n.

Algorithm 1 `Recherche naïve` (T, P)

```
1: n ← T.longueur
2: m ← P.longueur
3: pour s de 0 jusqu'à n − m faire                ▷ pour tous les décalages possibles
4:     si P[0...m − 1] = T[s...s + m − 1] alors
5:         afficher P apparaît avec le décalage s
6:     fin si
7: fin pour
```

(a) Quelle est la complexité de cet algorithme en fonction de n et de m dans le pire cas ? Et dans le meilleur cas ? Expliquer. Donner un exemple pour lequel le pire cas arrive et un autre qui illustre le meilleur cas.

(b) On suppose que tous les caractères du motif P sont différents. Proposer une amélioration de l'algorithme naïf dans ce cas.

(2.5 points)

3 Programmation linéaire

On considère le programme linéaire suivant :

$$
\begin{array}{lrcrcr}
\text{Maximiser} & & & 3x_1 & + & 4x_2 \\
\\
\text{sous les contraintes} & x_1 & + & 2x_2 & \leq & 10 \\
& -x_1 & - & x_2 & \geq & -8 \\
& 3x_1 & + & 5x_2 & \leq & 26 \\
& & & x_1, x_2 & \geq & 0
\end{array}
$$

1. Mettre le programme linéaire sous forme standard.

(0.5 point)

2. Trouver la solution du programme linéaire en détaillant les étapes de l'algorithme du simplexe.

(3.5 points)

4 Programmation - Graphes

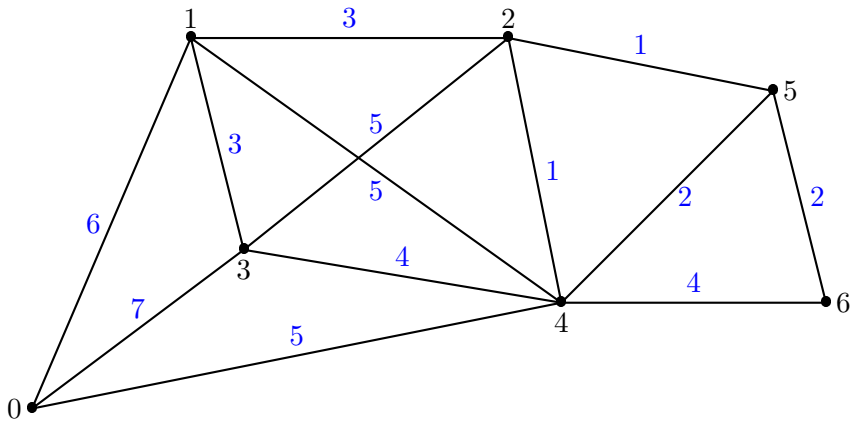
L'accès à internet n'est pas permis. Seuls exceptions sont le site `Moodle` et le site `https://jupyter.ens.uvsq.fr/`. Méthodes et astuces que l'on peut vouloir utiliser :

- On rappelle que `e in L` renvoie `True` si et seulement si l'élément `e` est dans `L`
- `L.sort()` : trie une liste.
- `L.pop(i)` : enlève et renvoie l'élément à l'indice i d'une liste.
- `L.remove(e)` : enlève l'élément e d'une liste s'il existe.
- `L.append(e)` : rajout d'un élément à une liste.
- `float('inf')` : représente l'infini en python.
- `None`

Dans cette partie, nous allons programmer l'algorithme de Prim qui prend en entrée un graphe non-orienté et un sommet du graphe et qui renvoie un arbre couvrant de poids minimal dont la racine est le sommet donné en entrée. Nous allons utiliser la représentation des graphes sous liste d'adjacence, mais sans utiliser les classes de Python. Dans tout ce qui suit, les sommets d'un graphe donné en entrée sont indexés de 0 à $n-1$ où n désigne la taille du graphe (les listes commencent à 0 en Python). Comme nous allons regarder les graphes pondérés, nous représenterons les graphes sous forme de liste d'adjacence avec des listes à trois dimensions. Plus précisément, si G désigne un graphe à n sommets représenté par liste d'adjacence, $G[i]$, $0 \leq i < n$, est une liste potentiellement vide et de taille inférieure à $n-1$ qui contient des éléments de la forme $[w, j]$ désignant l'existence d'une arête de i vers j avec le poids w. Par exemple, le graphe

```
Graphe_test = [
    [[6,1],[7,3],[5,4]],
    [[3,2],[5,4],[3,3],[6,0]],
    [[3,1],[5,3],[1,4],[1,5]],
    [[7,0],[3,1],[5,2],[4,4]],
    [[5,0],[4,3],[5,1],[1,2],[2,5],[4,6]],
    [[1,2],[2,4],[2,6]],
    [[4,4],[2,5]],
    ]
```

représente le graphe non-orienté suivant.



(a) Programmez une fonction `estGraphe` qui prend en entrée une liste de listes de listes, i.e. une liste en trois dimensions comme dans l'exemple ci-dessus) sensée représenter un graphe est vérifie que chaque élément dans la liste de liste de la forme $[w, j]$ pointe bien vers un sommet existant (indice j strictement plus petit que le nombre de listes d'adjacences (n)), qu'il n'y a pas de double arête (deux arêtes qui partent du même sommet et qui vont vers le même sommet) et qu'il n'y a pas de boucle (d'arête de i vers i). La fonction doit renvoyer un booléen `True` ou `False`.

(b) Programmez une fonction `estNonOriente` qui prend en entrée un graphe représenté sous forme de liste d'adjacence et qui renvoie `True` si le graphe est non-orienté et `False` sinon.

L'algorithme de Prim permet d'obtenir un arbre couvrant de poids minimal, c'est-à-dire un graphe composé des n sommets de votre graphe initial, à $n-1$ arêtes qui relient tous les sommets entre eux, et pour lequel la somme des poids de chaque arête prise est minimale. On suppose dans un premier temps que le graphe en entrée est connexe. L'idée consiste, à mettre intialement tous les sommets du graphe dans une liste F et au fur et à mesure prendre ces sommets un à un tout en les enlevant de F. La question est "comment les prendre ?". Pour cela, on rajoute un poids (COUT) associé à ces "sommets à prendre" et on sauvegarde ces poids dans un tableau COUT (de taille n). On initialise toutes les cases de COUT à ∞ et au fur et à mesure de l'algorithme, ces poids vont diminuer en fonction des arêtes parcourues, de manière à sélectionner des nouvelles arêtes pour notre arbre couvrant qui sont les meilleurs choix (`extraireMin`). Chaque poids contenu dans COUT, si différent de ∞, correspond donc à un poids d'une arête du graphe, et quand on sélectionne avec `extraireMin` un sommet i, on a donc "pris" l'arête correspondante au poids contenu dans COUT$[i]$ pour construire notre arbre couvrant, qui par défintion de `extraireMin` est le meilleur choix. Pour garder en mémoire les arêtes qui vont être sélectionnées dans l'arbre, on utilise un tableau en plus noté ARBRE de taille n, où ARBRE$[i]$ contient un élément de la forme $[w, j]$ où w est le poids de l'arête $\{i, j\}$. Ce tableau est mis à jour à chaque fois qu'on trouve une arête qui permet de rejoindre i avec un coût moindre et ARBRE$[s_0]$ contient $[0, \emptyset]$, puisque s_0 est la racine et n'a donc pas de père dans l'arborescence couvrante. À la fin de l'algorithme, le tableau ARBRE contient donc les $n-1$ arêtes de l'arbre couvrant minimal : ARBRE$[v] = [w, u] \Leftrightarrow \{u, v\} \in \mathcal{A}_{min}$ où $\mathcal{A}_{min}$ désigne l'ensemble des arêtes de notre arbre couvrant. Plus formellement, l'algorithme de Prim est donné en pseudo-code.

(c) Programmez la fonction `extraireMin` qui prend en entrée la liste F et le tableau COUT et qui enlève de la liste F le sommet qui possède le plus petit COUT et le renvoie.

(d) Programmez l'algorithme de Prim.

(f) Modifiez votre algorithme, afin que celui-ci renvoie -1 si le graphe en entrée n'est pas connexe, mais sans utiliser de parcours en largeur ou en profondeur.

Pour vous aider, voici les premières étapes de l'algorithme de Prim sur le graphe donné en exemple avec $s_0 = 0$.

Algorithm 2 PRIM$(G = (S, A, w), s_0 \in S)$

pour $u \in S$ **faire**
  COUT$[u] = \infty$
  ARBRE$[u] = [0, \emptyset]$
fin pour
$F \leftarrow S$
COUT$[s_0] = 0$
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow$ `extraireMin`$(F,$ COUT$)$
  pour $v \in$ SUCC(u) **faire**
    si $v \in F$ **and** $w(u, v) <$ COUT$[v]$ **alors**
      COUT$[v] \leftarrow w(u, v)$
      ARBRE$[v] = [u, w(u, v)]$
    fin si
  fin pour
fin tant que

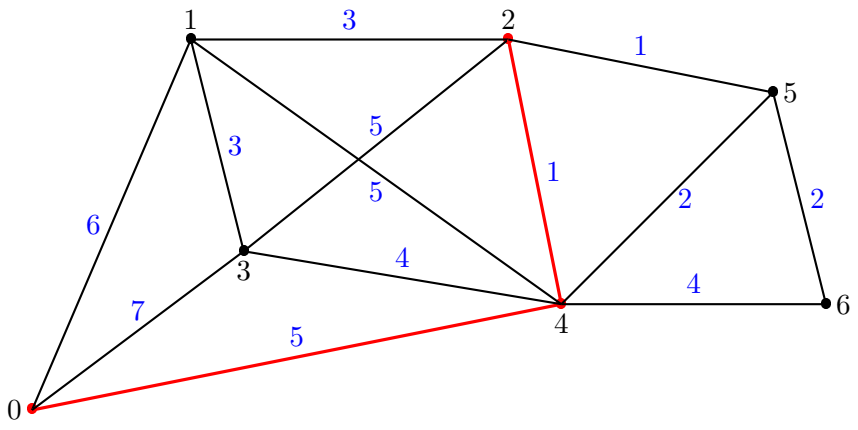
Après la première boucle tant que :

— $F = [1, 2, 3, 4, 5, 6]$, on prend le sommet $s_0 = 0$.
— COUT $= [0, 6, \infty, 7, 5, \infty, \infty]$
— ARBRE $= [[\emptyset, 0], [0, 6], [\emptyset, 0], [0, 7], [0, 5], [\emptyset, 0], [\emptyset, 0]]$

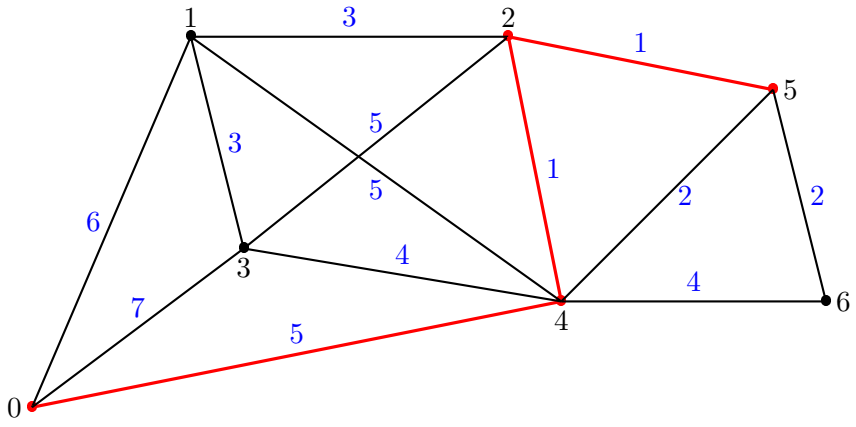
Après la deuxième boucle tant que :

— $F = [1, 2, 3, 5, 6]$, 4 est le sommet qui coûte le moins cher à relier, on le prend, et on a bien choisi l'arête $\{0, 4\}$, contenue dans ARBRE$[4]$.
— COUT $= [0, 5, 1, 4, 5, 2, 4]$, certains sommets qui n'étaient pas accessibles le sont, et d'autres sommets sont accessibles avec un coût moindre.
— ARBRE $= [[\emptyset, 0], [4, 5], [4, 1], [4, 4], [0, 5], [4, 2], [4, 4]]$, 0 reste la racine $(\emptyset)$ et on a mis à jour les arêtes les plus favorables dans ARBRE.

Après la troisième boucle :



Après la quatrième boucle :



Après la cinquième boucle (4 a déjà été pris) :