L3-INFO Algorithmique et Modélisation

UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Examen – 25 avril 2017 – durée 3 h

Sont interdits : les documents, les ordinateurs, les téléphones (incluant smartphone, tablettes,... tout ce qui contient un dispositif électronique).
Seuls les dictionnaires papier pour les personnes de langue étrangère sont autorisés.
Il sera tenu compte de la qualité de la rédaction et de la clarté de la présentation (2 pts).
Le barème indicatif : Exercice 1 : 4 pts ; Exercice 2 : 5 pts ; Exercice : 9 pts.

Exercice 1 : Découpage *(∼ 45 mn)*

On dispose d'une tige de longueur L qu'on souhaite découper pour produire des tiges plus petites de différentes longueur ; il y a k types de telles tiges. Chaque tige de type i ($1 \leqslant i \leqslant k$) a une longueur l_i que l'on suppose entière. Le nombre de tiges de chaque type que l'on peut produire n'est pas limité mais on cherche à éviter les pertes.

Par exemple, si $k = 3$, $l_1 = 7$; $l_2 = 3$; $l_3 = 6$, pour $L = 20$, on pourra produire deux tiges de type 1 et une de type 3. Pour une tige de longueur $L = 11$, on ne peut éviter une perte, avec par exemple une pièce de type 1 et une de type 2.

Étant donné des longueurs $l_1, \cdots l_k$ le problème est de décider s'il existe une découpe sans perte d'une tige de longueur L, c'est à dire s'il existe k entiers $n_1, \cdots n_k$ tels que

$$L = \sum_{i=1}^{k} n_i l_i.$$

1. **Algorithme naïf**
   Démontrer que l'algorithme glouton consistant à découper la tige en prenant en priorité la plus grande longueur d'abord ne minimise pas la perte.
2. **Algorithme récursif**
   Exprimer ce problème sous la forme d'une équation de récurrence (à la fois sur k et sur L) et en déduire un algorithme récursif. Calculer son coût (donner un ordre de grandeur).
3. **Algorithme de décision**
   En s'inspirant de l'algorithme de *rendu de monnaie*, proposer un algorithme ayant une complexité de l'ordre de $k \times L$. Modifier cet algorithme pour qu'il fournisse également une découpe sans perte $n_1, \cdots, n_k$ lorsqu'elle existe.
4. **(Bonus) Algorithme minimisant la perte**
   Modifier cet algorithme pour qu'il fournisse maintenant une découpe $n_1, \cdots, n_k$ qui minimise la perte.

Exercice 2 : Anagrammes *(∼ 45 mn)*

Une anagramme d'un mot est un autre mot ayant les mêmes lettres, mais dans un ordre différent. Par exemple `GUERISON` est une anagramme de `SOIGNEUR`, `ASPIRINE` de `PARISIEN` ou `MINISTRE` de `INTERIMS`.

L'objectifs de cet exercice est d'écrire un algorithme d'énumération de toutes les anagrammes d'un mot. Un mot sera représenté par un tableau de caractères `M[1...n]` de taille n.

1. Si le mot M possède n lettres différentes, donner le nombre total de configurations possibles des lettres pouvant représenter une anagramme. Donner l'ordre de grandeur de ce nombre pour $n = 8$.
2. Algorithme générique
   (a) Écrire un algorithme récursif qui énumère, c'est à dire `Visite` une et une seule fois, toutes les configurations possibles. (On utilisera un espace mémoire de l'ordre de grandeur de la taille du mot, la fonction `Visite` est supposée fournie et pourra par exemple tester l'appartenance de la configuration à un dictionnaire).
   (b) Justifier et illustrer cet algorithme en l'exécutant sur l'exemple `RIME`.
   (c) Faire la preuve de l'algorithme.

On dispose maintenant d'une fonction `PrefixeDico` qui indique si une configuration est préfixe d'un mot du dictionnaire

3. Adapter l'algorithme afin de n'explorer que les configurations dont le préfixe a été reconnu et faire sa preuve.

Les mots proposés peuvent maintenant comporter plusieurs fois la même lettre, par exemple `ETINCELLE` anagramme de `CLIENTELE` comporte plusieurs `L` et plusieurs `E`, cependant le mot `ETINCELLE` ne doit être `Visite` qu'une seule fois.

4. **(Bonus)** Comment adapter l'algorithme pour ne visiter qu'une seule fois chaque anagramme du mot ?

Problème : Tourner en rond *(∼ 1h30)*

On se donne un graphe orienté avec des arcs pondérés $\mathcal{G} = (\mathcal{X}, \mathcal{A}, \mathcal{P})$, $\mathcal{X}$ est l'ensemble des sommets, $\mathcal{A}$ l'ensemble des arcs et $\mathcal{P}$ les poids des arcs, $p(u, v)$ est le poids de l'arc entre le sommet u et le sommet v. L'algorithme de Dijkstra, vu en cours et en TD1/2, permet de construire l'ensemble des chemins de poids minimal issus d'un sommet s à tous les autres sommets accessibles à partir de s. [1]

Une des hypothèses pour que l'algorithme de Dijkstra soit correct est que les poids soient tous positifs.

1. Donner un exemple de graphe pondéré (avec certaines pondérations négatives) et d'exécution de l'algorithme de Dijkstra qui illustre la non correction de l'algorithme.

Lorsque les pondérations peuvent être négatives il n'existe pas forcément un plus court chemin.

1. On rappelle que le poids d'un chemin est la somme des poids des arcs qui le composent.

2. Donner un exemple de graphe pondéré pour lequel il n'existe pas de chemin de poids minimal entre s et un autre sommet accessible à partir de s.

On suppose qu'il existe un chemin de poids minimal entre s à chacun des autres sommets du graphe (il n'y a pas de circuit de poids négatif, un tel circuit est dit *absorbant*), on note $d^{\min}(x)$ le poids minimal d'un chemin de s à x

3. Démontrer que les variables $d^{\min}(.)$ vérifient l'équation de point fixe

$$d^{\min}(v) = \min_{(u,v)\in\mathcal{A}} \left\{ d^{\min}(u) + p(u,v) \right\} \text{ pour tout } v \in \mathcal{X}\backslash\{s\} \text{ avec } d^{\min}(s) = 0. \quad (1)$$

Indication : on pourra d'abord démontrer que $d^{\min}(v) \leqslant d^{\min}(u) + p(u,v)$ pour tout arc (u,v), puis montrer qu'il existe un arc (w,v) tel que $d^{\min}(v) = d^{\min}(w) + p(w,v)$.

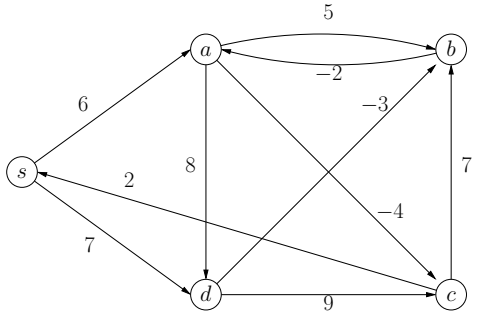
Considérons l'algorithme suivant :

Algorithme `CalculePoidsMin` ($\mathcal{G}$,s)
Données : Graphe pondéré $\mathcal{G} = (\mathcal{X}, \mathcal{A}, \mathcal{P})$,
$p(u,v)$ étant le poids de l'arc (u,v)
Résultat : Renvoie $d^{\min}(.)$

1 **foreach** $x \in \mathcal{X}\backslash\{s\}$ **do** $d(x) = +\infty$
2 $d(s) = 0$
3 **while** *il existe un arc* (u,v) *tel que* $d(v) > d(u) + p(u,v)$ ***do***
4 $\quad d(v) = \min\{d(v), d(u) + p(u,v)\}$
5 `Retourne` d

Algorithme 1 : Algorithme de calcul de $d^{\min}(.)$

4. Exécuter l'algorithme sur le graphe ci-dessous, on notera la séquence des arcs choisis et la suite des vecteurs d obtenus à la fin de chaque itération (ligne 4).



5. Si l'algorithme termine, démontrer que le vecteur d retourné est le point fixe de l'équation ci-dessus.
6. Démontrer que l'algorithme se termine. *Indication : trouver un variant de l'itération et utiliser le fait que le nombre de chemins "potentiellement de poids minimal" est fini.*
7. L'algorithme permet de calculer le poids minimal des chemins d'origine s. Modifier l'algorithme afin d'avoir le routage, c'est à dire, pour chaque sommet x, un chemin de poids minimal s à x.

*On remplace les lignes 3 et 4 (boucle **While**) de l'algorithme par*

```
6 repeat |X| − 1 fois
7     foreach (u, v) ∈ A do
8         d(v) = min {d(v), d(u) + p(u, v)}
```

8. (**Bonus**) S'il n'y a pas de circuit absorbant, montrer que le coût de cet algorithme peut être réduit à $|\mathcal{X}|.|\mathcal{A}|$.
9. Que se passe-t-il si le graphe possède un circuit absorbant ? Modifier l'algorithme pour qu'il détecte s'il existe un circuit absorbant ou renvoie le $d^{\min}(.)$.
10. Comparer cet algorithme avec l'algorithme de Dijkstra (on se donnera au préalable des critères de comparaison).