Informatique

DM

CENTRALE- MP- Option informatique - 2024

Pt.1

Créer un réseau routier entre différentes villes, en s'autorisant des intersections intermédiaires, dont la longueur totale est minimale est un problème très difficile à résoudre. Ce problème est une généralisation de deux problèmes pourtant faciles à résoudre : la recherche d'arbre couvrant de poids minimal, correspondant au cas où il n'y a pas d'intersections intermédiaires, et la recherche de plus court chemin, correspondant au cas où on ne souhaite relier que deux villes.

En 1836, un an seulement après l'apparition d'une ligne de train en Allemagne, Carl Friedrich Gauß s'interrogeait dans une lettre à l'astronome Christian Schumacher sur la meilleure manière de relier les villes de Hambourg, Brême, Hanovre et Brunswick par un réseau ferré.

FIGURE 1 – Les quatre villes allemandes et une manière théoriquement optimale de les relier.

De tels arbres trouvent des applications dans la construction de réseaux (électroniques, routiers, ...). Ils peuvent également apparaître dans des phénomènes naturels : des films de savon reliant des tiges entre deux plaques peuvent former un arbre avec des points intermédiaires.

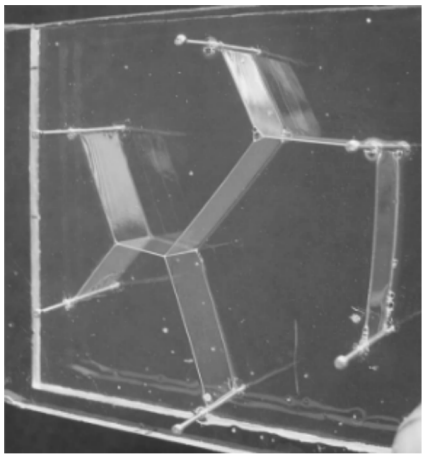

FIGURE 2 – Films de savon reliant des tiges métalliques entre deux plaques de plexiglas. Il y a trois sommets intermédiaires.

Consignes aux candidates et candidats

On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en `Computer Modern` à chasse fixe du point de vue informatique (par exemple `n`).

Les candidates et candidats devront répondre aux questions de programmation en utilisant le langage **OCaml**. S'ils écrivent une fonction non demandée par l'énoncé, ils devront préciser son rôle, ainsi que sa signature (son type). Les candidates et candidats sont également invités, lorsque c'est pertinent, à décrire le fonctionnement global de leurs programmes. On autorisera toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). Sauf précision de l'énoncé, l'utilisation d'autres modules sera interdite.

Lorsqu'une question de programmation demande l'écriture d'une fonction, la signature de la fonction demandée est indiquée à la fin de la question. Les candidates et candidats pourront écrire une fonction dont la signature est compatible avec celle demandée. Par exemple, si l'énoncé demande l'écriture d'une fonction `'int list -> int` qui renvoie le premier élément d'une liste d'entiers, écrire une fonction `'a list -> 'a` qui renvoie le premier élément d'une liste d'éléments quelconques sera considéré comme correct.

Une annexe disponible en fin de sujet rappelle différentes fonctions en OCaml.

Définitions et notations

Pour un ensemble fini E, on note $|E|$ le cardinal de E. On note $\mathcal{P}_2(E)$ l'ensemble des paires d'éléments de E, c'est-à-dire les sous-ensembles de E de cardinal 2.

On définit un **graphe non orienté** comme un couple $G = (S, A)$ tel que S est un ensemble fini d'éléments appelés *sommets* et A un ensemble de paires d'éléments distincts de S appelées *arêtes*, c'est-à-dire $A \subset \mathcal{P}_2(S)$. S'il n'y a pas d'ambiguïté, une paire $\{s, t\}$ sera notée st. Si st est une arête du graphe, on dit que s et t sont *adjacents*. Le graphe

$$G_1 = \Big(\{s_0, s_1, s_2, s_3, s_4, s_5\},\ \{s_0s_2,\ s_0s_3,\ s_0s_4,\ s_1s_2,\ s_1s_3,\ s_2s_3,\ s_2s_5,\ s_3s_4\}\Big)$$

est représenté sur la figure 3.

On appelle **sous-graphe** de $G = (S, A)$ un graphe $H = (X, B)$ tel que $X \subset S$ et $B \subset A \cap \mathcal{P}_2(X)$. Si $X \subset S$, on appelle **sous-graphe induit** par X le graphe $G[X] = (X, A \cap \mathcal{P}_2(X))$.

Pour $(s, t) \in V^2$, on appelle **chaîne de s à t** une suite de sommets distincts $(s_0, s_1, \ldots, s_k)$ telle que $s_0 = s$, $s_k = t$ et pour $i \in [\![1, k]\!]$, $s_{i-1}s_i \in A$. On appelle **cycle** de G une suite de sommets $(s_0, s_1, \ldots, s_k)$ telle que $k \geqslant 3$, $(s_0, \ldots, s_{k-1})$ est une chaîne, $s_0 = s_k$ et $s_0s_{k-1} \in A$.

Un graphe est dit **connexe** s'il existe toujours une chaîne entre deux sommets distincts. Si $G = (S, A)$, on appelle **composante connexe** de G un sous-ensemble X de S tel que $G[X]$ est connexe et pour tout $s \in S \setminus X$, $G[X \cup \{s\}]$ n'est pas connexe.

Un graphe est dit un **arbre** (à différencier des arbres enracinés) s'il est connexe et ne contient pas de cycle. On dit qu'un sous-graphe $T = (X, B)$ de $G = (S, A)$ est un **arbre couvrant** si $S = X$ et T est un arbre.

On appelle **graphe pondéré** un triplet (S, A, f) tel que (S, A) est un graphe et $f : A \to \mathbb{R}^+$ est une fonction qui à chaque arête associe un *poids* qui est un réel positif. La figure 3 contient une représentation graphique d'un graphe pondéré G_2.

Dans un graphe pondéré $G = (S, A, f)$, le **poids** d'un sous-graphe $H = (X, B, f)$ de G est la somme des poids des arêtes qui le composent, c'est-à-dire $f(H) = \sum_{a \in B} f(a)$.

FIGURE 3 – Les graphes G_1 et G_2.

1 Généralités

Q 1. Dessiner sans justifier un arbre couvrant de G_2 de poids minimal.

Q 2. Soit $G = (S, A)$ un graphe non orienté. Montrer les deux propriétés suivantes :

— Si G est connexe, alors $|A| \geqslant |S| - 1$.
— Si G est sans cycle, alors $|A| \leqslant |S| - 1$.

Q 3. En déduire l'équivalence entre les trois propriétés suivantes, pour $G = (S, A)$ un graphe non orienté :

— G est un arbre.
— G est connexe et $|A| = |S| - 1$.
— G est sans cycle et $|A| = |S| - 1$.

Pour les deux questions suivantes, on suppose que $G = (S, A, f)$ est un graphe pondéré tel que f est injective (toutes les arêtes ont un poids différent).

Q 4. Montrer que G possède un unique arbre couvrant de poids minimal.

Q 5. Soit $T^* = (S, B^*)$ l'unique arbre couvrant de poids minimal de G. On suppose que $a \in A$ est une arête dont le poids est maximal dans un cycle de G. Montrer que $a \notin B^*$.

Les trois questions qui suivent ont pour objectif d'implémenter un algorithme de tri efficace.

Q 6. Écrire une fonction `separation` telle que si `lst` est une liste, alors `separation lst` renvoie un couple $(\texttt{lst}_1, \texttt{lst}_2)$ tel que chaque élément de `lst` apparaît soit dans $\texttt{lst}_1$, soit dans $\texttt{lst}_2$, et les tailles de $\texttt{lst}_1$ et $\texttt{lst}_2$ diffèrent d'au plus 1.

```
1 separation : 'a list -> 'a list * 'a list
```

Q 7. Écrire une fonction `fusion` telle que si `lst1` et `lst2` sont deux listes triées par ordre croissant, alors `fusion lst1 lst2` renvoie une liste triée par ordre croissant contenant tous les éléments de `lst1` et `lst2`.

```
fusion : 'a list -> 'a list -> 'a list
```

Q 8. En déduire une fonction `tri_fusion` qui prend en argument une liste et renvoie une liste triée par ordre croissant contenant les mêmes éléments. Quelle est la complexité temporelle de cette fonction ? (On ne demande pas de justifier.)

```
tri_fusion : 'a list -> 'a list
```

2 Algorithme de Kruskal inversé

Q 9. L'algorithme de Kruskal permet de trouver un arbre couvrant de poids minimal dans un graphe pondéré connexe. En voici une description en pseudo-code.

Entrées : Graphe pondéré $G = (S, A, f)$ non orienté connexe

$B \leftarrow$ copie de A
Trier B par ordre décroissant de poids
pour $a \in B$ par ordre décroissant de poids **faire**
  si $(S, B \setminus \{a\})$ est connexe **alors**
    $B \leftarrow B \setminus \{a\}$
  fin si
fin pour
renvoyer (S, B)

Algorithme 1 – Algorithme de Kruskal inversé

Appliquer l'algorithme de Kruskal inversé au graphe G_3 de la figure 4. On ne demande pas la description de l'exécution détaillée, mais simplement une représentation graphique de l'arbre obtenu.

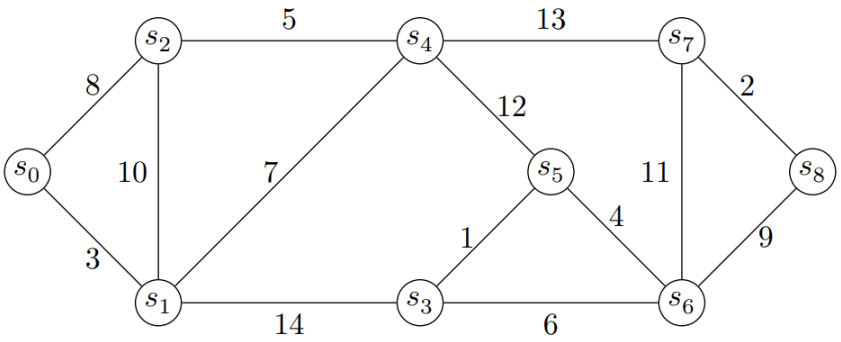

FIGURE 4 – Le graphe pondéré G_3.

On implémente un graphe pondéré par tableau de listes d'adjacences en OCaml, selon le type suivant :

```
type graphe = (int * float) list array
```

Si $G = (S, A, f)$ est implémenté par un objet `g` de type `graphe`, alors :

- l'ensemble des sommets est $S = [\![0, n-1]\!]$, où $n = |S|$;
- `g` est un tableau de taille n tel que pour tout $s \in S$, `g.(s)` est une liste contenant des couples (t, p) tels que $st \in A$ et $f(st) = p$.

Les listes d'adjacence ne sont pas nécessairement triées par ordre croissant des sommets ou des poids. Par exemple, le graphe G_2 de la figure 3 peut être créé par :

```
let g2 = [|[(2, 7.); (3, 6.); (1, 10.); (4, 7.)]; [(2, 5.); (0, 10.)];
          [(0, 7.); (1, 5.); (3, 9.)]; [(0, 6.); (2, 9.)]; [(0, 7.)]|]
```

On rappelle que les opérations de comparaisons (`max`, `min`, `<`, `>`, ...) peuvent s'effectuer sur des uplets selon l'ordre lexicographique.

Q 10. Écrire une fonction `tri_aretes` telle que si `g` est la représentation d'un graphe pondéré $G = (S, A, f)$, alors `tri_aretes g` renvoie la liste des triplets $(f(st), s, t)$, triée par ordre décroissant des $f(st)$, tels que $st \in A$ et $s < t$. Cette liste ne devra pas contenir de doublons.

```
tri_aretes : graphe -> (float * int * int) list
```

Q 11. Écrire une fonction `connectes` telle que si `g` est la représentation d'un graphe pondéré $G = (S, A, f)$, $(s, t) \in S^2$ et `poids_max` est un flottant, alors `connectes g s t poids_max` renvoie un booléen qui vaut `true` si et seulement s'il existe un chemin de s à t dans G ne passant que par des arêtes dont le poids est strictement inférieur à `poids_max`. La fonction devra avoir une complexité linéaire en $|S| + |A|$ et on demande de justifier brièvement.

```
connectes : graphe -> int -> int -> float -> bool
```

Q 12. En déduire une fonction `kruskal_inverse` qui prend en argument un graphe $G = (S, A, f)$ et renvoie le graphe obtenu selon le principe de l'algorithme de Kruskal inversé. On supposera que le graphe G donné en argument est connexe et que la fonction de pondération f est injective.

```
kruskal_inverse : graphe -> graphe
```

Q 13. Montrer que si G est connexe, le graphe renvoyé par l'algorithme de Kruskal inversé appliqué à G est un arbre couvrant de G.

Q 14. Déterminer la complexité temporelle de la fonction `kruskal_inverse`.

Annexe : rappels de programmation

Cette annexe rappelle le fonctionnement de différentes fonctions des modules `List` et `Array` :

- `List.iter (f: 'a -> unit)(lst: 'a list): unit` fait un appel à la fonction `f` pour chaque élément de la liste `lst`. Cette fonction s'exécute en temps linéaire en la taille de la liste, si la fonction `f` se calcule en temps constant ;
- `List.rev (lst: 'a list): 'a list` renvoie une liste contenant les mêmes éléments que `lst`, mais dans l'ordre inverse. Cette fonction s'exécute en temps linéaire en la taille de la liste ;
- `Array.length (tab: 'a array): int` renvoie la longueur du tableau `tab`. Cette fonction s'exécute en temps constant ;
- `Array.make (n: int)(x: 'a): 'a array` renvoie un tableau de longueur n dont toutes les cases contiennent la valeur `x`. Cette fonction s'exécute en temps linéaire en n ;
- `Array.make_matrix (m: int)(n: int)(x: 'a): 'a array array` renvoie une matrice de dimension $m \times n$ dont toutes les cases contiennent la valeur `x`. Cette fonction s'exécute en temps linéaire en $m \times n$.