TP n° 4 – Arbres couvrants minimaux

vie@jill-jenn.net/tp
thomas.h@ens.fr

1 Oups !

Aujourd'hui, Yorel Reivax fait joujou avec des vecteurs.

```
let n = 43 in
let t = make_vect n 0 in t.(n);;
```

C'est une erreur hyper classique et pourtant tu la fais. Les autres autour de toi ne la font plus mais toi, tu **continues** à la faire. Oui, c'est à **toi** que je parle.

2 Préliminaires

Un graphe $G = (V, \to)$ est défini par l'ensemble $V = \{1, \ldots, n\}$ de ses *sommets* et la relation d'adjacence $\to$ (on parle d'*arête* « st » lorsque $s \to t$, qui peut être noté $st \in \to$). Si la relation $\to$ est symétrique, on parlera de graphe *non orienté* ($s \leftrightarrow t$).

On dit qu'il existe un chemin entre s et t ($s \to^* t$) lorsqu'il existe des sommets $s_0, \ldots, s_r$ de V tels que $s = s_0 \to s_1 \to \ldots \to s_r = t$. Un *cycle* est un chemin d'un sommet vers lui-même. Un *arbre* est un graphe sans cycle. Un graphe non orienté est dit *connexe*[1] si pour tous sommets s et t, il existe un chemin de s à t.

On peut pondérer les arêtes d'un graphe avec une fonction $p : \to \to \mathbb{R}$, dite de *pondération*.

Liste d'arêtes	Liste des éléments de $\to$ avec leur poids et, à part, la taille de V.	`(int * int * int) list * int`
Matrice d'adjacence	`M.(i).(j)` $= \begin{cases} \texttt{Some}(p(ij)) \text{ si } i \to j \\ \texttt{None} \text{ sinon.} \end{cases}$	`int option vect vect`
Listes d'adjacence	`A.(i)` contient la liste des sommets adjacents à i (ses « voisins »), avec le poids de l'arête associée.	`(int * int) list vect`

TABLE 1 – Représentations les plus communes d'un graphe en mémoire.

Question 1. Écrire la fonction `lists_of_matrix` qui convertit une matrice d'adjacence en listes d'adjacence pour un graphe pondéré.

3 Arbre couvrant minimal

On considère maintenant un graphe non orienté G connexe et pondéré à poids positifs.

Un *arbre couvrant* du graphe $G = (V, \to)$ est un arbre $\mathcal{A} = (V, \to')$ connexe tel que $\to' \subset \to$. L'objectif de ce TP est de trouver un arbre couvrant minimal (soit, de poids minimum) de G.

3.1 Algorithme de Prim

L'algorithme de Prim construit un arbre couvrant minimal $\mathcal{A}$ en procédant de la façon suivante :
- $\mathcal{A} \leftarrow$ un sommet s quelconque de V.
- Parmi toutes les arêtes accessibles depuis $\mathcal{A}$, on en extrait une de poids minimal.
- Si elle relie deux sommets déjà ajoutés à l'arbre, on la jette avec détermination, sinon on l'ajoute à $\mathcal{A}$, on met à jour les arêtes accessibles depuis $\mathcal{A}$ et on retourne à l'étape précédente.

1. Un graphe orienté est dit *simplement connexe* quand $s \to^* t$, *fortement connexe* quand $s \to^* t$ et $t \to^* s$.

Question 2. Montrer que l'algorithme de Prim termine.

Pour l'implémenter, nous avons besoin d'une structure pour représenter l'ensemble des arêtes accessibles depuis $\mathcal{A}$. Cette structure (SPOILER : un *tas*, parfois appelé *file à priorité*) doit nous permettre efficacement :

- d'y insérer de nouveaux éléments
- et d'en extraire l'élément minimum.

Tout d'abord, nous allons simuler un tas avec une liste triée (SPOILER : ce n'est pas très efficace...).

Question 3. Écrire les fonctions `insère` et `extrait_min` pour une liste triée (avec une fonction de comparaison quelconque `compare` qui sera fournie en argument). Quelle est leur complexité ?

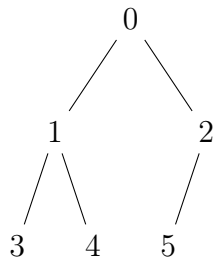
Question 4. Écrire la fonction `maj_accessibles` qui, quand un sommet est ajouté à l'arbre, met à jour (à l'aide de la fonction `insère`) la liste des arêtes accessibles depuis $\mathcal{A}$.

Question 5. Écrire une fonction `prim_bof` qui implémente l'algorithme de Prim en utilisant une liste triée. Quelle est sa complexité ?

3.2 Une implémentation avec des tas

Un *tas-min* est un arbre binaire qui vérifie les deux propriétés suivantes :

- Chaque nœud a une valeur supérieure à celle de son père (la racine a donc la valeur minimale).
- L'arbre est complet : pour un tas de taille n, si on numérote ses sommets par un parcours un largeur alors le père de i est $\lfloor \frac{i-1}{2} \rfloor$ et ses fils sont $2i+1$ et $2i+2$ [2].



La taille du tas étant appelée à être modifiée, celui-ci est représenté par un couple `(t, n)` où `t` est un tableau de la taille maximale dont on aura besoin et `n` la taille réelle du tas.

Question 6. Écrire une fonction qui transforme un tableau quelconque en tas-min. Vous vous aiderez d'une fonction `entasser` telle que `entasser prefix <` [3] `t i` suppose que les arbres enracinés en `2*i + 1` et `2*i + 2` dans le tas `t` sont bien des tas-min, vérifie que l'arbre enraciné en `i` est un tas-min et rétablit la propriété de tas-min si elle n'est pas vérifiée.

Question 7. Si un tableau `t` est un tas, alors tout tableau préfixe `[|t.(0);t.(1);...;t.(i-1)|]` est aussi un tas. Comment implémenter efficacement les fonctions `insère` et `extrait-min` pour un tas-min ? Faites-le. Quelle est leur complexité ?

Question 8. Écrire une fonction `prim_cool` qui implémente l'algorithme de Prim en utilisant un tas-min. Quelle est sa complexité ?

Question 9. L'algorithme de tri par tas utilise un *tas-max* [4] de la manière suivante :

- Transformer le tableau à trier en tas-max.
- Tant que la taille du tas est positive, échanger le premier et le dernier élément du tas puis rétablir la propriété de tas-max sur les $n-1$ éléments restants (on considère que le dernier élément ne fait plus partie du tas).

Implémenter cet algorithme.

2. Les tas sont donc le plus souvent représentés en mémoire par un tableau de taille n.
3. On choisit ici `prefix <` comme fonction de comparaison.
4. En remplaçant « supérieure » par « inférieure » dans la définition du *tas-min*, on obtient un *tas-max*.