DS2 (sujet type Mines Centrale)

1 Problème sur les arbres couvrants (C)

Dans ce problème, un graphe G est un couple (S, A) où :

- S est un ensemble fini dont les éléments sont les sommets de G ;
- $A = (a_0, a_1, \ldots, a_{m-1})$ est la suite des arêtes de G, une arête étant une partie $a = \{s, t\}$ de S de cardinal 2. Les sommets s et t sont appelés les extrémités de l'arête a et on dira que a relie s et t. Si s et t sont reliés par une arête, on dit qu'ils sont voisins ou adjacents.

Ainsi, les graphes sont non orientés et il n'y a pas d'arête reliant un sommet à lui-même.
Par convention, nous noterons n (respectivement m) le nombre de sommets (respectivement d'arêtes) du graphe et nous supposerons que $S = \{0, 1, \ldots, n-1\} = S_n$.

Un graphe sera représenté par le type C suivant :

```
struct graphe = {int size;
                 int* degrees;
                 int** adj;};

typedef struct graphe graphe;
```

Si G est un graphe représenté par `g`, alors le nombre de sommets `n` du graphe est donné par le champs size, le tableau `g.degrees` de taille n contient dans sa case d'indice `i` le degré du sommet `i` et le tableau `g.adj[i]` est de taille `g.degrees[i]` et contient les voisins du sommet `i` dans G.

Soit $G = (S, A)$ un graphe.

- Un chemin dans G est une suite $c = (s_0, s_1, \ldots, s_{j-1}, s_j, \ldots, s_{k-1}, s_k)$ où pour tout j compris entre 1 et k, s_{j-1} et s_j sont des sommets voisins. On dira que c est un chemin de s_0 à s_k de longueur k. Par convention, pour s sommet de G, il existe un chemin de longueur nulle de s à s.
- La composante connexe d'un sommet s de G, notée C_s , est l'ensemble des sommets t de G tels qu'il existe un chemin de s à t.
- On dit que G est connexe si pour tous sommets s et t de G, il existe un chemin de s à t.
- Un cycle dans G est un chemin de longueur $k \geq 2$ d'un sommet à lui-même et dont les arêtes sont deux à deux distinctes. On dit que G est acyclique s'il ne contient aucun cycle.
- Un arbre est un graphe connexe acyclique.

1.1 Caractérisation des arbres

Soit $G = (S_n, A)$ un graphe à n sommets et m arêtes, représenté par `g`. Les arêtes de G sont donc numérotées $A = (a_0, a_1, \ldots, a_{m-1})$.

Q1 Montrer que les composantes connexes de G forment une partition de S_n , c'est-à-dire que :

1. $\forall s \in S_n, C_s \neq \emptyset$;
2. $S_n = \bigcup_{s \in S_n} C_s$;
3. Pour tous sommets s et t, soit $C_s = C_t$, soit $C_s \cap C_t = \emptyset$.

Q2 Montrer que si s et t sont deux sommets de G tels que $t \in C_s$, il existe un plus court chemin de s à t et que les sommets d'un plus court chemin sont deux à deux distincts.

Pour $k \in \{0, 1, \ldots, m\}$, G_k désigne le graphe $(S_n, (a_0, a_1, \ldots, a_{k-1}))$ obtenu en ne conservant que les k premières arêtes de G.

Q3 On suppose que G est un arbre. Montrer que pour tout $k \in \{0, 1, \ldots, m-1\}$, les extrémités de a_k appartiennent à deux composantes connexes différentes de G_k. En déduire que $m = n - 1$.

Q4 Montrer que les trois propriétés suivantes sont équivalentes :

1. G est un arbre ;
2. G est connexe et $m = n - 1$.
3. G est acyclique et $m = n - 1$.

Nous souhaitons écrire une fonction qui teste si un graphe est un arbre. Nous allons pour cela utiliser une structure de données permettant de manipuler les partitions de l'ensemble S_n : si $\mathcal{P} = \{X_1, X_2, \ldots, X_k\}$ est une partition de S_n, on choisit dans chaque partie X_i, un élément particulier r_i, appelé représentant de X_i. Notre structure de données doit nous permettre :

- de calculer, pour $s \in S_n$, le représentant de la partie X_i contenant s ; cet élément sera également appelé représentant de s ;
- pour deux entiers s et t représentant des parties distinctes X_i et X_j, de transformer $\mathcal{P}$ en réunissant X_i et X_j, s ou t devenant le représentant de la partie $X_i \cup X_j$.

Nous représenterons une partition $\mathcal{P} = \{X_1, X_2, \ldots, X_k\}$ de S_n par une forêt : chaque X_i est représenté par un arbre dont les noeuds sont étiquetés par les éléments de X_i et de racine le représentant r_i de X_i , les arcs étant orientés vers la racine. Nous noterons $h(r_i)$ la hauteur de l'arbre X_i , c'est-à-dire la longueur de sa plus longue branche.

Ainsi, $\mathcal{P}_9 =$ `{{0, 2}, {1, 5, 6, 8}, {3, 4, 7}}` est une partition de S_9 et peut par exemple être représentée par la forêt de la figure suivante :

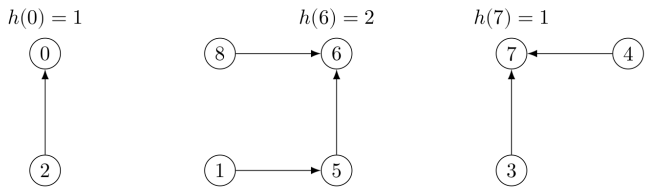

Figure 3 Une représentation de $\mathcal{P}_9 = \{\{0, 2\}, \{1, 5, 6, 8\}, \{3, 4, 7\}\}$

Le calcul du représentant d'un entier $s \in S_n$ se fait donc en remontant jusqu'à la racine de l'arbre (ce qui justifie l'orientation des arcs). Dans l'exemple, les représentants de 2, 1 et 7 sont respectivement 0, 6 et 7. Une partition $\mathcal{P}$ de $S_n = \{0, 1, \ldots, n-1\}$ sera représentée par un tableau d'entiers `P` de taille `n` de sorte que pour tout $s \in \{0, 1, \ldots, n-1\}, P[s]$ est le père de s si s n'est pas un représentant, et est égal à $-1 - h(s)$ si s est un représentant. La partition $\mathcal{P}_9$ peut ainsi être représentée par le tableau : `{-2, 5, 0, 7, 7, 6, -3, -2, 6}`

La réunion de deux parties X_i et X_j de représentants s et t distincts se fait selon la méthode suivante :

- si $h(s) > h(t)$, s est choisi pour représentant de la partie $X_i \cup X_j$ et devient le père de t ;
- si $h(s) \leq h(t)$, t est choisi pour représentant de la partie $X_i \cup X_j$ et devient le père de s.

Q5 Donner le résultat de la réunion de X_1 et X_2 dans $\mathcal{P}_9$.

Q6 Écrire une fonction `int representant(int* tab, int size, int s)` qui, appliquée à un tableau représentant une partition de S_n de taille `size` et à $s \in S_n$, renvoie le représentant de s. On demande que la fonction proposée soit récursive.

Q7 Écrire une fonction `void union(int* tab, int size, int s, int t)` qui, appliquée à un tableau représentant une partition de S_n de taille `size` et à deux représentants s et t distincts, modifie la partition en réunissant les arbres associés à s et t, selon la méthode expliquée ci-dessus, sans oublier, si nécessaire, de modifier $h(s)$ ou $h(t)$.

On note $\mathcal{P}_n(0)$ la partition de S_n où toutes les parties sont des singletons.

Q8 Soit $\mathcal{P}$ une partition de S_n construite à partir de $\mathcal{P}_n(0)$ par des réunions successives selon la méthode précédente. Montrer que si s est le représentant d'une partie $X \in \mathcal{P}$, alors le cardinal de X vérifie $|X| \geq 2^{h(s)}$.

Q9 En déduire les complexités des deux fonctions précédentes dans le pire des cas en fonction de n pour une partition $\mathcal{P}$ construite à partir de $\mathcal{P}_n(0)$.

Q10 Citer et expliquer le principe de la méthode au programme qui permet d'améliorer la complexité amortie de ces opérations. Ecrire une fonction `representant_opt` qui inclut cette optimisation.

Q11 Écrire une fonction `bool est_un_arbre(graphe g)` qui, appliquée à un graphe `g` représentant un graphe G, renvoie `true` si G est un arbre et `false` sinon.

1.2 Algorithme de Wilson : arbre couvrant aléatoire

Soit $G = (S_n, A)$ un graphe connexe et soit un entier $r \in S_n$. Un arbre couvrant de G enraciné en r est un arbre $T = (S_n, B)$ tel que $B \subset A$ et dont r a été choisi pour racine.

On convient de représenter un tel arbre en suivant la même idée que pour représenter une partition dans la partie précédente : par un tableau tel que la case s du tableau contient l'indice du père de s dans l'arbre si s n'est pas lui-même la racine, et -1 sinon :

```
struct arbre = {int size; int* peres;};
typedef struct arbre arbre;
```

Dans cette partie, nous supposons que $G = (S_n, A)$ est un graphe connexe et r un sommet de ce graphe. Nous cherchons à construire un arbre couvrant aléatoire de G enraciné en r.

Nous allons pour cela faire évoluer dynamiquement un arbre T , représenté par un tableau parent de longueur n vérifiant :

- la case r de parent contient -1 ;
- si $s \in S_n$ n'est pas un sommet de T, alors la case s de parent contient -2;
- si $s \in S_n$ est un sommet de T différent de la racine, alors la case s de parent contient le père de s dans T.

La construction de l'arbre T se fait en suivant l'algorithme 1.

Algorithme 1 Algorithme de Wilson de création d'un arbre couvrant aléatoire

entrée : G graphe connexe, r sommet de G
sortie : arbre couvrant aléatoire de G enraciné en r

```
1 : T ← ({r}, ∅)
2 : pour tout sommet s de G faire
3 :     si s n'est pas un sommet de T alors
4 :         c ← chemin-aléatoire(G, T, s)
5 :         greffer(c, T)
6 :     fin si
7 : fin pour
8 : renvoyer T
```

La greffe d'un chemin élémentaire qui termine par un sommet de T et dont c'est le seul sommet en commun avec T se fait de manière naturelle, en ajoutant ce chemin.

Le calcul d'un chemin aléatoire dans le graphe entre un sommet qui n'appartient pas à l'arbre et un sommet qui appartient à l'arbre se fait en suivant l'algorithme 2.

Algorithme 2 Algorithme chemin-aléatoire

entrée : G graphe connexe, T arbre sous-graphe de G, s sommet de G et pas de T
sortie : $c = (s_0 = s, s_1, \ldots, s_k)$ chemin aléatoire élémentaire de G partant de s dont le seul sommet dans T est s_k

$c \leftarrow (s)$
tant que le dernier sommet de c n'appartient pas à T **faire**
    on note $c = (s_0, s_1, \ldots, s_k)$
    soit u un voisin de s_k dans G (choisi aléatoirement et uniformément)
    si il existe $i \in [\![0, k]\!]$ tel que $u = s_i$ **alors**
        $c \leftarrow (s_0, \ldots, s_i)$
    sinon $c \leftarrow (s_0, \ldots, s_k, u)$
    fin si
fin tant que
renvoyer c

On représente un chemin élémentaire en C par le type :

```
struct chemin = {int debut; int fin; int* suivant; int size;};
typedef struct chemin chemin;
```

de telle sorte que si le chemin $c = (s_0, s_1, \ldots, s_k)$ est représenté par c, alors :

- le champ `debut` de `c` contient s_0,
- son champ `fin` contient s_k,
- son champ `suivant` est un tableau de taille `size` et pour tout $j \in \{0, \ldots, k-1\}$, la case d'indice s_j de ce tableau contient la valeur s_{j+1},
- les valeurs des cases du champ `suivant` dont les indices n'appartiennent pas à $\{s_0, \ldots, s_{k-1}\}$ n'ont pas d'importance.

Q12 Expliquer pourquoi le type `chemin` ne peut pas représenter un chemin non élémentaire.

Q13 Écrire une fonction `void greffer(arbre* a, chemin* c)` qui implémente la greffe d'un chemin sur un arbre, en supposant (sans avoir à le vérifier) que le chemin possède un unique sommet qui appartient à l'arbre : son dernier sommet.

Q14 Écrire une fonction `chemin* chemin_aleatoire(graphe g,arbre* a,int s)` qui exécute l'algorithme 2. On pourra utiliser la commande `rand()%n` quirenvoie uniformément un entier compris entre 0 et $n-1$.

Q15 Écrire une fonction `arbre* wilson(graphe g,int r)` qui implémente l'algorithme 1. On prendra les initiatives nécessaires pour éviter les fuites mémoire tout en respectant les prototypes des fonctions imposées.

2 Problème sur les automates (Ocaml)

Langages et mots

On appelle *alphabet* tout ensemble fini de lettres. On note généralement l'alphabet Σ.

On note $\Sigma^\star$ l'ensemble de tous les mots formés sur l'alphabet Σ.

La *longueur* (ou la *taille*) d'un mot $w \in \Sigma^\star$ est son nombre de lettres et se note $|w|$. Le *mot vide*, noté ε, est le seul mot de longueur nulle.

Si un mot $w \in \Sigma^\star$ est de longueur $|w| = n$, on le note $w = a_0 a_1 ... a_{n-1}$, où les a_i sont des lettres de Σ.

Un *langage* sur l'alphabet Σ est un ensemble $L \subset \Sigma^\star$.

L'*étoile de Kleene* d'un langage L, notée $L^\star$, est le plus petit langage qui inclut L, qui contient ε et qui est stable par concaténation.

La concaténation de deux langages L et L' est notée $L \cdot L'$, souvent abrégé en LL' lorsqu'il n'y a pas d'ambiguïté.

Automates finis

Un *automate fini non déterministe* sur un alphabet Σ est un quadruplet $A = (Q, I, F, T)$, où Q est un ensemble fini d'*états*, $I \subset Q$ est le sous-ensemble des *états initiaux*, $F \subset Q$ est le sous-ensemble des *états finaux* et l'ensemble $T \subset Q \times \Sigma \times Q$ est l'ensemble des *transitions*, étiquetées par les lettres de l'alphabet Σ.

Si $(q, a, q') \in T$, on note $q \xrightarrow{a} q'$ cette transition.

Pour représenter graphiquement un automate, on utilise une flèche entrante pour désigner un état initial et une flèche sortante pour désigner un état final, comme l'illustre l'exemple de la figure 1.

Un mot $w = a_0 ... a_{n-1}$ est reconnu par l'automate A s'il existe une succession de transitions :

$$q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \; q_{n-1} \xrightarrow{a_{n-1}} q_n \qquad \text{avec} \qquad q_0 \in I \quad \text{et} \quad q_n \in F.$$

On dira que le mot w *étiquette un chemin* dans l'automate A allant de q_0 à q_n.

Le langage d'un automate A, noté L_A, est exactement l'ensemble des mots reconnus par l'automate A. On dit alors que A *reconnait* L_A. Un langage est dit *reconnaissable* s'il est le langage d'un automate fini.

Un *automate fini déterministe* sur un alphabet Σ est un quadruplet $A = (Q, \{q_0\}, F, \delta)$, où l'ensemble des états initiaux est un singleton (un unique état initial) et où l'ensemble des transitions T est remplacé par une fonction de transition δ définie sur un sous-ensemble de $Q \times \Sigma$ et à valeurs dans Q. Pour chaque couple $(q, a) \in Q \times \Sigma$, il existe au plus une transition (q, a, q') qui, si elle existe, est telle que $q' = \delta(q, a)$.

L'automate est déterministe *complet* si la fonction de transition δ est définie sur $Q \times \Sigma$. Dans ce cas, on définit la *fonction de transition étendue* $\delta^\star$ sur $Q \times \Sigma^\star$ par

$$\forall q \in Q, \quad \begin{cases} \delta^\star(q, \varepsilon) = q \\ \delta^\star(q, wa) = \delta\big(\delta^\star(q, w), a\big) \end{cases} \quad \forall w \in \Sigma^\star, \forall a \in \Sigma$$

Les automates seront représentés par le type Caml suivant

```
type automate = { nb : int;                                  (* nombre d'états *)
                  init : int list ;                          (* états initiaux *)
                  final : int list;                          (* états finaux *)
                  trans : (int * char * int) list} ;;        (* transitions *)
```

l'ensemble d'états Q d'un automate implémenté étant toujours supposé être un intervalle d'entiers $[\![0, n-1]\!]$.