DS 3

CHEMIN DE LARGEUR MAXIMUM

Durée : 4h.

Définitions et notations

- Dans l'ensemble du sujet et sauf mention contraire explicite, on considère des graphes non orientés.
- Un *graphe* est donc un couple (V, E) où $E \subseteq \mathcal{P}_2(V)$ (ensemble des parties à deux éléments de V).
- S'il n'y a pas d'ambiguïté, une arête $\{x, y\}$ pourra être notée xy.
- On notera $G + xy$ le graphe G auquel on a ajouté l'arête xy et $G - xy$ le graphe G auquel on a enlevé l'arête xy.
- Un *graphe pondéré* est un triplet (V, E, f) où $f : E \to \mathbb{R}_+$ est une fonction dite de *pondération*.

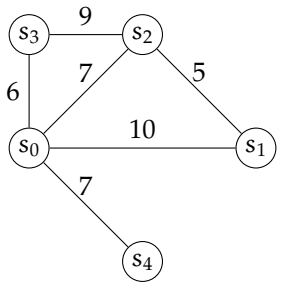

FIGURE 1 – Le graphe G_0.

I Généralités

► **Question 1** Dessiner sans justifier un arbre couvrant de G_0 de poids minimal.

► **Question 2** Soit $G = (V, E)$ un graphe non orienté. On fixe une numérotation $\{a_1, \dots, a_p\}$ des arêtes, et l'on note $G_i = (V, \{a_1, \dots, a_i\})$ pour $0 \leqslant i \leqslant p$ (le graphe G_0 n'a donc aucune arête). Montrer par récurrence que le graphe G_i possède au moins $n - i$ composantes connexes, et en déduire que si G est connexe, alors $|E| \geqslant |A| - 1$.

► **Question 3** Avec les notations précédentes, montrer que si G_i possède au moins $n - i + 1$ composantes connexes, alors G_i possède un cycle. En déduire que si G est acyclique, alors $|E| \leqslant |V| - 1$.

► **Question 4** En déduire l'équivalence entre les trois propriétés suivantes, pour $G = (V, E)$ un graphe non orienté :

(a) G est un arbre ;

(b) G est connexe et $|E| = |V| - 1$;

(c) G est sans cycle et $|E| = |V| - 1$.

Pour les deux questions suivantes, on suppose que $G = (V, E, f)$ est un graphe pondéré avec une fonction de pondération f injective (deux arêtes distinctes ne peuvent donc pas avoir le même poids).

Si X est une partie de V, on note $E(X)$ l'ensemble des arêtes $xy \in E$ telles que $x \in X$ et $y \notin x$.

► **Question 5** Soit $X \subset V$ telle que $X \neq \emptyset$ et $\overline{X} \neq \emptyset$ (où $\overline{X}$ est le complémentaire de X dans V). Montrer que si T est un arbre couvrant minimal de G, alors T contient l'arête de poids minimal de $E(X)$.

► **Question 6** Montrer que G possède un unique arbre couvrant de poids minimal.

2 Principe de l'algorithme de Prim

2.1 Étude théorique

On considère l'algorithme ci-dessous, dit *algorithme de Prim*.

Algorithme 1 – Prim

Entrées : Un graphe pondéré non orienté connexe $G = (V, E, \rho)$.
Sorties : Un arbre couvrant minimal T de G.

$F \leftarrow \emptyset$
$X \leftarrow \{x_0\}$ (un sommet quelconque de G)
tant que $X \neq V$ **faire**
  Trouver $xy \in E$ de poids minimal telle que $x \in X$ et $y \in V \setminus X$.
  $F \leftarrow F \cup \{xy\}$
  $X \leftarrow X \cup \{y\}$
renvoyer $T = (X, F)$

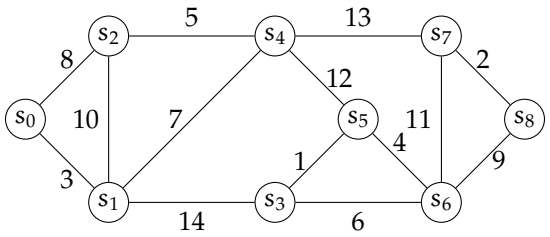

Figure 2 – Le graphe pondéré G_1.

▶ **Question 7** Appliquer l'algorithme de Prim au graphe G_1 de la figure 2. On ne demande pas la description de l'exécution détaillée mais simplement une représentation graphique de l'arbre obtenu.

▶ **Question 8** Montrer que l'algorithme de Prim (appliqué à un graphe G connexe) termine et renvoie un arbre couvrant de G.

▶ **Question 9** On suppose que G est un graphe pondéré connexe dont la fonction de pondération est injective. Montrer que l'algorithme de Prim renvoie un arbre couvrant de poids minimal de G.

▶ **Question 10** Généraliser ce résultat sans l'hypothèse d'injectivité de la fonction de pondération.

2.2 Implémentation

On représente les graphes en C sous forme de tableau de listes d'adjacence avec les structures suivantes :

```c
struct edge {
    int x;
    int y;
    double weight;
};
typedef struct edge edge_t;

struct graph {
    int n;
    int *degrees;
    edge_t **adj;
};
typedef struct graph graph_t;
```

- Pour un graphe G, représenté par un objet `g` de type `graph_t` :
  - `g.n` est le nombre de sommets de G, les sommets étant numérotés $0, \dots, n-1$;
  - `g.degrees` est un tableau de n entiers tel que `g.degrees[i]` soit le degré du sommet `i` ;
  - `g.adj` est un tableau de taille n, tel que `g.adj[i]` soit un tableau de taille `g.deg[i]`.
- Pour un sommet u, les éléments du tableau `g.adj[u]` correspondent aux arêtes incidentes au sommet u. Si `g.adj[u][j]` est égal à `e`, alors `e.x` sera égal à u et `e.y` indiquera l'autre extrémité de l'arête. `e.weight` correspond au poids de l'arête.
- Ainsi, chaque arête uv du graphe sera représentée deux fois dans la structure :
  - une fois comme élément de `g.adj[u]`, avec le champ `x` égal à u et le champ `y` égal à v ;
  - une fois comme élément de `g.adj[v]`, avec le champ `x` égal à v et le champ `y` égal à u.

► **Question 11** Écrire une fonction `nb_edges` prenant en entrée un pointeur vers un graphe G et renvoyant son nombre d'arêtes.

```
int nb_edges(graph_t* g);
```

Pour implémenter efficacement l'algorithme de Prim, on va utiliser une structure de file de priorité. Ces files de priorité contiendront des arêtes (des objets de type `edge_t`) et utiliseront les poids comme priorités. On suppose que la structure a déjà été implémentée en utilisant un tas binaire, et que l'interface suivante est fournie :

```
// Création d'une file vide.
// L'argument capacity indique le nombre maximal
// d'arêtes que la file pourra contenir.
// La complexité de cette fonction est en O(capacity).
heap_t *heap_create(int capacity);

// Libération des ressources associées à une file
// Complexité O(1)
void heap_free(heap_t *heap);

// Détermine si la file est vide
// Complexité O(1)
bool heap_is_empty(heap_t *heap);

// Ajoute une arête à la file.
void heap_push(heap_t *heap, edge_t pair);

// Renvoie l'arête de poids minimal présente dans la file,
// et la supprime de la file.
// Erreur si la file est vide.
edge_t heap_extract_min(heap_t *heap);
```

► **Question 12** Rappeler rapidement le principe de l'insertion d'un élément dans un tas binaire et de l'extraction du minimum, et en déduire les complexités des fonctions `heap_push` et `heap_extract_min` (en supposant bien sûr qu'elles sont correctement implémentées).

► **Question 13** Écrire une fonction `prim` prenant en entrée un graphe $G = (V, E, f)$, supposé connexe, et renvoyant un arbre couvrant minimum de G sous la forme d'un tableau de $|V| - 1$ arêtes. On demande une complexité en $O(|E| \log |V|)$, que l'on justifiera.

```
edge_t *prim(graph_t *g);
```

► **Question 14** Rappeler le principe de l'algorithme de Kruskal et sa complexité temporelle en fonction de $|V|$ et $|E|$ lorsqu'il est appliqué à un graphe pondéré connexe $G = (V, E, f)$. Comparer avec l'algorithme de Prim.

3 Application au problème du goulot maximal

On appelle *capacité* d'un chemin $\sigma = (x_0, \dots, x_n)$, et l'on note $c(\sigma)$, le minimum des poids de ses arêtes :

$$c(\sigma) = \min_{0 \leqslant i < n-1} f(x_i x_{i+1})$$

On note $c^*_G(x, y)$ la capacité maximale entre x et y dans G, c'est-à-dire la valeur maximale de $c(\sigma)$ pour σ un chemin de G reliant x à y. On appelle *goulot maximal* de x à y un chemin de x à y de capacité maximale. Dans toute cette partie, on suppose que le graphe $G = (V, E, f)$ est connexe.

▶ **Question 15** Déterminer, sans justification, un goulot maximal entre les sommets s_0 et s_8 du graphe G_1 de la figure 2, ainsi que la valeur de $c^*(s_0, s_8)$.

▶ **Question 16** Soient $s, t \in V$ et $T = (V, F)$ un arbre couvrant **maximal** de G. Montrer qu'un chemin de s à t dans T est nécessairement un goulot maximal de s à t dans G.

▶ **Question 17** Expliquer comment calculer en temps $O(|E| \log |V|)$ un arbre couvrant maximal de G (on ne demande pas d'écrire le code).

On suppose à présent que l'on dispose d'une fonction

```
graph_t* mst(graph_t* g);
```

qui prend en entrée un graphe pondéré connexe G et renvoie un arbre couvrant maximal de G (sous la forme d'un graphe pondéré).

▶ **Question 18** Écrire une fonction `parent` qui prend en argument un graphe $T = (V, F)$, dont on garantit qu'il est un arbre, et un sommet $u \in V$ et renvoie un tableau d'entiers `t` de taille $n = |V|$ tel que :

- `t[u]` vaut -1 ;
- pour $v \neq u$, `t[v]` est le parent de v dans l'arbre T enraciné en u.

On exige une complexité linéaire en le nombre n de sommets du graphe, que l'on justifiera.

```
int *parent(graph_t *g, int u);
```

▶ **Question 19** En déduire une fonction `int *goulot_max(graphe G, int s, int t, int* lc)` qui prend en argument un graphe et deux sommets `s` et `t` de ce graphe, ainsi qu'un pointeur `lc` vers un entier et renvoie un goulot maximal entre `s` et `t` sous forme d'un tableau constitué des sommets de ce chemin. La fonction devra modifier l'entier pointé par `lc` pour y stocker la taille du chemin ainsi renvoyé.

4 Interlude : médiane en temps linéaire

Jusqu'à la fin du sujet, les questions de programmation sont à traiter en langage OCaml.

On s'intéresse dans cette partie au calcul de la médiane d'une liste en temps linéaire en la taille de la liste. On va pour cela résoudre le problème plus général de la *sélection*.

Si $t = x_0, \dots, x_{n-1}$ est une liste de nombres, on définit $\texttt{select}(t, k)$ comme l'élément qui serait en position k dans la version triée par ordre croissant de t (en numérotant les positions à partir de 0). Ainsi :

- $\texttt{select}(t, 0)$ est le minimum de t ;
- $\texttt{select}(t, n-1)$ est le maximum de t ;
- $\texttt{select}(t, \lfloor n/2 \rfloor)$ est la médiane de t, que l'on notera $\texttt{med}(t)$.

▶ **Question 20** Écrire une fonction OCaml `separe` prenant en entrée une liste de `u` d'éléments de type `'a` et une valeur `seuil` de type `'a` et renvoyant un quadruplet `(petits, grands, nb_petits, nb_egaux)` tel que :

- `petits` est la liste des éléments de `u` strictement inférieurs à `seuil` ;
- `grands` est la liste des éléments de `u` strictement supérieurs à `seuil` ;
- `nb_petits` est la longueur de `petits` ;
- `nb_egaux` est le nombre d'éléments de `u` égaux à `seuil`.

On exige une complexité linéaire en la longueur de `u`.

```
val separe : 'a list -> 'a -> 'a list * 'a list * int * int
```

▶ **Question 21** On suppose définie une fonction `mediane_bloc` qui prend en entrée une liste non vide **de taille au plus 5** et renvoie sa médiane. Cette fonction s'exécute (évidemment) en temps constant.

Écrire une fonction `medianes_5` prenant en entrée une liste $u = u_0, \dots, u_{n-1}$ où $n = 5k + r > 0$ avec $0 \leqslant r < 5$ et renvoyant la liste, de longueur $\lceil n/5 \rceil$, constituée de :

- la médiane de $(u_0, \dots, u_4)$;
- la médiane de $(u_5, \dots, u_9)$;
- ...
- la médiane de $u_{5(k-1)}, \dots, u_{5k-1}$;
- si $r \neq 0$, la médiane de $(u_{5k}, \dots, u_{5k+r-1})$.

On exige une complexité linéaire en la longueur de la liste.

```
val medianes_5 : 'a list -> 'a list
```

On propose à présent la fonction suivante pour le problème de la sélection :

```
1  let rec select u k =
2    let n = List.length u in
3    if n = 1 then List.hd u
4    else
5      let liste_medianes = medianes_5 u in
6      let nb_medianes = 1 + (n - 1) / 5 in
7      let mom = select liste_medianes (nb_medianes / 2) in
8      let petits, grands, nb_petits, nb_egaux = separe u mom in
9      if k < nb_petits then ...
10     else if k < nb_petits + nb_egaux then ...
11     else select grands ...
```

▶ **Question 22** Compléter, en justifiant très rapidement, les lignes 9 à 11.

▶ **Question 23** Justifier que la complexité dans le pire cas $T(n)$ de cette fonction vérifie $T(n) \leqslant T(n/5) + T(7n/10) + An$ pour une certaine constante A.

▶ **Question 24** En déduire, en justifiant soigneusement, qu'il existe une constante B telle que $T(n) \leqslant Bn$.

5 Goulot maximal en temps linéaire

5.1 Étude théorique

On va maintenant chercher à résoudre le problème en temps linéaire en la taille de G (que l'on supposera connexe).

▶ **Question 25** Montrer que si $c^*(G, x, y)$ est connu, alors on peut déterminer un goulot maximal entre x et y en temps linéaire en la taille de G.

Pour $w \in \mathbb{R}$, on définit $E(w) = \{e \in E \mid f(e) \geqslant w\}$ et $G(w) = (V, E(w), f)$.
Soient alors :

- $G_0(w), \dots, G_{q-1}(w)$ les composantes connexes de $G(w)$;
- pour $i, j \in [0 \dots q-1]$,
$$S(w, i, j) = \{xy \in E \mid x \in G_i(w) \text{ et } y \in G_j(w)\}.$$

On définit $\overline{G}(w)$ comme le graphe non orienté dont :

- l'ensemble de sommets est $G_0(w), \dots, G_{q-1}(w)$;
- l'ensemble d'arêtes est l'ensemble des ij tels que $S(w, i, j)$ soit non vide ;
- le poids de l'arête ij est $\max_{e \in S(w,i,j)} f(e)$.

▶ **Question 26** Montrer que pour tout réel w, on a :

- si x et y sont dans la même composante $G_i(w)$, alors $c^*(G, x, y) = c^*(G_i(w), x, y)$;
- si $x \in G_i(w)$ et $y \in G_j(w)$, avec $i \neq j$, alors $c^*(G, x, y) = c^*(\overline{G}(w), i, j)$.

▶ **Question 27** En déduire un algorithme qui répond au problème en temps linéaire en la taille du graphe. On donnera l'algorithme en pseudo-code, en justifiant précisément sa correction et sa complexité, et l'on pourra supposer pour simplifier que les poids sont distincts.
Indication : on utilisera le résultat de la partie précédente sur le calcul de la médiane en temps linéaire.

▶ **Question 28** Cet algorithme peut-il être facilement adapté au cas où le graphe est orienté en remplaçant les composantes connexes par les composantes fortement connexes ?

5.2 Implémentation

On rappelle ici quelques fonctions utiles du module `Hashtbl`.

- `('a, 'b) Hashtbl.t` est le type d'un dictionnaire ayant des clés de type `'a` et des valeurs de type `'b` ;
- `Hashtbl.length : ('a, 'b) Hashtbl.t -> int`, qui donne le nombre de clés présentes dans la table.
- `Hashtbl.create : int -> ('a, 'b) Hashtbl.t`
`Hashtbl.create 1` renvoie une table de hachage vide (l'entier donné n'a pas d'importance).
- `Hahstbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b option`
`Hahstbl.find_opt h x` renvoie `Some y` si `y` est la valeur associée à la clé `x` dans `h`, ou `None` si la clé `x` n'est pas présente dans `h`.
- `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit`
`Hashtbl.replace h x y` associe la clé `x` à la valeur `y` dans la table `h`, en remplaçant l'ancienne association pour `x` s'il y en avait une (et en créant l'association dans le cas contraire).
- `Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit`
`Hashtbl.iter f h` équivaut à `f x1 y1; ... ; f xn yn; ()`, où $(x_1, y_1), \dots, (x_n, y_n)$ sont les associations présentes dans la table `h`, dans un ordre non spécifié.

On supposera que `Hashtbl.iter` s'exécute en temps proportionnel à la somme des complexités des appels à `f` effectués, et que les autres fonctions listées ci-dessus sont en temps constant.

On représente un graphe pondéré en OCaml en utilisant le type suivant :

```
type pondere = (int, (int * float) list) Hashtbl.t
```

Les sommets du graphe sont les clés de la table, et la valeur associée à x est la liste des couple (y, w) où y est un voisin de x et $w = f(xy)$.

On définit également un autre type pour représenter un graphe sous forme d'un ensemble d'arêtes :

```
type ensemble_aretes = ((int * int), float) Hashtbl.t
```

Dans ce type, les clés sont les couples (x, y) telles que xy soit une arête du graphe (chaque arête est donc présente deux fois, le graphe n'étant pas orienté). La valeur associée au couple (x, y) est $f(x, y)$. Les sommets du graphe sont définis implicitement comme étant les entiers x apparaissant dans une clé.

Remarque
Dans les deux cas, rien n'oblige les sommets du graphe à être numérotés consécutivement à partir de zéro.

▸ **Question 29** Écrire une fonction prenant un objet de type `ensemble_aretes` et renvoyant l'objet de type `pondere` correspondant. On demande une complexité linéaire en le nombre d'arêtes.

```
val construit_graphe : ensemble_aretes -> pondere
```

▸ **Question 30** Écrire une fonction `calcule_composantes` prenant en entrée un graphe $G = (V, E, f)$ à n sommets et un poids w et renvoyant un table de hachage `t` telle que :

- les clés de `t` sont exactement les sommets de G ;
- les valeurs associées sont des entiers compris entre 0 et $q - 1$, où q est le nombre de composantes connexes de $G(w)$;
- pour $u, v \in [0 \dots n-1]$, les valeurs associées à u et à v sont égales si et seulement si u et v sont dans la même composante connexe de $G(w)$.

On exige une complexité linéaire en la taille du graphe.

```
val calcule_composantes : pondere -> float -> (int, int) Hashtbl.t
```

▸ **Question 31** Écrire une fonction `calcule_gbar` prenant en entrée un graphe G, un flottant w et un dictionnaire `t` tel que renvoyé par l'appel `calcule_composantes g w` et renvoyant le graphe $\overline{G}(w)$. La numérotation des sommets de ce graphe sera celle donnée par le dictionnaire `t` (autrement dit, la composante du sommet u portera le numéro `Hashtbl.find t u` dans le graphe renvoyé).
On exige une complexité linéaire en la taille du graphe.

```
val calcule_gbar : pondere -> (int, int) Hashtbl.t -> float -> pondere
```

▸ **Question 32** Écrire une fonction `extrait_composante` prenant en entrée un graphe G, un flottant w, un dictionnaire `t` tel que renvoyé par l'appel `calcule_composantes g w`, un numéro `i` de composante (qui sera donc égal à l'une des clés de `t`) et renvoyant le graphe $G_i(w)$. La numérotation des sommets dans ce graphe sera la même que dans le graphe fourni en argument. On exige une complexité linéaire en la taille du graphe.

```
val extrait_composante : pondere -> (int, int) Hashtbl.t -> int -> float -> pondere
```

▸ **Question 33** Écrire une fonction `poids_median` prenant en entrée un graphe G et renvoyant la médiane du poids de ses arêtes. On exige une complexité linéaire en la taille du graphe.

```
val poids_median : pondere -> float
```

▸ **Question 34** Écrire finalement une fonction `capacite_max` prenant en entrée un graphe $G = (V, E, f)$ et deux sommets $u, v \in V$ et renvoyant la capacité maximale d'un chemin reliant u et v dans G. Le graphe G sera supposé connexe, et l'on exige une complexité linéaire en la taille de G.

```
val capacite_max : pondere -> int -> int -> float
```