Composition d'informatique n°4

Sujet unique (Durée : 4 heures)

L'utilisation de la calculatrice **n'est pas autorisée** pour cette épreuve.

⋆ ⋆ ⋆

Arbres de Steiner

Comment relier quatre villes en minimisant la longueur totale du réseau, quitte à s'autoriser des jonctions intermédiaires ? Les arbres de Steiner sont des objets de la théorie des graphes qui répondent à cette question et trouvent des applications dans la construction de réseaux de télécommunications, routiers, ... C'est également un phénomène qui se forme naturellement, par exemple lors de la minimisation de la surface de contact entre plusieurs bulles de savon, ou dans les recherches de liens de parenté entre groupes d'êtres-vivants (arbres phylogénétiques). La recherche d'un arbre de Steiner est un problème d'optimisation difficile et nous étudierons comment la recherche d'arbres couvrants peut fournir des approximations raisonnables.

Le problème est découpé en quatre parties. La première partie concerne des propriétés générales sur les arbres couvrants et les forêts. La deuxième partie montre que le problème de l'arbre de Steiner est un problème difficile à résoudre. La troisième partie présente un algorithme probabiliste permettant de calculer un arbre couvrant de poids minimum en temps linéaire, puis, la quatrième et dernière partie établit que malgré sa difficulté, il est possible d'approximer le problème de l'arbre de Steiner en un temps raisonnable en utilisant des arbres couvrants.

Consignes

Les questions de programmation doivent être traitées en langage OCaml uniquement. 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.

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`).

Préliminaires

Si $G = (S, A)$ est un graphe non orienté, on dit que $H = (X, B)$ est un **sous-graphe** de G si $X \subseteq S$ et $B \subseteq A \cap \mathcal{P}_2(X)$. Pour $X \subseteq S$, le **sous-graphe de G induit par** X est le graphe $G[X] = (X, A \cap \mathcal{P}_2(X))$. Un **sous-arbre** de G est un sous-graphe de G qui est un arbre.

On appelle **composante connexe** de G une partie $X \subseteq S$ telle que $G[X]$ est connexe, et pour tout $s \in S \setminus X$, $G[X \cup \{s\}]$ n'est pas connexe. S'il n'y a pas d'ambiguïté, on utilisera le terme composante connexe pour désigner aussi bien la partie X que le sous-graphe $G[X]$.

Un graphe non orienté est appelé **forêt** s'il ne contient pas de cycle. On remarque qu'un graphe est une forêt si et seulement si chacune de ses composantes connexes est un arbre. Une **sous-forêt** de G est un sous-graphe de G qui est une forêt. Un graphe $F = (X, B)$ est une **forêt couvrante de** G si c'est une sous-forêt de G telle que $X = S$ et possédant autant de composantes connexes que G.

Si $G = (S, A, f)$ est un graphe non orienté pondéré, avec $f : A \to \mathbb{R}_+$, on appelle **poids de** G la somme des poids des arêtes, c'est-à-dire $f(G) = \sum_{a \in A} f(a)$. La **distance** entre $s \in S$ et $t \in S$, notée $d(s, t)$, est le poids

minimal d'un chemin de s à t. S'il n'y a pas d'ambiguïté, lorsqu'on considère un sous-graphe $H = (X, B)$ de (S, A), on considère que les pondérations des arêtes de B sont données par $f_{|B}$.

1 Arbres et forêts

Question 1 Représenter graphiquement, sans justifier, une forêt couvrante de poids minimal du graphe G_0 représenté figure 1.

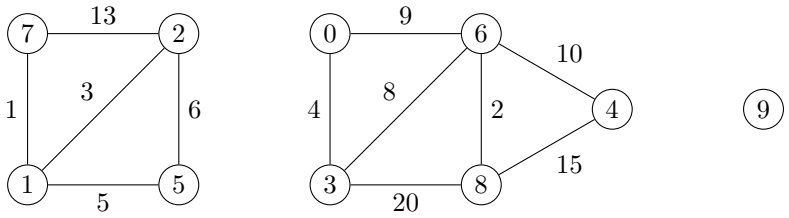

FIGURE 1 – Le graphe pondéré G_0.

Question 2 Rappeler le principe de l'algorithme de Kruskal et sa complexité temporelle lorsqu'il est appliqué à un graphe non orienté pondéré $G = (S, A, f)$.

Question 3 On admet que l'algorithme de Kruskal renvoie un arbre couvrant de poids minimal lorsqu'il est appliqué à un graphe G non orienté pondéré connexe. Montrer qu'il renvoie une forêt couvrante de poids minimal de G si on supprime l'hypothèse de connexité.

Pour les deux questions suivantes, on suppose $G = (S, A, f)$ un graphe pondéré non orienté tel que f est injective. On admet que dans ce cas, G possède une unique forêt couvrante de poids minimal, qu'on note $F^* = (S, B^*)$.

Question 4 Montrer la **propriété de cycle** : pour tout cycle C dans G, l'arête de poids maximal de C n'est pas dans B^*.

Question 5 Montrer la **propriété de coupe** : pour toute partie non vide X telle que $X \subsetneq S$, l'arête de poids minimal de A parmi celles ayant exactement une extrémité dans x, si elle existe, est dans B^*.

2 Problème de l'arbre de Steiner

On définit les problèmes de décision suivants :

- `Couverture des arêtes par les sommets (CPS)` :
  - **Instance** : un graphe non orienté $G = (S, A)$ et $k \in \mathbb{N}$.
  - **Question** : existe-t-il une couverture des arêtes de G par les sommets de cardinal $\leqslant k$, c'est-à-dire un ensemble $X \subseteq S$ tel que $|X| \leqslant k$ et $\forall a \in A$, $a \cap X \neq \emptyset$?
- `Arbre de Steiner (ADS)` :
  - **Instance** : un graphe non orienté connexe $G = (S, A)$, $X \subseteq S$ et $k \in \mathbb{N}$.
  - **Question** : existe-t-il un arbre de Steiner de G ayant X comme sommets terminaux de poids inférieur ou égal à k, c'est-à-dire un sous-arbre $T = (Y, B)$ de G tel que $X \subseteq Y \subseteq S$ et $|B| \leqslant k$?

On admet que le problème `CPS` est NP-complet.

Question 6 Montrer que ADS $\in$ NP.

Pour un graphe non orienté $G = (S, A)$, on appelle **graphe d'incidence** de G le graphe $G_I = (S_I, A_I)$ tel que $S_I = S \cup A$ et $A_I = \mathcal{P}_2(S) \cup \{\{s, a\} \mid s \in S, a \in A, s \in a\}$.

Question 7 Représenter graphiquement le graphe d'incidence du graphe G_1 représenté en figure 2.

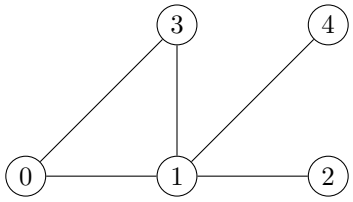

FIGURE 2 – Le graphe non orienté G_1.

Question 8 Montrer que pour un graphe non orienté $G = (S, A)$, G possède une couverture par sommets de taille k si et seulement si G_I possède un arbre de Steiner à $|A| + k - 1$ arêtes ayant A comme sommets terminaux.

Question 9 En déduire la NP-complétude de ADS.

3 Calcul linéaire d'une forêt couvrante minimale

Dans cette partie, on étudie un algorithme probabiliste dont l'objectif est de renvoyer une forêt couvrante de poids minimal d'un graphe non orienté. Il s'agit d'un algorithme récursif qui procèdera en fusionnant des sommets et en ne conservant qu'une partie des arêtes entre les nouveaux sommets fusionnés.

Dans l'ensemble de cette partie, on suppose que les fonctions de pondérations sont injectives.

3.1 Phase de Borůvka

Pour $G = (S, A, f)$ un graphe non orienté pondéré et $F = (S, B)$ une sous-forêt de G ayant $C_0, \ldots, C_{m-1}$ comme composantes connexes, on définit la **contraction de G selon F**, notée $C_F(G) = (S', A', f')$, de la manière suivante :

- S' est l'ensemble des composantes connexes de F qui ne sont pas des sommets isolés dans G ;
- A' est l'ensemble des arêtes de G entre les composantes connexes de F, c'est-à-dire :

$$A' = \{\{C_i, C_j\} \mid \exists s \in C_i, \exists t \in C_j, \{s, t\} \in A\}$$

- f' correspond aux poids minimaux des arêtes entre les composantes, c'est-à-dire que pour $\{C_i, C_j\} \in A'$:

$$f'(C_i, C_j) = \min\{f(s, t) \mid s \in C_i, t \in C_j\}$$

Par exemple, pour $B_0 = \{\{1, 7\}, \{2, 5\}, \{0, 6\}, \{3, 8\}\}$, la contraction du graphe $G_0 = (S_0, A_0, f_0)$ selon la forêt $F_0 = (S_0, B_0)$ est représentée figure 3.

Question 10 On pose $B = \{\{1, 2\}, \{1, 5\}, \{0, 3\}, \{6, 8\}\}$. Représenter graphiquement, sans justification, la contraction du graphe G_0 de la figure 1 selon la forêt $F = (S_0, B)$.

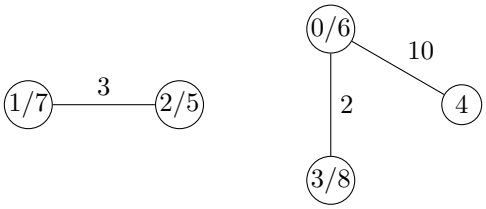

FIGURE 3 – La contraction de G_0 selon F_0.

On représente un graphe pondéré en OCaml par un objet de type :

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

Ainsi, si `g` est un objet de type `graphe` représentant $G = (S, A, f)$, alors $S = [\![0, n-1]\!]$ où n est la longueur du tableau `g`, et si $s \in S$, alors `g.(s)` est une liste contenant, dans un ordre quelconque, tous les couples `(t, p)` tels que $\{s, t\} \in A$ et $f(s, t) = p$.

La figure 4 représente une implémentation du graphe G_0 représenté figure 1.

```
let g0 = [| [(3, 4); (6, 9)]; [(2, 3); (5, 5); (7, 1)];
            [(1, 3); (5, 6); (7, 13)]; [(0, 4); (6, 8); (8, 20)];
            [(6, 10); (8, 15)]; [(1, 5); (2, 6)];
            [(0, 9); (3, 8); (4, 10); (8, 2)]; [(1, 1); (2, 13)];
            [(3, 20); (4, 15); (6, 2)]; [] |];;
```

FIGURE 4 – Une implémentation du graphe G_0.

Lors d'une contraction, les arêtes auront des extrémités dont les numéros vont être modifiés. Pour garder trace des modifications, on propose de travailler avec un type de **graphe avec mémoire**. Pour ce faire, chaque arête gardera en mémoire l'arête d'origine dont elle est issue. L'implémentation est la suivante :

```
type voisin = {nom : int; poids : int; origine : int * int};;

type graphe_mem = voisin list array;;
```

Ainsi, si `g_mem` est un graphe avec mémoire représentant un graphe $G = (S, A, f)$, alors $S = [\![0, n-1]\!]$ où n est la longueur du tableau `g_mem`, et si $s \in S$, alors `g_mem.(s)` est une liste contenant, dans un ordre quelconque, les voisins `v` de type `voisin` tels que $a = \{s, \texttt{v.nom}\} \in A$, $f(a) =$ `v.poids` et `v.origine` est le couple désignant l'arête du graphe initial dont est issue l'arête a.

La figure 5 correspond à une implémentation de $C_{F_0}(G_0)$ représentée figure 3. On a renommé les sommets pour avoir une numérotation commençant à 0 et consécutive. Notons qu'ici, chaque sommet n'a qu'un seul voisin, mais les listes d'adjacence peuvent être plus grandes.

```
let h0 = [| [{nom = 3; poids = 2; origine = (6, 8)};
             {nom = 4; poids = 10; origine = (4, 6)}];
            [{nom = 2; poids = 3; origine = (1, 2)}];
            [{nom = 1; poids = 3; origine = (1, 2)}];
            [{nom = 0; poids = 2; origine = (6, 8)}];
            [{nom = 0; poids = 10; origine = {4, 6}] |];;
```

FIGURE 5 – Une implémentation de $C_{F_0}(G_0)$.

Question 11 Écrire une fonction `convertir (g: graphe) : graphe_mem` qui convertit un graphe sans mémoire en un graphe avec mémoire, en supposant que chaque arête est sa propre arête d'origine.

Pour $G = (S, A, f)$ un graphe pondéré non orienté et $s \in S$ un sommet non isolé, on note $v_{\min}(s)$ le voisin de s qui minimise les poids des arêtes des voisins de s, c'est-à-dire tel que $f(s, v_{\min}(s)) = \min\{f(s,t) \mid \{s,t\} \in A\}$.

Lors du calcul d'une forêt couvrante de poids minimal, une étape sera répétée plusieurs fois : la **phase de Borůvka**. Elle se décompose ainsi, étant donné un graphe $G = (S, A, f)$ pondéré non orienté :

- poser $B = \{\{s, v_{\min}(s)\} \mid s \in S\}$;
- poser $F = (S, B)$;
- renvoyer $C_F(G)$.

Question 12 On pose $F^* = (S, B^*)$ l'unique forêt couvrante de poids minimal de G. Montrer qu'avec les notations précédentes, $B \subseteq B^*$.

Question 13 Montrer qu'avec les notations précédentes, le nombre de sommets de $C_F(G)$ vaut au plus $\frac{|S|}{2}$.

Question 14 Écrire une fonction `voisin_min (lst: voisin list) : voisin` qui prend en argument une liste de voisins et renvoie celui qui minimise le poids. La fonction renverra un voisin sentinelle dont le nom vaut -1 si la liste est vide.

Question 15 En déduire une fonction `foret_boruvka (g: graphe_mem) (fcm: graphe) : graphe` qui prend en argument un graphe $G = (S, A, f)$ avec mémoire et un arbre `fcm` sans mémoire et :

- calcule $B = \{\{s, v_{\min}(s)\} \mid s \in S\}$ et modifie `fcm` en lui ajoutant toutes les arêtes d'origine des arêtes de B ;
- calcule et renvoie $F = (S, B)$ sous forme d'un graphe sans mémoire.

On rappelle les opérations suivantes sur les tables de hachage, présentes dans le module `Hashtbl` :

- `create : int -> ('a, 'b) Hashtbl.t` crée une table dont la capacité initiale est donnée en argument. En pratique, on pourra mettre 1 comme argument ;
- `mem : ('a, 'b) Hashtbl.t -> 'a -> bool` prend en argument une table et une clé et renvoie un booléen qui vaut `true` si et seulement s'il existe une association avec cette clé ;
- `find : ('a, 'b) Hashtbl.t -> 'a -> 'b` prend en argument une table et une clé et renvoie la valeur associée à cette clé si elle existe, ou lève l'exception `Not_found` sinon ;
- `replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` prend en argument une table, une clé et une valeur et **ajoute** une association (clé, valeur) s'il n'existe pas d'association avec la clé, ou **remplace** l'ancienne association sinon ;
- `iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit` prend en argument une fonction et une table, et applique la fonction à chaque association (clé, valeur) de la table, dans un ordre non spécifié.

On admet que les quatre premières fonctions ont une complexité constante en moyenne, et la dernière a une complexité linéaire en le nombre d'associations si la fonction donnée en argument a une complexité constante.

Question 16 Écrire une fonction `elim_doublons (lst: voisin list) : voisin list` qui prend en argument une liste de voisins pouvant contenir des noms en doublons, dans un ordre quelconque, et renvoie une liste sans doublon, en ayant gardé, pour chaque nom, le voisin ayant le poids minimal. La fonction devra avoir une complexité linéaire en la taille de la liste en moyenne.

La figure 6 correspond à un exemple d'appel à la fonction `elim_doublons`.

Question 17 Écrire une fonction `composantes (g: graphe_mem) (f: graphe) : int array * int` qui prend en argument un graphe $G = (S, A, f)$ avec mémoire et une sous-forêt $F = (S, B)$ de G et renvoie un couple `(comp, m)` tel que m est le nombre de composantes connexes de F de cardinal $\geqslant 2$, et pour $s \in S$, `comp.(s)` vaut -1 si s est un sommet isolé dans G, ou le numéro de la composante connexe de s dans F sinon, la numérotation choisie étant consécutive en partant de 0.

```
# elim_doublons [{nom = 18; poids = 13; origine = (24, 35)};
                 {nom = 9; poids = 8; origine = (24, 26)};
                 {nom = 39; poids = 21; origine = (2, 24)};
                 {nom = 9; poids = 15; origine = (11, 26)};
                 {nom = 39; poids = 2; origine = (11, 19)}];;
- : voisin list = [{nom = 18; poids = 13; origine = (24, 35)};
                   {nom = 9; poids = 8; origine = (24, 26)};
                   {nom = 39; poids = 2; origine = (11, 19)}]
```

FIGURE 6 – Un exemple d'appel à `elim_doublons`.

Question 18 En déduire une fonction `phase_boruvka (g: graphe_mem) (fcm: graphe) : graphe_mem` qui calcule une phase de Borůvka pour un graphe avec mémoire donné en argument, tout en mettant ajoutant à `fcm` les arêtes calculées à la question 15.

Question 19 Quelle est la complexité temporelle moyenne de la fonction précédente ?

3.2 Sélection aléatoire d'arêtes

Pour $G = (S, A, f)$ un graphe pondéré non orienté, on appelle **demi-graphe de G** un graphe $\frac{1}{2}G = (S, A')$ obtenu en conservant chaque arête de A avec une probabilité $\frac{1}{2}$.

On rappelle que `Random.int k` renvoie un entier choisi aléatoirement et uniformément entre 0 et $k-1$.

Question 20 Écrire une fonction `demi_graphe (g: graphe_mem) : graphe` qui prend en argument un graphe avec mémoire G et renvoie sous forme de graphe sans mémoire un graphe $\frac{1}{2}G$.

Soit $G = (S, A, f)$ un graphe pondéré non orienté et $F = (S, B)$ une sous-forêt de G. Une arête $a \in A$ est dite **F-lourde** si $(S, B \cup \{a\})$ contient un cycle dont a est l'arête de poids maximal. Elle est dite **F-légère** sinon.

Question 21 Montrer que si F est une sous-forêt de G et $F^* = (S, B^*)$ est une forêt couvrante de poids minimal de G, alors B^* ne contient aucune arête F-lourde.

On considère les deux algorithmes suivants :

Algorithme : version 1
Entrée : Graphe $G = (S, A, f)$ pondéré non orienté
Début algorithme
  Calculer $H = \frac{1}{2}G$.
  Calculer $F = (S, B)$ une forêt couvrante de poids minimal de H.
  Renvoyer F.

Algorithme : version 2

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

1 **Début algorithme**
2 Poser $B = \emptyset$.
3 **Pour** chaque arête $a \in A$ par poids croissant **Faire**
4 **Si** $(S, B \cup \{a\})$ ne contient pas de cycle **Alors**
5 Tirer à pile ou face.
6 **Si** le résultat est pile **Alors**
7 $B \leftarrow B \cup \{a\}$
8 **Renvoyer** $F = (S, B)$.

Question 22 Montrer que les graphes renvoyés par les deux versions d'algorithmes suivent la même distribution de probabilité.

Question 23 Montrer que si F est un arbre renvoyé par l'algorithme version 2, pour $a \in A$, a ne vérifie pas la condition du **Si** à la ligne 4 si et seulement si a est une arête F-lourde.

Une variable aléatoire suit une **loi binomiale négative de paramètres** k si elle correspond au nombre de fois où on a obtenu un face lors d'une suite de tirs à pile ou face, jusqu'à obtenir k fois un pile. On admet que si Y est une variable aléatoire suivant une loi binomiale négative de paramètre k, alors $\mathbb{E}(Y) = 2k$ (pour obtenir k piles, il faut tirer en moyenne $2k$ pièces).

Question 24 Montrer que si F est un arbre renvoyé par l'algorithme version 1, alors le nombre d'arêtes F-légères de $G = (S, A)$ est en moyenne inférieur à $2|S|$.

3.3 Calcul de forêt couvrante minimale

L'algorithme de calcul de forêt couvrante minimale d'un graphe $G = (S, A, f)$ est le suivant :

1. si $|S| = 1$, renvoyer G ;
2. appliquer deux phases de Borůvka successivement à G, **en gardant en mémoire** les arêtes conservées dans les sous-arbres construits. Noter $G' = (S', A', f')$ le graphe obtenu ;
3. calculer $H = \frac{1}{2}G'$;
4. calculer **récursivement** $F' = (S', B')$ une forêt couvrante de poids minimal de H **sans garder en mémoire** les arêtes conservées ;
5. supprimer toutes les arêtes F'-lourdes de G'. Noter $G'' = (S', A'', f')$ le graphe obtenu ;
6. calculer **récursivement** une forêt couvrante de poids minimal de G'' **en gardant en mémoire** les arêtes conservées
7. renvoyer (S, B^*) où B^* est formé des arêtes conservées.

On admet que l'étape 5 peut se faire en temps linéaire en $|S'| + |A'|$. Elle prendra la forme d'une fonction `alleger (g': graphe_mem) (f': graphe) : graphe_mem`.

Question 25 Écrire deux fonctions mutuellement récursives qui suivent l'algorithme décrit ci-dessus :

- `calcul_fcm (g: graphe) : graphe` calcule une forêt couvrante de poids minimal d'un graphe sans mémoire selon l'algorithme précédent ;
- `maj_fcm (g: graphe_mem) (fcm: graphe) : unit` modifie une forêt sans mémoire `fcm` en cours de construction selon les arêtes d'origine d'une forêt couvrante minimale d'un graphe `g` avec mémoire.

Question 26 Montrer que l'algorithme décrit précédemment renvoie une forêt couvrante de poids minimal lorsqu'il est appliqué à un graphe pondéré non orienté.

Pour $G = (S, A, f)$, on considère un arbre binaire dont les nœuds correspondent aux graphes sur lesquels des appels récursifs à l'algorithme précédent sont effectués dans un calcul avec G comme argument, l'enfant gauche d'un nœud étant le graphe H de l'étape 4, l'enfant droit étant le graphe G'' de l'étape 6.

Pour l'ensemble de l'analyse de complexité, on admet que le calcul de `elim_doublons` peut se faire en temps linéaire en la taille de la liste, dans tous les cas (et pas seulement en moyenne). Par ailleurs, quitte à les supprimer, on pourra supposer que tous les graphes considérés n'ont pas de sommets isolés.

Question 27 Montrer que la complexité temporelle totale d'un appel à l'algorithme sur un graphe $G = (S, A, f)$ est égale à la la somme du nombre de sommets et d'arêtes de tous les graphes dans l'arbre binaire des appels.

Question 28 Montrer que l'arbre des appels est de hauteur $\mathcal{O}(\log |S|)$. En déduire que la complexité dans le pire cas de l'algorithme est $\mathcal{O}((|S| + |A|) \log |S|)$.

On appelle **branche gauche** une suite de graphes $(G_1, G_2, \ldots, G_k)$ telle que :

- G_1 est la racine de l'arbre des appels ou l'enfant droit de son parent ;
- pour $i \in [\![1, k-1]\!]$, G_{i+1} est l'enfant gauche de G_i ;
- G_k n'a pas d'enfants.

Question 29 Montrer qu'en moyenne, le nombre total d'arêtes d'une branche gauche $(G_1, G_2, \ldots, G_k)$ vaut au plus $2|A_1|$, où $G_1 = (S_1, A_1)$.

Question 30 En déduire qu'en moyenne, un appel à l'algorithme sur un graphe $G = (S, A, f)$ s'effectue en temps $\mathcal{O}(|S| + |A|)$. Quelle ville-casino décrit ce type d'algorithme ?

4 Approximation

On considère dans cette partie le problème d'optimisation `Arbre de Steiner pondéré (ADSP)` :

* **Instance** : un graphe pondéré non orienté connexe $G = (S, A, f)$, et $X \subseteq S$.
* **Solution** : un arbre de Steiner de G ayant X comme sommets terminaux, c'est-à-dire un sous-arbre $T = (Y, B)$ de G tel que $X \subseteq Y \subseteq S$.
* **Optimisation** : minimiser le poids de T.

Question 31 Justifier que si on sait résoudre le problème d'optimisation de l'arbre de Steiner pondéré en temps polynomial, alors on sait résoudre le problème de décision de l'arbre de Steiner non pondéré en temps polynomial.

Question 32 Résoudre le problème `Arbre de Steiner pondéré` pour le graphe G_2 représenté figure 7 et $X = \{0, 4, 5\}$. On attend uniquement une représentation graphique du résultat sans justification.

Pour $G = (S, A, f)$ un graphe pondéré non orienté connexe, f à valeurs positives, et $X \subseteq S$, on considère l'algorithme suivant :

- construire $H = (X, \mathcal{P}_2(X), d)$ un graphe pondéré complet, où d représente la distance dans G ;
- construire $T_1 = (X, B_1)$ un arbre couvrant de poids minimal de H ;
- poser Y l'ensemble des sommets qui apparaissent dans un plus court chemin de s à t dans G, pour chaque arête $\{s, t\} \in B_1$;
- construire et renvoyer $T = (Y, B)$ un arbre couvrant de poids minimal de $G[Y]$.

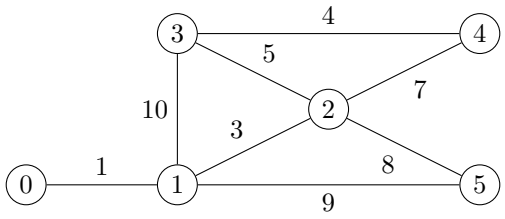

FIGURE 7 – Le graphe pondéré G_2.

Question 33 Montrer que l'algorithme précédent renvoie un arbre de Steiner ayant X comme sommets terminaux (pas nécessairement de poids minimal).

Question 34 Déterminer la complexité temporelle de l'algorithme précédent en fonction de $|S|$ et $|A|$. On détaillera succinctement les complexités des différentes étapes.

Question 35 Montrer que l'algorithme précédent est une 2-approximation de `ADSP`.

⋆ ⋆ ⋆