Théorie des Graphes
Graphes arboricoles et forestiers

TPS 1A INOC

theoleyre@unistra.fr

Objectif : qu'est ce qu'un arbre et quels sont les différents algorithmes permettant de les calculer et exploiter ?

Durée : 2 heures

Exercice 1: Arbres et forêts couvrants

Soit $G = (S, A)$ un graphe non orienté, avec S l'ensemble de ses sommets et A celui de ses arêtes.

1. Rappeler une condition nécessaire portant sur le nombre de ses arêtes pour que G soit un arbre.
2. Montrer que si G est connexe il admet au moins un arbre couvrant, et que cet arbre couvrant est unique si et seulement si G est un arbre.
3. Montrer que si G est un arbre, il admet 2^{n-1} forêts couvrantes, où n est le nombre de sommets de G.

1. Soit G connexe et $|S| = n$. Alors G est un arbre seulement si $|A| \leq n - 1$.
2. Soit G connexe, on a alors $|S| \geq n$. On pose $G_0 = G$ et $A_0 = A$ On peut construire un arbre couvrant à partir de G de la façon suivante : soit $a_0 \in A_0$ faisant partie d'un cycle de G_0, construire le graphe $G_1 = (S, A_0 \setminus \{a\})$. Alors par construction, G_1 est connexe. On réitère cette opération jusqu'à ce que G_m ait exactement $n-1$ arêtes, et on obtient un arbre couvrant.

   $\Leftarrow$: Supposons que G soit un arbre, et que $G_1 = (S, A_1)$ et $G_2 = (S, A_2)$ soient deux arbres couvrants de G. Alors $\exists a_1 \in A_1$ et $a_1 \notin A_2$. Alors $A_2 \cup \{a_1\} \subseteq A$, et $|A| \geq n$, ce qui est absurde car G est un arbre.

   Alternativement, soit $a_1 = (u, v) \in A_1$. Soit c le chemin de u à v dans A_2. Alors $c \oplus (u, v)$ [a] crée bien un cycle dans le graphe d'origine, ce qui aboutit à une contradiction.

   $\Rightarrow$: Supposons que G admette un unique arbre couvrant. Supposons que G admette un cycle $s_0, s_1, s_2, \ldots, s_{m-1}$. On peut appliquer l'algorithme précédent en commençant par supprimer l'arête $a_1 = (s_0, s_1)$ pour obtenir un arbre couvrant $G_1 = (S, A_1)$ avec $a_1 \notin A_1$. On peut aussi l'appliquer en commençant par supprimer l'arête $a_2 = (s_1, s_2)$. On aura alors construit un arbre couvrant $G_2 = (S, A_2)$ avec $a_1 \in A_2$, et G_1 est différent de G_2, ce qui est absurde.

   Autrement dit, on prend le cycle de G on peut créer 2 arbres différents (la première vs. la dernière arête sont enlevées). Comme par hypothèse, il n'y a qu'un seul graphe recouvrant, il n'y a donc pas de cycle $\leftrightarrow$ G est donc un arbre (sans cycle).
3. Si G est un arbre, il est sans cycle. Une forêt couvrante est un graphe partiel de G (i.e. tout graphe partiel de G est par définition une forêt), et il y a une bijection entre les graphes partiels de G et l'ensemble des sous-ensembles de A, qui sont au nombre de 2^{n-1}, si G a n sommets.

   On peut alternativement raisonner par récurrence : toutes les forêts qui contiennent a et celles qui ne le contiennent pas.

a. avec $\oplus$ l'opérateur concaténant des chemins

Exercice 2: Implémentation

On propose les structures de données suivantes pour représenter un arbre par la liste de ses sommets, chaque sommet contenant un pointeur vers son père qui définit l'arborescence. La racine de l'arbre pointe vers `NULL`.

```
//un sommet est défini par son id et son père
typedef struct sommet {
   int id;
   struct sommet * pere;
}sommet;

//un arbre est défini par sa racine et la liste de ses sommets (id / père)
struct arbre {
```

```
   sommet * racine;
   struct liste arborescence;
}

//un maillon de la liste chainée
struct maillon {
   sommet * elem;
   struct maillon * suivant;
};

// la liste commence par le maillon de tête
struct liste {
   struct maillon * tete;
};

//initialise une liste FIFO
void liste_init(struct liste *l);

//retourne vrai ssi la liste est vide
bool liste_estVide(struct liste *l);

//ajoute un élément en queue de la liste
void liste_ajout_queue(struct liste *l, sommet *elem);

//retourne l'élément en tête de liste (sans l'enlever)
sommet * liste_tete(struct liste *l);

//enlève un élément en tête de liste
sommet * liste_enleve_tete(struct liste *l);
```

Écrire le corps de la fonction de prototype :

```
void traitement_Profondeur (graphe G, sommet depuis, void (*traitement) (sommet *) );
```

qui applique à chaque sommet de l'arbre enraciné en `depuis` (par exemple la racine) la fonction `traitement`, avec une stratégie de parcours en profondeur dans un premier temps. On ne demande pas d'implémenter la fonction `traitement` qui pourrait être, **par exemple**, la fonction d'affichage suivante.

```
void affiche_sommet (sommet * s) {
        printf("sommet %i, ",s->id);
}
```

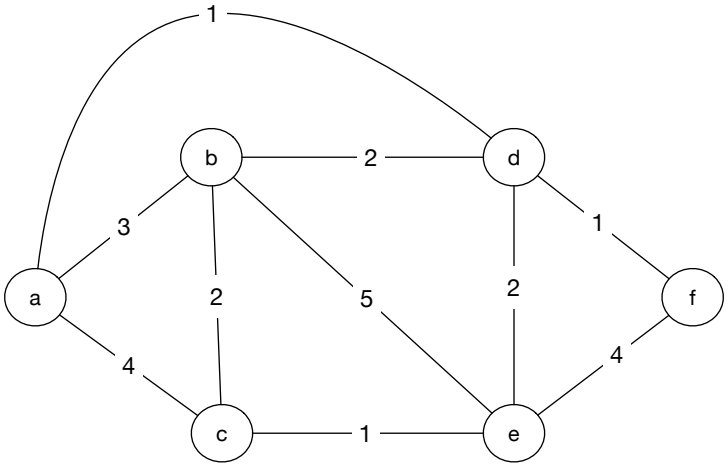
```
void traitement_Profondeur(struct arbre *ar, sommet *depuis, void (*traitement) (sommet *) ){
  struct liste fils;
  liste_init(&fils);

  struct liste temp;
  liste_init(&temp);

  traitement(depuis);

  ///liste des fils
  while (!liste_estVide(ar->arborescence)) {
    // tant qu'il existe un sommet duquel je suis père,
    // je l'ajoute à la liste des sommets à traiter
    if ( ((liste_tete(ar->arborescence))->pere!=NULL)
       &&  ( (liste_tete(ar->arborescence))->pere->id == depuis->id ) )
            liste_ajout_queue( &fils, tete_liste(ar->arborescence) );

    // je stocke dans une liste temporaire le sommet que je viens de parcourir
    liste_ajout_queue( &temp, supp_tete_liste(&ar->arborescence) );
  }

  //j'ai traité tous les sommets : rajoutons les maintenant à l'arbre comme c'était
  while (!liste_estVide(temp))
     liste_ajout_queue( &ar->arborescence, supp_tete_liste(&temp) );

  // maintenant que j'ai la liste des sommets, je traite chacun d'eux récursivement
  while (!est_vide_liste(fils))
   traitement_Profondeur(ar, supp_tete_liste(&fils), traitement );

  return;
}
```

```
void traitement_Largeur (struct arbre * ar, sommet * depuis, void (*traitement) (sommet *) ){
   struct liste temp;
   liste_init(&temp);
   struct file traite;
   liste_init(&traite);
   sommet * courant = depuis;

   //je commence par depuis
   liste_ajoute_queue(&traite,  depuis);

   //tant que la liste n'est pas vide
   while (!est_vide_file(traite)) {
      courant = defile (&traite);
      traitement(courant);

      ///recherche des fils
      while (!liste_estVide(ar->arborescence)) {
         if ( ((liste_tete(ar->arborescence))->pere!=NULL) &&  ( (tete_liste(ar->arborescence))->pere->
            liste_ajoute_queue( &traite, tete_liste(ar->arborescence) );
   ajout_liste( &temp, supp_tete_liste(&ar->arborescence) );
       }
       while (!liste_estVide(temp))
           ajout_liste( &ar->arborescence, supp_tete_liste(&temp) );
  }
  return;

}
```

NB : Je remarque que les fils sont bien traités dans l'ordre : j'ordonne par niveau. Tous ceux à 1 arête de la racine vont être ajoutés dans la première étape. Puis tous ceux du niveau k seront ajoutés via un sommet de niveau k-1, tous traités ensemble auparavant.

Exercice 3: Arbres couvrants de poids minimum

On considère le graphe donné par la figure ci-après. Déterminer un arbre couvrant de poids minimal en utilisant successivement les algorithmes de Kruskal, puis de Prim.

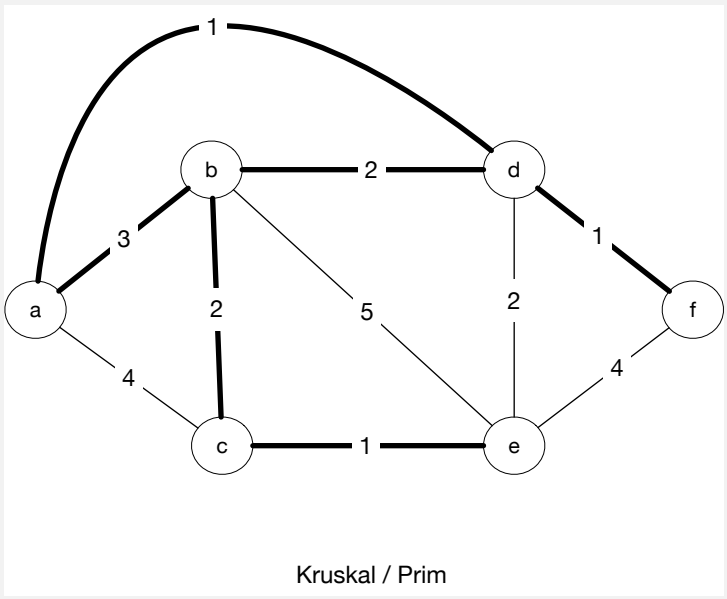


On part de a
Kruskal additif :

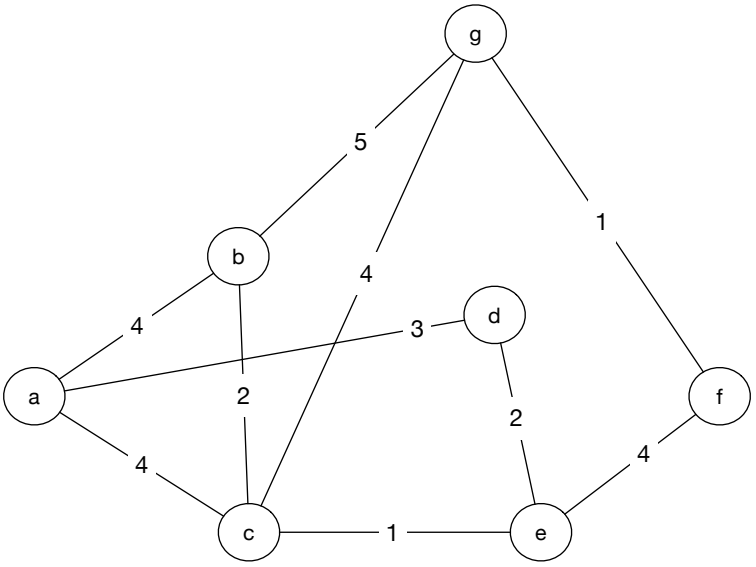
1. classement des arêtes : ad, df,ce,bc,de,ab,ef,ac,be
2. on garde une arête tant qu'elle ne crée pas un cycle
3. ad : ok
4. df : ok
5. ce : ok
6. bc : ok
7. de : non
8. etc.

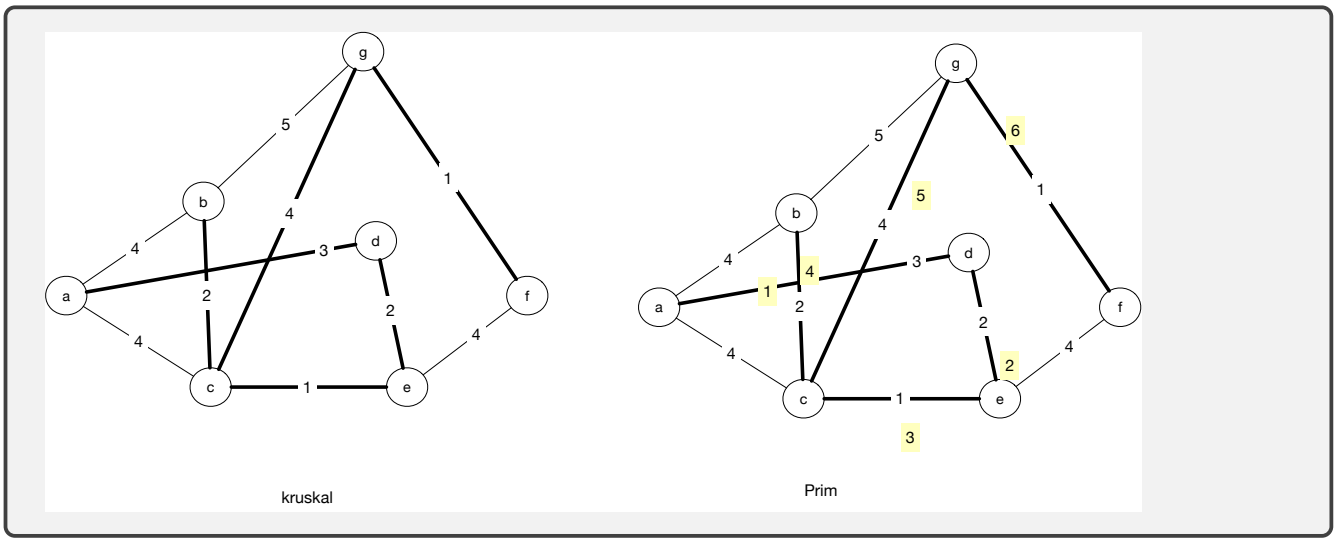
Prim :

1. On part du sommet a
2. on prend l'arc sortant de poids min qui ne crée pas de cycle : ad
3. puis df
4. puis db
5. etc.



Même question avec le graphe ci-après.





Exercice 4: Algorithme de Prim

Montrer que l'algorithme de Prim termine et est correct. Évaluer son coût dans le pire des cas en justifiant.

L'algorithme termine : à chaque itération, un sommet est traité. Ainsi, le nombre de sommets restant est strictement décroissant : l'algorithme termine.
Par ailleurs, par construction, l'algorithme de Prim crée un arbre. En effet, le résultat est connexe (ajout glouton d'un sommet). Par ailleurs n-1 arêtes ont été ajoutées après l'addition de n sommets (i.e. l'ensemble de départ est un sommet sans arête). Le résultat est donc un arbre (d'ailleurs, on vérifie l'absence de cycle à chaque ajout...
Montrons maintenant qu'il s'agit d'un des arbres recouvrant de poids maximum. Quelques notations s'imposent : soit T l'arbre fourni par l'algorithme de Prim et $e_1, e_2, ..., e_{n?1}$ la suite de ses arêtes, dans l'ordre où elles ont été ajoutées. Pour chaque ARPM T de G, soit $k(T)$ le plus grand entier tel que la suite d'arêtes $e_1, e_2, ..., e_{k(T)}$ soit dans T ($k(T) \geq 0$). Soit T' un ARPM de G tel quel l'entier $k = k(T')$ soit maximal pour les ARPM de G (c.a.d. T' est l'arbre de poids maximum qui a le plus d'arêtes en commun avec G). En d'autres termes, T' est l'arbre de poids max qui a le plus d'arêtes en communs avec T ($e_1, e_2, ..., e_{k(T)}$).
Soit V_k l'ensemble des sommets de G qui sont extrémités des arêtes $e_1, e_2, ..., e_k$ (V_0 ne contient que le sommet initial).
Utilisons une démonstration par l'absurde : on suppose que T n'est pas un ARPM. On a $k < n - 1$ (puisque T n'est pas un ARPM).

1.0 e_{k+1} est la première arête de T qui n'est pas dans T' :

1.1 $e_{k+1} = (s, t)$ a une extrémité (disons s) dans V_k, et l'autre (disons t) qui n'y est pas ;

1.2 Donc $T' \bigcup e_{k+1}$ est connexe, et contient un cycle C dont e_{k+1} est une arête ;

1.3 Il existe alors une autre arête e du cycle C reliant un sommet de V_k à un sommet qui n'est pas dans V_k ;

1.4 e_{k+1} a été choisi par l'algorithme de Prim, et le poids de e est donc plus grand (ou égal) à celui de e_{k+1} ;

1.5 Considérons alors l'arbre $T' \bigcup e_{k+1} \setminus e$: c'est un ARPM (car le poids de e est au moins égal à celui de e_{k+1}) qui coincide avec T sur $k + 1$ arêtes.

On aboutit à une contradiction sur notre hypothèse de départ (coincide sur seulement les k premières arêtes), et donc $k = n - 1$ et T est un ARPM.

Supposons que nous utilisions des tableaux pour stocker nos variables. L'initialisation est en O(|S|). Il y a par ailleurs |S|-1 étapes, pour chacune desquelles il faut trouver le minimum (en O(|S|)). Un test d'absence de cycle est nécessaire. Cependant, on peut maintenir l'ensemble des sommets déjà traités et interdire les arêtes sortantes n'ayant pas un sommet non traité comme extrémité. Ainsi, le test à chaque itération peut itérer sur chaque arête,en O(|A|) (un SI qui regarde le poids ET le marquage des extrémités en un seul test). La complexité est donc au final en $O(|A|.|S|)$.

Exercice 5: Graphe Biparti

Pouvez vous démontrer qu'un graphe est biparti si et seulement s'il ne contient aucun cycle de longueur impaire ?

Supposons qu'un graphe est décomposé en deux partitions V_1 et V_2 telles que toute arête relie un sommet de V_1 à un sommet de V_2. Considérons un cycle $c = \{u_0, ..., u_k\}$ quelconque. Les arêtes ne peuvent relier que deux sommets des deux partitions V_1 et V_2. Ainsi, si $(u_k, u_{k+1}) \in V_1, (u_{k+1}, u_{k+2}) \in V_2$ (ou inversement). Enfin, un cycle se termine en un même sommet. Nous avons donc autant d'arêtes $(u_k, u_{k+1}) \in V_1$ que d'arêtes $(u_k, u_{k+1}) \in V_2$. Le cycle est donc de longueur paire.

Démontrons maintenant la réciproque. Supposons que l'on ait un graphe G qui ne comporte que des cycles de longueur paire. Pour simplifier, supposons également que G est connexe (si jamais il ne l'était pas, on pourrait raisonner individuellement sur chaque composante connexe).
Comme G est connexe, il possède un arbre couvrant T. Notons r une racine de cet arbre. Créons deux ensembles de sommets U et V de la manière suivante :

1. V_1 contient r et tous les sommets de T se trouvant à une profondeur paire ;
2. V_2 contient les sommets de T se trouvant à une profondeur impaire.

Les ensembles de sommets V_1 et V_2 sont disjoints et forment une partition de G ($V_1 \cup V_2 = G$ et $V_1 \cap V_2 = \emptyset$). Il faut montrer maintenant qu'il n'existe aucune arête interne à V_1 et V_2. Raisonnons par l'absurde. Supposons qu'il existe deux sommets u et v de G dans V_1 (respectivement V_2) et reliés par une arête e. Le chemin qui relie u à v dans T est de longueur paire (somme de deux longueurs paires ou impaires) par construction de V_1 (respectivement V_2). Ainsi, nous pouvons créer un cycle de longueur impaire, en ajoutant l'arête e, ce qui aboutit à une contradiction.

Exercice 6: Largeur arborescente

1. Montrer qu'un arbre T admet une décomposition arborescente de largeur 1.

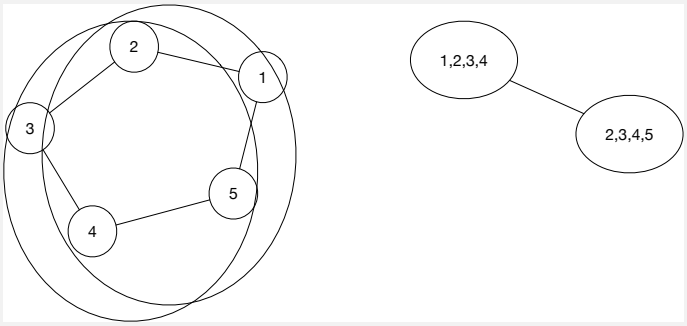
Rappel d'une décomposition arborescente : chaque arête doit être dans au moins un "sac", l'ensemble des sacs comportant un sommet doit former un ensemble connexe, et les sacs doivent former un arbre. La largeur arborescente est dans ce cas là la cardinalité du plus grand sac (nombre de sommets qu'il contient) minoré de 1.

Il *"suffit"* donc de construire une décomposition de telle sorte que chaque arête soit représentée par un (et un seul) "sac". Ainsi, chaque arête est bien trivialement dans au moins un sac.

Relions maintenant les sacs correspondant à des arêtes incidentes (qui ont un sommet commun). Trivialement, un sommet appartient donc à un ensemble connexe de "sacs".

Démontrons que les sacs forment bien un arbre, et démontrons le par l'absurde. Supposons que 2 sacs (u, v) et (x, y) aient deux chemins disjoints qui les rejoignent. Par construction, la suite de sacs correspond à des arêtes incidentes dans le graphe d'origine. Ainsi, nous avons deux chemin entre u et x. Nous pouvons donc créer un cycle, concaténation de ces deux chemins, ce qui aboutit bien à une contradiction.

2. Donner une décomposition arborescente du graphe C_5 [1] de largeur 3.



3. Cette décomposition est-elle la plus petite pour C_5 ? Quelle est la largeur de décomposition d'un graphe C_n ?

Sa largeur arborescente est 2 : il admet une décomposition arborescente composée des ensembles f_i, tels que :

- $f_1 = \{1, 2\}$
- $f_i = \{1, i, i+1\}$ pour $i > 2$

avec $V = \{i\}_{i\in[1,n]}$, et les arcs $E = \big(\cup_{i\in[1,m]}(i, i+1)\big) \cup (1, m)$
$f_i | i > 2$ forme le plus grand ensemble, comprenant 3 sommets.

1. anneau comportant 5 sommets.