Chapître ?? : arbres couvrants minimaux

KRUSKAL vs PRIM

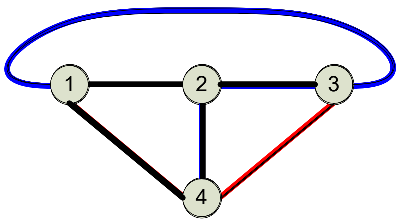
Support dispo sur Arche/L3/Section5

1. Définitions et applications

- Définition 1

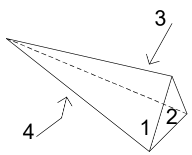
  Un *arbre couvrant* un graphe connexe G = (V,E) est un ensemble acyclique d'arêtes qui connecte tous les sommets de G.

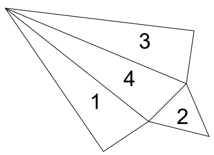
- Exemple :

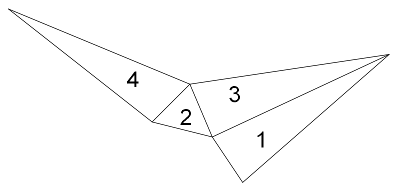


NON !!

- Application : patrons (dépliages) d'un polyèdre ; 1 dépliage = 1 arbre couvrant du graphe d'adjacence des faces





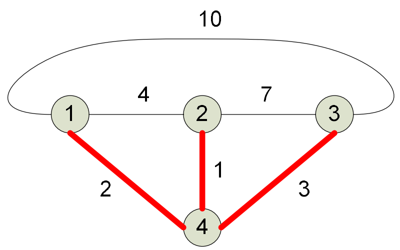


1. Définitions et applications

- Remarque : on parle d'*arbre* (couvrant) puisqu'un graphe connexe sans cycles est, par définition, un arbre

- Définition 2 :
  - Soit un graphe non orienté G = (V,E), pondéré par une fonction de E vers $\Re$
  - un Arbre Couvrant de Poids Minimum (ACPM) est tel que la somme des poids de ses arêtes est minimum.

- Exemple :



- Application :

  Imaginons que chaque nœud soit un ordinateur et que le poids de l'arête entre x et y indique la longueur de câble nécessaire pour relier les ordinateurs x et y.

  Alors, l'ACPM donne la plus courte longueur de câble nécessaire pour relier tous les ordinateurs.

1. Définitions et applications

- Un ACPM contient n-1 arêtes. En effet :
  - L'ACPM est un arbre qui contient tous les sommets du graphe, donc n sommets
  - Dans un arbre, tout sommet a un père, sauf la racine ➔ il y a n-1 arêtes.

- Le calcul d'un ACPM est un problème d'optimisation :
  - chaque arbre couvrant est une solution
  - Il faut trouver la meilleure solution (minimiser son poids)

- Le premier algorithme a été publié en 1926 par Boruvka

- Les deux algos les plus connus, Kruskal et Prim, sont des algorithmes gloutons.

2. Les algorithmes

- Conventions :
  - l'ACPM : T
  - l'ensemble de ses arêtes : A
  - le graphe construit à partir de A (donc les arêtes de A et leurs sommets) : G [A]

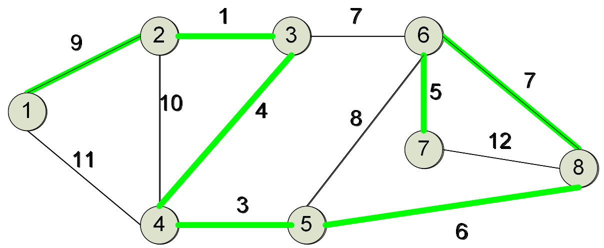
- Algorithme de Kruskal

  A = ∅

  Tant que | A | < n - 1

  On prend la première / prochaine arête par poids croissant

  On l'ajoute à A sauf si elle engendre un cycle dans G [A].



- Remarques :
  - pendant l'exécution, G [A] est une forêt car les arêtes de A peuvent former plusieurs composantes connexes (plusieurs arbres)
  - Il est pratique de trier préalablement les arêtes par poids croissants

2. Les algorithmes

- Algorithme de prim

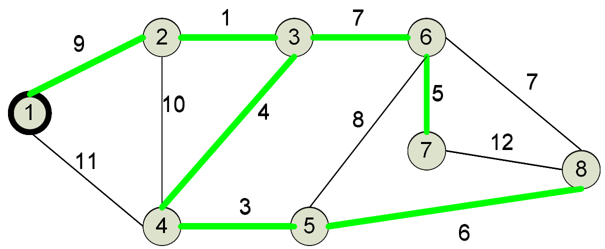
  A = ∅

  On choisit un sommet quelconque comme racine du futur ACPM

  Puis, n-1 fois

  on ajoute une arête de moindre poids parmi celles qui relient l'arbre à un sommet extérieur à l'arbre

- G [A] est en permanence un arbre unique (G [A] est connexe).



2. Les algorithmes

- Algorithme générique
  Les algos de Prim et Kruskal ont la même structure générique :

  ACPM-GENERIQUE (G,w)

  A = ∅
  tant que A ne forme pas un arbre couvrant
  Trouver une arête sûre {u,v} pour A
  Ajouter {u,v} à A (⇔ A = A ∪ { {u,v} })

  retourner A

Kruskal

On trie les arêtes par poids croissants
NumAr = 0 ; A = ∅

Tant que | A | < n - 1
NumAr ++
si G [A ∪ {NumAr ^è^ arête}] acyclique
A = A ∪ {NumAr ^è^ arête}

retourner A

Prim

On choisit un sommet quelconque comme racine du futur ACPM

A = ∅

Puis, n-1 fois

on ajoute une arête de moindre poids parmi celles qui relient l'arbre à un sommet extérieur à l'arbre

retourner A

3. Arête sûre

- Définition :

  Une arête {u,v} est une arête sûre pour A si :

  1) A est un sous-ensemble d'un ACPM
  2) A $\cup$ { {u,v} } reste un sous-ensemble d'un ACPM

  C'est-à-dire qu'en ajoutant à A une arête sûre pour A, on ne fait pas d'erreur : il existe une extension de A qui soit un ACPM.

  ➔ ACPM-GENERIQUE respecte l'invariant « A est un sous-ensemble d'un ACPM ».

  ➔ ACPM-GENERIQUE trouve toujours un ACPM

  ➔ Si Prim et Kruskal choisissent à chaque fois une arête sure, ils trouvent bien un ACPM.

3. Arête sûre

- Comment trouver des arêtes sures ?

- Définition

  Une **coupure** (S, V-S) d'un graphe G = (V,E) est une partition des sommets en deux ensembles S et V-S.

- Définition :

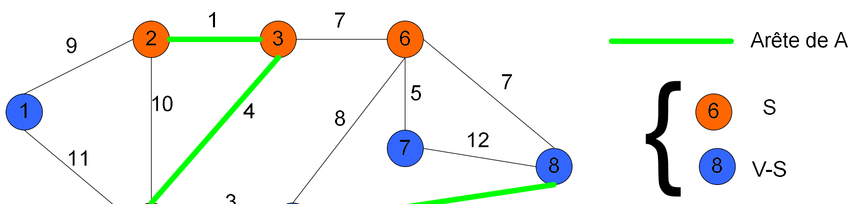
  Soit G = (V,E) un graphe non orienté

  Soit A $\subseteq$ E un ensemble d'arêtes

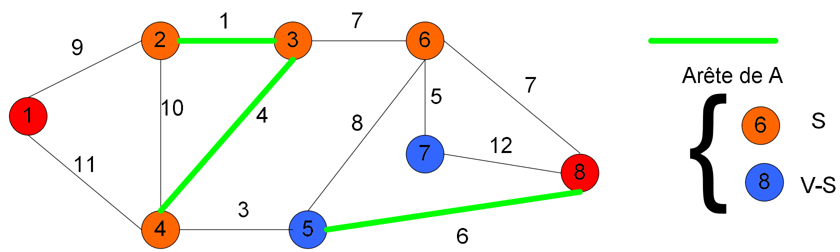
  Soit (S, V-S) une coupure de G

  On dit que la coupure (S, V-S) **respecte** A ssi aucune arête de A ne relie un sommet de S et un sommet de V-S

- Exemple



La coupure respecte A



La coupure ne respecte pas A

3. Arête sûre

- Comment trouver des arêtes sures ? Avec le théorème suivant (à admettre)

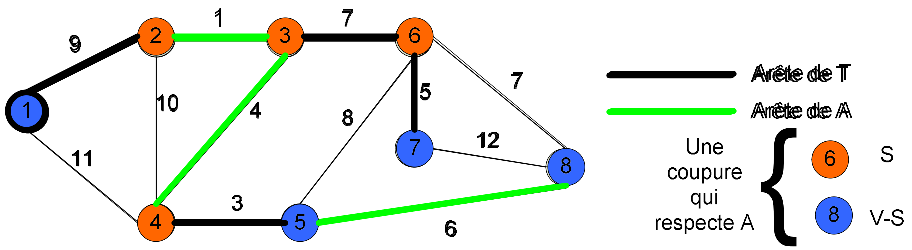
  Théorème :

  Soit A un ensemble d'arêtes ayant le potentiel pour devenir un ACPM
  ($\Leftrightarrow A \subseteq T$ où T est un ACPM)

  Soit (S, V-S) une coupure qui respecte A
  Soit {u,v} une arête de poids minimal parmi toutes les arêtes qui traversent la coupure

  Alors {u,v} est une arête sûre pour A.

- Exemple



- Avec cette coupure (S = {2,3,4,6}, V-S = {1,5,7,8})
  - Les arêtes {1,2}, {1,4}, {4,5}, {5,6}, {6,7}, {6,8} traversent la coupure
  - {4,5} est une arête sûre

3. Arête sûre

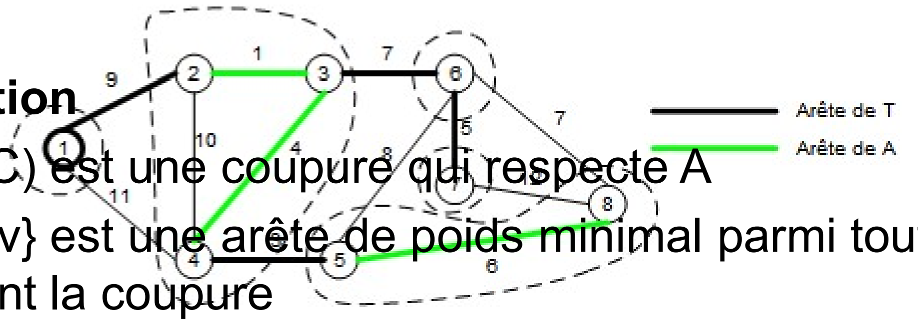
- **Corollaire :**

  Soit A un ensemble d'arêtes ayant le potentiel pour devenir un ACPM
  ($\Leftrightarrow A \subseteq T$ où T est un ACPM)

  Soit G_A le graphe (V,A)
  Soit C une composante connexe de G_A

  Soit {u,v} une arête de poids minimal parmi toutes les arêtes qui relient C à une autre composante connexe de G_A.

  Alors {u,v} est une arête sûre pour A.



- **Démonstration**

  (C, V – C) est une coupure qui respecte A

  donc {u,v} est une arête de poids minimal parmi toutes celles qui traversent la coupure

  donc (théorème) {u,v} est une arête sûre pour A.

4. Preuves de Prim

Prim

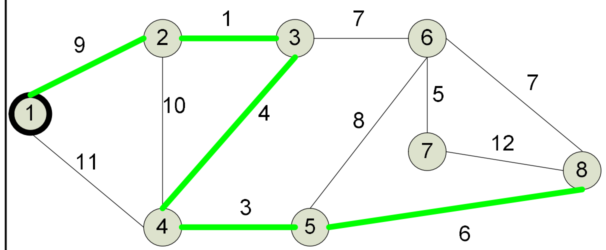
A = ∅

On choisit un sommet quelconque comme racine du futur ACPM

Puis, n-1 fois

on ajoute une arête de moindre poids parmi celles qui relient l'arbre à un sommet extérieur à l'arbre

retourner A



- Au moment d'ajouter une arête :
  - A forme une composante connexe du graphe G_A = (V,A) (puisque chaque arête ajoutée à A « est en contact par une extrémité » avec A)
  - donc toute arête qui relie A à un sommet extérieur à A relie la composante connexe de A à une autre
  - donc, parmi ces arêtes, celle qui est de moindre poids est sûre (corollaire)

donc l'arête ajoutée à A est sûre
donc Prim est valide.

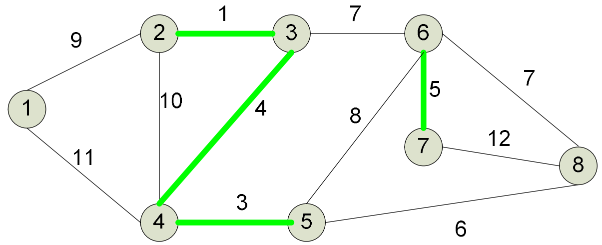
4. Preuves de Kruskal

Kruskal

On trie les arêtes par poids croissants
NumAr = 0 ; A = ∅

Tant que | A | < n - 1
NumAr ++
si G [A ∪ {NumAr è arête}] est acyclique
A = A ∪ {NumAr è arête}

retourner A



- Au moment d’ajouter une arête
  - A forme plusieurs composantes connexes et chaque sommet isolé aussi.
  - Si l’arête choisie reliait deux sommets de la même composante, elle créerait un cycle mais c’est interdit.
  - Donc l’arête relie deux composantes différentes.
  - De plus, les arêtes sont traitées par poids croissant donc l’arête choisie est une arête de poids minimum parmi les arêtes qui relient deux composantes connexes
- Donc c’est une arête sûre
- Donc Kruskal est valide.

4. Preuves de Kruskal et Prim

- Remarque :
  - bien que les deux algorithmes soient différents, ils donnent le même ACPM si ce dernier est unique.
  - on remarque que dans Kruskal, comme dans beaucoup d’algos gloutons, un tri préalable est nécessaire.
  - si toutes les arêtes ont des poids différents, il n’y a qu’un seul ACPM

5. Rendre Prim efficace

Prim

A = ∅

On choisit un sommet quelconque comme racine du futur ACPM

Puis, n-1 fois

on ajoute une arête de moindre poids parmi celles qui relient l'arbre à un sommet extérieur à l'arbre

retourner A

- Pour implémenter efficacement l'algorithme de Prim, l'important est de faciliter la sélection de la nouvelle arête à ajouter dans l'arbre constitué des arêtes de A.

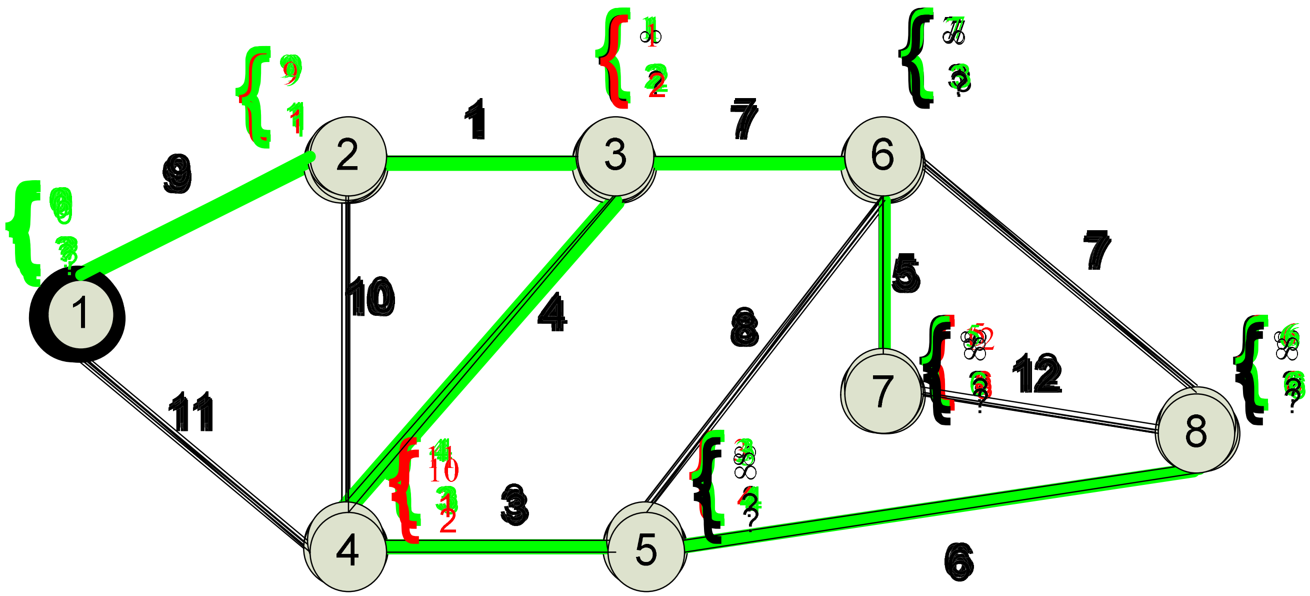
➔une structure de données (SdD) qui contient tous les sommets qui ne sont pas encore dans l'arbre

➔chaque sommet v est muni de deux informations :

  - clé (v) qui représente le poids de la plus légère arête qui relie v à un sommet de A. S'il n'y a pas de telle arête, clé (v) = +∞ (**Clé est un nom malheureux**)
  - π (v), qui représente le sommet d'où part cette arête

5. Rendre Prim efficace

- Exemple



5. Rendre Prim efficace

- La SdD S qui stocke les sommets doit offrir les services suivants :
  - Ajouter un élément x : INSERER (x, S)
  - Retirer l'élément de plus petite clé :
    x = EXTRAIRE-MIN (S)
  - Diminuer à c la clé d'un élément x : DIMINUER-CLE (S, x, c)

➔ c'est une file de priorité

5. Rendre Prim efficace

Avec une file de priorité, voici l'aspect de Prim

```
Pour tout sommet v
        Clé (v) = ∞                       // Clé (v) = distance de V à l'ACPM

Clé (r) = 0, π (r) = NIL                  // r = racine choisie

F = ∅                                     // On ajoute tous les sommets dans la file de priorité
Pour tout sommet v
        INSERER (v, F)

ACPM = ∅
Tant que F ≠ ∅
        u = EXTRAIRE-MIN (F)
        si u ≠ r alors ACPM = ACPM U { (π (u), u) } Fsi          // ACPM += moindre arête sortante

        // Mise à jour éventuelle de Clé et π sur les voisins de u
        Pour tout voisin v de u
                si v ∈ F et w ({u,v}) < Clé (v)
                        π (v) = u
                        Clé (v) = w ({u,v})
                        DIMINUER-CLE (F, v, w ({u,v})
```

Attention :

- « si v ∈ F » ne doit pas impliquer de parcours de la file (marquage)
- « DIMINUER-CLE (F, v, w ({u,v}) » non plus

5. Rendre Prim efficace

- Si la file de priorité est réalisée avec un **tas binômial**, Prim est en :

  $O\ (m.\log\ (n))$

- Si la file de priorité est réalisée avec un **tas de Fibonacci**, Prim est en :

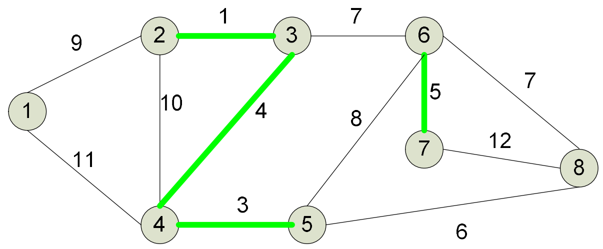
  $O\ (m + n.\log\ (n))$

5. Rendre Kruskal efficace

```
Kruskal
    On trie les arêtes par poids croissants
    NumAr = 0 ; A = ∅

    Tant que | A | < n - 1
        NumAr ++
        si G [A ∪ {NumAr è arête}] acyclique
            A = A ∪ {NumAr è arête}

    retourner A
```



- Il faut savoir rapidement si l'arête courante crée un cycle, autrement dit, si ses deux extrémités sont dans la même composante connexe de G_A = (V,A).
- Il faut aussi pouvoir réunir efficacement deux composantes connexes

- Pour cela, on utilise une structure de données d'**ensembles disjoints** (UNION-FIND structure)

5. Rendre Kruskal efficace

- La structure de données UNION-FIND permet de gérer plusieurs ensembles *disjoints,* chacun étant représenté par un de ses éléments.

- Elle offre les services suivants :
  - CRÉER-ENSEMBLE (x) : création d’un singleton contenant x. Le représentant de cet ensemble est x. x ne doit pas appartenir déjà à un ensemble.

  - UNION (x,y) : réunion des ensembles de x et y. Le représentant du nouvel ensemble est l’un quelconque de ses éléments.

  - TROUVER-ENSEMBLE (x) : restitue le représentant de l’ensemble qui contient x

5. Rendre Kruskal efficace

Algo de Kruskal :

```
A = ∅
Pour chaque sommet v de V
    CRÉER-ENSEMBLE (v)

Trier les arêtes de E par pondérations croissantes

Pour chaque arête {u,v} de E par ordre croissant de pondération
    si TROUVER-ENSEMBLE (u) ≠ TROUVER-ENSEMBLE (v)
            A = A ∪ {{u,v}}
            UNION (u,v)

Retourner A
```

5. Rendre Kruskal efficace

- Avec une bonne réalisation de la structure UNION-FIND, Kruskal s'exécute en :

$$O\ (m + n.\log\ (n))$$

(comme Prim avec un tas de Fibonacci)