Algorithmique et Complexité
Cours 3/7 : Arbre couvrant de poids minimal

CentraleSupélec – Gif

ST2 – Gif

Plan

Plan

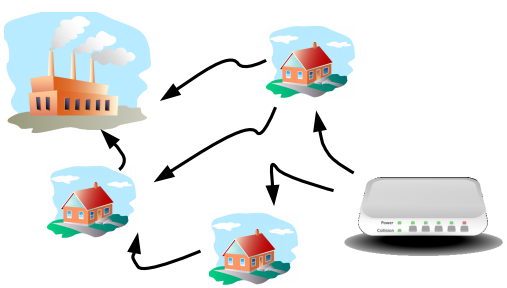
Fournisseur d'accès d'Internet

Problème

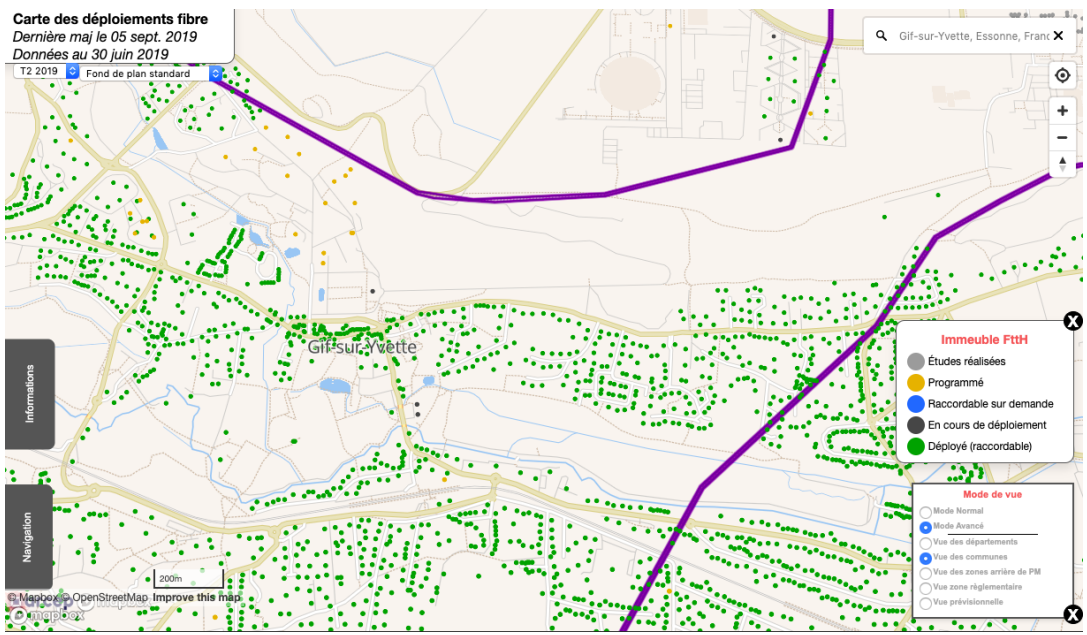
Connecter n sites par une fibre optique.

$\rightarrow$ On connaît le coût des travaux pour poser un tronçon de fibre entre une paire de sites v_i et v_j, $i, j = 1, 2, \ldots, n$

(il n'est pas toujours possible de poser la fibre entre deux sites).



Réseau fibre Gif - source : ARCEP



Fournisseur d'accès d'Internet

Problème

Connecter n sites par une fibre optique.

$\rightarrow$ On connaît le coût des travaux pour poser un tronçon de fibre entre une paire de sites v_i et v_j, $i, j = 1, 2, \ldots, n$

(il n'est pas toujours possible de poser la fibre entre deux sites).

Objectif

Mettre en place une nouvelle infrastructure optique en dépensant le moins d'argent possible.

Fournisseur d'accès d'Internet

Problème

Connecter n sites par une fibre optique.

$\rightarrow$ On connaît le coût des travaux pour poser un tronçon de fibre entre une paire de sites v_i et v_j, $i, j = 1, 2, \ldots, n$

(il n'est pas toujours possible de poser la fibre entre deux sites).

Objectif

Mettre en place une nouvelle infrastructure optique en dépensant le moins d'argent possible.

Nature du problème

C'est un problème d'optimisation.

Modélisation du problème

Réseau → graphe

- Sites → sommets d'un graphe non orienté
- Tronçons de fibre possibles → arêtes entre deux sommets
- Coût des travaux → poids de l'arête

Modélisation du problème

Réseau → graphe

- Sites → sommets d'un graphe non orienté
- Tronçons de fibre possibles → arêtes entre deux sommets
- Coût des travaux → poids de l'arête

Instance du problème

Un graphe connexe $G = (V, E)$ à n sommets, non-orienté, pondéré à poids strictement positifs

On note ω la fonction de pondération $\omega : E \rightarrow]0, +\infty[$

Modélisation du problème

Introduction de la notion de sous-graphe

Soit $G = (V, E)$ un graphe. Un sous-graphe de G est un couple (V', E') tel que :

- $V' \subseteq V$ (pas d'autres sommets que ceux de G)
- $E' \subseteq E$ (pas d'autres arêtes que celles de G)
- $E' \subseteq V' \times V'$ (arêtes entre éléments de V')

Modélisation du problème

Introduction de la notion de sous-graphe

Soit $G = (V, E)$ un graphe. Un sous-graphe de G est un couple (V', E') tel que :

- $V' \subseteq V$ (pas d'autres sommets que ceux de G)
- $E' \subseteq E$ (pas d'autres arêtes que celles de G)
- $E' \subseteq V' \times V'$ (arêtes entre éléments de V')

Retour au problème : que veut-on faire ?

Trouver un sous-graphe $T = (V', E')$ de G :

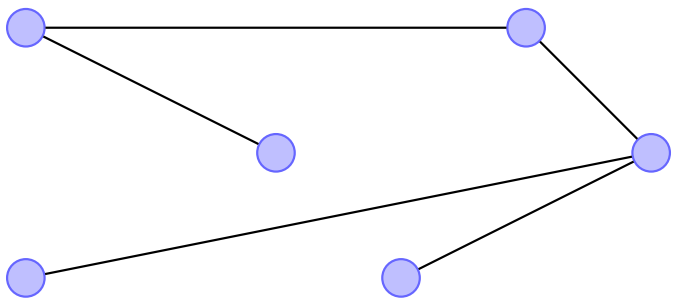
- Comportant tous les sommets de G, $V = V'$ (mais pas nécessairement toutes les arêtes)
- Connexe
- La somme des poids des arêtes ($\sum_{a \in E'} \omega(a)$) est minimale.

Quelques définitions

Arbre (rappel)

Un graphe non-orienté connexe et acyclique est un arbre.

Exemple :



Forêt

Une forêt est un ensemble fini d'arbres.

Propriété

Théorème

T, le sous graphe de G recherché est un arbre.

Propriété

Théorème

T, le sous graphe de G recherché est un arbre.

Démonstration.

Graphe non-orienté, connexe et acyclique ?

T est un graphe non-orienté et connexe (par définition)

□

Propriété

Théorème

T, le sous graphe de G recherché est un arbre.

Démonstration.

Graphe non-orienté, connexe et acyclique ?

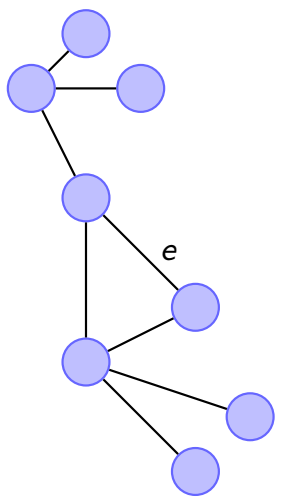
T est un graphe non-orienté et connexe (par définition)

Acyclique : preuve par l'absurde

Supposons que T ait un cycle : on retire une arête e du cycle (n'importe laquelle). Notons T_1 le sous-graphe de T ne contenant pas e.

$\sum_{a \in E_{T_1}} \omega(a) = \sum_{a \in E_T} \omega(a) - \omega(e)$

Donc T_1 a un coût plus petit que T (minimal).



□

Arbre couvrant de poids minimal

En anglais : Minimal Spanning Tree (MST)

Données

- $G = (V, E)$, $|V| = n$, non-orienté, connexe
- Fonction de pondération $\omega : E \longrightarrow \mathbb{R}_+^*$

Objectif

Trouver un arbre $T = (V, E_T), E_T \subseteq E$ tel que T est un arbre couvrant de G et $\sum_{e \in E_T} \omega(e)$ est minimale.

Arbre couvrant de poids minimal

En anglais : Minimal Spanning Tree (MST)

Données

- $G = (V, E)$, $|V| = n$, non-orienté, connexe
- Fonction de pondération $\omega : E \longrightarrow \mathbb{R}_+^*$

Objectif

Trouver un arbre $T = (V, E_T), E_T \subseteq E$ tel que T est un arbre couvrant de G et $\sum_{e \in E_T} \omega(e)$ est minimale.

Approche exhaustive (*brute-force*) : On énumère tous les T possibles et on compare leurs poids.

Complexité : $\mathcal{O}(\mathcal{C}_{|E|}^{|V|-1})$. On va éviter.

Plan

Algorithmes gloutons (greedy)

Définition

Un algorithme glouton est un algorithme qui :

- Construit une solution « pas à pas » ;

  ex : construction Lego ou multiplication en ligne

- Effectue à chaque étape un choix qui optimise un critère local ;

  c.-à-d. une évaluation de la situation courante

- Ne remet pas en cause les choix précédents.

Algorithmes gloutons (greedy)

Définition

Un algorithme glouton est un algorithme qui :

- Construit une solution « pas à pas » ;

  ex : construction Lego ou multiplication en ligne

- Effectue à chaque étape un choix qui optimise un critère local ;

  c.-à-d. une évaluation de la situation courante

- Ne remet pas en cause les choix précédents.

- On choisit localement la meilleure solution : aucune garantie d'atteindre l'optimum global.
- On ne revient pas en arrière : on va directement vers une solution.
- Exemple ?

Algorithmes gloutons (greedy)

Définition

Un algorithme glouton est un algorithme qui :

- Construit une solution « pas à pas » ;

  ex : construction Lego ou multiplication en ligne

- Effectue à chaque étape un choix qui optimise un critère local ;

  c.-à-d. une évaluation de la situation courante

- Ne remet pas en cause les choix précédents.

- On choisit localement la meilleure solution : aucune garantie d'atteindre l'optimum global.
- On ne revient pas en arrière : on va directement vers une solution.
- Exemple : Plus court chemin
  - critère local : noeud de la frontière de distance minimale

Arbre couvrant de poids minimum : approche générique

Objectif

Étant donné $G = (V, E)$ et $w : E \longrightarrow \mathbb{R}_+^*$.

➜ Il faut construire pas à pas un sous ensemble d'arêtes du graphe.

Principe général

- Au début, $E_T = \emptyset$
- À chaque itération, on choisit une nouvelle arête (u, v) telle que $E_T \cup (u, v)$ est toujours un sous-ensemble d'un arbre couvrant de poids minimal pour G.

➜ On dit que (u, v) est une arête sûre pour E_T

Cet algorithme glouton atteint donc un optimum global...

Arbre couvrant de poids minimum : approche générique

Algorithme général

```
def MST(V,E):
  E_T = []
  while !isSolution(E_T,V,E):
    e = safeEdge(E_T,V,E)
    E_T.append(e)
  return E_T
```

Arbre couvrant de poids minimum : approche générique

Algorithme général

```
def MST(V,E):
    E_T = []
    while !isSolution(E_T,V,E):
        e = safeEdge(E_T,V,E)
        E_T.append(e)
    return E_T
```

Comment choisir une arête sûre ?

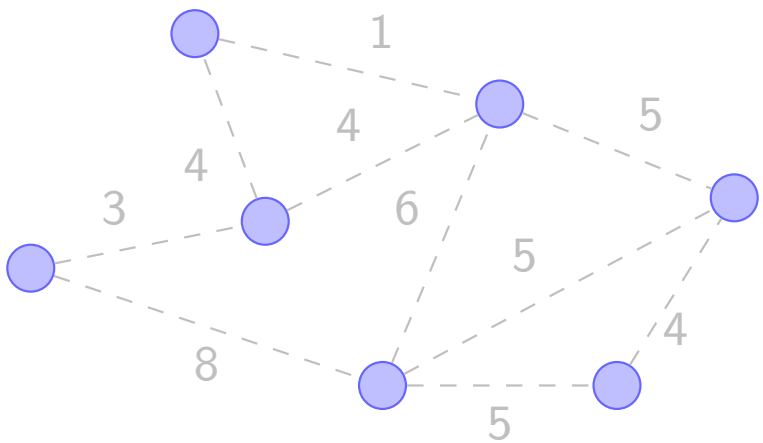
Nous verrons bientôt comment...

Commençons par découvrir deux algorithmes qui suivent ce schéma.

Approche de Kruskal (1956)

Aperçu

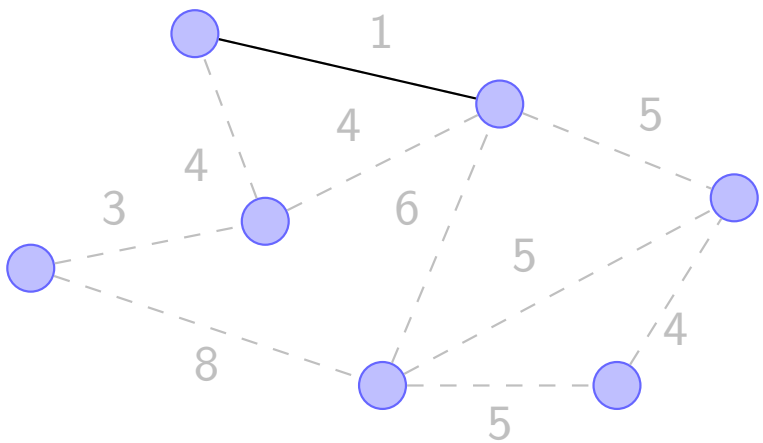
- **Initialisation :** une forêt T comportant tous les sommets de G et aucune arête
- **Itération :** ajouts à l'ensemble T d'une arête de poids minimal sans créer de cycle
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Kruskal (1956)

Aperçu

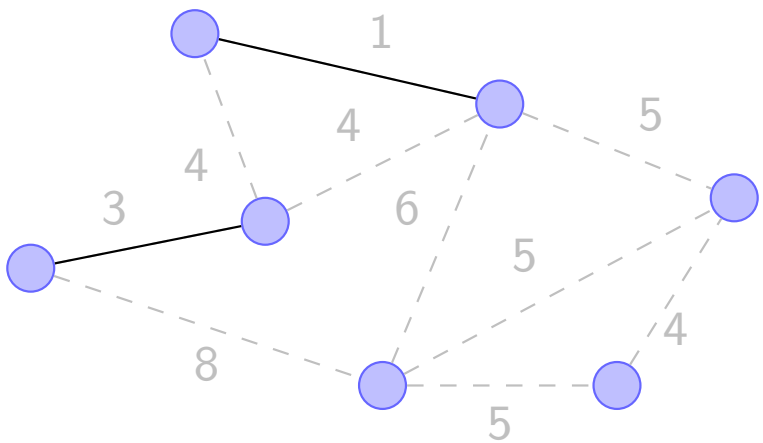
- **Initialisation :** une forêt T comportant tous les sommets de G et aucune arête
- **Itération :** ajouts à l'ensemble T d'une arête de poids minimal sans créer de cycle
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Kruskal (1956)

Aperçu

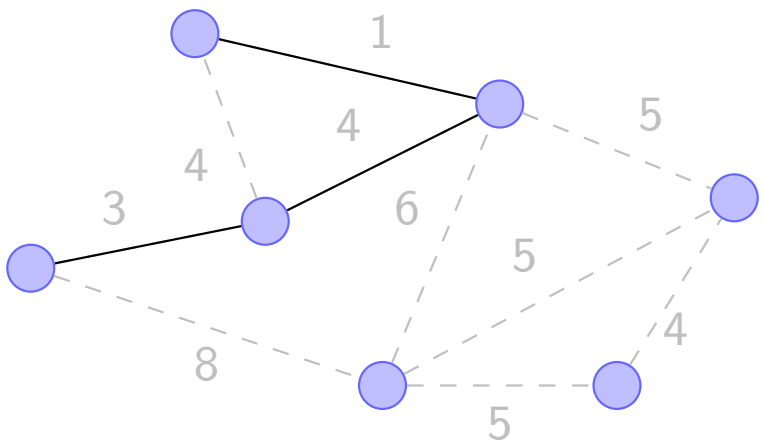
- **Initialisation :** une forêt T comportant tous les sommets de G et aucune arête
- **Itération :** ajouts à l'ensemble T d'une arête de poids minimal sans créer de cycle
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Kruskal (1956)

Aperçu

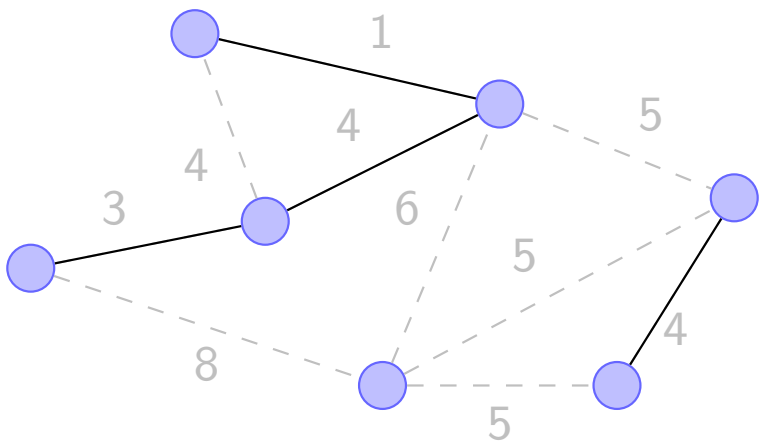
- **Initialisation :** une forêt T comportant tous les sommets de G et aucune arête
- **Itération :** ajouts à l'ensemble T d'une arête de poids minimal sans créer de cycle
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Kruskal (1956)

Aperçu

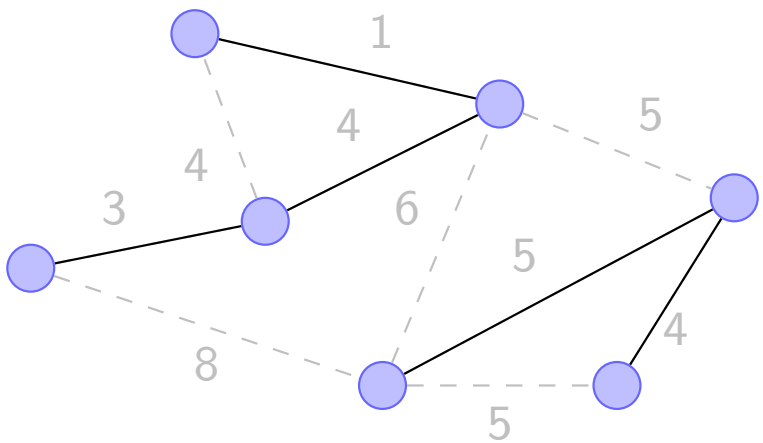
- **Initialisation :** une forêt T comportant tous les sommets de G et aucune arête
- **Itération :** ajouts à l'ensemble T d'une arête de poids minimal sans créer de cycle
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Kruskal (1956)

Aperçu

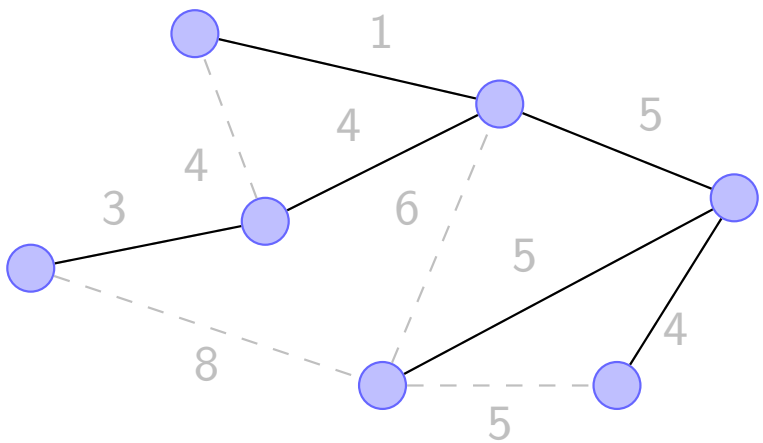
- **Initialisation :** une forêt T comportant tous les sommets de G et aucune arête
- **Itération :** ajouts à l'ensemble T d'une arête de poids minimal sans créer de cycle
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Kruskal (1956)

Aperçu

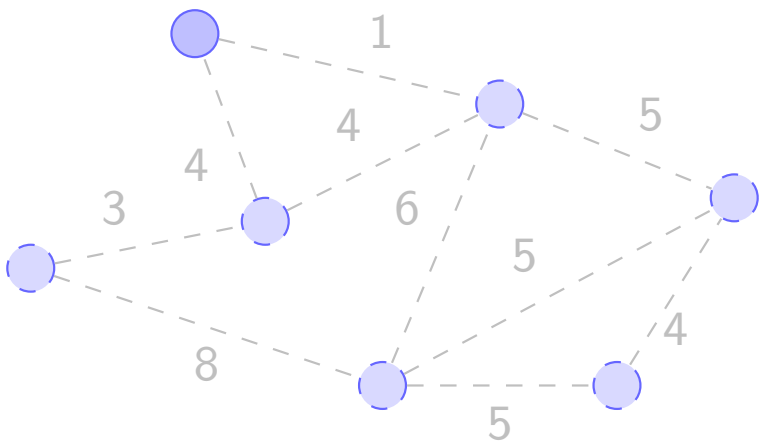
- **Initialisation :** une forêt T comportant tous les sommets de G et aucune arête
- **Itération :** ajouts à l'ensemble T d'une arête de poids minimal sans créer de cycle
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Prim (1957)

Aperçu

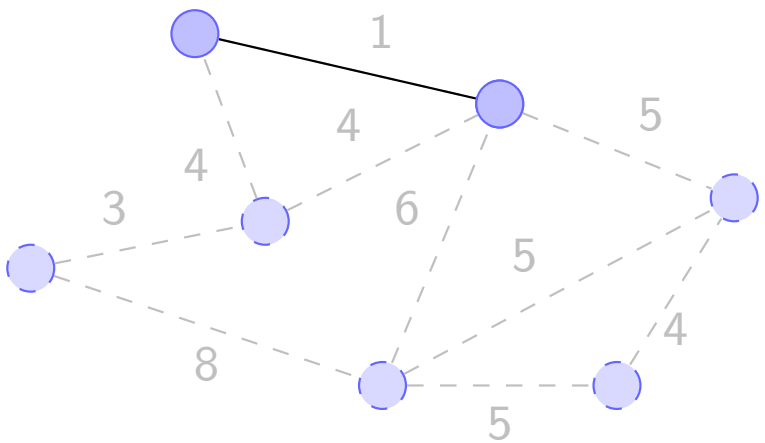
- **Initialisation :** un arbre T comportant un seul sommet quelconque de G
- **Itération :** ajout à l'arbre T d'une arête de poids minimal reliant T à un sommet non-encore présent
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Prim (1957)

Aperçu

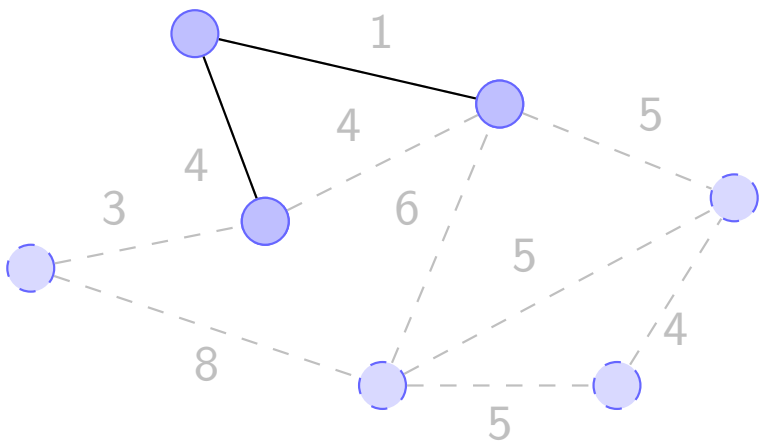
- **Initialisation :** un arbre T comportant un seul sommet quelconque de G
- **Itération :** ajout à l'arbre T d'une arête de poids minimal reliant T à un sommet non-encore présent
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Prim (1957)

Aperçu

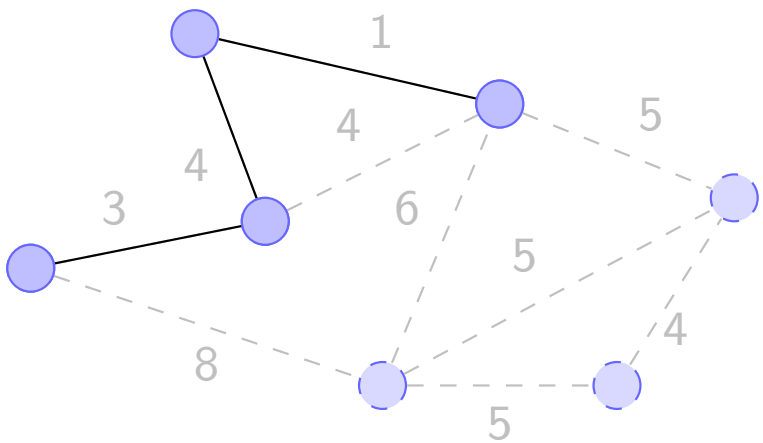
- **Initialisation :** un arbre T comportant un seul sommet quelconque de G
- **Itération :** ajout à l'arbre T d'une arête de poids minimal reliant T à un sommet non-encore présent
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Prim (1957)

Aperçu

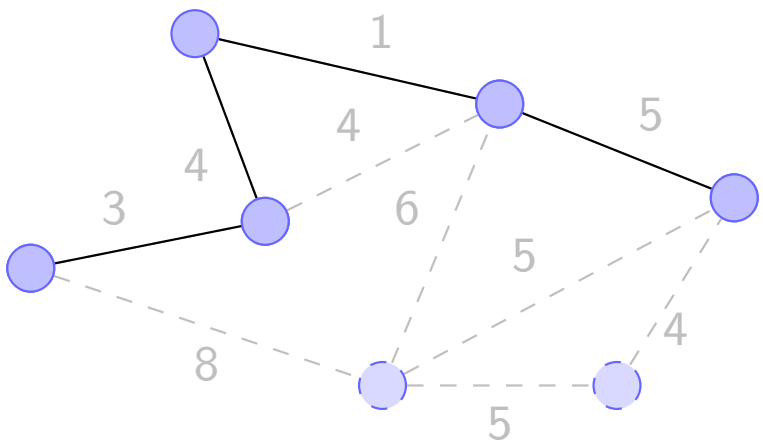
- **Initialisation :** un arbre T comportant un seul sommet quelconque de G
- **Itération :** ajout à l'arbre T d'une arête de poids minimal reliant T à un sommet non-encore présent
- **Arrêt :** après avoir ajouté $n-1$ arêtes



Approche de Prim (1957)

Aperçu

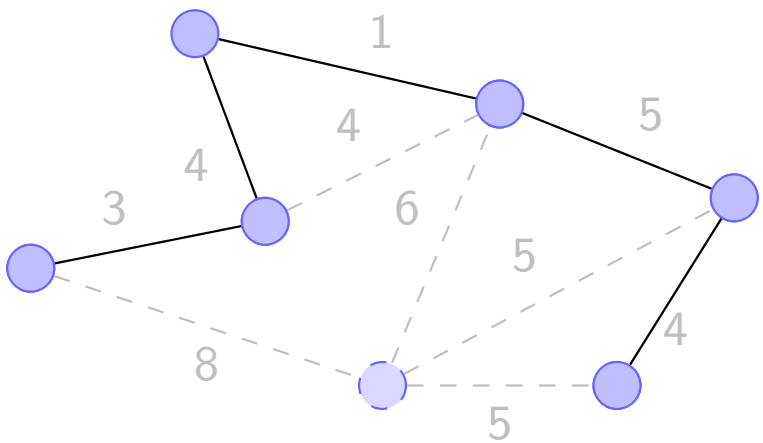
- **Initialisation :** un arbre T comportant un seul sommet quelconque de G
- **Itération :** ajout à l'arbre T d'une arête de poids minimal reliant T à un sommet non-encore présent
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Prim (1957)

Aperçu

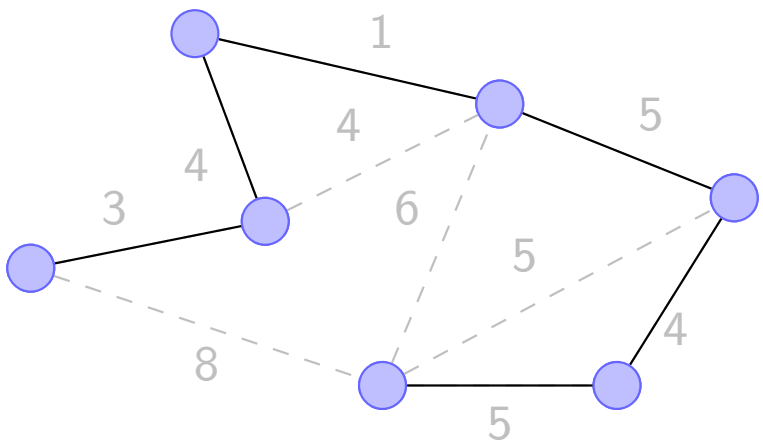
- **Initialisation :** un arbre T comportant un seul sommet quelconque de G
- **Itération :** ajout à l'arbre T d'une arête de poids minimal reliant T à un sommet non-encore présent
- **Arrêt :** après avoir ajouté $n - 1$ arêtes



Approche de Prim (1957)

Aperçu

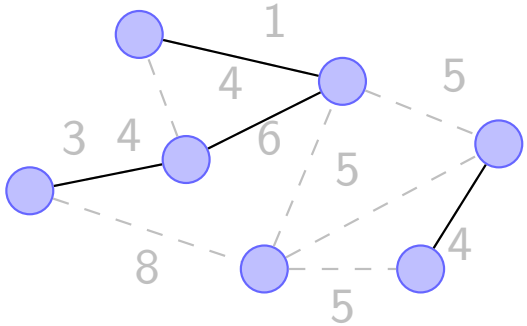
- **Initialisation :** un arbre T comportant un seul sommet quelconque de G
- **Itération :** ajout à l'arbre T d'une arête de poids minimal reliant T à un sommet non-encore présent
- **Arrêt :** après avoir ajouté $n - 1$ arêtes

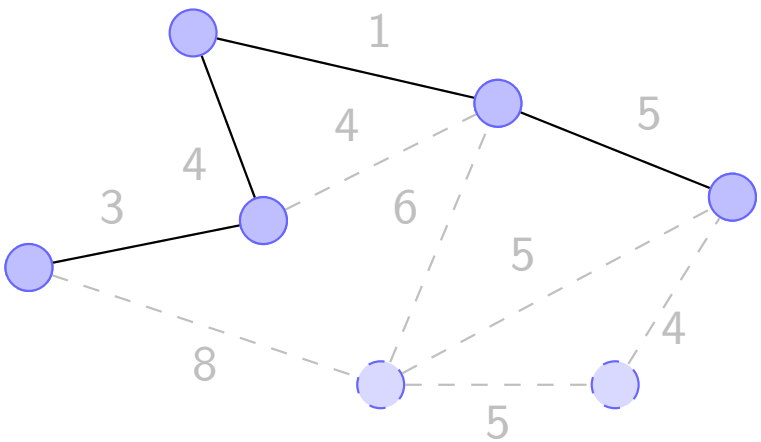


Observations

Différence entre les deux approches

- Kruskal est toujours couvrant (forêt) et construit la connexité
- Prim est toujours connexe (sous-arbre) et construit la couverture
- Les deux se préservent des cycles





Observations

Différence entre les deux approches

- Kruskal est toujours couvrant (forêt) et construit la connexité
- Prim est toujours connexe (sous-arbre) et construit la couverture
- Les deux se préservent des cycles
- Les deux s'arrêtent après $n - 1$ arêtes. Pourquoi ?

Observations

Différence entre les deux approches

- Kruskal est toujours couvrant (forêt) et construit la connexité
- Prim est toujours connexe (sous-arbre) et construit la couverture
- Les deux se préservent des cycles
- Les deux s'arrêtent après $n-1$ arêtes. Pourquoi ?

Théorème

Soit T un graphe à n sommets.
Ces trois propositions sont équivalentes :

- T est un arbre : connexe et acyclique
- T est connexe et a $n-1$ arêtes
- T est acyclique et a $n-1$ arêtes

Sauriez-vous en faire la preuve ?

Détails de l'algorithme de Prim

Principe

Maintenir une structure `nextnodes` contenant les nœuds restants tout en mettant à jour leur distance à l'arbre courant :

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Détails de l'algorithme de Prim

Principe

Maintenir une structure `nextnodes` contenant les nœuds restants tout en mettant à jour leur distance à l'arbre courant :

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Itération

1. Extraire le sommet de `nextnodes` de distance minimale à l'arbre courant T
2. L'ajouter à l'arbre T avec l'arête minimale correspondante
3. Mettre à jour les distances de ses voisins

Détails de l'algorithme de Prim

Principe

Maintenir une structure `nextnodes` contenant les nœuds restants tout en mettant à jour leur distance à l'arbre courant :

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Itération

1. Extraire le sommet de `nextnodes` de distance minimale à l'arbre courant T
2. L'ajouter à l'arbre T avec l'arête minimale correspondante
3. Mettre à jour les distances de ses voisins

➜ Ça ressemble beaucoup à PCC (BFS) !

Détails de l'algorithme de Prim

Principe

Maintenir une structure `nextnodes` contenant les nœuds restants tout en mettant à jour leur distance à l'arbre courant :

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Itération

1. Extraire le sommet de `nextnodes` de distance minimale à l'arbre courant T
2. L'ajouter à l'arbre T avec l'arête minimale correspondante
3. Mettre à jour les distances de ses voisins

➜ Ça ressemble beaucoup à PCC (BFS) !

A la fin

`nextnodes` est vide et T est un arbre couvrant de poids minimal

Algorithme de Prim pour résoudre le problème MST

```
def Prim_MST(graph,s):
    nextnodes = nodes(graph)
    parent = {}
    dist = {}
    dist[s] = 0
    Au premier tour c'est toujours s qui sera choisi

    while len(nextnodes)>0:
        x = extract_min_dist(nextnodes,dist)
        mise à jour des voisins:
        for y in neighbors(graph, x):
            new_dist = distance(graph,x,y)
            if (y in nextnodes) and \
                    (y not in dist or new_dist < dist[y]):
                dist[y] = new_dist
                parent[y] = x

    return parent
```

Plus courts Chemins vs Prim

Prim

```
def Prim_MST(graph,s):
  nextnodes = nodes(graph)
  parent = {}
  dist = {}; dist[s] = 0
  while len(nextnodes)>0:
    x = extract_min_dist(nextnodes,dist)
    for y in neighbors(graph, x):


      new_dist = distance(graph,x,y)

      if (y in nextnodes) and \
           (y not in dist or \
           new_dist < dist[y]):
        dist[y] = new_dist
        parent[y] = x
  return parent
```

Plus courts Chemins

```
def shortest_path(graph,s):
  frontier = [s]
  parent = {}; parent[s] = None
  dist = {}; dist[s] = 0
  while len(frontier)>0:
    x = extract_min_dist(frontier,dist)
    for y in neighbors(graph, x):
      if y not in parent:
        frontier.append(y)
      new_dist = dist[x] + \
                 distance(graph,x,y)

      if y not in dist or \
           dist[y] > new_dist:
        dist[y] = new_dist
        parent[y] = x
  return parent
```

Plus courts Chemins vs Prim

Prim

```
def Prim_MST(graph,s):
  nextnodes = nodes(graph)
  parent = {}
  dist = {}; dist[s] = 0
  while len(nextnodes)>0:
    x = extract_min_dist(nextnodes,dist)
    for y in neighbors(graph, x):


      new_dist = distance(graph,x,y)

      if (y in nextnodes) and \
           (y not in dist or \
           new_dist < dist[y]):
        dist[y] = new_dist
        parent[y] = x
  return parent
```

Plus courts Chemins

```
def shortest_path(graph,s):
  frontier = [s]
  parent = {}; parent[s] = None
  dist = {}; dist[s] = 0
  while len(frontier)>0:
    x = extract_min_dist(frontier,dist)
    for y in neighbors(graph, x):
      if y not in parent:
        frontier.append(y)
      new_dist = dist[x] + \
                 distance(graph,x,y)

      if y not in dist or \
           dist[y] > new_dist:
        dist[y] = new_dist
        parent[y] = x
  return parent
```

La différence provient du stockage dans la file

- BFS : éléments classés par nombre d'arêtes depuis s
- Plus court chemin : éléments classés par distance à s
- Prim : éléments classés par distance à l'arbre en construction

Détails de l'algorithme de Kruskal

Au départ :

La forêt F (ensemble d'arbres) est composé de n sommets isolés (n arbres singletons).

Détails de l'algorithme de Kruskal

Au départ :

La forêt F (ensemble d'arbres) est composé de n sommets isolés (n arbres singletons).

Itération

Regrouper deux arbres voisins par une arête minimale qu'on ajoute à T.

Invariant : F est une forêt (sans cycle)

Détails de l'algorithme de Kruskal

Au départ :

La forêt F (ensemble d'arbres) est composé de n sommets isolés (n arbres singletons).

Itération

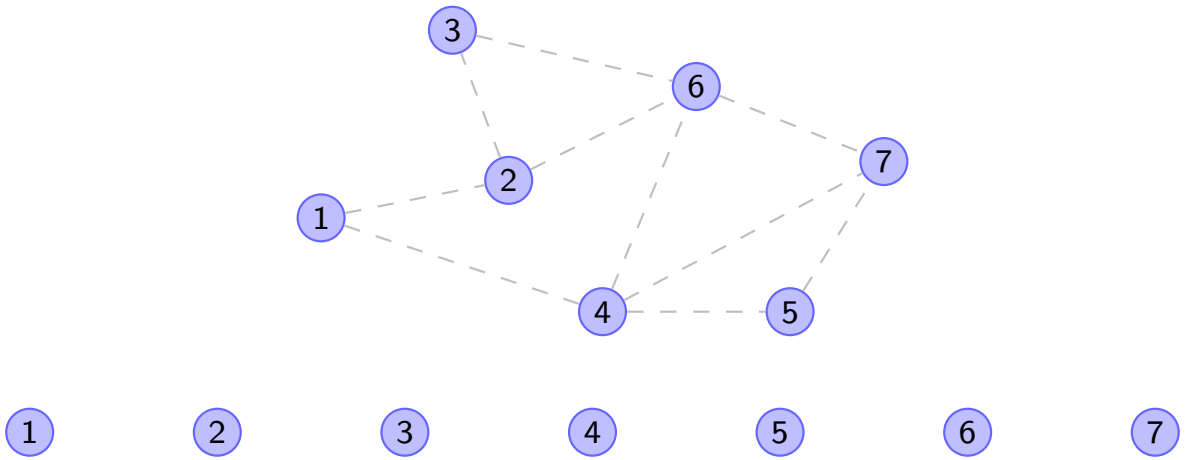
Regrouper deux arbres voisins par une arête minimale qu'on ajoute à T.

Invariant : F est une forêt (sans cycle)

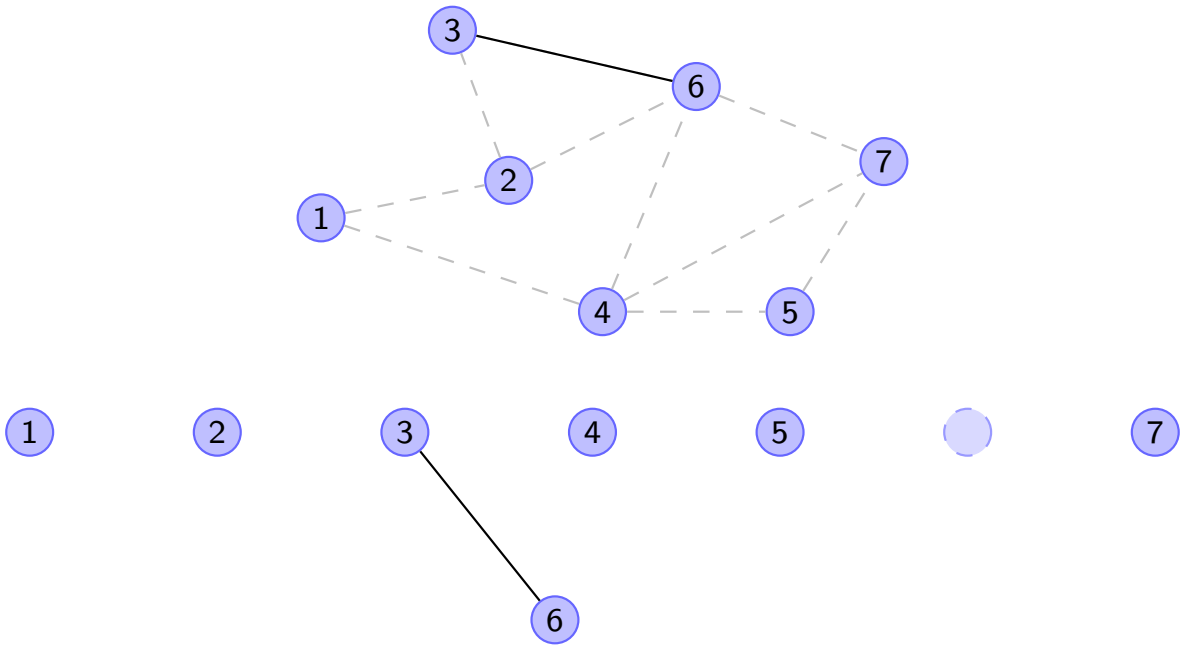
A la fin :

Il ne reste plus qu'un arbre $T = (V, E_T)$ dans F et il est couvrant de poids minimal.

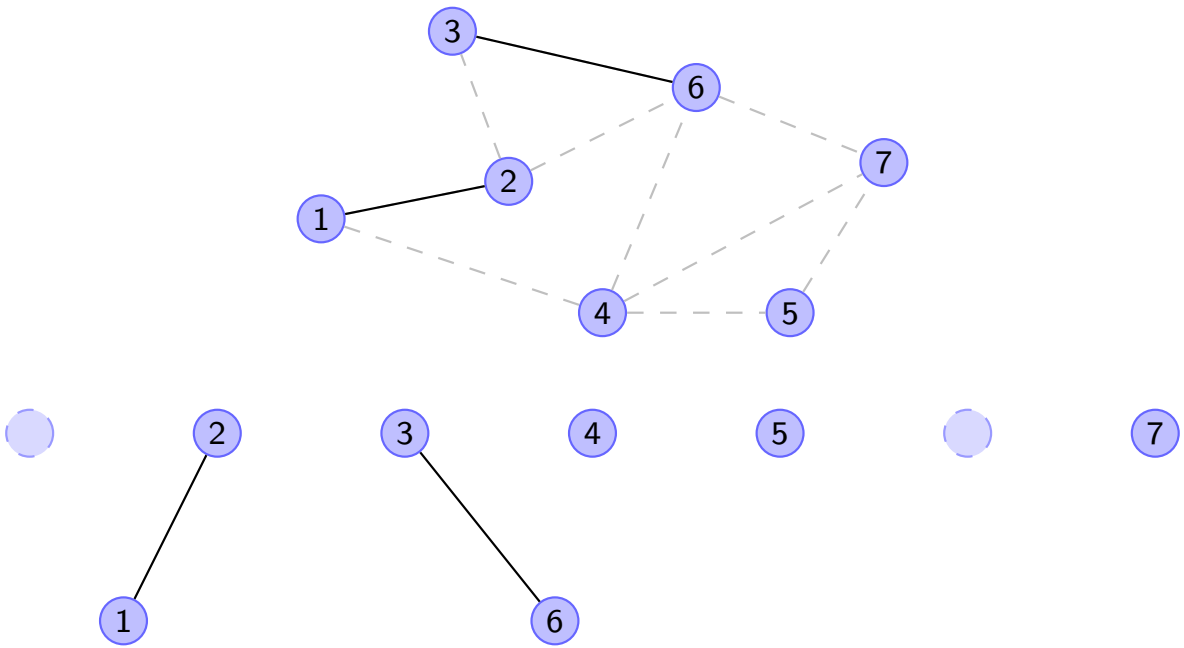
Forêt $\Longrightarrow$ Arbre



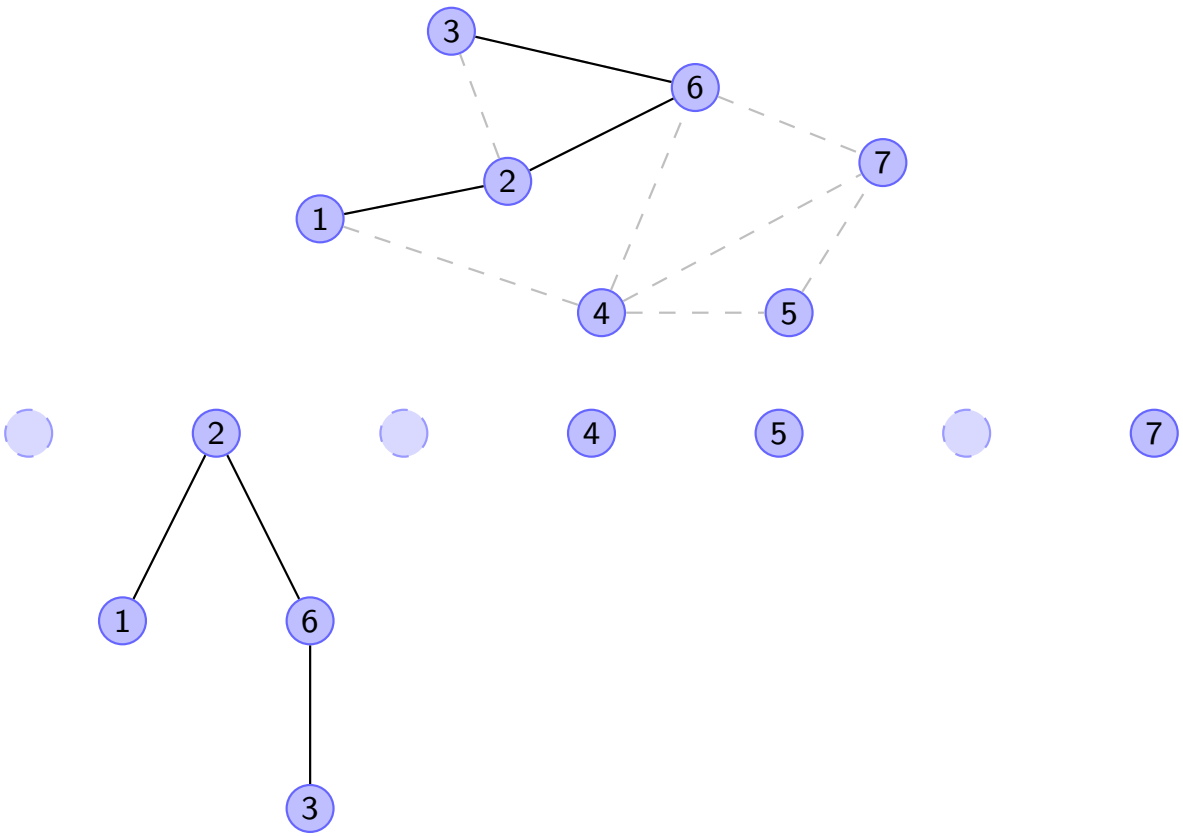
Forêt $\Longrightarrow$ Arbre



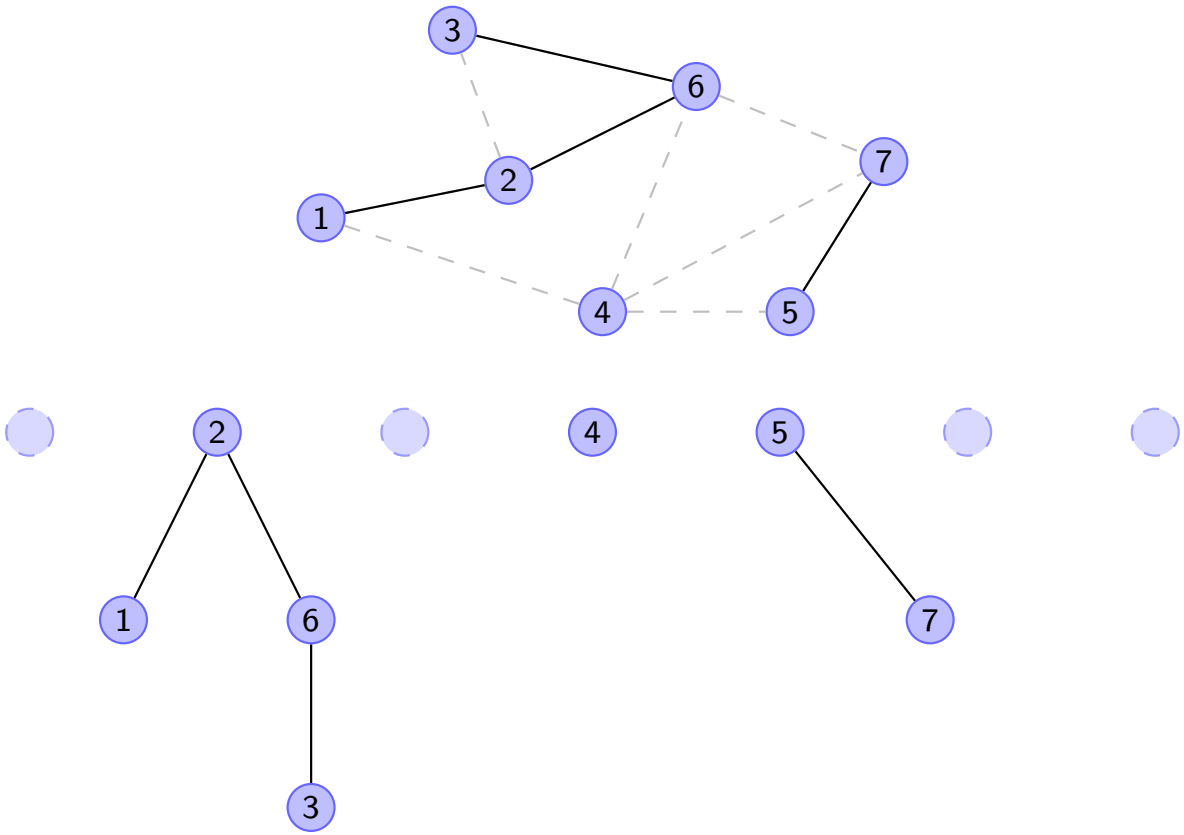
Forêt $\Longrightarrow$ Arbre



Forêt $\Longrightarrow$ Arbre



Forêt $\Longrightarrow$ Arbre



Algorithme de Kruskal pour résoudre le problème MST

```
def Kruskal_MST(graph):
    max = len(nodes(graph))-1
    MST = []                    # list of selected edges
    forest = {}                 # set of trees
    for v in nodes(graph):
        tree = create_tree(v)
        forest.add(tree)

    cpt = 0
    for (u, v) in sort_by_weight(edges(graph)):
        if find_tree(u, forest) != find_tree(v, forest):
            MST.append((u, v))
            merge_trees(u, v, forest)

            cpt = cpt + 1
            if cpt == max:
                break
    return MST
```

Complexité

Algorithme de Kruskal (détaillée plus loin)

La complexité de l'algorithme de Kruskal est en $\mathcal{O}(|E|\log(|E|))$

➜ Avec $|E| \approx |V|^2$ on obtient une complexité $\mathcal{O}(|V|^2\log(|V|))$

Complexité

Algorithme de Kruskal (détaillée plus loin)

La complexité de l'algorithme de Kruskal est en $\mathcal{O}(|E|\log(|E|))$

➜ Avec $|E| \approx |V|^2$ on obtient une complexité $\mathcal{O}(|V|^2 \log(|V|))$

Algorithme de Prim

La complexité de l'algorithme de Prim est égale à la complexité de l'algorithme des plus courts chemins :

➜ Avec $|E| \approx |V|$ on obtient une complexité $\mathcal{O}(|V|\log(|V|))$ tas binaire

➜ Avec $|E| \approx |V|^2$ on obtient une complexité $\mathcal{O}(|V|^2)$ liste

Complexité

Algorithme de Kruskal (détaillée plus loin)

La complexité de l'algorithme de Kruskal est en $\mathcal{O}(|E|\log(|E|))$

➜ Avec $|E| \approx |V|^2$ on obtient une complexité $\mathcal{O}(|V|^2 \log(|V|))$

Algorithme de Prim

La complexité de l'algorithme de Prim est égale à la complexité de l'algorithme des plus courts chemins :

➜ Avec $|E| \approx |V|$ on obtient une complexité $\mathcal{O}(|V|\log(|V|))$ tas binaire

➜ Avec $|E| \approx |V|^2$ on obtient une complexité $\mathcal{O}(|V|^2)$ liste

Conséquence

Une solution optimale du problème MST peut être obtenue en temps polynomial !

Optimalité

Comment être certains que les algorithmes de Prim et de Kruskal construisent bien un arbre couvrant minimal ?

Rappel

Algorithme glouton pour construire un MST T partir de $G = (V, E)$:

- Au début, $E' = \emptyset$
- À chaque itération, on choisit une nouvelle arête sûre (u, v)
  ➜ $E' = E' \cup (u, v)$ doit toujours être un sous-ensemble d'un MST de G
- A la fin quand $|E'| = |V| - 1$ on obtient une MST $T = (V, E_T = E')$

Optimalité

Comment être certains que les algorithmes de Prim et de Kruskal construisent bien un arbre couvrant minimal ?

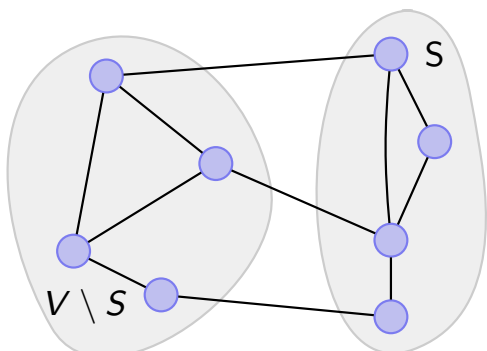
Rappel

Algorithme glouton pour construire un MST T partir de $G = (V, E)$:

- Au début, $E' = \emptyset$
- À chaque itération, on choisit une nouvelle arête sûre (u, v)
  ➜ $E' = E' \cup (u, v)$ doit toujours être un sous-ensemble d'un MST de G
- A la fin quand $|E'| = |V| - 1$ on obtient une MST $T = (V, E_T = E')$

Notion de coupe

➜ Une coupe est une partition des sommets en deux ensembles non vides S et $(V \backslash S)$ disjoints



Propriété : arête sûre et coupe

Définitions

Considérons $S \subset V$ et $E' \subseteq E$. On dit que :

- une arête traverse la coupe $(S, V \backslash S)$ si une extrémité est dans S et l'autre est dans $(V \backslash S)$
- la coupe $(S, V \backslash S)$ respecte E' s'il n'y a pas d'arêtes dans E' qui traverse la coupe

Propriété : arête sûre et coupe

Définitions

Considérons $S \subset V$ et $E' \subseteq E$. On dit que :

- une arête traverse la coupe $(S, V \backslash S)$ si une extrémité est dans S et l'autre est dans $(V \backslash S)$
- la coupe $(S, V \backslash S)$ respecte E' s'il n'y a pas d'arêtes dans E' qui traverse la coupe

Propriété de l'algorithme glouton

- E' un sous ensemble d'un MST T de G $\qquad E' \subseteq E_T \subseteq E$
- $(S, V \backslash S)$ une coupe qui respecte E'
- (u, v) une arête de poids minimal qui traverse la coupe $(S, V \backslash S)$

➜ alors (u, v) est une arête sûre pour E'.

c'est-à-dire que $E' \cup (u, v)$ sera toujours un sous-ensemble d'un MST de G

Propriété : arête sûre et coupe

Définitions

Considérons $S \subset V$ et $E' \subseteq E$. On dit que :

- une arête traverse la coupe $(S, V \backslash S)$ si une extrémité est dans S et l'autre est dans $(V \backslash S)$
- la coupe $(S, V \backslash S)$ respecte E' s'il n'y a pas d'arêtes dans E' qui traverse la coupe

Propriété de l'algorithme glouton

- E' un sous ensemble d'un MST T de G $\qquad E' \subseteq E_T \subseteq E$
- $(S, V \backslash S)$ une coupe qui respecte E'
- (u, v) une arête de poids minimal qui traverse la coupe $(S, V \backslash S)$

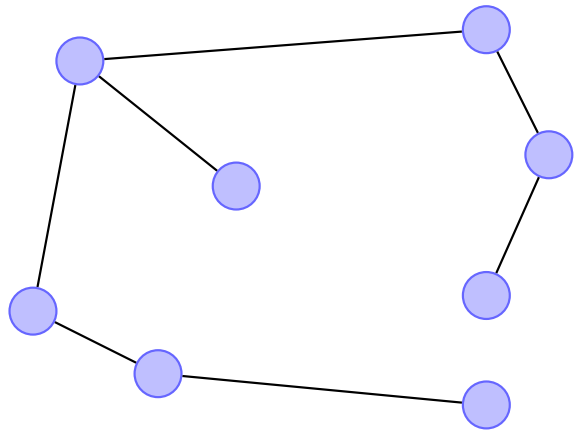
➜ alors (u, v) est une arête sûre pour E'.

c'est-à-dire que $E' \cup (u, v)$ sera toujours un sous-ensemble d'un MST de G

Ça tombe bien, c'est ce que font Kruskal et Prim...

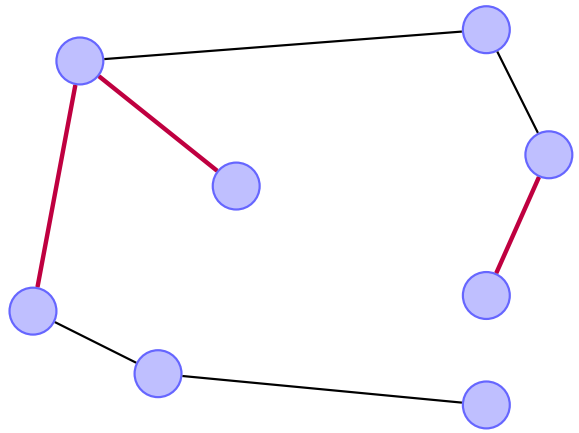
Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$



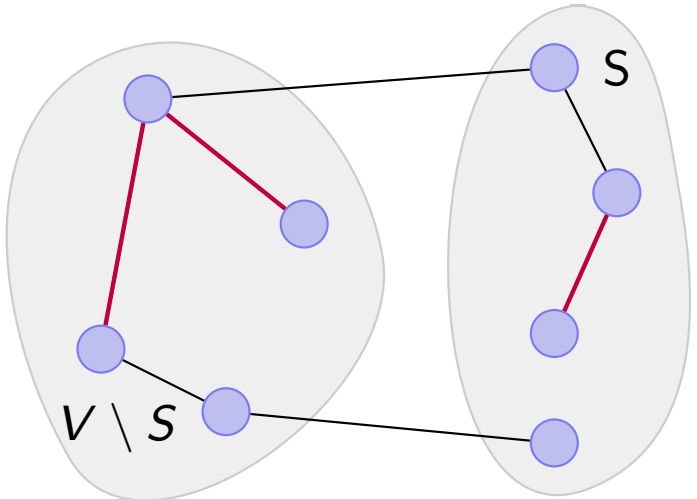
Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$



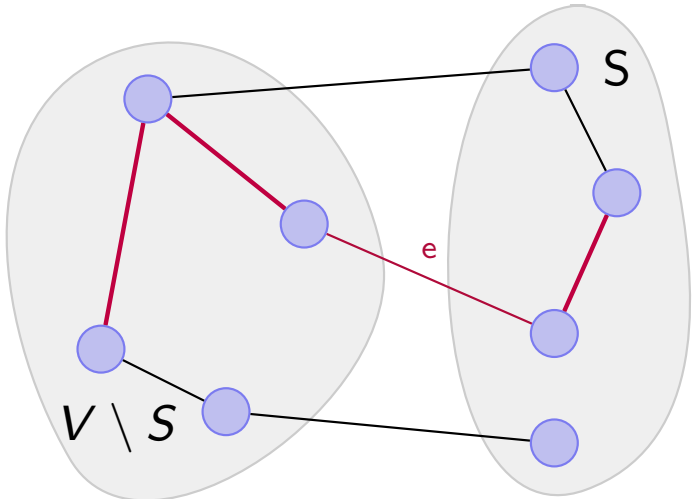
Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$
- Soit $(S, V \backslash S)$, une coupe de G respectant E'



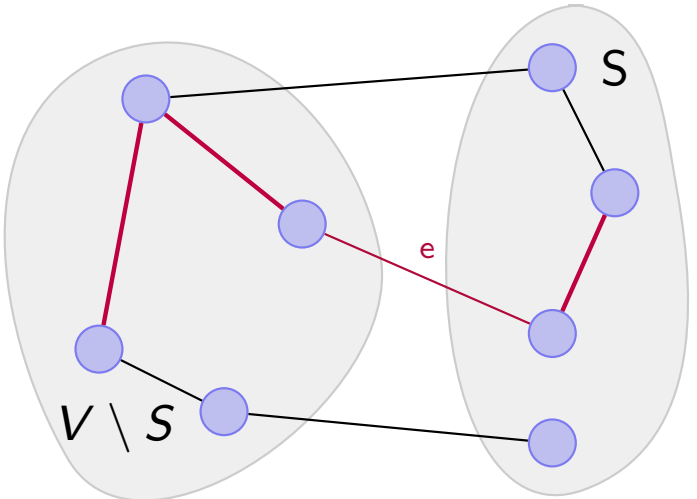
Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$
- Soit $(S, V \backslash S)$, une coupe de G respectant E'
- Soit e, l'arête de poids minimal qui traverse la coupe : supposons que T ne contient pas e



Preuve

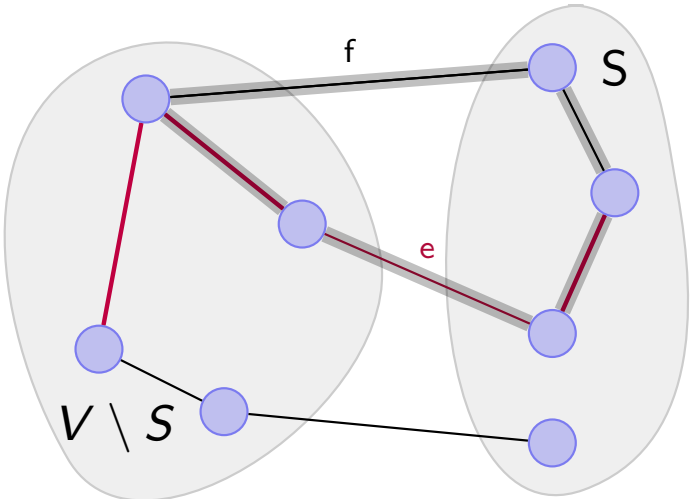
- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$
- Soit $(S, V \backslash S)$, une coupe de G respectant E'
- Soit e, l'arête de poids minimal qui traverse la coupe : supposons que T ne contient pas e



Montrons qu'il est possible de construire un arbre T' incluant $E' \cup \{e\}$

Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$
- Soit $(S, V \backslash S)$, une coupe de G respectant E'
- Soit e, l'arête de poids minimal qui traverse la coupe : supposons que T ne contient pas e

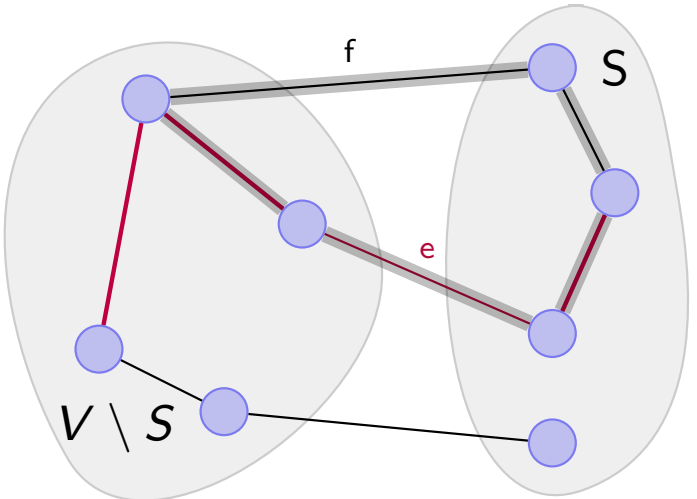


Montrons qu'il est possible de construire un arbre T' incluant $E' \cup \{e\}$

- L'ajout de e à T introduit nécessairement un cycle (noté C).
- Soit $f \neq e$ une arête de C entre S et $V \backslash S$.

Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$
- Soit $(S, V \backslash S)$, une coupe de G respectant E'
- Soit e, l'arête de poids minimal qui traverse la coupe : supposons que T ne contient pas e



Montrons qu'il est possible de construire un arbre T' incluant $E' \cup \{e\}$

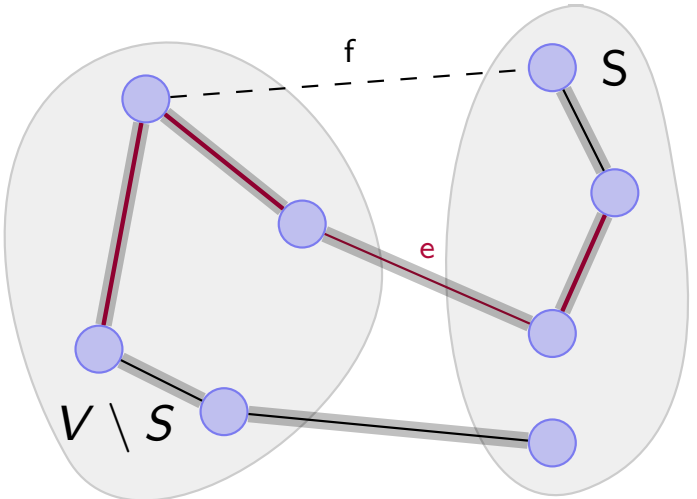
- L'ajout de e à T introduit nécessairement un cycle (noté C).
- Soit $f \neq e$ une arête de C entre S et $V \backslash S$.

➜ Par définition de e, on a $\omega(e) \leq \omega(f)$

➜ La coupe respecte E', donc f n'est pas dans E'.

Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$
- Soit $(S, V \backslash S)$, une coupe de G respectant E'
- Soit e, l'arête de poids minimal qui traverse la coupe : supposons que T ne contient pas e



Montrons qu'il est possible de construire un arbre T' incluant $E' \cup \{e\}$

- L'ajout de e à T introduit nécessairement un cycle (noté C).
- Soit $f \neq e$ une arête de C entre S et $V \backslash S$.

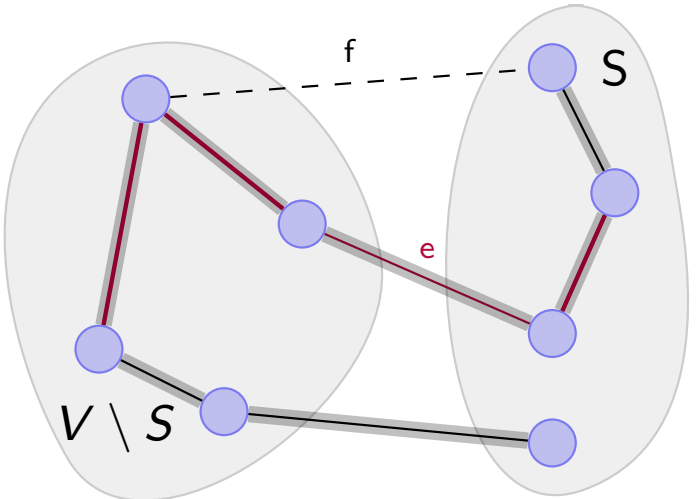
➜ Par définition de e, on a $\omega(e) \leq \omega(f)$

➜ La coupe respecte E', donc f n'est pas dans E'.

- Soit T' l'arbre passant par e au lieu de f $\quad E_{T'} = (E_T \backslash \{f\}) \cup \{e\}$.
  - ➜ T' est couvrant
  - ➜ T' est minimal $\quad \Omega(T') = \Omega(T) - \omega(f) + \omega(e) \leq \Omega(T)$
  - ➜ T' inclut $E' \cup \{e\}$ $\quad E' \cup \{e\} \subseteq E_{T'}$

Preuve

- Soit T un arbre couvrant minimal d'un graphe $G = (V, E)$, incluant $E' \subseteq E_T \subseteq E$
- Soit $(S, V \backslash S)$, une coupe de G respectant E'
- Soit e, l'arête de poids minimal qui traverse la coupe : supposons que T ne contient pas e



Montrons qu'il est possible de construire un arbre T' incluant $E' \cup \{e\}$

- L'ajout de e à T introduit nécessairement un cycle (noté C).
- Soit $f \neq e$ une arête de C entre S et $V \backslash S$.

➜ Par définition de e, on a $\omega(e) \leq \omega(f)$

➜ La coupe respecte E', donc f n'est pas dans E'.

- Soit T' l'arbre passant par e au lieu de f $\quad E_{T'} = (E_T \backslash \{f\}) \cup \{e\}$.
  - ➜ T' est couvrant
  - ➜ T' est minimal $\quad \Omega(T') = \Omega(T) - \omega(f) + \omega(e) \leq \Omega(T)$
  - ➜ T' inclut $E' \cup \{e\}$ $\quad E' \cup \{e\} \subseteq E_{T'}$

➜ Choisir e de poids minimum parmi les arêtes qui traversent une coupe qui respecte E' donne toujours une arête sûre !

Cas particulier de Prim et Kruskal

Choix d'une arête sûre pour augmenter E'

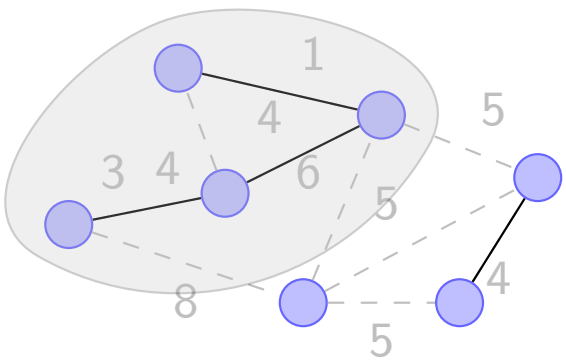
Prim

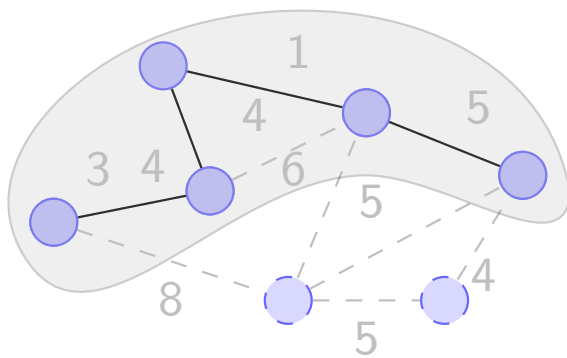
- Coupe : S = ensemble sommets extrémités d'une arête de E'

Kruskal

- Arête minimale $\rightarrow$ choix de la coupe !

➜ Couper entre les deux composantes qu'on va relier





Plan

Implémentation de l'algorithme de Kruskal

Choix d'une structure de données

Il faut pouvoir représenter T et ses différentes composantes (sous-arbres).

Contraintes

On souhaite réaliser les opérations suivantes :

- initialiser la structure (forêt "d'arbustes")
- pour chaque sommet v déterminer l'indice d'une composante dont il fait partie
- fusionner deux composantes

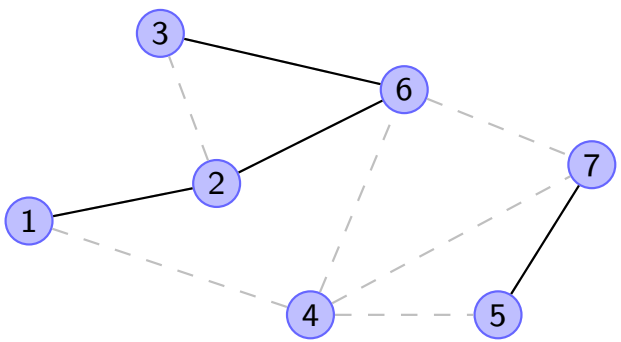
Approche naïve

Structure la plus simple

Un tableau `Kruskal_tab` de taille n dont chaque case i contient l'indice de la composante contenant le sommet i.

```
Kruskal_tab = [1,1,1,4,5,1,5]
```

et on garde la liste des arêtes choisies `[(3,6),(1,2),...]`



Approche naïve

Structure la plus simple

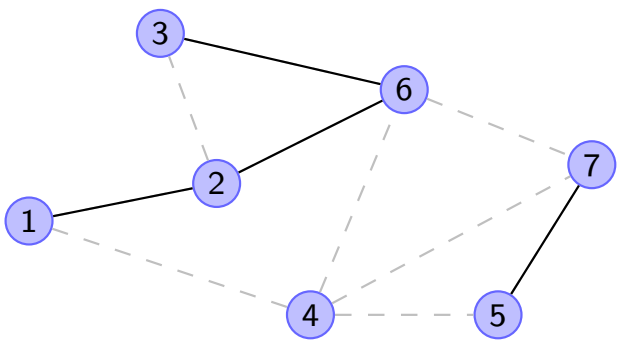
Un tableau `Kruskal_tab` de taille n dont chaque case i contient l'indice de la composante contenant le sommet i.

```
Kruskal_tab = [1,1,1,4,5,1,5]
```

et on garde la liste des arêtes choisies `[(3,6),(1,2),...]`

Coût

- initialisation : $\mathcal{O}(|V|)$
- déterminer l'indice : $\mathcal{O}(1)$
- fusion : $\mathcal{O}(|V|)$



Approche à base de *Union-Find*

Principe

Structure de données pour **partitionner** un ensemble

- **Trouver** la partition à laquelle appartient un élément
- **Unir** les éléments de deux partitions

Approche à base de *Union-Find*

Principe

Structure de données pour **partitionner** un ensemble

- **Trouver** la partition à laquelle appartient un élément
- **Unir** les éléments de deux partitions

Plusieurs approches

- Listes chaînées
- Arborescences

Approche à base de *Union-Find*

Principe

Structure de données pour **partitionner** un ensemble

- **Trouver** la partition à laquelle appartient un élément
- **Unir** les éléments de deux partitions

Plusieurs approches

- Listes chaînées
- Arborescences

Meilleur coût : Arborescences

- initialisation : $\mathcal{O}(|V|)$
- déterminer l'indice : $\mathcal{O}(\log(|V|))$ (hauteur de l'arbre)
- fusion : $\mathcal{O}(\log(|V|))$ (équilibrer l'arbre)

Complexité de l'algorithme de Kruskal

Coût total

- tri initial des arêtes : $\mathcal{O}(|E|\log(|E|))$

  Donc $\mathcal{O}(|E|\log(|V|))$ *car* $|E| \leq |V|^2$
- initialisation de la structure de T : $\mathcal{O}(|V|)$
- détermination des indices faite au pire $2 \times |E|$ fois
  - par tableau : $\mathcal{O}(|E|)$
  - par arborescences : $\mathcal{O}(|E|\log(|V|))$
- fusion faite au pire $|V| - 1$ fois
  - par tableau : $\mathcal{O}(|V|^2)$
  - par arborescences : $\mathcal{O}(|V|\log(|V|))$

Plan

Motivation du Clustering (partitionnement de données)

- Soit un ensemble d'objets avec des distances les séparant :
  - les objets peuvent être des images, pages web, personnes, documents ....
- fonction de distance
  - une petite distance correspond à une grande similarité.

Motivation du Clustering (partitionnement de données)

- Soit un ensemble d'objets avec des distances les séparant :
  - les objets peuvent être des images, pages web, personnes, documents ....
- fonction de distance
  - une petite distance correspond à une grande similarité.

Objectif du Clustering

grouper les objets dans des clusters, où chaque cluster est un ensemble d'objets similaires.

Formalisation du problème de Clustering

- Soit O un ensemble de n objets $o_1, o_2, \ldots, o_n$.
- Pour chaque paire o_i et o_j, nous disposons d'une distance positive $d(o_i, o_j)$.

Formalisation du problème de Clustering

- Soit O un ensemble de n objets $o_1, o_2, \ldots, o_n$.
- Pour chaque paire o_i et o_j, nous disposons d'une distance positive $d(o_i, o_j)$.
- Soit un entier naturel k, un k-clustering de O est une partition de O en k sous-ensembles non-vides $C_1, C_2, \ldots C_k$ (clusters).

Formalisation du problème de Clustering

- Soit O un ensemble de n objets $o_1, o_2, \ldots, o_n$.
- Pour chaque paire o_i et o_j, nous disposons d'une distance positive $d(o_i, o_j)$.
- Soit un entier naturel k, un k-clustering de O est une partition de O en k sous-ensembles non-vides $C_1, C_2, \ldots C_k$ (clusters).
- Le spacing du clustering est la plus petite distance entre deux objets appartenant à des clusters différents :

$$\text{spacing}(C_1, C_2, \ldots C_k) = \min\{d(a, b),\ i \neq j \wedge a \in C_i \wedge b \in C_j\}$$

Formalisation du problème de Clustering

- Soit O un ensemble de n objets $o_1, o_2, \ldots, o_n$.
- Pour chaque paire o_i et o_j, nous disposons d'une distance positive $d(o_i, o_j)$.
- Soit un entier naturel k, un k-clustering de O est une partition de O en k sous-ensembles non-vides $C_1, C_2, \ldots C_k$ (clusters).
- Le spacing du clustering est la plus petite distance entre deux objets appartenant à des clusters différents :

$$\text{spacing}(C_1, C_2, \ldots C_k) = \min\{d(a, b),\ i \neq j \wedge a \in C_i \wedge b \in C_j\}$$

Clustering of Maximum Spacing Problem

Trouver un k-clustering de O dont le spacing est maximum parmi tous les k-clusterings possibles.

Algorithme de résolution

- Intuition ?

Algorithme de résolution

- Intuition ?

Appliquer l'algorithme de Kruskal et s'arrêter avant l'ajout des dernières $k-1$ arêtes.

Algorithme de résolution

- Intuition ?

Appliquer l'algorithme de Kruskal et s'arrêter avant l'ajout des dernières $k-1$ arêtes.

- Quelle est la valeur du *spacing* ?

Algorithme de résolution

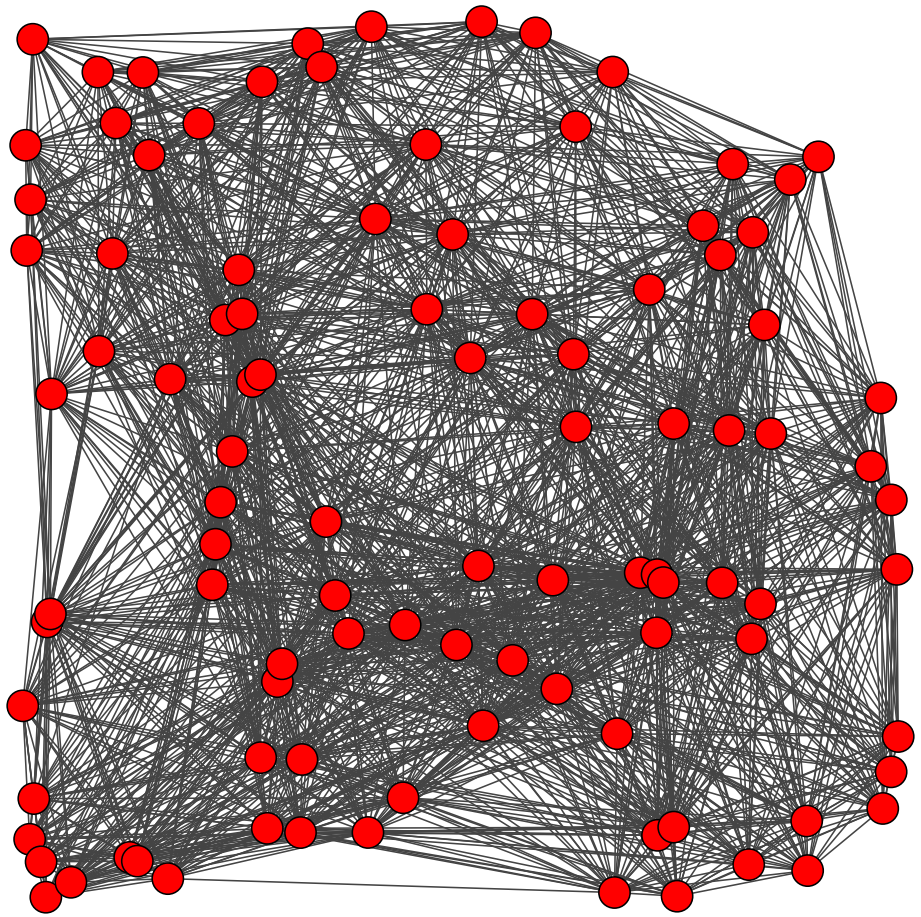
- Intuition ?

Appliquer l'algorithme de Kruskal et s'arrêter avant l'ajout des dernières $k-1$ arêtes.

- Quelle est la valeur du *spacing* ?
  - c'est le poids de la $(k-1)^{eme}$ plus grande arête du MST généré par l'algorithme de Kruskal.

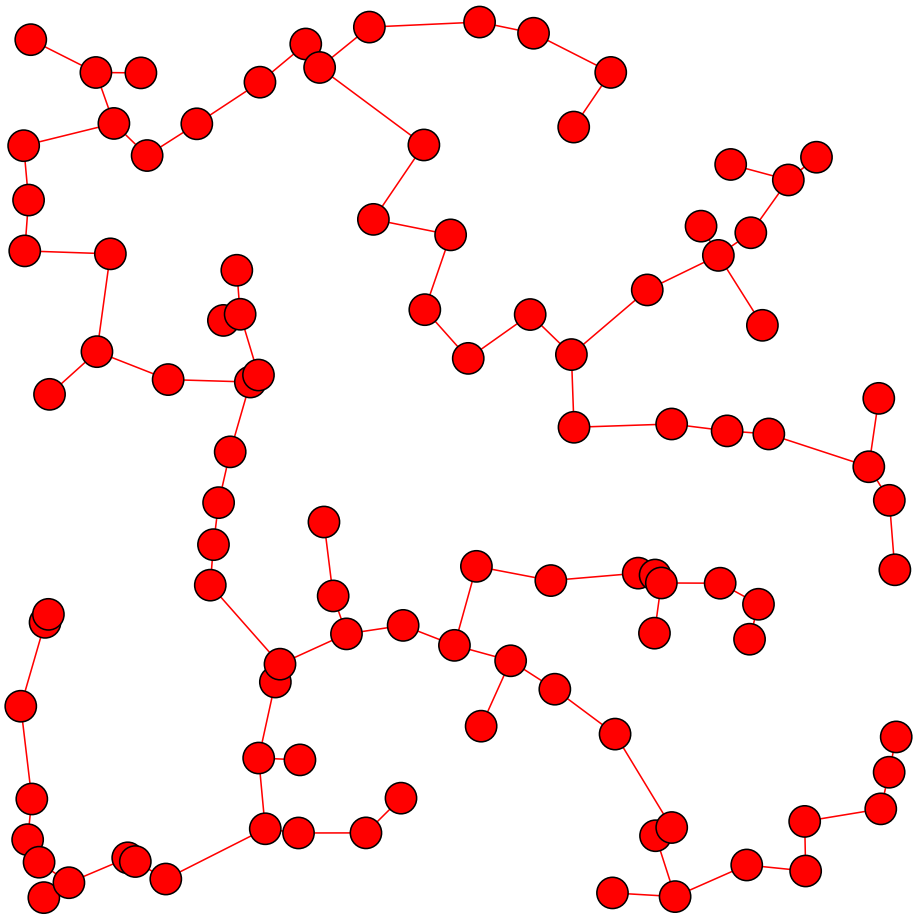
Exemple

Considerons un graphe complet dont les poids sont les distances cartésiennes entre noeuds



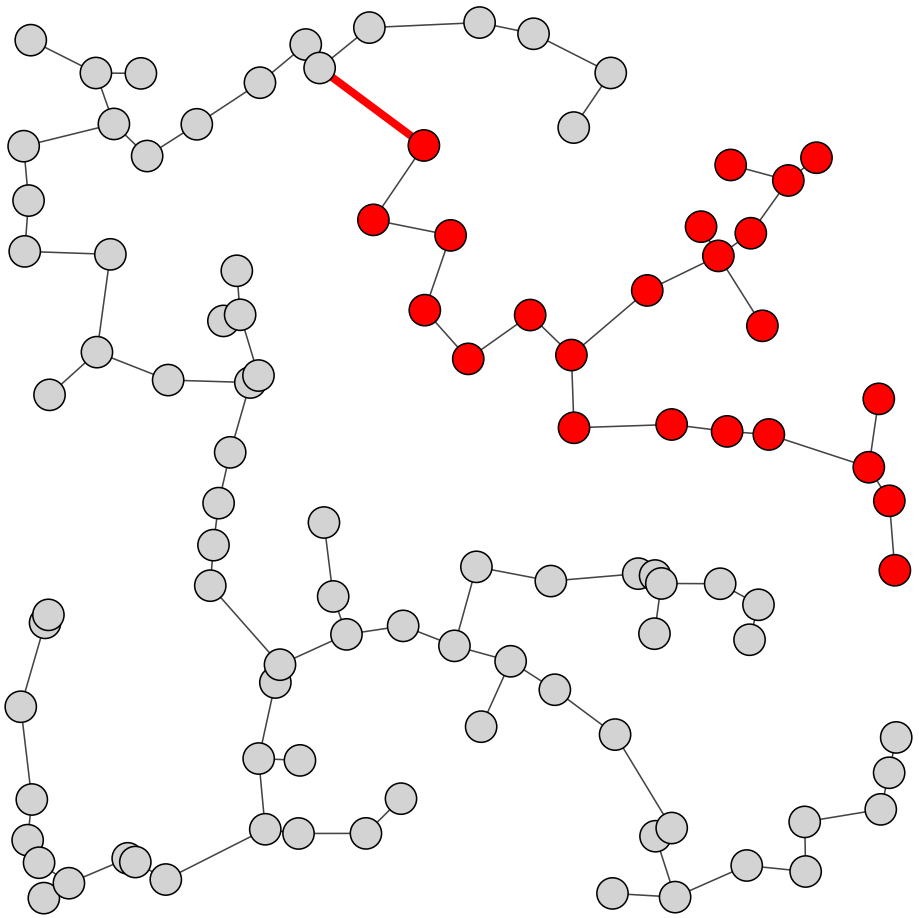
Exemple

Arbre couvrant minimal : Kruskal



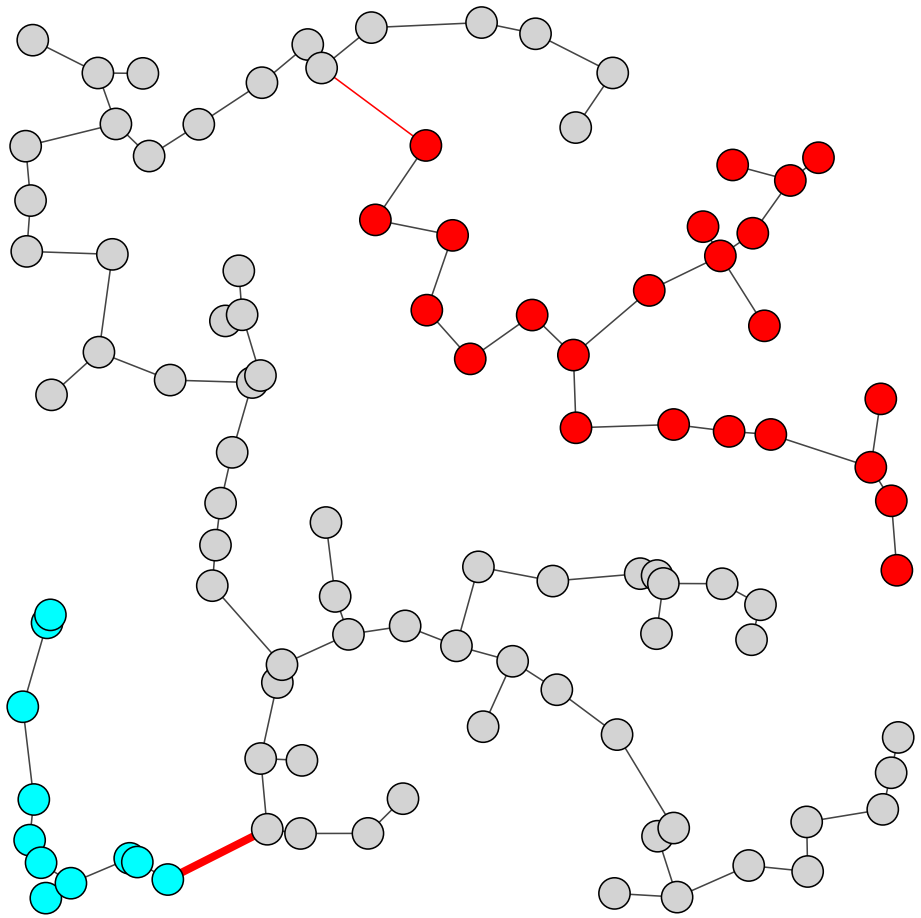
Exemple

2 clusters avant l'ajout de la dernière arête



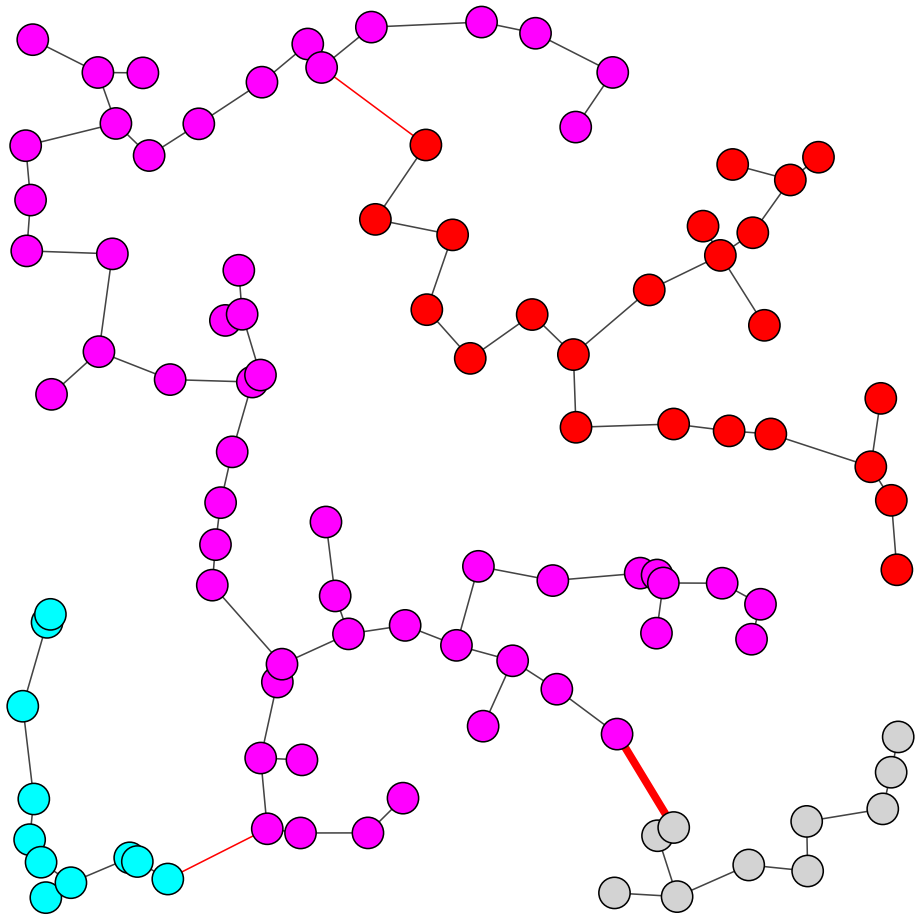
Exemple

3 clusters avant l'ajout des 2 dernières arêtes



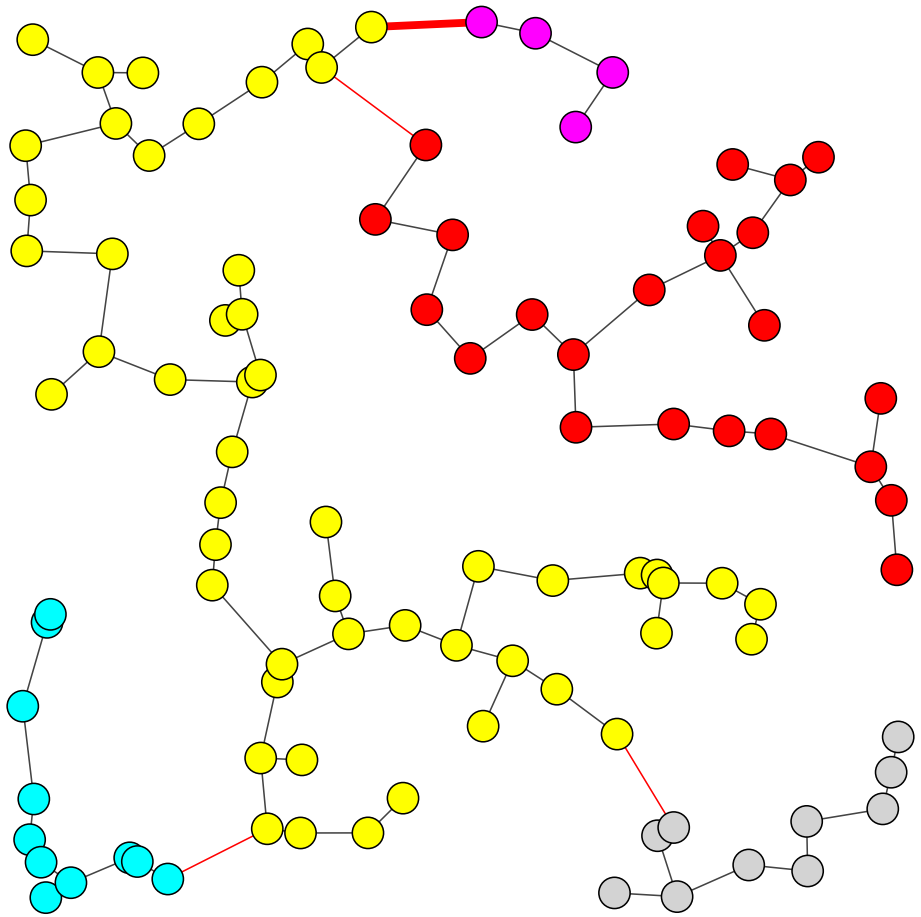
Exemple

Ainsi de suite . . .



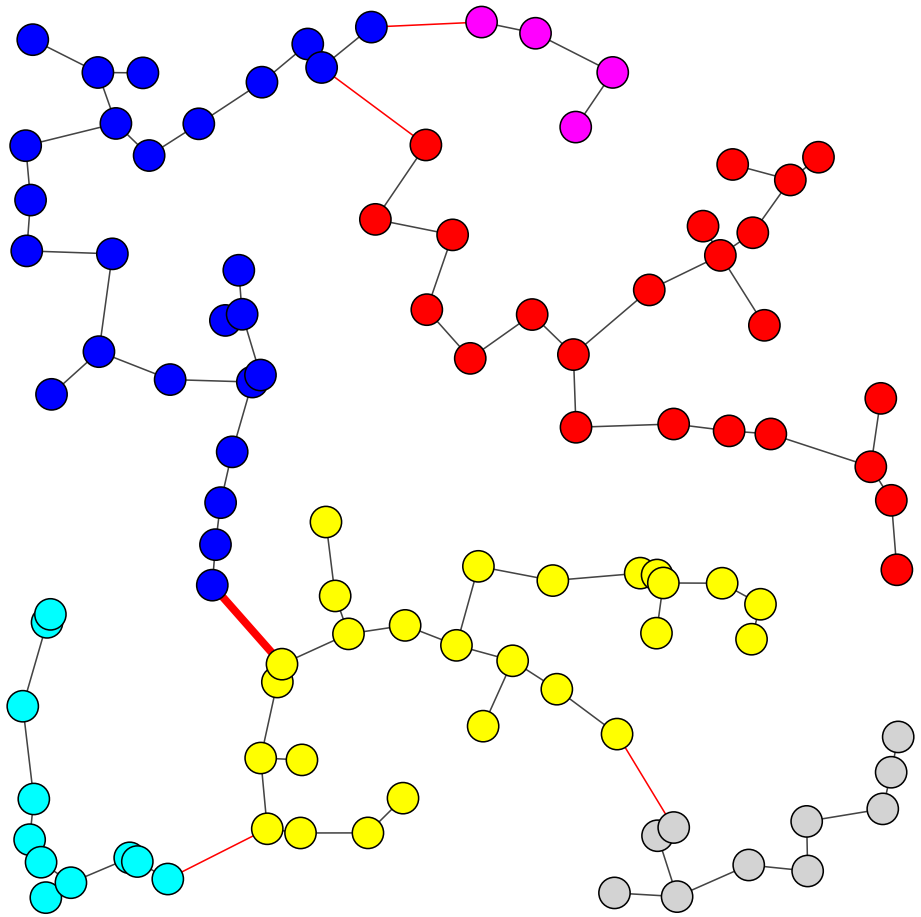
Exemple

Ainsi de suite . . .



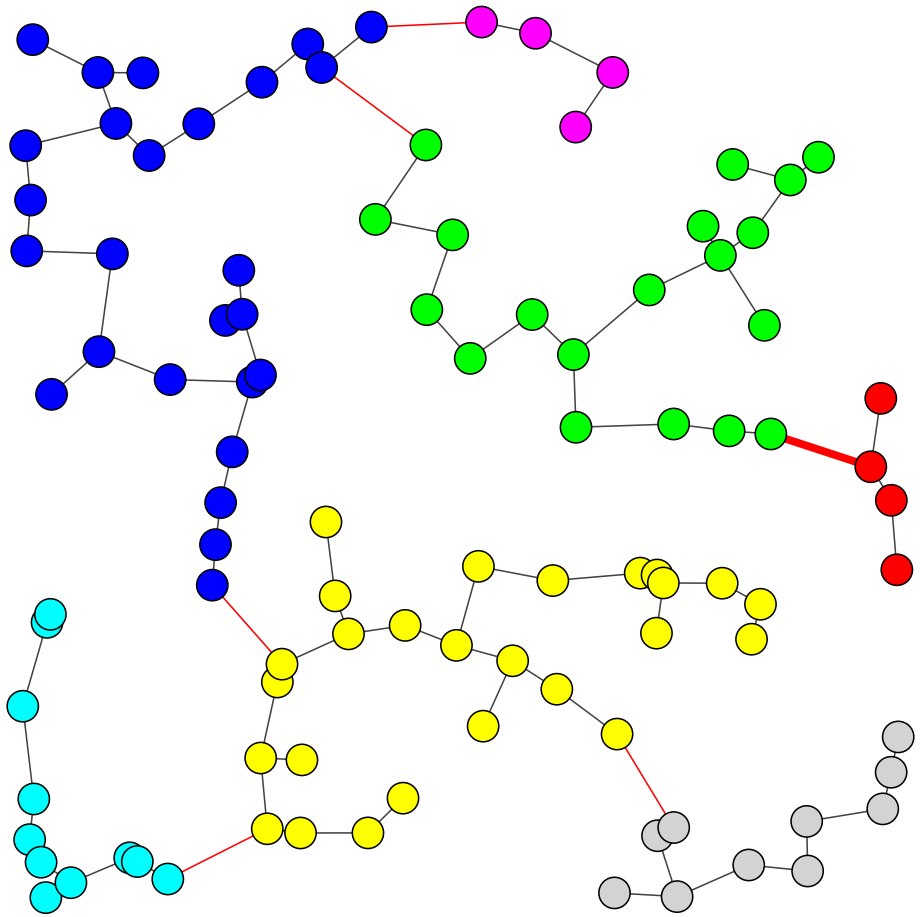
Exemple

Ainsi de suite . . .



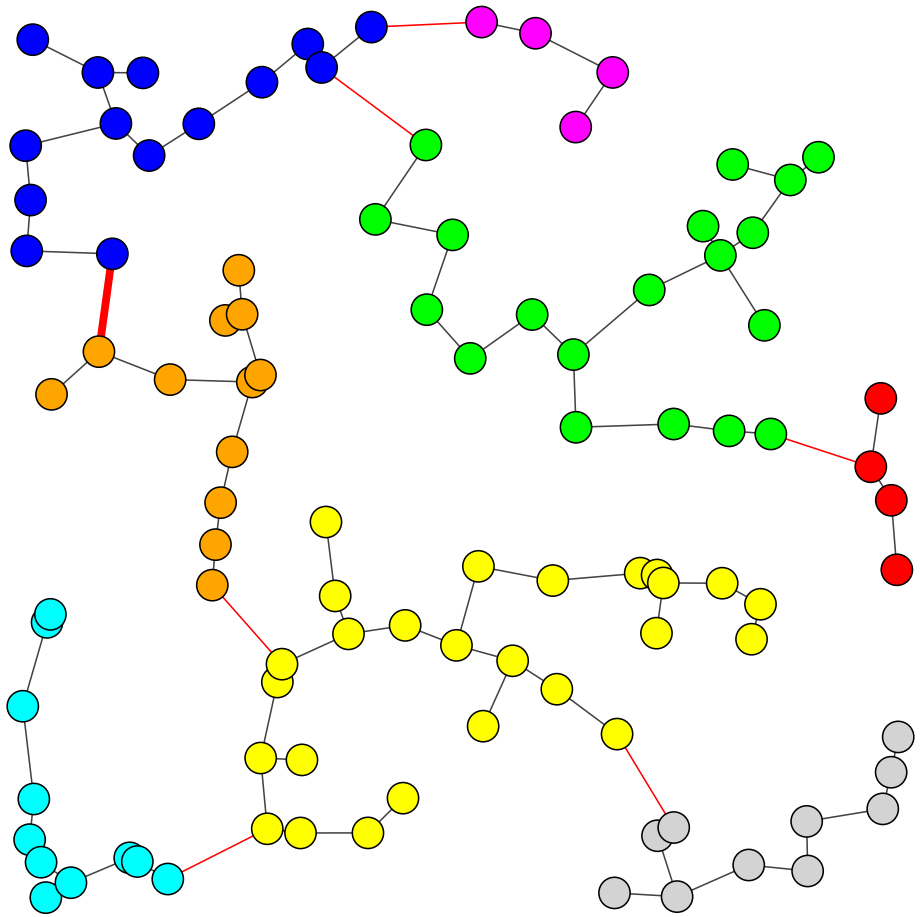
Exemple

Ainsi de suite . . .



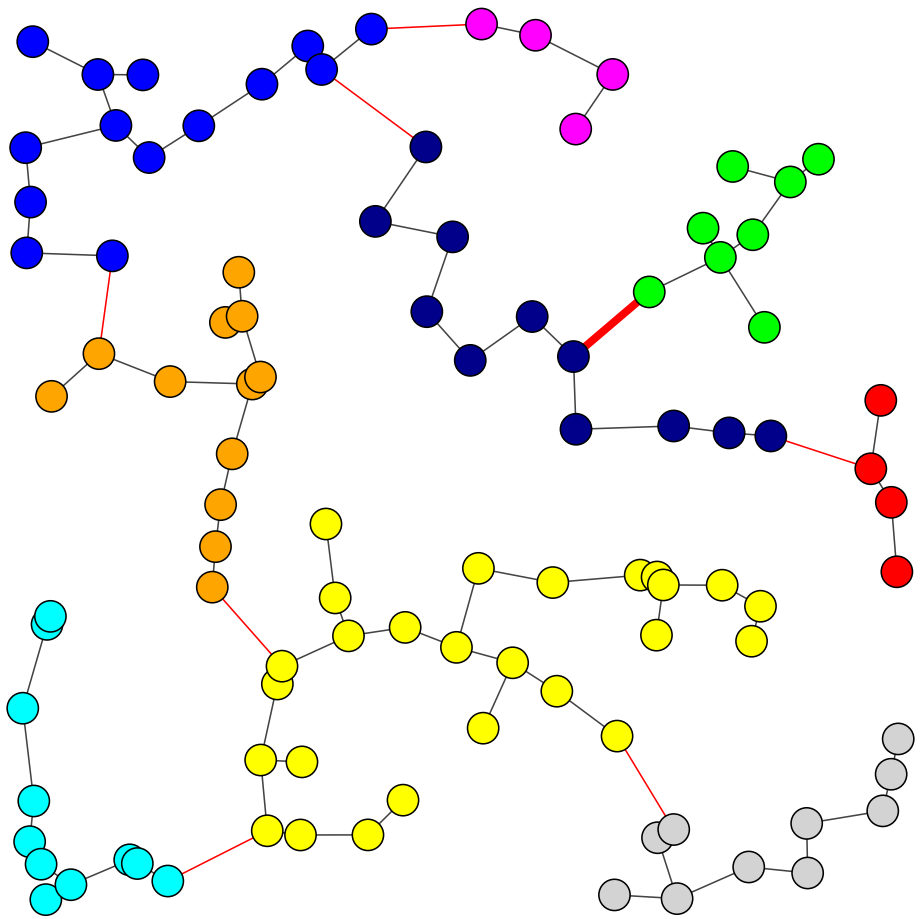
Exemple

Ainsi de suite . . .



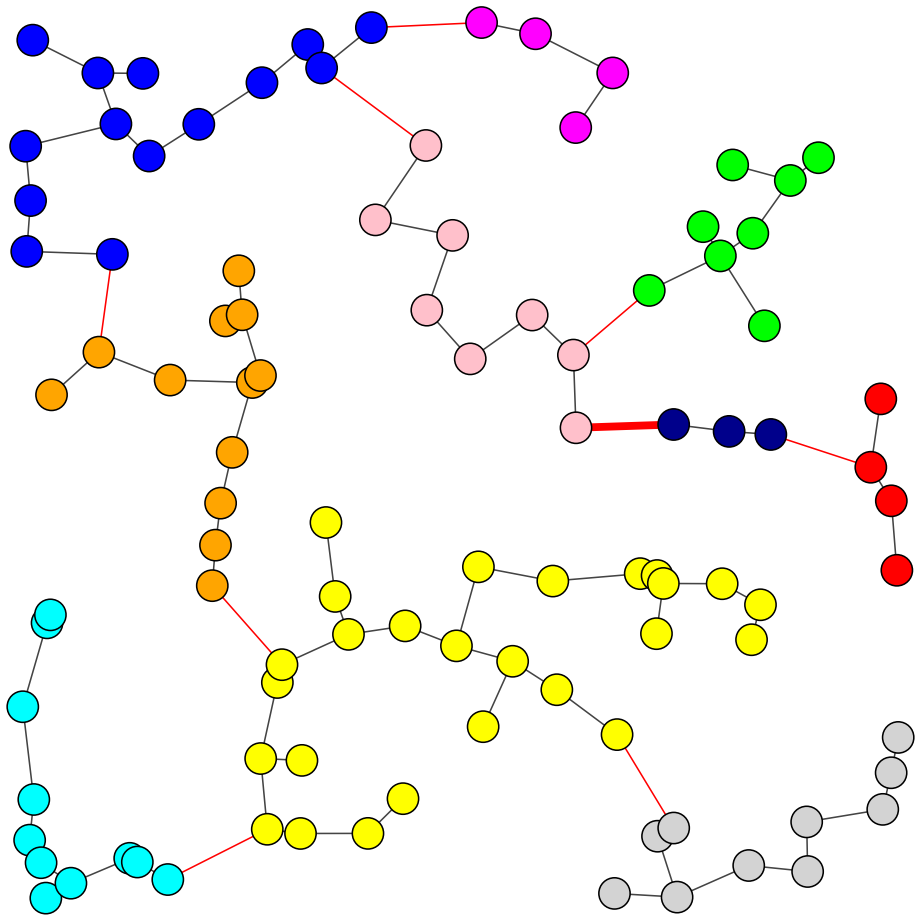
Exemple

Ainsi de suite . . .



Exemple

10 clusters avec spacing maximal !



Plan

Les points à retenir

- Il existe des algorithmes efficaces pour construire un arbre couvrant de poids minimal
  - Kruskal $\rightarrow$ ajout d'une arête de poids minimal
  - Prim $\rightarrow$ ajout du plus proche voisin à l'arbre en construction
  - ➜ Les deux algorithmes ne donnent pas la même solution...
    ...mais ce sont toutes des solutions optimales !
- Structure de données pour implémenter l'algorithme
  - ➜ Impact sur le temps de calcul (complexité en temps)
  - ➜ Structure de type union-find
- Applications à de nombreux problèmes en informatique (exemple : *clustering*)