Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 11 : Arbre couvrant
Tri topologique

Benjamin Wack

2021 – 2022

La dernière fois

- Problèmes d'optimisation
- Algorithmes gloutons
- Parcours de graphes

Aujourd'hui

- Arbre couvrant : algorithmes de Prim et Kruskal
- Graphe orienté : tri topologique (en bonus)

Plan

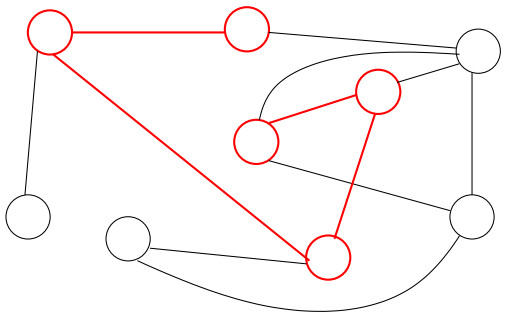
Chaînes, connexité

Chaîne

Dans un graphe non orienté, une chaîne est une suite de sommets $(x_0, x_1, \ldots x_k)$ telle que pour tout $0 < i \leq k$, (x_{i-1}, x_i) est une arête.
Les sommets x_0 et x_k sont les **extrémités** de la chaîne.

Longueur d'une chaîne

La longueur d'une chaîne est le nombre d'arêtes qui composent cette chaîne, c'est-à-dire le nombre de sommets moins un (4 dans cet exemple).



Graphe connexe

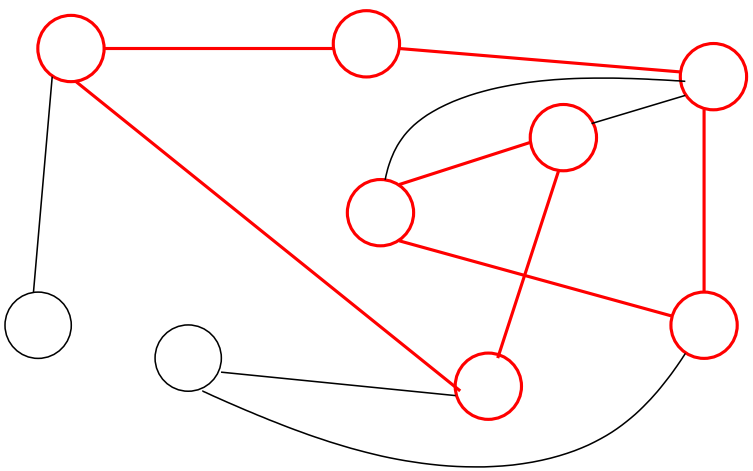
Un graphe est connexe si et seulement si, pour tout couple de sommets x et y, il existe une chaîne entre x et y.

Cycle

Cycle

Un cycle est une chaîne :

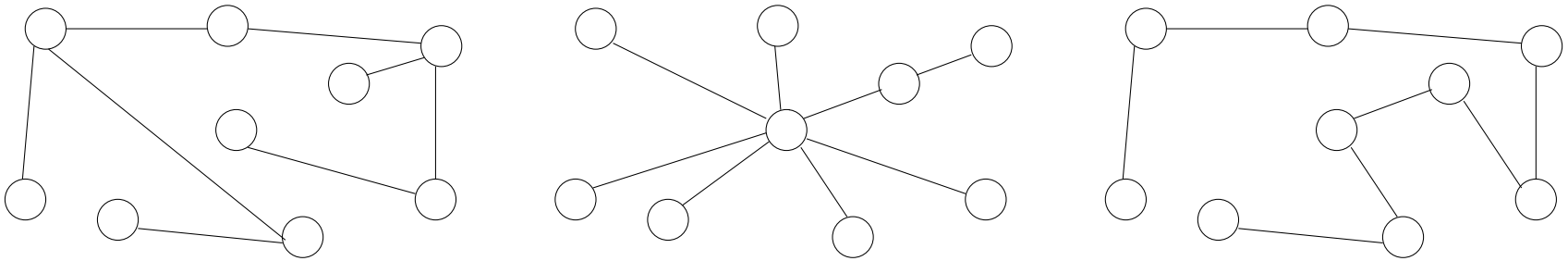
- dont les extrémités sont identiques ;
- de longueur k supérieure ou égale à 3 ;
- telle que pour tout $0 \leq i < k-1$, x_i est distinct de x_{i+2} *(sans aller-retour)*.



Arborescence

Arbre

Un arbre est un graphe connexe et sans cycle.



Attention, notion différente des arbres utilisés habituellement en algo :

- Aucune contrainte d'arité
- Pas de notion de racine

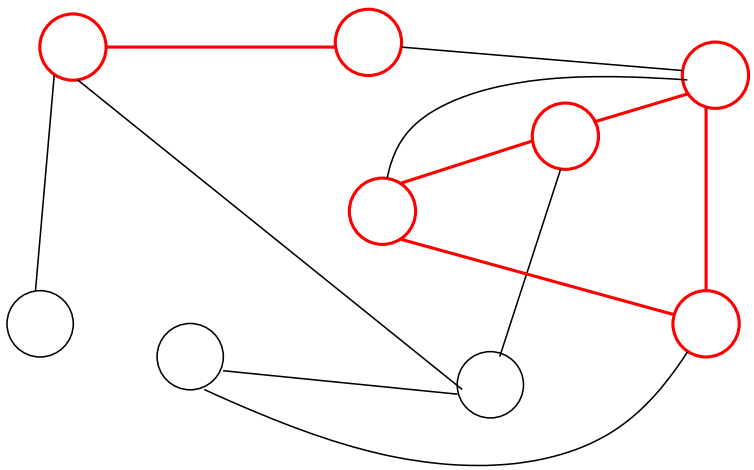
Propriétés évidentes

- Entre deux sommets donnés d'un arbre, il existe toujours exactement une chaîne (élémentaire).
- Un arbre à n sommets comporte $n - 1$ arêtes.

Sous-graphe

Sous-graphe

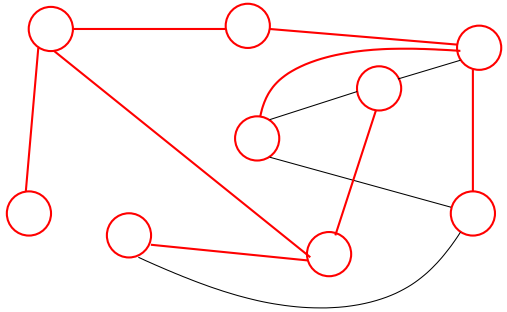
Si $G = (X, R)$, un sous-graphe de G est un graphe $H = (Y, Q)$ avec $Y \subseteq X$ et $Q \subseteq R$. Les extrémités de toutes les arêtes de Q font évidemment partie de Y.



Un sous-graphe de G est dit couvrant s'il contient tous les sommets de G. Attention, un sous-graphe couvrant n'est pas forcément connexe.

Arbre couvrant

Tout graphe connexe admet un arbre couvrant.



Si A est un sous-graphe couvrant d'un graphe G ayant V sommets, les caractérisations suivantes sont équivalentes :

- A est un arbre couvrant de G
- A est sans cycle et **possède $V-1$ arêtes**
- A est connexe et **possède $V-1$ arêtes**
- **on ne peut pas ajouter une arête à A sans créer un cycle**
- **on ne peut pas retirer une arête à A sans briser sa connexité**

Caractérisations particulièrement utiles pour l'écriture d'algorithmes :

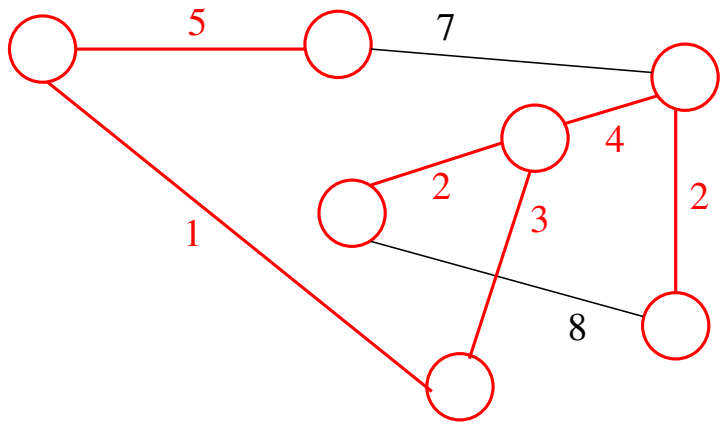
- le nombre d'arêtes est un bon critère d'arrêt ;
- l'absence de cycle (resp. la connexité) est un invariant à maintenir.

Dans un graphe pondéré

Rappel

Un **graphe aux arêtes pondérées** est un graphe muni d'une fonction de poids p : *Arêtes* $\rightarrow \mathbb{Z}$ (ou $\mathbb{R}$).

Le **poids d'un sous-graphe** est la somme des poids de ses arêtes.
On recherche alors un arbre couvrant de poids minimal (en anglais *Minimum Spanning Tree*).



Application

Problème proposé (et résolu) en 1926 par Otakar Borůvka pour la construction de réseaux électriques efficaces.

O. Borůvka (1899-1995)
(photo JanFrancuBrno)

- Les sommets de G représentent des lieux à connecter : villes, ordinateurs, composants électroniques, etc.
- Les arêtes de G représentent les liens (routes, câbles...) *possibles* entre ces lieux, avec les coûts effectifs de création de ces liens.
- L'arbre couvrant minimal est le réseau le moins coûteux ne laissant aucun lieu isolé.

Attention : on minimise le coût global du réseau, pas les longueurs des chemins dans l'arbre.
Il existe des variantes du problème contraignant la forme de l'arbre obtenu : degré borné pour chaque sommet ; diamètre borné ; etc.

Algorithmes de détermination d'un arbre couvrant

Deux idées duales :

$A := (X, \emptyset)$ (le graphe vide)
tant que $nbAretes(A) < V - 1$ **faire**
  Choisir une arête de G qui ne crée pas de cycle.
  Ajouter cette arête à A.

$A := (X, R)$ (le graphe G)
tant que $nbAretes(A) > V - 1$ **faire**
  Choisir une arête de A qui n'est pas indispensable à la connexité.
  Retirer cette arête à A.

Par construction le graphe A obtenu est un arbre couvrant (pour peu que G soit connexe).

Critère de choix

- ne pas créer de cycle : assez facile
- vérifier la connexité : moins facile (algorithme **Reverse-Delete**)

Choisir une arête

Toute arête qui ne crée pas de cycle convient.
Pour que l'arbre soit minimal, il faut aussi tenir compte des poids.

Là encore, deux politiques :

Connexité d'abord : Algorithme de Prim

On choisit l'arête de poids minimal **parmi celles incidentes à** A.
En cours d'algorithme A est un **arbre**.

Il suffit qu'une extrémité de l'arête choisie soit hors de A.

Minimalité d'abord : Algorithme de Kruskal

On choisit l'arête de poids minimal **dans tout le graphe**.
En cours d'algorithme A est une **forêt**.

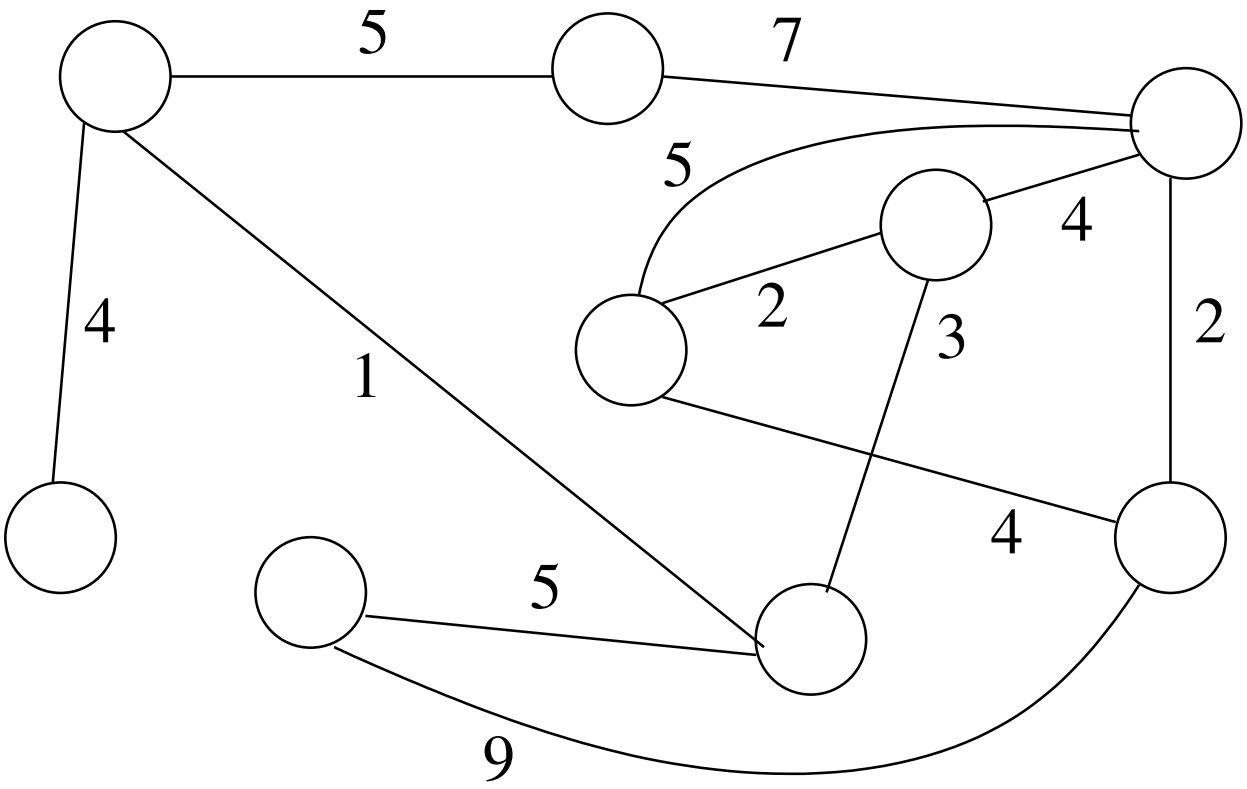
Il faut mémoriser si deux sommets sont dans des **composantes connexes** distinctes : utilisation de *Union-Find*.

Algorithmes gloutons

Algorithme de Prim

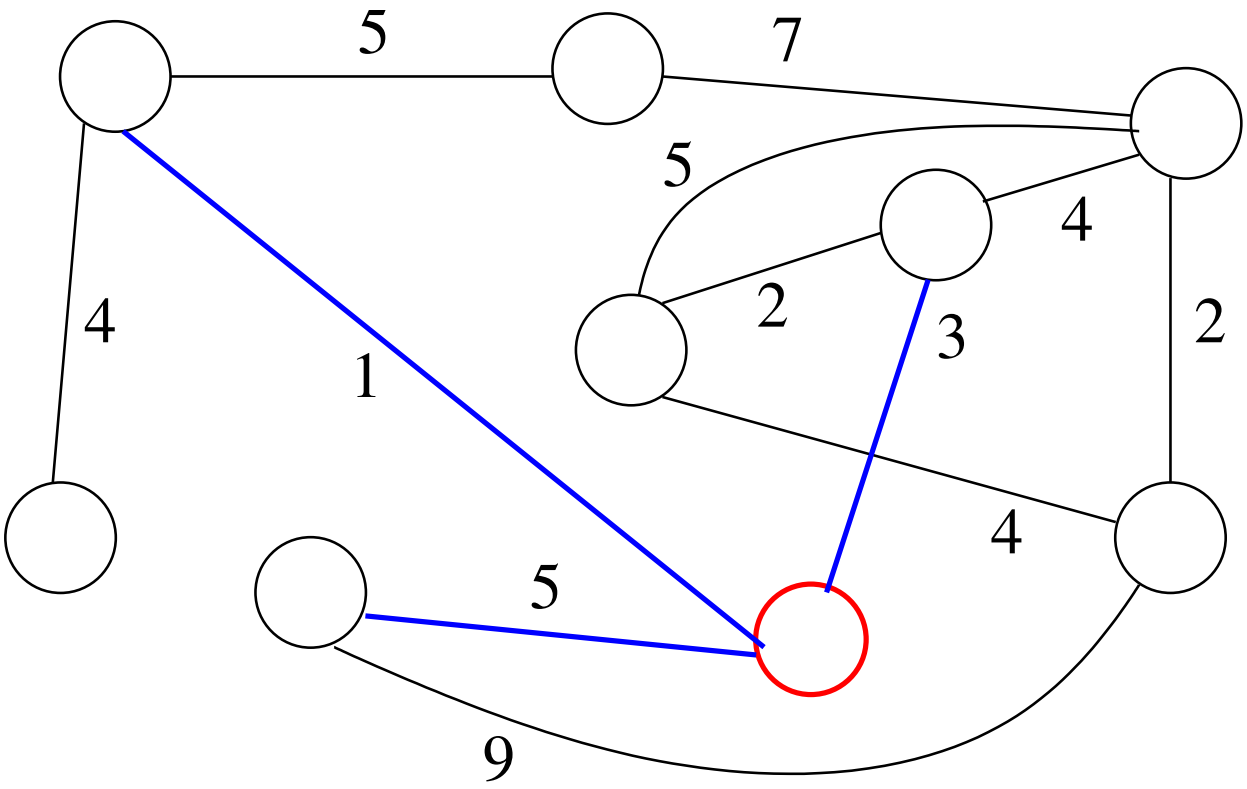
$A := (\emptyset, \emptyset)$
ajouterSommet(choisirSommet(G), A)
while *nbSommets(A)* $< V$
    // Recherche de l'arête min incidente à A
    $m := +\infty$
    foreach $x \in A$ **do**
        foreach $y \in$ *ensVoisins*(x) **do**
            if $y \notin$ *ensSommets*(A) *et* *poids*$(x, y) < m$
                $m :=$ *poids*(x, y)
                $(xmin, ymin) := (x, y)$
    ajouterSommet(*ymin*, A)
    ajouterArete($(xmin, ymin)$, A)
return A

Déroulement sur un exemple



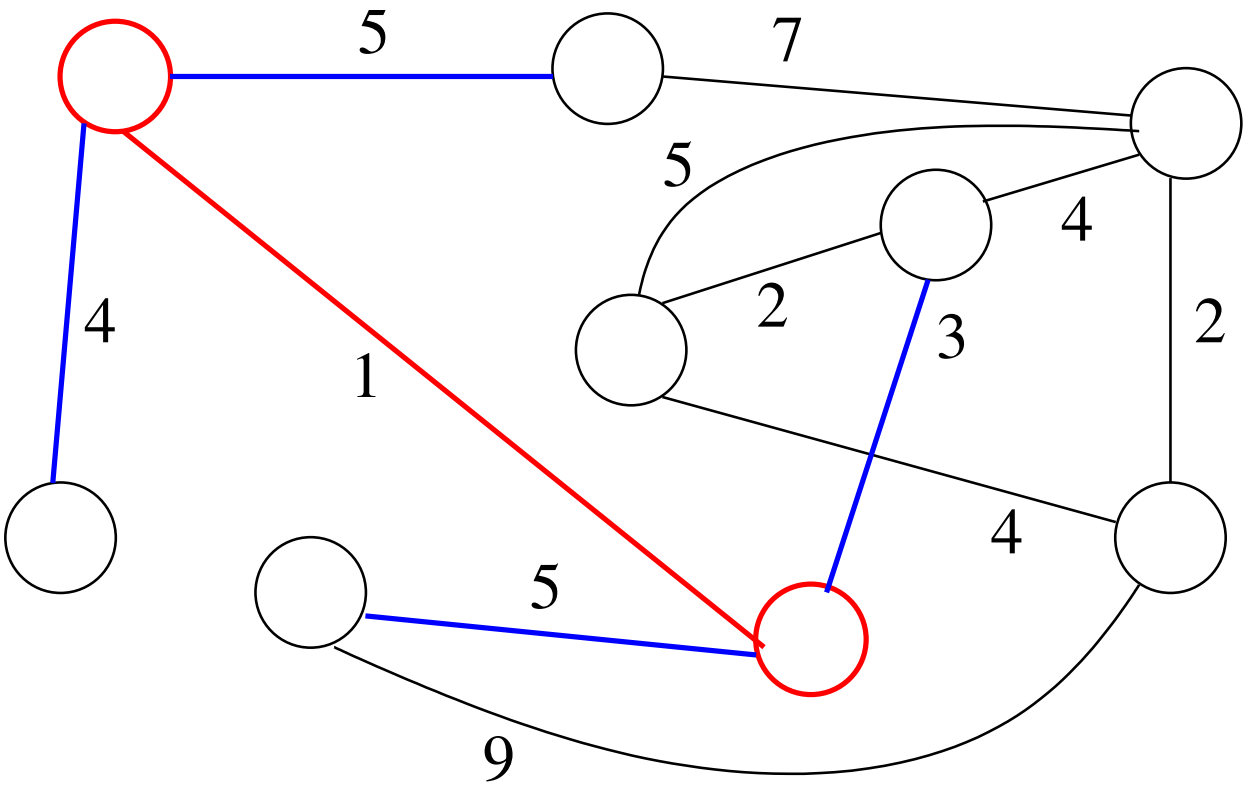
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



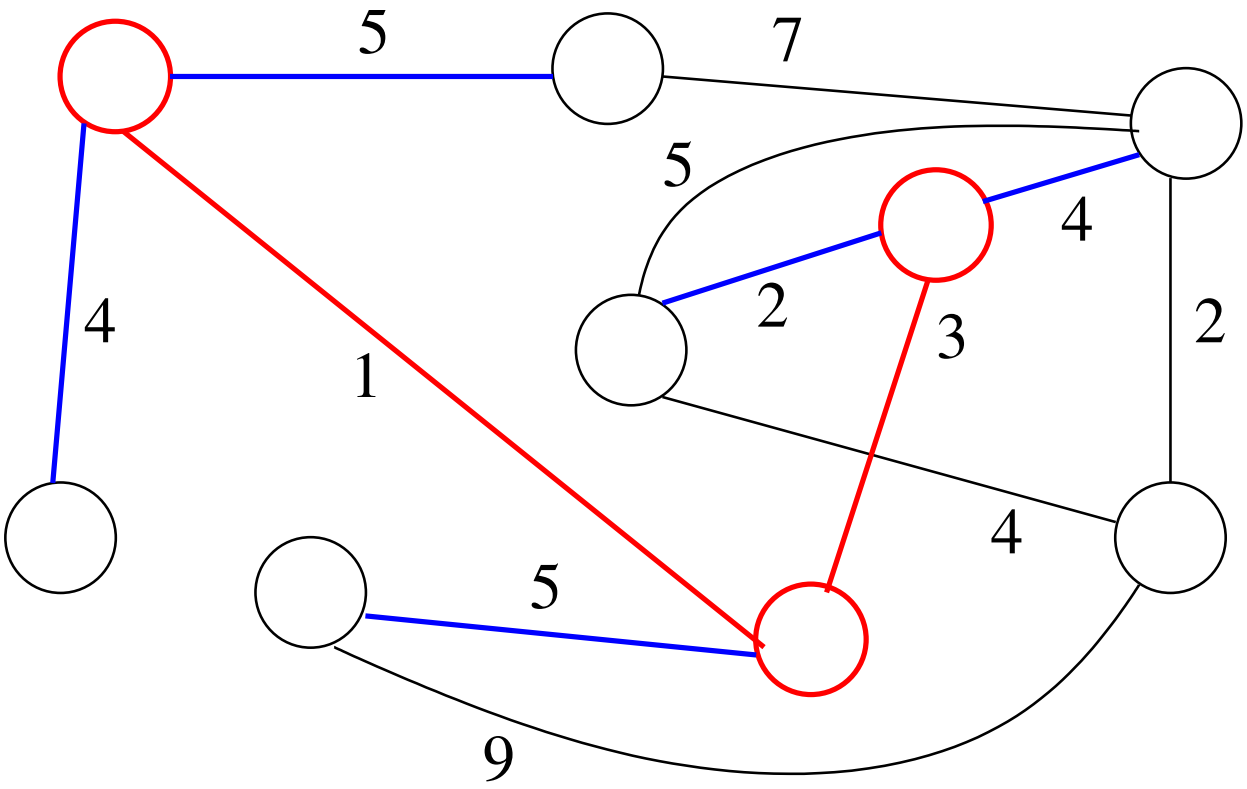
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



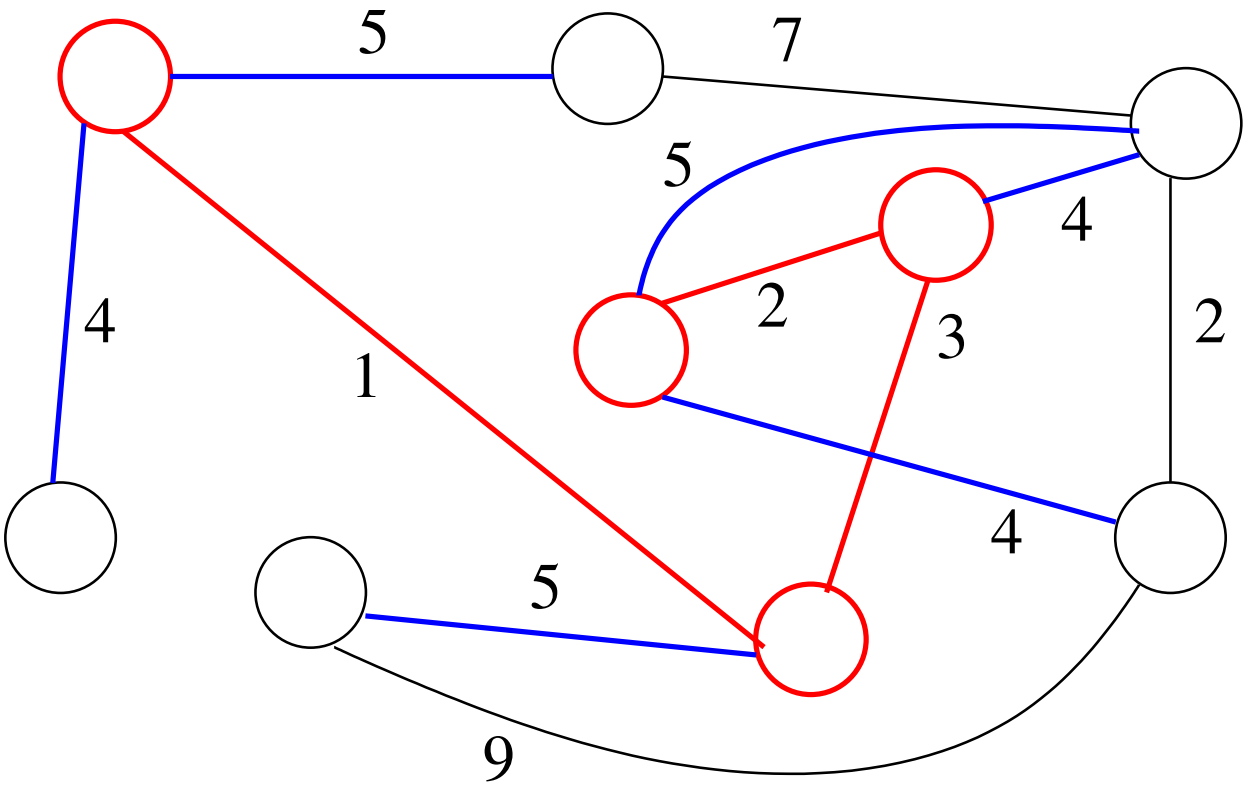
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



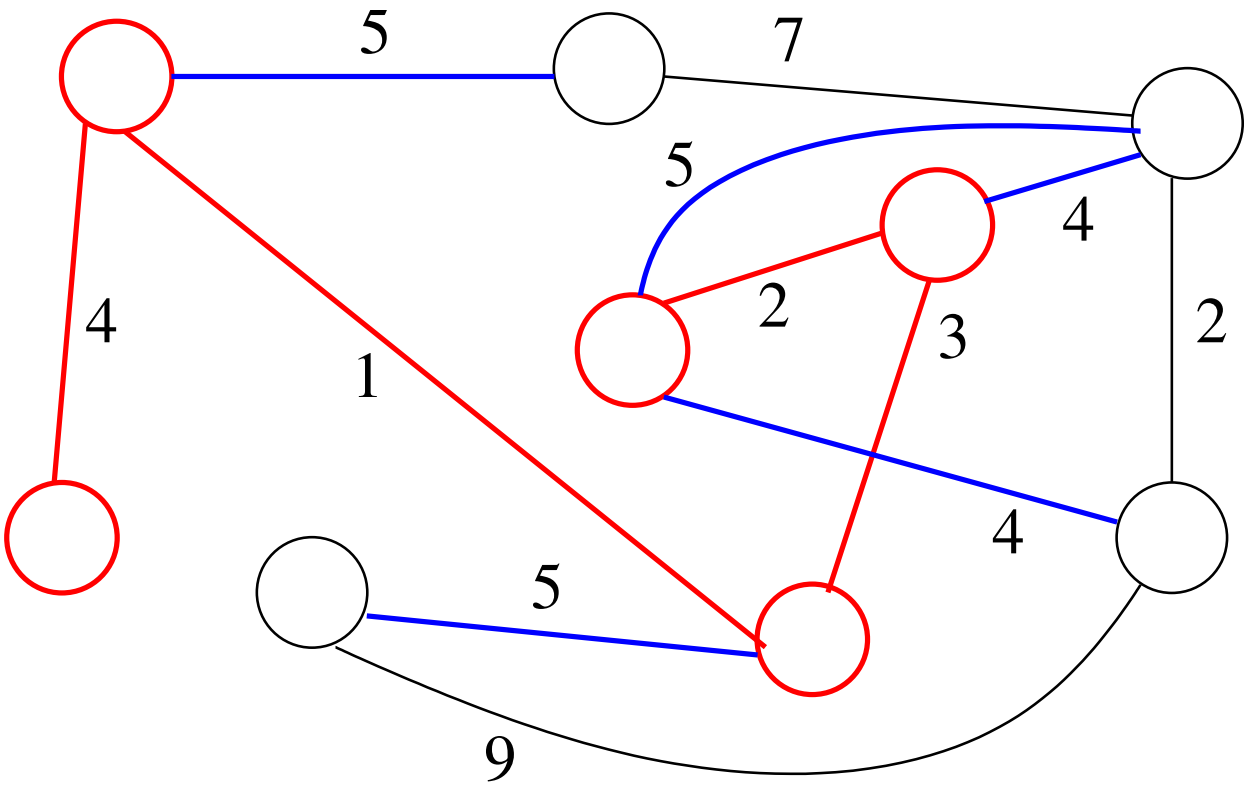
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



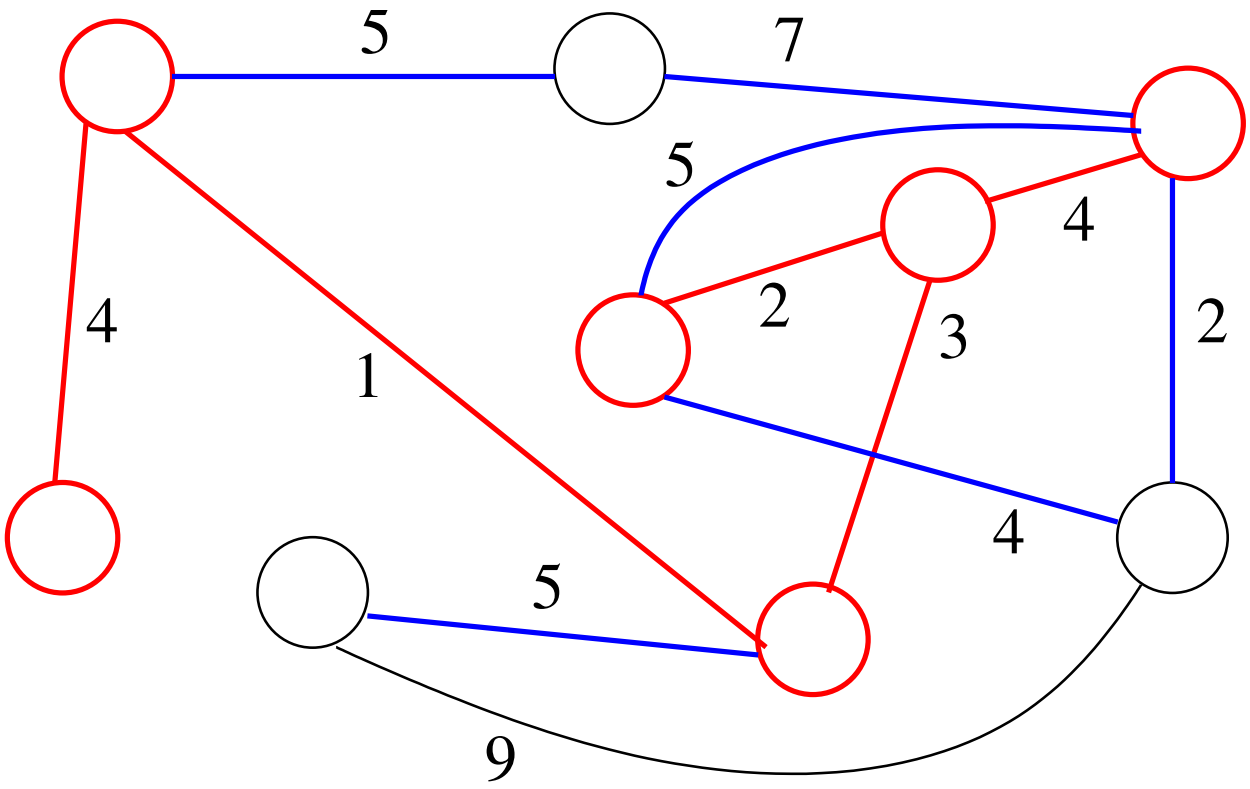
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



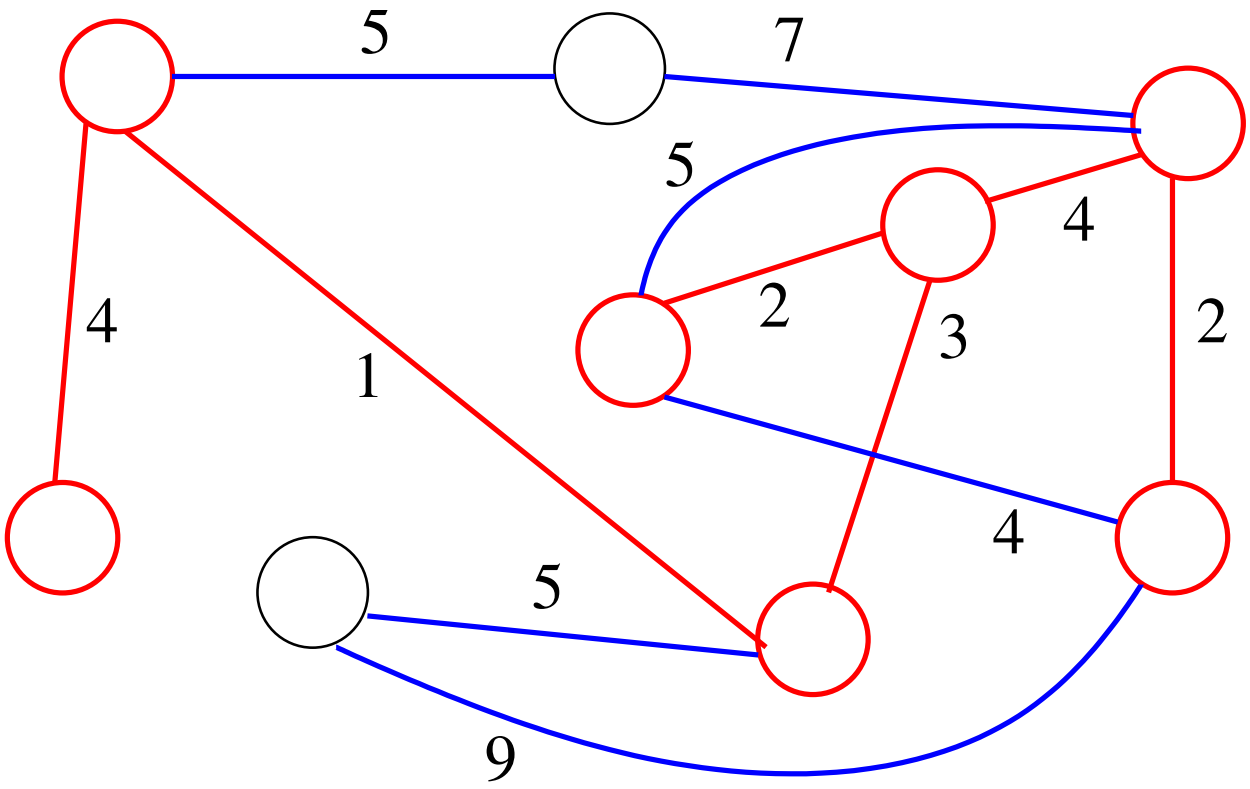
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



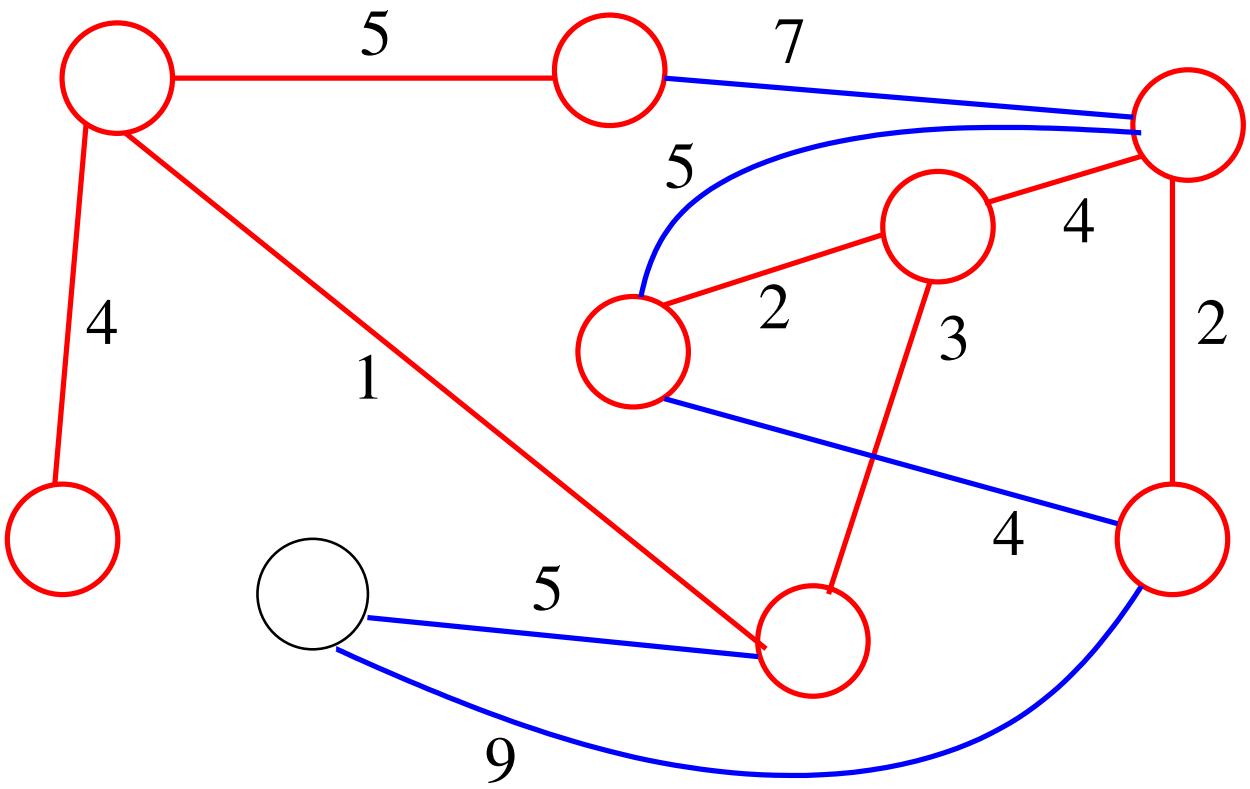
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



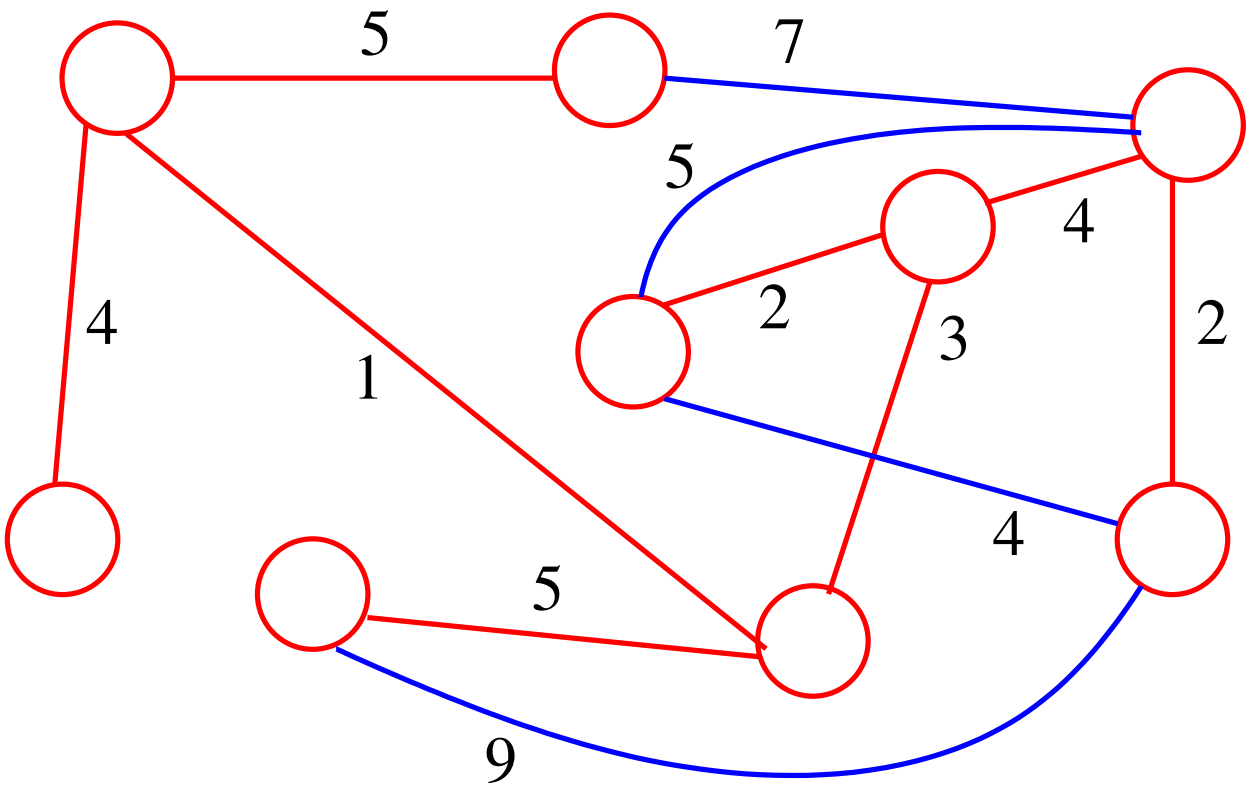
A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



A en rouge, arêtes incidentes en bleu

Déroulement sur un exemple



A en rouge, arêtes incidentes en bleu

Caractère glouton

On retrouve les caractéristiques d'un algorithme glouton :

- La solution construite est un ensemble (d'arêtes).
- Les solutions admissibles sont les arbres couvrants.
- On recherche une solution de poids total minimal.
- L'algorithme de Prim procède uniquement par ajout d'arêtes (et de sommets) dans A.
- Le choix de la prochaine arête n'est basé que sur son poids.

L'algorithme a donc un coût en $\mathcal{O}(V \times C)$ où C est le coût du choix de la prochaine arête.

Démontrer sa correction revient à prouver que A est toujours un sous-graphe d'un arbre couvrant minimal.

Complexité (naïve)

Choix de la prochaine arête

foreach $x \in A$ **do**
  foreach $y \in ensVoisins(x)$ **do**
    if $y \notin ensSommets(A)$ *et* $poids(x, y) < m$
      $m := poids(x, y)$
      $(xmin, ymin) := (x, y)$

Critère $y \notin ensSommets(A)$

Pour éviter de parcourir systématiquement les sommets de A, on maintient un tableau de booléens : mise à jour et consultation en $\mathcal{O}(1)$.
À ne pas oublier cependant : l'initialisation de ce tableau est en $\mathcal{O}(V)$.

Parcours des voisins des sommets de A

Même si la représentation choisie est efficace, coût en $\mathcal{O}(V)$.

D'où un coût total en $\mathcal{O}(V^2)$.

Raffinement

Observations :

- Le parcours de toutes les arêtes est inévitable.
- Une arête non retenue à une étape peut devenir l'arête minimale par la suite.

Seule marge d'amélioration : faciliter la recherche du minimum.

- On utilise un **tas** min.
- À tout moment le tas contient les **arêtes candidates** (c.-à-d. incidentes à A).
- À chaque fois qu'un sommet x est ajouté à A on insère dans le tas toutes les arêtes (x, z) telles que $z \notin A$.
- Une arête est supprimée du tas :
  - lorsqu'on l'ajoute à A
  - ou si on s'aperçoit qu'elle relie deux sommets de A.

Algorithme amélioré

$A := (\emptyset, \emptyset)$
$T :=$ TasVide()
$x :=$ choisirSommet(G)
ajouterSommet(x, A)
foreach $y \in ensVoisins(x)$ **do**
  Insérer((x, y), T)
while $nbSommets(A) < V$
  $(x, y) :=$ ExtraireMin(T)
  if $x \notin ensSommets(A)$ `// ou y respectivement...`
    ajouterArete((x, y), A)
    ajouterSommet(x, A)
    foreach $z \in ensVoisins(x)$ **do**
      if $z \notin A$
        Insérer((x, z), T)
return A

Complexité améliorée

Attention

La boucle **while** peut être exécutée plus de n fois (car elle n'ajoute pas toujours l'arête (x, y) à A).

En revanche :

- Chaque arête de G est insérée dans T exactement une fois (lorsque sa première extrémité est ajoutée à A).
- Chaque arête de G est extraite de T au plus une fois.
- La taille de T est bornée par E.

L'insertion et la suppression dans un tas à k éléments a un coût $\mathcal{O}(\log k)$, d'où un coût total pour cet algorithme en $\mathcal{O}(E \log E)$.

Par ailleurs $E \leq V^2$, et $\log V^2 = 2 \log V$. Il est donc également correct d'écrire que l'algorithme de Prim a une complexité en $\mathcal{O}(E \log V)$.

Encore mieux ?

Il est possible de maintenir plutôt un tas de **sommets candidats**, ordonnés par le poids de l'arête qui les relie à A :

- gain de mémoire : tas de taille V
- gain de temps : insertion/extraction en $\log V$
  (On rappelle cependant que $\log E \leq 2 \log V$.)
- seulement V tours de boucle
- mais mise à jour des poids dans le tas au fur et à mesure que A s'agrandit, potentiellement une fois pour chaque arête

Au total on reste en $\mathcal{O}((E + V) \log V) = \mathcal{O}(E \log V)$.

Enfin il existe des implantations du tas encore plus efficaces que le tableau qui permettent d'atteindre $\mathcal{O}(E + V \log V)$.

Correction

À la fin de l'algorithme :

- A comporte $V - 1$ arêtes et V sommets ;
- A ne contient aucun cycle (invariant).

C'est donc un arbre couvrant pour G.

Pour démontrer son optimalité, on ajoute à l'invariant la propriété (usuelle pour les algorithmes gloutons) :

L'arbre A est un sous-arbre d'un arbre couvrant minimal.

Initialisation : le graphe vide (ou réduit à un sommet) est un sous-arbre de tout arbre couvrant.
Propagation : Supposons qu'en un point de l'algorithme, A soit un sous-arbre d'un arbre couvrant minimal T. Lorsqu'on ajoute une arête (x, y) à A :

- Si $(x, y) \in T$ alors trivialement $A \cup \{(x, y)\} \subseteq T$.

Correction (suite)

Sinon, on considère la chaîne qui relie x et y dans T.

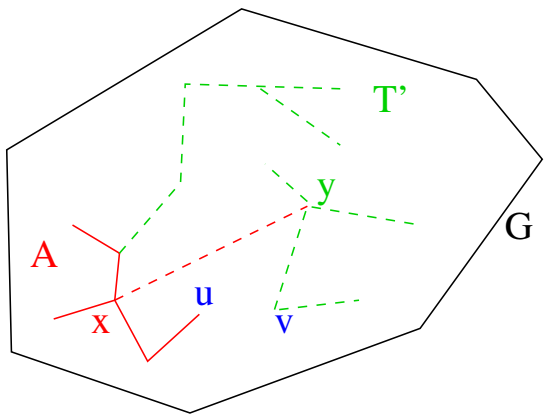
Puisque $x \in A$ et $y \notin A$, il existe au moins une arête (u, v) dans T telle que $u \in A$ et $v \notin A$.
Puisque l'algorithme choisit toujours l'arête candidate minimale, $poids(x, y) \leq poids(u, v)$.

Alors $T' = T \backslash \{(u, v)\} \cup \{(x, y)\}$ est :

- un arbre couvrant (il hérite de la connexité de T et possède $n - 1$ arêtes)
- de poids minimal :

$poids(T) - poids(u, v) + poids(x, y) \leq poids(T)$

- contenant $A \cup \{(x, y)\}$.



L'invariant est donc propagé après ajout d'une arête à A.

En fin d'algorithme A est inclus dans un arbre couvrant minimal et il comporte $V - 1$ arêtes : c'est un arbre couvrant minimal.

Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find(x)* $\neq$ *Find(y)*
        ajouterArete((x, y), *Sol*)
        Union(x, y)

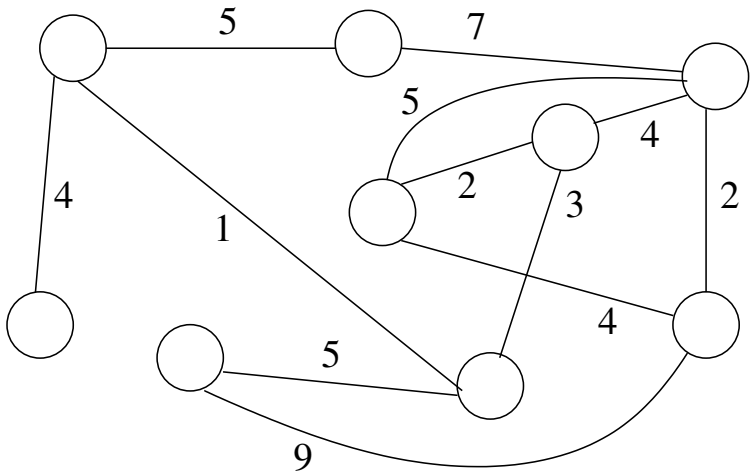


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

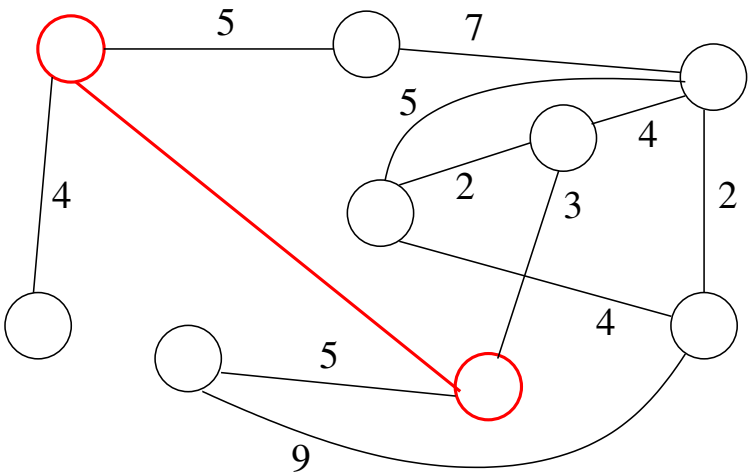


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

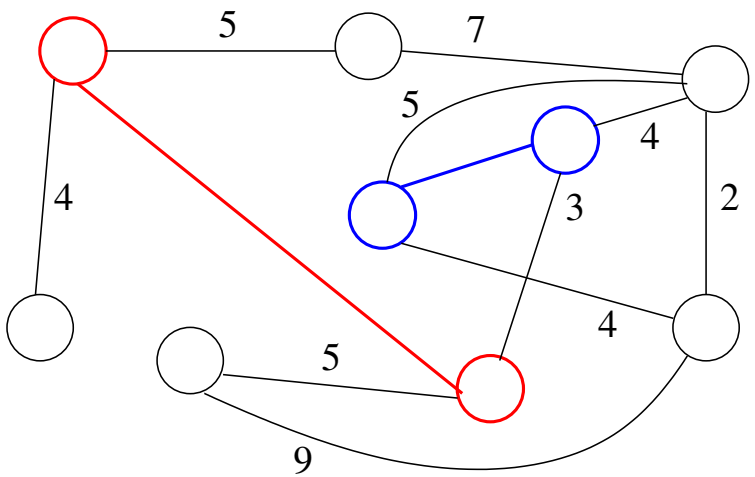


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find(x)* $\neq$ *Find(y)*
        ajouterArete((x, y), *Sol*)
        Union(x, y)

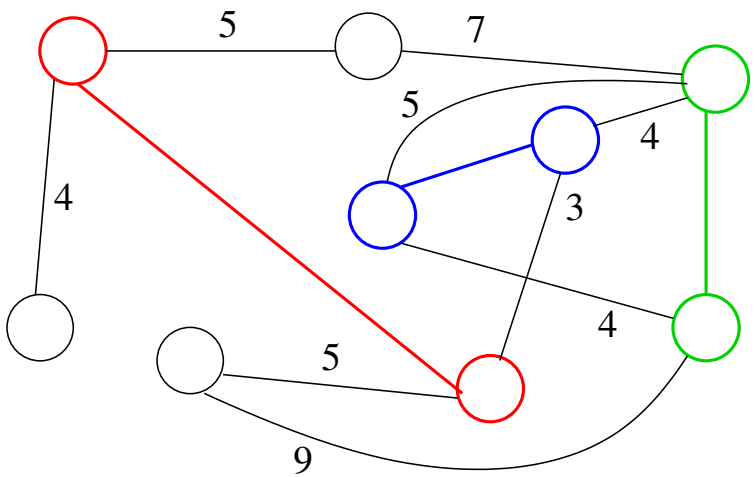


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

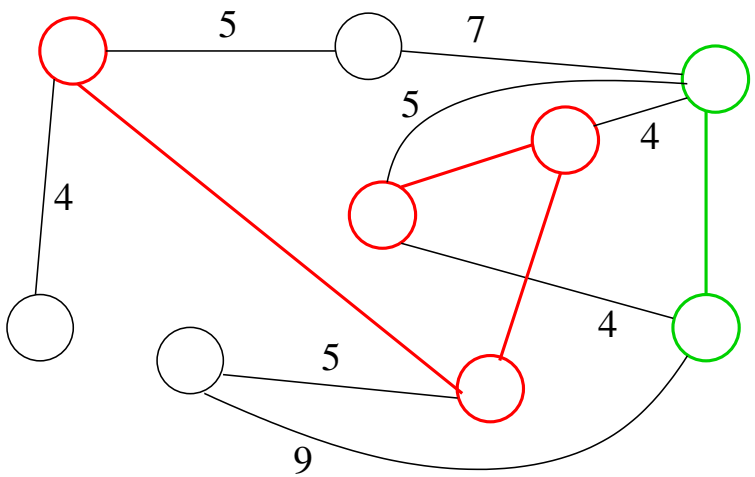


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

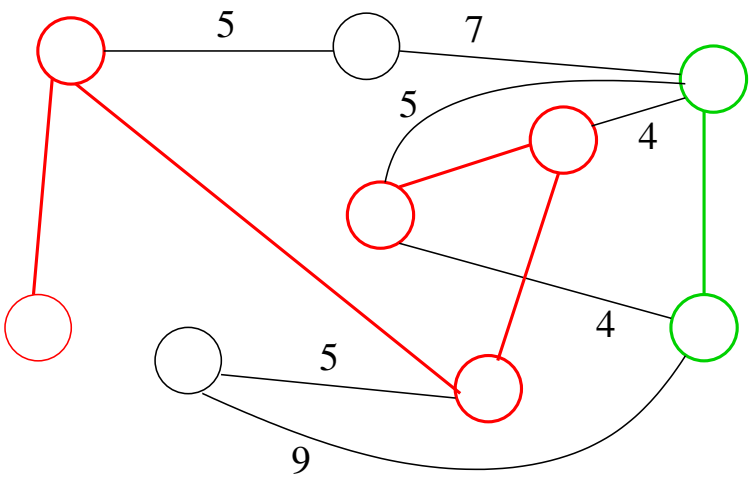


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

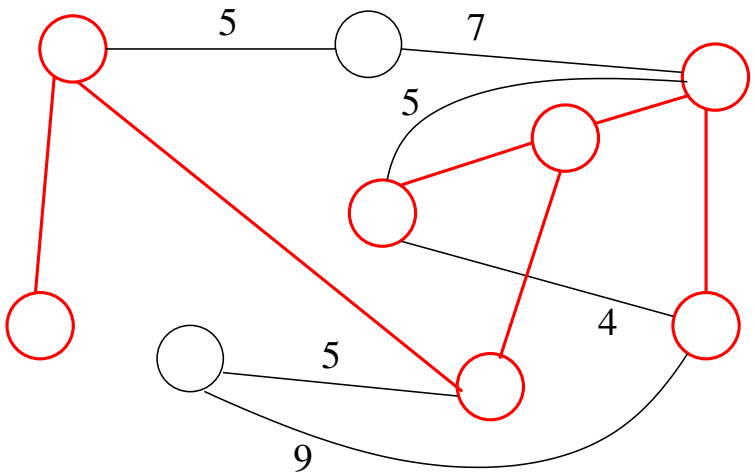


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

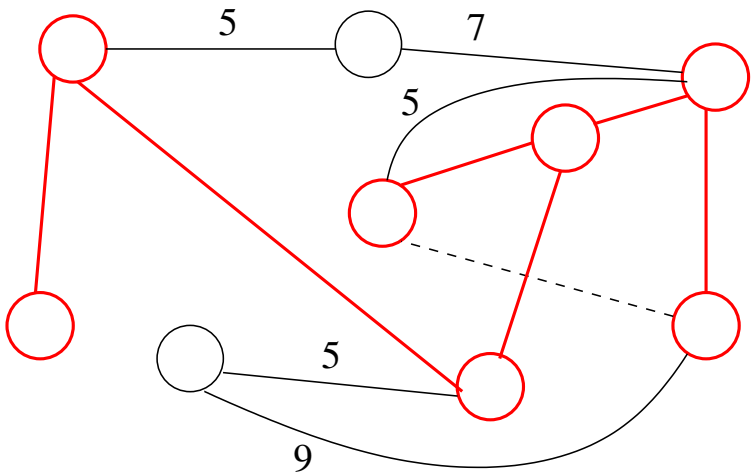


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

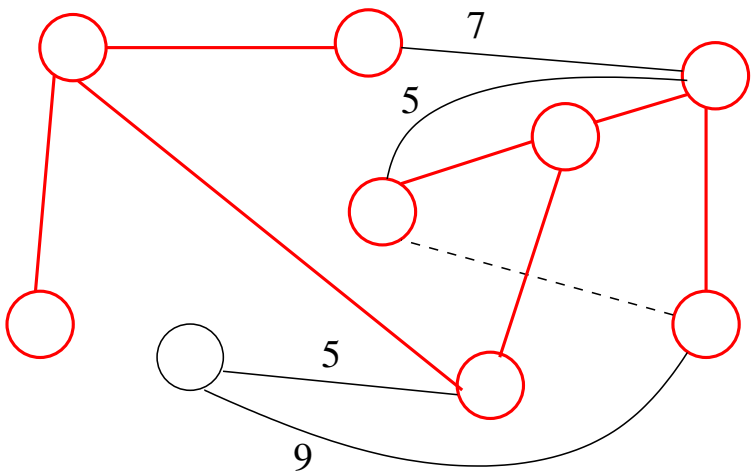


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)

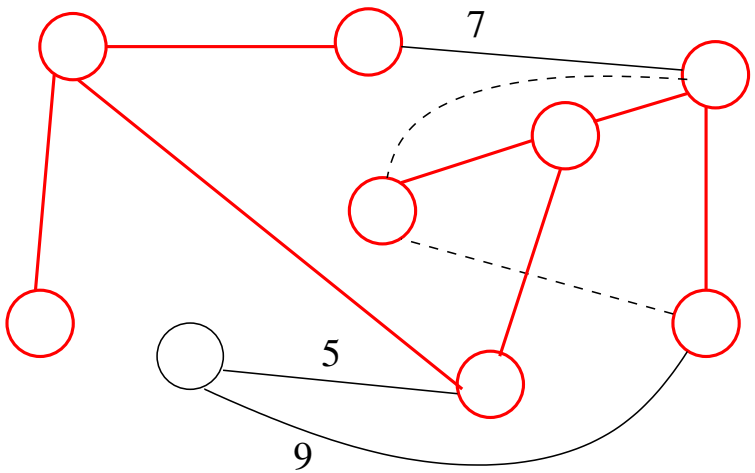
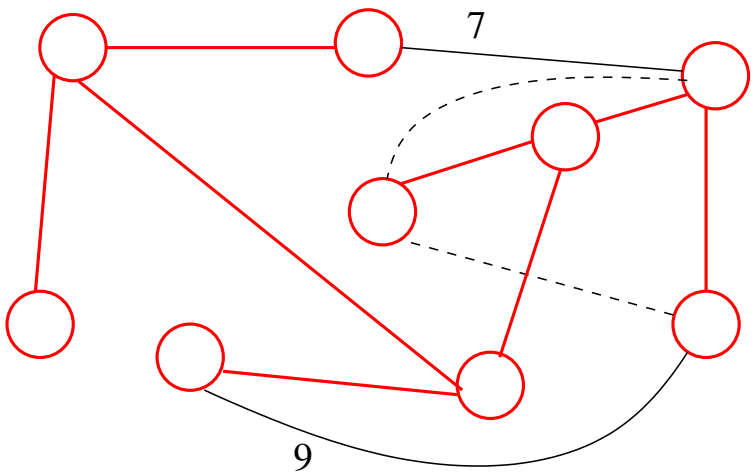


Schéma de l'algorithme de Kruskal

Trier les arêtes de G par poids croissant.
Initialiser le graphe *Sol* à $(X, \emptyset)$.
Initialiser une structure de *Union-Find*.

foreach *arête* (x, y) *(par poids croissant)* **do**
    if *Find*$(x) \neq$ *Find*(y)
        ajouterArete((x, y), *Sol*)
        Union(x, y)



Analyse de coût

En supposant pour simplifier que les coûts de Find et de Union sont constants :

- tri des arêtes : $\mathcal{O}(E \log E)$
- initialisation de *Union-Find* : $\mathcal{O}(V)$
- boucle de construction de l'arbre : $\mathcal{O}(E)$

Donc un coût total en $\mathcal{O}(E \log E)$.

Rappel : avec l'algorithme de Prim on peut espérer une complexité en $\mathcal{O}(E + V \log V)$.
D'où le choix suivant :

- Si le graphe est dense, $E \simeq V^2$: on choisit Prim
- Si le graphe est creux, $E \simeq V$: pas de différence...
  mais on prend plutôt Kruskal !
  (structure de données simple / si les arêtes sont faciles à trier...)

Bilan du semestre

- Complexité d'un algorithme (au pire, au mieux, en moyenne)
- Preuve de correction, de terminaison
- Structures de données, type abstrait
- Structures usuelles : pile, file, FAP, arbre, graphe
- Arbre de recherche, arbre de codage, tas, arbre couvrant...
- Schémas algorithmiques : diviser pour régner, glouton

Graphe orienté

Définition

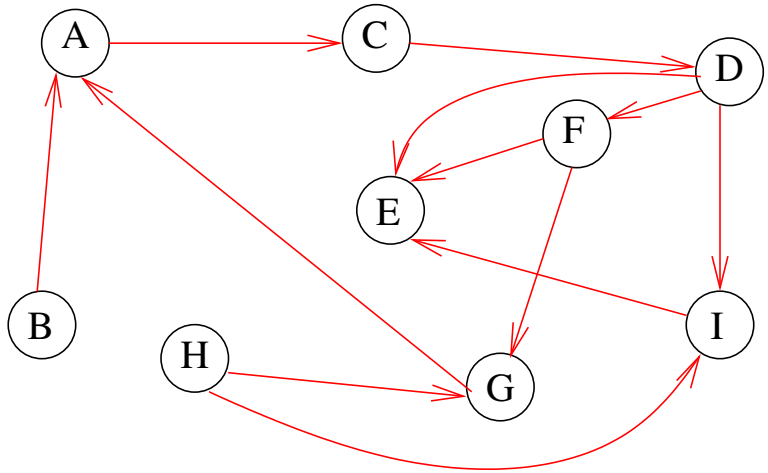
Un graphe orienté est un couple $G = (X, R)$:

- X est l'ensemble des sommets
- R est l'ensemble des arcs : c'est une relation binaire sur X (ensemble de couples de X).

Arc

Un arc est un couple de sommets (x, y).

- x est appelé origine de l'arc ;
- y est appelé extrémité.

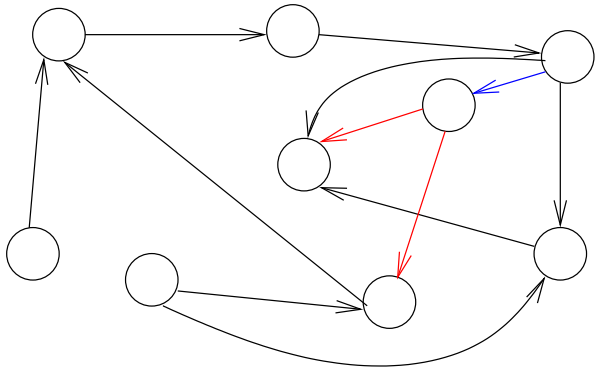


Sur l'exemple : $R = \{(B, A), (A, C), (C, D), (D, E), (D, F), (D, I), \ldots\}$

Voisinages

Successeur

On appelle successeur d'un sommet x tout sommet y tel que (x, y) est un arc du graphe.



Degré d'un sommet

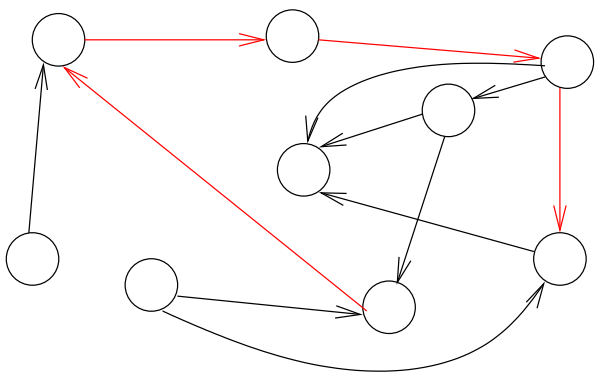
Le degré d'un sommet x est le nombre d'arcs dont x est origine ou extrémité :

- le demi-degré extérieur est le nombre d'arcs dont x est origine ;
- le demi-degré intérieur est le nombre d'arcs dont x est extrémité.

Chemins

Chemin

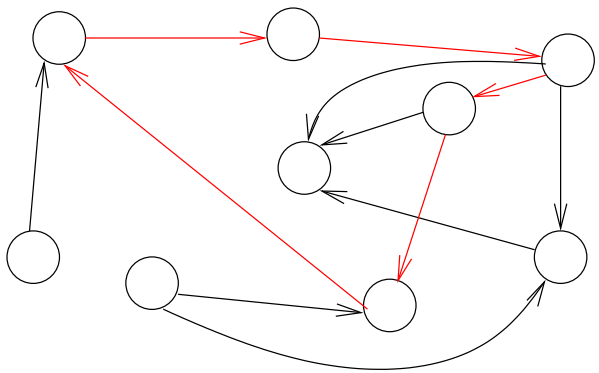
Un chemin est une suite de sommets $(x_0, x_1, \ldots x_k)$ telle que
pour tout $0 < i \leq k$, (x_{i-1}, x_i) est un arc du graphe.
x_0 est l'**origine** du chemin, et x_k est son **extrémité**.



Circuits

Circuit

Un circuit est un chemin de longueur non nulle dont l'origine est identique à l'extrémité.
En général, par commodité, on considère que tous les circuits de la forme $(x_i, x_{i+1}, \ldots x_{k-1}, x_0, x_1, \ldots x_i)$ constituent le même circuit.

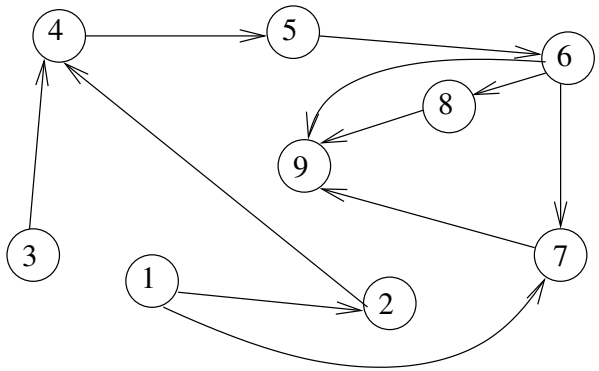


Le problème du tri topologique

Définition

Un tri topologique d'un graphe orienté $G = (X, R)$ est un **ordre total** des sommets de G tel que

pour tout arc $(x, y) \in R$ *alors* x *apparaît avant* y.



Non unicité du tri topologique

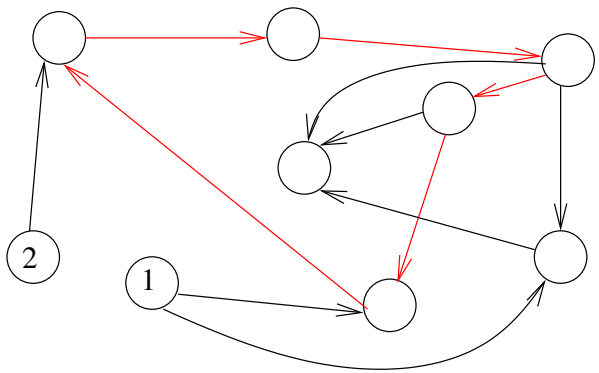
Il existe en général plusieurs ordres corrects.
Dans l'algorithme qui suit, selon la structure choisie pour l'ensemble E on obtient un ordre différent.

Le problème du tri topologique

Définition

Un tri topologique d'un graphe orienté $G = (X, R)$ est un **ordre total** des sommets de G tel que

pour tout arc $(x, y) \in R$ *alors* x *apparaît avant* y.



Corollaire : il existe un tri topologique si et seulement si le graphe ne comporte pas de circuit.

Algorithme de tri topologique

TRI_TOPOLOGIQUE (Graphe g)
Données : Un graphe g
Résultat : Les sommets de g sont numérotés selon un tri topologique
On renvoie une valeur spéciale *ko* si c'est impossible

```
E := EnsembleVide()
foreach x ∈ ensSommets(g) do
    if estVide(ensPredecesseurs(x))
        Insérer( x, E )
while non estVide(E)
    x := ExtraireElement( E )
    Numéroter( x )
    foreach y ∈ ensSuccesseurs(x) do
        supprimerArc( x, y, g )
        if estVide(ensPredecesseurs(y))
            Insérer( y, E )
if nbArcs(g) ≠ 0      // g comporte un circuit
    return ko
```

Analyse de l'algorithme de tri topologique

À propos des prédécesseurs :

- dans certaines représentations les ensembles de prédécesseurs peuvent être immédiatement disponibles ;
- il est aussi possible de précalculer le demi-degré intérieur de chaque sommet (a priori en $\mathcal{O}(E + V)$), puis de le maintenir à jour lors des suppressions d'arcs.

Aspect glouton

On reconnaît les caractéristiques d'un algorithme glouton :

- la réponse au problème est une séquence ordonnée
- choix définitifs (on ne fait qu'enfiler dans F)
- sommet à enfiler choisi sur un critère local (son demi-degré intérieur)

Dans le pire des cas :

- chaque sommet est visité : $\mathcal{O}(V)$
- chaque arc sortant de chaque sommet est supprimé : $\mathcal{O}(E)$

d'où une complexité en $\mathcal{O}(E + V)$.