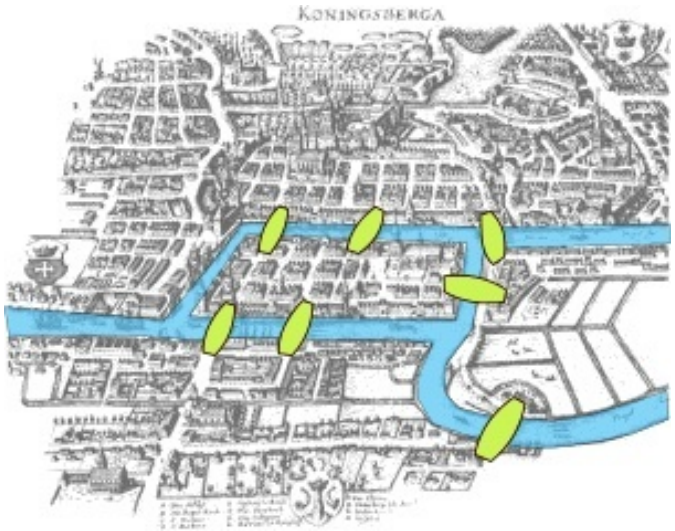
Graphes

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Graphes : des problèmes mathématiques ...

1736 : le problème des sept ponts de Königsberg

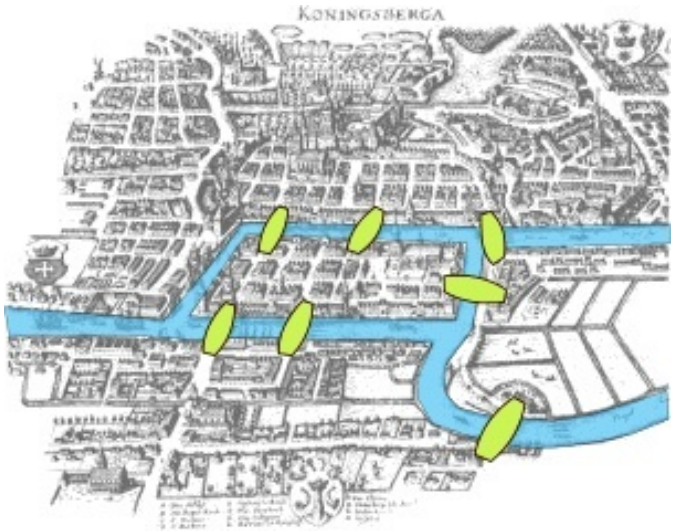
Euler : existe-t-il un chemin permettant de passer une et une seule fois par chacun des sept ponts de la ville et qui se termine là où il a commencé ?

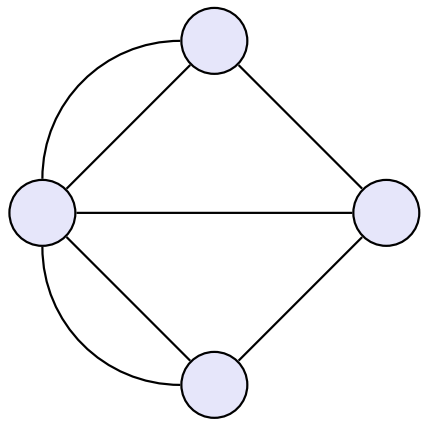


Graphes : des problèmes mathématiques ...

1736 : le problème des sept ponts de Königsberg

Euler : existe-t-il un chemin permettant de passer une et une seule fois par chacun des sept ponts de la ville et qui se termine là où il a commencé ?



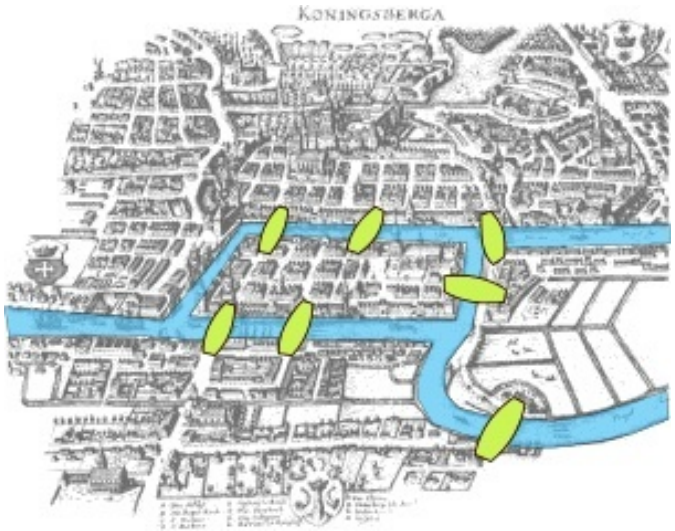


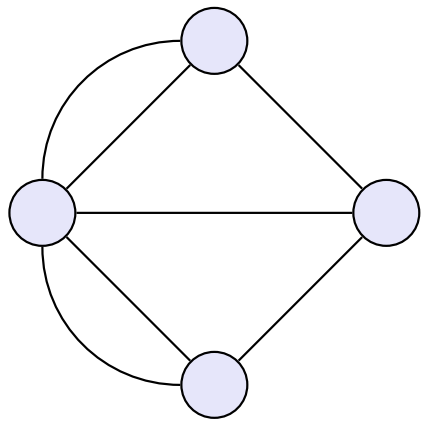
Graphe eulérien : il existe un cycle passant une et une seule fois par chacune des arêtes.

Graphes : des problèmes mathématiques ...

1736 : le problème des sept ponts de Königsberg

Euler : existe-t-il un chemin permettant de passer une et une seule fois par chacun des sept ponts de la ville et qui se termine là où il a commencé ?





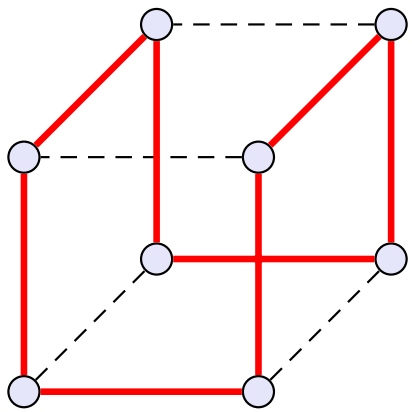
Graphe eulérien : il existe un cycle passant une et une seule fois par chacune des arêtes.

Théorème d'Euler : un graphe connexe est eulérien si et seulement si de chaque sommet ne part d'un nombre pair d'arêtes.

Graphes : des problèmes mathématiques ...

XIX^{e} siècle : graphes hamiltoniens

HAMILTON : existe-t-il un chemin passant une et une seule fois par chacun des sommets d'un dodécahèdre avant de revenir à son point de départ ?

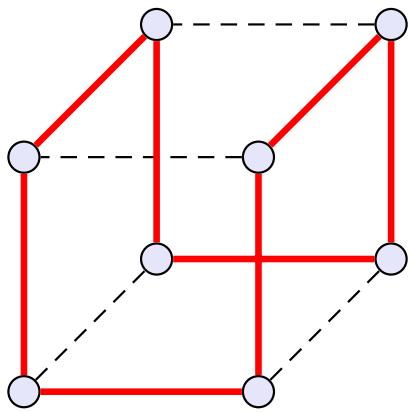


C'est le cas de tous les solides platoniciens — polyèdres réguliers convexes — (ici le cube).

Graphes : des problèmes mathématiques ...

XIX^{e} siècle : graphes hamiltoniens

HAMILTON : existe-t-il un chemin passant une et une seule fois par chacun des sommets d'un dodécahèdre avant de revenir à son point de départ ?



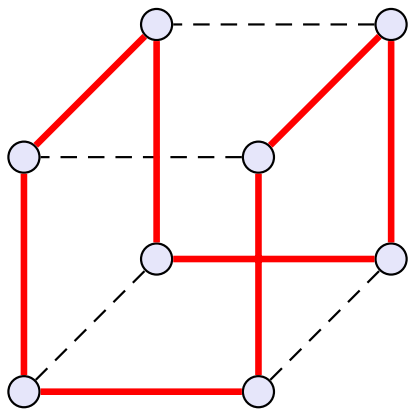
C'est le cas de tous les solides platoniciens — polyèdres réguliers convexes — (ici le cube).

Graphe hamiltonien : il existe un cycle passant une et une seule fois par chacun des sommets.

Graphes : des problèmes mathématiques ...

XIX^{e} siècle : graphes hamiltoniens

HAMILTON : existe-t-il un chemin passant une et une seule fois par chacun des sommets d'un dodécahèdre avant de revenir à son point de départ ?



C'est le cas de tous les solides platoniciens — polyèdres réguliers convexes — (ici le cube).

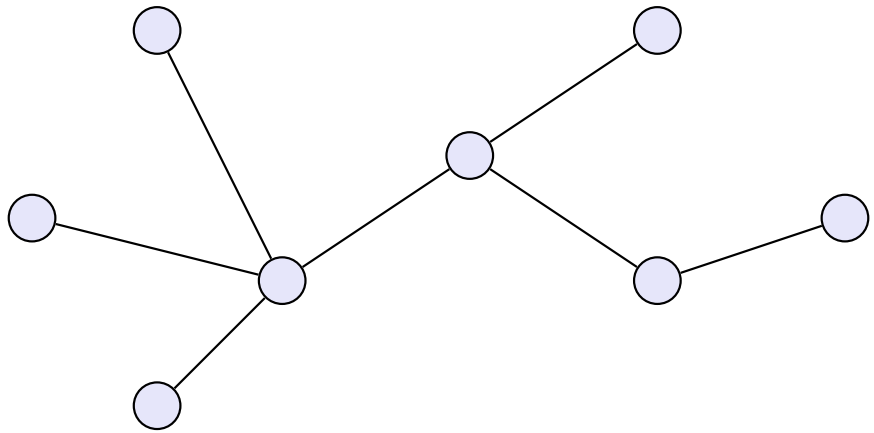
Graphe hamiltonien : il existe un cycle passant une et une seule fois par chacun des sommets.

Il existe des conditions suffisantes pour assurer qu'un graphe est hamiltonien (ou ne l'est pas) mais pas de caractérisation efficace.

Graphes : des problèmes mathématiques ...

XIX^{e} siècle : arbres

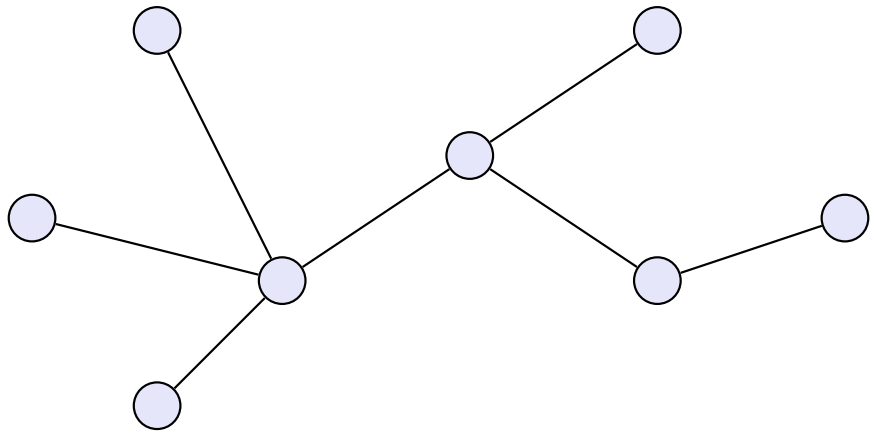
CAYLEY : étude des graphes connexes et acycliques (les arbres).



Graphes : des problèmes mathématiques ...

XIX^{e} siècle : arbres

CAYLEY : étude des graphes connexes et acycliques (les arbres).

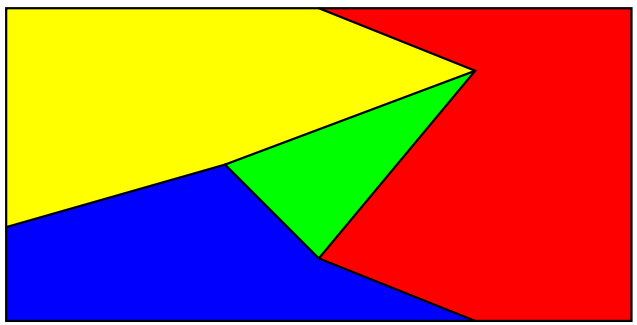


Formule de Cayley : si $n \geqslant 2$, on peut construire exactement n^{n-2} arbres distincts à partir de n sommets.

Graphes : des problèmes mathématiques ...

XX^{e} siècle : le problème des quatre couleurs

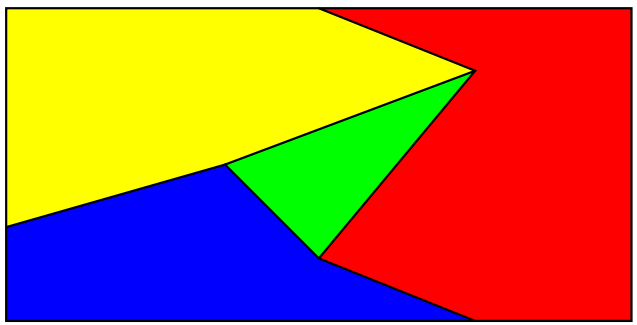
Guthrie (1852) : peut-on, en n'utilisant que quatre couleurs, colorer n'importe quelle carte découpée en régions connexes, de sorte que deux régions adjacentes reçoivent deux couleurs distinctes ?

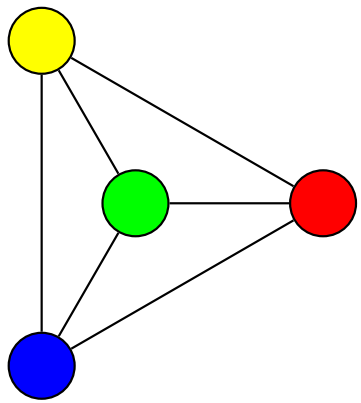


Graphes : des problèmes mathématiques ...

XX^{e} siècle : le problème des quatre couleurs

Guthrie (1852) : peut-on, en n'utilisant que quatre couleurs, colorer n'importe quelle carte découpée en régions connexes, de sorte que deux régions adjacentes reçoivent deux couleurs distinctes ?



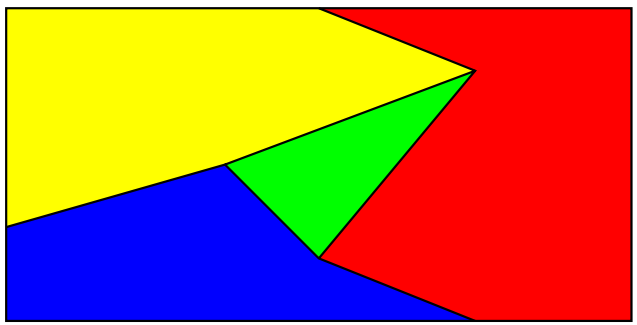


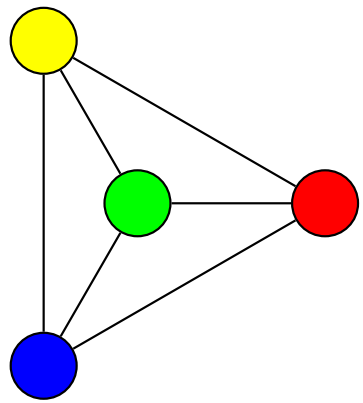
Théorème d'Appel et Haken (1976) : tout graphe planaire peut être colorié avec quatre couleurs au plus.

Graphes : des problèmes mathématiques ...

XX^{e} siècle : le problème des quatre couleurs

Guthrie (1852) : peut-on, en n'utilisant que quatre couleurs, colorer n'importe quelle carte découpée en régions connexes, de sorte que deux régions adjacentes reçoivent deux couleurs distinctes ?





Théorème d'Appel et Haken (1976) : tout graphe planaire peut être colorié avec quatre couleurs au plus.

Pour la première fois, la preuve nécessite l'usage d'un ordinateur pour étudier les cas critiques.
À l'heure actuelle il n'existe pas de preuve ne faisant pas appel à un ordinateur.

Graphes : . . . mais aussi informatiques

Parcours d'un graphe

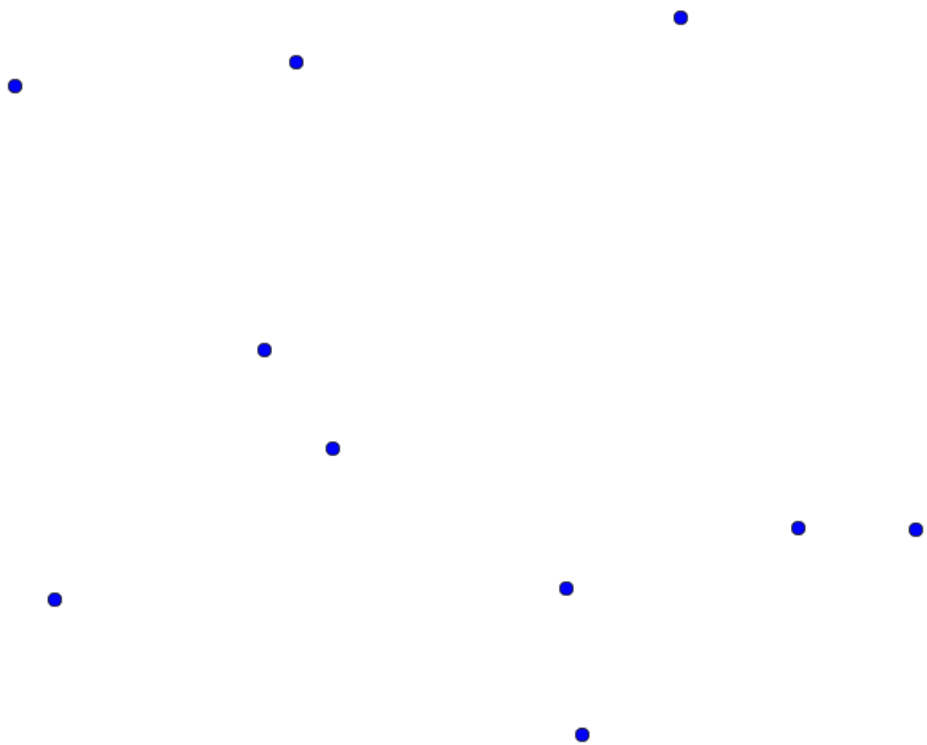
Quel algorithme utiliser pour parcourir tous les sommets d'un graphes ?
$\longrightarrow$ parcours en largeur ou en profondeur.

Graphes : . . . mais aussi informatiques

Parcours d'un graphe

Quel algorithme utiliser pour parcourir tous les sommets d'un graphes ?
$\longrightarrow$ parcours en largeur ou en profondeur.

Problème du voyageur de commerce : recherche d'un cycle hamiltonien de poids minimal dans un graphe complet.

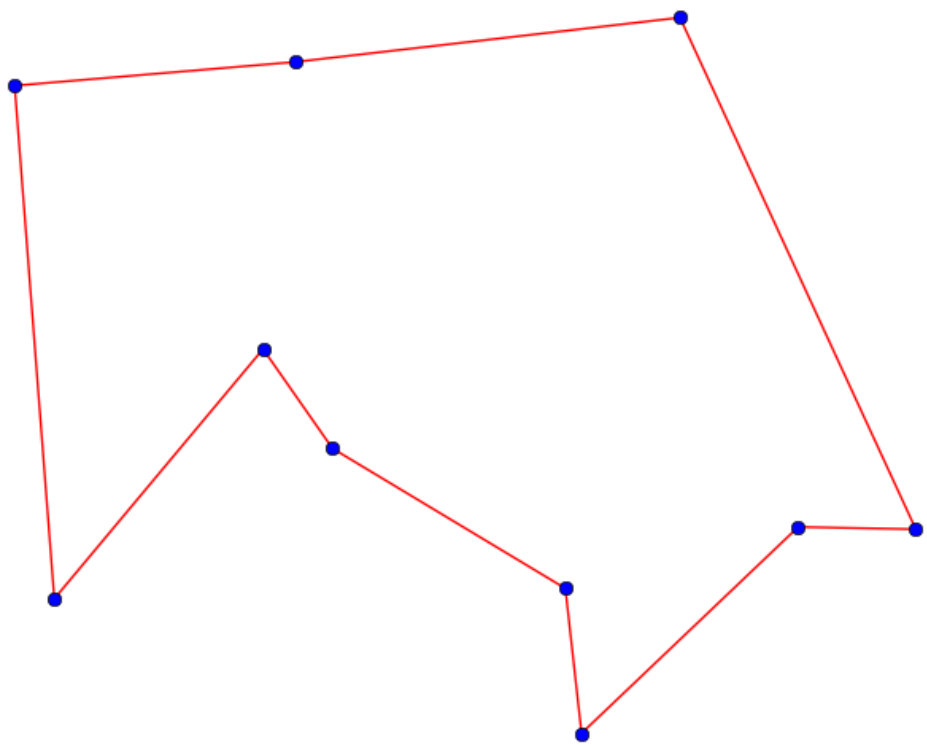


Graphes : ... mais aussi informatiques

Parcours d'un graphe

Quel algorithme utiliser pour parcourir tous les sommets d'un graphes ?
$\longrightarrow$ parcours en largeur ou en profondeur.

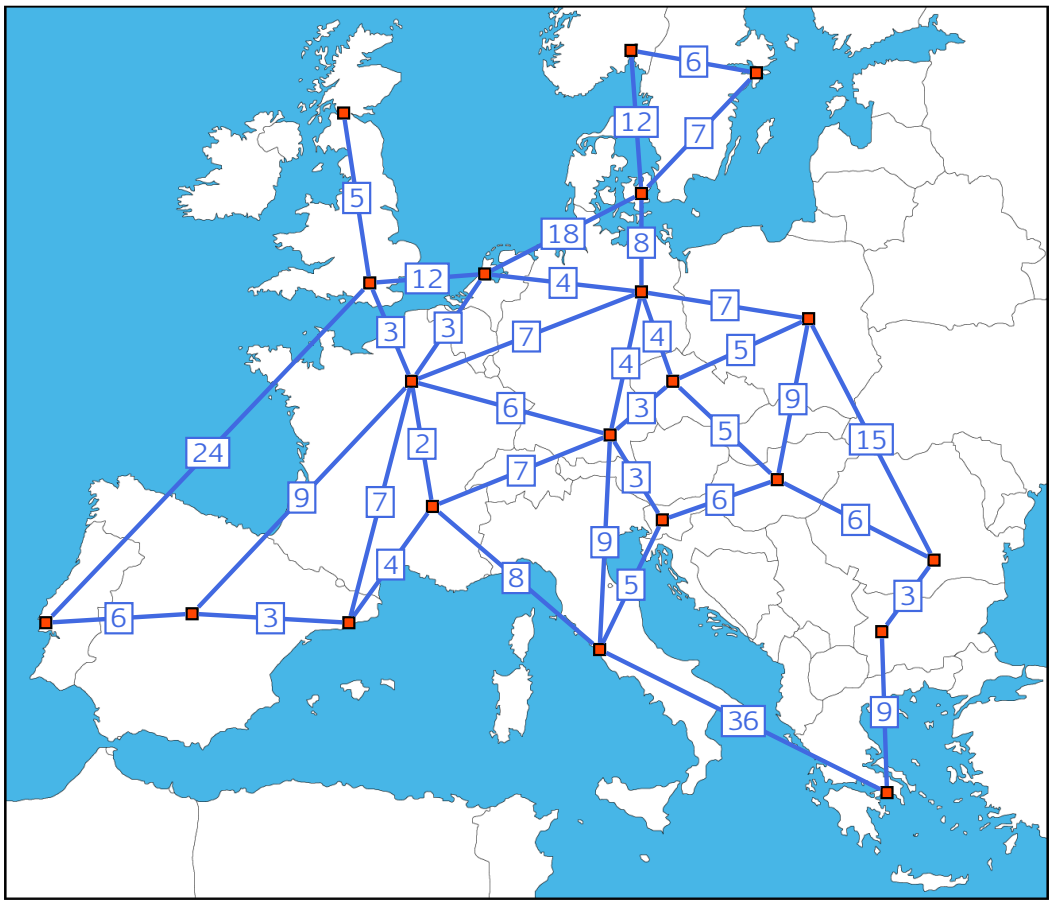
Problème du voyageur de commerce : recherche d'un cycle hamiltonien de poids minimal dans un graphe complet.



Graphes : . . . mais aussi informatiques

Plus court chemin

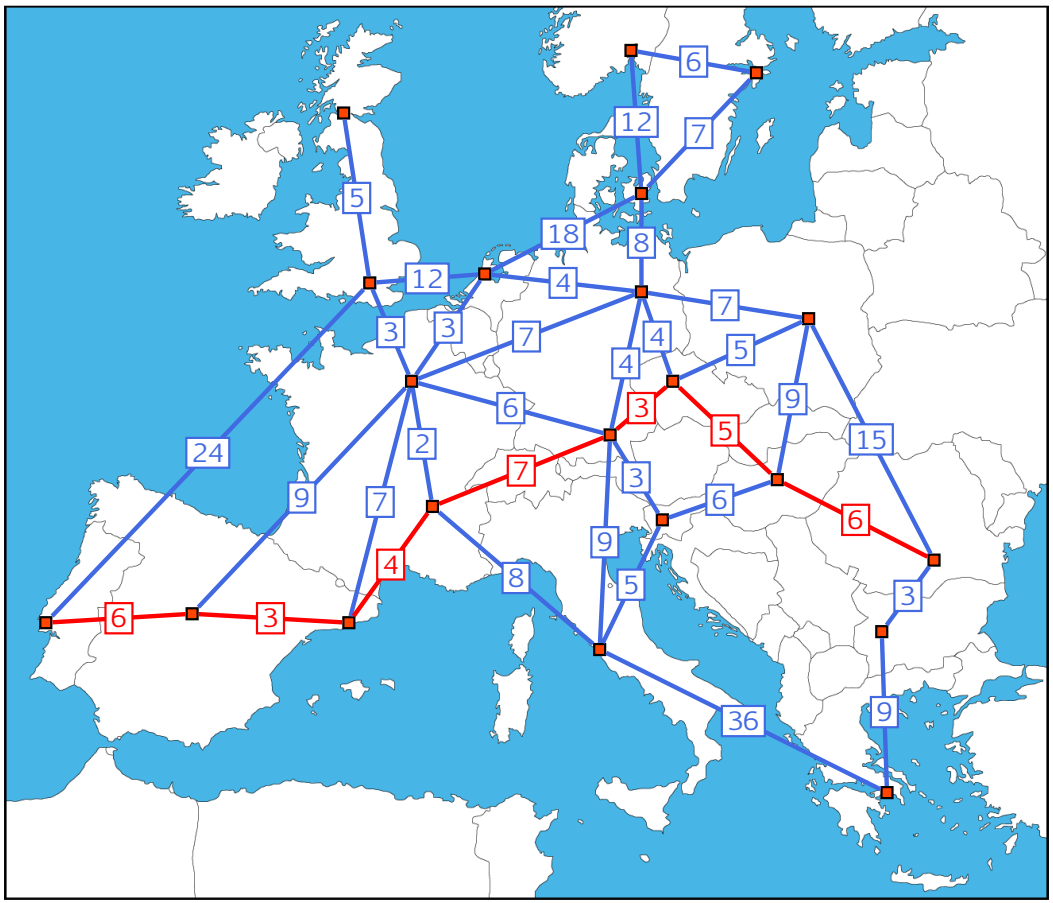
Comment trouver le chemin de poids minimal entre deux sommets d'un graphe pondéré ? ⟶ Algorithmes de Floyd-Warshall, de Dijkstra . . .



Graphes : . . . mais aussi informatiques

Plus court chemin

Comment trouver le chemin de poids minimal entre deux sommets d'un graphe pondéré ? ⟶ Algorithmes de Floyd-Warshall, de Dijkstra . . .



Le plus court chemin de Lisbonne à Bucarest.

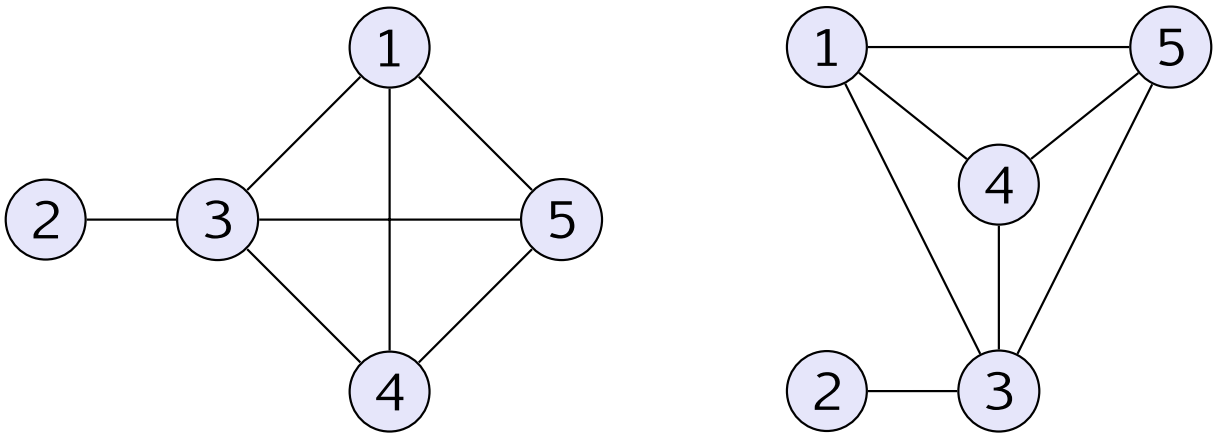
Graphes non orientés

Un graphe $G = (V, E)$ est défini par ses sommets $V = \{v_1, v_2, \ldots, v_n\}$ et ses arêtes $E = \{e_1, e_2, \ldots, e_m\}$ (arête = paire non ordonnée de sommets).

Graphes non orientés

Un graphe $G = (V, E)$ est défini par ses sommets $V = \{v_1, v_2, \ldots, v_n\}$ et ses arêtes $E = \{e_1, e_2, \ldots, e_m\}$ (arête = paire non ordonnée de sommets).

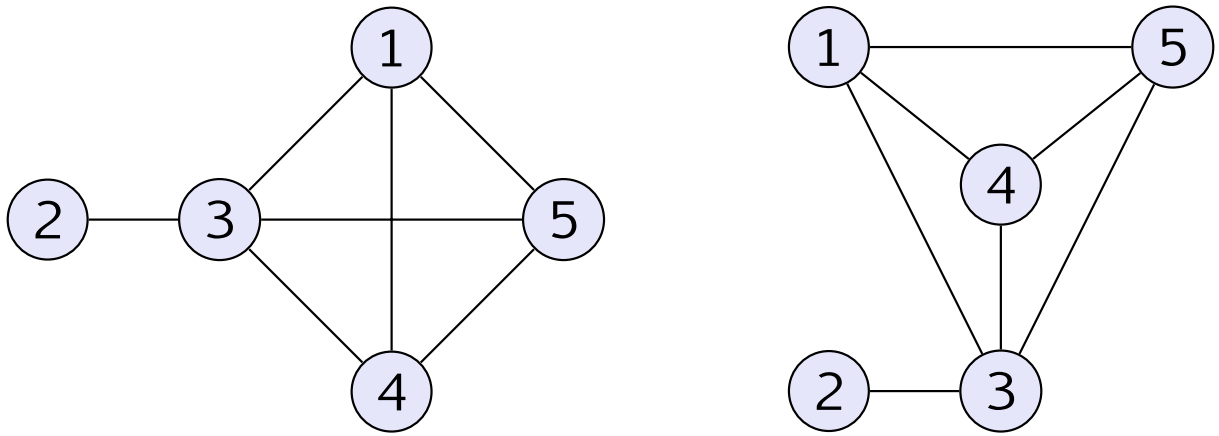
Exemple :
$V = \{1, 2, 3, 4, 5\}$ et $E = \{(1,3), (1,4), (1,5), (2,3), (3,4), (3,5), (4,5)\}$.



Graphes non orientés

Un graphe $G = (V, E)$ est défini par ses sommets $V = \{v_1, v_2, \ldots, v_n\}$ et ses arêtes $E = \{e_1, e_2, \ldots, e_m\}$ (arête = paire non ordonnée de sommets).

Exemple :
$V = \{1, 2, 3, 4, 5\}$ et $E = \{(1,3), (1,4), (1,5), (2,3), (3,4), (3,5), (4,5)\}$.



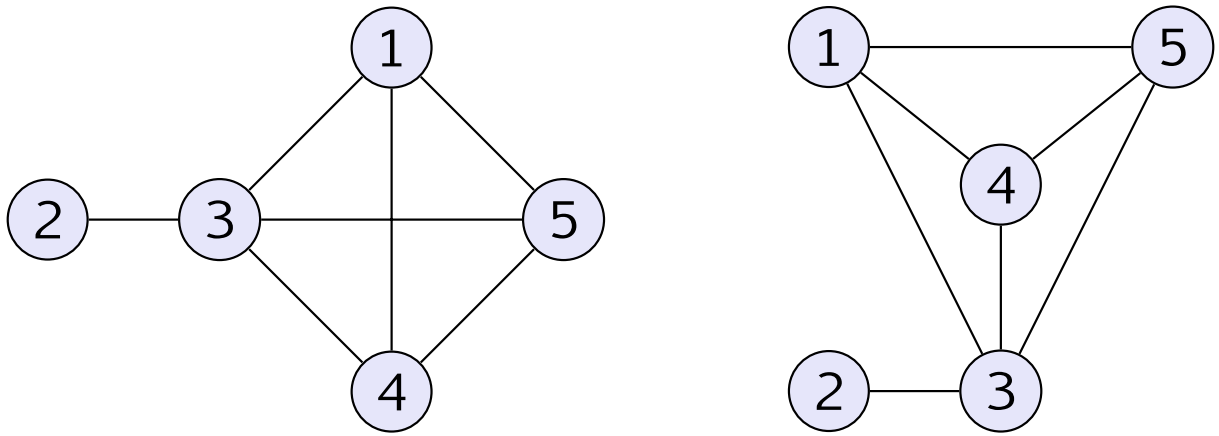
G est un graphe d'ordre 5 ; il possède :

- un sommet de degré 1 (le sommet 2) ;
- trois sommets de degré 3 (les sommets 1, 4 et 5) ;
- un sommet de degré 4 (le sommet 3).

Graphes non orientés

Un graphe $G = (V, E)$ est défini par ses sommets $V = \{v_1, v_2, \ldots, v_n\}$ et ses arêtes $E = \{e_1, e_2, \ldots, e_m\}$ (arête = paire non ordonnée de sommets).

Exemple :
$V = \{1, 2, 3, 4, 5\}$ et $E = \{(1,3), (1,4), (1,5), (2,3), (3,4), (3,5), (4,5)\}$.



G est un graphe d'ordre 5 ; il possède :

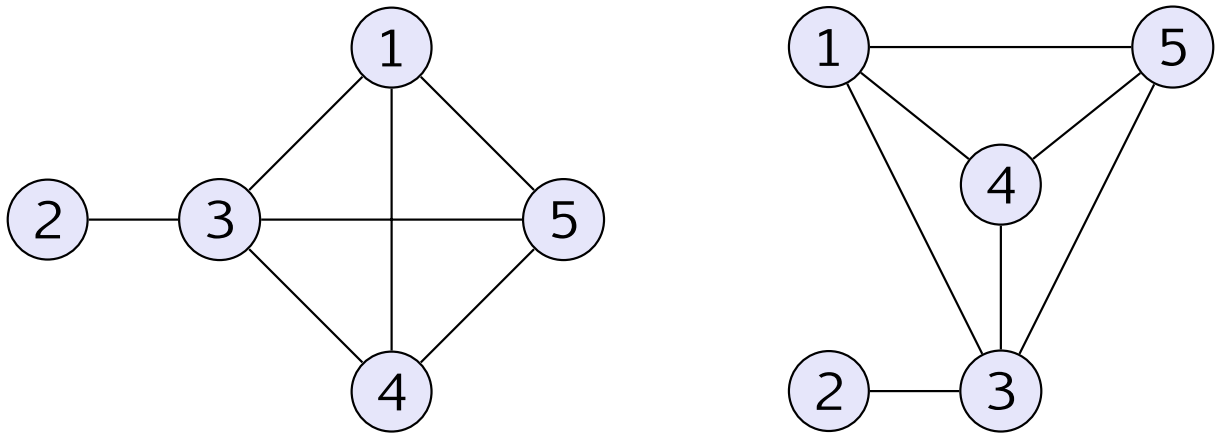
- un sommet de degré 1 (le sommet 2) ;
- trois sommets de degré 3 (les sommets 1, 4 et 5) ;
- un sommet de degré 4 (le sommet 3).

On supposera les graphes simples : $\forall v \in V, (v, v) \notin E$.

Graphes non orientés

Un graphe $G = (V, E)$ est défini par ses sommets $V = \{v_1, v_2, \ldots, v_n\}$ et ses arêtes $E = \{e_1, e_2, \ldots, e_m\}$ (arête = paire non ordonnée de sommets).

Exemple :
$V = \{1, 2, 3, 4, 5\}$ et $E = \{(1,3), (1,4), (1,5), (2,3), (3,4), (3,5), (4,5)\}$.



Si G est un graphe non orienté simple, $\sum_{v \in V} \deg(v) = 2|E|$.

(Dans la somme des degrés des sommets chaque arête est comptée deux fois.)

Graphes non orientés

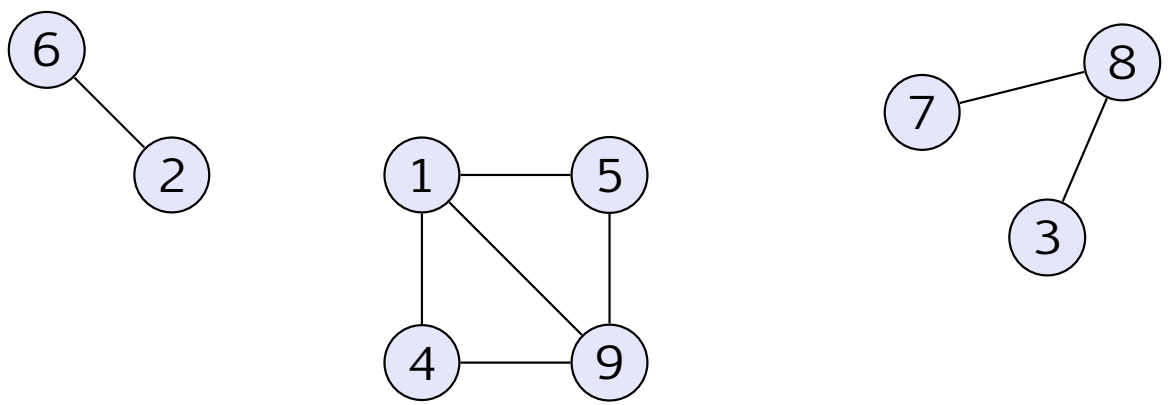
Chemins

Un chemin de longueur k reliant les sommets a et b est une suite finie $x_0 = a, x_1, \ldots, x_k = b$ de sommets tel que $\forall i \in [\![0, k-1]\!]$, $(x_i, x_{i+1}) \in E$. Le chemin est cyclique lorsque $a = b$.

Graphes non orientés

Chemins

Un chemin de longueur k reliant les sommets a et b est une suite finie $x_0 = a, x_1, \ldots, x_k = b$ de sommets tel que $\forall i \in [\![0, k-1]\!]$, $(x_i, x_{i+1}) \in E$. Le chemin est cyclique lorsque $a = b$.
La distance entre a et b est la plus petite des longueurs des chemins reliant a et b, s'il en existe.
Lorsque tous les sommets sont à distance finie les uns des autres, on dit que le graphe est connexe. Un graphe non connexe peut être décomposé en plusieurs composantes connexes, qui sont des sous-graphes connexes maximaux.



Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Preuve par récurrence sur n.

- Ce résultat est évident si $n = 1$.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Preuve par récurrence sur n.

- Ce résultat est évident si $n = 1$.
- Si $n > 1$, on distingue deux cas.
  - Si G possède un sommet x de degré 1, on le supprime ainsi que l'arête qui le relie au graphe. Le graphe ainsi obtenu reste connexe donc comporte au moins $n-2$ arêtes, ce qui prouve que G possède au moins $n-1$ arêtes.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Preuve par récurrence sur n.

- Ce résultat est évident si $n = 1$.
- Si $n > 1$, on distingue deux cas.
  - Si G possède un sommet x de degré 1, on le supprime ainsi que l'arête qui le relie au graphe. Le graphe ainsi obtenu reste connexe donc comporte au moins $n-2$ arêtes, ce qui prouve que G possède au moins $n-1$ arêtes.
  - Dans le cas contraire, tous les sommets de G sont au moins de degré 2. Or la somme des degrés d'un graphe est égal à deux fois son nombre d'arêtes, donc G possède au moins n arêtes.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Un graphe G dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Un graphe G dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

On part d'un sommet v_1, et on construit une suite finie de sommets $v_1, v_2, \ldots, v_k$ de la façon suivante :

- $v_i \notin \{v_1, v_2, \ldots, v_{i-1}\}$;
- v_i est voisin de v_{i-1}.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Un graphe G dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

On part d'un sommet v_1, et on construit une suite finie de sommets $v_1, v_2, \ldots, v_k$ de la façon suivante :

- $v_i \notin \{v_1, v_2, \ldots, v_{i-1}\}$;
- v_i est voisin de v_{i-1}.

Puisque les sommets de G sont en nombre fini, cette construction se termine. Or v_k est au moins de degré 2 donc possède, outre v_{k-1}, un autre voisin v_j dans la séquence. Alors $(v_j, v_{j+1}, \ldots, v_k, v_j)$ est un cycle de G.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Un graphe G dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

Un graphe acyclique d'ordre n comporte au plus $n-1$ arêtes.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Un graphe G dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

Un graphe acyclique d'ordre n comporte au plus $n-1$ arêtes.

Preuve par récurrence sur n.

- Ce résultat est évident si $n = 1$.

Graphes non orientés

Chemins

Un graphe connexe d'ordre n possède au moins $n-1$ arêtes.

Un graphe G dont tout sommet est de degré supérieur ou égal à 2 possède au moins un cycle.

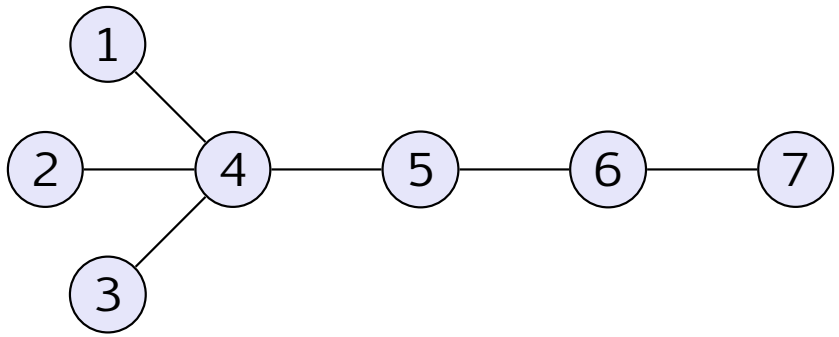
Un graphe acyclique d'ordre n comporte au plus $n-1$ arêtes.

Preuve par récurrence sur n.

- Ce résultat est évident si $n = 1$.
- Si $n > 1$, on considère un graphe acyclique d'ordre n. D'après le résultat précédent G possède au moins un sommet x de degré 0 ou 1. On le supprime ainsi que l'arête qui lui est reliée. Le graphe obtenu est toujours acyclique et d'ordre $n-1$ donc par hypothèse de récurrence possède au plus $n-2$ arêtes. Ainsi, G possède au plus $n-1$ arêtes.

Arbres

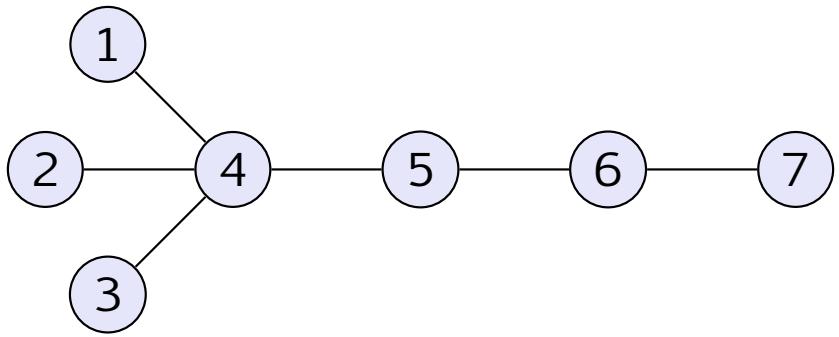
Un arbre est un graphe connexe acyclique.



Un arbre d'ordre n possède exactement $n - 1$ arêtes.

Arbres

Un arbre est un graphe connexe acyclique.

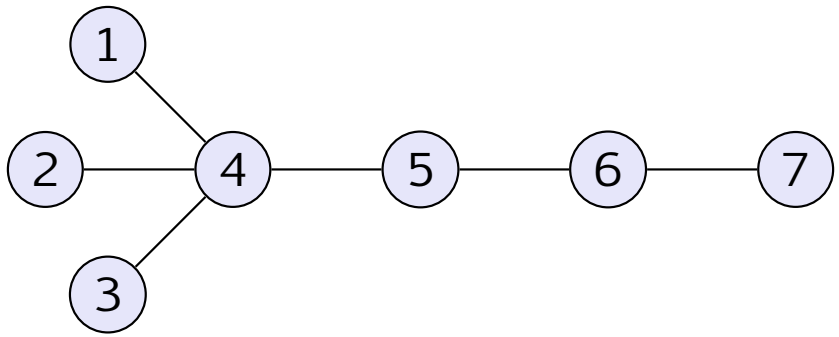


Pour un graphe G d'ordre n, il y a équivalence entre :

1. G est un arbre ;
2. G est un graphe connexe à $n-1$ arêtes ;
3. G est un graphe acyclique à $n-1$ arêtes.

Arbres

Un arbre est un graphe connexe acyclique.



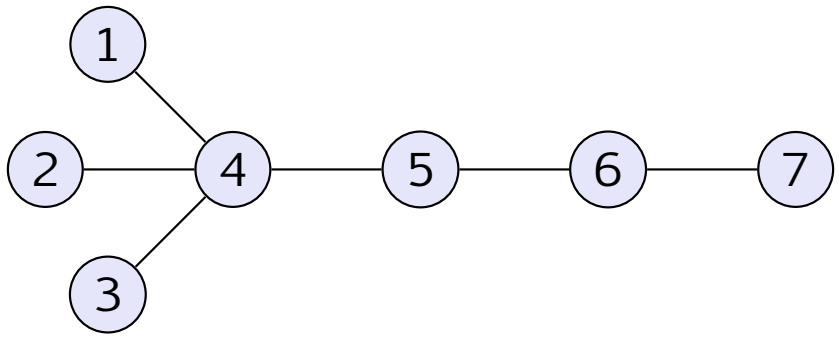
Pour un graphe G d'ordre n, il y a équivalence entre :

1. G est un arbre ;
2. G est un graphe connexe à $n-1$ arêtes ;
3. G est un graphe acyclique à $n-1$ arêtes.

Il suffit de montrer l'équivalence des propriétés 2 et 3.

Arbres

Un arbre est un graphe connexe acyclique.



Pour un graphe G d'ordre n, il y a équivalence entre :

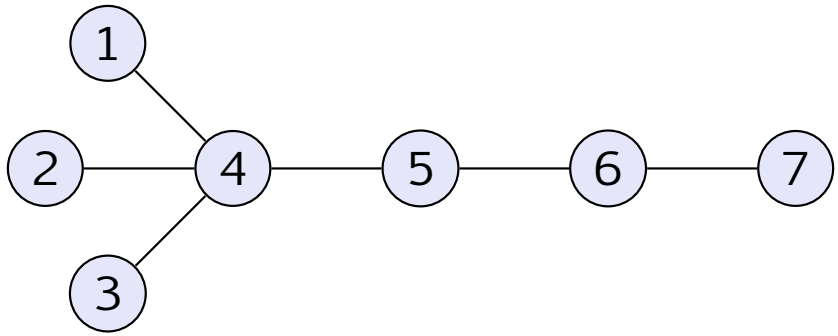
1. G est un arbre ;
2. G est un graphe connexe à $n-1$ arêtes ;
3. G est un graphe acyclique à $n-1$ arêtes.

Il suffit de montrer l'équivalence des propriétés 2 et 3.

- Supposons qu'un graphe connexe G à $n-1$ arêtes possède un cycle. La suppression d'une arête de ce cycle crée un graphe à $n-2$ arêtes toujours connexe, ce qui est absurde. G est donc acyclique.

Arbres

Un arbre est un graphe connexe acyclique.



Pour un graphe G d'ordre n, il y a équivalence entre :

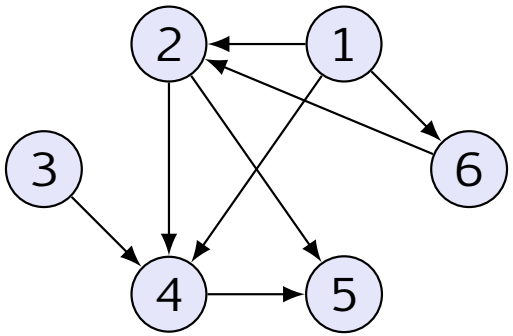
1. G est un arbre ;
2. G est un graphe connexe à $n-1$ arêtes ;
3. G est un graphe acyclique à $n-1$ arêtes.

Il suffit de montrer l'équivalence des propriétés 2 et 3.

- Supposons qu'un graphe acyclique G à $n-1$ arêtes ne soit pas connexe. Il existe deux sommets x et y qui ne peuvent être reliés. On ajoute une arête entre x et y. Le graphe obtenu est toujours acyclique mais possède n arêtes, ce qui est absurde. G est donc connexe.

Graphes orientés

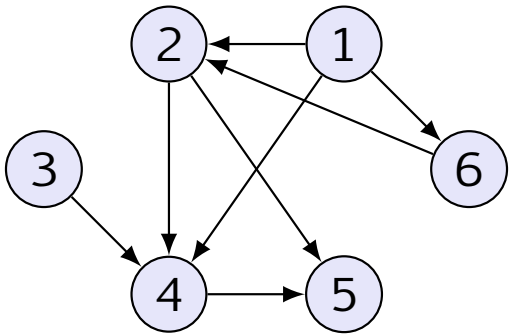
On obtient un graphe orienté en distinguant la paire de sommets (a, b) de la paire (b, a).



$V = \{1, 2, 3, 4, 5, 6\}$
$E = \{(1, 2), (1, 4), (1, 6), (2, 4), (2, 5), (3, 4), (4, 5), (6, 2)\}$

Graphes orientés

On obtient un graphe orienté en distinguant la paire de sommets (a, b) de la paire (b, a).



$V = \{1, 2, 3, 4, 5, 6\}$
$E = \{(1, 2), (1, 4), (1, 6), (2, 4), (2, 5), (3, 4), (4, 5), (6, 2)\}$

Le degré sortant d'un sommet est le nombre d'arcs dont il est la première composante, le degré entrant le nombre d'arcs dont il est la seconde composante).
Le sommet 1 a un degré sortant égal à 3 et un degré entrant égal à 0 ;
le sommet 2 a un degré entrant et un degré sortant égaux à 2.

Graphes orientés

Chemins

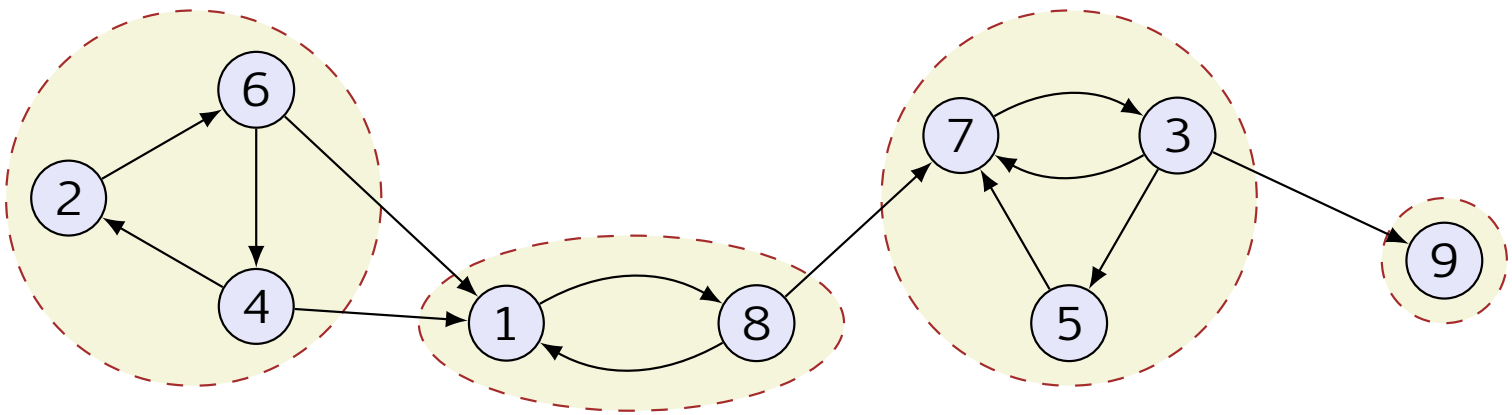
Les notions de chemin et de distance s'étendent au cas des graphes orientés.

Graphes orientés

Chemins

Les notions de chemin et de distance s'étendent au cas des graphes orientés.

Un graphe orienté est dit fortement connexe lorsque pour tout couple de sommets (a, b) il existe un chemin reliant a à b et un chemin reliant b à a.



Un graphe orienté peut être décomposé en composantes fortement connexes.

Arbre enraciné

Si a et b sont deux sommets distincts d'un arbre G, il existe un unique chemin reliant a et b.

Arbre enraciné

Si a et b sont deux sommets distincts d'un arbre G, il existe un unique chemin reliant a et b.

G est connexe, ce qui assure l'existence d'un tel chemin. S'il en existe un deuxième, on parcourt un cycle en empruntant le premier entre a et b puis le second entre b et a, ce qui contredit le caractère acyclique de G.

Arbre enraciné

Si a et b sont deux sommets distincts d'un arbre G, il existe un unique chemin reliant a et b.

Conséquence : il est possible de choisir arbitrairement un sommet r d'un arbre G puis d'orienter les arêtes de ce graphe de sorte qu'il existe un chemin reliant r à tous les autres sommets.

Arbre enraciné

Si a et b sont deux sommets distincts d'un arbre G, il existe un unique chemin reliant a et b.

Conséquence : il est possible de choisir arbitrairement un sommet r d'un arbre G puis d'orienter les arêtes de ce graphe de sorte qu'il existe un chemin reliant r à tous les autres sommets.

Par récurrence sur l'ordre n de l'arbre G.

- Si $n = 1$, il n'y a pas d'arête à orienter.

Arbre enraciné

Si a et b sont deux sommets distincts d'un arbre G, il existe un unique chemin reliant a et b.

Conséquence : il est possible de choisir arbitrairement un sommet r d'un arbre G puis d'orienter les arêtes de ce graphe de sorte qu'il existe un chemin reliant r à tous les autres sommets.

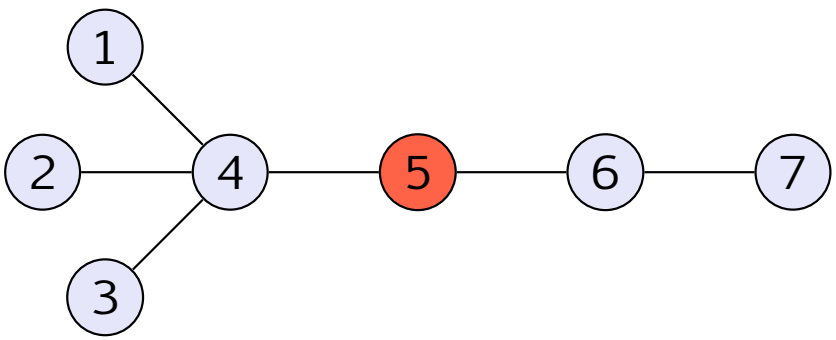
Par récurrence sur l'ordre n de l'arbre G.

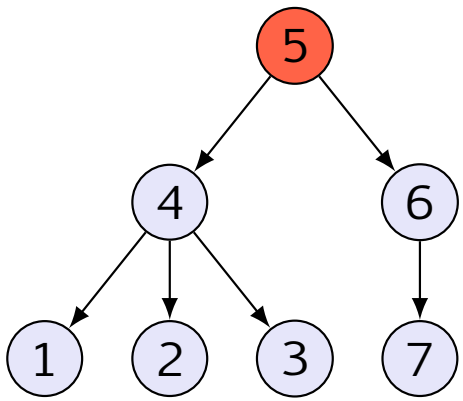
- Si $n = 1$, il n'y a pas d'arête à orienter.
- Si $n > 1$, nous savons qu'un arbre G d'ordre n possède $n - 1$ arêtes donc que le degré de G est égal à $2n - 2$. Il existe au moins deux sommets de degré 1 donc au moins un qui soit différent de r ; notons-le a.
  Si on supprime ce sommet de G ainsi que l'arête qui le relie à l'arbre, on obtient un arbre d'ordre $n - 1$ à qui on peut appliquer l'hypothèse de récurrence pour l'enraciner en r. Il reste à orienter l'arête supprimée en direction de a pour enraciner G en r.

Arbre enraciné

Si a et b sont deux sommets distincts d'un arbre G, il existe un unique chemin reliant a et b.

Conséquence : il est possible de choisir arbitrairement un sommet r d'un arbre G puis d'orienter les arêtes de ce graphe de sorte qu'il existe un chemin reliant r à tous les autres sommets.



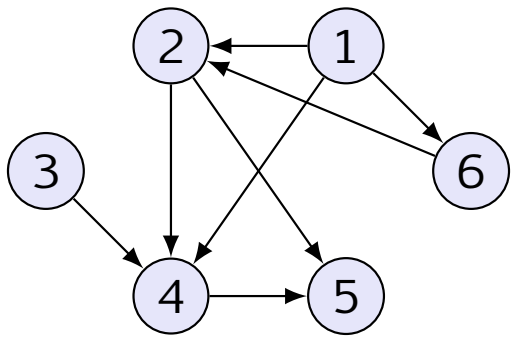


Mise en œuvre pratique

Liste d'adjacence (première version)

Un sommet est formé d'un identifiant et de la liste des identifiants de ses voisins ; un graphe est une liste de sommets.

```
type 'a sommet = {Id : 'a ; Voisins : 'a list} ;;
type 'a graph == 'a sommet list ;;
```




```
# let g = [{Id = 1; Voisins = [2; 4; 6]};
           {Id = 2; Voisins = [4; 5]};
           {Id = 3; Voisins = [4]};
           {Id = 4; Voisins = [5]};
           {Id = 5; Voisins = []};
           {Id = 6; Voisins = [2]}] ;;
```

Avantage : ajout et suppression aisée des sommets et des arêtes ;
Inconvénient : pas d'accès rapide à un sommet ou une arête particulière.

Mise en œuvre pratique

Liste d'adjacence (première version)

Un sommet est formé d'un identifiant et de la liste des identifiants de ses voisins ; un graphe est une liste de sommets.

```
type 'a sommet = {Id : 'a ; Voisins : 'a list} ;;
type 'a graph == 'a sommet list ;;
```

Ajout d'une arête :

```
let rec ajoute_arete g a b = match g with
  | []                    -> failwith "ajoute_arete"
  | s::q when s.Id = a && mem b s.Voisins -> g
  | s::q when s.Id = a -> {Id = a ; Voisins = b::(s.Voisins)}::q
  | s::q               -> s::(ajoute_arete q a b) ;;
```

Mise en œuvre pratique

Liste d'adjacence (première version)

Un sommet est formé d'un identifiant et de la liste des identifiants de ses voisins ; un graphe est une liste de sommets.

```
type 'a sommet = {Id : 'a ; Voisins : 'a list} ;;
type 'a graph == 'a sommet list ;;
```

Suppression d'une arête :

```
let rec supprime_arete g a b =
  let rec aux = function
    | []              -> failwith "supprime_arete"
    | t::q when t = b -> q
    | t::q            -> t::(aux q)
  in match g with
    | []                  -> failwith "supprime_arete"
    | s::q when s.Id = a -> {Id = a ; Voisins = aux s.Voisins}::q
    | s::q                -> s::(supprime_arete q a b) ;;
```

Mise en œuvre pratique

Liste d'adjacence (première version)

Un sommet est formé d'un identifiant et de la liste des identifiants de ses voisins ; un graphe est une liste de sommets.

```
type 'a sommet = {Id : 'a ; Voisins : 'a list} ;;
type 'a graph == 'a sommet list ;;
```

Ajout d'un sommet :

```
let rec ajoute_sommet g a = match g with
  | []                  -> [{Id = a ; Voisins = []}]
  | s::q when s.Id = a -> g
  | s::q                -> s::(ajoute_sommet q a) ;;
```

Mise en œuvre pratique

Liste d'adjacence (première version)

Un sommet est formé d'un identifiant et de la liste des identifiants de ses voisins ; un graphe est une liste de sommets.

```
type 'a sommet = {Id : 'a ; Voisins : 'a list} ;;
type 'a graph == 'a sommet list ;;
```

Suppression d'un sommet :

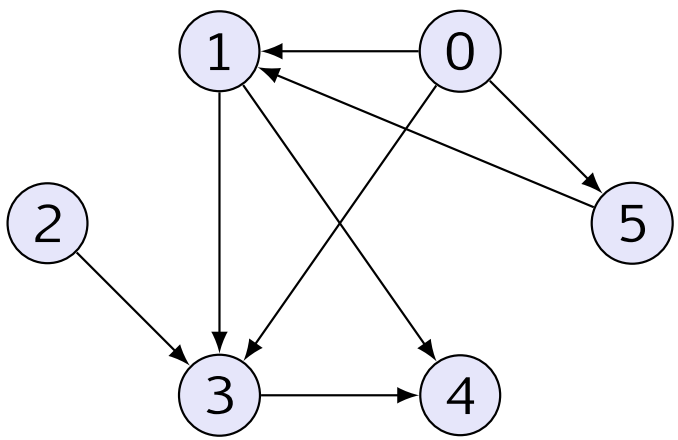
```
let rec supprime_sommet g a =
  let rec aux = function
    | []               -> []
    | t::q when t = a -> q
    | t::q             -> t::(aux q)
  in match g with
    | []                    -> []
    | s::q when s.Id = a -> supprime_sommet q a
    | s::q          -> let ns = {Id = s.Id ; Voisins = aux s.Voisins}
                       in ns::(supprime_sommet q a) ;;
```

Mise en œuvre pratique

Liste d'adjacence (seconde version)

On modifie cette première représentation en utilisant un tableau pour stocker les listes d'adjacence ⟶ on fait coincider indices du tableau et identifiants des sommets.

```
type voisin == int list ;;
type graphe == voisin vect ;;
```




```
# let g = [| [1; 3; 5];
             [3; 4];
             [3];
             [4];
             [];
             [1] |] ;;
```

Cette nouvelle représentation des listes d'adjacence est toujours aussi économique en espace ($\Theta(n+p)$) et permet encore l'ajout ou la suppression d'une arête, mais plus l'ajout ou la suppression d'un sommet.

Mise en œuvre pratique

Désorientation d'un graphe

Inconvénient de la représentation par listes d'adjacence : pour déterminer si un arc (a, b) est présent dans un graphe, il n'existe pas de moyen plus rapide que de rechercher b dans la liste d'adjacence de a.
En particulier, il est difficile de déterminer rapidement si un graphe est non orienté.

Mise en œuvre pratique

Désorientation d'un graphe

Inconvénient de la représentation par listes d'adjacence : pour déterminer si un arc (a, b) est présent dans un graphe, il n'existe pas de moyen plus rapide que de rechercher b dans la liste d'adjacence de a.
En particulier, il est difficile de déterminer rapidement si un graphe est non orienté.
Désorienter un graphe consiste à calculer le plus petit graphe non orienté g' contenant g.

```
let desoriente g =
  let n = vect_length g in
  let aux i j = if not mem i g.(j) then g.(j) <- i::g.(j) in
  for i = 0 to n-1 do
    do_list (aux i) g.(i)
  done ;;
```

Mise en œuvre pratique

Matrice d'adjacence

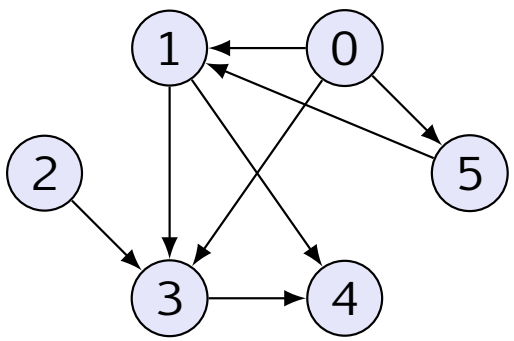
Avantage de la représentation par listes d'adjacence : quantité minimale de mémoire $\Theta(n+p)$.
Inconvénient : pas d'accès à coût constant aux arêtes.

Mise en œuvre pratique

Matrice d'adjacence

Avantage de la représentation par listes d'adjacence : quantité minimale de mémoire $\Theta(n+p)$.
Inconvénient : pas d'accès à coût constant aux arêtes.
Matrice d'adjacence : ordonner les sommets $V = \{v_1, v_2, \ldots, v_n\}$ et utiliser une matrice $M \in \mathcal{M}_n(\{0,1\})$ pour représenter les arêtes :

$$m_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in E \\ 0 & \text{sinon} \end{cases}$$



$$M = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Avantage : ajout et suppression d'une arête à coût constant ;
Inconvénient : coût spatial en $\Theta(n^2)$.

Matrice d'adjacence

Calcul de la matrice d'adjacence à partir des listes d'adjacence :

```
let graphe_to_mat g =
  let n = vect_length g in
  let m = make_matrix n n 0 in
  for a = 0 to n-1 do
      do_list (function b -> m.(a).(b) <- 1) g.(a)
    done ;
  m ;;
```

Matrice d'adjacence

Calcul de la matrice d'adjacence à partir des listes d'adjacence :

```
let graphe_to_mat g =
  let n = vect_length g in
  let m = make_matrix n n 0 in
  for a = 0 to n-1 do
      do_list (function b -> m.(a).(b) <- 1) g.(a)
    done ;
  m ;;
```

Fonction réciproque :

```
let mat_to_graphe m =
  let n = vect_length m in
  let g = make_vect n [] in
  for a = 0 to n-1 do
    for b = 0 to n-1 do
      if m.(a).(b) = 1 then g.(a) <- b::g.(a)
    done
  done ;
  g ;;
```

Parcours d'un graphe

On maintient à jour deux listes : la liste des sommets rencontrés («déjà-Vus») et la liste des sommets en cours de traitement («àTraiter»).

```
procedure PARCOURS(sommet : s_0)
    àTraiter ← s_0
    déjàVus ← s_0
    while àTraiter ≠ ∅ do
        àTraiter → s
        traiter(s)
        for t ∈ voisins(s) do
            if t ∉ déjàVus then
                àTraiter ← t
                déjàVus ← t
```

Parcours d'un graphe

On maintient à jour deux listes : la liste des sommets rencontrés («déjà-Vus») et la liste des sommets en cours de traitement («àTraiter»).

```
procedure PARCOURS(sommet : s_0)
    àTraiter ← s_0
    déjàVus ← s_0
    while àTraiter ≠ ∅ do
        àTraiter → s
        traiter(s)
        for t ∈ voisins(s) do
            if t ∉ déjàVus then
                àTraiter ← t
                déjàVus ← t
```

Arborescence associée à un parcours : à chaque parcours débutant par un sommet s_0 peut être associé un arbre enraciné en s_0 : on débute avec le graphe $(\{s_0\},\emptyset)$ et à chaque insertion d'un nouveau sommet t dans la liste « àTraiter » on ajoute le sommet t et l'arête (s,t).

Parcours d'un graphe

On maintient à jour deux listes : la liste des sommets rencontrés («déjà-Vus») et la liste des sommets en cours de traitement («àTraiter»).

procedure PARCOURS(*sommet* : s_0)
    àTraiter $\leftarrow s_0$
    déjàVus $\leftarrow s_0$
    while *àTraiter* $\neq \emptyset$ ***do***
        àTraiter $\rightarrow s$
        traiter(s)
        for $t \in$ *voisins*(s) ***do***
            if $t \notin$ *déjàVus* ***then***
                àTraiter $\leftarrow t$
                déjàVus $\leftarrow t$

Coût du parcours : le coût des manipulations de « àTraiter » est un $\mathrm{O}(n)$; le temps consacré à scruter les listes de voisinage est un $\mathrm{O}(p)$ *à condition de déterminer si un sommet a déjà été vu en coût constant*. Dans ce cas, le coût total d'un parcours est un $\mathrm{O}(n+p)$.

Parcours d'un graphe

On maintient à jour deux listes : la liste des sommets rencontrés («déjà-Vus») et la liste des sommets en cours de traitement («àTraiter»).

```
procedure PARCOURS(sommet : s_0)
    àTraiter ← s_0
    déjàVus ← s_0
    while àTraiter ≠ ∅ do
        àTraiter → s
        traiter(s)
        for t ∈ voisins(s) do
            if t ∉ déjàVus then
                àTraiter ← t
                déjàVus ← t
```

Coût du parcours : le coût des manipulations de « àTraiter » est un $O(n)$; le temps consacré à scruter les listes de voisinage est un $O(p)$ *à condition de déterminer si un sommet a déjà été vu en coût constant*. Dans ce cas, le coût total d'un parcours est un $O(n+p)$.
Pour réaliser cette condition, on utilise un tableau «déjàVus» destiné à marquer chaque sommet au moment où il entre dans la liste «àTraiter».

Parcours d'un graphe

On maintient à jour deux listes : la liste des sommets rencontrés («déjà-Vus») et la liste des sommets en cours de traitement («àTraiter»).

```
procedure PARCOURS(sommet : s_0)
    àTraiter ← s_0
    déjàVus ← s_0
    while àTraiter ≠ ∅ do
        àTraiter → s
        traiter(s)
        for t ∈ voisins(s) do
            if t ∉ déjàVus then
                àTraiter ← t
                déjàVus ← t
```

Le type de parcours dépend du choix de la structure de données choisie pour représenter «àTraiter» :

- Représentation par une file : parcours en largeur ;
- Représentation par une pile : parcours en profondeur.

Parcours en largeur

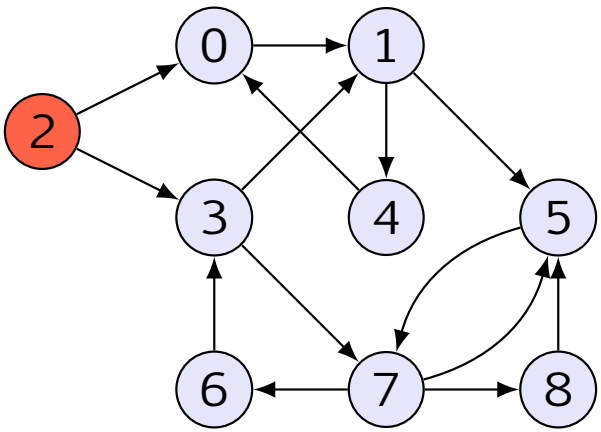
Le parcours en largeur (BFS, pour *Breadth First Search*) consiste à utiliser une file d'attente pour stocker les sommets à traiter.

```
#open "queue" ;;

let bfs g s =
  let dejavu = make_vect (vect_length g) false
  and atraiter = new() in
  add s atraiter ; dejavu.(s) <- true ;
  let rec ajoute_voisin = function
    | []                      -> ()
    | t::q when dejavu.(t) -> ajoute_voisin q
    | t::q                    -> add t atraiter ; dejavu.(t) <- true ;
                                 ajoute_voisin q
  in
  try while true do
      let s = take atraiter in
      traitement s ;
      ajoute_voisin g.(s)
    done
  with Empty -> () ;;
```

Parcours en largeur

Illustration



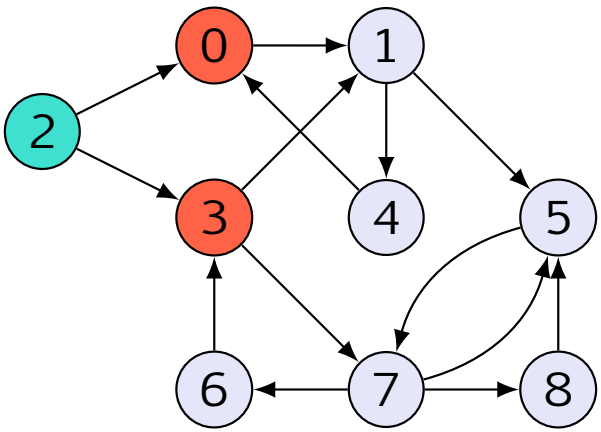
2

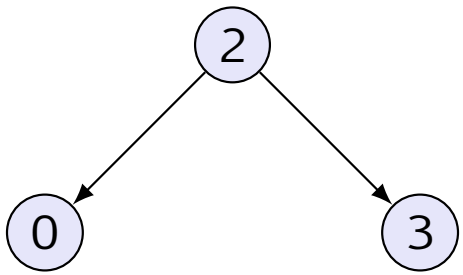
Éléments à traiter : 2

Éléments déjà traités :

Parcours en largeur

Illustration



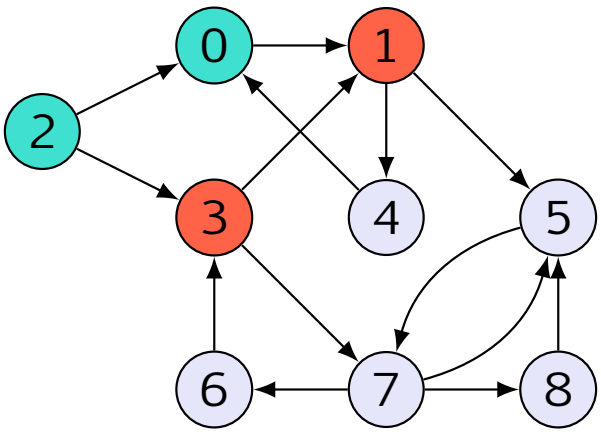


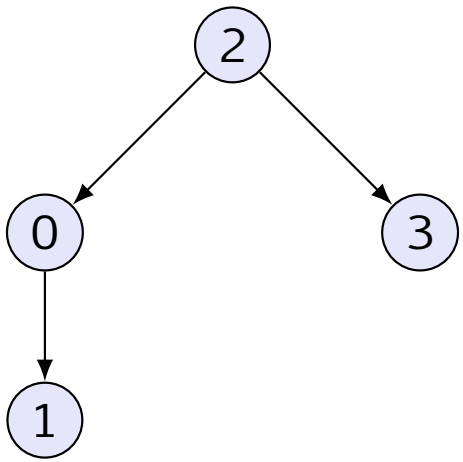
Éléments à traiter : 0 3

Éléments déjà traités : 2

Parcours en largeur

Illustration



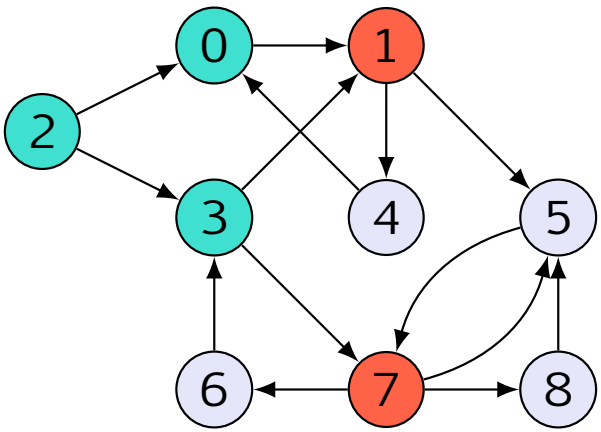


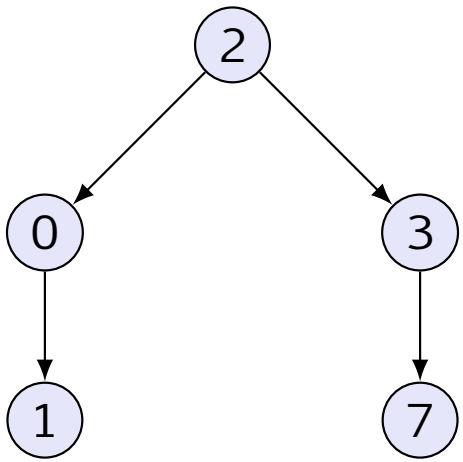
Éléments à traiter : 3 | 1

Éléments déjà traités : 2, 0

Parcours en largeur

Illustration



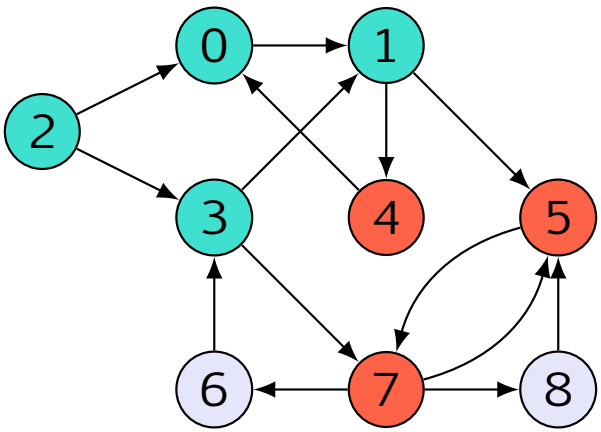


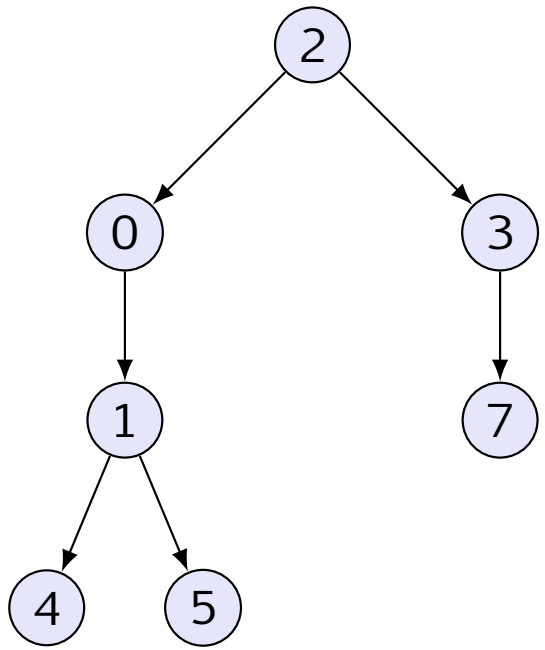
Éléments à traiter : | 1 | 7 |

Éléments déjà traités : 2, 0, 3

Parcours en largeur

Illustration



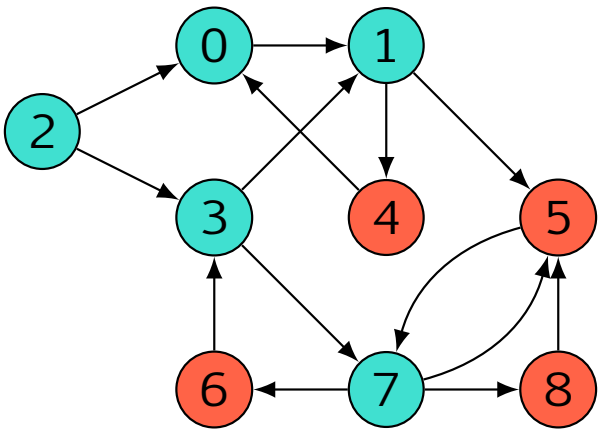


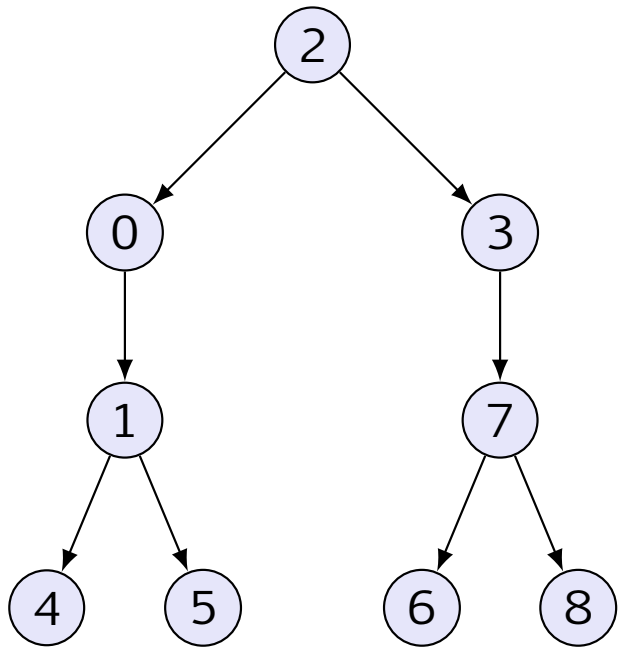
Éléments à traiter : | 7 | 4 | 5 |

Éléments déjà traités : 2, 0, 3, 1

Parcours en largeur

Illustration



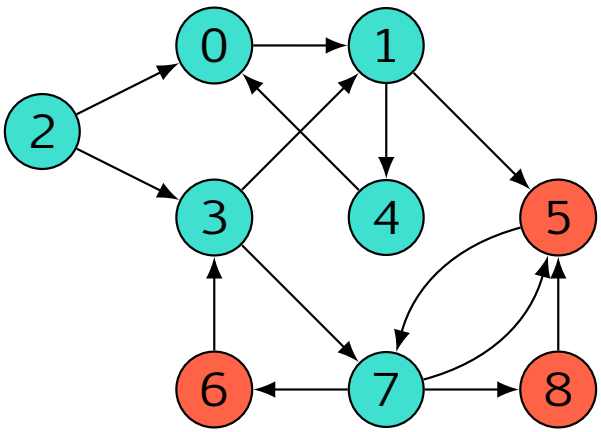


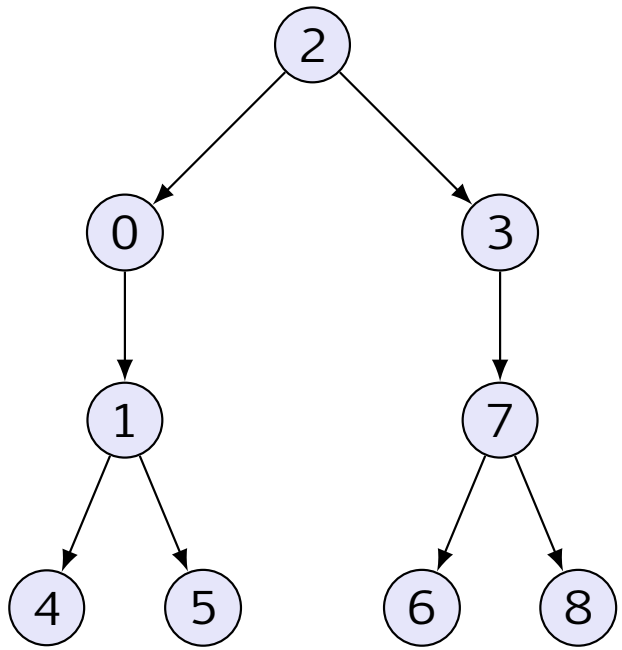
Éléments à traiter : | 4 | 5 | 6 | 8 |

Éléments déjà traités : 2, 0, 3, 1, 7

Parcours en largeur

Illustration



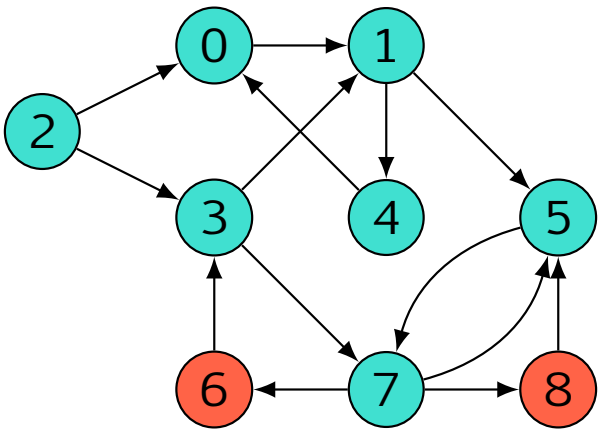


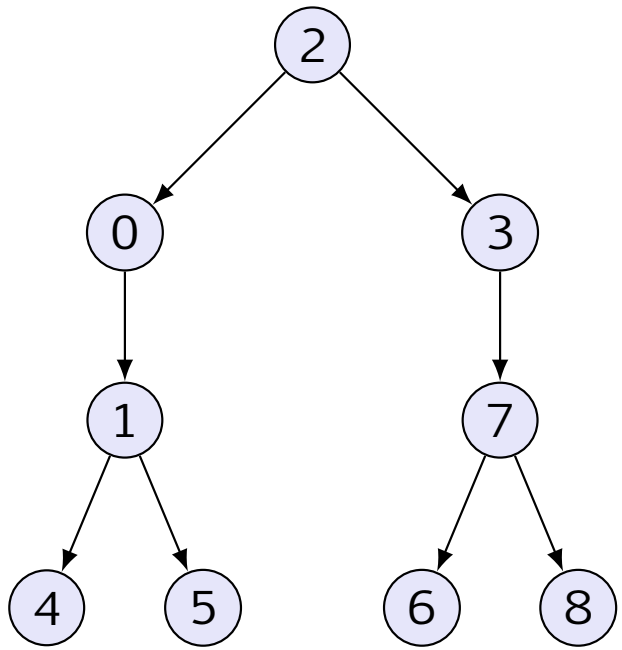
Éléments à traiter : | 5 | 6 | 8 |

Éléments déjà traités : 2, 0, 3, 1, 7, 4

Parcours en largeur

Illustration



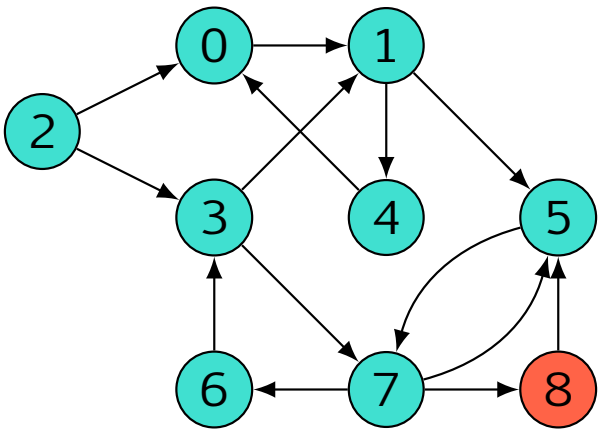


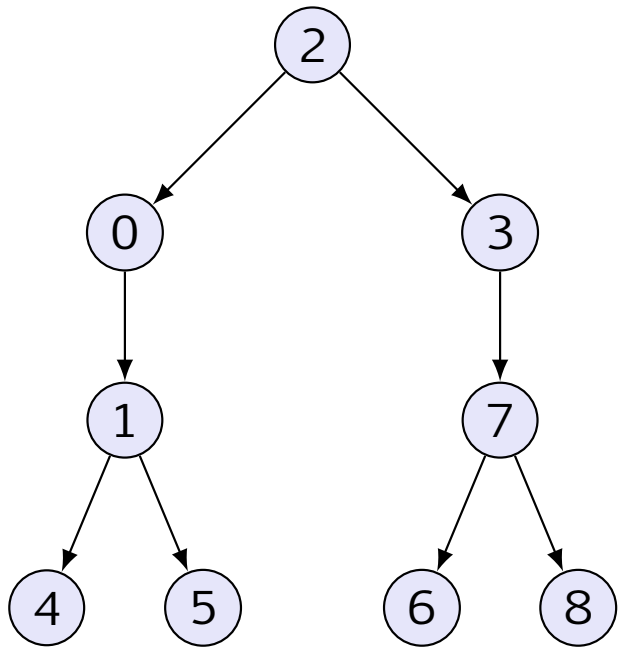
Éléments à traiter : 6 | 8

Éléments déjà traités : 2, 0, 3, 1, 7, 4, 5

Parcours en largeur

Illustration



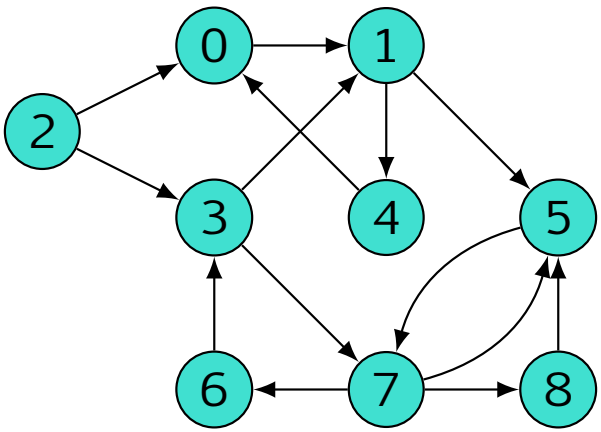


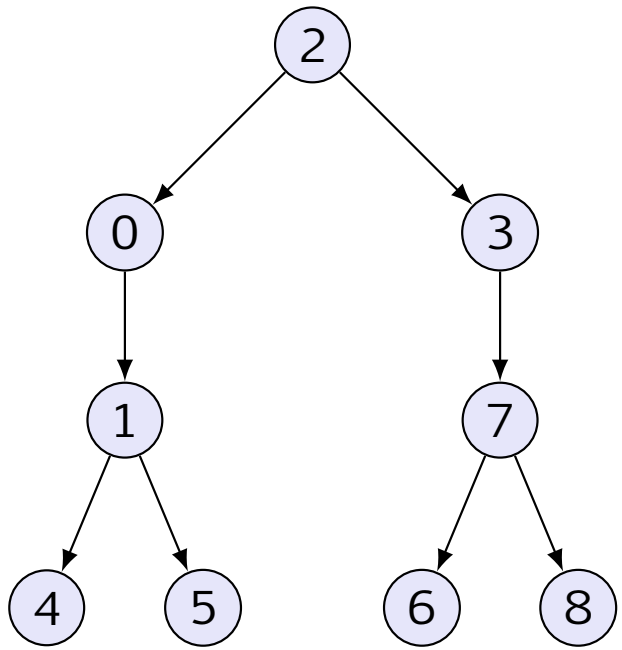
Éléments à traiter : 8

Éléments déjà traités : 2, 0, 3, 1, 7, 4, 5, 6

Parcours en largeur

Illustration





Éléments à traiter :

Éléments déjà traités : 2, 0, 3, 1, 7, 4, 5, 6, 8

Parcours en profondeur

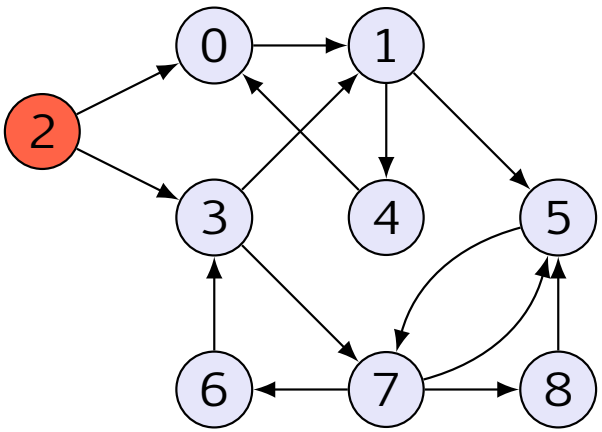
Le parcours en profondeur (DFS, pour *Depth First Search*) consiste à utiliser une pile pour stocker les sommets à traiter.

```
#open "stack" ;;

let dfs g s =
  let dejavu = make_vect (vect_length g) false
  and atraiter = new() in
  push s atraiter ; dejavu.(s) <- true ;
  let rec ajoute_voisin = function
    | []                      -> ()
    | t::q when dejavu.(t) -> ajoute_voisin q
    | t::q                    -> push t atraiter ; dejavu.(t) <- true ;
                                 ajoute_voisin q
  in
  try while true do
      let s = pop atraiter in
      traitement s ;
      ajoute_voisin g.(s)
    done
  with Empty -> () ;;
```

Parcours en profondeur

Illustration



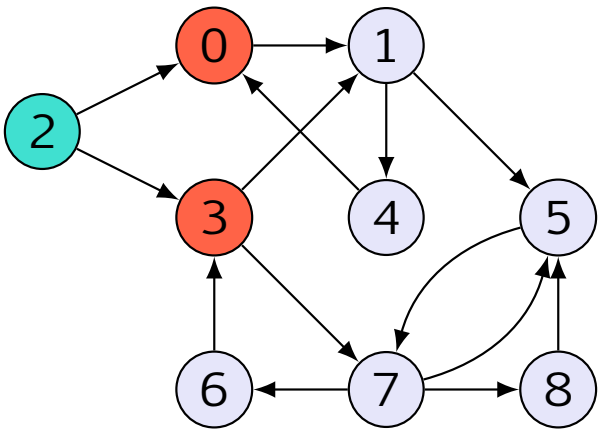
2

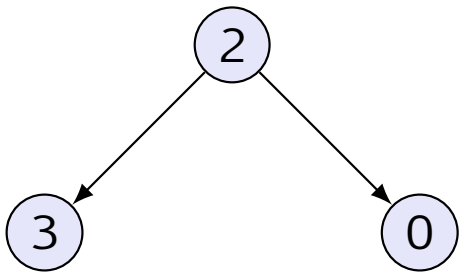
Éléments déjà traités :

Éléments à traiter : 2

Parcours en profondeur

Illustration





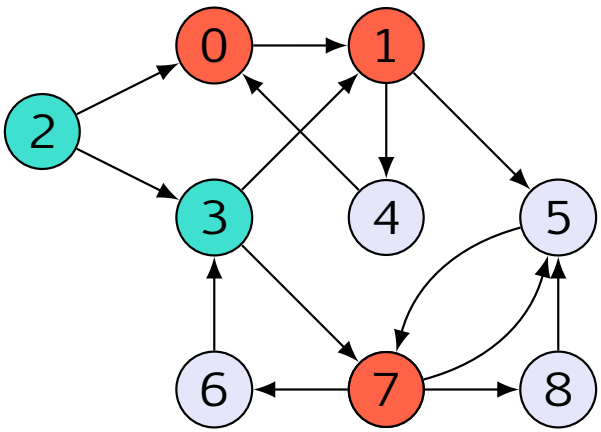
Éléments à traiter :

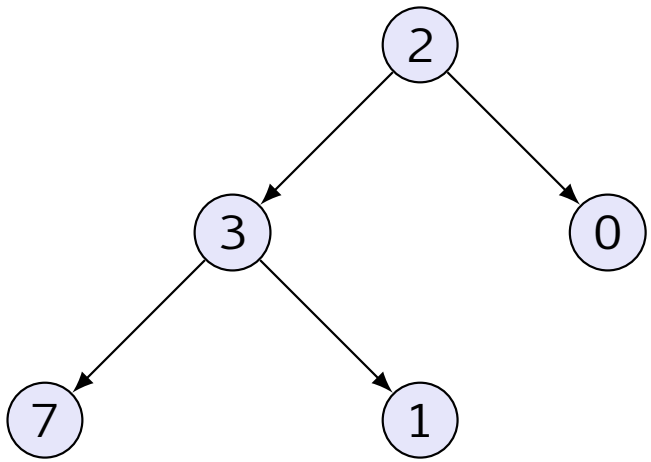
3
0

Éléments déjà traités :
2

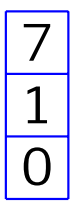
Parcours en profondeur

Illustration





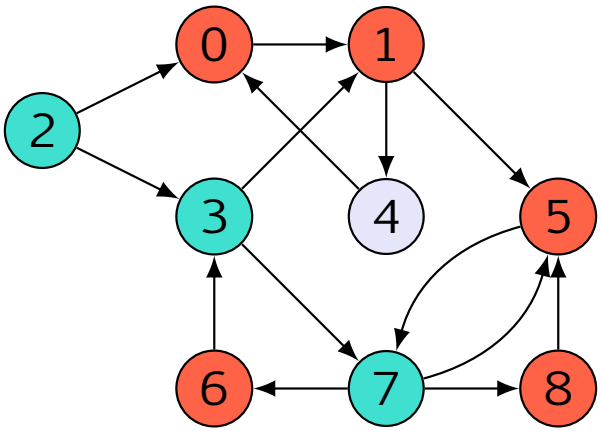
Éléments à traiter :

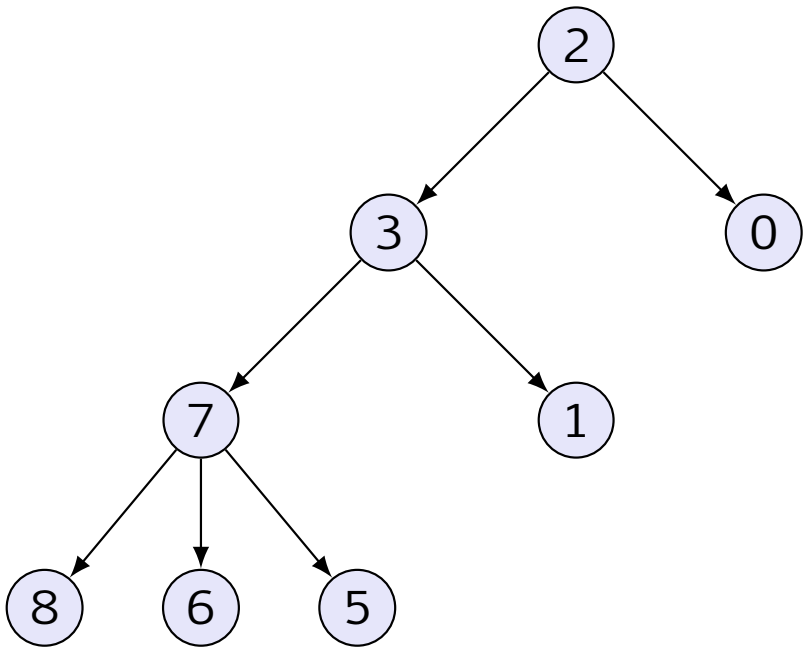


Éléments déjà traités :
2, 3

Parcours en profondeur

Illustration



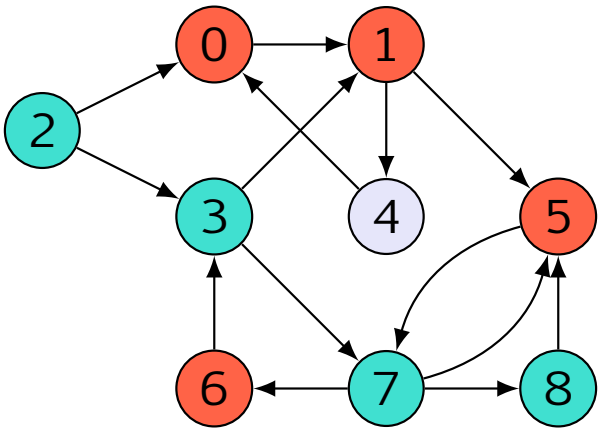


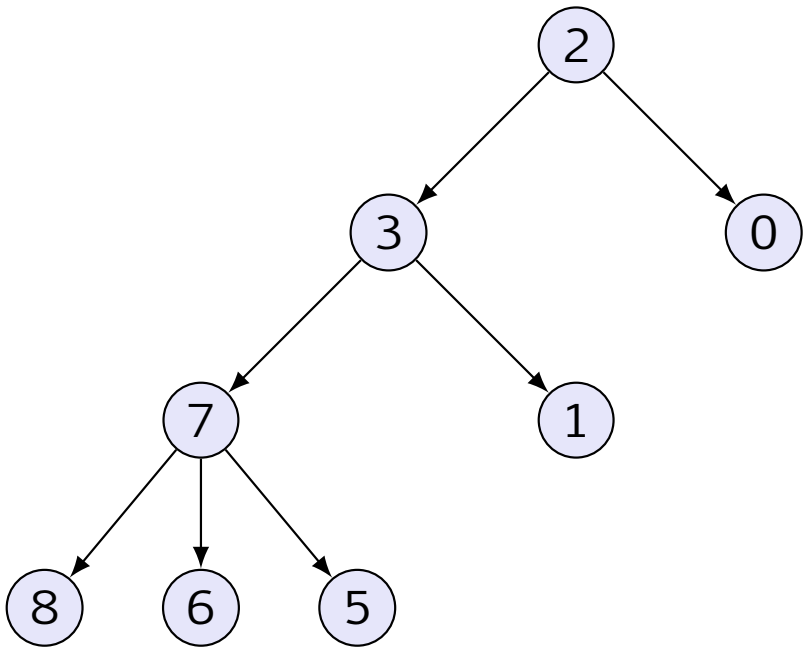
Éléments à traiter :

Éléments déjà traités :
2, 3, 7

Parcours en profondeur

Illustration





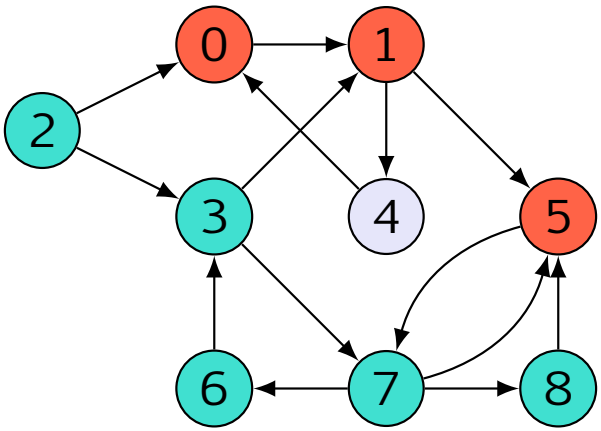
Éléments à traiter :

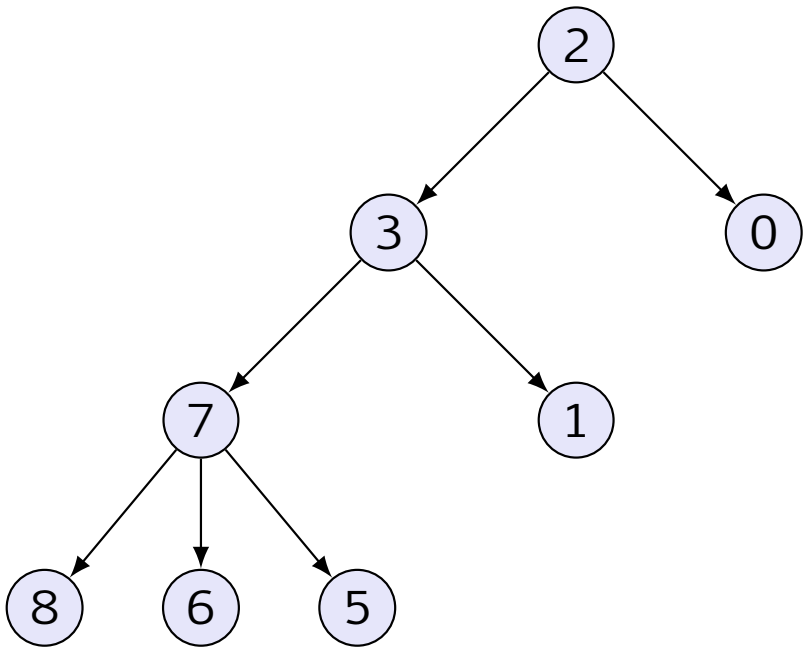
6
5
1
0

Éléments déjà traités :
2, 3, 7, 8

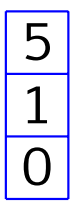
Parcours en profondeur

Illustration





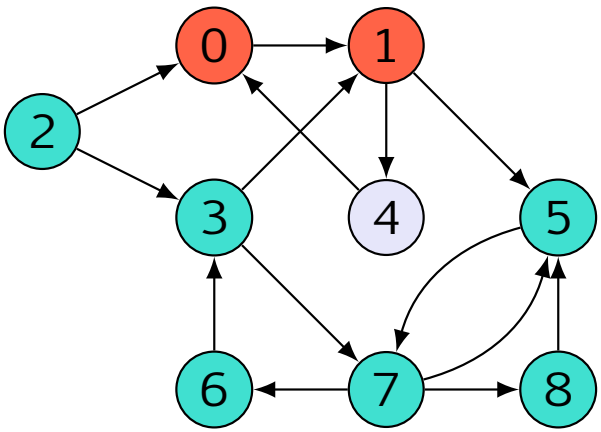
Éléments à traiter :

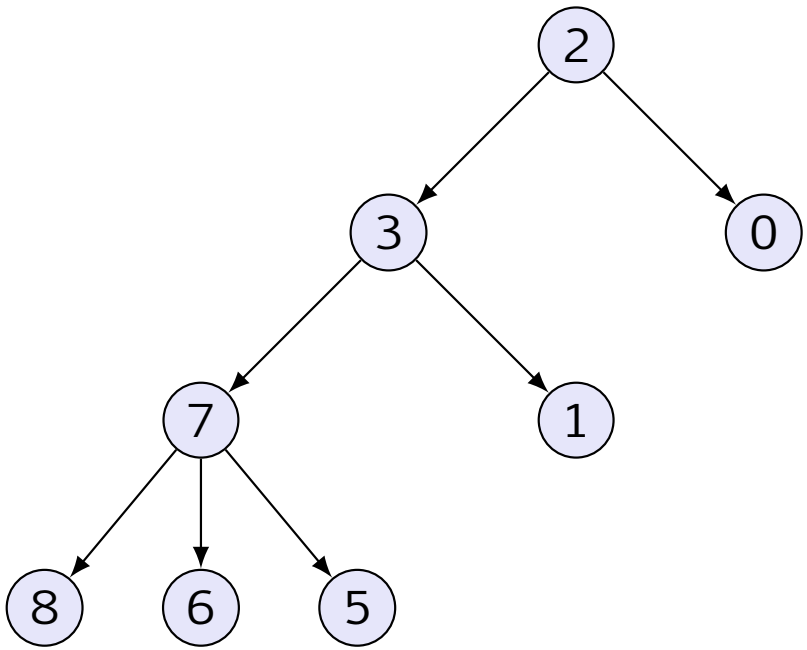


Éléments déjà traités :
2, 3, 7, 8, 6

Parcours en profondeur

Illustration





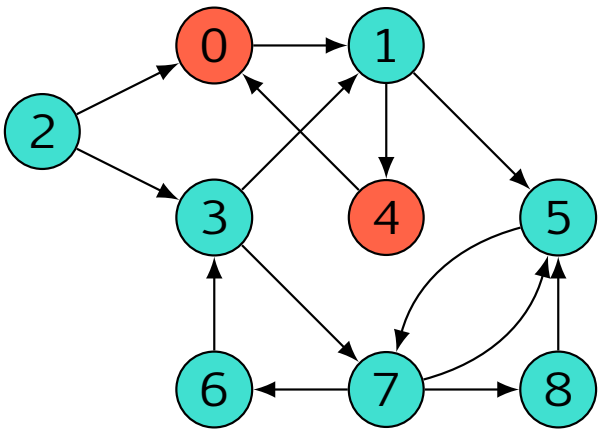
Éléments à traiter :

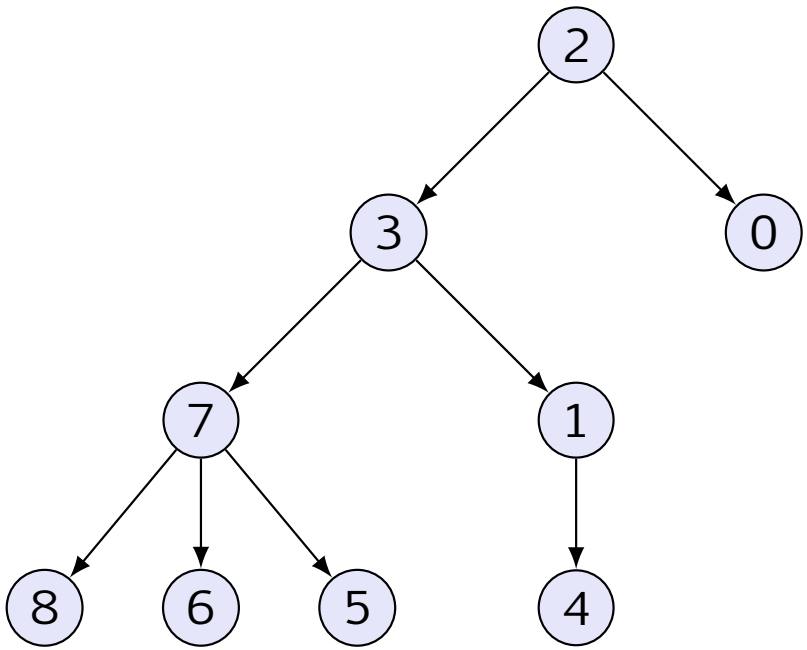
1
0

Éléments déjà traités :
2, 3, 7, 8, 6, 5

Parcours en profondeur

Illustration





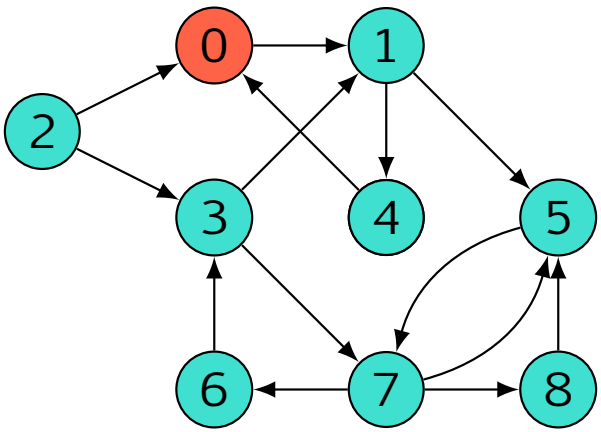
Éléments à traiter :

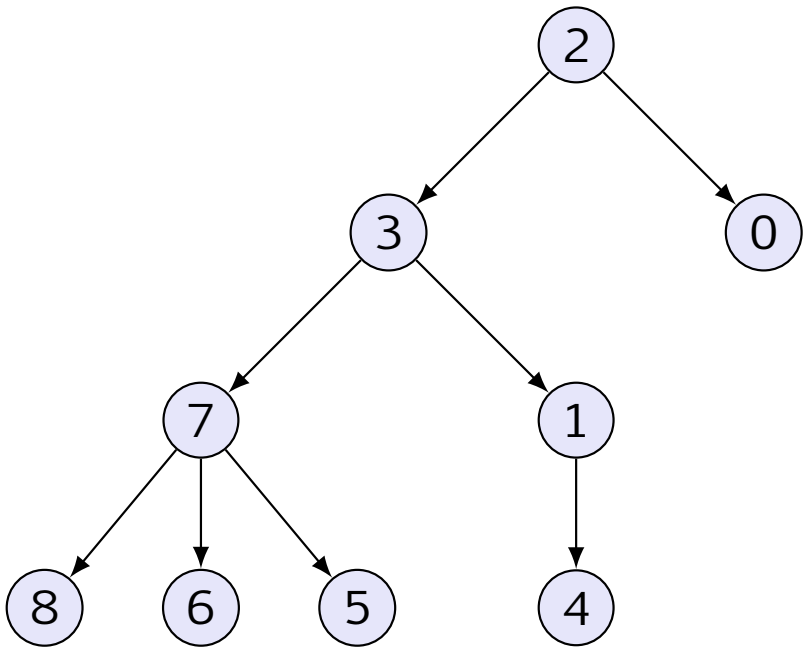
4
0

Éléments déjà traités :
2, 3, 7, 8, 6, 5, 1

Parcours en profondeur

Illustration



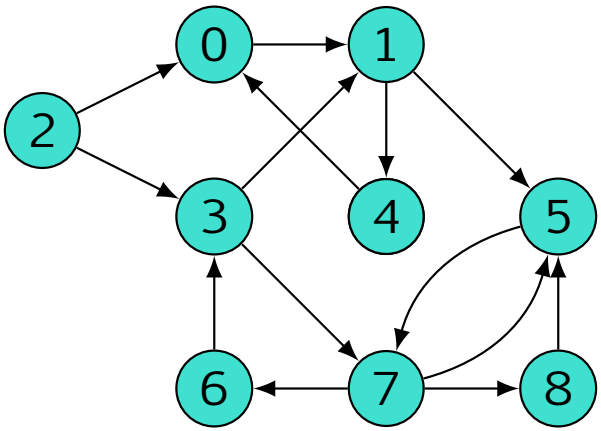


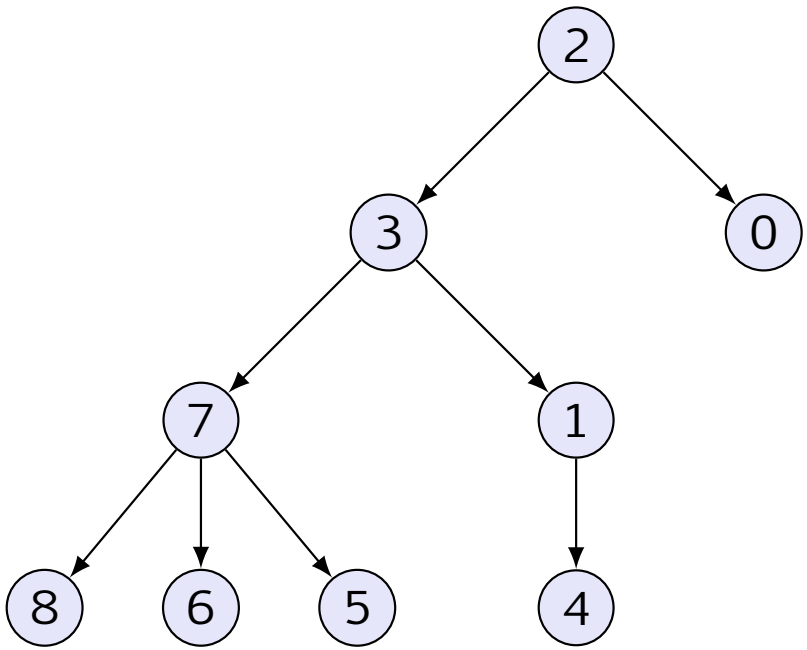
Éléments à traiter : 0

Éléments déjà traités :
2, 3, 7, 8, 6, 5, 1, 4

Parcours en profondeur

Illustration





Éléments à traiter :

Éléments déjà traités :
2, 3, 7, 8, 6, 5, 1, 4, 0

Parcours en profondeur

Version récursive

Notons enfin que l'usage d'une pile laisse présager l'existence d'un algorithme récursif pour le DFS :

```
let dfs_rec g s =
  let dejavu = make_vect (vect_length g) false in
  let rec aux = function
    | s when dejavu.(s) -> ()
    | s                 -> dejavu.(s) <- true ;
                           traitement s ;
                           do_list aux g.(s)
  in aux s ;;
```

Calcul des composantes connexes

d'un graphe non orienté

Les sommets atteints correspondent à la composante connexe du sommet initial. Pour obtenir toutes les composantes connexes d'un graphe, il suffit de reprendre un nouveau parcours à partir d'un sommet non encore atteint, tant qu'il en existe.

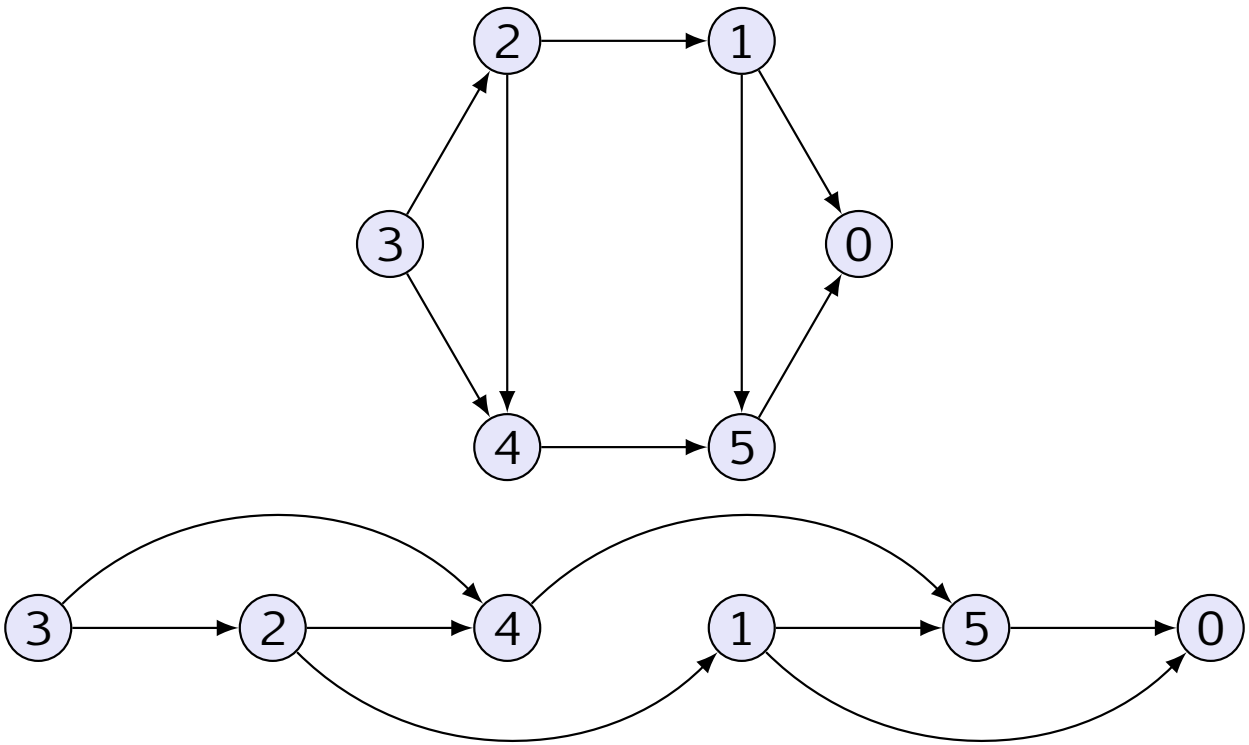
```
let liste_composantes g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           it_list dfs (s::lst) g.(s)
  and aux comp = function
    | s when s = vect_length g -> comp
    | s when dejavu.(s)        -> aux comp (s+1)
    | s                        -> aux ((dfs [] s)::comp) (s+1)
  in aux [] 0 ;;
```

lst est un accumulateur qui transporte les sommets faisant partie de la composante connexe en cours d'exploration.

comp est un accumulateur qui transporte les composantes connexes déjà trouvées.

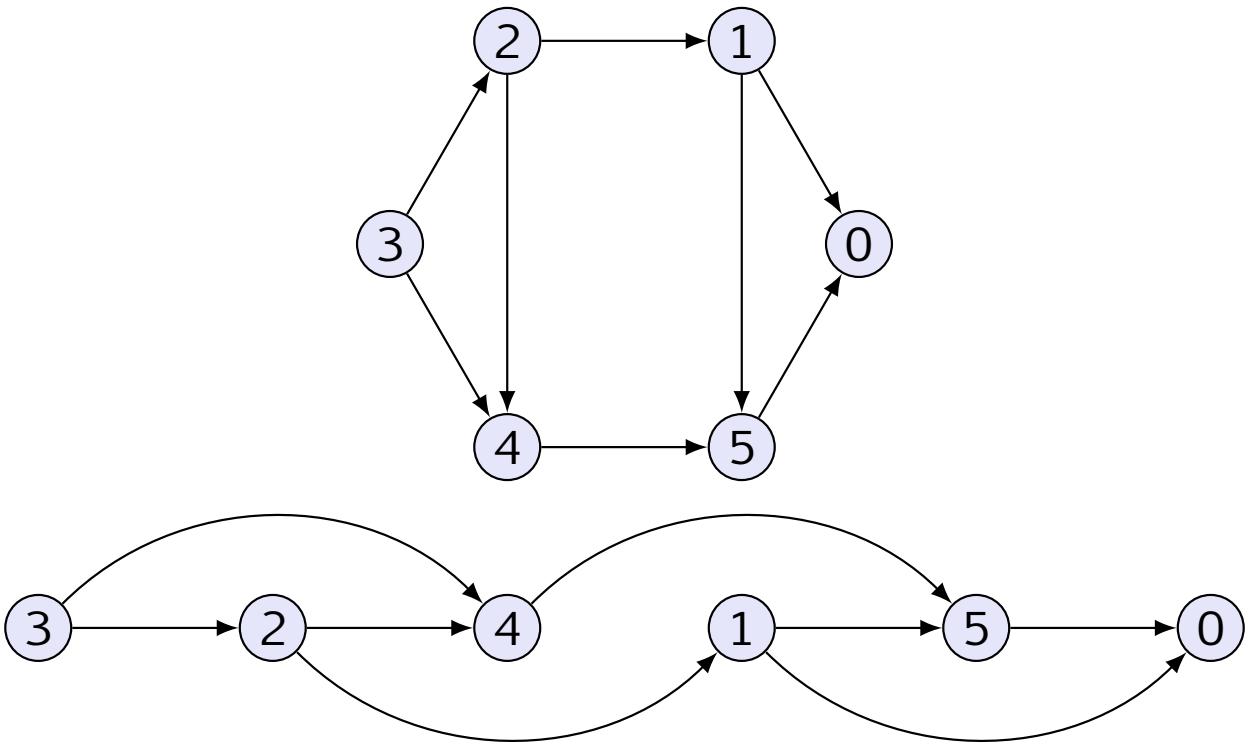
Tri topologique

On considère un graphe orienté acyclique $G = (V, E)$ dont les arcs orientés définissent un ordre partiel sur les sommets. Un tri topologique des sommets prolonge cet ordre en un ordre total.



Tri topologique

On considère un graphe orienté acyclique $G = (V, E)$ dont les arcs orientés définissent un ordre partiel sur les sommets. Un tri topologique des sommets prolonge cet ordre en un ordre total.



À partir de chaque sommet vierge on effectue un parcours en profondeur ; une fois le parcours achevé ce sommet est inséré en tête d'une liste chaînée. On réitère ce procédé jusqu'à exhaustion des sommets vierges.

Tri topologique

```
let tri_topologique g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           s::(it_list dfs lst g.(s))
  and aux ord = function
    | s when s = vect_length g -> ord
    | s when dejavu.(s)        -> aux ord (s+1)
    | s                        -> aux (dfs ord s) (s+1)
  in aux [] 0 ;;
```

Tri topologique

```
let tri_topologique g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           s::(it_list dfs lst g.(s))
  and aux ord = function
    | s when s = vect_length g -> ord
    | s when dejavu.(s)        -> aux ord (s+1)
    | s                        -> aux (dfs ord s) (s+1)
  in aux [] 0 ;;
```

S'il existe un arc reliant le sommet a au sommet b alors a rentre après b dans la liste chaînée.

Tri topologique

```
let tri_topologique g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           s::(it_list dfs lst g.(s))
  and aux ord = function
    | s when s = vect_length g -> ord
    | s when dejavu.(s)        -> aux ord (s+1)
    | s                        -> aux (dfs ord s) (s+1)
  in aux [] 0 ;;
```

S'il existe un arc reliant le sommet a au sommet b alors a rentre après b dans la liste chaînée.

On considère le moment où a est vu pour la première fois et ses voisins examinés. À ce moment a n'est pas encore dans la liste chaînée.

Tri topologique

```
let tri_topologique g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           s::(it_list dfs lst g.(s))
  and aux ord = function
    | s when s = vect_length g -> ord
    | s when dejavu.(s)        -> aux ord (s+1)
    | s                        -> aux (dfs ord s) (s+1)
  in aux [] 0 ;;
```

S'il existe un arc reliant le sommet a au sommet b alors a rentre après b dans la liste chaînée.

On considère le moment où a est vu pour la première fois et ses voisins examinés. À ce moment a n'est pas encore dans la liste chaînée.

- Si b n'a pas encore été vu, le parcours en profondeur se poursuit à partir de b ; une fois ce dernier achevé b rentre dans la liste chaînée, et a y rentrera plus tard.

Tri topologique

```
let tri_topologique g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           s::(it_list dfs lst g.(s))
  and aux ord = function
    | s when s = vect_length g -> ord
    | s when dejavu.(s)        -> aux ord (s+1)
    | s                        -> aux (dfs ord s) (s+1)
  in aux [] 0 ;;
```

S'il existe un arc reliant le sommet a au sommet b alors a rentre après b dans la liste chaînée.

On considère le moment où a est vu pour la première fois et ses voisins examinés. À ce moment a n'est pas encore dans la liste chaînée.

- Si b a déjà été vu et est déjà rentré dans la liste chaînée, le résultat est acquis.

Tri topologique

```
let tri_topologique g =
  let dejavu = make_vect (vect_length g) false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           s::(it_list dfs lst g.(s))
  and aux ord = function
    | s when s = vect_length g -> ord
    | s when dejavu.(s)        -> aux ord (s+1)
    | s                        -> aux (dfs ord s) (s+1)
  in aux [] 0 ;;
```

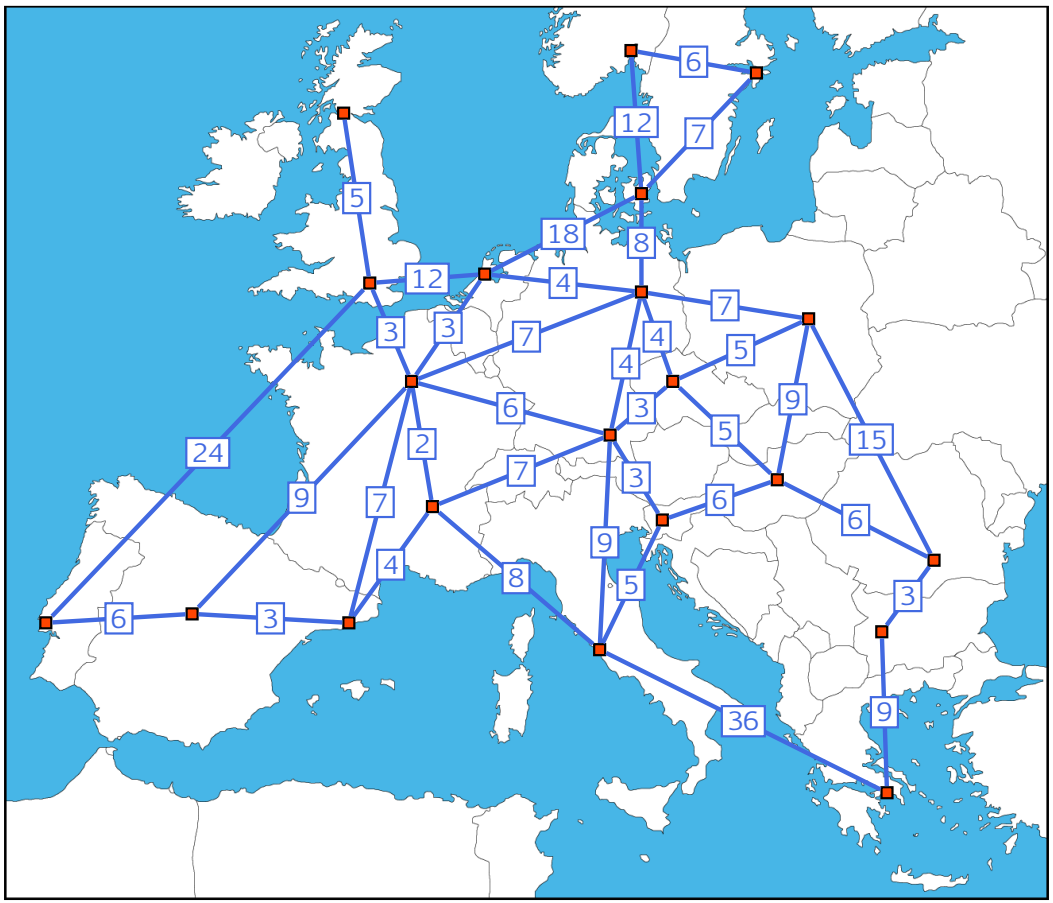
S'il existe un arc reliant le sommet a au sommet b alors a rentre après b dans la liste chaînée.

On considère le moment où a est vu pour la première fois et ses voisins examinés. À ce moment a n'est pas encore dans la liste chaînée.

- Si b a déjà été vu mais n'est pas encore dans la liste chaînée, l'algorithme est en train de parcourir une branche issue de b, avec pour conséquence l'existence d'un chemin reliant b à a. Puisque b est voisin de a, ceci implique l'existence d'un cycle dans G, ce qui est exclu.

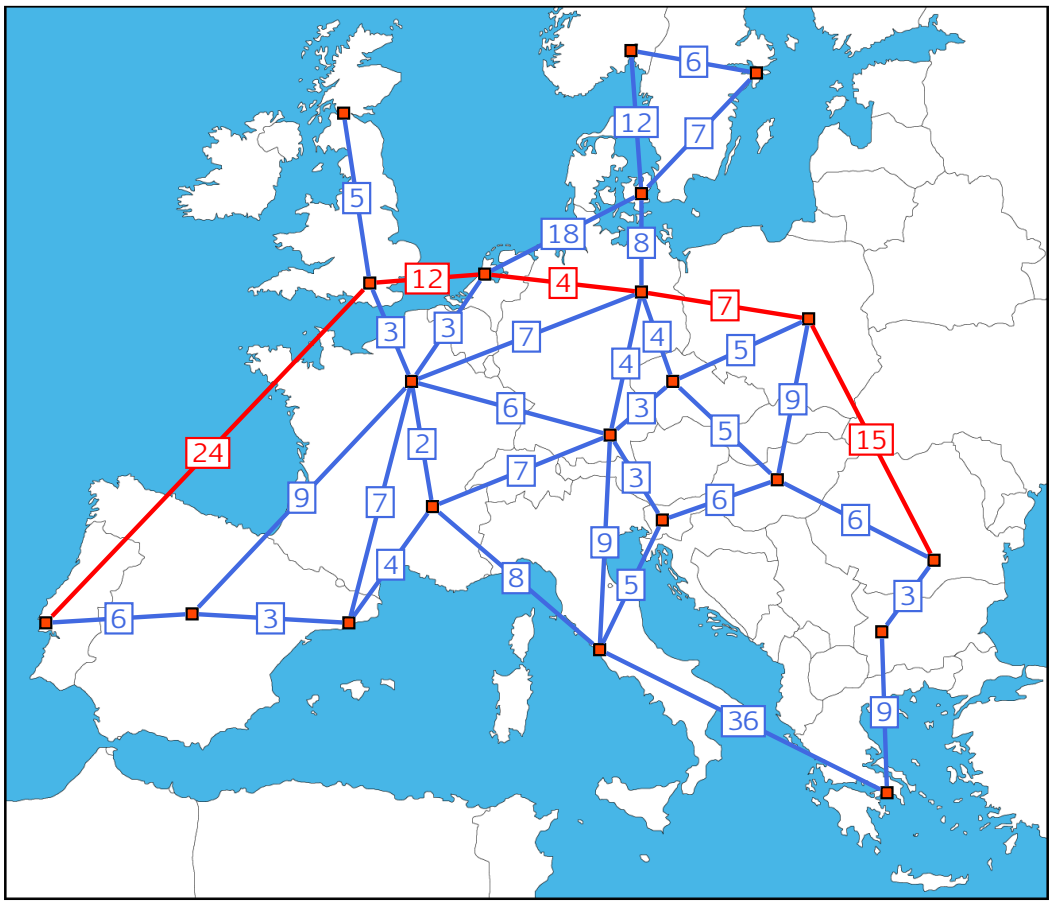
Plus court chemin

Dans de nombreux problèmes s'ajoute à chaque arête une pondération ; on définit dans ce cas le poids d'un chemin : c'est la somme des poids des arêtes qui le composent.



Plus court chemin

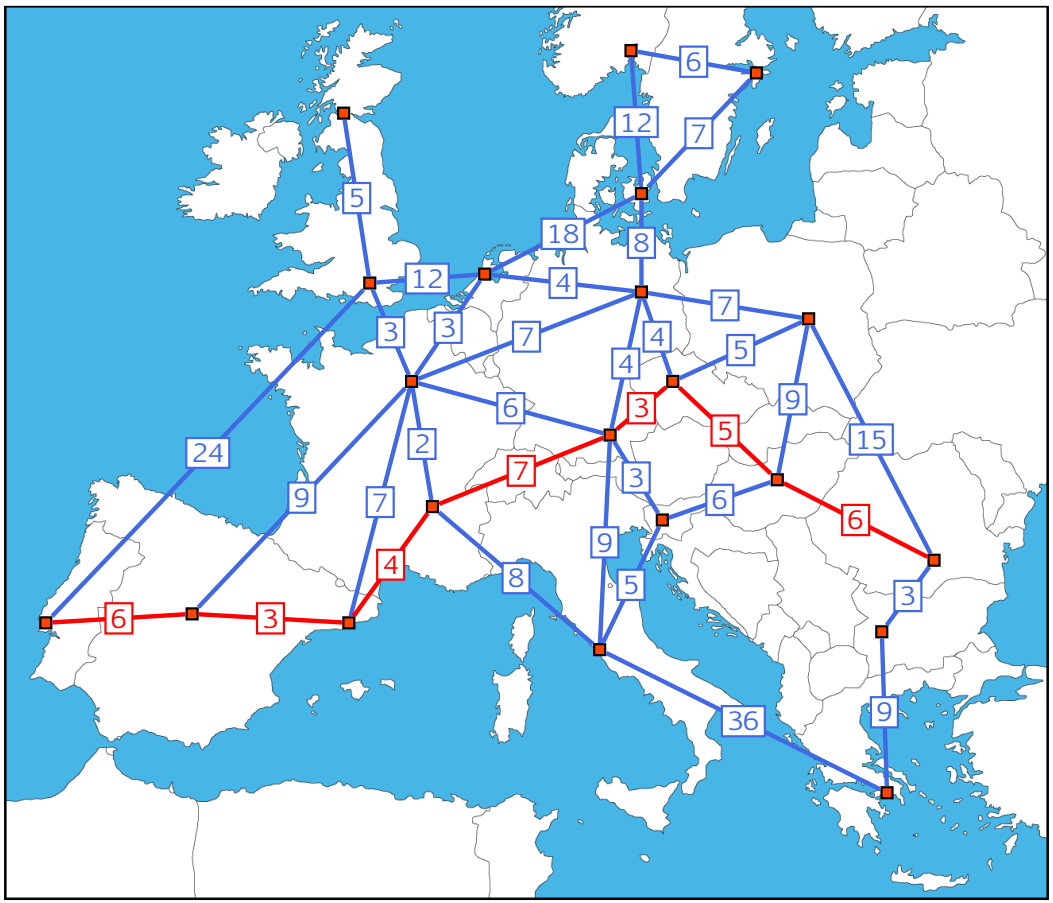
Dans de nombreux problèmes s'ajoute à chaque arête une pondération ; on définit dans ce cas le poids d'un chemin : c'est la somme des poids des arêtes qui le composent.



Le trajet Lisbonne - Londres - Amsterdam - Berlin - Varsovie - Bucarest a un poids égal à 62.

Plus court chemin

Dans de nombreux problèmes s'ajoute à chaque arête une pondération ; on définit dans ce cas le poids d'un chemin : c'est la somme des poids des arêtes qui le composent.



Le trajet Lisbonne - Madrid - Barcelone - Lyon - Munich - Prague - Budapest - Bucarest a un poids égal à 34.

Plus court chemin

Étant donné un graphe $G = (V, E)$, on appelle pondération une application $w : E \to \mathbb{R}$, et on dit que $w(a, b)$ est le poids de l'arête (a, b).
En général on prolonge w sur $V \times V$ en posant : $w(a, b) = 0$ si $a = b$ et $w(a, b) = +\infty$ si $(a, b) \notin E$.
Le poids d'un chemin est la somme des poids des arêtes qui le composent.
On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b.

Plus court chemin

Étant donné un graphe $G = (V, E)$, on appelle pondération une application $w : E \to \mathbb{R}$, et on dit que $w(a, b)$ est le poids de l'arête (a, b).
En général on prolonge w sur $V \times V$ en posant : $w(a, b) = 0$ si $a = b$ et $w(a, b) = +\infty$ si $(a, b) \notin E$.
Le poids d'un chemin est la somme des poids des arêtes qui le composent.
On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b.

Attention à la présence de poids négatifs

S'il existe un chemin menant de a et b et comprenant un circuit fermé de poids strictement négatif, il convient de poser $\delta(a, b) = -\infty$.

Plus court chemin

Étant donné un graphe $G = (V, E)$, on appelle pondération une application $w : E \to \mathbb{R}$, et on dit que $w(a, b)$ est le poids de l'arête (a, b).
En général on prolonge w sur $V \times V$ en posant : $w(a, b) = 0$ si $a = b$ et $w(a, b) = +\infty$ si $(a, b) \notin E$.
Le poids d'un chemin est la somme des poids des arêtes qui le composent.
On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b.

Il existe trois problème de plus courts chemins :

1. calculer le chemin de poids minimal entre une source a et une destination b ;

Plus court chemin

Étant donné un graphe $G = (V, E)$, on appelle pondération une application $w : E \to \mathbb{R}$, et on dit que $w(a, b)$ est le poids de l'arête (a, b).
En général on prolonge w sur $V \times V$ en posant : $w(a, b) = 0$ si $a = b$ et $w(a, b) = +\infty$ si $(a, b) \notin E$.
Le poids d'un chemin est la somme des poids des arêtes qui le composent.
On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b.

Il existe trois problème de plus courts chemins :

1. calculer le chemin de poids minimal entre une source a et une destination b ;
2. calculer les chemins de poids minimal entre une source a et tout autre sommet du graphe ;

Plus court chemin

Étant donné un graphe $G = (V, E)$, on appelle pondération une application $w : E \to \mathbb{R}$, et on dit que $w(a, b)$ est le poids de l'arête (a, b).
En général on prolonge w sur $V \times V$ en posant : $w(a, b) = 0$ si $a = b$ et $w(a, b) = +\infty$ si $(a, b) \notin E$.
Le poids d'un chemin est la somme des poids des arêtes qui le composent.
On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b.

Il existe trois problème de plus courts chemins :

1. calculer le chemin de poids minimal entre une source a et une destination b ;
2. calculer les chemins de poids minimal entre une source a et tout autre sommet du graphe ;
3. calculer tous les chemins de poids minimal entre deux sommets quelconques du graphe.

Plus court chemin

Étant donné un graphe $G = (V, E)$, on appelle pondération une application $w : E \to \mathbb{R}$, et on dit que $w(a, b)$ est le poids de l'arête (a, b).
En général on prolonge w sur $V \times V$ en posant : $w(a, b) = 0$ si $a = b$ et $w(a, b) = +\infty$ si $(a, b) \notin E$.
Le poids d'un chemin est la somme des poids des arêtes qui le composent.
On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b.

Il existe trois problème de plus courts chemins :

1. calculer le chemin de poids minimal entre une source a et une destination b ;
2. calculer les chemins de poids minimal entre une source a et tout autre sommet du graphe ;
3. calculer tous les chemins de poids minimal entre deux sommets quelconques du graphe.

L'algorithme de Floyd-Warshall résout le 3^e problème s'il n'existe pas de cycle de poids négatif.
L'algorithme de Dijkstra résout le 2^e problème lorsque les poids sont tous positifs.

Plus court chemin

Étant donné un graphe $G = (V, E)$, on appelle pondération une application $w : E \to \mathbb{R}$, et on dit que $w(a, b)$ est le poids de l'arête (a, b).
En général on prolonge w sur $V \times V$ en posant : $w(a, b) = 0$ si $a = b$ et $w(a, b) = +\infty$ si $(a, b) \notin E$.
Le poids d'un chemin est la somme des poids des arêtes qui le composent.
On notera $\delta(a, b)$ le poids du plus court chemin allant de a à b.

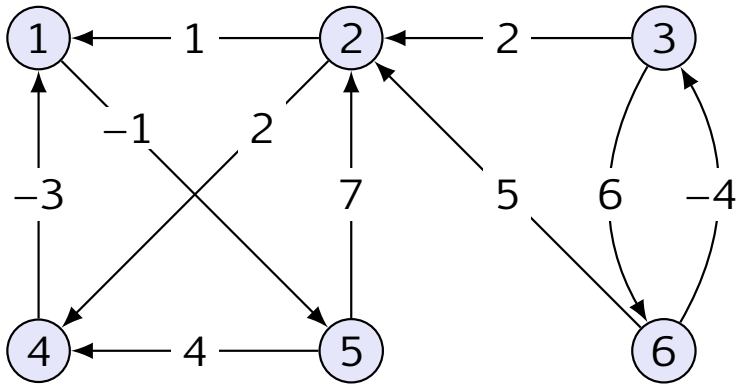
Principe de sous-optimalité :

Si $a \rightsquigarrow b$ est un plus court chemin qui passe par c, alors $a \rightsquigarrow c$ et $c \rightsquigarrow b$ sont eux aussi des plus courts chemins.

S'il existait un chemin plus court entre a et c, il suffirait de le suivre lors du trajet entre a et b pour contredire le caractère minimal du trajet $a \rightsquigarrow b$.

Algorithme de Floyd-Warshall

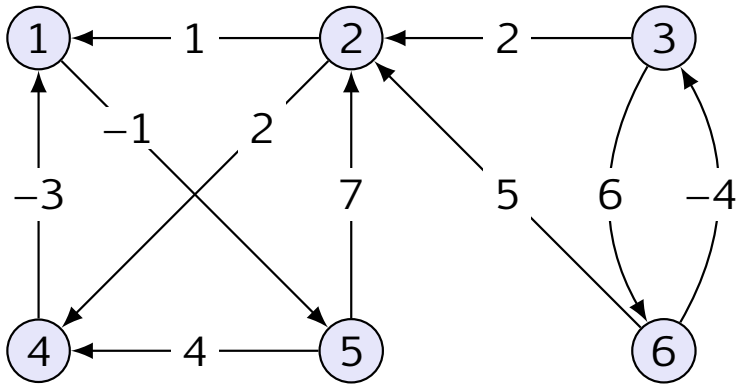
La matrice d'adjacence désigne désormais la matrice $M = (w(v_i, v_j))$.



$$M = \begin{pmatrix} 0 & +\infty & +\infty & +\infty & -1 & +\infty \\ 1 & 0 & +\infty & 2 & +\infty & +\infty \\ +\infty & 2 & 0 & +\infty & +\infty & 6 \\ -3 & +\infty & +\infty & 0 & +\infty & +\infty \\ +\infty & 7 & +\infty & 4 & 0 & +\infty \\ +\infty & 5 & -4 & +\infty & +\infty & 0 \end{pmatrix}$$

Algorithme de Floyd-Warshall

La matrice d'adjacence désigne désormais la matrice $M = (w(v_i, v_j))$.



$$M = \begin{pmatrix} 0 & +\infty & +\infty & +\infty & -1 & +\infty \\ 1 & 0 & +\infty & 2 & +\infty & +\infty \\ +\infty & 2 & 0 & +\infty & +\infty & 6 \\ -3 & +\infty & +\infty & 0 & +\infty & +\infty \\ +\infty & 7 & +\infty & 4 & 0 & +\infty \\ +\infty & 5 & -4 & +\infty & +\infty & 0 \end{pmatrix}$$

L'algorithme de Floyd-Warshall consiste à calculer la suite de matrices $M^{(k)}, 0 \leqslant k \leqslant n$ avec $M^{(0)} = M$ et :

$$\forall k < n, \quad \forall (i,j) \in \mathbb{N}^2, \qquad m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

Algorithme de Floyd-Warshall

La matrice d'adjacence désigne désormais la matrice $M = (w(v_i, v_j))$.

$$m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

Si G ne contient pas de cycle de poids strictement négatif, $m_{ij}^{(k)}$ est égal au poids du chemin minimal reliant v_i à v_j et ne passant que par des sommets de la liste $v_1, v_2, \ldots, v_k$.
En particulier, $m_{ij}^{(n)}$ est le poids minimal d'un chemin reliant v_i à v_j.

Algorithme de Floyd-Warshall

La matrice d'adjacence désigne désormais la matrice $M = (w(v_i, v_j))$.

$$m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

Si G ne contient pas de cycle de poids strictement négatif, $m_{ij}^{(k)}$ est égal au poids du chemin minimal reliant v_i à v_j et ne passant que par des sommets de la liste $v_1, v_2, \ldots, v_k$.
En particulier, $m_{ij}^{(n)}$ est le poids minimal d'un chemin reliant v_i à v_j.

- Si $k = 0$, $m_{ij}^{(0)} = m_{ij}$ est le poids du chemin minimal reliant v_i à v_j sans passer par aucun autre sommet.

Algorithme de Floyd-Warshall

La matrice d'adjacence désigne désormais la matrice $M = (w(v_i, v_j))$.

$$m_{ij}^{(k+1)} = \min\left(m_{ij}^{(k)}, m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}\right).$$

Si G ne contient pas de cycle de poids strictement négatif, $m_{ij}^{(k)}$ est égal au poids du chemin minimal reliant v_i à v_j et ne passant que par des sommets de la liste $v_1, v_2, \ldots, v_k$.
En particulier, $m_{ij}^{(n)}$ est le poids minimal d'un chemin reliant v_i à v_j.

- Si $k < n$, considérons un chemin $v_i \rightsquigarrow v_j$ ne passant que par les sommets $v_1, \ldots, v_{k+1}$ et de poids minimal.

  Si ce chemin ne passe pas par v_{k+1}, son poids total est égal à $m_{ij}^{(k)}$.
  Si ce chemin passe par v_{k+1}, les chemins $v_i \rightsquigarrow v_{k+1}$ et $v_{k+1} \rightsquigarrow v_j$ sont minimaux (le principe de sous-optimalité) et ne passent que par des sommets de $v_1, \ldots, v_k$ donc son poids vaut $m_{i,k+1}^{(k)} + m_{k+1,j}^{(k)}$.

  Donc $m_{ij}^{(k+1)}$ est le poids minimal d'un plus court chemin reliant v_i et v_j et ne passant que par des sommets de la liste $v_1, \ldots, v_{k+1}$.

Algorithme de Floyd-Warshall

Mise en œuvre pratique

On suppose les poids à valeurs entières :

```
type poids = Inf | P of int ;;

let som = fun
  | Inf _       -> Inf
  | _ Inf       -> Inf
  | (P a) (P b) -> P (a + b) ;;

let mini = fun
  | Inf x       -> x
  | x Inf       -> x
  | (P a) (P b) -> P (min a b) ;;

let inferieur = fun
  | Inf _       -> false
  | _ Inf       -> true
  | (P a) (P b) -> a < b ;;
```

Algorithme de Floyd-Warshall

Mise en œuvre pratique

```
let floydwarshall w =
  let n = vect_length w in
  let m = make_matrix n n Inf in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- w.(i).(j)
    done
  done ;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        m.(i).(j) <- mini m.(i).(j) (som m.(i).(k) m.(k).(j))
      done
    done
  done ;
  m ;;
```

Le calcul des matrices $M^{(k)}$ peut se faire en place puisque

$$m_{i,k+1}^{(k+1)} = m_{i,k+1}^{(k)} \quad \text{et} \quad m_{k+1,j}^{(k+1)} = m_{k+1,j}^{(k)}.$$

Algorithme de Floyd-Warshall

Mise en œuvre pratique

```
let floydwarshall w =
  let n = vect_length w in
  let m = make_matrix n n Inf in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- w.(i).(j)
    done
  done ;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        m.(i).(j) <- mini m.(i).(j) (som m.(i).(k) m.(k).(j))
      done
    done
  done ;
  m ;;
```

Coût temporel : $\Theta(n^3)$

Coût spatial : $\Theta(n^2)$.

Algorithme de Floyd-Warshall

Mémorisation des plus courts chemins

```
let pluscourtschemins w =
  let n = vect_length w in
  let m = make_matrix n n Inf
  and c = make_matrix n n [] in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- w.(i).(j)
    done
  done ;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        let l = som m.(i).(k) m.(k).(j) in
        if inferieur l m.(i).(j) then
          (m.(i).(j) <- l ; c.(i).(j) <- c.(i).(k)@[k+1]@c.(k).(j))
      done
    done
  done ;
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if i <> j && m.(i).(j) <> Inf then c.(i).(j) <- [i+1]@c.(i).(j)@[j+1]
    done
  done ;
  c ;;
```

Fermeture transitive d'un graphe

Algorithme de Warshall

On souhaite déterminer si deux sommets a et b peuvent être reliés par un chemin allant de a à b.
On utilise la matrice d'adjacence en utilisant les valeurs booléennes **`true`** pour dénoter l'existence d'une arête et **`false`** pour en marquer l'absence, puis on applique l'algorithme de Floyd-Warshall en utilisant la relation de récurrence :

$$m_{ij}^{(k+1)} = m_{ij}^{(k)} \text{ ou } \left(m_{i,k+1}^{(k)} \text{ et } m_{k+1,j}^{(k)} \right).$$

$m_{ij}^{(k)}$ dénote l'existence ou non d'un chemin reliant v_i et v_j en ne passant que par des sommets de $\{v_1, \ldots, v_k\}$.

Fermeture transitive d'un graphe

Algorithme de Warshall

$$m_{ij}^{(k+1)} = m_{ij}^{(k)} \text{ ou } \left(m_{i,k+1}^{(k)} \text{ et } m_{k+1,j}^{(k)} \right).$$

```
let warshall w =
  let n = vect_length w in
  let m = make_matrix n n false in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      m.(i).(j) <- w.(i).(j) = 1
    done
  done ;
  for k = 0 to n-1 do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        m.(i).(j) <- m.(i).(j) || (m.(i).(k) && m.(k).(j))
      done
    done
  done ;
  m ;;
```

Algorithme de Dijkstra

Important : on suppose tous les poids positifs.

Algorithme de Dijkstra

Important : on suppose tous les poids positifs.
À partir d'une source $s \in V$ on fait évoluer une partition $(S, \overline{S})$ de $[\![1, n]\!]$ et un tableau d de sorte que :

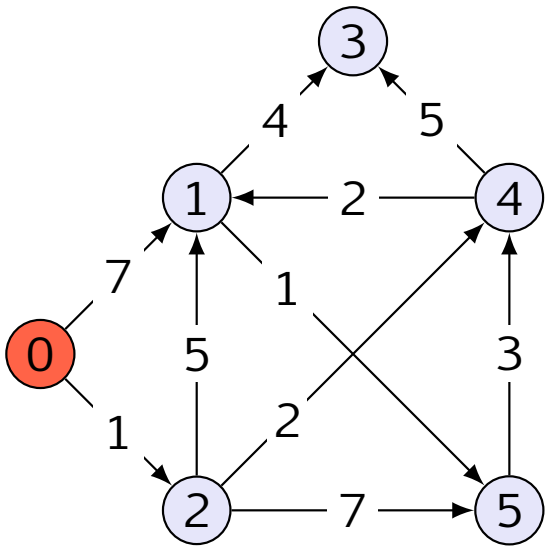
- $\forall v \in S,\ d_v = \delta(s, v)$;
- $\forall v \in \overline{S},\ d_v = \delta_S(s, v)$ (on se restreint aux chemins tracés dans S).

function Dijkstra*(sommet : s)*
  $S \leftarrow \{s\}$
  for all $v \in V$ ***do***
    $d_v \leftarrow w(s, v)$
  while $\overline{S} \neq \emptyset$ ***do***
    Soit $u \in \overline{S} \;\Big|\; d_u = \min\left\{d_v \;\middle|\; v \in \overline{S}\right\}$
    $S \leftarrow S \cup \{u\}$
    for $v \in \overline{S}$ ***do***
      $d_v \leftarrow \min\Big(d_v, d_u + w(u, v)\Big)$
  return d

À chaque itération on choisit $u \in \overline{S}$ tel que d_u est minimal, on le transfere dans S, et on modifie le tableau d en conséquence.

Algorithme de Dijkstra

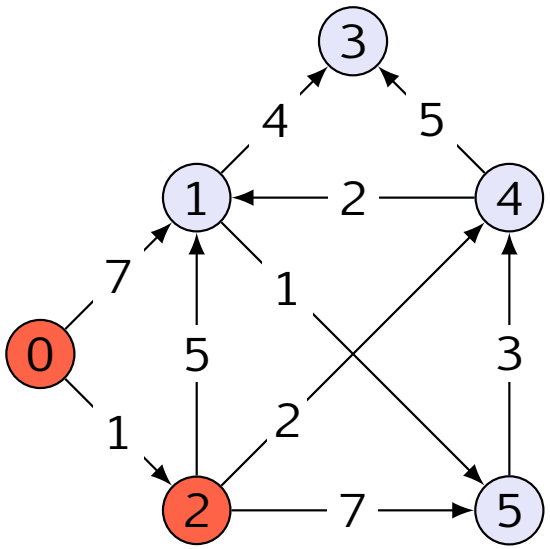
Exemple



S	0	1	2	3	4	5
{0}	·	7	1	∞	∞	∞

Algorithme de Dijkstra

Exemple

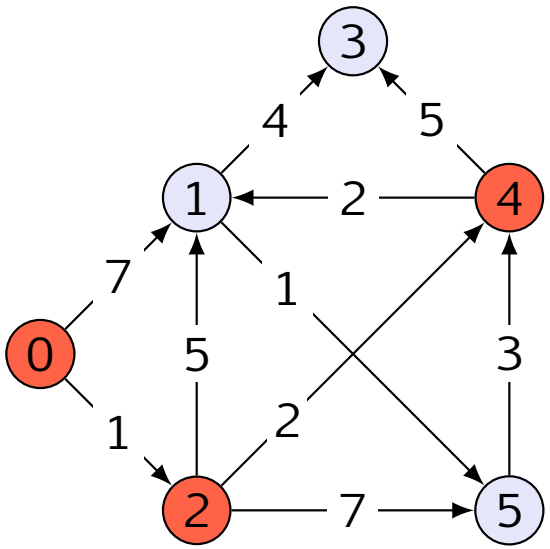


S	0	1	2	3	4	5
$\{0\}$	·	7	1	∞	∞	∞
$\{0,2\}$	·	6	·	∞	3	8

- $\delta(0,2) = 1$, on trouve un chemin plus court pour 1, 4 et 5.

Algorithme de Dijkstra

Exemple

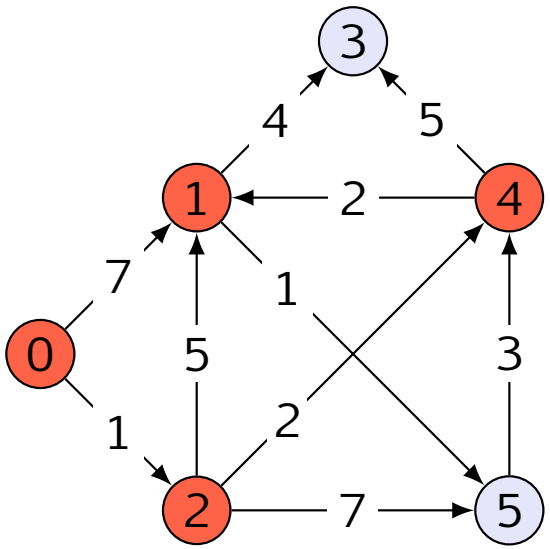


S	0	1	2	3	4	5
$\{0\}$	·	7	1	∞	∞	∞
$\{0,2\}$	·	6	·	∞	3	8
$\{0,2,4\}$	·	5	·	8	·	8

- $\delta(0,2) = 1$, on trouve un chemin plus court pour 1, 4 et 5.
- $\delta(0,4) = 3$, on trouve un chemin plus court pour 1 et 3.

Algorithme de Dijkstra

Exemple

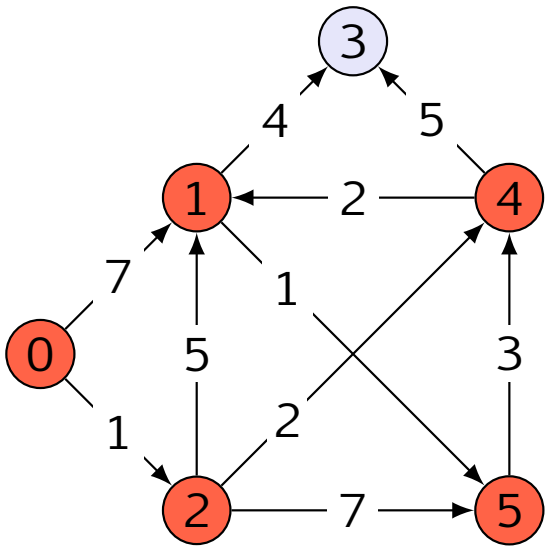


S	0	1	2	3	4	5
$\{0\}$	·	7	1	∞	∞	∞
$\{0,2\}$	·	6	·	∞	3	8
$\{0,2,4\}$	·	5	·	8	·	8
$\{0,2,4,1\}$	·	·	·	8	·	6

- $\delta(0,2) = 1$, on trouve un chemin plus court pour 1, 4 et 5.
- $\delta(0,4) = 3$, on trouve un chemin plus court pour 1 et 3.
- $\delta(0,1) = 5$, on trouve un chemin plus court pour 5.

Algorithme de Dijkstra

Exemple

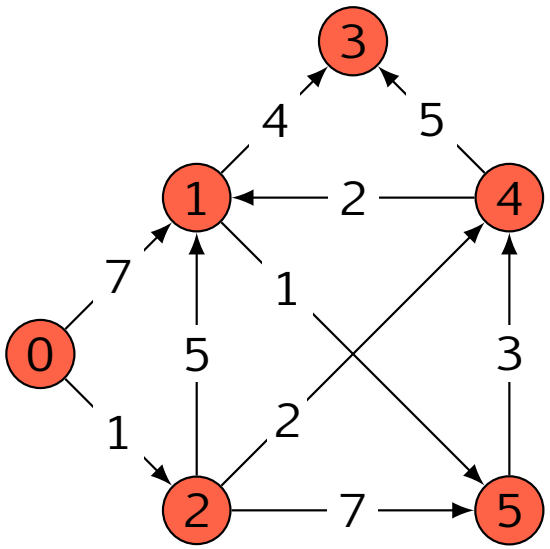


S	0	1	2	3	4	5
$\{0\}$	·	7	1	∞	∞	∞
$\{0,2\}$	·	6	·	∞	3	8
$\{0,2,4\}$	·	5	·	8	·	8
$\{0,2,4,1\}$	·	·	·	8	·	6
$\{0,2,4,1,5\}$	·	·	·	8	·	·

- $\delta(0,2) = 1$, on trouve un chemin plus court pour 1, 4 et 5.
- $\delta(0,4) = 3$, on trouve un chemin plus court pour 1 et 3.
- $\delta(0,1) = 5$, on trouve un chemin plus court pour 5.
- $\delta(0,5) = 6$.

Algorithme de Dijkstra

Exemple



S	0	1	2	3	4	5
$\{0\}$	·	7	1	∞	∞	∞
$\{0,2\}$	·	6	·	∞	3	8
$\{0,2,4\}$	·	5	·	8	·	8
$\{0,2,4,1\}$	·	·	·	8	·	6
$\{0,2,4,1,5\}$	·	·	·	8	·	·
$\{0,2,4,1,5,3\}$	·	·	·	·	·	·

- $\delta(0,2) = 1$, on trouve un chemin plus court pour 1, 4 et 5.
- $\delta(0,4) = 3$, on trouve un chemin plus court pour 1 et 3.
- $\delta(0,1) = 5$, on trouve un chemin plus court pour 5.
- $\delta(0,5) = 6$.
- $\delta(0,3) = 8$.

Algorithme de Dijkstra

Preuve de validité

On prouve par récurrence sur $|S|$ l'invariant suivant :

$$\forall u \in S, \quad d_u = \delta(s,u)$$

$$\forall v \in \overline{S}, \quad d_v = \min\left\{ d_u + w(u,v) \;\middle|\; u \in S \right\}$$

Algorithme de Dijkstra

Preuve de validité

On prouve par récurrence sur $|S|$ l'invariant suivant :

$$\forall u \in S, \quad d_u = \delta(s,u)$$

$$\forall v \in \overline{S}, \quad d_v = \min\left\{ d_u + w(u,v) \,\middle|\, u \in S \right\}$$

- Lorsque $|S| = 1$, $S = \{s\}$, $d_s = 0$ et $\forall v \neq s$, $d_v = w(s,v)$.

Algorithme de Dijkstra

Preuve de validité

On prouve par récurrence sur $|S|$ l'invariant suivant :

$$\forall u \in S, \quad d_u = \delta(s,u)$$

$$\forall v \in \overline{S}, \quad d_v = \min\left\{d_u + w(u,v) \,\middle|\, u \in S\right\}$$

- Lorsque $|S| > 1$ on distingue trois cas :

1. $u \in S$: dans ce cas $d_u = \delta(s,u)$ n'est pas modifié.

Algorithme de Dijkstra

Preuve de validité

On prouve par récurrence sur $|S|$ l'invariant suivant :

$$\forall u \in S, \quad d_u = \delta(s,u)$$

$$\forall v \in \overline{S}, \quad d_v = \min\left\{d_u + w(u,v) \,\middle|\, u \in S\right\}$$

- Lorsque $|S| > 1$ on distingue trois cas :

1. $u \in S$: dans ce cas $d_u = \delta(s,u)$ n'est pas modifié.
2. u entre dans S : dans ce cas $u \in \overline{S}$ vérifie $\forall v \in \overline{S}$, $d_u \leqslant d_v$ et il s'agit de prouver que $d_u = \delta(s,u)$.
   Notons v le premier sommet de $\overline{S}$ dans un plus court chemin $s \rightsquigarrow u$. Alors $s \rightsquigarrow v$ est un plus court chemin.
   On a $\delta(s,u) \geqslant \delta(s,v)$ (les poids sont positifs) et $d_v = \min\left\{d_x + w(x,v) \,\middle|\, x \in S\right\} = \min\left\{\delta(s,x) + w(x,v) \,\middle|\, x \in S\right\} = \delta(s,v)$ donc $\delta(s,u) \geqslant d_v$. Par ailleurs, $d_u \leqslant d_v$ donc $\delta(s,u) \geqslant d_u$. Mais d_u est le poids d'un chemin menant de s à u donc en définitive $d_u = \delta(s,u)$.

Algorithme de Dijkstra

Preuve de validité

On prouve par récurrence sur $|S|$ l'invariant suivant :

$$\forall u \in S, \quad d_u = \delta(s,u)$$

$$\forall v \in \overline{S}, \quad d_v = \min\left\{d_u + w(u,v) \,\middle|\, u \in S\right\}$$

- Lorsque $|S| > 1$ on distingue trois cas :

1. $u \in S$: dans ce cas $d_u = \delta(s,u)$ n'est pas modifié.
2. u entre dans S : dans ce cas $u \in \overline{S}$ vérifie $\forall v \in \overline{S}, d_u \leqslant d_v$ et il s'agit de prouver que $d_u = \delta(s,u)$.
   Notons v le premier sommet de $\overline{S}$ dans un plus court chemin $s \rightsquigarrow u$. Alors $s \rightsquigarrow v$ est un plus court chemin.
   On a $\delta(s,u) \geqslant \delta(s,v)$ (les poids sont positifs) et $d_v = \min\left\{d_x + w(x,v) \,\middle|\, x \in S\right\} = \min\left\{\delta(s,x) + w(x,v) \,\middle|\, x \in S\right\} = \delta(s,v)$ donc $\delta(s,u) \geqslant d_v$. Par ailleurs, $d_u \leqslant d_v$ donc $\delta(s,u) \geqslant d_u$. Mais d_u est le poids d'un chemin menant de s à u donc en définitive $d_u = \delta(s,u)$.
3. v reste dans $\overline{S}$. Dans ce cas d_v est modifié pour continuer à respecter l'égalité $d_v = \min\left\{d_u + w(u,v) \,\middle|\, u \in S\right\}$.

Algorithme de Dijkstra

Étude de la complexité

Première approche :

- $n-1$ transferts de $\overline{S}$ vers S ;
- à chacun des transferts, un calcul de minimum au sein du tableau d et une modification du dit tableau.

Coût total : $O(n^2)$.

Algorithme de Dijkstra

Étude de la complexité

Première approche :

- $n-1$ transferts de $\overline{S}$ vers S ;
- à chacun des transferts, un calcul de minimum au sein du tableau d et une modification du dit tableau.

Coût total : $\mathrm{O}(n^2)$.

Amélioration possible : utiliser une file de priorité pour représenter $\overline{S}$.

- récupération de l'élément minimal d'un tas-min + reformation du tas $= \mathrm{O}(\log n)$;
- mise à jour d'une valeur de d : $\mathrm{O}(\log n)$.

Coût total $= \mathrm{O}(n \log n) + \mathrm{O}(p \log n) = \mathrm{O}((n+p)\log n)$.

Algorithme de Dijkstra

Étude de la complexité

Première approche :

- $n-1$ transferts de $\overline{S}$ vers S ;
- à chacun des transferts, un calcul de minimum au sein du tableau d et une modification du dit tableau.

Coût total : $\mathrm{O}(n^2)$.

Amélioration possible : utiliser une file de priorité pour représenter $\overline{S}$.

- récupération de l'élément minimal d'un tas-min + reformation du tas $= \mathrm{O}(\log n)$;
- mise à jour d'une valeur de d : $\mathrm{O}(\log n)$.

Coût total $= \mathrm{O}(n \log n) + \mathrm{O}(p \log n) = \mathrm{O}((n+p)\log n)$.

$\longrightarrow$ démarche intéressante dès lors que :

$$p \log n = \mathrm{O}(n^2) \quad \text{soit} \quad p = \mathrm{O}\Big(\frac{n^2}{\log n}\Big).$$

Algorithme de Dijkstra

Mise en œuvre pratique

Nécessité d'accéder en coût constant à chaque liste d'adjacence d'un sommet :

```
type voisin == int list ;;
type graphe == voisin vect ;;
```

Algorithme de Dijkstra

Mise en œuvre pratique

Nécessité d'accéder en coût constant à chaque liste d'adjacence d'un sommet :

```
type voisin == int list ;;
type graphe == voisin vect ;;
```

Représentation de $\overline{S}$ par un tas-min : un vecteur **t** de taille n de type *int vect*. La case d'indice 0 contient l'indice du dernier élément du tas.

Algorithme de Dijkstra

Mise en œuvre pratique

Nécessité d'accéder en coût constant à chaque liste d'adjacence d'un sommet :

```
type voisin == int list ;;
type graphe == voisin vect ;;
```

Représentation de $\overline{S}$ par un tas-min : un vecteur **t** de taille n de type *int vect*. La case d'indice 0 contient l'indice du dernier élément du tas.

Nécessité d'accéder en coût constant à chaque élément du tas : on marque l'emplacement de chaque sommet dans **t** dans un tableau **m**, de sorte que **t.(m.(i)) = i**.

Algorithme de Dijkstra

Mise en œuvre pratique

Nécessité d'accéder en coût constant à chaque liste d'adjacence d'un sommet :

```
type voisin == int list ;;
type graphe == voisin vect ;;
```

Représentation de $\overline{S}$ par un tas-min : un vecteur **t** de taille n de type *int vect*. La case d'indice 0 contient l'indice du dernier élément du tas.

Nécessité d'accéder en coût constant à chaque élément du tas : on marque l'emplacement de chaque sommet dans **t** dans un tableau **m**, de sorte que **t.(m.(i)) = i**.

On suppose données les fonctions :

- **ajoute d t m**, qui ajoute un sommet au tas **t** ;
- **extrait d t m**, qui extrait du tas le sommet de distance minimale ;
- **remonte d t m**, qui reforme le tas à partir d'un indice passé en paramètre lorsque le tableau **d** est modifié.

Algorithme de Dijkstra

Mise en œuvre pratique

```
let dijkstra g w s =
  let n = vect_length g in
  let d = make_vect n Inf
  and t = make_vect n 0 and m = make_vect n 0 in
  for k = 0 to n-1 do
    d.(k) <- w.(s).(k) ;
    if k <> s then ajoute d t m k ;
  done ;
  let rec modif i = function
    | []   -> ()
    | j::q -> let x = som d.(i) w.(i).(j) in
              if inferieur x d.(j) then
                (d.(j) <- x ; remonte d t m m.(j)) ;
              modif i q
  in
  for k = 0 to n-1 do
    let i = extrait d t m in
    modif i g.(i)
  done ;
  d ;;
```

Algorithme de Dijkstra

Détermination du chemin de poids minimal

Pour garder trace du chemin parcouru, on utilise un tableau c qui mémorise i dans c_j lorsque d_j est remplacé par $d_i + w(v_i, v_j)$.

À la fin de l'algorithme c_j contient le sommet précédant v_j dans un chemin minimal allant de v_a à v_j, ce qui permet de reconstituer le chemin optimal une fois l'algorithme terminé.

Algorithme de Dijkstra

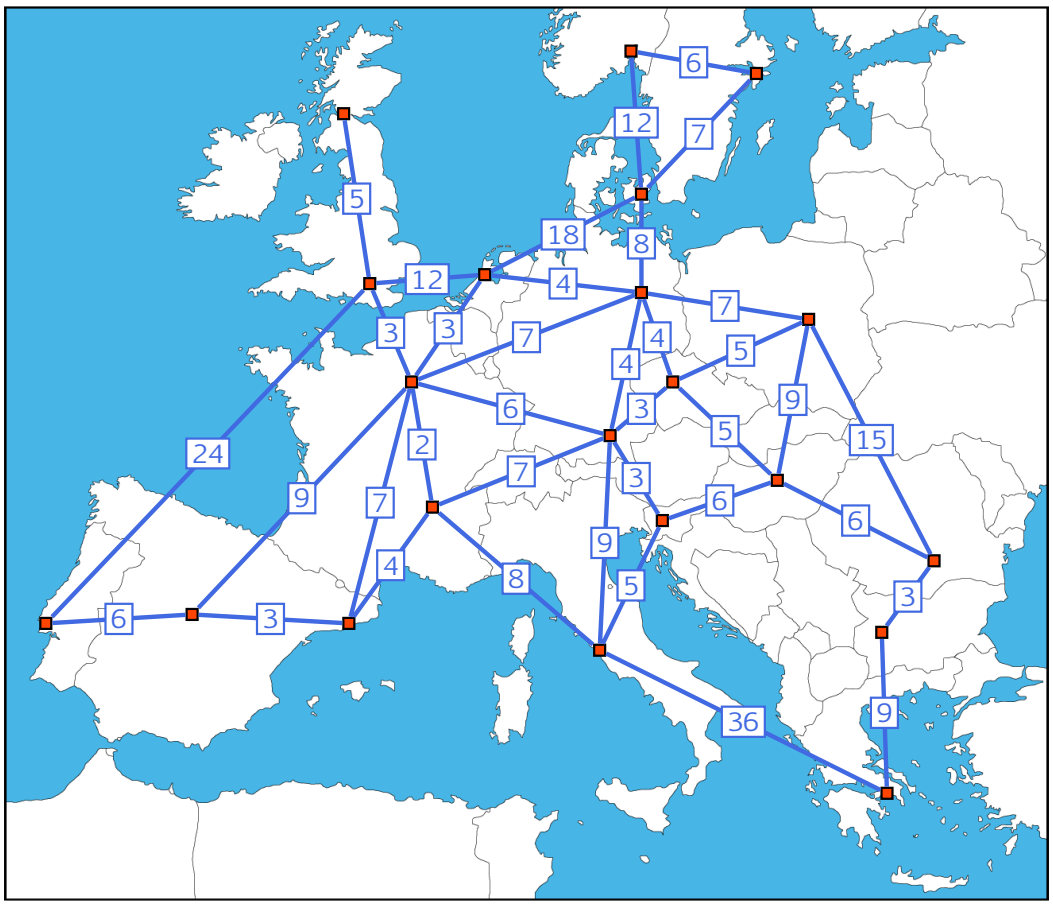
Détermination du chemin de poids minimal

```
let pluscourtchemin g w a b =
  let n = vect_length g in
  let d = make_vect n Inf and c = make_vect n 0
  and t = make_vect n 0 and m = make_vect n 0 in
  for k = 0 to n-1 do
    d.(k) <- w.(a).(k) ;
    if d.(k) <> Inf then c.(k) <- a ;
    if k <> a then ajoute d t m k ;
  done ;
  let rec modif i = function
    | []   -> ()
    | j::q -> let x = som d.(i) w.(i).(j) in
              if inferieur x d.(j) then
                (d.(j) <- x ; c.(j) <- i ; remonte d t m m.(j)) ;
              modif i q
  in
  let rec affiche_chemin l =
    if hd l = a then l else affiche_chemin (c.(hd l)::l)
  in
  let rec aux s =
    let i = extrait d t m in
    modif i g.(i) ;
    if i = b then affiche_chemin [b] else aux (i::s)
  in aux [a] ;;
```

Arbre couvrant minimal

d'un graphe non orienté

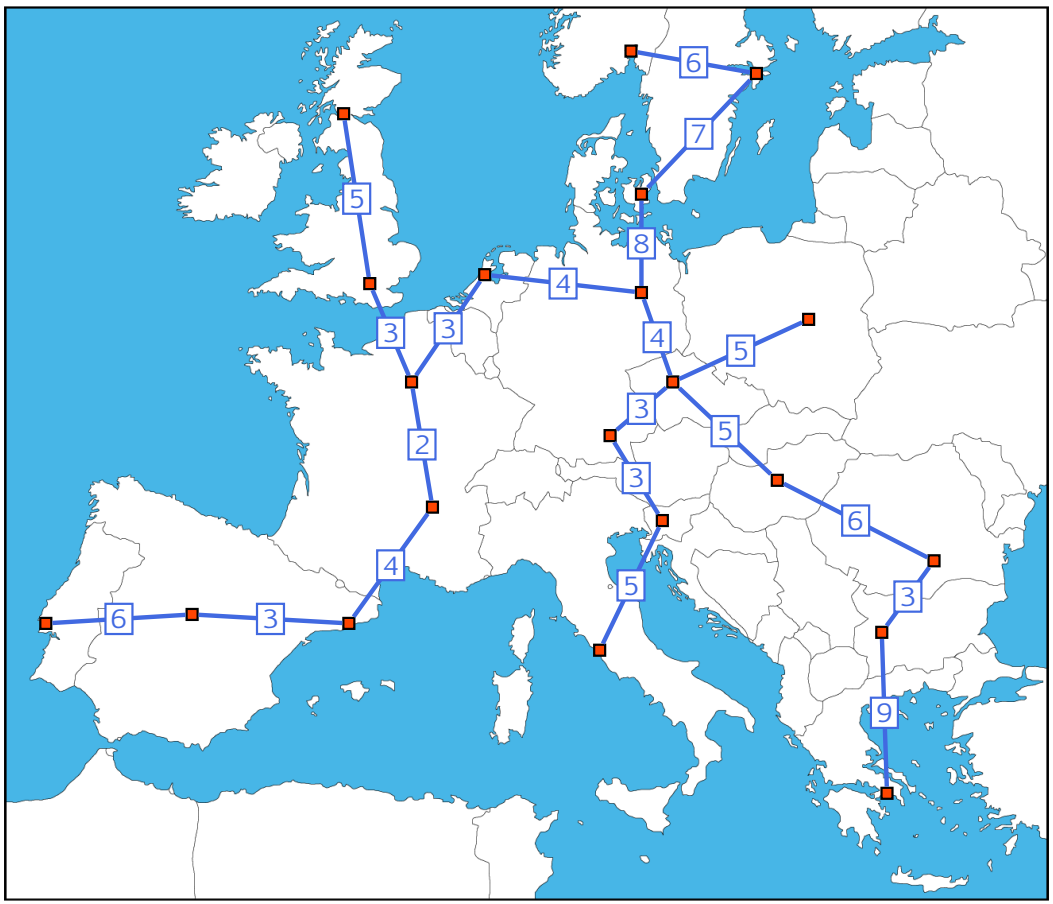
On considère un graphe non orienté connexe $G = (V, E)$ muni d'une pondération $w : E \to \mathbb{R}_+^*$ à valeurs strictement positives.



Arbre couvrant minimal

d'un graphe non orienté

On considère un graphe non orienté connexe $G = (V, E)$ muni d'une pondération $w : E \to \mathbb{R}_+^*$ à valeurs strictement positives.



Un *sous-graphe* $G' = (V', E')$ est dit couvrant lorsque il est lui-aussi connexe et $V' = V$. On souhaite en trouver un de poids minimal.

Arbre couvrant minimal

d'un graphe non orienté

G possède un sous-graphe couvrant minimal, et ce dernier est un arbre (un graphe acyclique connexe).

Arbre couvrant minimal

d'un graphe non orienté

G possède un sous-graphe couvrant minimal, et ce dernier est un arbre (un graphe acyclique connexe).

L'ensemble des sous-graphes couvrants est non vide puisqu'il contient G, et il est fini, ce qui justifie l'existence d'un sous-graphe G' couvrant de poids minimal.

Si ce sous-graphe contient un cycle, on pourrait supprimer une arête de ce cycle et le sous-graphe obtenu serait toujours couvrant et de poids strictement inférieur, ce qui est absurde.

Arbre couvrant minimal

d'un graphe non orienté

G possède un sous-graphe couvrant minimal, et ce dernier est un arbre (un graphe acyclique connexe).

L'ensemble des sous-graphes couvrants est non vide puisqu'il contient G, et il est fini, ce qui justifie l'existence d'un sous-graphe G' couvrant de poids minimal.

Si ce sous-graphe contient un cycle, on pourrait supprimer une arête de ce cycle et le sous-graphe obtenu serait toujours couvrant et de poids strictement inférieur, ce qui est absurde.

Il existe deux algorithmes classiques pour résoudre le problème de l'arbre couvrant de poids minimum :

Arbre couvrant minimal

d'un graphe non orienté

G possède un sous-graphe couvrant minimal, et ce dernier est un arbre (un graphe acyclique connexe).

L'ensemble des sous-graphes couvrants est non vide puisqu'il contient G, et il est fini, ce qui justifie l'existence d'un sous-graphe G' couvrant de poids minimal.

Si ce sous-graphe contient un cycle, on pourrait supprimer une arête de ce cycle et le sous-graphe obtenu serait toujours couvrant et de poids strictement inférieur, ce qui est absurde.

Il existe deux algorithmes classiques pour résoudre le problème de l'arbre couvrant de poids minimum :

- l'algorithme de PRIM $\rightarrow$ un arbre est un graphe connexe à $n-1$ arêtes ;

Arbre couvrant minimal

d'un graphe non orienté

G possède un sous-graphe couvrant minimal, et ce dernier est un arbre (un graphe acyclique connexe).

L'ensemble des sous-graphes couvrants est non vide puisqu'il contient G, et il est fini, ce qui justifie l'existence d'un sous-graphe G' couvrant de poids minimal.

Si ce sous-graphe contient un cycle, on pourrait supprimer une arête de ce cycle et le sous-graphe obtenu serait toujours couvrant et de poids strictement inférieur, ce qui est absurde.

Il existe deux algorithmes classiques pour résoudre le problème de l'arbre couvrant de poids minimum :

- l'algorithme de PRIM → un arbre est un graphe connexe à $n-1$ arêtes ;
- l'algorithme de KRUSKAL → un arbre est un graphe acyclique à $n-1$ arêtes.

Dans les deux cas, les arêtes sont choisies en suivant une heuristique gloutonne optimale.

Algorithme de Prim

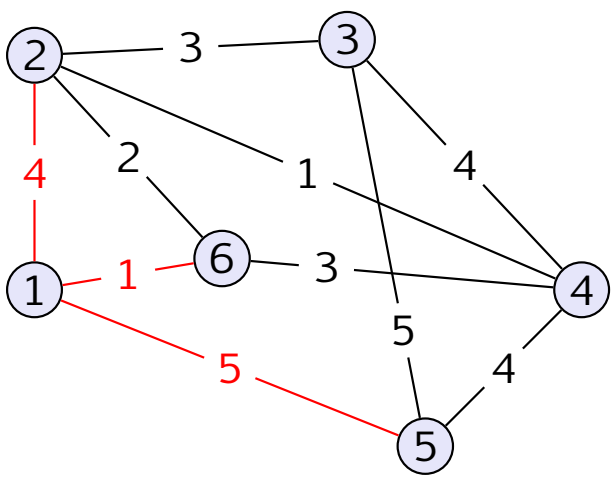
On maintient un sous-graphe partiel connexe $G' = (S, A)$ en connectant un nouveau sommet à chaque étape jusqu'à obtenir un arbre couvrant. À chaque étape on ajoute l'arête reliant un sommet de S à un sommet de $V \setminus S$ de poids le plus faible.

function PRIM*(graphe connexe : $G = (V, E)$)*
  $S \leftarrow \{s_0\}$ ▷ *un sommet choisi arbitrairement*
  $A = \emptyset$
  while $S \neq V$ ***do***
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ *et* $w(a, b)$ *minimal*
    $S \leftarrow S \cup \{b\}$
    $A \leftarrow A \cup \{(a, b)\}$
  return (S, A)

Algorithme de Prim

Exemple

function PRIM(*graphe connexe* : $G = (V, E)$)
  $S \leftarrow \{s_0\}$ ▷ *un sommet choisi arbitrairement*
  $A = \emptyset$
  while $S \neq V$ ***do***
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ *et* $w(a, b)$ *minimal*
    $S \leftarrow S \cup \{b\}$
    $A \leftarrow A \cup \{(a, b)\}$
  return (S, A)

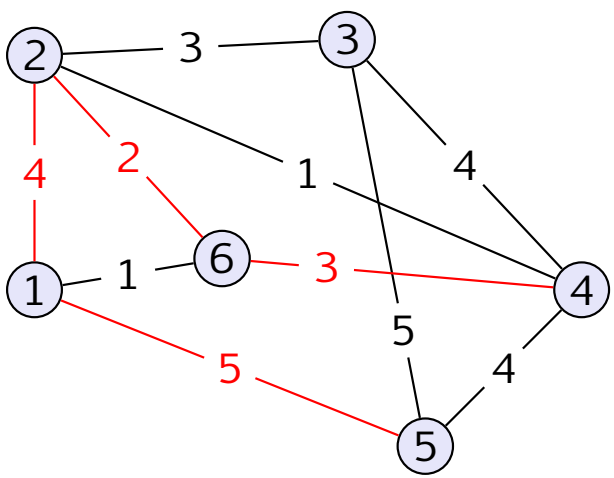


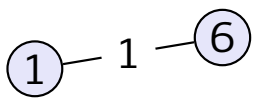
1

Algorithme de Prim

Exemple

function PRIM(*graphe connexe* : $G = (V, E)$)
  $S \leftarrow \{s_0\}$ ▷ *un sommet choisi arbitrairement*
  $A = \emptyset$
  while $S \neq V$ ***do***
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ *et* $w(a, b)$ *minimal*
    $S \leftarrow S \cup \{b\}$
    $A \leftarrow A \cup \{(a, b)\}$
  return (S, A)

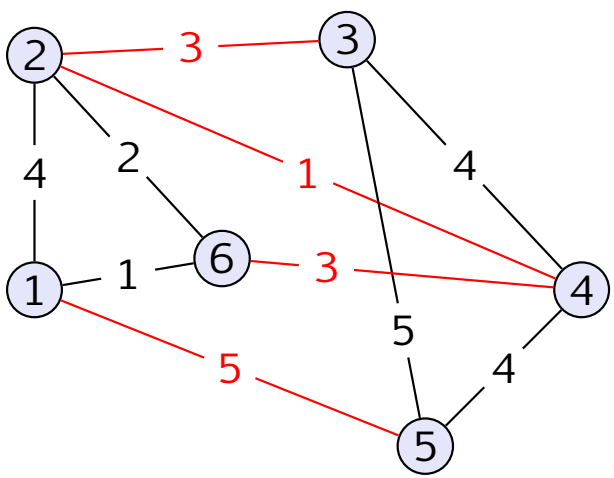


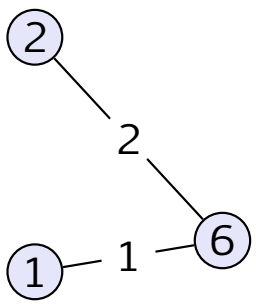


Algorithme de Prim

Exemple

function PRIM(*graphe connexe* : $G = (V, E)$)
  $S \leftarrow \{s_0\}$ ▷ *un sommet choisi arbitrairement*
  $A = \emptyset$
  while $S \neq V$ ***do***
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ *et* $w(a, b)$ *minimal*
    $S \leftarrow S \cup \{b\}$
    $A \leftarrow A \cup \{(a, b)\}$
  return (S, A)

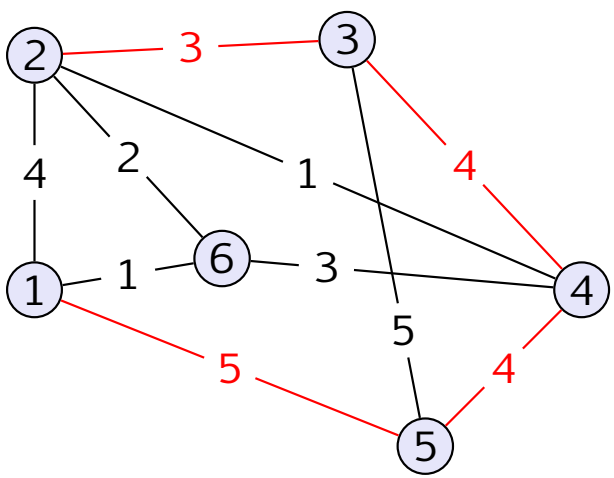


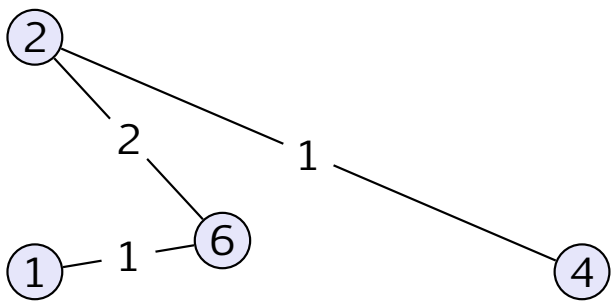


Algorithme de Prim

Exemple

function PRIM(*graphe connexe* : $G = (V, E)$)
  $S \leftarrow \{s_0\}$ ▷ *un sommet choisi arbitrairement*
  $A = \emptyset$
  while $S \neq V$ ***do***
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ *et* $w(a, b)$ *minimal*
    $S \leftarrow S \cup \{b\}$
    $A \leftarrow A \cup \{(a, b)\}$
  return (S, A)

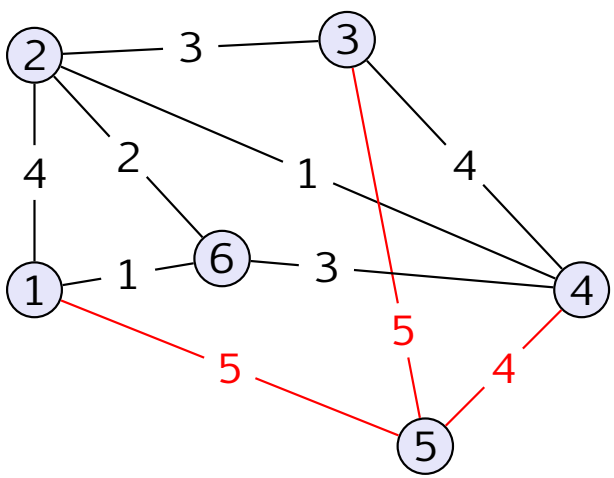


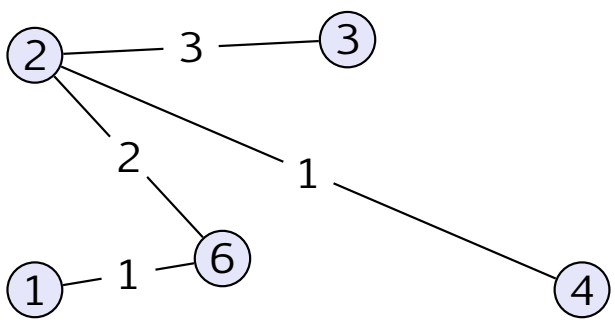


Algorithme de Prim

Exemple

function PRIM(*graphe connexe* : $G = (V, E)$)
  $S \leftarrow \{s_0\}$ ▷ *un sommet choisi arbitrairement*
  $A = \emptyset$
  while $S \neq V$ ***do***
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ *et* $w(a, b)$ *minimal*
    $S \leftarrow S \cup \{b\}$
    $A \leftarrow A \cup \{(a, b)\}$
  return (S, A)

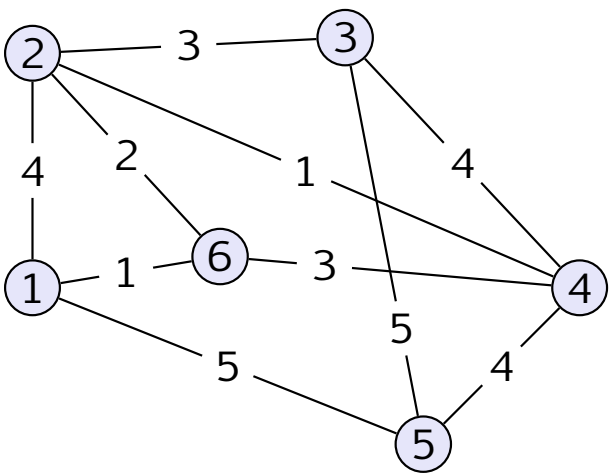


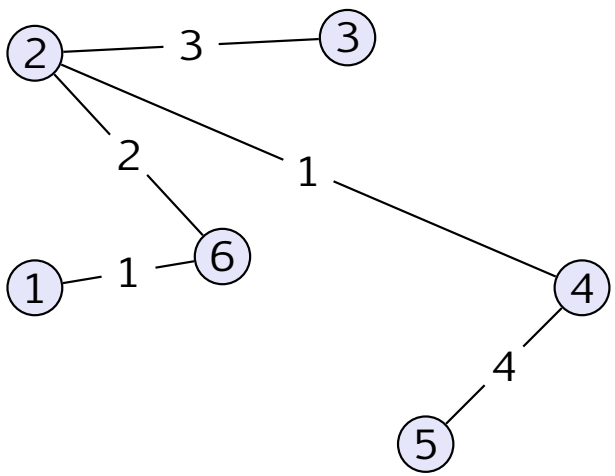


Algorithme de Prim

Exemple

function PRIM(*graphe connexe* : $G = (V, E)$)
  $S \leftarrow \{s_0\}$ ▷ *un sommet choisi arbitrairement*
  $A = \emptyset$
  while $S \neq V$ ***do***
    Soit $(a, b) \in E \mid (a, b) \in S \times (V \setminus S)$ *et* $w(a, b)$ *minimal*
    $S \leftarrow S \cup \{b\}$
    $A \leftarrow A \cup \{(a, b)\}$
  return (S, A)





Algorithme de Prim

L'algorithme de Prim calcule un arbre couvrant de poids minimal.

Algorithme de Prim

L'algorithme de Prim calcule un arbre couvrant de poids minimal.

Le graphe construit par cet algorithme est un graphe connexe couvrant à n sommets et $n-1$ arêtes donc est un arbre.

Algorithme de Prim

L'algorithme de Prim calcule un arbre couvrant de poids minimal.

Le graphe construit par cet algorithme est un graphe connexe couvrant à n sommets et $n-1$ arêtes donc est un arbre.
On prouve par récurrence sur $|S|$ qu'à chaque étape il existe un arbre couvrant de poids minimal qui contient le graphe (S, A).

Algorithme de Prim

L'algorithme de Prim calcule un arbre couvrant de poids minimal.

Le graphe construit par cet algorithme est un graphe connexe couvrant à n sommets et $n-1$ arêtes donc est un arbre.
On prouve par récurrence sur $|S|$ qu'à chaque étape il existe un arbre couvrant de poids minimal qui contient le graphe (S, A).

- C'est évident lorsque $|S| = 1 : A = \emptyset$.

Algorithme de Prim

L'algorithme de Prim calcule un arbre couvrant de poids minimal.

Le graphe construit par cet algorithme est un graphe connexe couvrant à n sommets et $n-1$ arêtes donc est un arbre.
On prouve par récurrence sur $|S|$ qu'à chaque étape il existe un arbre couvrant de poids minimal qui contient le graphe (S, A).

- C'est évident lorsque $|S| = 1 : A = \emptyset$.
- Si $|S| \geqslant 1$, soit T un arbre couvrant de poids minimal contenant (S, A), et soit (a, b) l'arête que l'on ajoute à A.

Algorithme de Prim

L'algorithme de Prim calcule un arbre couvrant de poids minimal.

Le graphe construit par cet algorithme est un graphe connexe couvrant à n sommets et $n-1$ arêtes donc est un arbre.
On prouve par récurrence sur $|S|$ qu'à chaque étape il existe un arbre couvrant de poids minimal qui contient le graphe (S, A).

- C'est évident lorsque $|S| = 1 : A = \emptyset$.
- Si $|S| \geqslant 1$, soit T un arbre couvrant de poids minimal contenant (S, A), et soit (a, b) l'arête que l'on ajoute à A.
  - Si $(a, b) \in T$, T convient toujours.

Algorithme de Prim

L'algorithme de Prim calcule un arbre couvrant de poids minimal.

Le graphe construit par cet algorithme est un graphe connexe couvrant à n sommets et $n-1$ arêtes donc est un arbre.
On prouve par récurrence sur $|S|$ qu'à chaque étape il existe un arbre couvrant de poids minimal qui contient le graphe (S, A).

- C'est évident lorsque $|S| = 1 : A = \emptyset$.
- Si $|S| \geqslant 1$, soit T un arbre couvrant de poids minimal contenant (S, A), et soit (a, b) l'arête que l'on ajoute à A.
  - Si $(a, b) \in T$, T convient toujours.
  - Sinon on ajoute cette arête à T : on crée nécessairement un cycle. Ce cycle parcours à la fois des éléments de S (parmi eux, a) et des éléments de $V \setminus S$ (parmi eux, b). Il existe donc nécessairement une autre arête $(a', b') \neq (a, b)$ de ce cycle tel que $a' \in S$ et $b' \in V \setminus S$. On note T' l'arbre obtenu en supprimant (a', b'). Il s'agit de nouveau d'un arbre couvrant et il est de poids minimal car $w(a, b) \leqslant w(a', b')$.

Algorithme de Prim

Étude de la complexité

Si p dénote le nombre d'arêtes du graphe G, la recherche naïve de l'arête de poids minimal (a, b) est un $\mathrm{O}(p)$ et le coût total un $\mathrm{O}(np)$.

Algorithme de Prim

Étude de la complexité

Si p dénote le nombre d'arêtes du graphe G, la recherche naïve de l'arête de poids minimal (a, b) est un $\mathrm{O}(p)$ et le coût total un $\mathrm{O}(np)$.

On peut faire mieux en procédant à un pré-traitement des sommets consistant à déterminer pour chacun d'eux l'arête incidente de poids minimal qui le relie à un sommet de S.
Dès lors, le coût de la recherche de l'arête de poids minimal devient un $\mathrm{O}(n)$, et une fois le nouveau sommet ajouté à S, il suffit de mettre à jour les voisins de celui-ci.

Algorithme de Prim

Étude de la complexité

Si p dénote le nombre d'arêtes du graphe G, la recherche naïve de l'arête de poids minimal (a, b) est un $\mathrm{O}(p)$ et le coût total un $\mathrm{O}(np)$.

On peut faire mieux en procédant à un pré-traitement des sommets consistant à déterminer pour chacun d'eux l'arête incidente de poids minimal qui le relie à un sommet de S.
Dès lors, le coût de la recherche de l'arête de poids minimal devient un $\mathrm{O}(n)$, et une fois le nouveau sommet ajouté à S, il suffit de mettre à jour les voisins de celui-ci.
Sachant que le coût du pré-traitement est un $\mathrm{O}(p) = \mathrm{O}(n^2)$, le coût total de l'algorithme est un $\mathrm{O}(n^2)$.

Algorithme de Prim

Étude de la complexité

Si p dénote le nombre d'arêtes du graphe G, la recherche naïve de l'arête de poids minimal (a, b) est un $\mathrm{O}(p)$ et le coût total un $\mathrm{O}(np)$.

On peut faire mieux en procédant à un pré-traitement des sommets consistant à déterminer pour chacun d'eux l'arête incidente de poids minimal qui le relie à un sommet de S.
Dès lors, le coût de la recherche de l'arête de poids minimal devient un $\mathrm{O}(n)$, et une fois le nouveau sommet ajouté à S, il suffit de mettre à jour les voisins de celui-ci.
Sachant que le coût du pré-traitement est un $\mathrm{O}(p) = \mathrm{O}(n^2)$, le coût total de l'algorithme est un $\mathrm{O}(n^2)$.

Remarque. L'utilisation d'un tas pour stocker les différents sommets n'appartenant pas à S permet de réduire le coût, qui devient un $\mathrm{O}(p \log n)$.

$\rightarrow$ c'est intéressant dès lors que $p = \mathrm{O}\left(\dfrac{n^2}{\log n}\right)$.

Algorithme de Kruskal

On maintient un graphe partiel acyclique (une forêt) jusqu'à ne plus obtenir qu'une seule composante connexe (un arbre). On débute avec le graphe à n sommets et aucune arête, et à chaque étape on ajoute une arête permettant de réunir deux composantes connexes distinctes et de poids minimal.

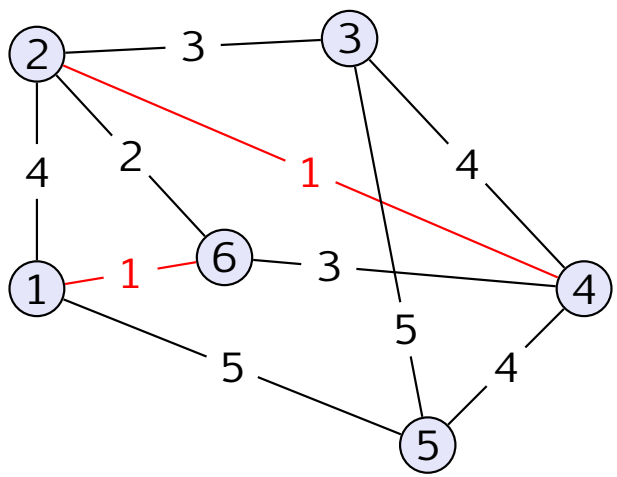
function KRUSKAL($graphe : G = (V, E)$)
    $A = \emptyset$
    $E_t = tri\_croissant(E)$
    for $(a, b) \in E_t$ **do**
        if $A \cup \{(a, b)\}$ *est acyclique* **then**
            $A \leftarrow A \cup \{(a, b)\}$
    return (V, A)

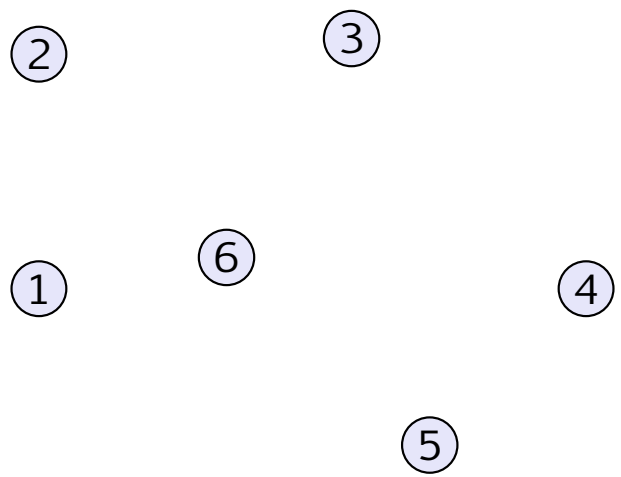
On commence par trier par ordre croissant de poids les arêtes de G.

Algorithme de Kruskal

Exemple

function KRUSKAL($graphe : G = (V, E)$)
  $A = \emptyset$
  $E_t = tri\_croissant(E)$
  for $(a, b) \in E_t$ ***do***
    if $A \cup \{(a, b)\}$ *est acyclique* ***then***
      $A \leftarrow A \cup \{(a, b)\}$
  return (V, A)

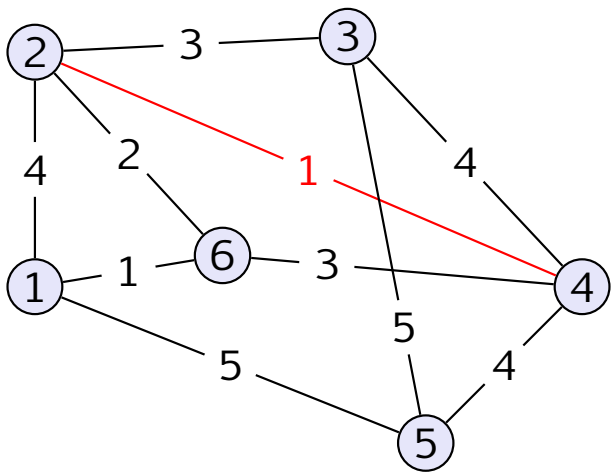


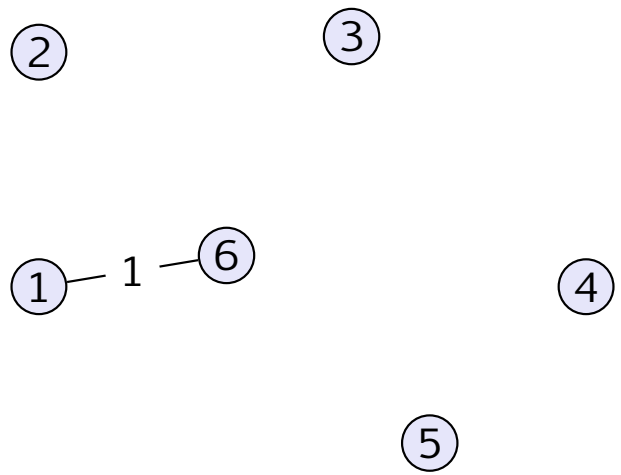


Algorithme de Kruskal

Exemple

function KRUSKAL($graphe : G = (V, E)$)
    $A = \emptyset$
    $E_t = tri\_croissant(E)$
    for $(a, b) \in E_t$ **do**
        if $A \cup \{(a, b)\}$ *est acyclique* **then**
            $A \leftarrow A \cup \{(a, b)\}$
    return (V, A)

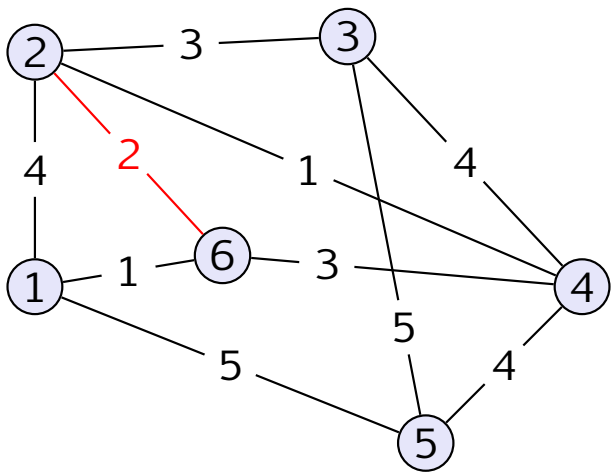


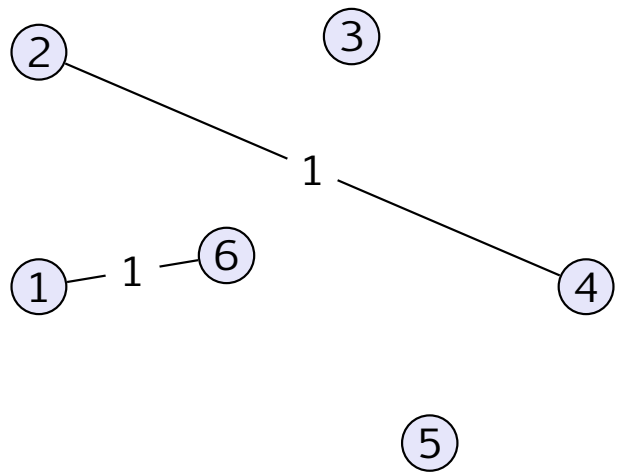


Algorithme de Kruskal

Exemple

function KRUSKAL($graphe : G = (V, E)$)
    $A = \emptyset$
    $E_t = tri\_croissant(E)$
    for $(a, b) \in E_t$ ***do***
        if $A \cup \{(a, b)\}$ *est acyclique* ***then***
            $A \leftarrow A \cup \{(a, b)\}$
    return (V, A)

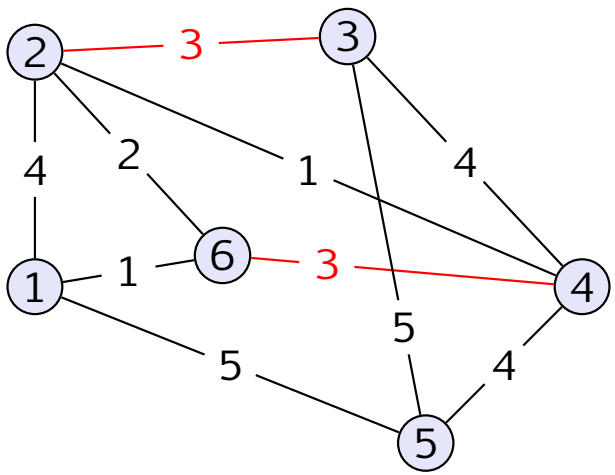


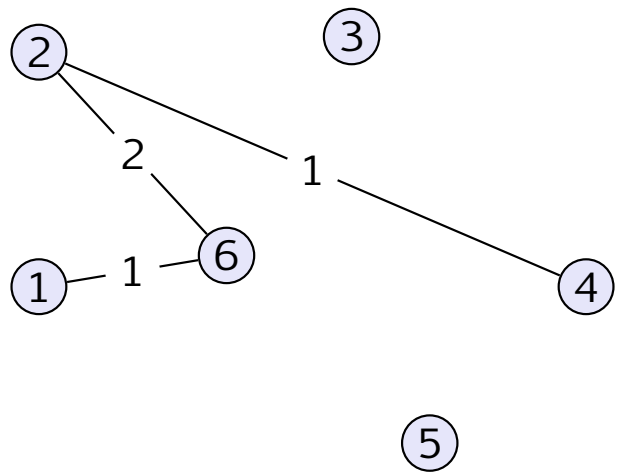


Algorithme de Kruskal

Exemple

function KRUSKAL($graphe : G = (V, E)$)
    $A = \emptyset$
    $E_t = tri\_croissant(E)$
    for $(a, b) \in E_t$ ***do***
        if $A \cup \{(a, b)\}$ *est acyclique* ***then***
            $A \leftarrow A \cup \{(a, b)\}$
    return (V, A)

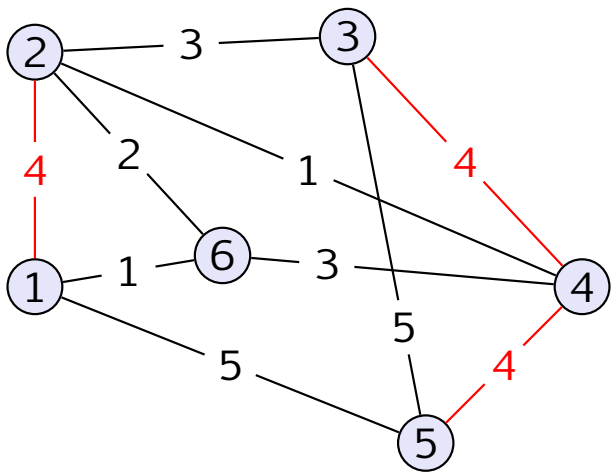


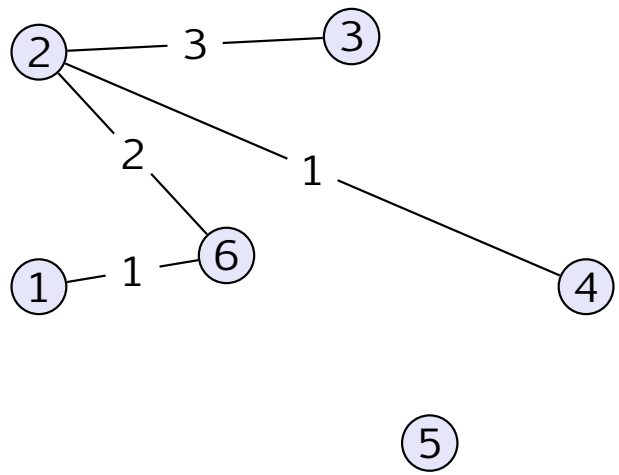


Algorithme de Kruskal

Exemple

function KRUSKAL($graphe : G = (V, E)$)
    $A = \emptyset$
    $E_t = tri\_croissant(E)$
    for $(a, b) \in E_t$ ***do***
        if $A \cup \{(a, b)\}$ *est acyclique* ***then***
            $A \leftarrow A \cup \{(a, b)\}$
    return (V, A)

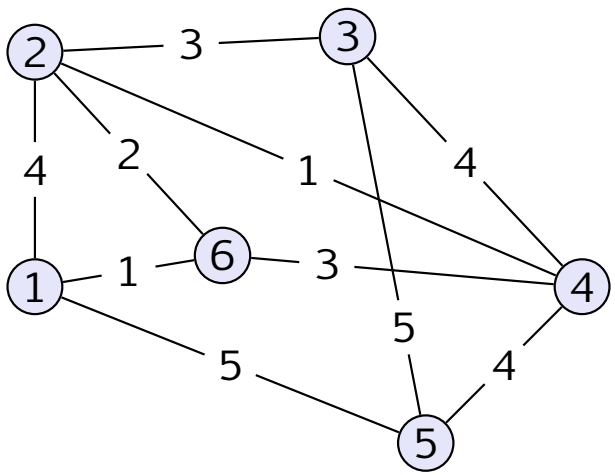


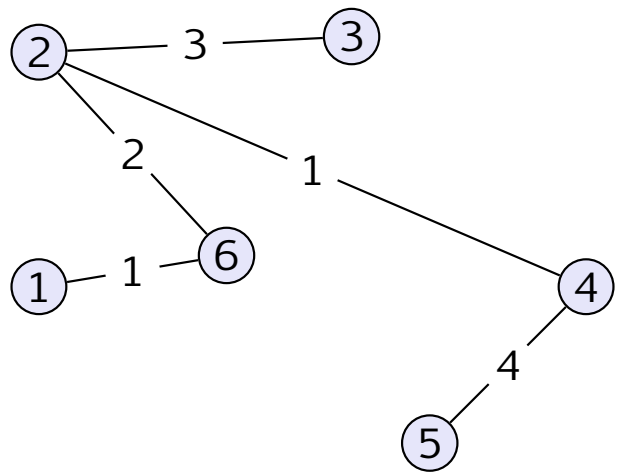


Algorithme de Kruskal

Exemple

function KRUSKAL($graphe : G = (V, E)$)
    $A = \emptyset$
    $E_t = tri\_croissant(E)$
    for $(a, b) \in E_t$ ***do***
        if $A \cup \{(a, b)\}$ *est acyclique* ***then***
            $A \leftarrow A \cup \{(a, b)\}$
    return (V, A)





Algorithme de Kruskal

Exemple

function KRUSKAL($graphe : G = (V, E)$)
    $A = \emptyset$
    $E_t = tri\_croissant(E)$
    for $(a, b) \in E_t$ **do**
        if $A \cup \{(a, b)\}$ *est acyclique* **then**
            $A \leftarrow A \cup \{(a, b)\}$
    return (V, A)

Cet algorithme s'applique à un graphe non nécessairement connexe et retourne dans ce cas une forêt couvrante de poids minimal de G. Si on sait que le graphe est connexe, on peut stopper cet algorithme dès lors que $|A| = |V| - 1$.

Algorithme de Kruskal

Toutes les forêts couvrantes d'un graphe G ont même nombre d'arêtes.

Notons $G = (V, E)$, et $C_1, \ldots, C_p$ les composantes connexes de G. Le nombre d'arêtes d'une forêt couvrante de G est alors égale à :

$$\sum_{i=1}^{p} \Big(|C_i| - 1 \Big) = |G| - p.$$

Algorithme de Kruskal

Toutes les forêts couvrantes d'un graphe G ont même nombre d'arêtes.

L'algorithme de Kruskal calcule une forêt couvrante de poids minimal.

Algorithme de Kruskal

Toutes les forêts couvrantes d'un graphe G ont même nombre d'arêtes.

L'algorithme de Kruskal calcule une forêt couvrante de poids minimal.

Notons que le graphe construit est bien une forêt couvrante (on ne crée pas de cycle).

Algorithme de Kruskal

Toutes les forêts couvrantes d'un graphe G ont même nombre d'arêtes.

L'algorithme de Kruskal calcule une forêt couvrante de poids minimal.

Notons que le graphe construit est bien une forêt couvrante (on ne crée pas de cycle).
On note $A = (e_1, \ldots, e_k)$ les arêtes choisies rangées par ordre de poids croissant, et on considère une autre forêt couvrante $F = (V, A')$ dont les arêtes $A' = (e'_1, \ldots, e'_k)$ sont elles aussi rangées par ordre de poids croissant. Nous allons montrer que pour tout $i \in [\![1, k]\!]$ on a $w(e_i) \leqslant w(e'_i)$.

Algorithme de Kruskal

Toutes les forêts couvrantes d'un graphe G ont même nombre d'arêtes.

L'algorithme de Kruskal calcule une forêt couvrante de poids minimal.

Notons que le graphe construit est bien une forêt couvrante (on ne crée pas de cycle).
On note $A = (e_1, \dots, e_k)$ les arêtes choisies rangées par ordre de poids croissant, et on considère une autre forêt couvrante $F = (V, A')$ dont les arêtes $A' = (e'_1, \dots, e'_k)$ sont elles aussi rangées par ordre de poids croissant. Nous allons montrer que pour tout $i \in [\![1, k]\!]$ on a $w(e_i) \leqslant w(e'_i)$.
On suppose qu'il existe $i \in [\![1, k]\!]$ tel que $w(e'_i) < w(e_i)$, et on considère le graphe $G' = (V, E')$, avec $E' = \left\{ e \in E \;\middle|\; w(e) \leqslant w(e'_i) \right\}$.
Appliqué à G', l'algorithme de Kruskal se déroule comme sur G et retourne un ensemble d'arêtes inclus dans $\{e_1, \dots, e_{i-1}\}$, autrement dit une forêt couvrante de G' comportant *au plus* $i - 1$ arêtes. Or la forêt $F' = (V, A'')$ avec $A'' = (e'_1, \dots, e'_i)$ est une forêt couvrante de G' qui comporte i arêtes, ce qui contredit le résultat du lemme précédent.

Algorithme de Kruskal

Étude de la complexité

L'algorithme de Kruskal consiste avant tout :

- à trier les arêtes ;
- à les énumérer par ordre croissant.

L'usage d'un tas s'impose.

Algorithme de Kruskal

Étude de la complexité

L'algorithme de Kruskal consiste avant tout :

- à trier les arêtes ;
- à les énumérer par ordre croissant.

L'usage d'un tas s'impose. $\rightarrow$ Le coût de la formation du tas est un $O(p)$.

Par ailleurs, il existe des structures de données (Union-Find) qui permettent de gérer efficacement une partition d'objets pour représenter l'évolution des différentes composantes connexes. Avec une telle structure, le coût total de l'algorithme est un $O(p \log p)$.

Algorithme de Kruskal

Étude de la complexité

L'algorithme de Kruskal consiste avant tout :

- à trier les arêtes ;
- à les énumérer par ordre croissant.

L'usage d'un tas s'impose. $\rightarrow$ Le coût de la formation du tas est un $O(p)$.

Par ailleurs, il existe des structures de données (Union-Find) qui permettent de gérer efficacement une partition d'objets pour représenter l'évolution des différentes composantes connexes. Avec une telle structure, le coût total de l'algorithme est un $O(p \log p)$.

Sachant que $p = O(n^2)$ on peut simplifier le coût en $O(p \log n)$, identique au coût de l'algorithme de Prim.