Algorithmique avancée

L3 Informatique

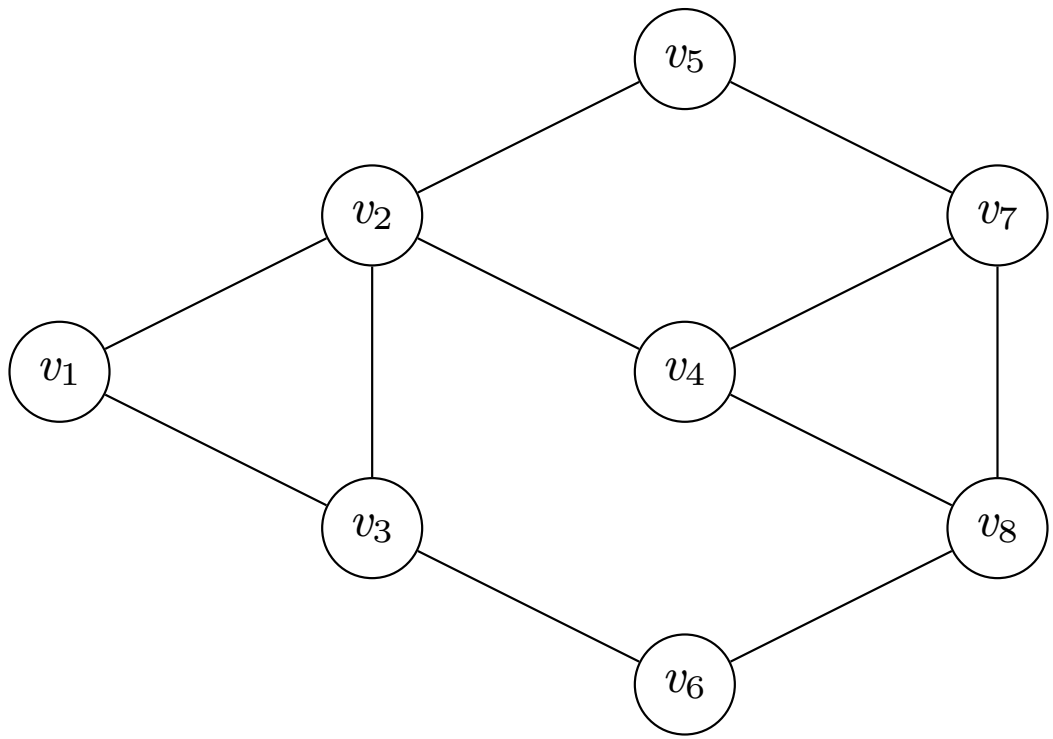
Etienne Birmelé

I. Introduction : Notion de graphe

Notion de graphe

Graphe

Un graphe G est un couple (V, E) où V est l'ensemble des sommets et $E \subset V \times V$ est l'ensemble des arêtes.
Il est représenté graphiquement par un ensemble de points (les sommets) reliés par des segments (les arêtes).



Notion de graphe

- Le graphe est l'objet mathématique permettant de décrire des interactions entre sujets.
- Applications aussi diverses que l'informatique, les télécommunications, la sociologie ou la biologie.
- Principe : traduire le problème posé en un problème d'algorithmique à résoudre sur des graphes.

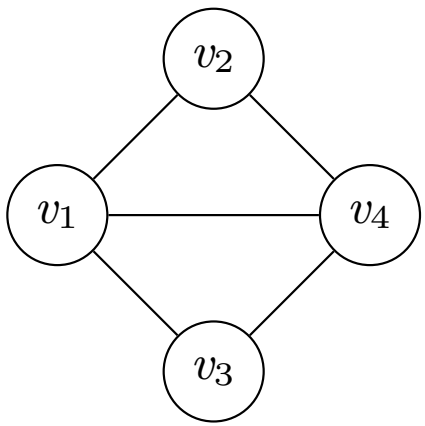
Exemple : Attribution de fréquences à des émetteurs.

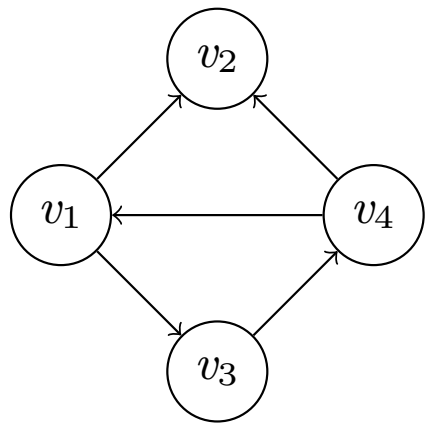
Deux émetteurs trop proches et non séparés par un obstacle naturel ne peuvent recevoir le même fréquence. Le problème devient un problème de coloration de graphes.

Types de graphes

Suivant le type de problème que l'on cherche à résoudre, on peut considérer des graphes

- non-orienté ou orienté.

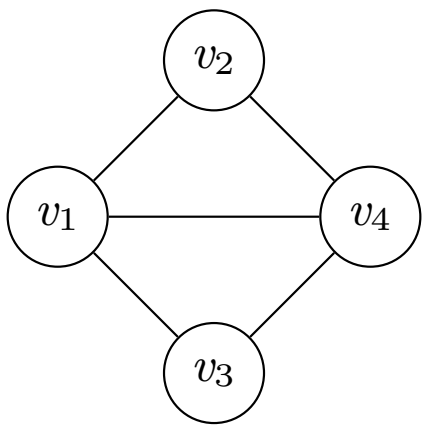


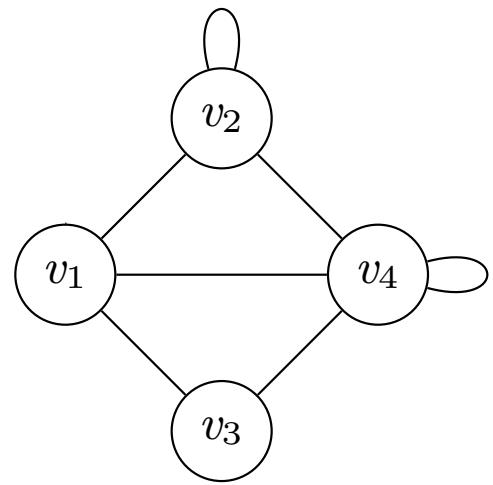


Types de graphes

Suivant le type de problème que l'on cherche à résoudre, on peut considérer des graphes

- non-orienté ou orienté.
- avec ou sans boucle

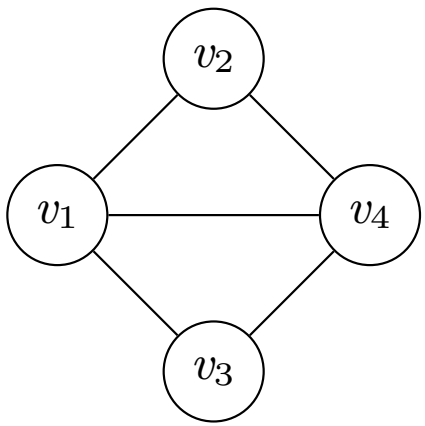


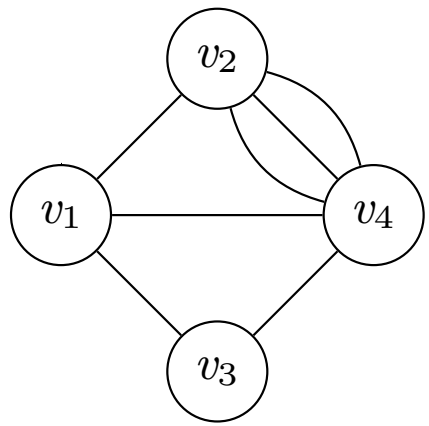


Types de graphes

Suivant le type de problème que l'on cherche à résoudre, on peut considérer des graphes

- non-orienté ou orienté.
- avec ou sans boucle
- simple ou avec des arêtes multiples

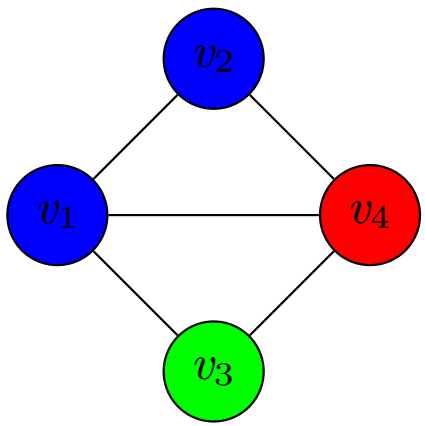


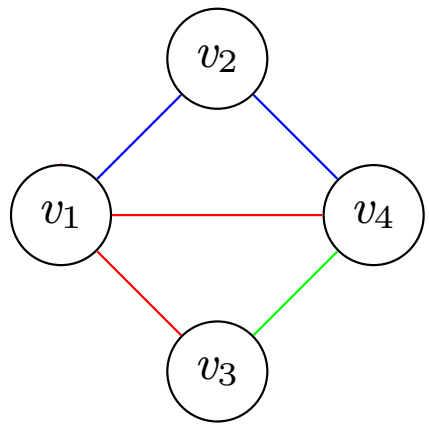


Types de graphes

Suivant le type de problème que l'on cherche à résoudre, on peut considérer des graphes

- non-orienté ou orienté.
- avec ou sans boucle
- simple ou avec des arêtes multiples
- colorés (sommets ou arêtes)

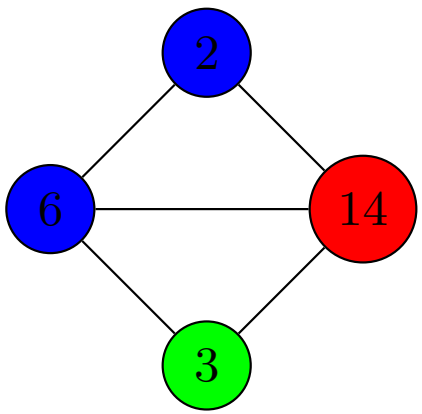


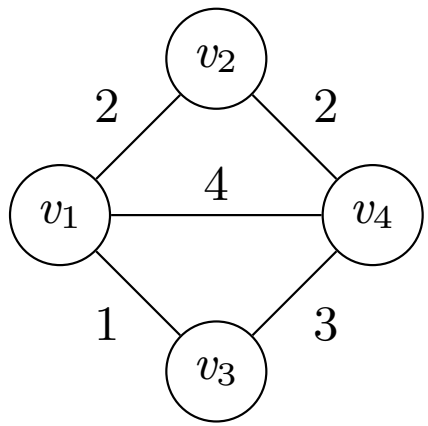


Types de graphes

Suivant le type de problème que l'on cherche à résoudre, on peut considérer des graphes

- non-orienté ou orienté.
- avec ou sans boucle
- simple ou avec des arêtes multiples
- colorés (sommets ou arêtes)
- valués (sommets ou arêtes)





Degrés, densité

Notation

On note $n(G) = |V|$ et $m(G) = |E|$, ou n et m quand il n'y a pas d'ambiguité.

Densité d'un graphe

La densité d'un graphe simple G est le nombre d'arêtes présentes divisé par le nombre d'arêtes possibles :

$$dens(G) = \frac{2m}{n(n-1)}$$

- La densité est toujours comprise entre 0 et 1. Une densité de 0 correspond à un graphe sans arête, une densité de 1 correspond à un graphe complet, également appelé clique.
- Cette notion peut être élargie à d'autres familles de graphes, il suffit de savoir calculer le nombre d'arêtes possibles. Si on considère les graphes orientés autorisant deux arêtes en sens contraire et des boucles, on obtient n^2 arêtes au maximum donc $d(G) = \frac{m}{n^2}$.

La densité est une caractéristique globale du graphe mais ne reflète pas sa structure locale.

Degrés, densité

Definition

Le degré d'un sommet v, noté $d(v)$, désigne son nombre de voisins.
Dans le cas des graphes dirigés, on distingue le degré sortant $d^+(v)$ et le degré entrant $d^-(v)$.
Dans le cas de graphe valué, le degré considéré est parfois la somme des poids des arêtes incidentes au sommet v.

Proposition

Un graphe non orienté vérifie

$$\sum_{v \in V(G)} d(v) = 2m.$$

Un graphe orienté vérifie

$$\sum_{v \in V(G)} d^+(v) = \sum_{v \in V(G)} d^-(v) = m.$$

Codage d'un graphe

Il y a principalement trois types de codages pour un graphe :

Matrice d'adjacence matrice carrée M de taille $n \times n$ tel que M_{uv} représente l'interaction entre u et v : 0 s'il n'y a pas d'arête, 1 si il y a une arête de u vers v.
Dans le cas non-dirigé, cette matrice est symétrique.
Dans le cas arête-valué, on remplace le coefficient 1 par le poids de l'arête concernée.

Liste d'arêtes matrice de taille $m \times 2$, chaque ligne représentant les deux sommets reliés par une arête.
Dans le cas orienté, l'ordre d'apparition des sommets sur la ligne indique le sens de l'arête.
Dans le cas arête-valué, on ajoute une troisième colonne contenant les poids.

Liste de voisinages Liste ayant un élément par sommet. Cet élément contient l'identité d'un sommet puis la liste de ses voisins (seulement les voisins externes dans les cas d'un graphe orienté).

Codage d'un graphe

	Stockage	u et v sont-ils voisins ?	degré d'un sommet v
Matrice d'adjacence	$O(n^2)$	$O(1)$	$O(n)$
Liste d'arête	$O(m)$	$O(m)$	$O(m)$
Liste d'adjacence	$O(m)$	$O(n)$	$O(n)$

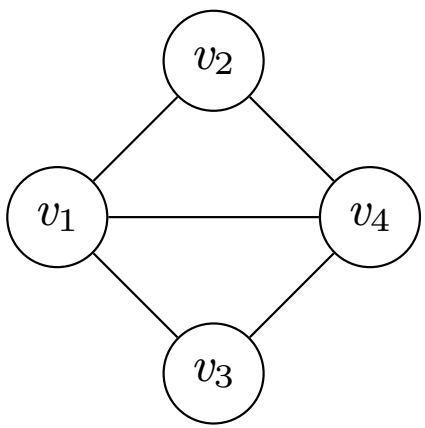
- Les deux dernières facons de stocker un graphe sont beaucoup plus efficaces, surtout dans le cas de graphes creuses ($m << n^2$).

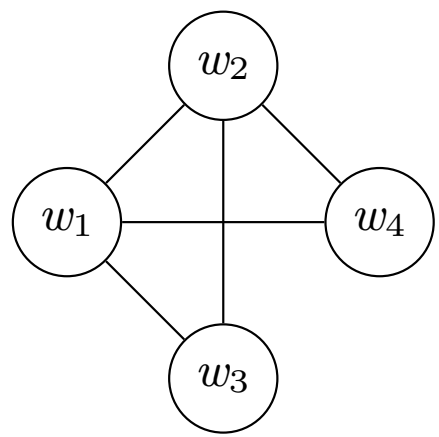
Isomorphisme de graphes

La représentation graphique d'un graphe n'est pas unique.

Isomorphisme de graphes

Deux graphes $G = (V(G), E(G))$ et $H = (V(H), E(H))$ sont isomorphes s'il existe une bijection $\phi : E(G) \to E(H)$ telle que $(u, v) \in E(G)$ si et seulement si $(\phi(u), \phi(v)) \in E(H)$.

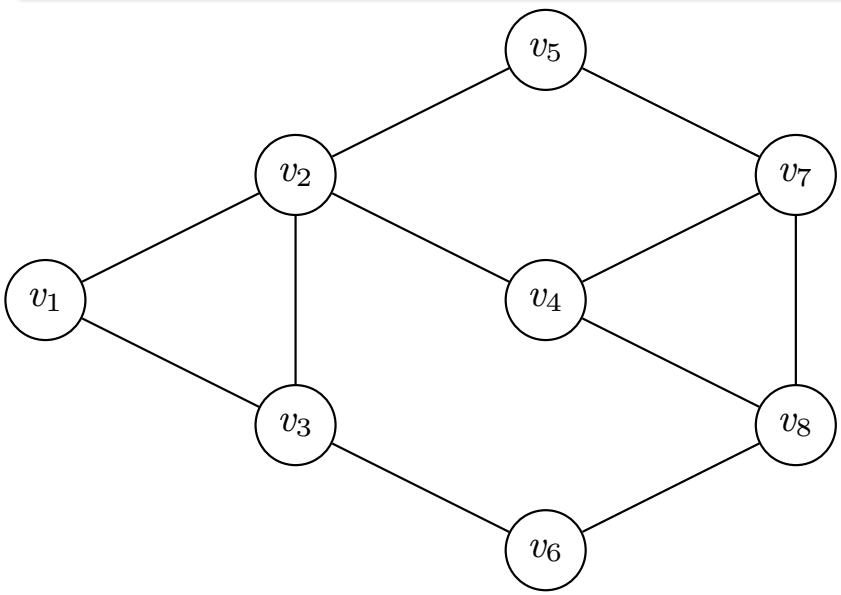




- Dans le cas des graphes valués, il faut que la bijection préserve aussi les poids.

Chemin et marche

Un chemin de longueur k entre deux sommets u et v est une suite de k arêtes (u_i, u_{i+1}) tels que $u_0 = u$, $u_k = v$ et tous les u_i sont disjoints.
Dans le cas de graphes orientés, un chemin orienté nécessite l'orientation des arêtes dans le sens $\overrightarrow{u_i u_{i+1}}$.
Si les arêtes et les sommets ne sont pas tous disjoints, on parle de marche entre u et v.

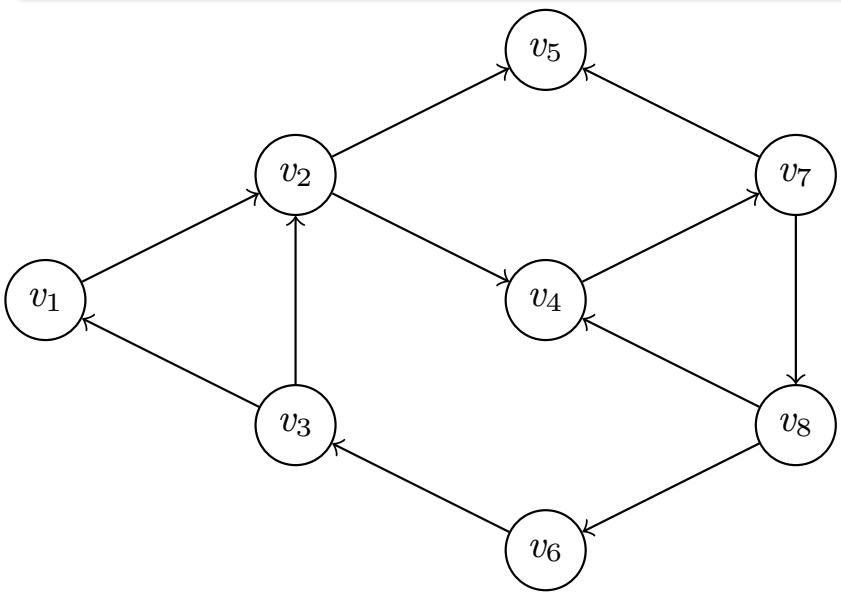


- v_6, v_3, v_2, v_4 est un chemin de longueur 3.
- v_7, v_4, v_8, v_7, v_5 est une marche.

Chemins et marches

Chemin et marche

Un chemin de longueur k entre deux sommets u et v est une suite de k arêtes (u_i, u_{i+1}) tels que $u_0 = u$, $u_k = v$ et tous les u_i sont disjoints.
Dans le cas de graphes orientés, un chemin orienté nécessite l'orientation des arêtes dans le sens $\overrightarrow{u_i u_{i+1}}$.
Si les arêtes et les sommets ne sont pas tous disjoints, on parle de marche entre u et v.

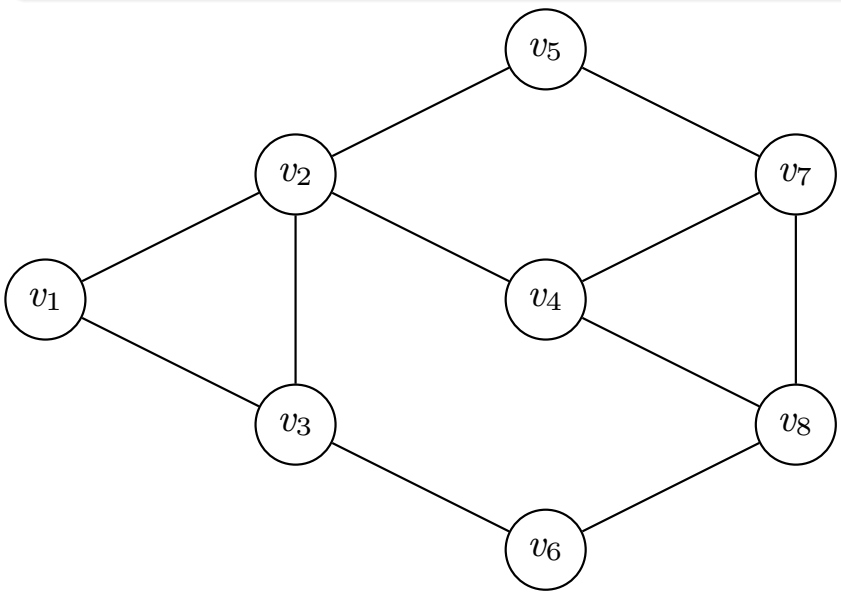


- v_6, v_3, v_2, v_4 est un chemin orienté.
- v_4, v_8, v_7, v_5 n'en est pas un.

Cycles

Definition

Un cycle de longueur k est une suite de k arêtes (u_i, u_{i+1}) tels que $u_0 = u_k$ et tous les u_i sont disjoints.
Dans le cas de graphes orientés, un cycle orienté nécessite l'orientation des arêtes dans le sens $\overrightarrow{u_i\, u_{i+1}}$.

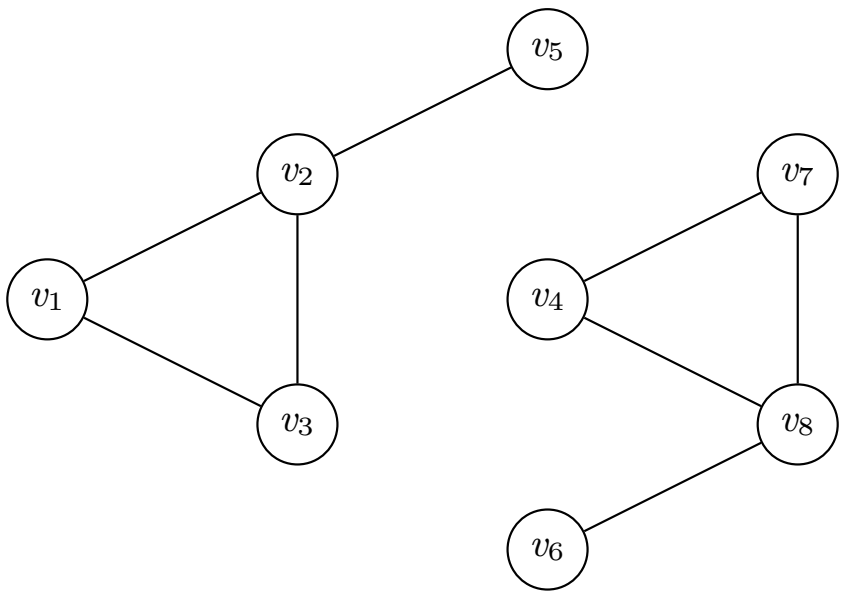


- $v_6, v_3, v_2, v_5, v_7, v_8$ est un cycle de longueur 6.

Graphe connexe

Connexité

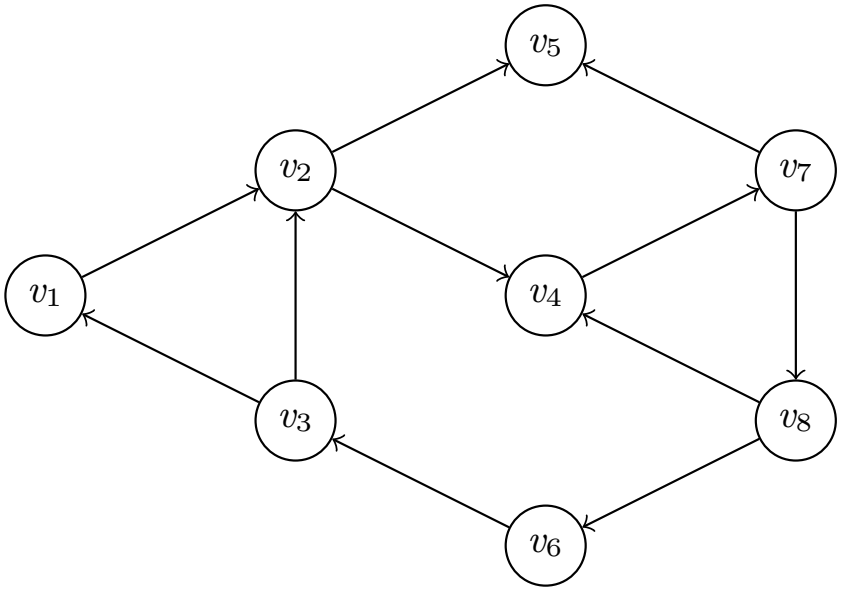
Un graphe non orienté est connexe si toute paire de sommets est reliée par un chemin.
Si le graphe n'est pas connexe, il peut être décomposé de façon unique en composantes connexes, qui sont les ensembles maximaux de sommets induisant des sous-graphes connexes.

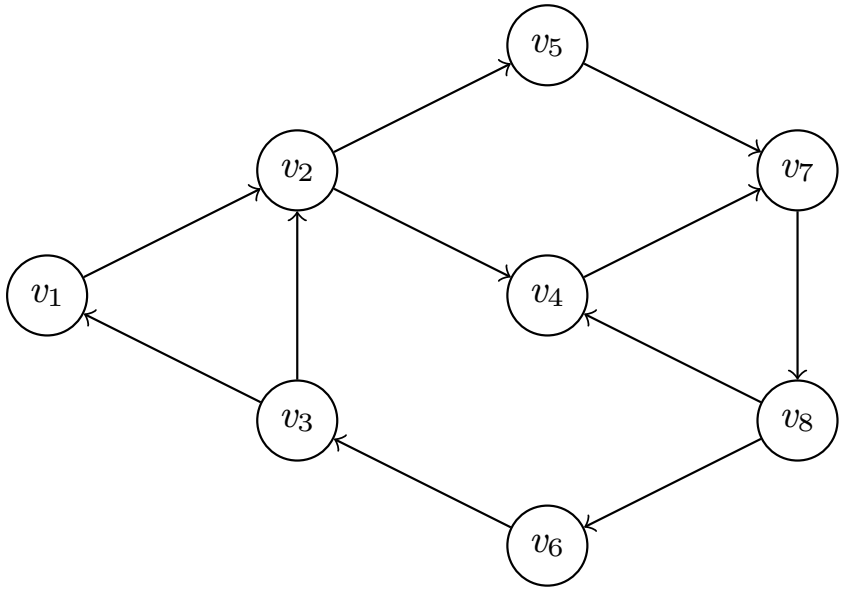


Graphe orienté fortement connexe

Definition

Un graphe orienté est fortement connexe si pour toute paire (u, v) de sommets, il existe un chemin orienté de u vers v et un chemin orienté de v vers u.
Il est dit simplement connexe si le graphe non-orienté sous-jacent est connexe.





Distance et diamètre

Definition

La distance entre deux sommets u et v appartenant à la même composante connexe d'un graphe arête-valué est le poids minimum d'un chemin entre u et v.

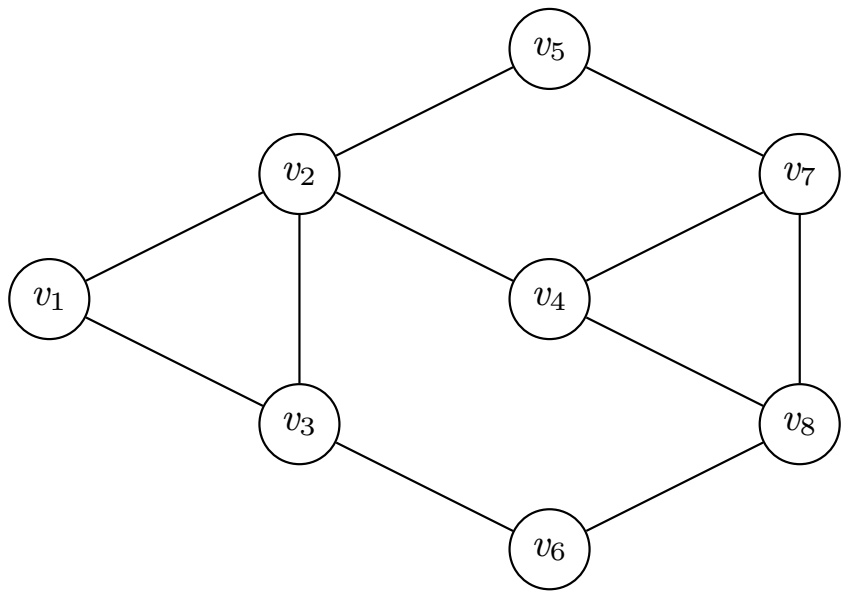
- Dans le cas de graphe non-valué, cela revient à considérer la longueur du plus court chemin.
- Dans le cas des graphes dirigés, il faut préciser si on se restreint aux chemins dirigés (auquel cas la distance n'est plus symétrique et donc plus une distance au sens mathématique !) ou si on travaille avec le graphe non-dirigé sous-jacent.
- Dans le cas de graphes non connexes, les sommets appartenant à des composantes différentes sont considérés comme ayant une distance infinie.

Distance et diamètre

Definition

Le diamètre d'un graphe est la plus grande distance existant entre deux sommets d'un graphe.

$$diam(G) = \max(d(u,v)|u,v \in V) = \min(d|\forall u,v \in V, d(u,v) \leq d)$$



- La distance de v_3 à v_5 est de 2.
- Quel est le diamètre de G ?

Pourquoi ce module

Problème

Un GPS doit trouver le chemin le plus court entre deux sommets dans un graphe valué à quelques millions de sommets (distance kilométrique, prix, ...).

Pourquoi ce module

Problème

Un GPS doit trouver le chemin le plus court entre deux sommets dans un graphe valué à quelques millions de sommets (distance kilométrique, prix, ...).

- Essayer tous les chemins est beaucoup trop long !
- Utiliser un algorithme glouton est faux (choisir à chaque carrefour la route la plus courte ne vous amènera pas au bon endroit de façon optimale).

Pourquoi ce module

Problème

Un GPS doit trouver le chemin le plus court entre deux sommets dans un graphe valué à quelques millions de sommets (distance kilométrique, prix, ...).

- Essayer tous les chemins est beaucoup trop long !
- Utiliser un algorithme glouton est faux (choisir à chaque carrefour la route la plus courte ne vous amènera pas au bon endroit de façon optimale).

Objectifs

- Trouver un algorithme intelligent.
- Déterminer sa complexité pour estimer à quel taille de graphe is sera applicable.
- Prouver qu'il est juste.

La preuve théorique est une partie essentielle. Elle ne saurait se résumer à un dessin ou un exemple !

II. Parcours de graphes : arbres couvrants

II.1 Arbres

Définition

Arbre

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

Proposition

Un graphe connexe est un arbre si et seulement si toute suppression d'arête le rend non-connexe.

En d'autres termes, un arbre est un graphe connexe minimal.

Arbre enraciné

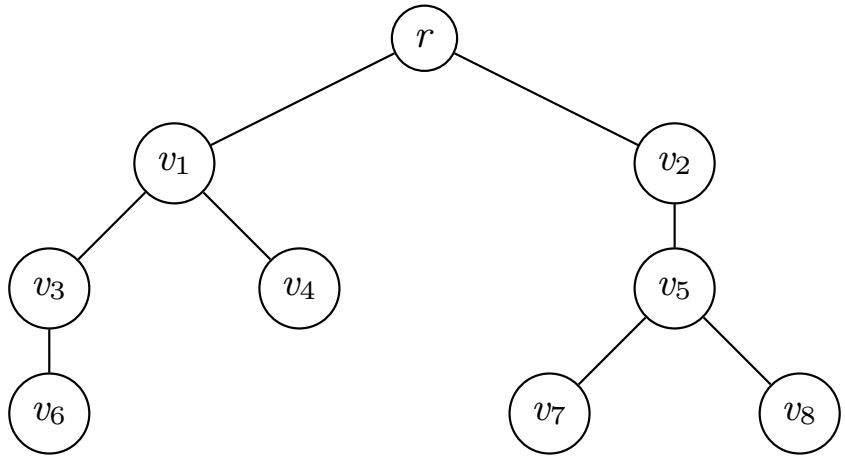
Définition

Considérons un entier n et un graphe G dont les sommets ont une étiquette appelée $niveau$ construit par le processus suivant :

- on initialise $V(G)$ à un sommet r de niveau 0, $E(G)$ à l'ensemble vide.
- pour i entre 0 et n,
  pour tout sommet v de niveau i,
  on ajoute à $V(G)$ un nombre fini de sommets $w_1, \ldots, w_{k_v}$ dont le niveau est $i+1$, et on ajoute à $E(G)$ les arêtes $\{(v, w_i), 1 \leq i \leq w_i\}$.

Le graphe G est un arbre enraciné. r est appelé racine de G.
Les sommets $(w_1, \ldots, w_{k_v}$ de niveau $i+1$ liés à un sommet v de degré i sont appelés les fils de v. v est le père de ses sommets. On en déduit la notion de descendants et d'ancêtres d'un sommet.
Un sommet sans fils est appelé une feuille.

Arbre enraciné



Proposition

Tout arbre enraciné est un arbre. De plus, pour tout arbre T et tout sommet de $r \in V(T)$, T peut être construit comme un arbre enraciné de racine r.

Conséquence : Tout arbre a $n - 1$ arêtes.

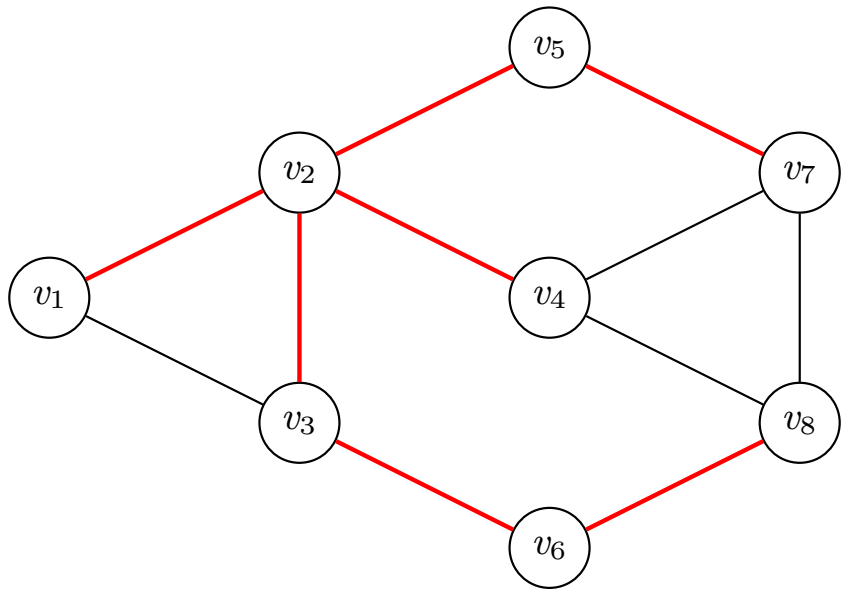
Intérêt des arbres

1. Structure adéquate pour certaines modélisations (évolution, pedigrees...). De nombreux problèmes ne pouvant pas être résolus en temps polynomial sur les graphes en général peuvent l'être en temps polynomial sur les arbres.
2. Structure naturelle pour résoudre des problèmes d'énumération.
   - Enumérer l'ensemble des mots de quatre lettres qu'on peut écrire avec ABC
   - Enumérer l'ensemble des mots de quatre lettres qu'on peut écrire avec ABC et contenant au moins 2 lettres A
3. Structure la moins lourde en termes d'arêtes pour encoder la connexité : n sommets reliés par un arbre forment un ensemble connexe, et on ne peut pas utiliser moins d'arêtes pour y arriver.

Arbre couvrant

Definition

Soit G un graphe. Un arbre couvrant de G est un sous-graphe T de G tel que T est un arbre et $V(T) = V(G)$.



Arbre couvrant

Théorème

Un graphe est connexe si et seulement si il admet un arbre couvrant.

Conséquences :

1. Un graphe connexe est un arbre si et seulement si il a $n - 1$ arêtes.
2. Décider si un graphe est connexe est équivalent à décider qu'il admet un arbre couvrant.

II.2 Parcours en largeur

Problème

Soit G un graphe.

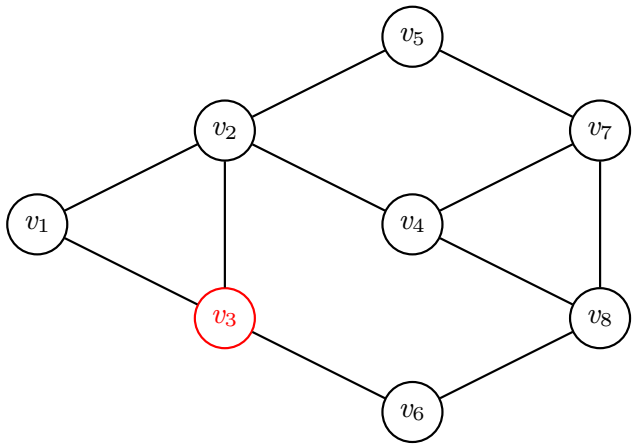
- Est-t il connexe ?
- Combien a-t-il de composantes connexes ?
- Lister tous les sommets dans la même composante connexe qu'un sommet d'intérêt.

Approche : Rechercher un (ou des) arbres couvrants, possiblement enracinés sur le sommet d'intérêt.

Arbre de parcours en largeur

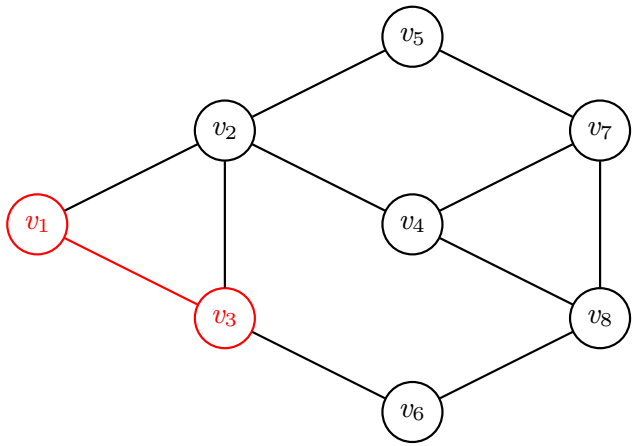
- Appelé BFS pour *Breadth-First Search*
- L'idée est de prendre un à un les sommets déjà visités et de nettoyer leur voisinage, en ajoutant tous leur voisins non-visités.
- D'un point de vue pratique, cela revient à utiliser une file, ou FIFO (First In, First Out), pour stocker les sommets visités.
- Le premier sommet de la file est le sommet courant, ses voisins sont ajoutés un à un en bout de file. Quand tous ses voisins sont visités, le sommet est supprimé de la file et ne la réintégrera plus.

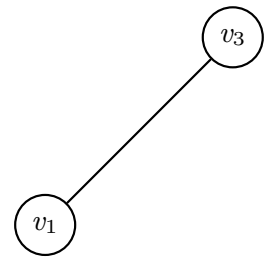
BFS : illustration



$$F = \{v_3\}$$

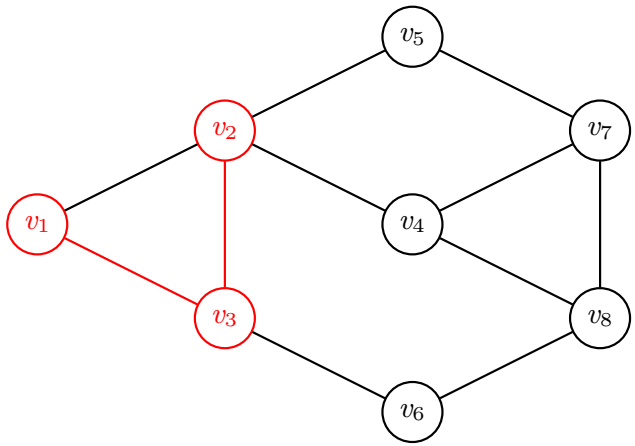
BFS : illustration

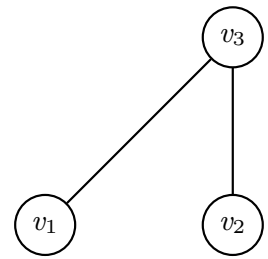




$F = \{v_3, v_1\}$

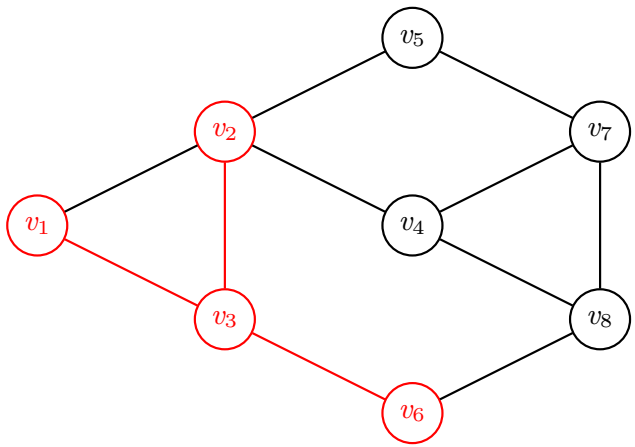
BFS : illustration

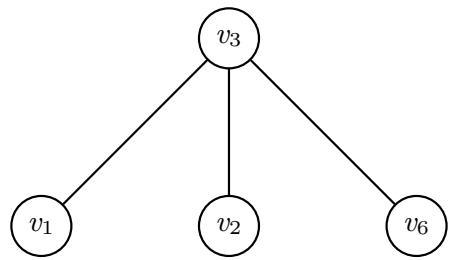




$$F = \{v_3, v_1, v_2\}$$

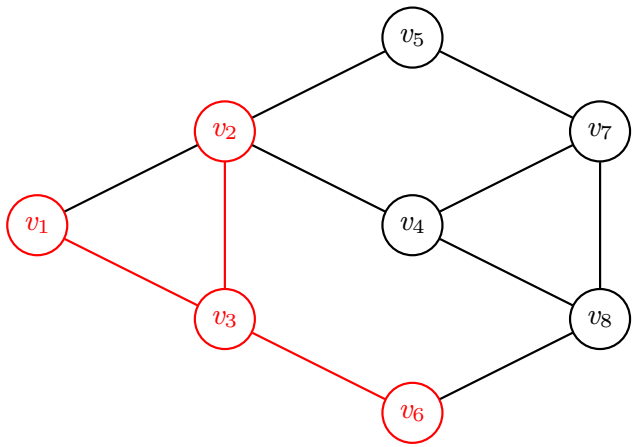
BFS : illustration

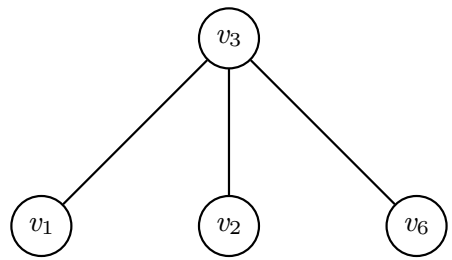




$F = \{v_3, v_1, v_2, v_6\}$
puis $F = \{v_1, v_2, v_6\}$

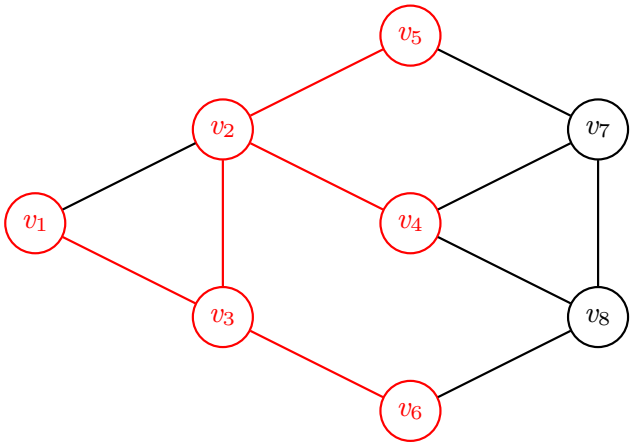
BFS : illustration

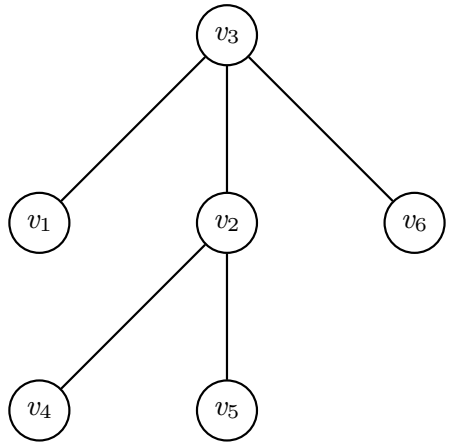




$F = \{v_1, v_2, v_6\}$
puis $F = \{v_2, v_6\}$.
Aucun sommet n'est ajouté à la file.

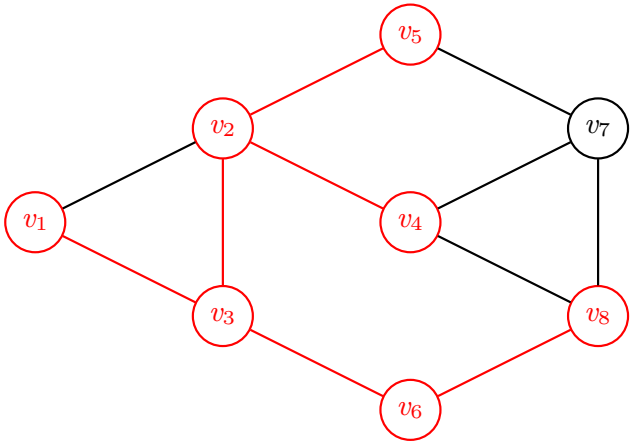
BFS : illustration

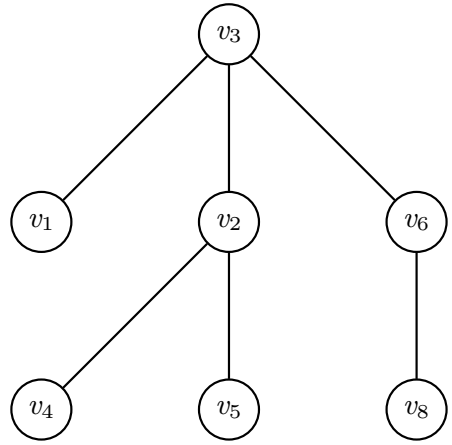




$F = \{v_2, v_6, v_4\}$
puis $F = \{v_2, v_6, v_4, v_5\}$
puis $F = \{v_6, v_4, v_5\}$

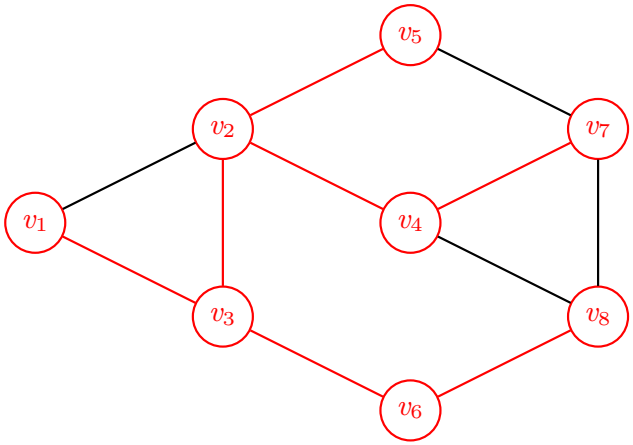
BFS : illustration

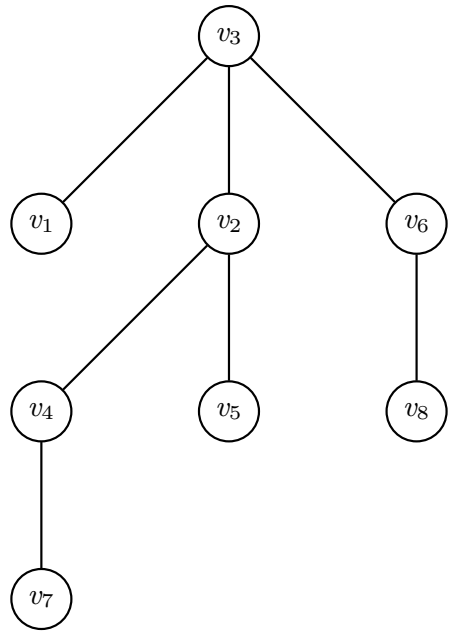




$F = \{v_6, v_4, v_5, v_8\}$
puis $F = \{v_4, v_5, v_8\}$

BFS : illustration





$F = \{v_4, v_5, v_8, v_7\}$
puis F se vide et l'algorithme s'arrête.

BFS : pseudo-code

Data: Un graphe G et un sommet v
Result: Un arbre T enraciné en v

$F = \{v\}$; $T = v$
for u *sommet de* G **do**
  u.visité=FALSE
end

while $F \neq \emptyset$ **do**
  $u =$ premier élément de F
  for w *voisin de* u **do**
    if w.*visité* =*FALSE* **then**
      w.visité=TRUE
      Ajouter w à F
      Ajouter w et (u, w) à T
    end
  end
  Supprimer u de F
end

Algorithm 1: Algorithme BFS

BFS

Proposition

BFS construit un arbre enraciné en v contenant tous les sommets de la composante connexe de v.

Proposition

BFS est de complexité $\mathcal{O}(m)$.

- Non-unicité du parcours suivant l'ordre dans lequel les voisins sont parcourus.
- En ajoutant une boucle choisissant un nouveau point de départ tant qu'il reste des sommets non-visités, on obtient un algortihme qui couvre tout graphe avec une forêt contenant autant d'arbres que le graphe a de composantes connexes.

BFS et distance

On considère un graphe G non valué.
Soit T un arbre BFS enraciné en r. Pour tout sommet u, on note $niv(u)$ le niveau de u dans T.

Proposition

Pour tout $(u, v) \in E(G)$, $|niv(u) - niv(v)| \leq 1$.

Proposition

Pour tout $u \in V(G)$, $niv(u) = d(u, r)$.

BFS orienté

On considère un graphe G orienté, et on applique BFS en n'ajoutant à chaque étape que les voisins extérieurs du sommet courant.

- Les sommets visités sont ceux qui peuvent être atteints depuis la racine par un chemin orienté.
- G ne contient aucune arête orientée d'un sommet u vers un sommet de niveau $\geq niv(u) + 2$.
- Le chemin du BFS de la racine vers tout sommet visité est un plus court chemin orienté.

II.3 Parcours en profondeur

Problème

Soit G un graphe.

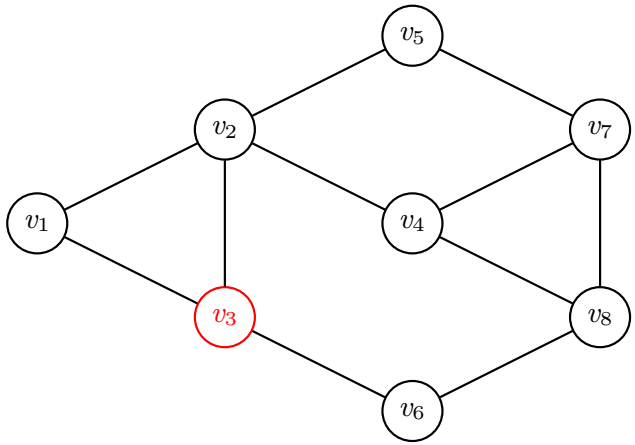
- Est-t il connexe ?
- Combien a-t-il de composantes connexes ?
- Lister tous les sommets dans la même composante connexe qu'un sommet d'intérêt.

Approche : Rechercher un (ou des) arbres couvrants, possiblement enracinés sur le sommet d'intérêt.

Arbre de parcours en profondeur

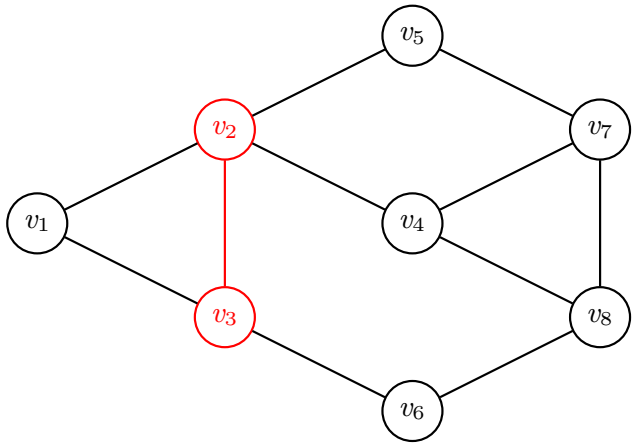
- Appelé DFS pour *Depth-First Search*
- L'idée est d'aller aussi loin que possible dans le graphe, et de rebrousser chemin quand on ne peut plus avancer.
- D'un point de vue pratique, cela revient à utiliser une pile, ou LIFO (Last In, First Out), pour stocker les sommets visités.
- Le sommet du haut de la pile est le sommet courant. S'il a un voisin non visité, celui-ci est ajouté sur la pile. S'il n'en a pas, le sommet courant est supprimé.
  Un sommet peut être le sommet courant à des moments distincts de l'algorithme mais lorsqu'il est supprimé de la pile, il ne la réintégrera plus.

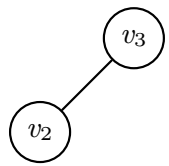
DFS : illustration



$$P = \{v_3\}$$

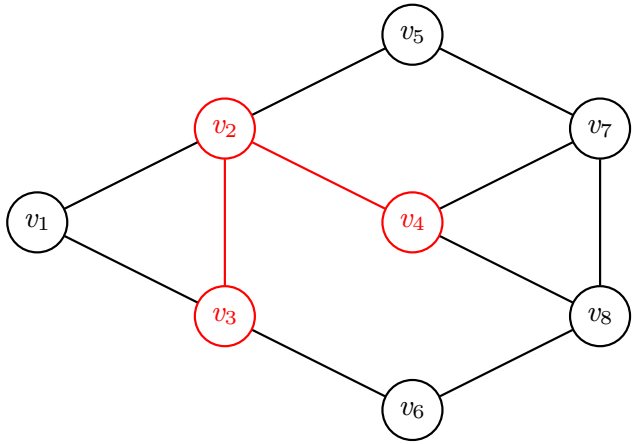
DFS : illustration

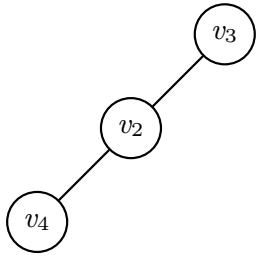




$$P = \{v_3, v_2\}$$

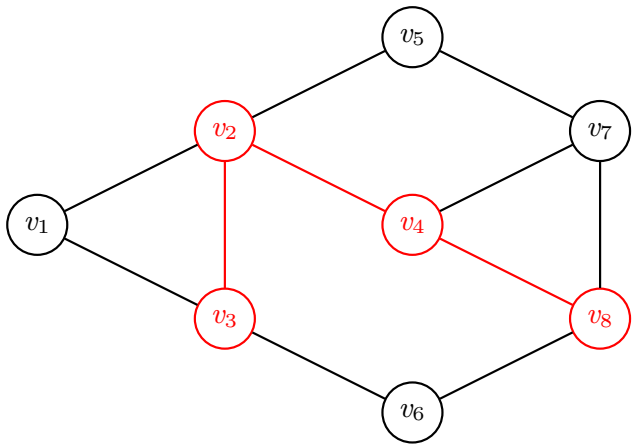
DFS : illustration

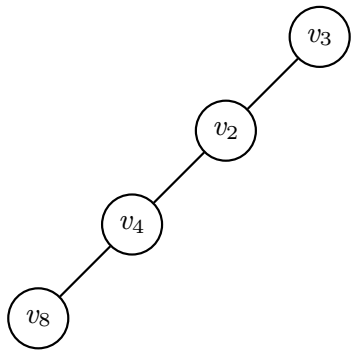




$$P = \{v_3, v_2, v_4\}$$

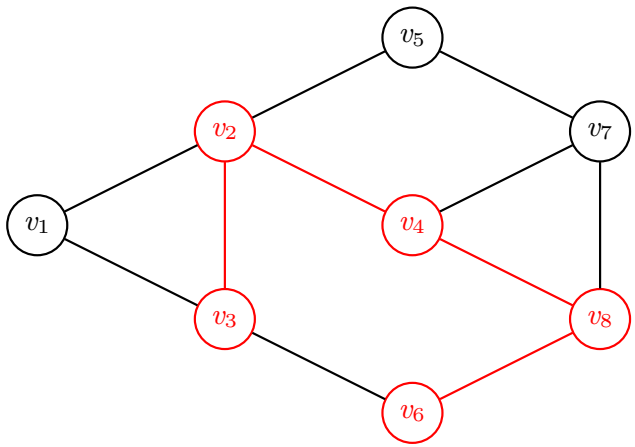
DFS : illustration

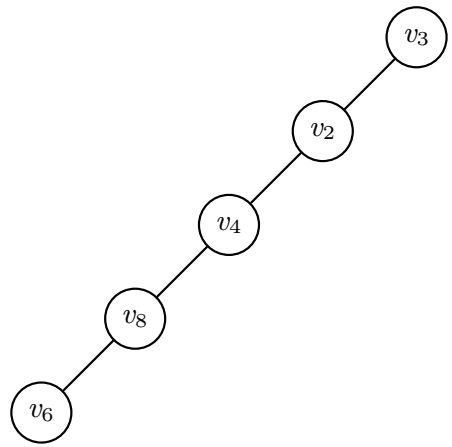




$$P = \{v_3, v_2, v_4, v_8\}$$

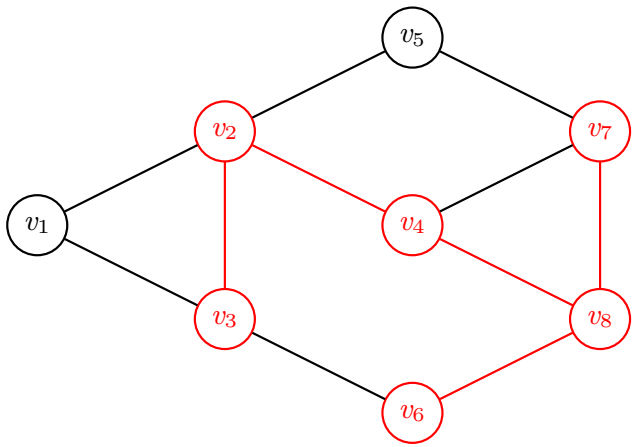
DFS : illustration

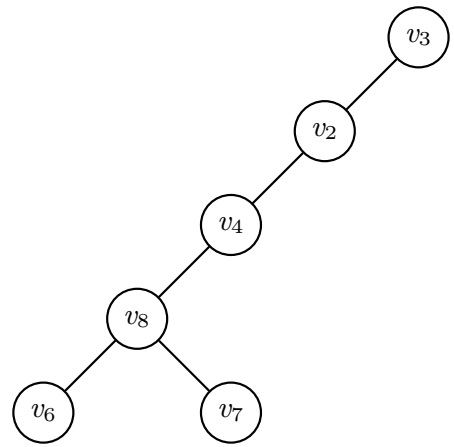




$P = \{v_3, v_2, v_4, v_8, v_6\}$
puis $P = \{v_3, v_2, v_4, v_8\}$

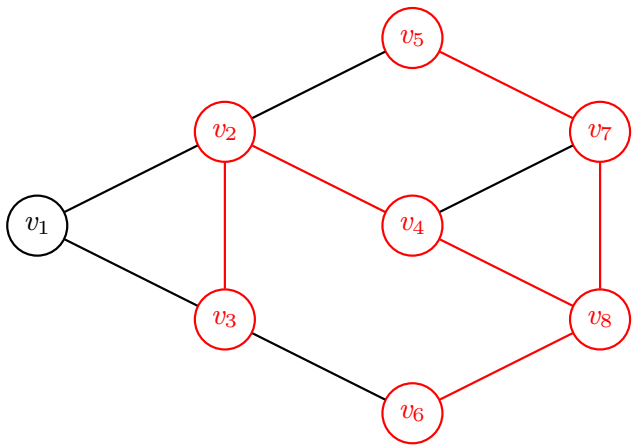
DFS : illustration

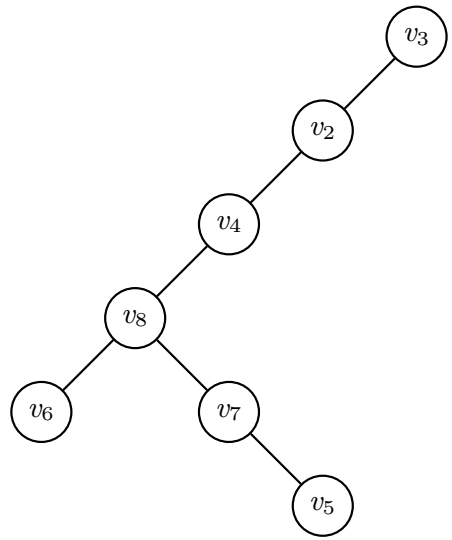




$$P = \{v_3, v_2, v_4, v_8, v_7\}$$

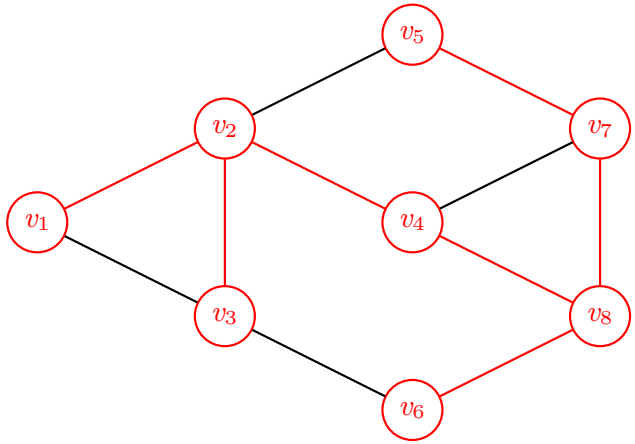
DFS : illustration

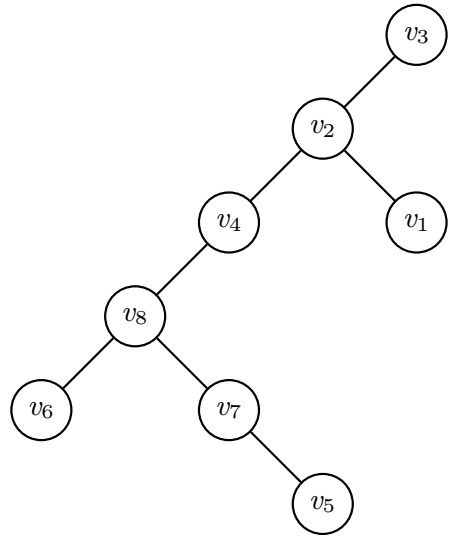




$P = \{v_3, v_2, v_4, v_8, v_7, v_5\}$
puis se dépile jusqu'à
$P = \{v_3, v_2\}$

DFS : illustration





$$P = \{v_3, v_2, v_1\}$$

puis se dépile jusqu'à être vide.

DFS : pseudo-code

Data: Un graphe G et un sommet v
Result: Un arbre T enraciné en v

$P = \{v\}$; $T = v$;
for u *sommet de* G **do**
  u.visité=FALSE
end
v.visité=TRUE

while $P \neq \emptyset$ **do**
  $u =$ dernier élément de P
  if *il existe* w *voisin de* u *avec* w*.visité=FALSE* **then**
    w.visité=TRUE
    Ajouter w à P
    Ajouter w et (u, w) à T
  end
  else
    Supprimer u de P ;
  end
end

Algorithm 2: Algorithme DFS

DFS : pseudo-code récursif

Data: Un graphe G et un sommet v
Result: Un arbre T enraciné en v

$P = \emptyset$; $T = v$
for u *sommet de* G **do**
  u.visité=FALSE
end

Function *DFS*(G,T,u)
  u.visité = TRUE
  for w *voisin de* u **do**
    if w.*visité=FALSE* **then**
      Ajouter w et (u, w) à T
      DFS(G,T,w)
    end
  end

DFS(G,T,v)

Algorithm 3: Algorithme DFS récursif

DFS

Proposition

DFS construit un arbre enraciné en v contenant tous les sommets de la composante connexe de v.

Proposition

DFS est de complexité $\mathcal{O}(m)$.

- Non-unicité du parcours suivant l'ordre dans lequel les voisins sont parcourus.
- En ajoutant une boucle choisissant un nouveau point de départ tant qu'il reste des sommets non-visités, on obtient un algorithme qui couvre tout graphe avec une forêt contenant autant d'arbres que le graphe a de composantes connexes.
- Toutes ces propriétés sont identiques avec BFS.

Topologie des DFS

Soit T un DFS sur un graphe G.

- Aucun rapport entre la distance à la racine et la distance dans G (ex : graphe réduit à un cycle).
- Par contre, aucune arête non découverte ne correspond à une arête transversale entre deux branches :

Proposition

Soit (u, v) une arête de G, u étant le premier sommet découvert par le DFS. Alors u est un ancêtre de v.

DFS orienté

On considère un graphe G orienté, et on applique DFS en n'ajoutant à chaque étape que les voisins extérieurs du sommet courant.

- Les sommets visités sont ceux qui peuvent être atteints depuis la racine par un chemin orienté.
- Si on trace les branches de T de la gauche vers la droite, il n'y a aucune arête dans G qui ajouterait à T une arête transversale de la gauche vers la droite.

II.4 Arbre couvrant de poids minimal

Problème

Soit G un graphe valué.
Trouver un arbre couvrant de poids minimal.

Applications

- directes : construire des réseaux (électriques, informatiques, ...) les moins chers possibles.
- indirectes : brique de base pour de nombreux autres algorithmes (par ex. approximation du voyageur de commerce).

Algorithme de Kruskal

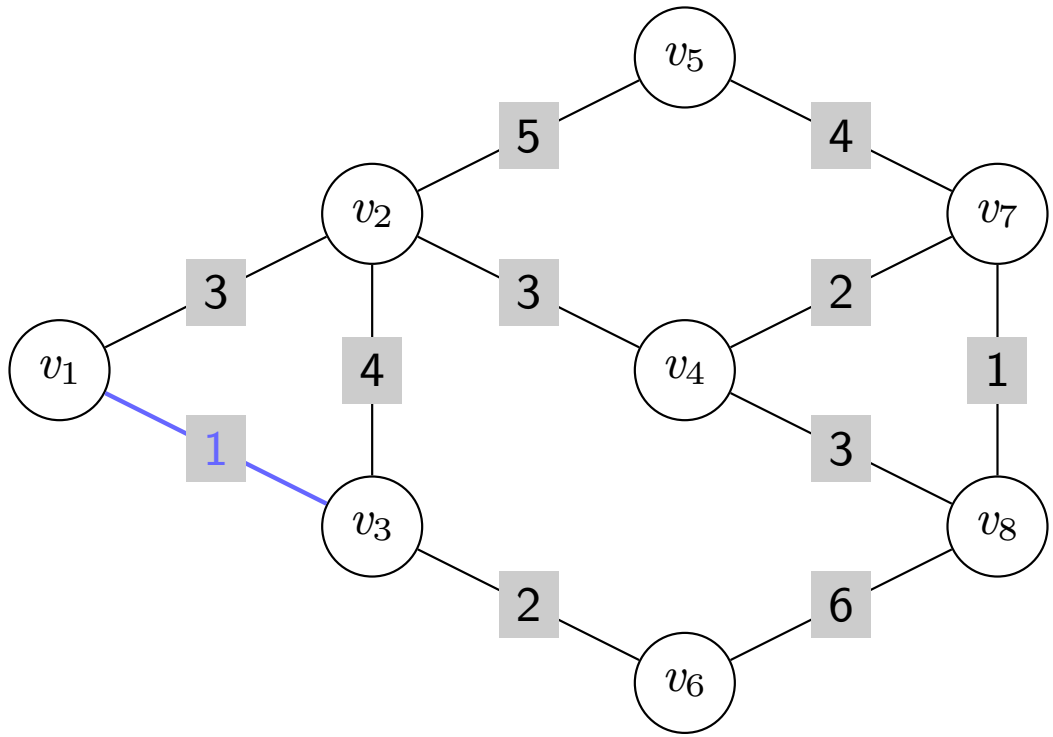
Data: Un graphe connexe valué G
Result: Un arbre couvrant T de poids minimal

Ranger les arêtes de G par poids croissant dans une liste L ;
F une forêt couvrante sans arêtes

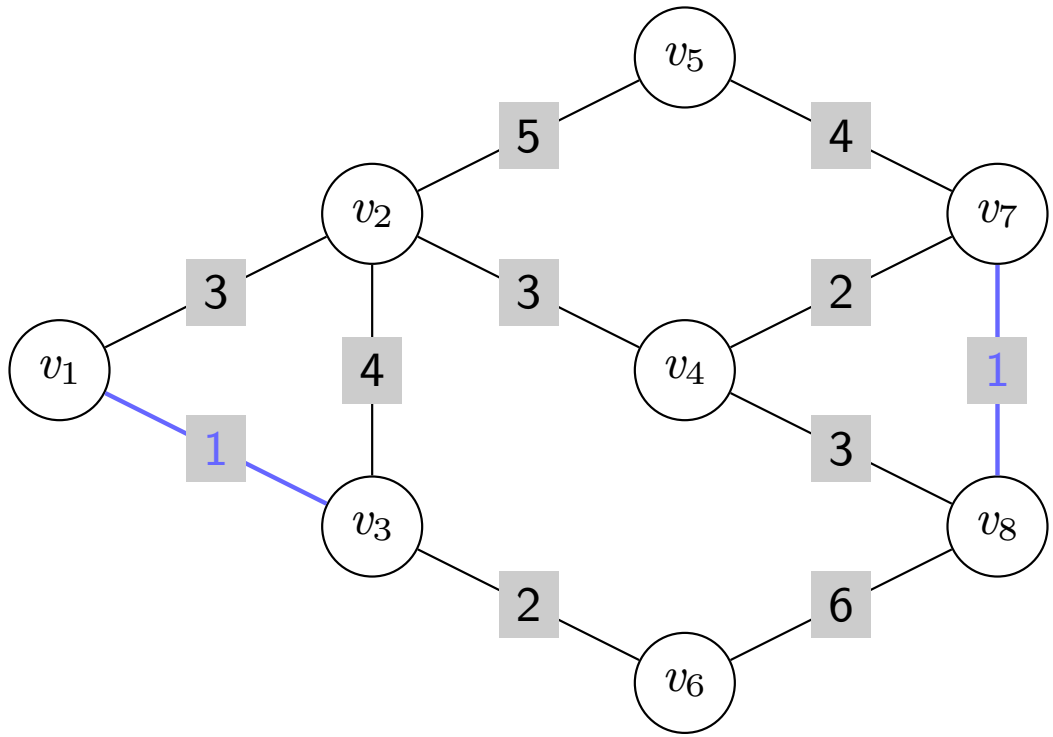
while *F n'est pas un arbre* **do**
    $e =$ première arête de L
    if *les extremités de e ne sont pas dans le même arbre de F* **then**
        $F = F + e$
    end
    Supprimer e de L
end

Algorithm 4: Algorithme de Kruskal

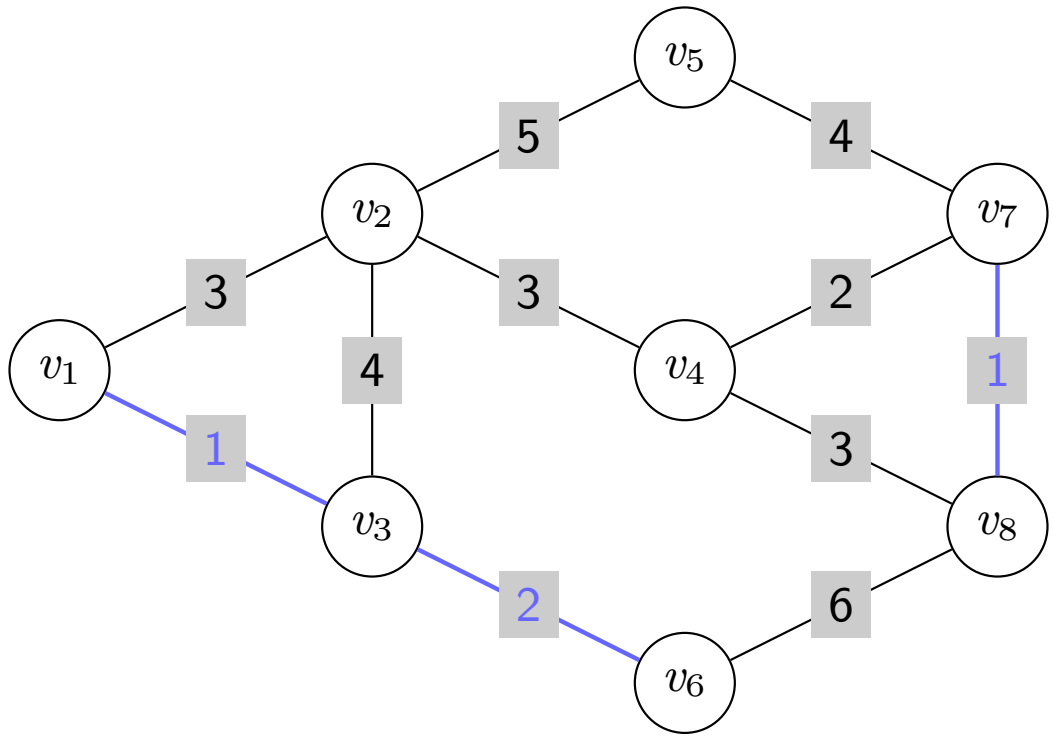
Algorithme de Kruskal : exemple



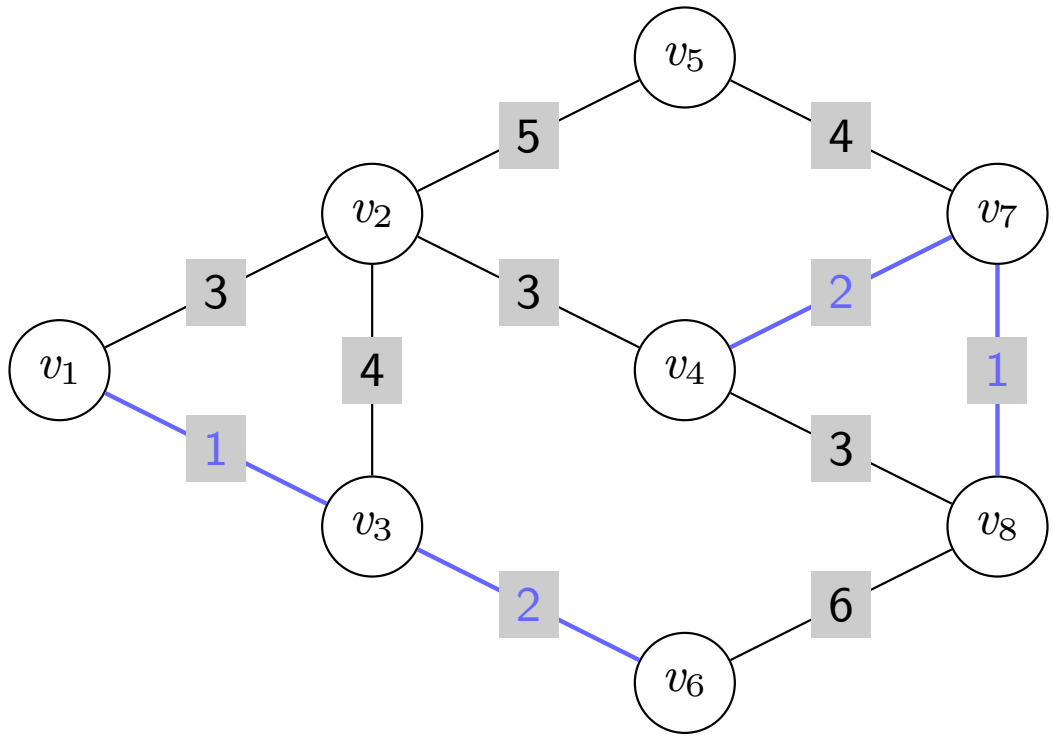
Algorithme de Kruskal : exemple



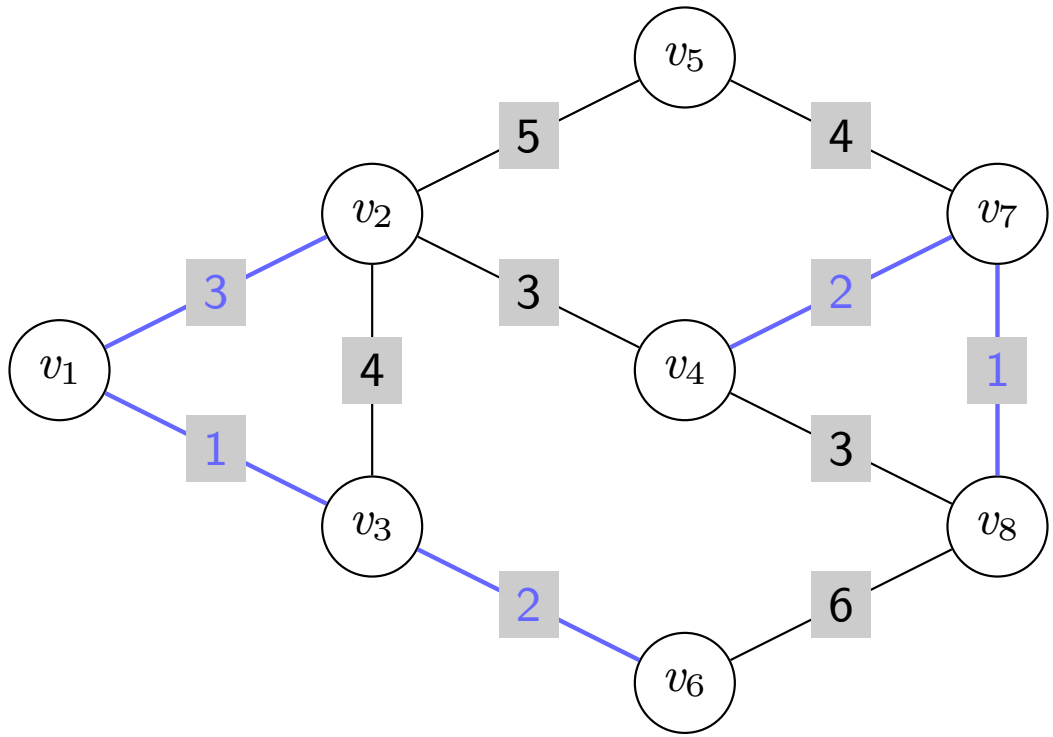
Algorithme de Kruskal : exemple



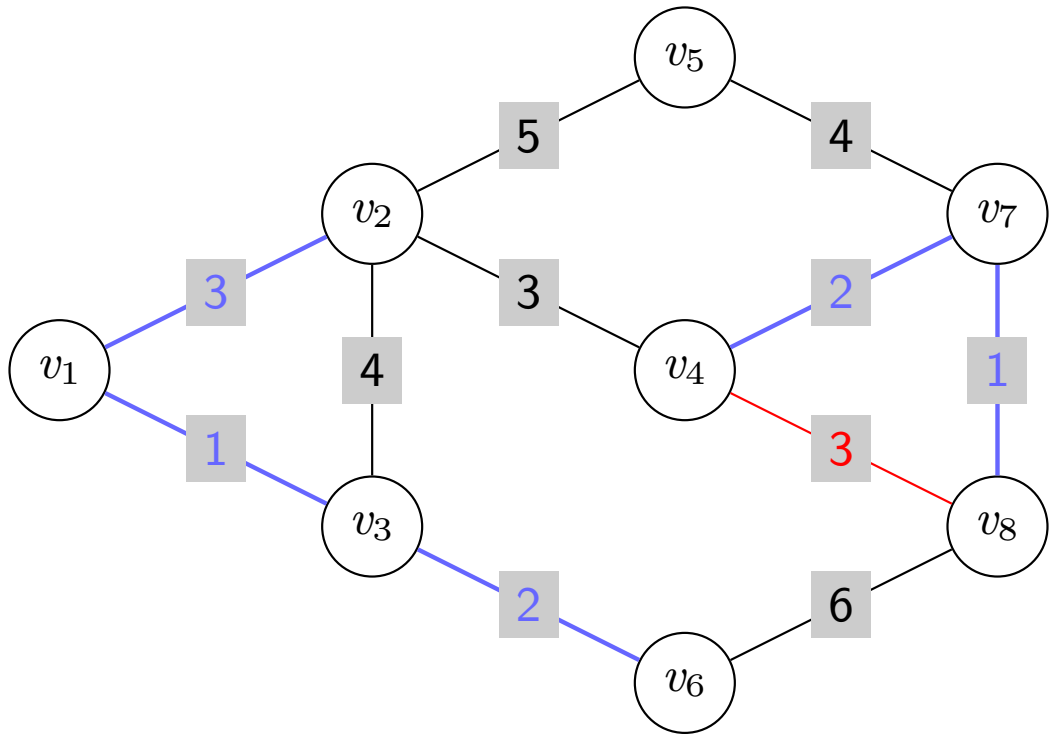
Algorithme de Kruskal : exemple



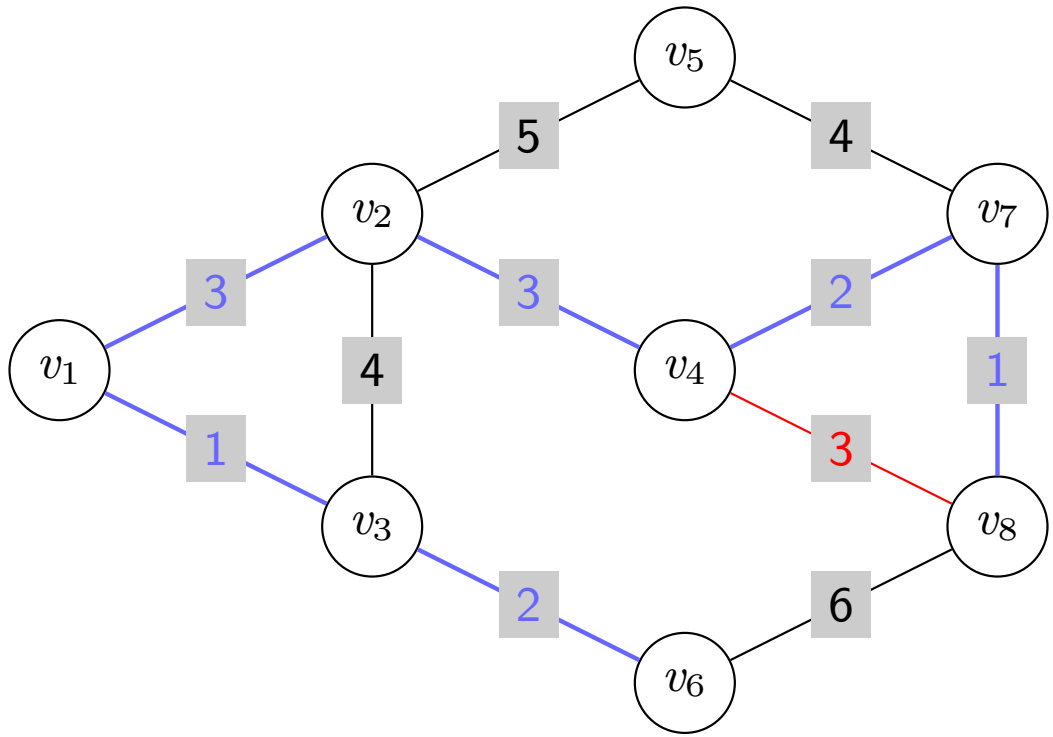
Algorithme de Kruskal : exemple



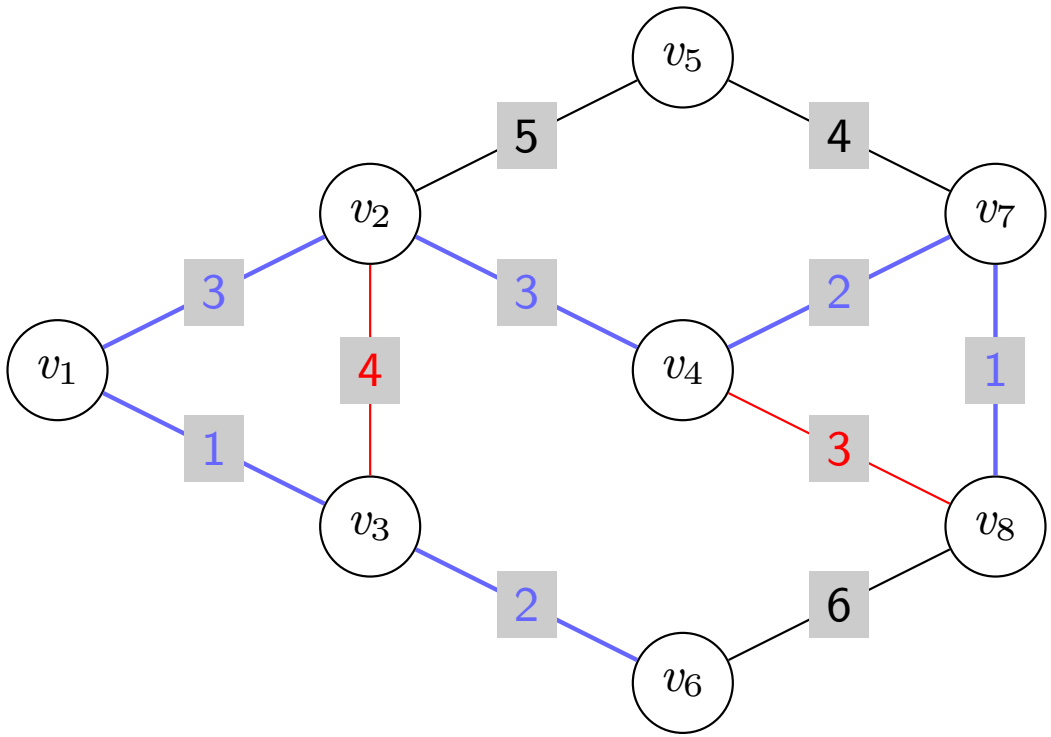
Algorithme de Kruskal : exemple



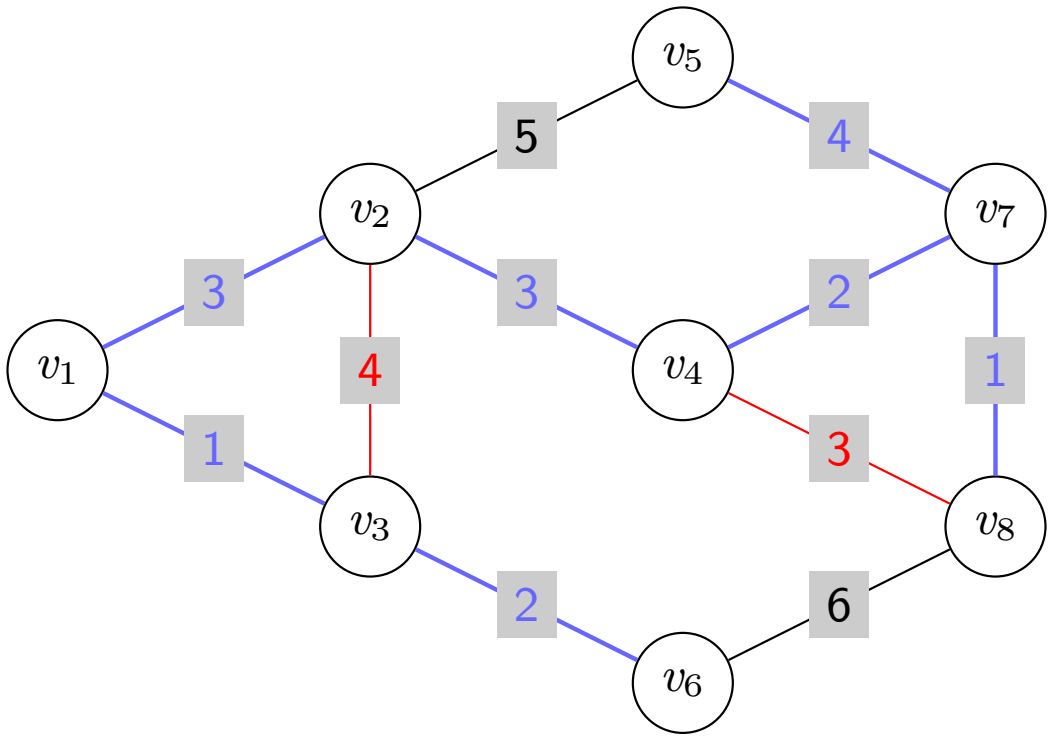
Algorithme de Kruskal : exemple



Algorithme de Kruskal : exemple



Algorithme de Kruskal : exemple



Algorithme de Kruskal

Proposition

L'algorithme de Kruskal renvoie bien, pour un graphe connexe, un arbre couvrant de poids minimal.

Proposition

Sa complexité est de $\mathcal{O}(m \log m)$.

- Si G n'est pas connexe, le résultat est une forêt dont chaque arbre est un arbre couvrant de poids minimal de l'une des composantes connexes de G.
- En cas d'existence d'arêtes de poids égal, la solution peut ne pas être unique.
- La manière de coder les composantes connexes est primordiale pour obtenir une complexité optimale.
- L'opération limitante du point de vue de la complexité est le tri.

Complexité de Kruskal : Union-Find

- Trier m arêtes se fait en $\mathcal{O}(m \log m)$
- Pour chaque arête (x, y), il faut
  - trouver la composante connexe de x et y : fonction $FIND$
  - comparer $FIND(x)$ et $FIND(y)$
  - si ils sont différents, fusionner les deux composantes connexes : opération $UNION$

Soit $\alpha(n, m)$ la somme des complexités de $FIND$ et $UNION$. La complexité de l'algorithme de Kruskal est alors $\mathcal{O}(m \log m + m\alpha(n, m))$.

Complexité de Kruskal : Union-Find

Une solution simple mais non-optimale

On stocke pour chaque sommet un nombre correspondant à sa composante connexe.

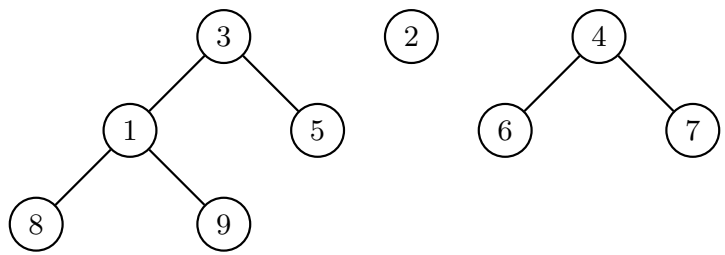
sommet	1	2	3	4	5	6	7	8	9
composante	1	2	1	3	1	3	3	1	1

$FIND$ est en temps constant, $UNION$ en $\mathcal{O}(n)$: $\alpha = \mathcal{O}(n)$.

Complexité de Kruskal : Union-Find

Autre solution : chaque composante connexe est représentée par un arbre enraciné

- il suffit de stocker son père pour chaque sommet, les racines étant leur propre père. Les racines servent de représentant de la classe.

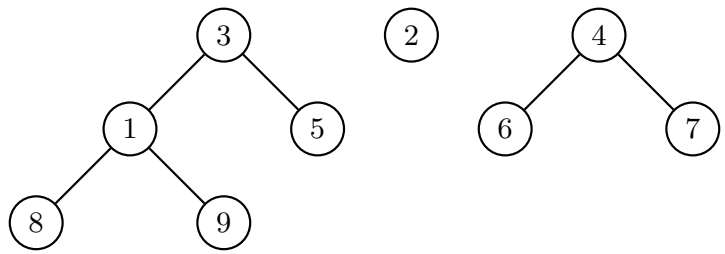


sommet	1	2	3	4	5	6	7	8	9
père	3	2	3	4	3	4	4	1	1

Complexité de Kruskal : Union-Find

Autre solution : chaque composante connexe est représentée par un arbre enraciné

- $FIND(x)$ revient à remonter l'arbre de père en père jusqu'à faire du sur-place. On est alors à la racine, et on renvoit la valeur de celle-ci : complexité $\mathcal{O}(h)$, où h est la hauteur de l'arbre.



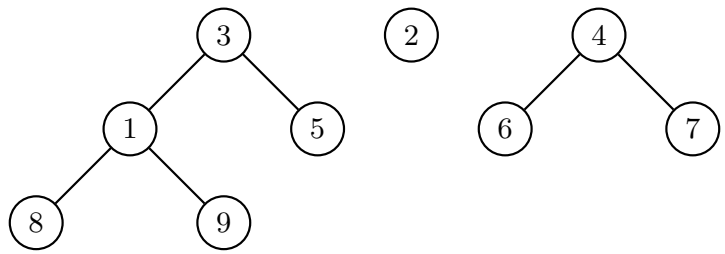
sommet	1	2	3	4	5	6	7	8	9
père	3	2	3	4	3	4	4	1	1

$FIND(8) = 3$

Complexité de Kruskal : Union-Find

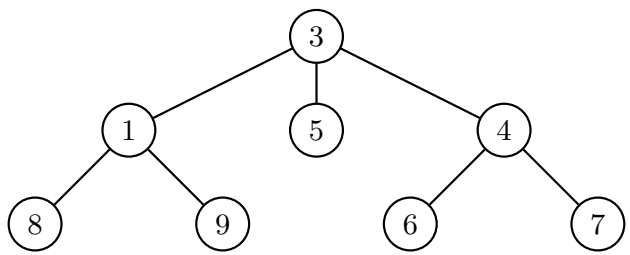
Autre solution : chaque composante connexe est représentée par un arbre enraciné

- $UNION$ consiste à rassembler les deux arbres en un seul. Pour cela, on déclare comme père de la racine du plus petit arbre la racine du plus grand : complexité en temps constant.



$UNION(1, 7)$

sommet	1	2	3	4	5	6	7	8	9
père	3	2	3	3	3	4	4	1	1



Complexité de Kruskal : Union-Find

Proposition

La hauteur d'un arbre avec s sommets, construit par une suite d'applications de $UNION$, est majoré par $1 + \log_2 s$.

On en déduit que $\alpha = \mathcal{O}(\log n)$ et donc que l'algorithme de Kruskal est de complexité $\mathcal{O}(m \log m)$.

Complexité de Kruskal : Union-Find

Proposition

La hauteur d'un arbre avec s sommets, construit par une suite d'applications de $UNION$, est majoré par $1 + \log_2 s$.

On en déduit que $\alpha = \mathcal{O}(\log n)$ et donc que l'algorithme de Kruskal est de complexité $\mathcal{O}(m \log m)$.

Complexité amortie

- Quand on remonte dans l'arbre vers la racine r, on fait de chaque sommet parcouru un fils de r.
- Opérer r $FIND$ et s $UNION$ se fait en $\mathcal{O}(r + sf(r, s))$ où f est une fonction qui croît très lentement ($f(n, m) \leq 4$ pour $n, m \leq 2^{2048}$).
- La sélection d'arêtes peut parfois être considérée comme de complexité $\mathcal{O}(m)$, ce qui permet d'obtenir une meilleure complexité (en particulier quand les poids des arêtes sont telles que le tri *radix* peut être utilisé)

Algorithme de Prim

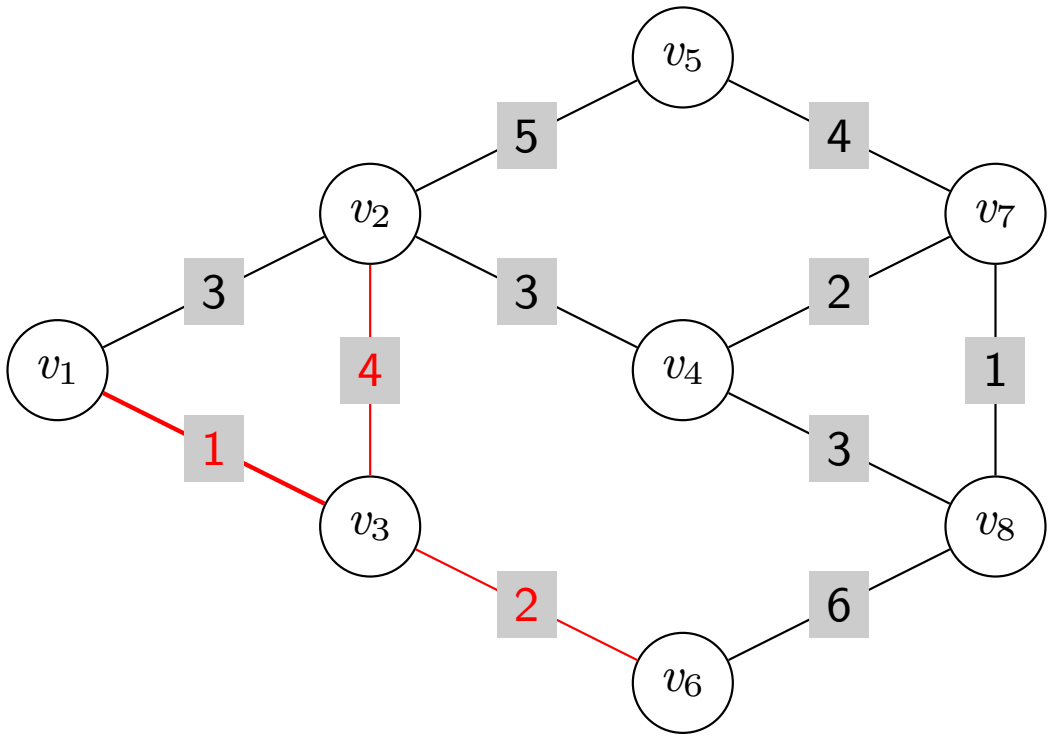
Data: Un graphe connexe valué G et un sommet v
Result: Un arbre couvrant T de poids minimal

$T = \{v\}$

while *il existe des arêtes de* T *vers* $G \setminus T$ **do**
    Trouver l'arête e de poids minimal de T vers $G \setminus T$
    $T = T + e$
end

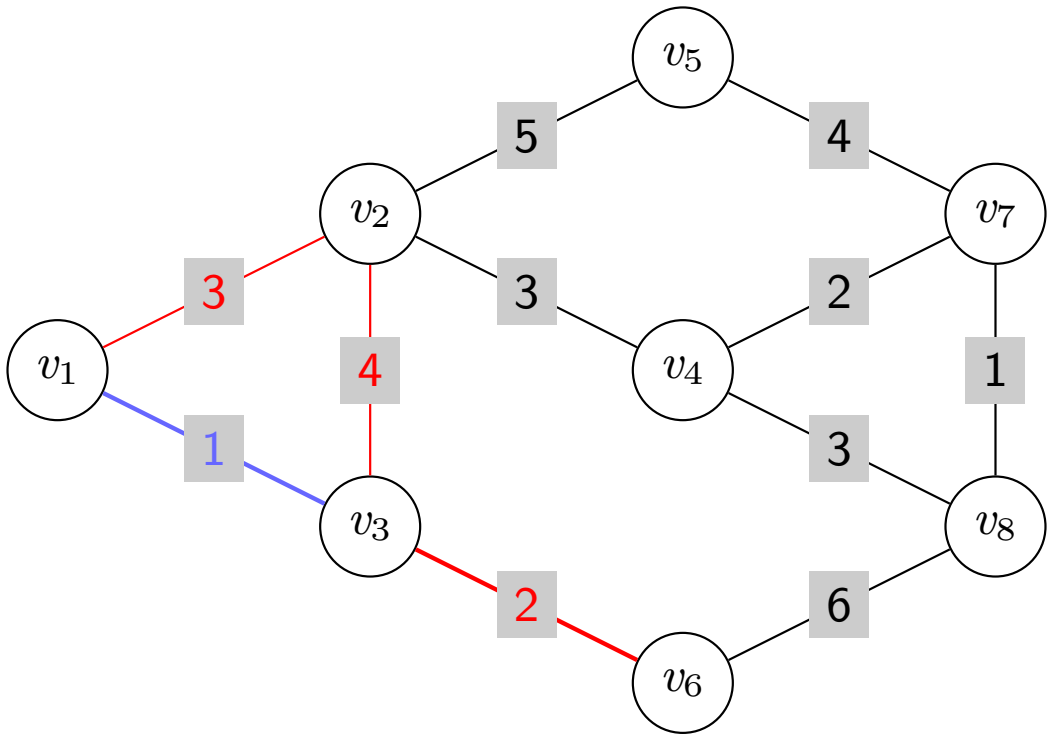
Algorithm 5: Algorithme de Prim

Algorithme de Prim : exemple



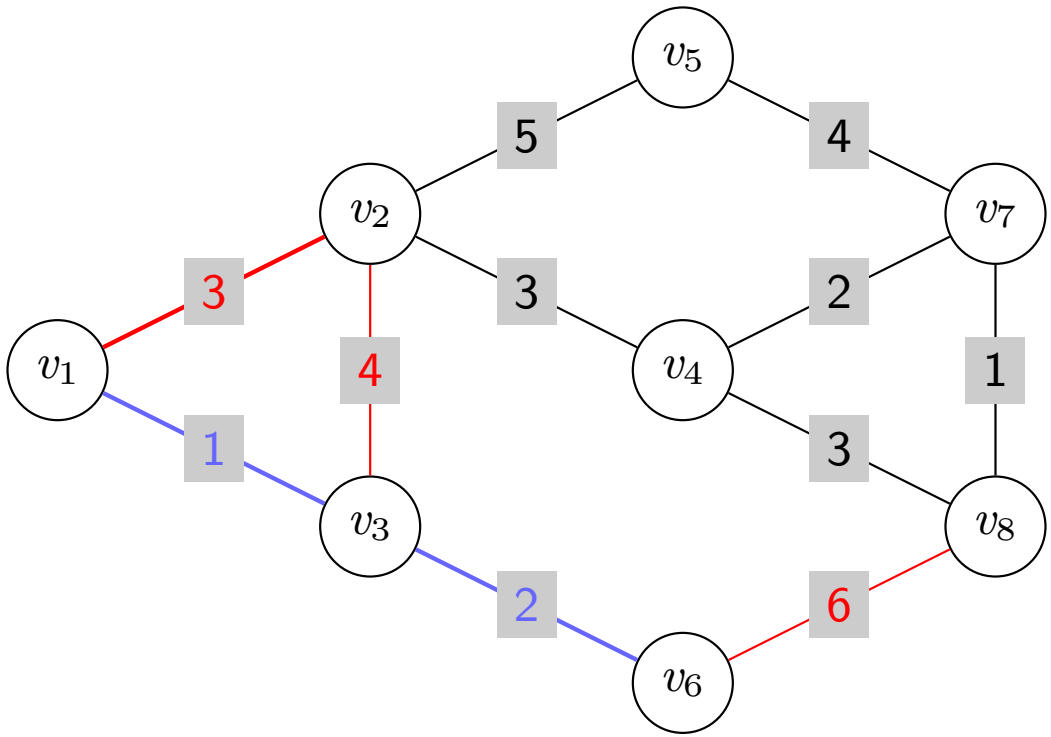
$V(T) = \{v_3\}$

Algorithme de Prim : exemple



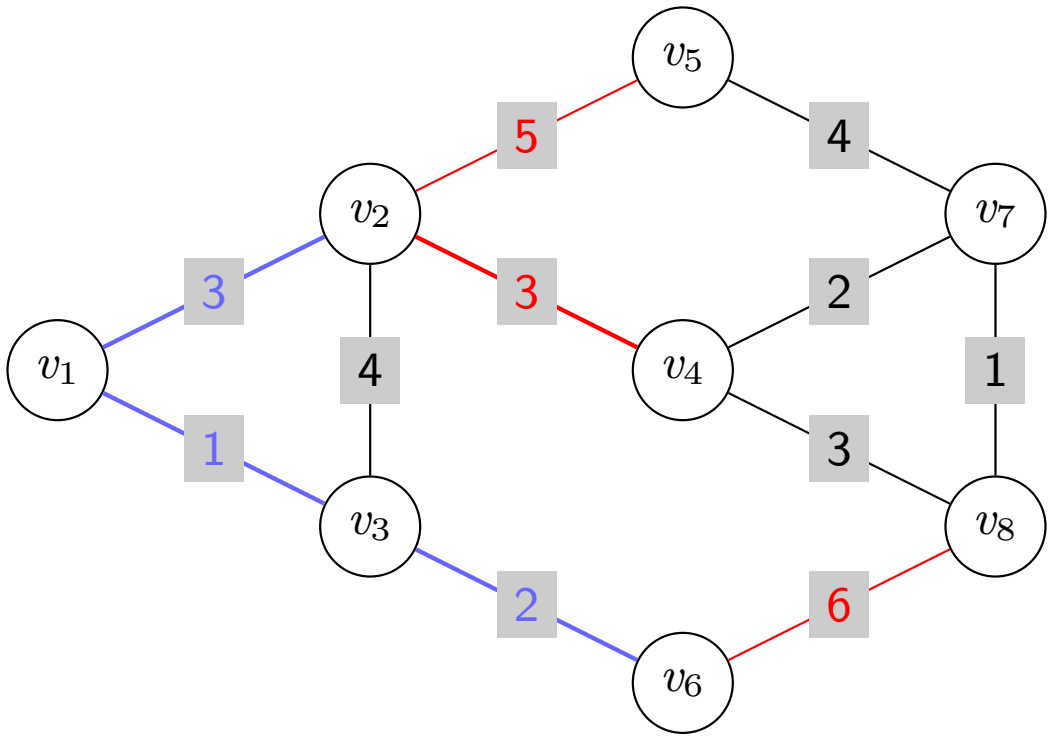
$V(T) = \{v_3, v_1\}$

Algorithme de Prim : exemple



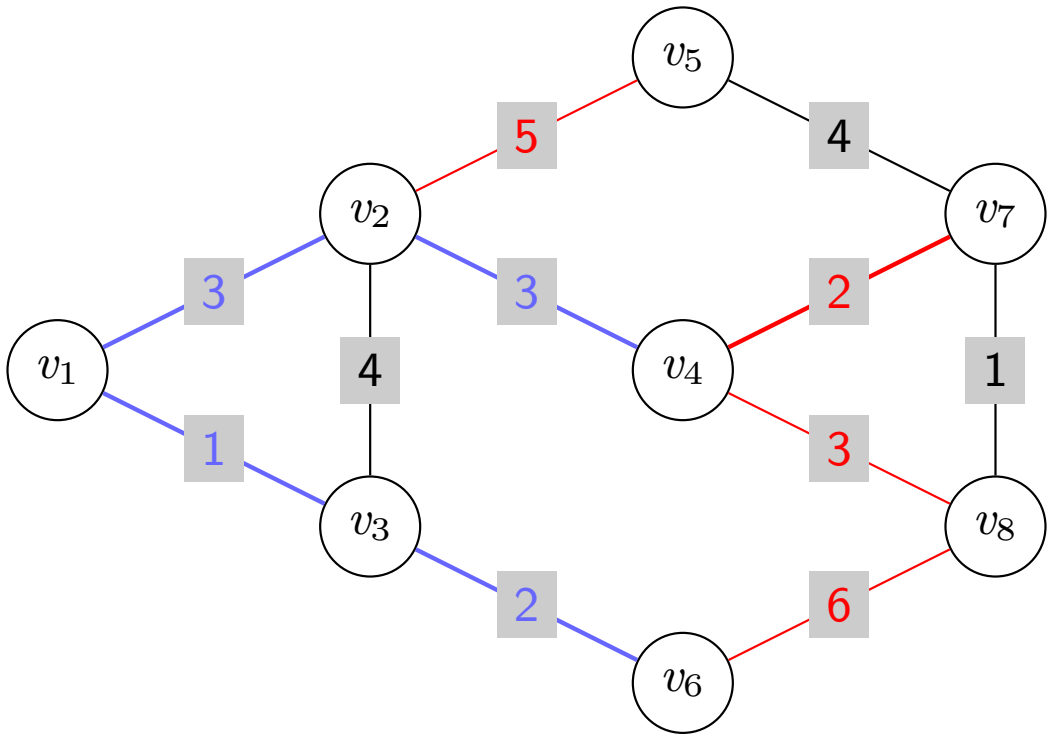
$V(T) = \{v_3, v_1, v_6\}$

Algorithme de Prim : exemple



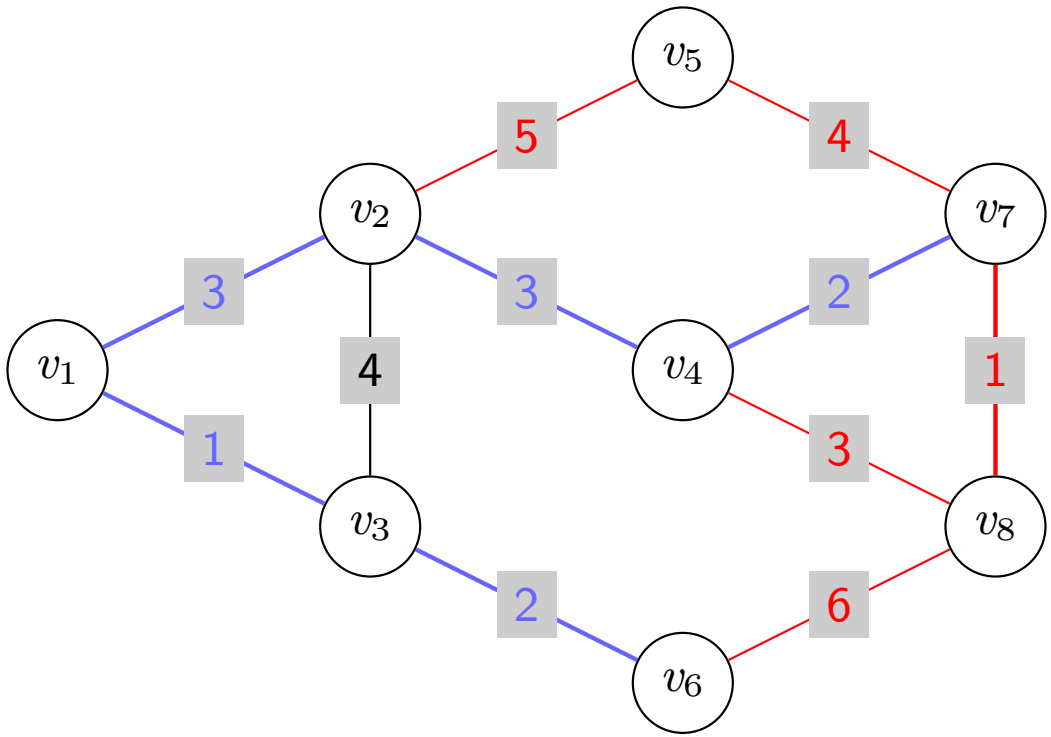
$V(T) = \{v_3, v_1, v_6, v_2\}$

Algorithme de Prim : exemple



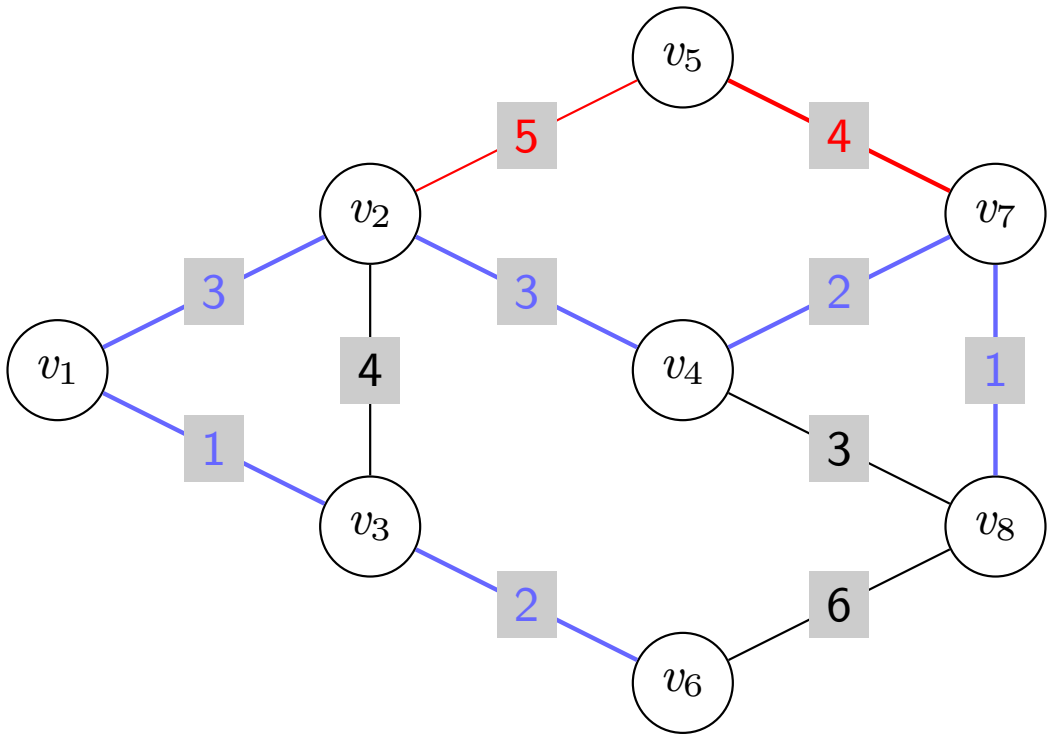
$V(T) = \{v_3, v_1, v_6, v_2, v_4\}$

Algorithme de Prim : exemple



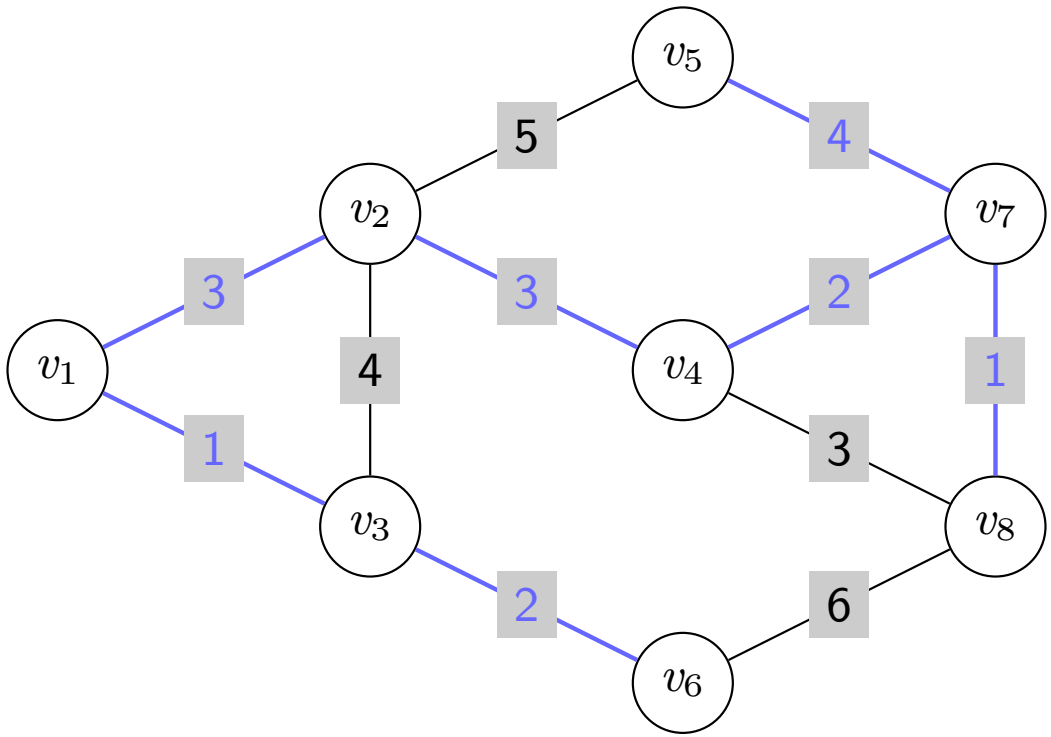
$V(T) = \{v_3, v_1, v_6, v_2, v_4, v_7\}$

Algorithme de Prim : exemple



$V(T) = \{v_3, v_1, v_6, v_2, v_4, v_7, v_8\}$

Algorithme de Prim : exemple



$V(T) = \{v_3, v_1, v_6, v_2, v_4, v_7, v_8, v_5\}$

Algorithme de Prim

Proposition

L'algorithme de Prim renvoie bien, pour un graphe connexe, un arbre couvrant de poids minimal.

Proposition

Sa complexité est de $\mathcal{O}(mn)$ pour un code naïf.
Elle peut être abaissée en $\mathcal{O}(m \log n)$ en codant à l'aide de tas binaires et en $\mathcal{O}(m + n \log n)$ à l'aide de tas de Fibonacci.

- Si G n'est pas connexe, le résultat est un arbre couvrant de poids minimal de la composante connexe de G. Le relancer tant qu'il existe des sommets non couverts permet de trouver une forêt couvrante de poids minimal.
- En cas d'existence d'arêtes de poids égal, la solution peut ne pas être unique.
- La complexité théorique (pire cas) est légèrement meilleure que celle de Kruskal au prix d'une implémentation minutieuse.
- Il ne nécessite pas l'exploration à priori de tout le graphe.

II.5 Algorithme de Dijkstra

Problème

On considère un graphe G valué et deux sommets u et v de G. Trouver le plus court chemin (au sens de la somme des poids) entre u et v.

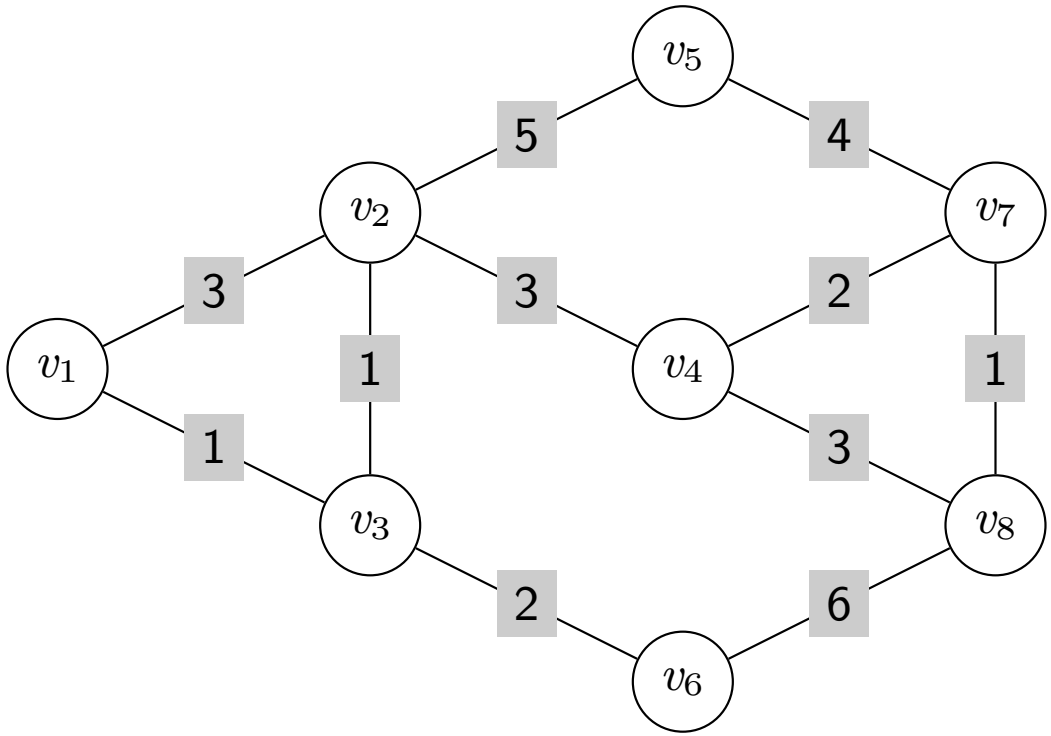
Problème

On considère un graphe G valué et deux sommets u et v de G. Trouver le plus court chemin (au sens de la somme des poids) entre u et v.

- l'approche gloutonne échoue ici.
- le problème est trivial à résoudre dans un arbre.
- l'algorithme de Dijkstra résout un problème plus général

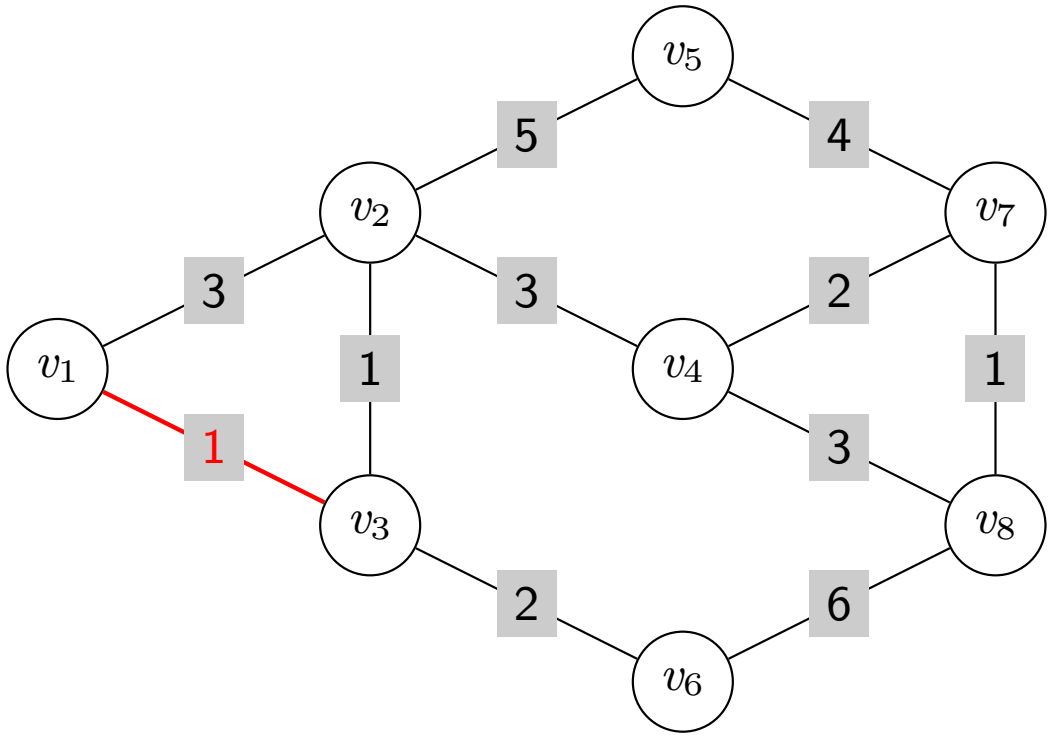
On considère un graphe valué G et un sommet u de G. Construire un arbre couvrant T_u tel que, pour tout sommet v, la distance de u à v est la même dans G et dans T_u.

Illustration



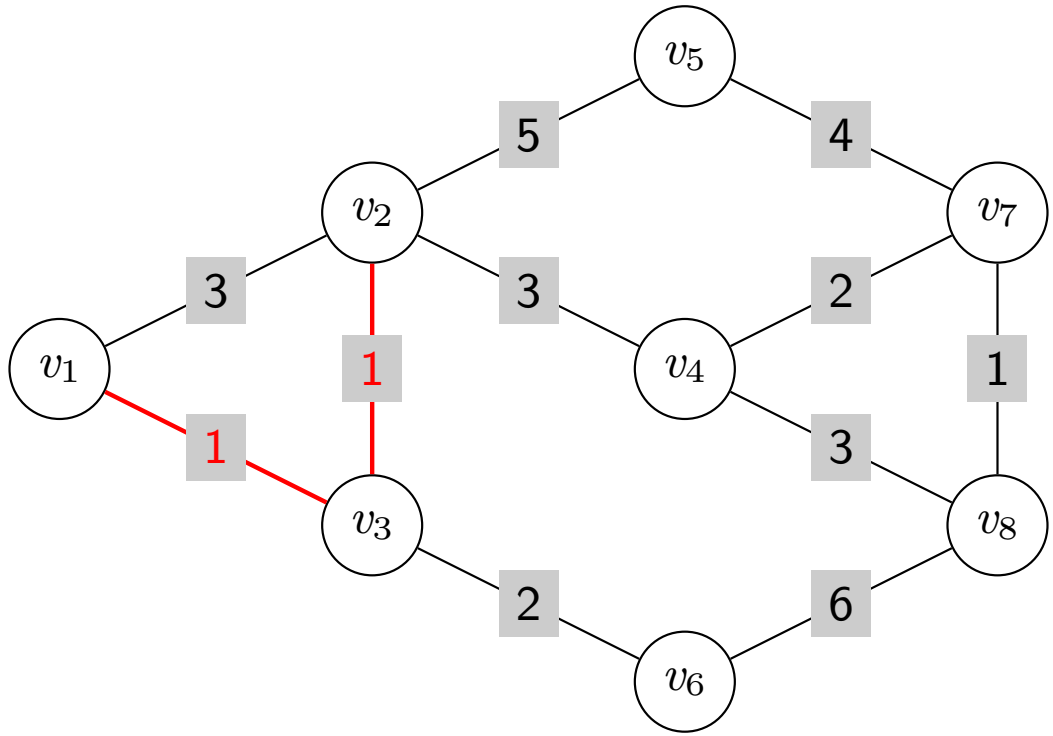
Sommet	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Distance (rouge si finale)	0	3	1	∞	∞	∞	∞	∞

Illustration



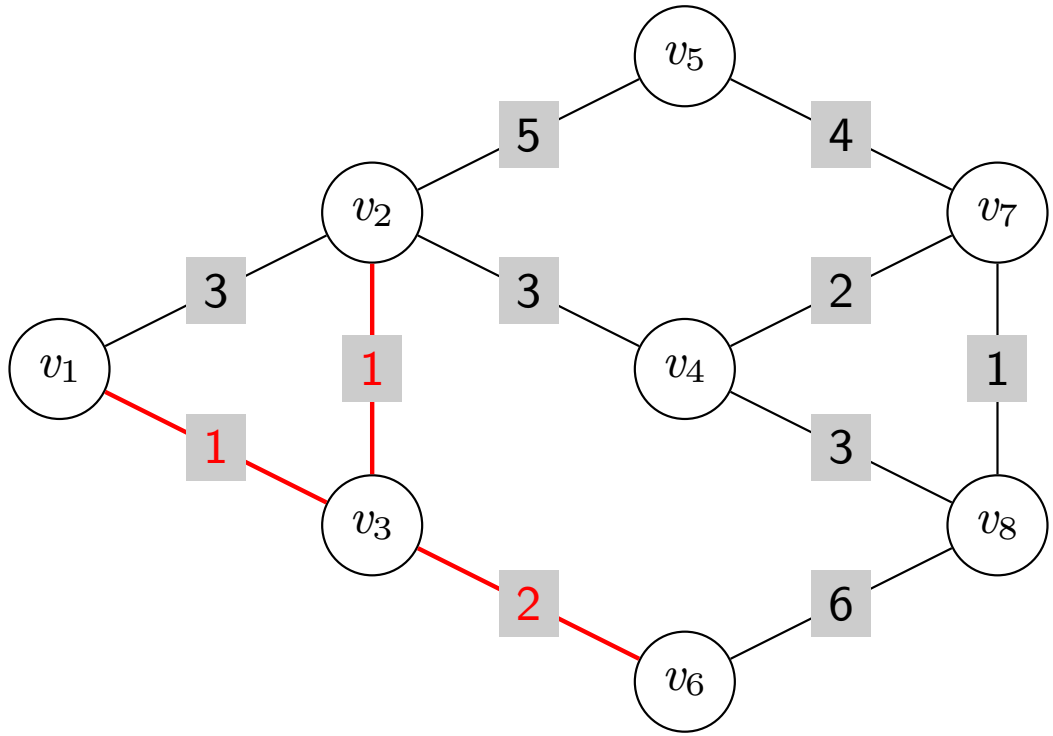
Sommet	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Distance (rouge si finale)	0	2	1	∞	∞	3	∞	∞

Illustration



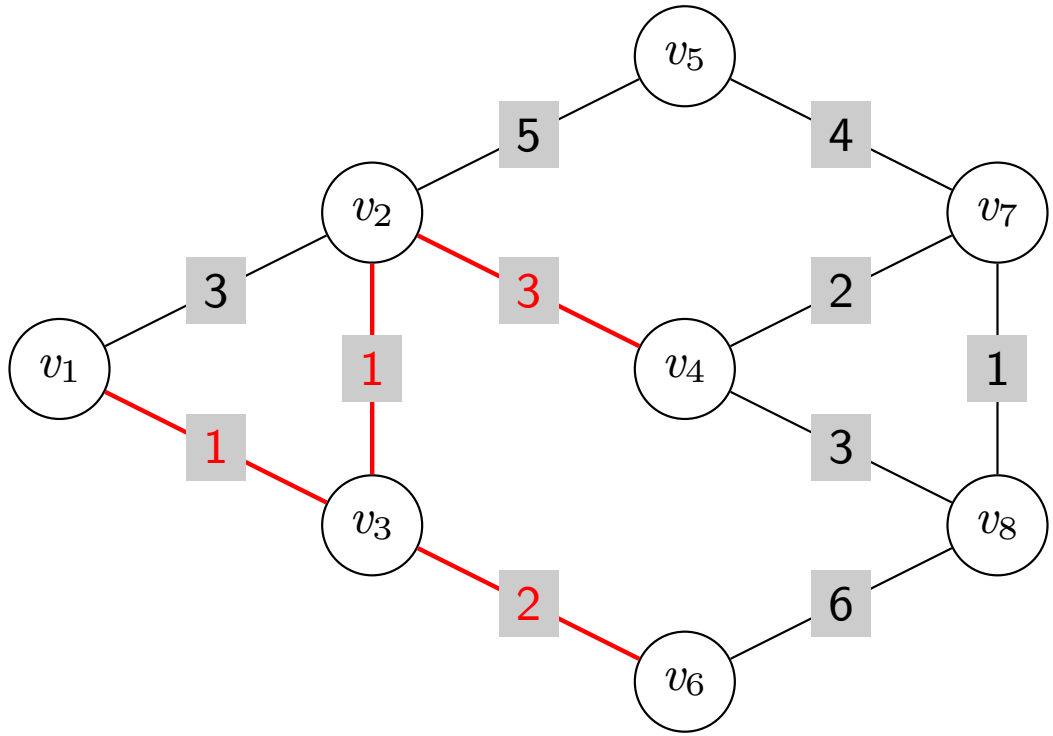
Sommet	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Distance (rouge si finale)	0	2	1	5	7	3	∞	∞

Illustration



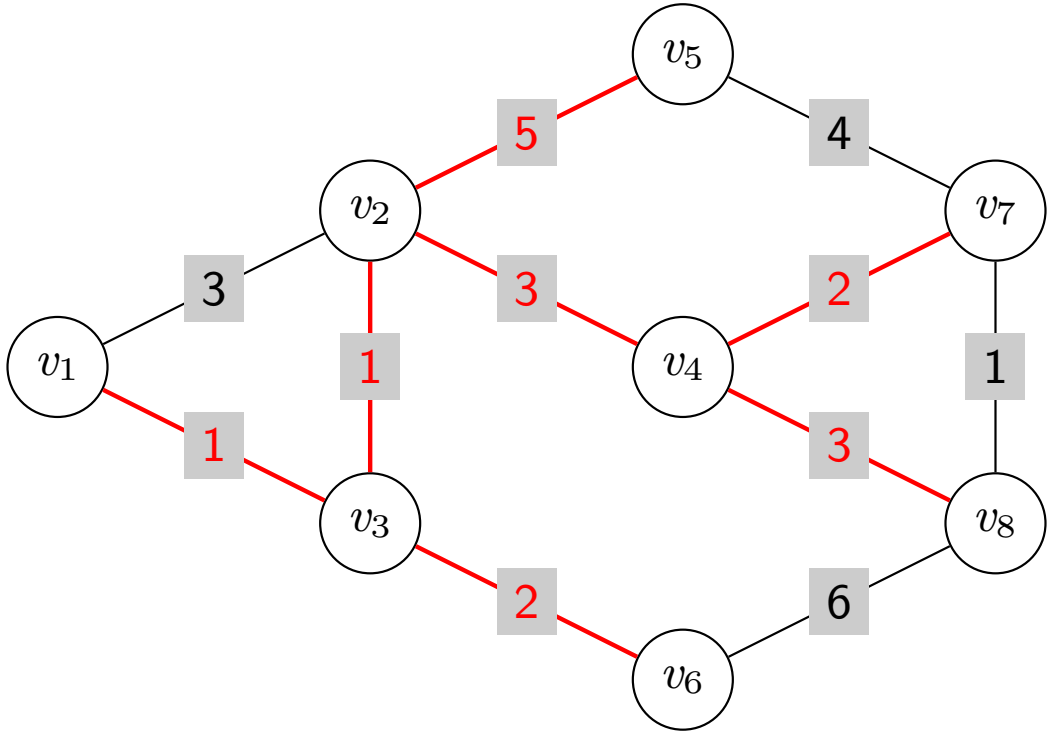
Sommet	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Distance (rouge si finale)	0	2	1	5	7	3	∞	9

Illustration



Sommet	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Distance (rouge si finale)	0	2	1	5	7	3	7	8

Illustration



Sommet	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Distance (rouge si finale)	0	2	1	5	7	3	7	8

Data: Un graphe connexe valué G de fonction de poids ω et un sommet u
Result: Un arbre couvrant T et la distance $D(v) = d_G(u, v)$ pour tout v

$T = \{u\}$
for $v \in V(G)$ **do**
  $dist\_prov(v) := \infty$; $pere(v) := \emptyset$
end
$dist\_finale(u) := 0$; $dernier\_ajout := u$

while $V(T) \neq V(G)$ **do**
  for v *voisin de* $dernier\_ajout$ **do**
    if $dist\_finale(dernier\_ajout) + \omega(dernier\_ajout, v) < dist\_prov(v)$
    then
      $dist\_prov(v) :=$
       $dist\_finale(dernier\_ajout) + \omega(dernier\_ajout, v)$
      $pere(v) := dernier\_ajout$
    end
  end
  Selectionner $v \notin V(T)$ tel que $dist\_prov$ est minimum
  Ajouter $(pere(v), v)$ a T
  $dist\_finale(v) := dist\_prov(v)$
  $dernier\_ajout := v$
end

Algorithm 6: Algorithme de Dijkstra

Algorithme de Dijkstra

Proposition

L'algorithme de Dijkstra renvoie bien la distance de tout sommet au sommet d'origine.

Proposition

La complexité est polynomiale. Elle est en $\mathcal{O}(n^2)$ pour un code naïf, et peut être réduite à $\mathcal{O}((m+n)\log n)$ à l'aide de tas binaire et même $\mathcal{O}(m+n\log n)$ à l'aide de tas de Fibonacci.

Algorithme de Floyd-Warshall

- L'algorithme de Dijkstra ne s'applique pas si le graphe contient des arêtes de poids négatif.

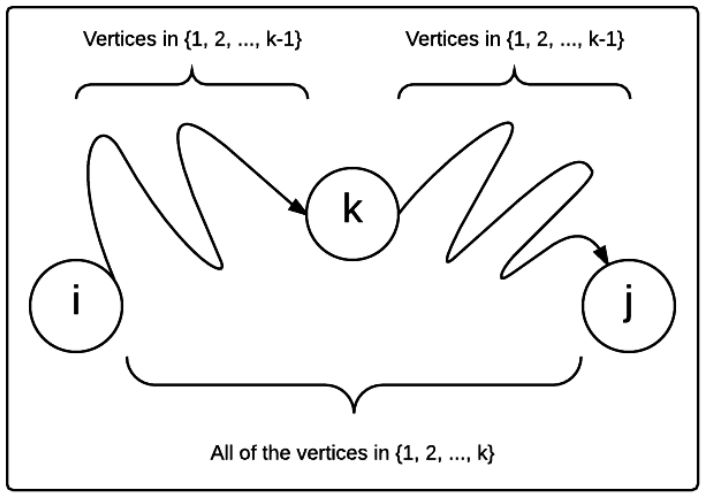
Floyd-Warshall

L'algorithme de Floyd-Warshall permet de calculer les distances entre toutes les paires de sommets, même si certaines arêtes sont de poids négatif, à condition qu'il n'y ait aucun cycle de poids strictement négatif.

Algorithme de Floyd-Warshall : principe

`https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`
On numérote les sommets de 1 à n.
On calcule pour tout k la matrice $\mathcal{W}^k$ telle que $\mathcal{W}^k_{uv}$ contient le poids du chemin de poids minimal entre u et v dont tous les sommets internes appartiennent à $\{1, \ldots, k\}$.

- $\mathcal{W}^k_{ij} = \min(\mathcal{W}^{k-1}_{ij}, \mathcal{W}^{k-1}_{ik} + \mathcal{W}^{k-1}_{kj})$
- $\mathcal{W}^n_{ij}$, $1 \leq k \leq n$ est le poids du plus court chemin.
- La complexité est de $\mathcal{O}(n^3)$.



III. Cycles couvrants - Introduction à la complexité

Problèmes considérés

Chinese Postman problem : Un postier cherche l'itinéraire le plus court pour accomplir sa tournée, sachant qu'il doit parcourir toutes les rues dont il a la charge au moins une fois, et revenir à son point de départ.

Travelling salesman problem (problème du voyageur de commerce) : Un agent commercial doit faire le tour des clients dont il a la charge et revenir à son point de départ, tout en minimisant un certain coût (euros, temps, kiomètres...).

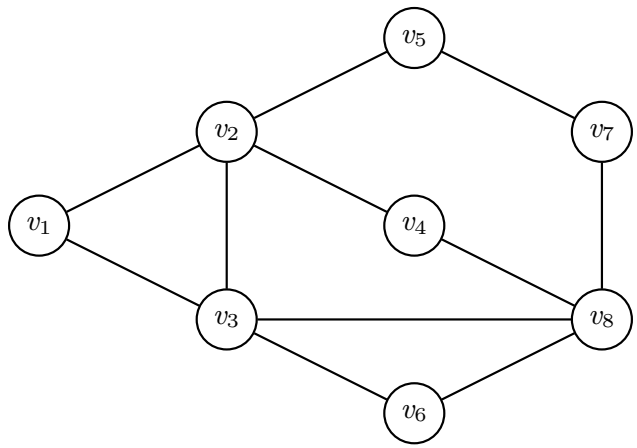
- Recherche de cycles couvrant de poids minimum dans un graphe arête-valué : le postier doit couvrir toutes les arêtes, alors que le voyageur doit 'seulement' couvrir tous les sommets.
- Cette différence rend ces deux problèmes très différents : celui du postier peut être résolu de façon exacte, alors que celui de l'agent ne peut faire l'objet que d'une résolution approchée (heuristique).

III.1 Graphes Eulériens

Graphes eulériens

Definition

Un *circuit eulérien* dans un graphe est une marche empruntant chaque arête exactement une fois et terminant en son point de départ.
Un graphe eulérien est un graphe admettant un circuit eulérien.



Circuit (ou cycle) eulérien : $\{v_1, v_3, v_6, v_8, v_7, v_5, v_2, v_4, v_8, v_3, v_2, v_1\}$

Graphes eulériens

Definition

Un *circuit eulérien* dans un graphe est une marche empruntant chaque arête exactement une fois et terminant en son point de départ.
Un graphe eulérien est un graphe admettant un circuit eulérien.

- Leonhard Euler est considéré comme le fondateur de la théorie des graphes. Il avait posé en 1738 la question de savoir s'il existait un moyen de traverser les septs ponts reliant les deux îles de la ville de Königsberg entre elles et au continent et de revenir au point de départ sans emprunter deux fois le même pont. Ce problème revient à la recherche d'un circuit eulérien.
- Dans le cas du postier, un circuit eulérien est la solution optimale. Reste à savoir s'il est possible de déterminer polynomialement l'existence d'un tel circuit.

Proposition

Un graphe connexe est eulérien si et seulement si tous ses sommets sont de degré pair.

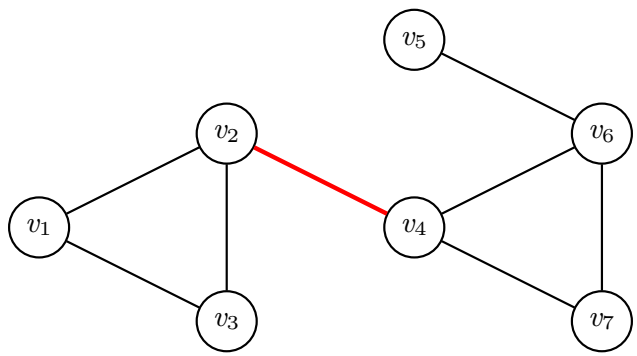
Proposition

Un graphe connexe est eulérien si et seulement si tous ses sommets sont de degré pair.

Avant de démontrer le théorème, il faut définir la notion de pont.

Definition

Un *pont* est une arête dont la suppression augmente le nombre de composantes connexes.



Proposition

De chaque côté d'un pont se trouve au moins un sommet de degré impair.

Algorithme de Fleury

`https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/`

1. Choisir un sommet v_0 et poser $W_0 = \emptyset$.
2. Supposons que le parcours (liste d'arête) $W_i = e_1 \dots e_m$ a été contruit. Soit v_m le dernier sommet de ce parcours.
3. Si il existe des arêtes adjacentes à v_m non-encore utilisées, on en choisit une appelée e_{m+1}. Sauf s'il n'y a pas le choix, on ne choisit pas e_{m+1} comme étant un pont dans $G_m = G - \{e_1, \dots, e_m\}$. On pose $W_{m+1} = W_m \cup e_{m+1}$.
4. Sinon, on arrête et on retourne W_m

On notera que quand on efface une arête après l'avoir sélectionné on retire également les points qui deviendraient isolés. On notera également que si aucune degré impair, on peut partir de n'importe quel sommet et si deux degrés impairs on part de l'un de ces deux sommets.

Chemin eulérien

Un parcours eulérien est une marche couvrant toutes les arêtes en les empruntant de façon unique (on supprime simplement la condition de retour au point de départ).

Proposition

Un graphe contient un parcours eulérien si et seulement si au plus deux de ses sommets ont un degré impair.

III.2 Chinese Postman Problem

Résolution du Chinese Postman Problem

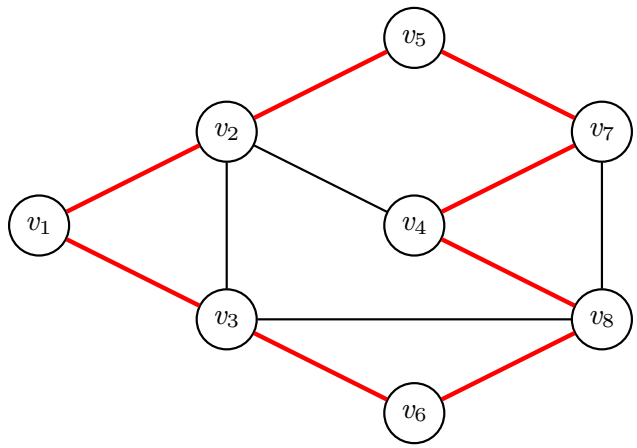
1. Enumérer l'ensemble S des sommets de degrés impair.
2. Pour toute paire de sommets de S, chercher un chemin de longueur minimale entre ces deux sommets (Algorithme de Dijkstra)
3. Créer un graphe complet H avec $V(H) = S$ dans lequel les arêtes sont valués par le poids du plus court chemin précédent.
4. Chercher un couplage parfait de poids minimal dans ce graphe. Un couplage est un ensemble d'arêtes tel que tout sommet soit incident à exactement une arête. Ce problème peut être résolu en temps polynomial par un algorithme de programmation linéeaire. (hors cadre de ce cours)
5. Doubler les arêtes le long des chemins correspondant à ce couplage minimal
6. Appliquer l'algorithme de Fleury sur le graphe résultant.

III.3 Graphes hamiltoniens

Graphe hamiltonien

Definition

Un *cycle hamiltonien* est un cycle qui passe par tous les sommets d'un graphe.
Un graphe est dit *hamiltonien* s'il admet un circuit hamiltonien.



- Dans le cas où toutes les arêtes sont de poids égaux et où il existe un circuit hamiltonien, ce dernier est la solution optimale pour le problème du voyageur de commerce.

III.4 Un peu de complexité

P et NP

- Un problème de décision est un problème dont la réponse est oui ou non.
- Un problème est dans la classe P si il peut être résolu en en temps polynomial en la taille de l'instance (ex : existence d'un chemin eulérien).
- Un problème est dans la classe NP si la validité d'une solution peut être vérifiée en un temps polynomial (ex : chemin hamiltonien).
- Un problème P' est NP-complet s'il est NP et si tout problème P dans NP peut se réduire à ce problème P' via une réduction polynomiale. Cela signifie que ce problème P' est au moins aussi difficile que tous les autres problèmes de la classe NP. son appartenance à P impliquerait que tout problème de NP est dans P. Mais en existe-t-il ?

P=NP ?

0n a facilement que P$\subset$NP. Il en résulte que :

- soit il existe un problème NP-complet qui est dans P et P=NP
- soit P et NP sont disjoints et un problème NP complet ne peut pas être résolu en un temps polynomial.

Savoir laquelle de ces assertions est vraie reste une conjecture ouverte mais la seconde est plus que vraisemblablement la bonne.

Problèmes NP-complets

- On démontre qu'un problème P_1 est NP complet en montrant que si on peut le résoudre en temps polynomial, alors on peut résoudre en temps polynomial un autre problème P_2 déjà connu comme étant NP-complet. On parle de réduction du problème P_2 au problème P_1.
- Karp (1972) a publié une liste de 21 problèmes NP-complets qui servent de référence. Le principal étant le problème 3-SAT.

Definition

Problème 3-SAT On considère n variables booléennes $x_1, \ldots, x_n$, et m clauses $C_1, \ldots, C_m$ de type *ou* incluant chacune 3 variables ou leur négation (ex : $C_1 = x_1 \vee \overline{x_2} \vee x_4$).
Décider s'il existe une affectation des variables telle que $C = C_1 \wedge \ldots \wedge C_m$ est satisfaite.

Problèmes NP-complets

- On démontre qu'un problème P_1 est NP complet en montrant que si on peut le résoudre en temps polynomial, alors on peut résoudre en temps polynomial un autre problème P_2 déjà connu comme étant NP-complet. On parle de réduction du problème P_2 au problème P_1.
- Karp (1972) a publié une liste de 21 problèmes NP-complets qui servent de référence. Le principal étant le problème 3-SAT.
- Ces notions s'étendent aux problèmes qui ne sont pas des problèmes de décision mais peuvent se décomposer en un nombre polynomial de problèmes de décision.
  Ex : trouver la longueur maximale d'un chemin se décompose en le problème de décision pour l'existence d'un chemin de longueur k pour tout k entre 1 et n.

III.5 Retour au voyageur de commerce

NP-complétude

Proposition

Le problème de décision du cycle hamiltonien est NP-complet.

Proposition

Le problème de décision du voyageur de commerce est NP-complet.

Il faut donc se contenter d'heuristiques, c'est-à-dire d'algorithmes donnant une solution approchée.