

L3 Informatique • Algorithmique avancée

TD 2 – Arbres couvrants

Cours partie 2 : Parcours de graphes : arbres couvrants

Arbre

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

Connexité, graphe connexe (Définition) : Un graphe est connexe si, pour toute paire de sommets u et v de G , il existe un chemin entre u et v .

Un graphe acyclique est un graphe ne contenant aucun cycle.

Dans un graphe non orienté, un cycle est une suite d'arêtes consécutives (chaîne simple) dont les deux sommets extrémités sont identiques.

Arbre couvrant

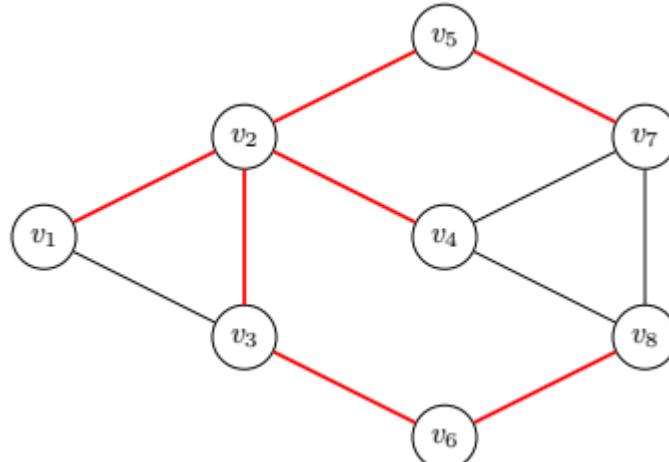
Définition

Soit G un graphe.

Un arbre couvrant de G

est un sous-graphe T de G
tel que

- T est un arbre
- Et $V(T) = V(G)$.



Un arbre couvrant d'un graphe G est un sous-graphe de G qui est connexe, sans cycle et contient tous les sommets de G .

Une forêt est une réunion disjointe d'arbres.

→ Proposer à partir du **parcours en profondeur** un procédé permettant de déterminer une **forêt couvrante** pour tout graphe G.

→ Dans le cas non-orienté, que peut-on dire du nombre d'arbres composant la **forêt couvrante** ?

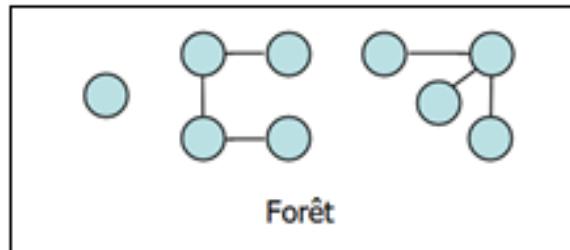
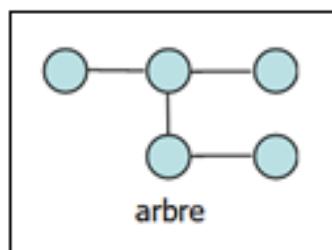
→ En déduire que la taille de la forêt ne dépend pas du sommet de départ choisi.

→ Les constatations précédentes sont-elles encore vraies dans le cas orienté ?

Exercice 2.1 Exercice de cours

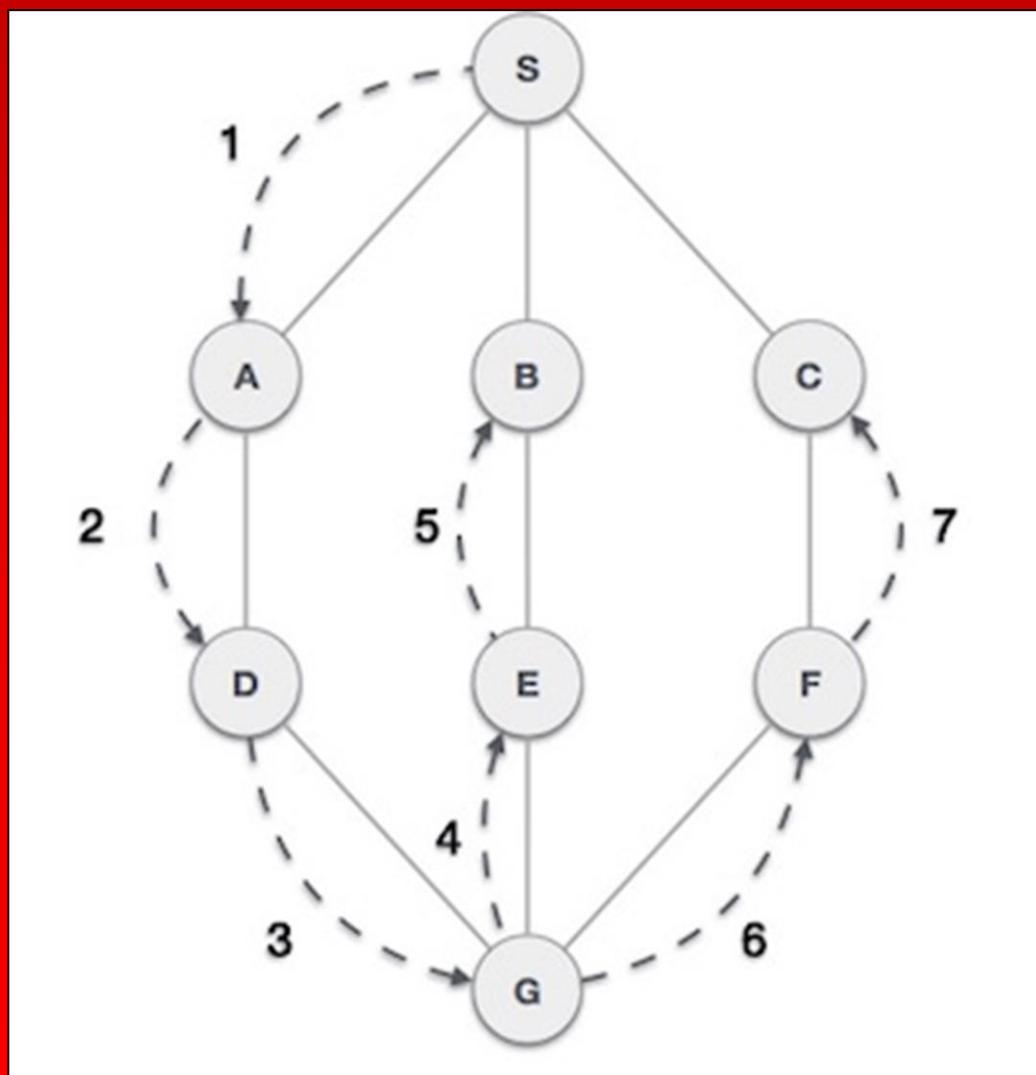
Une forêt est un graphe non orienté, acyclique, dont tous les composants connectés sont des arbres.

En d'autres termes, le graphique se compose d'une **union disjointe** d'arbres.



Soient G un graphe non-orienté et v un sommet de G .
On peut construire le parcours en profondeur de G partant de v ,

Cet algorithme revient à parcourir le graphe en avançant tant que possible et en remontant lorsque ce n'est plus possible, c'est-à-dire au père du sommet courant si ce dernier n'a pas de voisin non visités. Il s'arrête lorsque le sommet courant n'a plus de voisin non visité et que c'est la racine.



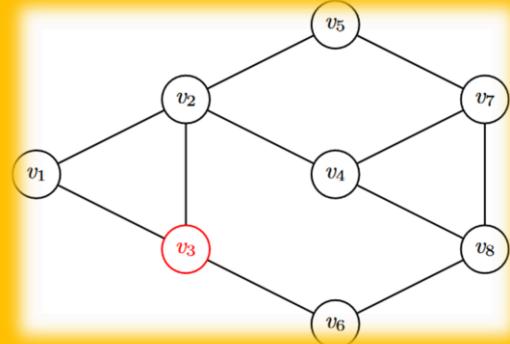
Source

https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

*Utiliser une **pile**, ou LIFO (Last In, First Out), pour stocker les sommets visité*

P : une pile pour stocker les sommets visités

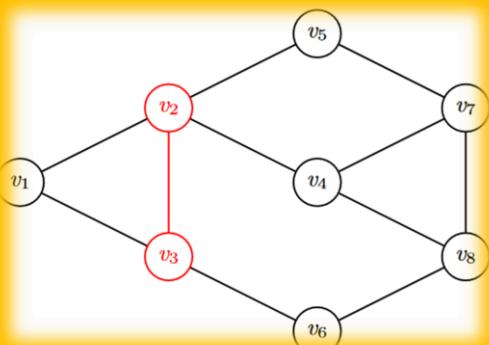
1



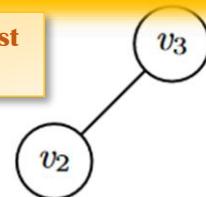
Le sommet du haut de la pile est le sommet courant.

$$P = \{v_3\}$$

2



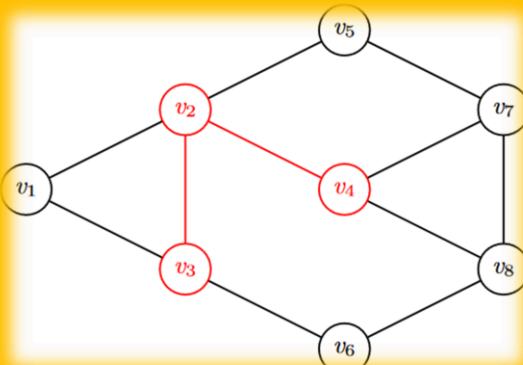
S'il a un voisin non visité, celui-ci est ajouté sur la pile



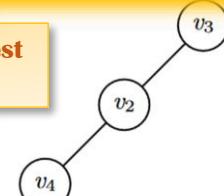
Le sommet du haut de la pile est le sommet courant.

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

3



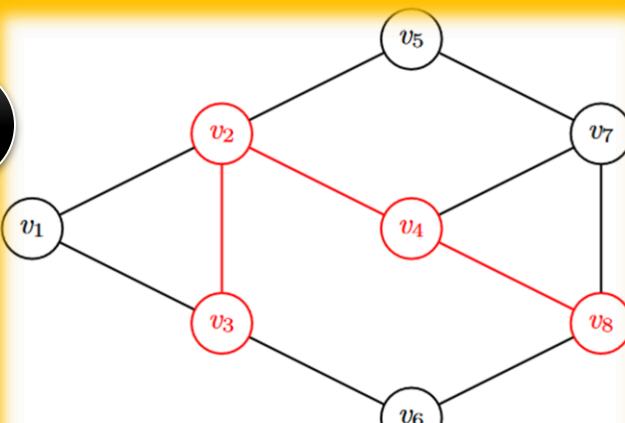
S'il a un voisin non visité, celui-ci est ajouté sur la pile



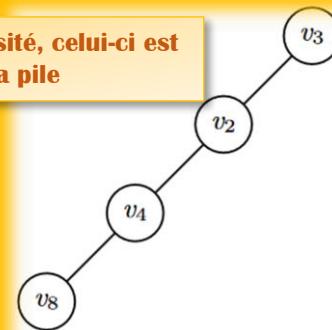
Le sommet du haut de la pile est le sommet courant.

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

4



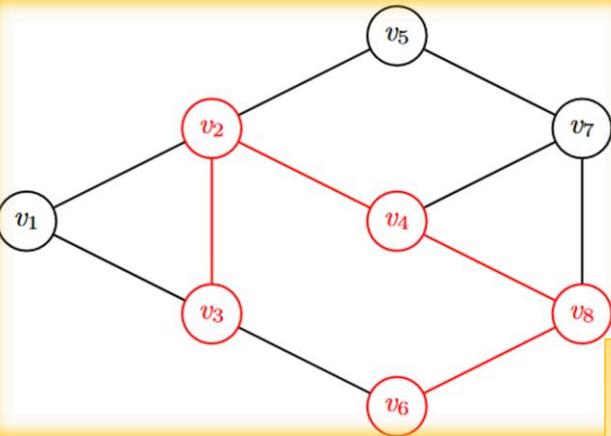
S'il a un voisin non visité, celui-ci est ajouté sur la pile



Le sommet du haut de la pile est le sommet courant.

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

5



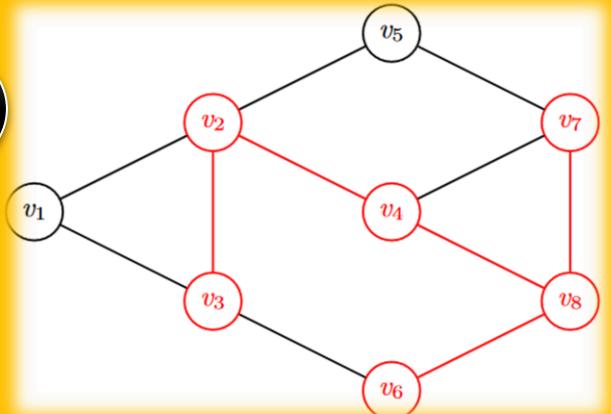
S'il a un voisin non visité, celui-ci est ajouté sur la pile

S'il n'en a pas, le sommet courant est supprimé.

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

Le sommet du haut de la pile est le sommet courant.

6

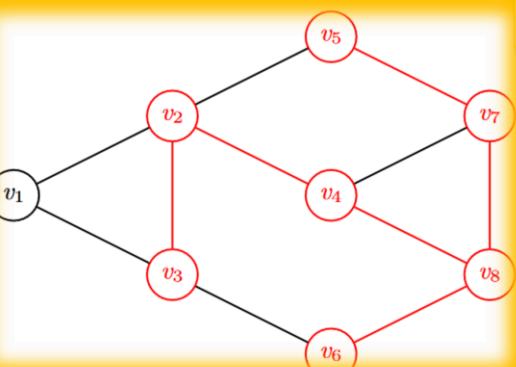


Le sommet du haut de la pile est le sommet courant.

S'il a un voisin non visité, celui-ci est ajouté sur la pile

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

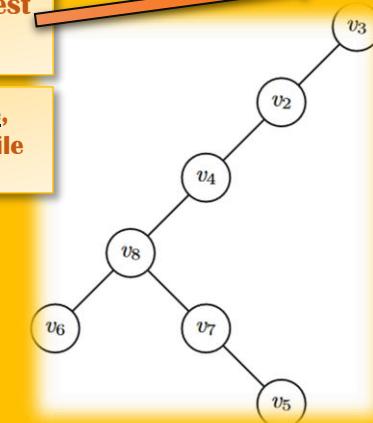
7



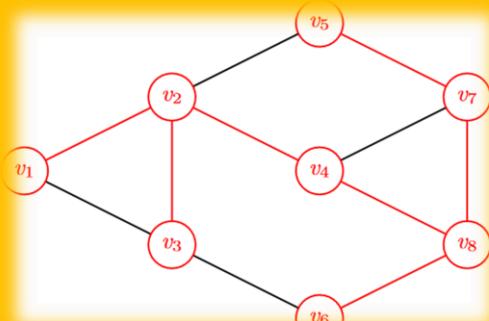
Le sommet du haut de la pile est le sommet courant.

S'il a un voisin non visité, celui-ci est ajouté sur la pile

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



8



$P = \{v_3, v_2, v_1\}$
puis se dépile jusqu'à être vide.



Algorithme DFS

pseudo-code

DFS(G, v)

Un graphe G et l'un de ses sommets v
Parcours en profondeur de G partant de v

$P = \{v\}$ $T = v$; **T** : un arbre

for u sommet de G **do**
 | $u.\text{visité} = \text{FALSE}$
end
 $v.\text{visité} = \text{TRUE}$

P : une pile pour stocker les sommets visités

L'idée est d'aller aussi loin que possible dans le graphe, et de rebrousser chemin quand on ne peut plus avancer.

while $P \neq \emptyset$ **do**

 | $u = \text{dernier élément de } P$

 | **if** il existe w voisin de u avec $w.\text{visité} = \text{FALSE}$ **then**

 | $w.\text{visité} = \text{TRUE}$

 | Ajouter w à P

S'il a un voisin non visité, celui-ci est ajouté sur la pile

 | Ajouter w et (u, w) à T

 | **end**

 | **else**

 | Supprimer u de P ;

S'il n'en a pas, le sommet courant est supprimé.

 | **end**

end

Result : Un arbre T enraciné en v

 **Chaque sommet est étiqueté deux fois**
une fois "visité FALSE" , une fois "visité TRUE"

Propriété du parcours en profondeur

Les arêtes sélectionnées lors du parcours DFS(G,s) forme un arbre couvrant pour la composante connexe de s

- ▶ 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.

Cours p. 41

Forêt couvrante

Une forêt couvrante
être un sous - graphe acyclique maximale du graphe donné,
ou un graphique composé équivalente d'

**un arbre couvrant dans chaque
composante connexe du graphe.**

→ Proposer à partir du **parcours en profondeur**
 un procédé permettant de déterminer une **forêt couvrante**
 pour tout graphe G .

On choisit n'importe quel sommet et on fait un parcours en profondeur.
 En revenant au point de départ, on choisit un autre sommet qui n'est pas encore « visité »
 et on recommence jusqu'au moment où il ne reste plus de sommets libres.

Forêt couvrante(G)

$$F = \emptyset$$

$$\forall v, v.\text{visité} = \text{FALSE}$$

Initialisation

Tant que $V(F) \neq V(G)$

choisir un sommet non visité v

$$T = \text{DFS}(G, v)$$

marquer tous les sommets de T comme visités

$$F = F \cup T$$

Un graphe G est un couple (V, E)
 où V (ou $V(G)$ en cas de confusion
 possible) est l'ensemble des sommets

Le résultat est une **forêt** : si une arête relie deux arbres T_1 et T_2 ,

$T_1 \cup T_2$ est connexe et le DFS l'aurait parcouru en une seule fois.

La forêt est couvrante puisque c'est le critère d'arrêt.

→ Dans le cas non-orienté, que peut-on dire du nombre d'arbres composant la forêt couvrante ?

→ En déduire que la taille de la forêt ne dépend pas du sommet de départ choisi.

Dans le cas non-orienté,

La forêt couvrante contient un arbre couvrant par composante connexe.

Chaque DFS (parcours en profondeur) découvre exactement la composante connexe du sommet de départ choisi : la taille de la forêt (le nombre d'arbres que contient la forêt) est donc toujours égale au nombre de composantes connexes de G , indépendamment des points de départ.

Cette forêt est de plus minimale en nombre d'arbres, puis qu'un graphe à k composante connexe (C.C.) ne peut pas être couvert par un nombre d'arbres $< k$

→ Les constatations précédentes sont-elles encore vraies dans le cas orienté ?

Dans le cas orienté,

Ce n'est plus le cas.

Si on choisit v_1 comme sommet de départ, on obtient $v_1 v_2 \dots v_n$

→ On aura une forêt couvrante à un arbre.

Si on choisit v_n comme sommet de départ, on obtient v_n

Puis si on choisit v_{n-1} comme sommet de départ,
etc...

→ On aura une forêt couvrante à n arbres.

On peut obtenir deux forêts de taille différentes pour le même graphe. Ces tailles ne peuvent donc pas correspondre à un nombre de composantes (fortement) connexes.

Connexité

Un graphe est **connexe** si entre deux sommets il existe toujours un chemin.

Pour un graphe orienté, il existe plusieurs notions de connexité. Un tel graphe est :

- ▶ **faiblement connexe** si le graphe obtenu en oubliant l'orientation est connexe ;
- ▶ **fortement connexe** si, d'un sommet à un autre, il existe toujours un chemin orienté.

Un graphe non connexe peut être décomposé en composantes connexes (ou composantes faiblement connexes pour les graphes orientés).

Soit G un graphe orienté et v un sommet de G .

On appelle *descendant* de v

tout sommet w tel qu'il existe un chemin orienté de v à w .

On appelle *ascendant* de v

tout sommet u tel qu'il existe un chemin orienté de u à v .

Question 1

Exercice 2.2

Exercice de cours

On considère le graphe de la figure 2.1.

Déterminer l'ensemble des *descendants* et des *ascendants* du sommet 4.

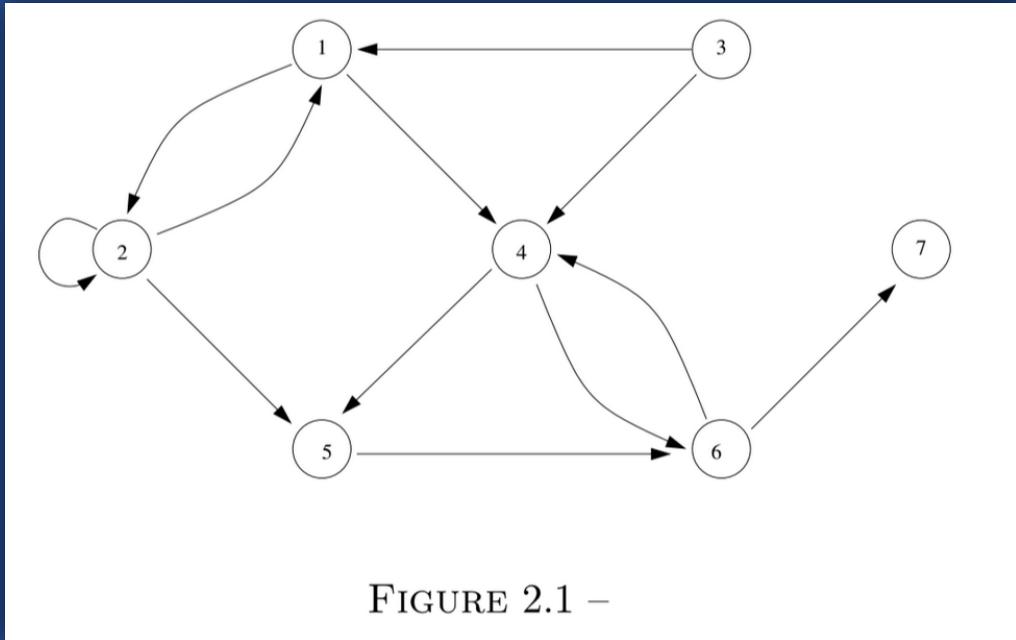


FIGURE 2.1 –

On appelle **descendant** de v

tout sommet w tel qu'il existe un chemin orienté de v à w .

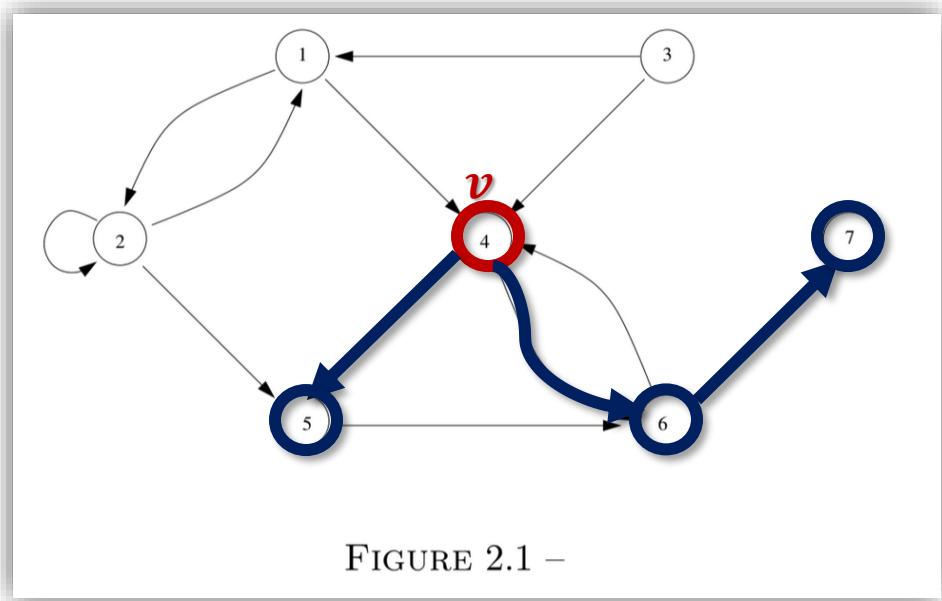


FIGURE 2.1 –

Descendants : 5, 6, 7

On appelle **ascendant** de v

tout sommet u tel qu'il existe un chemin orienté de u à v .

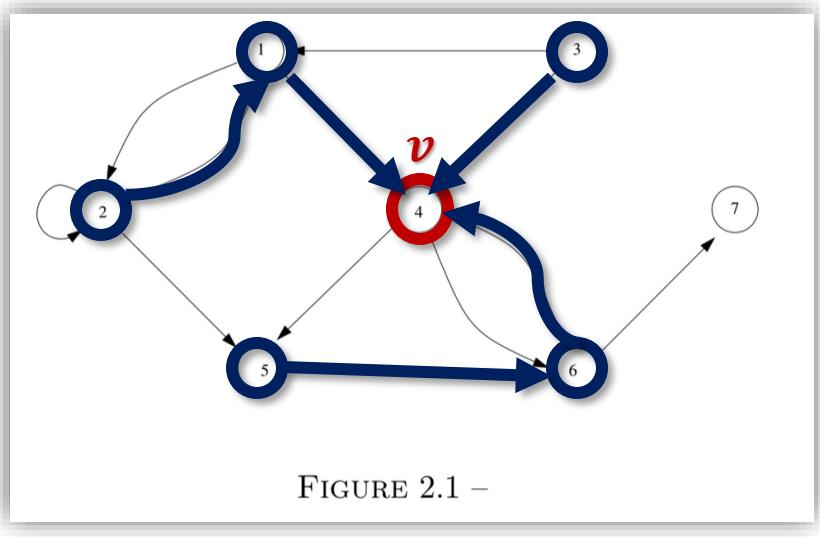


FIGURE 2.1 –

Ascendants : 1, 2, 3, 5, 6

Soit G un graphe orienté et v un sommet de G .

On appelle *descendant* de v

tout sommet w tel qu'il existe un chemin orienté de v à w .

On appelle *ascendant* de v

tout sommet u tel qu'il existe un chemin orienté de u à v .

Question 2

Proposer deux façons de déterminer les *ancêtres* et les *descendants* d'un sommet.

Quelle est la complexité de ces algorithmes ?

Exercice 2.2

Exercice de cours

On appelle *ascendant* (ou *ancêtre*) de v
tout sommet u tel qu'il existe un chemin orienté de u à v .

BFS	Breadth First Search	Parcours en largeur	$O(m)$
DFS	Depth First Search	Parcours en profondeur	$O(m)$

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

Pour les descendants,

On fait un parcours en profondeur orienté

ou un parcours en largeur orienté partant du sommet 4
et en s'arrêtant quand on revient à la racine.

Les sommets non-inclus dans cet arbre ne sont pas des descendants du sommet-racine.

Si on considère qu'un sommet peut être son propre descendant, on l'ajoute si le graphe a des cycles le contenant.

→ **Complexité** en $O(m)$ avec m le nombre d'arêtes.

Pour les descendants,

On inverse le sens de toutes les arêtes et on procède comme pour les descendants. (Plutôt que d'implémenter un nouvel algorithme pour les descendants).

→ **Complexité** en $O(m)$ avec m le nombre d'arêtes.

(La complexité est la même car pour faire le renversement des arêtes, on regarde 1 fois chaque arête et on la renverse en cout constant)

Soit G un graphe non-orienté connexe.

Un sommet v est dit **séparateur**

si $G - v$ (le graphe obtenu en supprimant v),
n'est plus connexe.

Connexité, graphe connexe

(Définition) : Un graphe est
connexe si, pour toute paire
de sommets u et v de G , il
existe un chemin entre u et v

Question 1

Exercice 2.3

Le graphe de la figure 2.2,
comporte-t-il un ou des sommet(s) **séparateur(s)** ,
et si oui le(s)quel(s) ?

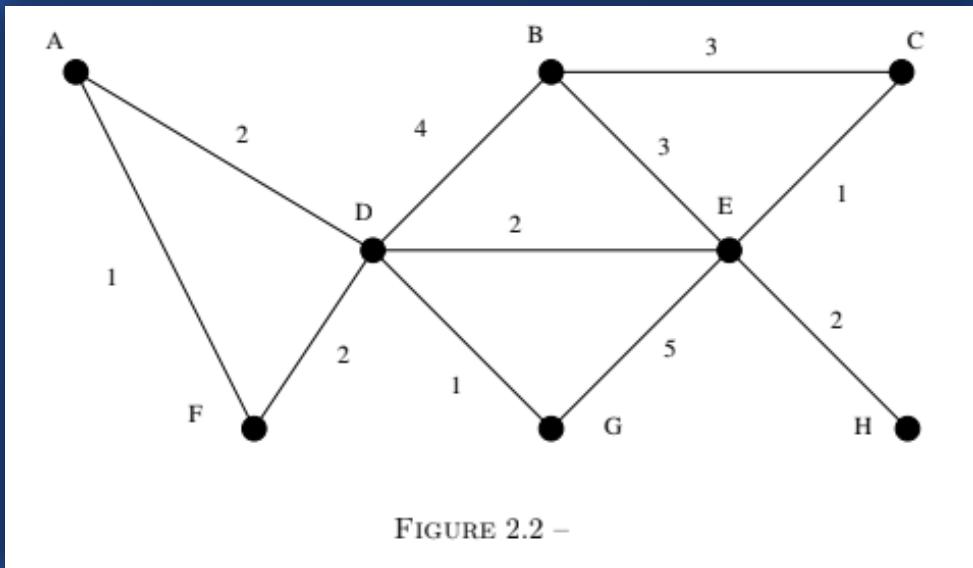
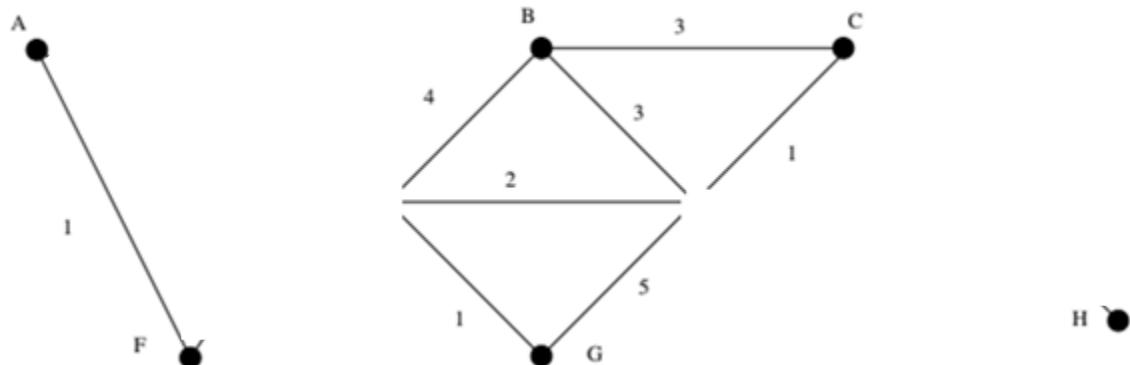


FIGURE 2.2 –

Les sommets séparateurs sont D et E :



Soit G un graphe non-orienté connexe.

Un sommet v est dit séparateur

si $G - v$ (le graphe obtenu en supprimant v),
n'est plus connexe.

Question 2

Exercice 2.3

Dessiner un parcours en profondeur sur ce graphe,
de racine D,

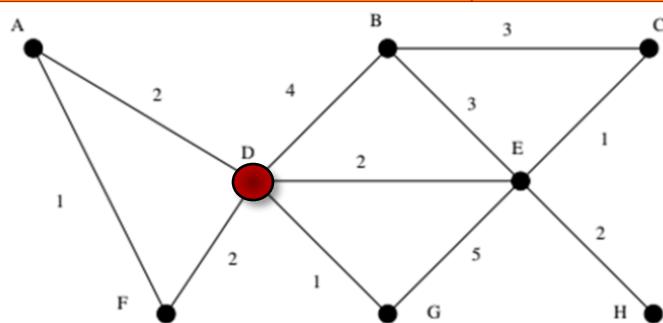
en le représentant comme un arbre enraciné
(la racine en haut et on descend d'un niveau à chaque parcours d'un nouveau sommet).

Que peut-on dire du degré de D dans cet arbre
et du nombre de composantes connexes dans le graphe privé de D ?

DFS

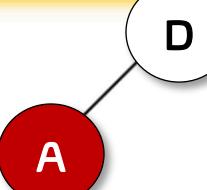
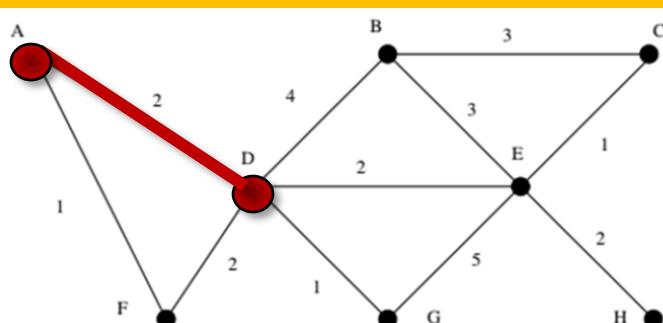
Depth First Search

Parcours en profondeur

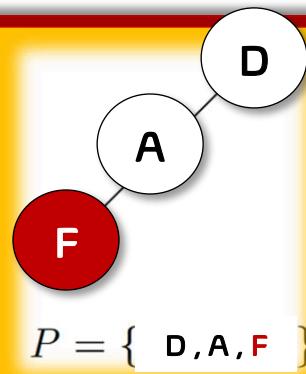
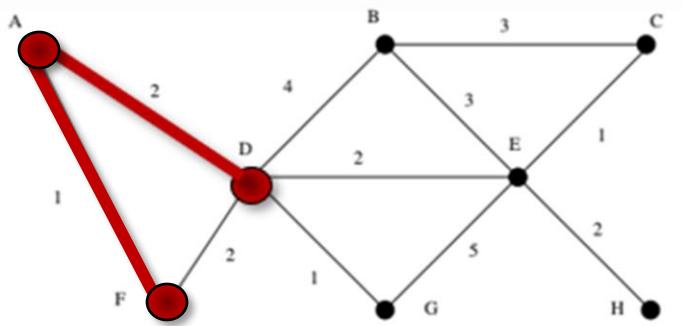


D

$$P = \{ D \}$$



$$P = \{ D, A \}$$



Le sommet du haut de la pile est le sommet courant

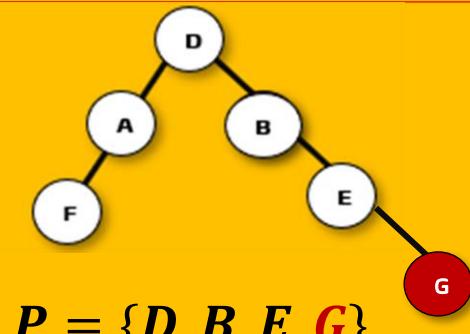
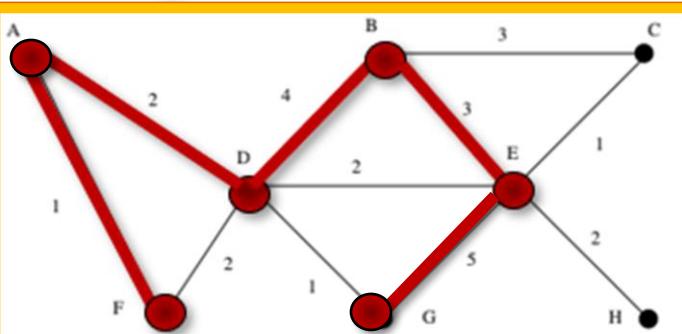
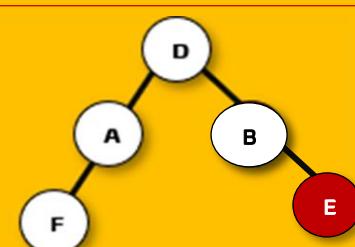
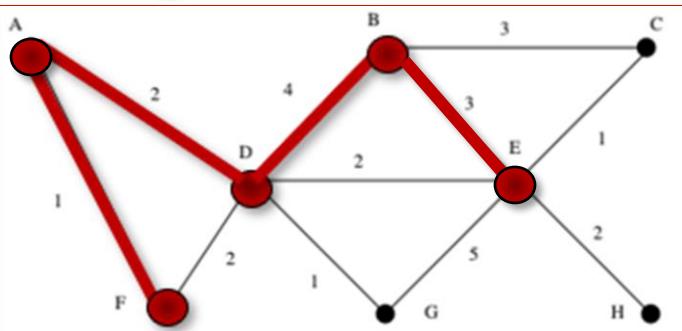
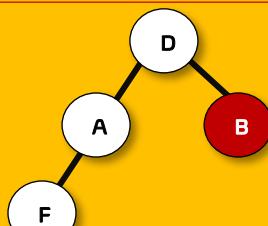
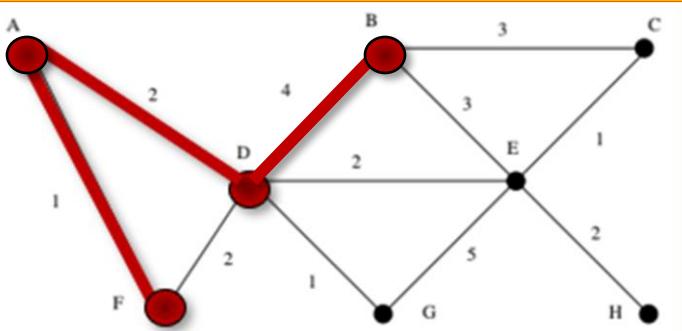
S'il a un voisin non visité, celui-ci est ajouté sur la pile. S'il n'en a pas, le sommet courant est supprimé.

Donc : $P = \{D, A\}$

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.

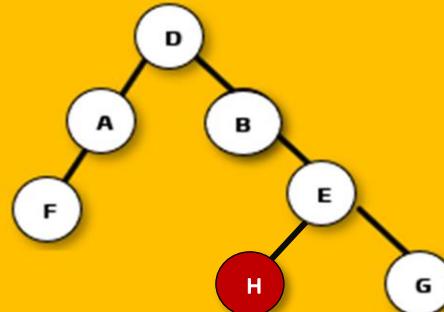
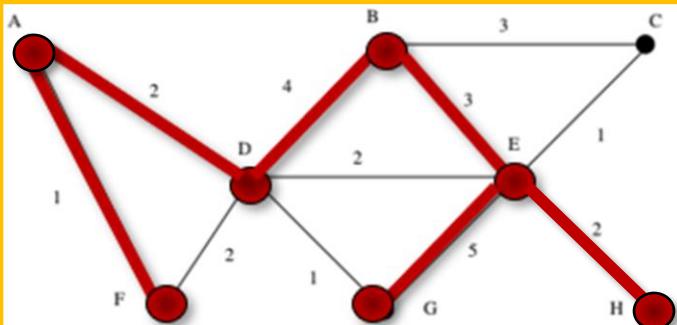
Donc : $P = \{D\}$



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.

Donc: $P = \{D, B, E\}$

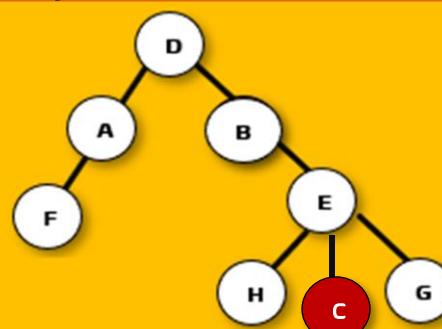
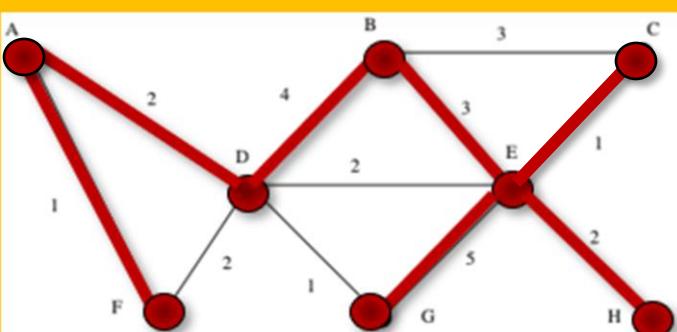


$P = \{D, B, E, H\}$

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.

Donc: $P = \{D, B, E\}$



$P = \{D, B, E, C\}$

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.

Donc: $P = \{D, B, E\}$

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.

Donc: $P = \{D, B\}$

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.

Donc: $P = \{D\}$

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.

Donc: $P = \emptyset$

se dépile jusqu'à être vide

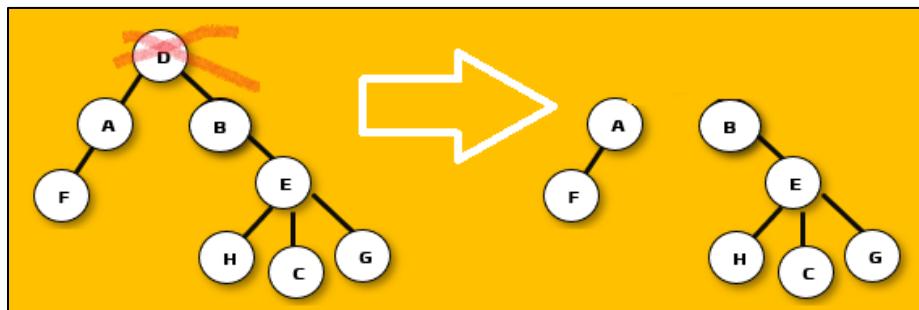
Que peut-on dire du degré de D dans cet arbre et du nombre de composantes connexes dans le graphe privé de D ?

Definition

Le **degré** d'un sommet v , noté $d(v)$, désigne son nombre de voisins.

D est de degré 2 dans cet arbre.

Supprimer **D** dans **G** crée 2 composantes connexes.



Dessiner un parcours en profondeur sur ce graphe ,

de racine D,

en le représentant comme un arbre enraciné

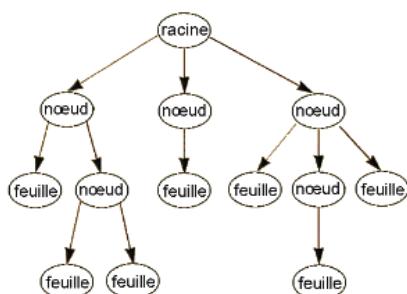
(la racine en haut et on descend d'un niveau à chaque parcours d'un nouveau sommet).

Un **arbre enraciné**, ou **arborescence**, est une structure de données constituée de sommets assemblés par niveaux dans lequel l'un des sommets se distingue des autres : on appelle ce sommet la **racine**.

on appellera souvent **noeuds** les sommets des arbres (enracinés).

Feuille, Noeud interne

Un noeud sans fils est appelé **noeud externe**, **noeud terminal** ou plus simplement **une feuille**. Un noeud qui n'est pas une feuille est un **noeud interne**.



Soit G un graphe non-orienté connexe.

Un sommet v est dit **séparateur**

si $G - v$ (le graphe obtenu en supprimant v),
n'est plus connexe.

Question 3

Exercice 2.3

On considère un graphe non-orienté quelconque G

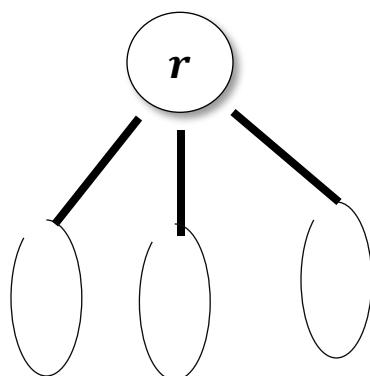
et un parcours en profondeur T de ce graphe, de racine r .

Justifier que les branches issues de chacun des fils de r dans T
forment des composantes connexes distinctes dans $G - r$.

En déduire une caractérisation du fait que r est un séparateur
en fonction de son degré dans T .

le graphe obtenu en supprimant r

Justifier que les branches issues de chacun des fils de r dans T
forment des composantes connexes distinctes dans $G - r$.



Soit S l'ensemble des sommets appartenant à la même branche $T - r$.

→ Il existe un chemin dans T entre toute paire de sommets de S .

Un chemin dans T étant aussi un chemin dans G , on en déduit que tous les sommets de S sont dans la même composante connexe de $G - r$.

Les sommets situés sur une même branche émanant de r sont dans la même composante connexe de $G - r$.

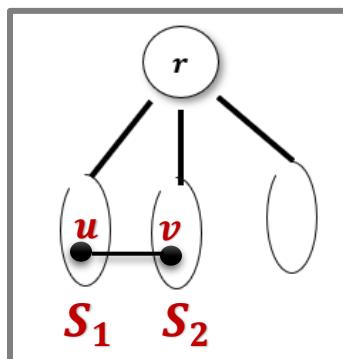
Raisonnement par l'absurde (version 1)

Justifier que les branches issues de chacun des fils de r dans T forment des composantes connexes distinctes dans $G - r$.

Supposons qu'il existe au moins deux ensembles de sommets S_1 et S_2 qui appartiennent à la même composante connexe de $G - r$.

Alors il existe un chemin entre S_1 et S_2 qui évite r (définition de connexité)

Ceci n'est possible que s'il existe une arête entre $u \in S_1$ et $v \in S_2$,



c'est-à-dire u et v non descendants l'un de l'autre.

Or, la structure d'arbre créée par le parcours en profondeur (arbre enraciné) interdit de telles arêtes.

Le raisonnement par l'absurde ou apagogie est une forme de raisonnement logique, philosophique, scientifique consistant soit à démontrer la vérité d'une proposition en prouvant l'absurdité de la proposition complémentaire (ou « contraire »), soit à montrer la fausseté d'une proposition en déduisant logiquement d'elle des conséquences absurdes.

Soit S l'ensemble des sommets appartenant à la même branche $T - r$

Connexité, graphe connexe (Définition) : Un graphe est connexe si, pour toute paire de sommets u et v de G , il existe un chemin entre u et v

Soit G un arbre enraciné de racine r et x un sommet de G .

Un sommet y appartenant au chemin unique allant de r à x est appelé ancêtre de x ,

et x est alors appelé descendant de y .

arbre enraciné : tous les nœuds sauf la racine ont un unique parent

On aboutit à une contradiction. Les ensembles S sont donc dans des composantes connexes distinctes de $G - r$. Chacun correspond donc à une C.C. (composantes connexes) distinctes de $G - r$. Chacun correspond donc à une C.C. (composantes connexes) de $G - r$.

Raisonnement par l'absurde vu en TD (grp 4)

Justifier que les branches issues de chacun des fils de r dans T forment des composantes connexes distinctes dans $G - r$.

Les sommets situés sur une même branche émanant de r sont dans la même composante connexe de $G - r$.

En effet, les arêtes de T sont des arêtes de G .

→ Tous ses sommets peuvent donc être reliés par des chemins.

Supposons qu'il existe une arête entre deux branches

Alors il existerait une arête présente dans G , et non dans T , qui ne relie pas deux sommets dont l'un est le descendant de l'autre dans T .

Ceci est impossible (cours) dans un DFS.

Chaque branche émanant (provenant) de r correspond donc à une composante connexe de $G - r$.

En déduire une caractérisation du fait que r est un séparateur en fonction de son degré dans T .

Definition

Le degré d'un sommet v , noté $d(v)$, désigne son nombre de voisins.

r est un séparateur ssi (si et seulement si)

il est de degré supérieur ou égal à 2 dans DFS enraciné en r

Soit G un graphe non-orienté connexe.

Un sommet v est dit **séparateur**

si $G - v$ (le graphe obtenu en supprimant v),
n'est plus connexe.

Question 4

Exercice 2.3

En déduire un algorithme permettant de dresser la liste des séparateurs d'un graphe non-orienté connexe.

Quelle est sa complexité ?

Remarque : Cet algorithme n'est pas optimal, il est possible d'énumérer les séparateurs en temps linéaire en le nombre d'arêtes.

Séparateurs = \emptyset

Pour tout $v \in V(G)$

$$T = DFS(G, v)$$

Si degré de v dans $T \geq 2$

$$\text{Séparateurs} = \text{Séparateurs} \cup \{v\}$$

Un graphe G est un couple (V, E)
où V (ou $V(G)$ en cas de confusion possible) est l'ensemble des sommets

Parcours en profondeur enraciné en v

v est un séparateur si et seulement si il est de degré supérieur ou égal à 2 dans DFS enraciné en v

DFS

Depth First Search

Parcours en profondeur

$O(m)$

On lance autant de DFS qu'il y a de sommets donc l'algorithme

est de complexité $O(n \cdot m)$

On note $n(G) = |V|$ et $m(G) = |E|$,

ou n et m quand il n'y a pas d'ambigüité.

Cette partie de l'algo est en temps constant.

On considère des graphes non-orienté, non-pondérés et connexes.

Graphes pondérés

Dans un graphe pondéré les arêtes (ou les arcs) sont, ben, **pondérés**, quoi. Autrement dit, on associe une valeur à chaque arête. Elles peuvent très bien être négatives.

Cela peut-être une distance : lorsque que l'on cherche un plus court chemin entre deux nœuds, il va de soit que la somme des pondérations des arêtes doit être aussi petit que possible.

Question 1

Soit v un sommet

Exercice 2.4

et T l'arbre obtenu en appliquant un parcours en largeur depuis v .

Montrer que pour tout sommet w ,

le chemin de T reliant v à w

est un plus court chemin entre v et w dans G .

En déduire un algorithme permettant de déterminer

le sommet le plus éloigné d'un sommet v .

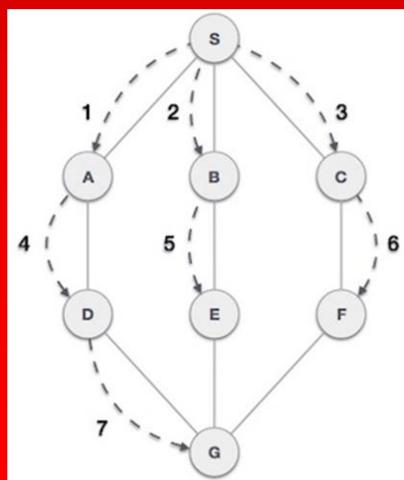
BFS

Breadth First Search

Parcours en
largeur

Soient G un graphe non-orienté et v un sommet de G .

On commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc.



Suite

Objectif - "découvrir" tous les sommets du graphe accessibles depuis un sommet origine v

« En largeur » - Intuitivement : tous le sommets à un certaine distance d sont rencontrés avant le premier sommet à distance supérieure à d

Idée - parcourir les sommets adjacents à v et les stocker dans une file.

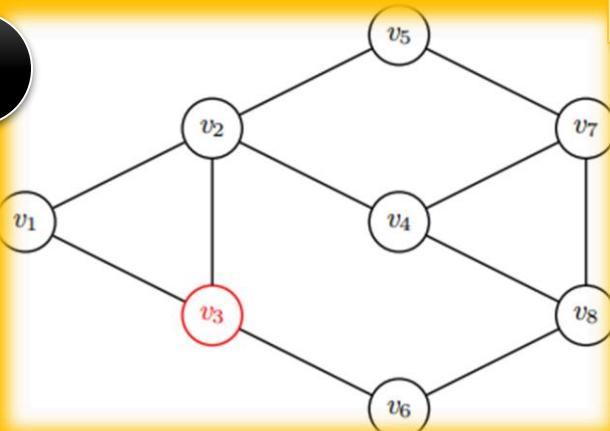
Pour chaque sommet défilé, parcourir ses adjacents et les enfiler.

Pour parcourir chaque sommet une seule fois : marquage des sommets rencontrés

L'algorithme de parcours en largeur permet de calculer les distances de tous les nœuds depuis un nœud source dans un graphe non pondéré (orienté ou non orienté). Il peut aussi servir à déterminer si un graphe non orienté est connexe.

Utiliser une file, ou FIFO (First In, First Out), pour stocker les sommets visités

1

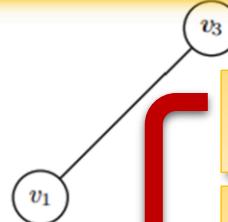
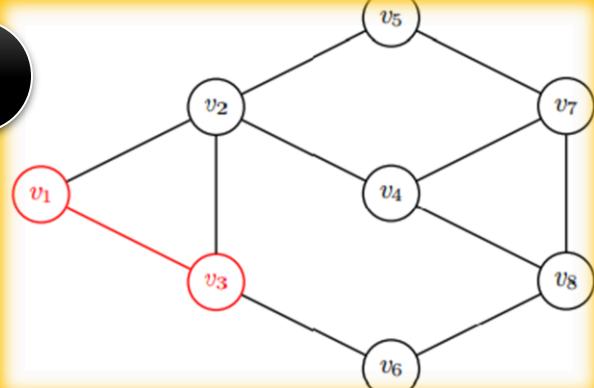


F : une file pour stocker les sommets visités

$$F = \{v_3\}$$

Le premier sommet de la file est le sommet courant

2

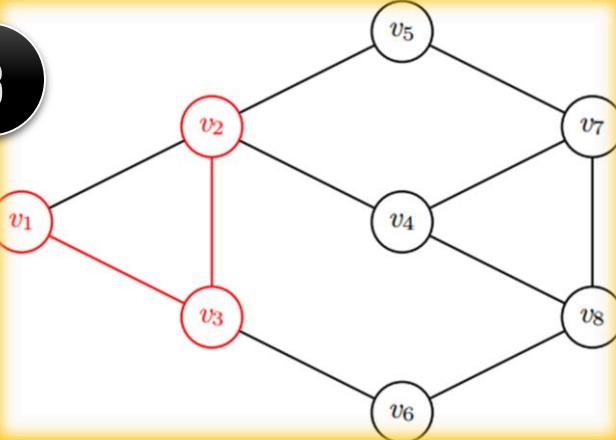


Le premier sommet de la file est le sommet courant

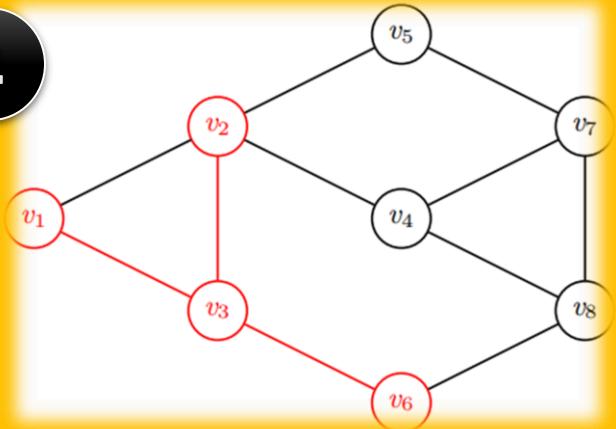
Ses voisins sont ajoutés un à un en bout de file

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

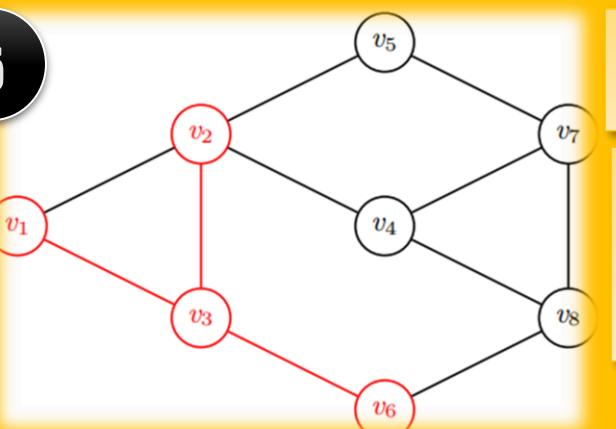
3



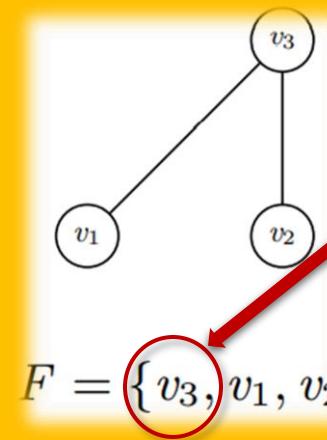
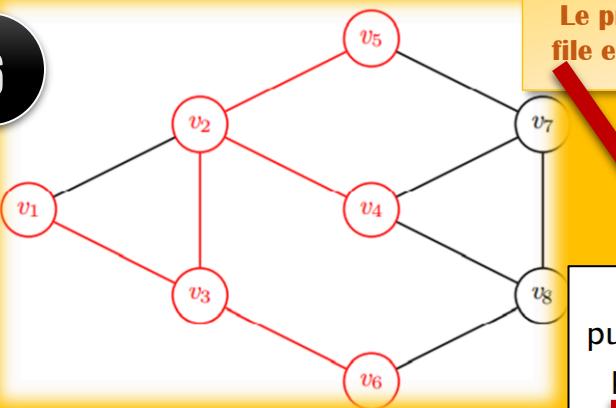
4



5



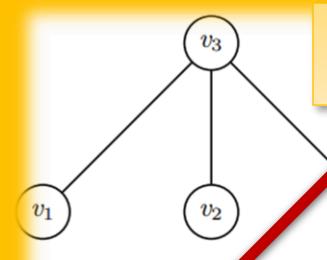
6



Le premier sommet de la file est le sommet courant

Ses voisins sont ajoutés un à un en bout de file

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



Le premier sommet de la file est le sommet courant

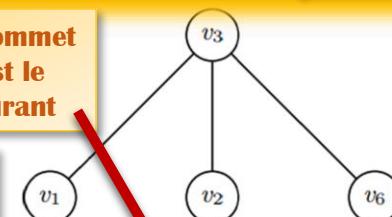
Ses voisins sont ajoutés un à un en bout de file

$$\begin{aligned} F &= \{v_3, v_1, v_2, v_6\} \\ \text{puis } F &= \{v_1, v_2, v_6\} \end{aligned}$$

Quand tous ses voisins sont visités, le sommet est supprimé de la file

Le premier sommet de la file est le sommet courant

Tous ses voisins sont déjà visités, le sommet est supprimé



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

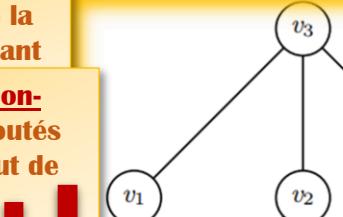
$$\text{puis } F = \{v_2, v_6\}.$$

Aucun sommet n'est ajouté à la file

Le premier sommet de la file est le sommet courant

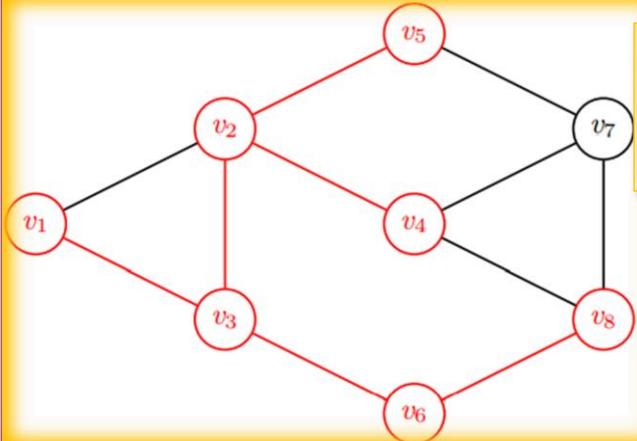
Ses voisins non-visités sont ajoutés un à un en bout de file

$$\begin{aligned} F &= \{v_2, v_6, v_4\} \\ \text{puis } F &= \{v_2, v_6, v_4, v_5\} \\ \text{puis } F &= \{v_6, v_4, v_5\} \end{aligned}$$



Quand tous ses voisins sont visités, le sommet est supprimé de la file

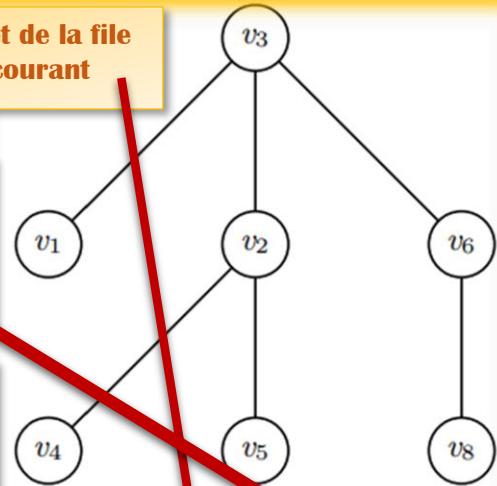
1



Le premier sommet de la file est le sommet courant

Ses voisins non-visités sont ajoutés un à un en bout de file

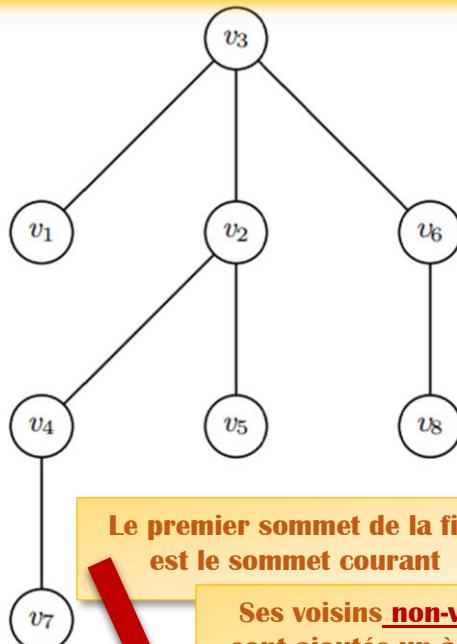
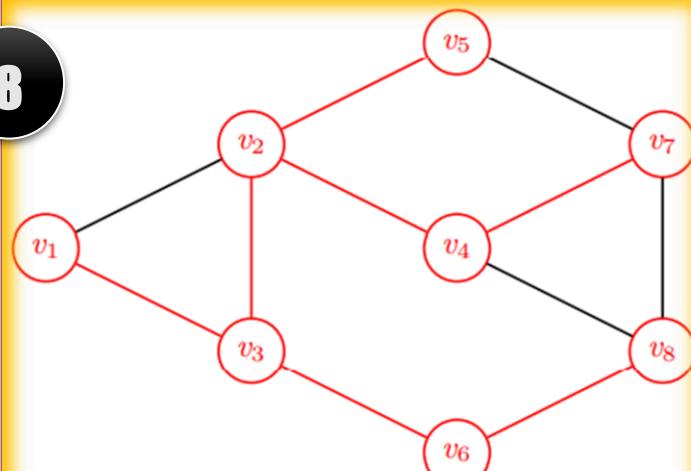
Quand tous ses voisins sont visités, le sommet est supprimé de la file



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

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

8



Quand tous ses voisins sont visités, le sommet est supprimé de la file

Le premier sommet de la file est le sommet courant

Ses voisins non-visités sont ajoutés un à un en bout de file

$$F = \{v_4, v_5, v_8, v_7\}$$

puis F se vide et l'algorithme s'arrête.

Algorithme BFS

pseudo-code

BFS(G, v)

Un graphe G et l'un de ses sommets v
Parcours en largeur de G partant de v

$F = \{v\}; T = v$

T : un arbre

for u sommet de G **do**

F : une file pour stocker les sommets visités.

+ $u.\text{visité} = \text{FALSE}$

end

while $F \neq \emptyset$ **do**

u = premier élément de F

Le premier sommet de la file est le sommet courant

for w voisin de u **do**

Ses voisins sont ajoutés un à un en bout de file.

if $w.\text{visité} = \text{FALSE}$ **then**

+ $w.\text{visité} = \text{TRUE}$

Ajouter w à F

Ajouter w et (u, w) à T

end

end

Supprimer u de F

Quand tous ses voisins sont visités, le sommet est supprimé de la file et ne la réintègrera plus.

end

Résultat : Un arbre T enraciné en v

Chaque sommet est étiqueté deux fois
une fois "visité FALSE" , une fois "visité TRUE"

Soit v un sommet

et T l'arbre obtenu en appliquant **un parcours en largeur** depuis v .

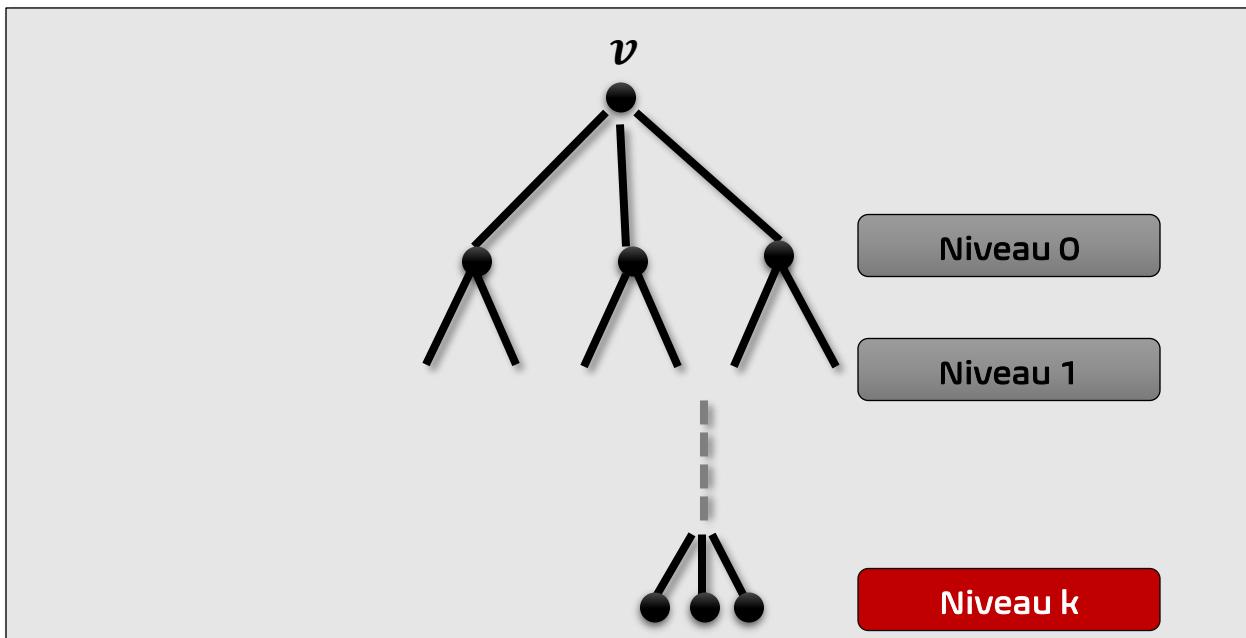
Montrer que pour tout sommet w ,

le chemin de T reliant v à w

est un plus court chemin entre v et w dans G .

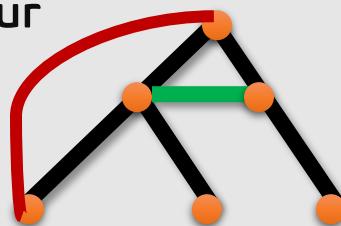
Soit k le niveau auquel apparaît w dans l'arbre en largeur.

Le chemin entre v et w dans l'arbre est de longueur k .



Or, il est impossible qu'une arête de G non présente dans T relie u_1 et u_2 dont les niveaux ne sont pas égaux ou successifs. (Une arête ne peut pas sauter une génération).

Parcours en largeur
IMPOSSIBLE
POSSIBLE



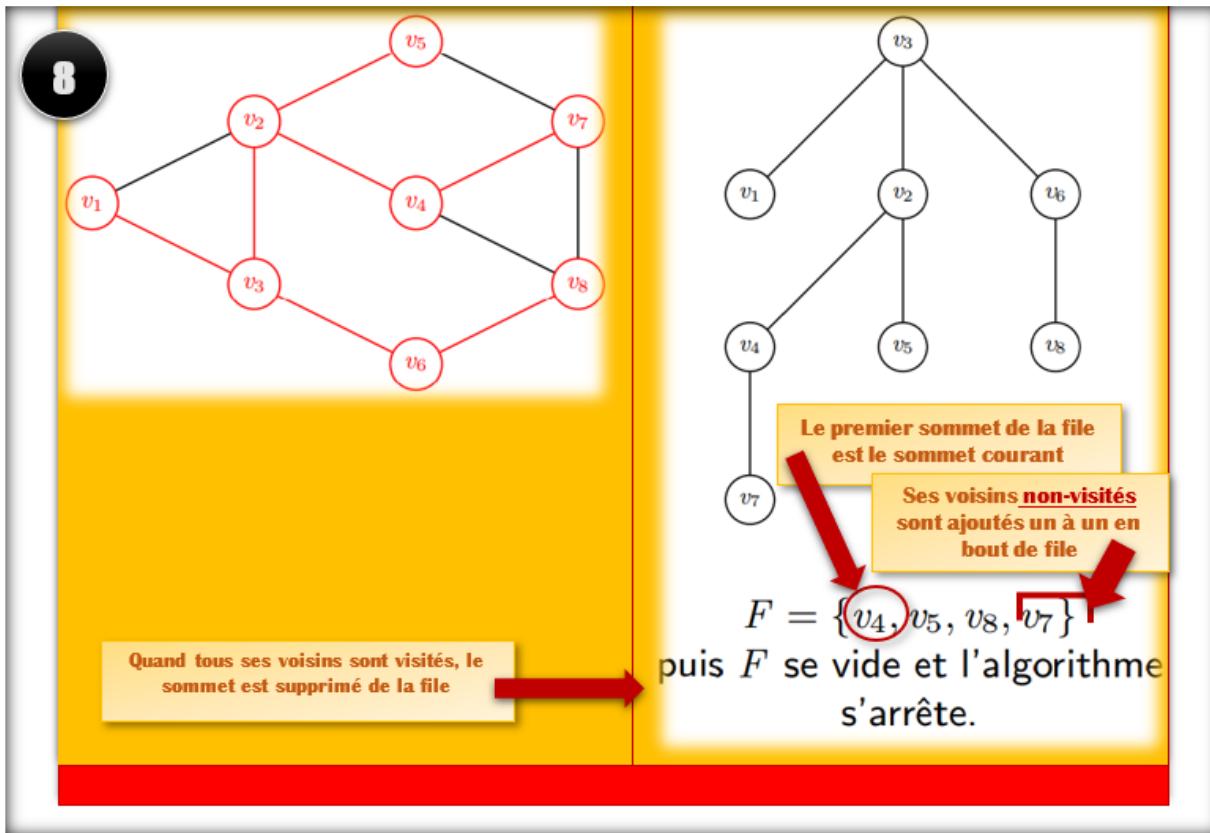
S'il existe un chemin plus court,
il y aurait une arête qui « saute »
un niveau dans $G \setminus T$, ce qui est
impossible dans un BFS.

Il est donc **impossible** d'aller de v à w en moins de k pas.

En déduire un algorithme permettant de déterminer
le sommet le plus éloigné d'un sommet v .

Algorithme : BFS + choisir le dernier sommet parcouru

Dans l'exemple du déroulement de l'algo BFS : le dernier sommet parcouru est u_7



On considère des graphes non-orienté, non-pondérés et connexes.

Graphes pondérés

Dans un graphe pondéré les arêtes (ou les arcs) sont, ben, **pondérés**, quoi. Autrement dit, on associe une valeur à chaque arête. Elles peuvent très bien être négatives. Cela peut-être une distance : lorsque que l'on cherche un plus court chemin entre deux nœuds, il va de soit que la somme des pondérations des arêtes doit être aussi petit que possible.

Question 2

Exercice 2.4

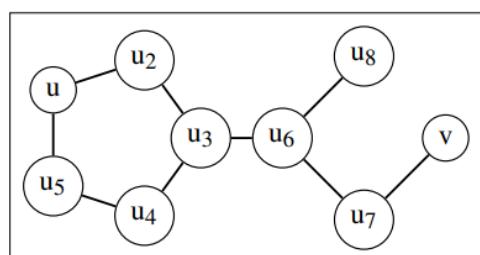
En déduire un algorithme pour calculer le diamètre d'un graphe non pondéré.

Quelle est sa complexité ?

1.IV.2.h. DIAMÈTRE (Définition) :

Le diamètre d'un graphe est la distance maximale qui soit entre toutes paires de sommets d'un graphe.

1.IV.2.h.1 Remarque : Dans le cas non-arête-valué, c'est le plus grand chemin le plus court entre tous les sommets d'un graphe.



diam(G) = 5 car $d(u, v) = 5$ et toutes les autres paires de sommets ont une distance inférieure ou égale à 5.

Lancer l'algorithme précédent en chaque sommet.

BFS

$O(m)$

On lance autant de BFS qu'il y a de sommets donc l'algorithme est de complexité $O(n \cdot m)$

On note $n(G) = |V|$ et $m(G) = |E|$,

ou n et m quand il n'y a pas d'ambigüité.

On considère des graphes non-orienté, non-pondérés et connexes.

Graphes pondérés

Dans un graphe pondéré les arêtes (ou les arcs) sont, ben, **pondérés**, quoi. Autrement dit, on associe une valeur à chaque arête. Elles peuvent très bien être négatives. Cela peut-être une distance : lorsque que l'on cherche un plus court chemin entre deux nœuds, il va de soit que la somme des pondérations des arêtes doit être aussi petit que possible.

Question 3

Pour des très grands graphes,
une complexité non linéaire peut être rédhibitoire (= insupportable).

Exercice 2.4

On considère l'algorithme consistant à

Etape 1

Choisir un sommet v quelconque,
lancer un BFS enraciné en v
choisir x comme le sommet le plus éloigné de v .

Etape 2

Lancer un BFS enraciné en v
choisir y comme le sommet le plus éloigné de x .

Etape 3

Retourner la distance x à y .

- Justifier le fait que cet algorithme est linéaire.
- Construire un contre-exemple montrant qu'il n'est pas exact.
- Montrer qu'il s'agit d'une 2-approximation,
c'est-à-dire que $d(x, y) \leq \text{diam}(G) \leq 2d(x, y)$

On pourra pour cela comparer $d(x, y)$ et $\text{diam}(G)$ à $d(v, x)$

Pour de très grands graphes, une approximation possible du diamètre

peut être obtenue en un temps linéaire comme suit :

Diamètre (Définition)

Le diamètre d'un graphe est la distance maximale qui soit entre toutes paires de sommets d'un graphe.

Etape 1

Choisir un sommet v quelconque,
lancer un BFS enraciné en v
choisir x comme le sommet le plus éloigné de v .

a. On choisit aléatoirement
un sommet v

Etape 2

Lancer un BFS enraciné en v
choisir y comme le sommet le plus éloigné de x .

b. On détermine à l'aide
d'un BFS un sommet x le
plus éloigné possible de v .

Etape 3

Retourner la distance x à y .

c. On détermine à l'aide
d'un BFS un sommet y le
plus éloigné possible de x .

d. On renvoie la distance
 $d(x, y)$

□ Justifier le fait que cet algorithme est linéaire.

Définitions (désignations des complexités courantes)

<i>notation</i>	<i>désignation</i>	<i>notation</i>	<i>désignation</i>
$\Theta(1)$	<i>constante</i>	$\Theta(n^2)$	<i>quadratique</i>
$\Theta(\log n)$	<i>logarithmique</i>	$\Theta(n^3)$	<i>cubique</i>
$\Theta(\sqrt{n})$	<i>racinaire</i>	$\Theta(n^k)$, $k \in \mathbb{N}, k \geq 2$	<i>polynomiale</i>
$\Theta(n)$	<i>linéaire</i>	$\Theta(a^n)$, $a > 1$	<i>exponentielle</i>
$\Theta(n \log n)$	<i>quasi-linéaire</i>	$\Theta(n!)$	<i>factorielle</i>

Retenir (Attention !!!)

La complexité d'un algorithme ne parle pas du **temps de calcul absolu** des implémentations de cet algorithme mais de la **vitesse avec laquelle ce temps de calcul augmente** quand la taille des entrées augmente.

On a effacé les constantes : $O(n) = O(2n)$.

On note $n(G) = |V|$ et $m(G) = |E|$.

ou n et m quand il n'y a pas d'ambigüité.

BFS

$O(m)$

Il s'agit de deux BFS $\rightarrow O(m)$.

Récupérer le niveau de y dans le dernier BFS est également linéaire (on peut même l'intégrer au BFS).

Etape 1

Choisir un sommet v quelconque,

lancer un BFS enraciné en v

choisir x comme le sommet le plus éloigné de v .

Etape 2

Lancer un BFS enraciné en v

choisir y comme le sommet le plus éloigné de x .

Etape 3

Retourner la distance x à y .



Construire un contre-exemple montrant qu'il n'est pas exact.

On dit que $d(x, y)$ est une **2-approximation** du diamètre.

Cette borne est optimale au sens où il peut y avoir des cas où on renvoie la moitié du diamètre réel.

Algorithmes d'approximation

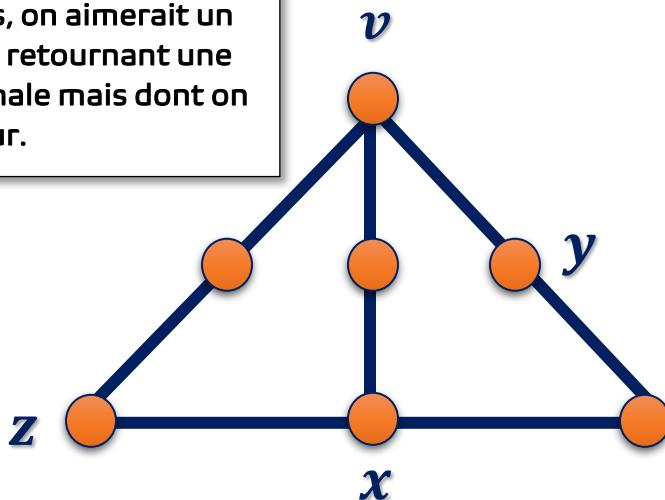
- Lorsqu'un problème d'optimisation est difficile, on sait qu'il est impossible de trouver un algorithme polynomial donnant une solution optimale.
- Il est assez naturel de se demander si relâcher de manière contrôlée la contrainte d'optimalité pourrait permettre d'obtenir une complexité polynomiale.
- En d'autres termes, on aimerait un algorithme rapide, renvoyant une solution non optimale mais dont on peut borner l'erreur.

« On dit que **A** est un algorithme de **r – approximation** »

Le ratio r donne la borne d'erreur de l'algorithme d'approximation par rapport à la solution optimale.

Par exemple

un algorithme de 2-approximation a un ratio 2 et retournera toujours une solution au pire deux fois plus grande (ou plus petite) que la solution optimale.



□ Sommet à distance max de $v \rightarrow x$

□ Sommet à distance max de $x \rightarrow y$

Pourtant, $d(x, y) = 2$ et $d(y, z) = 3$

□ Montrer qu'il s'agit d'une 2-approximation,

c'est-à-dire que $d(x, y) \leq \text{diam}(G) \leq 2d(x, y)$

On pourra pour cela comparer $d(x, y)$ et $\text{diam}(G)$ à $d(v, x)$

$d(\quad) \rightarrow \text{distance}$

$$d(x, y) \leq \text{diam}(G) \leq 2d(x, y)$$

$d(x, y) \leq \text{diam}(G)$

Par définition du diamètre.

1

Diamètre d'un graphe

Plus longue distance entre deux sommets d'un graphe connexe.

$$d(x, y) \leq \text{diam}(G) \leq 2d(x, y)$$

□ Soit (z_1, z_2) les extrémités d'un chemin diamétral.

$$\text{diam}(G) = d(z_1, z_2)$$

□ Par maximalité de x dans le BFS en v , en v ,

$$d(v, z_1) \leq d(v, x) \quad \text{et} \quad d(v, z_2) \leq d(v, x)$$

2 3

□ Par inégalité triangulaire,

4

L'inégalité triangulaire est le fait que, dans un triangle, la longueur d'un côté est inférieure à la somme des longueurs des deux autres côtés.

$$d(z_1, z_2) \leq d(v, z_1) + d(v, z_2)$$

$$d(z_1, z_2) \leq 2d(v, x)$$

□ Par maximalité de y dans le BFS enraciné en x ,

$$d(v, x) \leq d(y, x)$$

Donc $d(x, y) \leq \text{diam}(G) \leq 2d(x, y)$

Question 1

Exercice 2.5

Décrire un algorithme

qui, étant donné un graphe non-orienté G , permet

- Soit de renvoyer un cycle de G ,
- Soit de conclure que G est acyclique.

Un graphe acyclique est un graphe ne contenant aucun cycle.

Dans un graphe non orienté, un cycle est une suite d'arêtes consécutives (chaine simple) dont les deux sommets extrémités sont identiques.

On considère un DFS T

Proposition

Les descendants d'un nœud sont les nœuds de ses sous-arbres

Tout cycle de G

contient une arête entre deux sommets u et v

tels que u est un descendant de v dans T

Démonstration (preuve)

Une arête qui se trouve dans G et pas dans T

Tout cycle contient une arête de G/T puisque T est acyclique.

D'après le cours sur la structure des DFS,
une telle arête relie deux sommets dont l'un est le descendant de l'autre.

2.I.1.a. ARBRE (Définition) :

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

En d'autres termes, il n'a pas de cycles et n'est plus connexe si on lui retire une arête, quelqu'elle soit.

Propriétés des arbres construits par un parcours en profondeur

Propriété

Si G est un graphe connexe non orienté, toute arête de G qui n'est pas dans un arbre couvrant T résultant de son parcours en profondeur à partir de l'un de ses sommets relie deux sommets qui sont descendants l'un de l'autre dans T .

□ Il faut lancer un **parcours en profondeur**

(ou un **parcours en largeur** mais DFS -> cas le plus facile)

et changeant le critère d'arrêt :

- **Dès qu'un sommet a un voisin déjà visité, on s'arrête** en déclarant que l'arête entre ces deux sommets fait partie d'un cycle.
- **S'il n'y a pas de telle arête, le graphe est déclaré acyclique**

□ Pour reconstituer le cycle

Dans le cas d'un parcours en profondeur, cas le plus facile pour reconstituer le cycle,

il suffit de considérer le chemin dans l'arbre partant du voisin visité et allant jusqu'à l'autre extrémité de l'arête détectée comme faisant partie d'un cycle, car tout voisin déjà visité est forcément un ancêtre dans l'arbre.

Un ancêtre d'un nœud est soit son père, soit un ancêtre de son père

Algorithme

- **On fait un DFS.**
- **Si à un moment, un voisin v du sommet u apparaît comme déjà visité**
 - **On renvoie le cycle composé de l'arête vu et du chemin de u à v dans T**
- **Si le DFS se termine sans que ce soit le cas**
 - **Le graphe est acyclique.**

Question 2

Exercice 2.5

Reprendre la question précédente dans le cas d'un graphe orienté.

Déduire à partir de cet algorithme que, si tous les sommets d'un graphe orienté sont de degré sortant au moins 1, ce graphe contient un cycle orienté.

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.
 - L'ensemble des sommets parcourus au sein de l'arbre enraciné en u est l'ensemble des sommets v tel qu'il existe un chemin entre u et v . Si on choisit de parcourir le graphe en entier, le nombre d'arbres nécessaires dépend des choix des racines.
- ▶ 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.
 - si on dessine l'arbre enraciné en mettant les branches parcourues en premier à gauche, toute arête de G qui n'est pas dans l'arbre soit relie des descendants soit va de la droite vers la gauche

Cas des graphes dirigés

VOISINAGE EXTERNE (Définition) :

Dans le graphe G orienté, on appelle le voisinage externe du sommet v , noté $N^+(v)$, l'ensemble des sommets u vérifiant $(v; u) \in E(G)$, c'est-à-dire l'ensemble des voisins cibles d'une arête dont v la source.

VOISINAGE INTERNE (Définition) :

Dans le graphe G orienté, on appelle le voisinage interne du sommet v , noté $N^-(v)$, l'ensemble des sommets u vérifiant $(u; v) \in E(G)$, c'est-à-dire l'ensemble des voisins sources d'une arête dont v la cible.

On appelle **descendant de v**

tout sommet w tel qu'il existe un chemin orienté de v à w .

On appelle **ascendant de v** (ou **ancêtre**)

tout sommet u tel qu'il existe un chemin orienté de u à v .

On considère à nouveau $\text{DFS } T$

Proposition

Tout cycle de G

contient une arête \overrightarrow{uv} (entre deux sommets u et v)

tels que u est un descendant de v dans T

Démonstration (preuve)

On suppose que T est construit de haut en bas, et de gauche à droite.

D'après le cours, il ne peut pas y avoir d'arête dans G/T

qui va d'une branche à l'autre, de gauche à droite.

DFS 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.

il n'y a que des arêtes dans G/T qui descendent, qui remontent ou qui vont de droite à gauche.

Un cycle sans arête qui remonte est donc impossible puisqu'on ne pourrait aller que vers le bas (via T ou G/T) et vers la gauche.

Tout cycle contient donc une arête qui remonte de G/T

G/T

Une arête qui se trouve dans G et pas dans T

On peut reprendre l'algorithme du cas non-orienté, à part que si **v** est déjà marqué comme visité, il faut vérifier s'il appartient aux ancêtres de **u** avant de retourner un cycle.

Déduire à partir de cet algorithme que,
si tous les sommets d'un graphe orienté
sont de degré sortant au moins 1,
ce graphe contient un cycle orienté.

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** $d^-(v)$.

1.III.2.c Cas des graphes dirigés

1.III.2.d. VOISINAGE EXTERNE (Définition) :

Dans le graphe G orienté, on appelle le voisinage externe du sommet v , noté $N^+(v)$, l'ensemble des sommets u vérifiant $(v; u) \in E(G)$, c'est-à-dire l'ensemble des voisins cibles d'une arête dont v la source.

1.III.2.e. VOISINAGE INTERNE (Définition) :

Dans le graphe G orienté, on appelle le voisinage externe du sommet v , noté $N^-(v)$, l'ensemble des sommets u vérifiant $(u; v) \in E(G)$, c'est-à-dire l'ensemble des voisins sources d'une arête dont v la cible.

1.III.2.f. DEGRÉ SORTANT (Définition) :

On appelle degré sortant de v le nombre de ses voisins externes : $d^+(v) = |N^+(v)|$.

1.III.2.g. DEGRÉ ENTRANT (Définition) :

On appelle degré sortant de v le nombre de ses voisins internes : $d^-(v) = |N^-(v)|$

Si tous les sommets ont un degré extérieur ≥ 1 ,

On considère le sommet en bas à gauche de T . L'arête qui en sort ne peut pas descendre ni aller vers la droite. Elle remonte forcément créant donc un cycle.

Décrire un algorithme

Exercice 2.6

qui, étant donné un sommet v dans un graphe orienté, détermine un plus court cycle passant par v .

Indication : Lancer deux parcours en largeur.

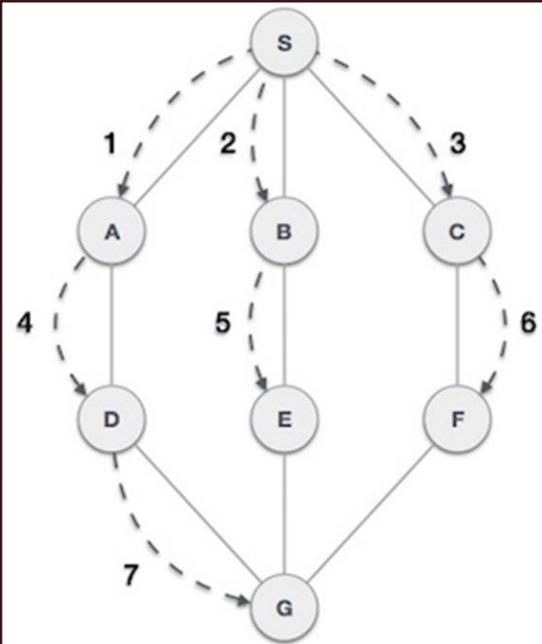
BFS

Breadth First Search

Parcours en largeur

Soient G un graphe non-orienté et v un sommet de G .

On commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc



Soit v un sommet

et T l'arbre obtenu en appliquant un parcours en largeur depuis v .

pour tout sommet w ,

le chemin de T reliant v à w

est un plus court chemin entre v et w dans G .

Exercice 2.4

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.

VOISINAGE EXTERNE (Définition) :

Dans le graphe G orienté, on appelle le voisinage externe du sommet v , noté $N^+(v)$, l'ensemble des sommets u vérifiant $(v; u) \in E(G)$, c'est-à-dire l'ensemble des voisins cibles d'une arête dont v la source.

VOISINAGE INTERNE (Définition) :

Dans le graphe G orienté, on appelle le voisinage interne du sommet v , noté $N^-(v)$, l'ensemble des sommets u vérifiant $(u; v) \in E(G)$, c'est-à-dire l'ensemble des voisins sources d'une arête dont v la cible.

- ▶ Les sommets visités sont ceux qui peuvent être atteints depuis la racine par un chemin orienté.
- ▶ Le chemin du BFS de la racine vers tout sommet visité est un plus court chemin orienté.

Parcours en largeur - cas d'un graphe orienté

Deux choix se présentent

- Soit on oublie l'orientation et on applique le cas non-orienté.
- Soit on autorise le parcours des arêtes uniquement dans le sens de leur orientation et on recommence avec les sommets pour lesquels il n'existe pas de chemin orienté partant du sommet de départ

Dans le premier cas, on obtient un arbre couvrant, mais dans lequel les flèches peuvent être parcourues dans les deux sens

tandis que dans le deuxième cas, on peut obtenir une forêt d'arbres couvrants et des résultats différents selon le sommet de départ

BFS et distance

r est le sommet-racine de G et son niveau est 0

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

- ▶ G ne contient aucune arête orientée d'un sommet u vers un sommet de niveau $\geq niv(u) + 2$.

Algorithme

Pour le faire avec deux parcours BFS,

On lance un BFS sur le graphe ainsi qu'un BFS sur le graphe ayant les orientations des arêtes orientées, et le sommet de plus petit niveau commun aux deux parcours permet de déterminer un graphe

- On lance un BFS orienté T sur G , enraciné en v .
 - Et pour tout u on stocke son niveau $m_1(u)$.
- On crée G' obtenu en changeant l'orientation des arêtes de G .
 - On lance un BFS T' enraciné en v sur G'
 - Et pour tout u on stocke son niveau $m_2(u)$
- On sélectionne $u \neq v$ tel que $m_1(u) + m_2(u)$ est minimum.
 - On retourne le cycle formé par le chemin de v à u dans T_1
 - Et celui de u à v dans T_2

Validité

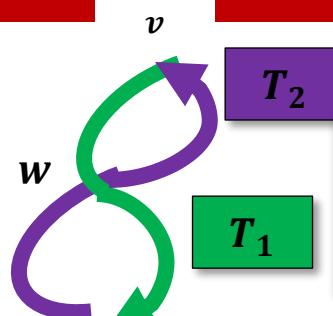
Le résultat est bien une marche orientée.

Pour vérifier que c'est un cycle,

il faut que les deux chemins dont on fait l'union n'ait pas de sommet commun.

Mais si c'était le cas,

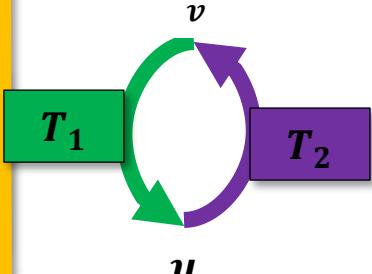
le sommet commun contredirait la minimalité de u :



*u est pas le sommet de plus petit niveau commun
DOC PAS CORRECT*

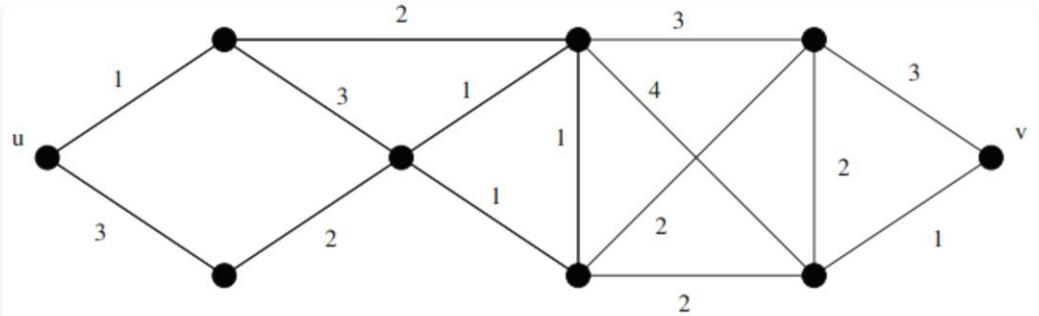
*Le cycle est bien un plus court cycle
car si ce n'est pas le cas,
on contredirait à nouveau la minimalité de u .*

CORRECT



Déterminer un arbre couvrant de poids minimal pour le graphe de la figure 2.3

Cours page 64



**Le faire deux fois,
avec Kruskal et Prim**

FIGURE 2.3 –

Exercice 2.7

Arbre couvrant de poids minimal

Dans le cas d'un graphe valué, on est intéressé par la recherche d'un arbre couvrant de poids minimal.

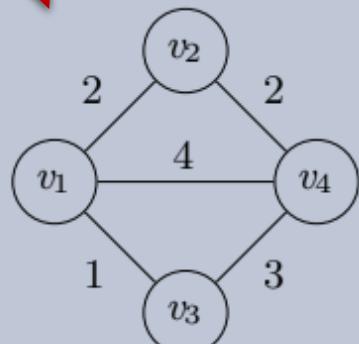
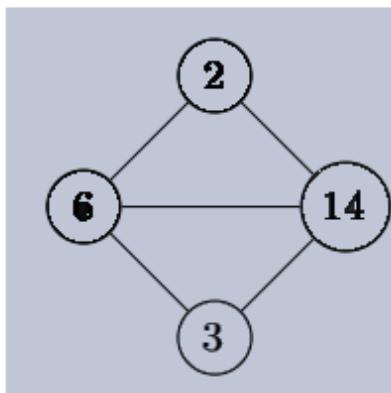
Problème

Soit G un graphe **valué**.

Trouver un arbre couvrant de poids minimal.

Types de graphes

- ▶ **valués** (sommets ou arêtes)



Un graphe sommets-valué est un graphe dont les sommets possèdent un vecteur de poids quantitatif.

Un graphe arêtes-valué est un graphe dont les arêtes possèdent un vecteur de poids quantitatif.

Un graphe non-valué est un graphe dont ni les sommets ni les arêtes ne possèdent de vecteur de poids quantitatif.

Algorithme de Kruskal

pseudo-code

L'algorithme construit un arbre couvrant minimum en sélectionnant des arêtes par poids croissant.

Plus précisément,

L'algorithme considère toutes les arêtes du graphe par poids croissant (en pratique, on trie d'abord les arêtes du graphe par poids croissant) et pour chacune d'elles, il la sélectionne si elle ne crée pas un cycle.

Kruskal(G)

Input : Un graphe connexe non-orienté valué G

$L = E(G)$

Une liste des arêtes ordonnées par poids croissant

$F = \emptyset$

L'arbre résultant

while F n'est pas un arbre **do**

e = première arête de L

$e = head(L)$, e est une arête

if les extrémités de e ne sont pas dans le même arbre de F **then**

$| F = F + e$

$F = F \cup \{e\}$

end

 Supprimer e de L

$L = L \setminus \{e\}$

end

return F

Output : F l'arbre couvrant de G de poids minimal

Si les extrémités de e ne sont pas dans le même arbre de F

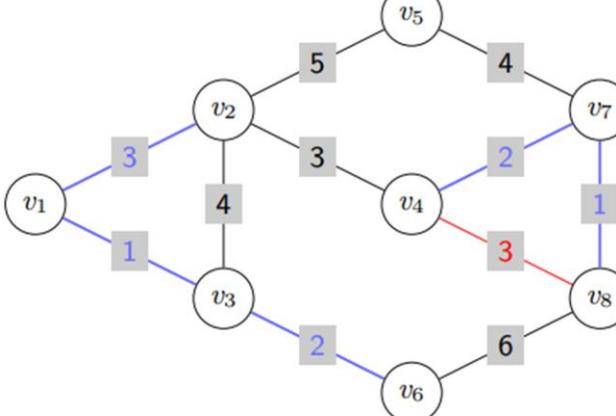
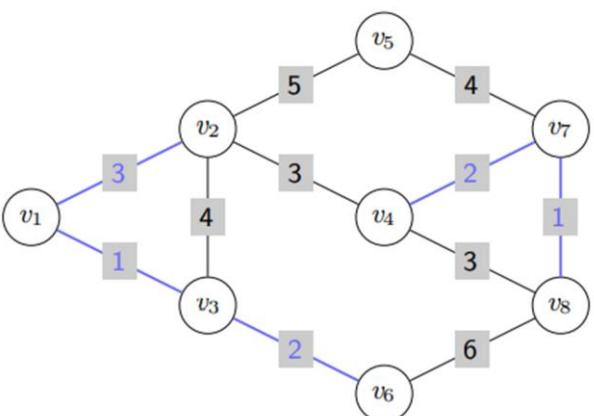
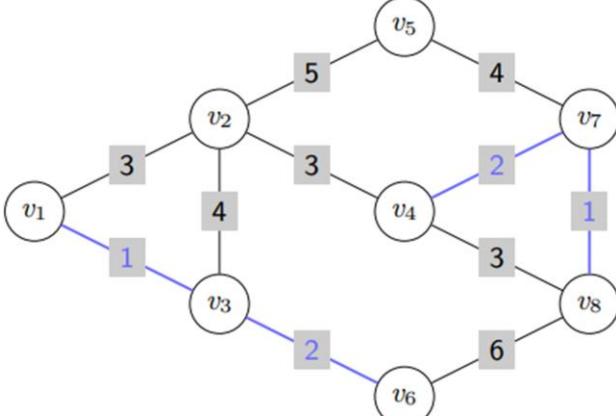
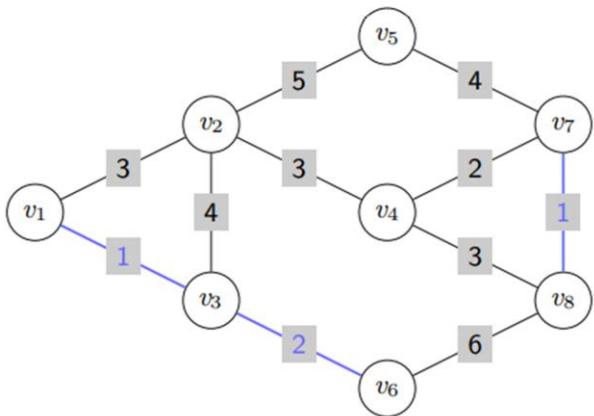
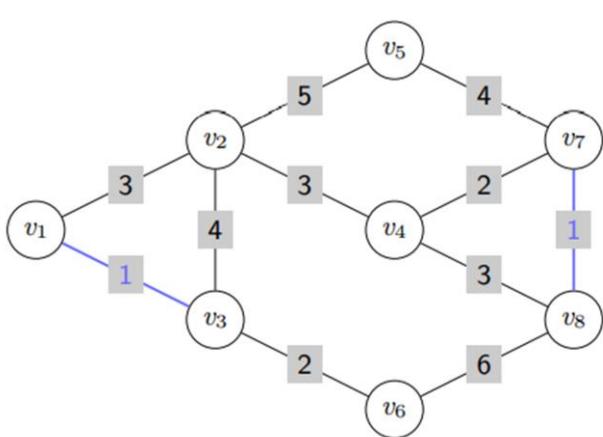
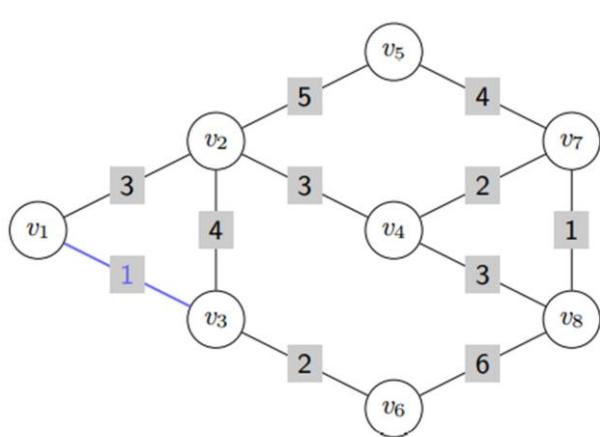
Examiner dans l'ordre chacune des arêtes $e = \{x, y\}$

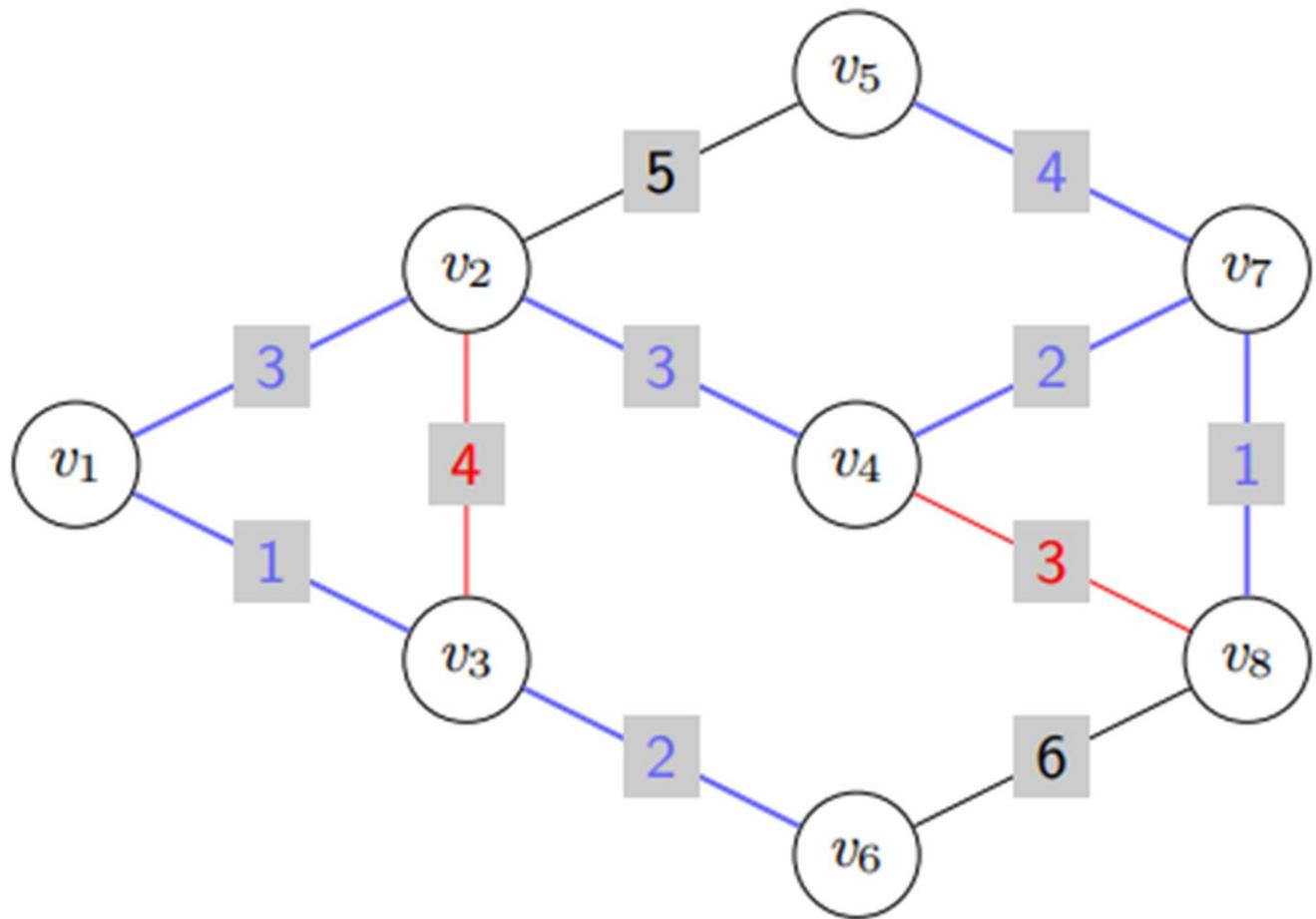
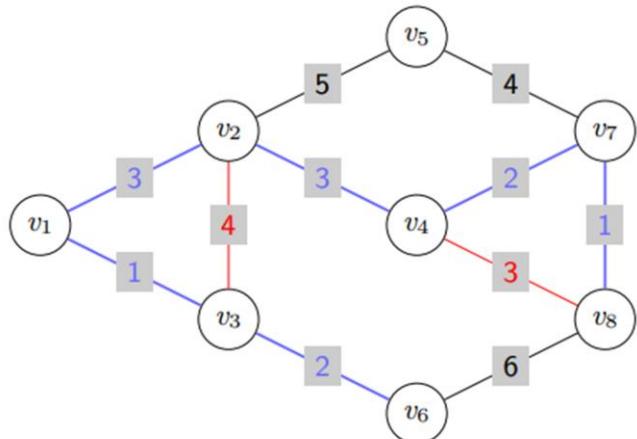
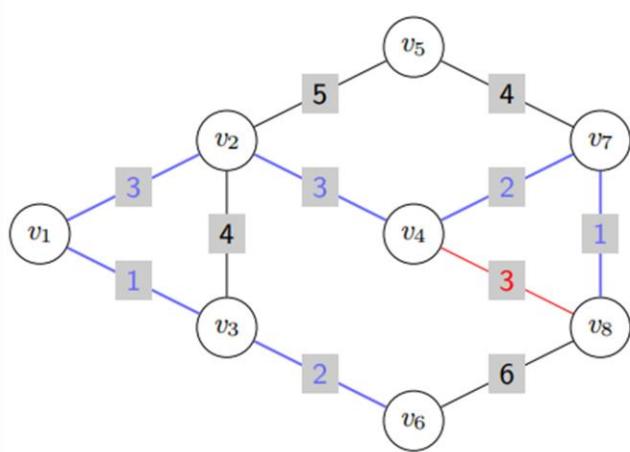
S'il n'existe pas de chaîne de x à y dans F

***Alors* $F = F + e$**

Si $F \cup \{e\}$ ne crée pas un cycle

Algorithme de Kruskal : exemple





Déterminer un arbre couvrant de poids minimal pour le graphe de la figure 2.3

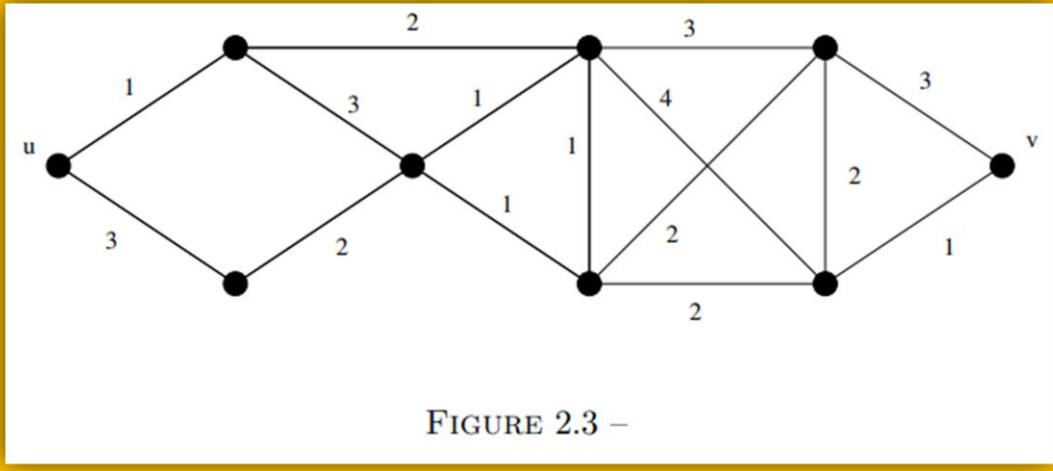
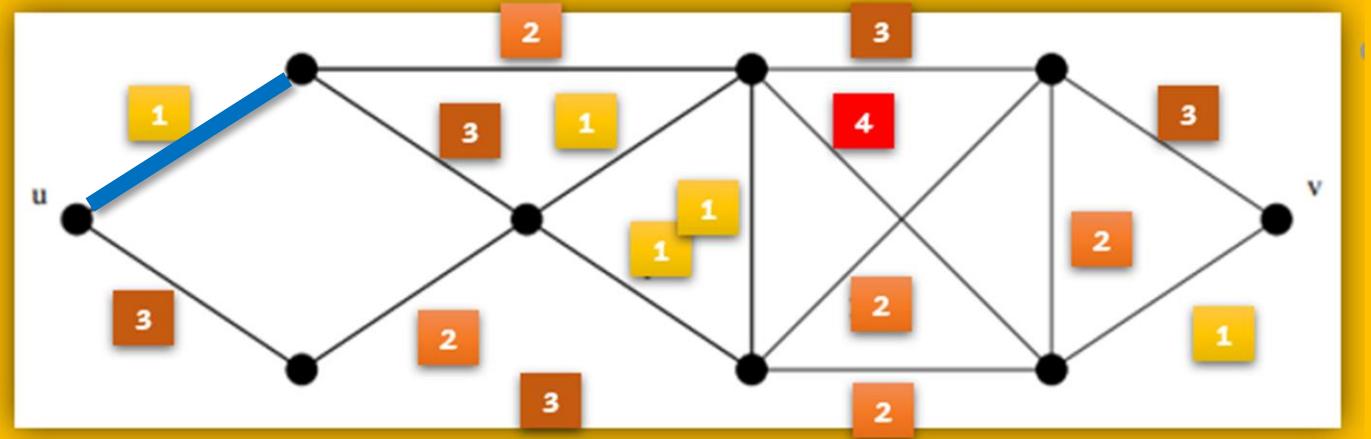
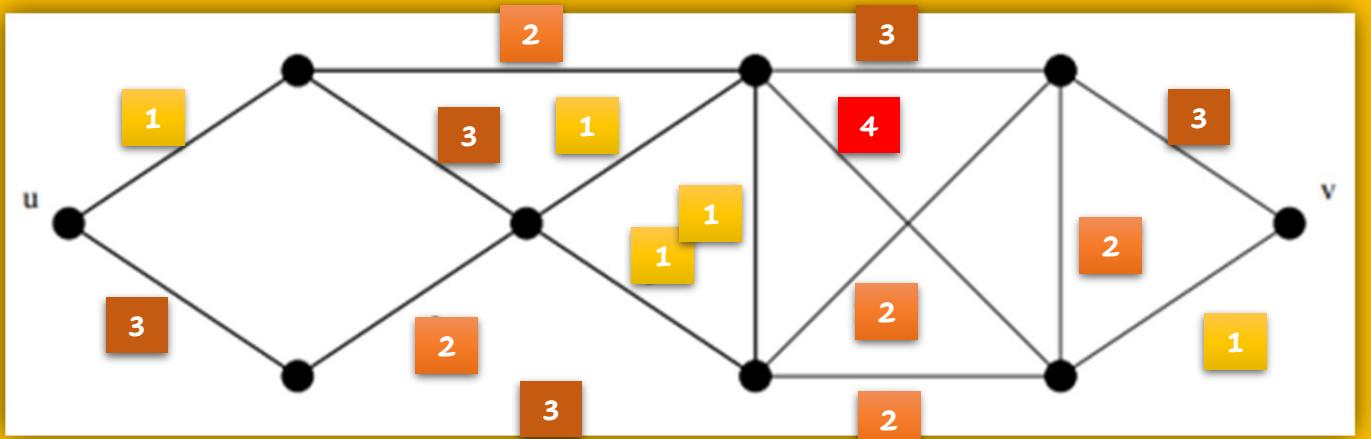
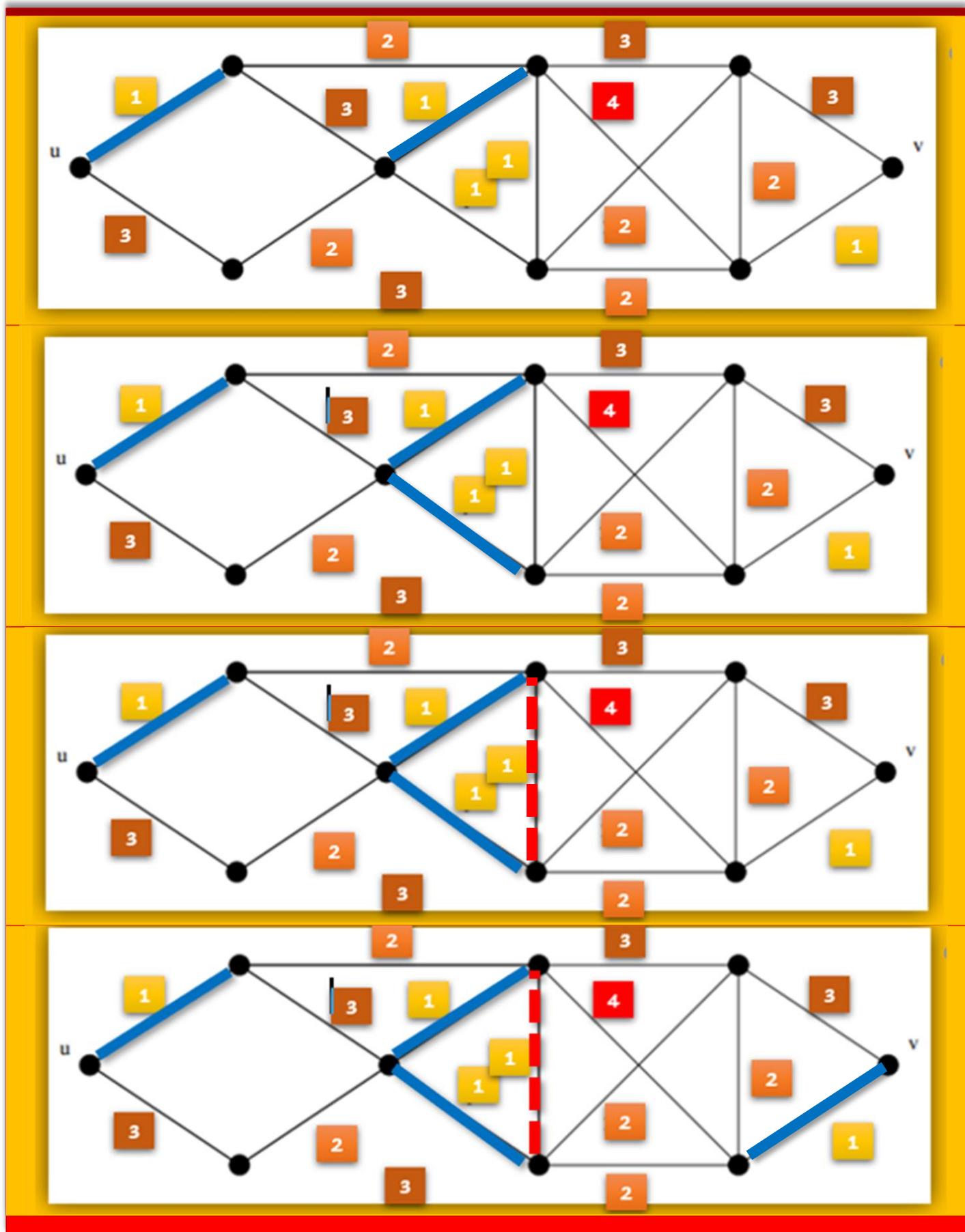
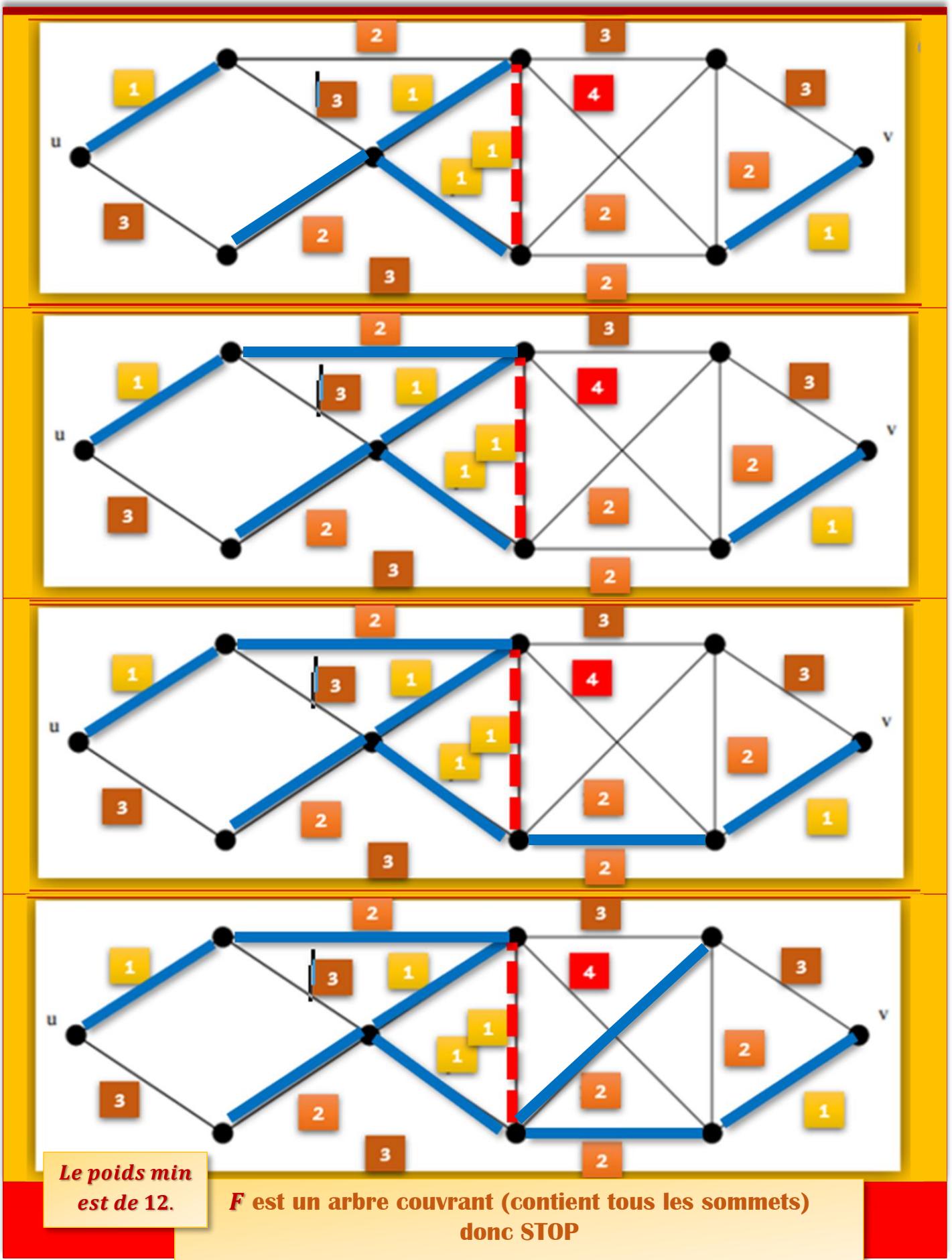


FIGURE 2.3 –







Algorithme de Prim

pseudo-code

L'algorithme de Prim est un algorithme qui calcule un arbre couvrant minimal dans un graphe connexe valué et non orienté.

L'algorithme consiste à faire croître un arbre depuis un sommet. On commence avec un seul sommet puis à chaque étape, on ajoute une arête de poids minimum ayant exactement une extrémité dans l'arbre en cours de construction. En effet, si ses deux extrémités appartenaient déjà à l'arbre, l'ajout de cette arête créerait un deuxième chemin entre les deux sommets dans l'arbre en cours de construction et le résultat contiendrait un cycle.

Prim(G, v)

Input : Un graphe connexe non-orienté valué G
et un sommet v

$T = \{v\}$

Initialiser T avec un sommet de G qu'on choisit

G/T

sommets qui se trouve dans G et pas dans T

while il existe des arêtes de T vers $G \setminus T$ do

tant que ($|V(T)| < |V(G)|$)

 Trouver l'arête e de poids minimal de T vers $G \setminus T$

$T = T + e$

$T = T \cup \{e\}$

$e = (u, v)$

= arête de poids min ayant $u \in V(T)$ et $v \in V(G) - V(T)$

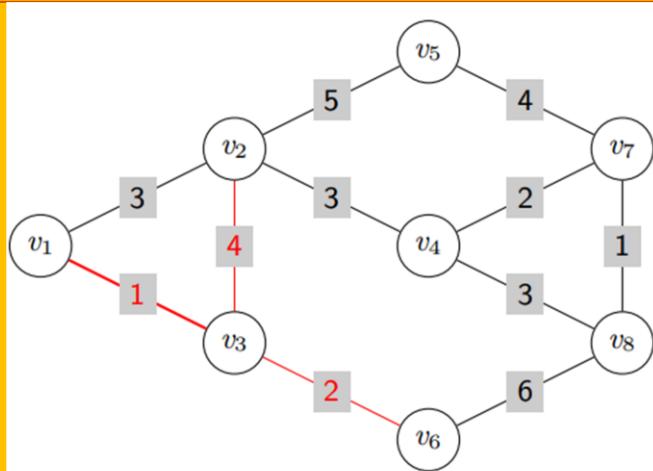
end

return T

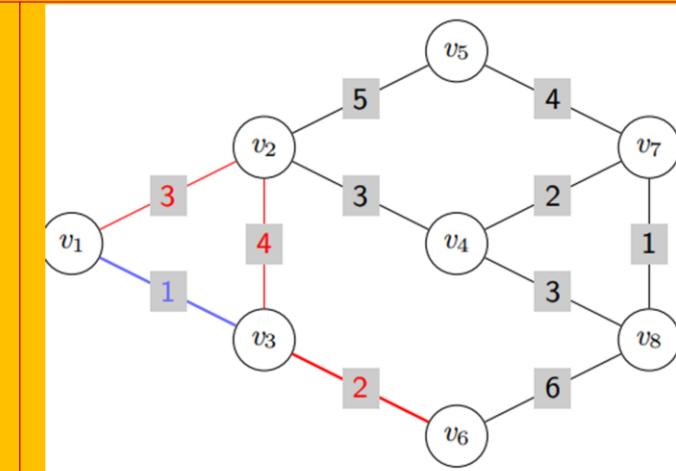
Output : T l'arbre couvrant de G de poids minimal

T est un arbre par construction puisque toute arête relie l'arbre existant à un sommet non encore découvert.

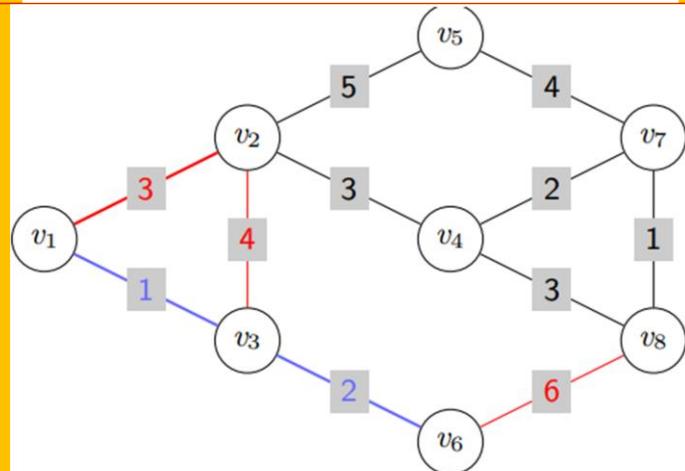
Algorithme de Prim : exemple



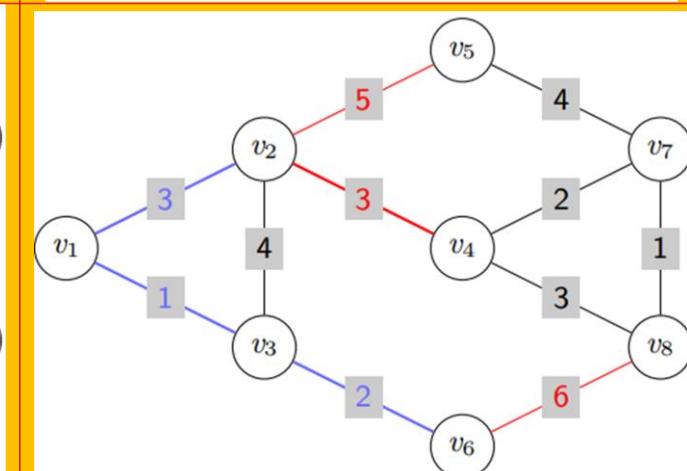
$$V(T) = \{v_3\}$$



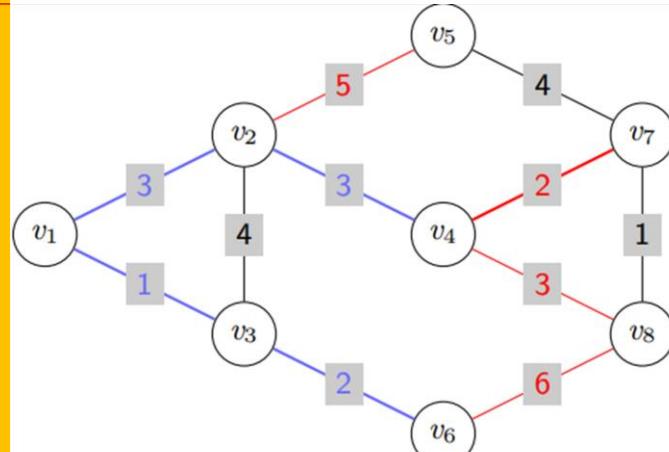
$$V(T) = \{v_3, v_1\}$$



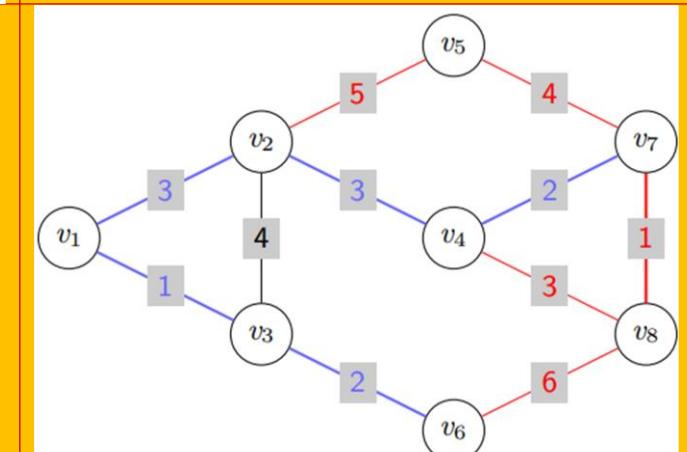
$$V(T) = \{v_3, v_1, v_6\}$$



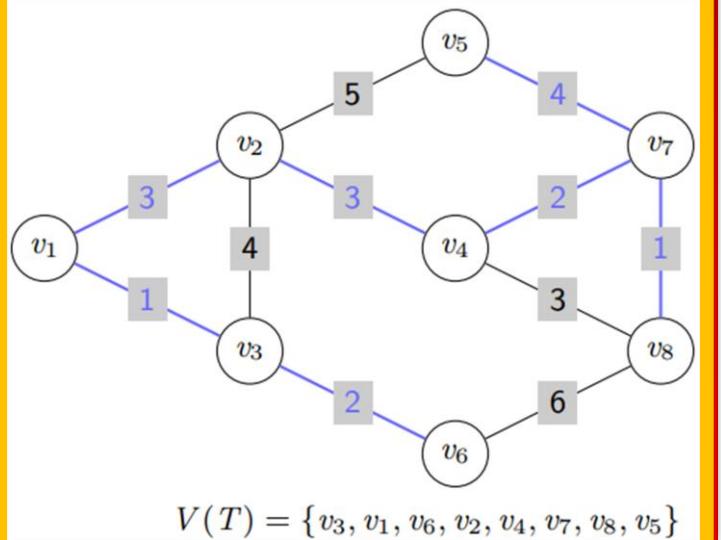
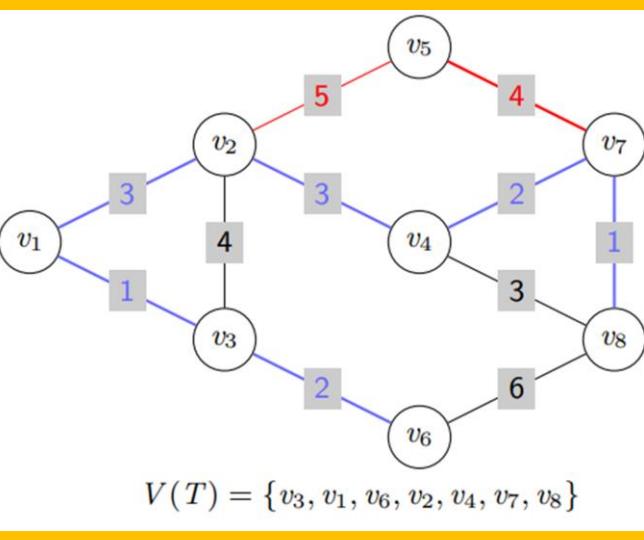
$$V(T) = \{v_3, v_1, v_6, v_2\}$$



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



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



Déterminer un arbre couvrant de poids minimal
pour le graphe de la figure 2.3

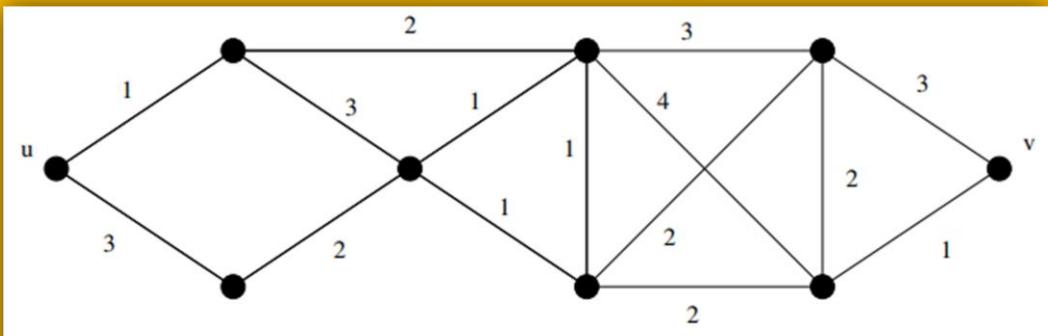
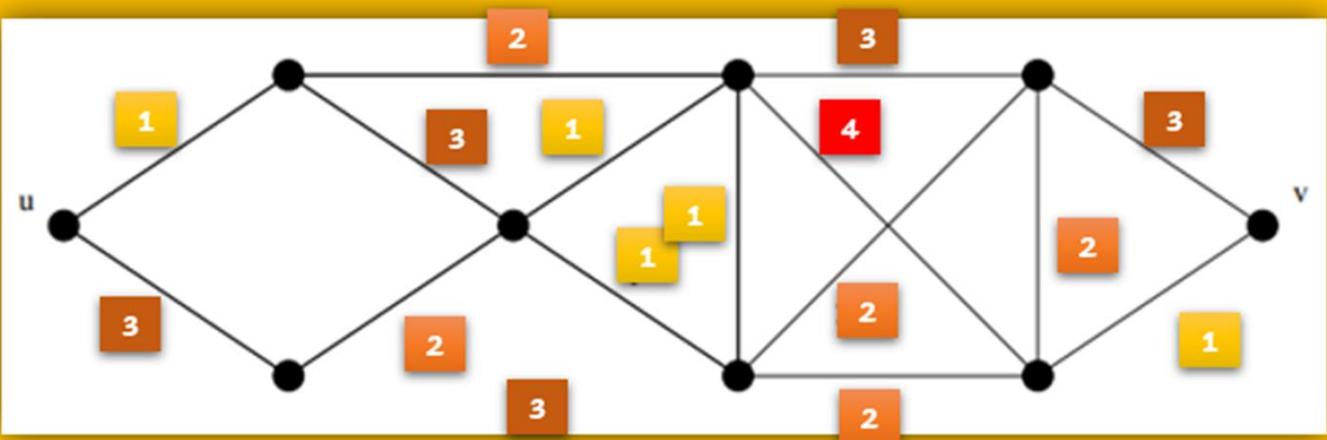
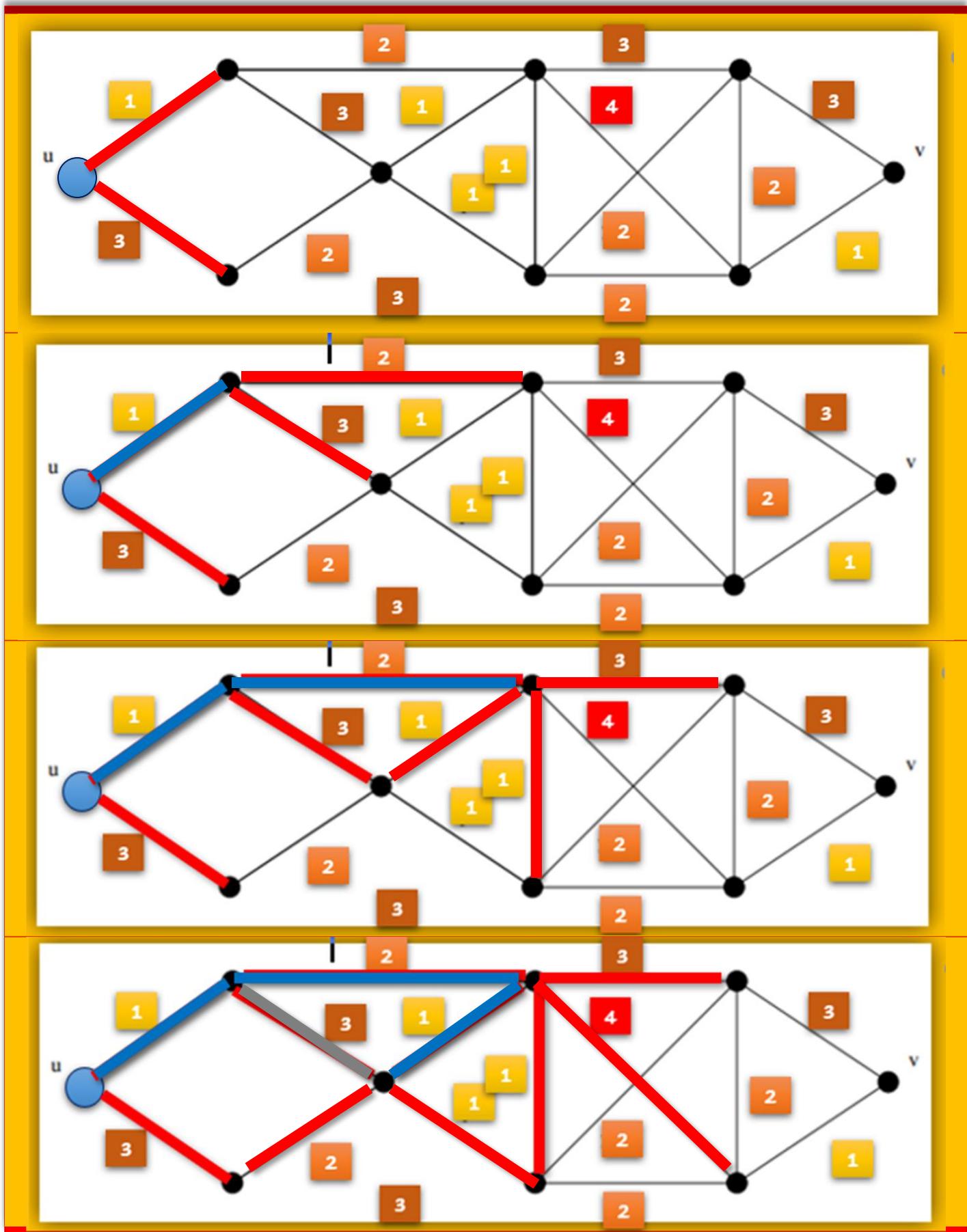
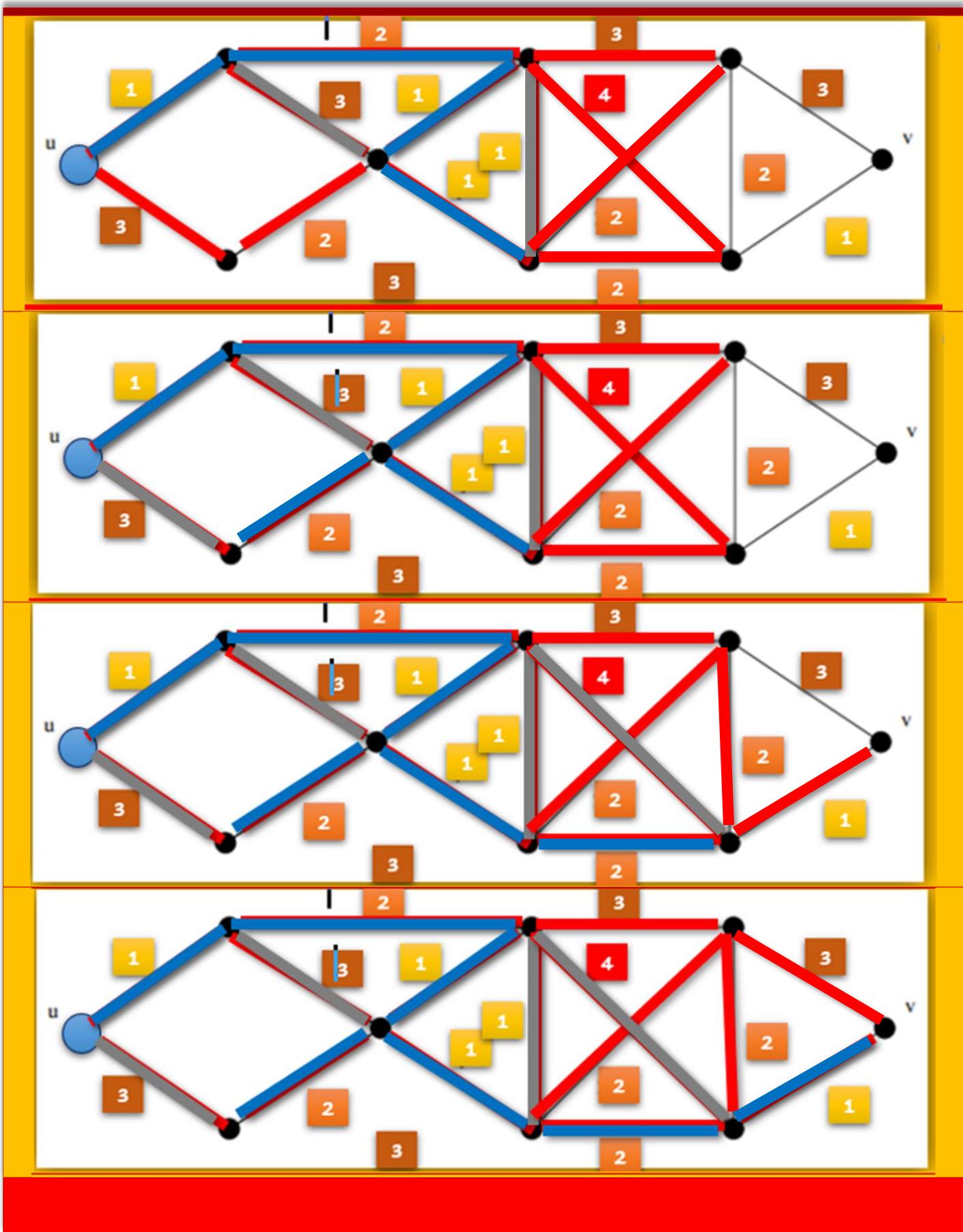
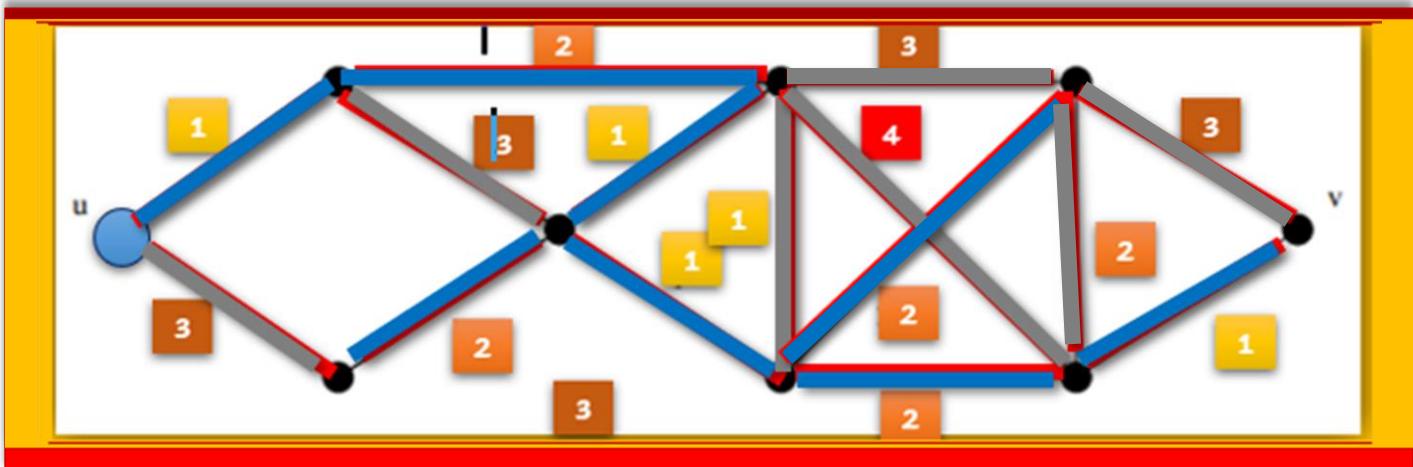


FIGURE 2.3 –









On considère un graphe valué **G**
et un ensemble **U** de sommets.

Exercice 2.8

Proposer un algorithme

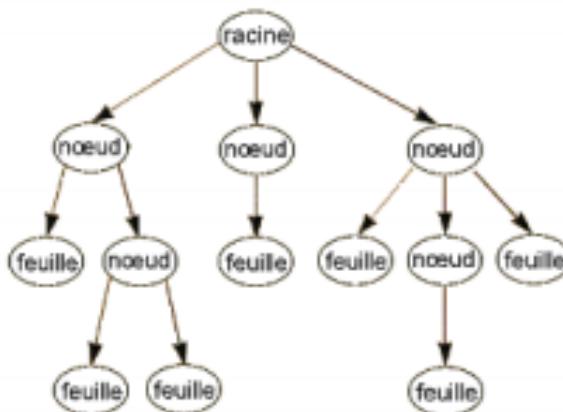
qui détermine un arbre couvrant de poids minimal parmi ceux
tels que tous les sommets de **U** sont des feuilles.

Démontrer sa validité et préciser sa complexité

on appellera souvent **noeuds** les sommets des arbres (enracinés).

Feuille, Noeud interne

Un noeud sans fils est appelé **noeud externe**, **noeud terminal** ou plus simplement **une feuille**. Un noeud qui n'est pas une feuille est un **noeud interne**.



Solution

Kruskal(G)

```
L = E(G)
F = ∅

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

return F
```

On considère un graphe valué G et un ensemble U de sommets.

Proposer un algorithme

qui détermine un arbre couvrant de poids minimal parmi ceux tels que tous les sommets de U sont des feuilles.

On reprend l'algorithme de Kruskal,

mis à part que à chaque étape on ne rejoute l'arête e que si

1. Elle ne crée pas de cycle
2. Elle n'est pas la deuxième arête incidente (accessoire, secondaire) à un sommet de U .

Algorithme de Kruskal

Proposition

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

La complexité est la même que celle de Kruskal

**Appliquer l'algorithme de Dijkstra
au graphe de la figure 1.2**

2.4

Exercice 2.9

pour déterminer le chemin de poids minimal entre u et v .

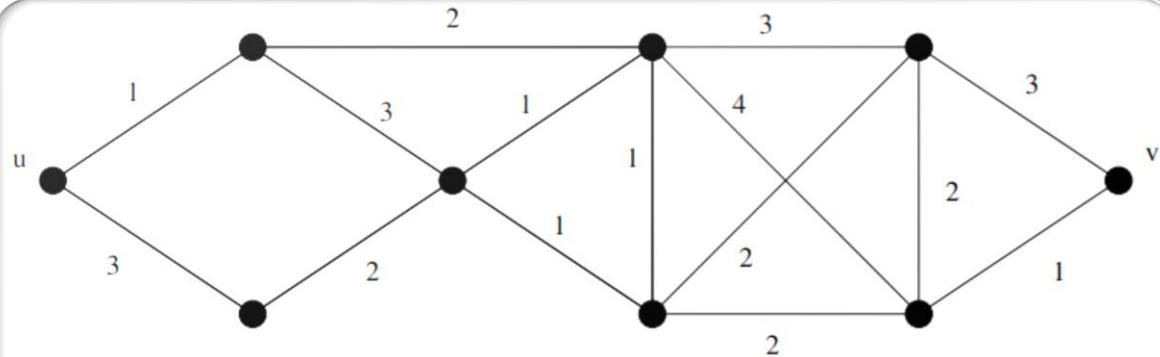


FIGURE 2.4 – .

L'algorithme de Dijkstra sert à résoudre le problème du plus court chemin.

- Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région.
- Plus précisément, il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs.
- On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée.

Algorithme de Dijkstra

Objectif Trouver le plus court chemin entre deux sommets donnés d'un graphe.

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 .

Problématique Calcul de la distance entre le sommet u et tous les autres sommets du graphe

- ▶ le problème est trivial à résoudre dans un arbre.
- ▶ l'algorithme de Dijkstra résout un problème plus général

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

Algorithme de Dijkstra

Input : Un graphe connexe valué G de fonction de poids ω et un sommet u de G

Dijkstra(G, u)

L'algorithme prend en entrée un graphe orienté pondéré par des réels positifs et un sommet source.

Il s'agit de construire progressivement un sous-graphe T dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet de départ. La distance correspond à la somme des poids des arcs empruntés.

```

 $T = \{u\}$   $V(T) = \text{noeud\_visites}$ 
for  $v \in V(G)$  do     Pour chaque sommet  $v$  de  $G$  faire
    |  $\text{dist\_prov}(v) := \infty$ ;  $\text{pere}(v) := \emptyset$ 
end     distance provisoire
 $\text{dist\_finale}(u) := 0$ ;  $\text{dernier\_ajout} := u$ 
        distance finale     dernier ajout à  $T$ 

```

Initialisation

Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle.

```

while  $V(T) \neq V(G)$  do     Tant que  $V(T)$  [= noeuds_visites] ne contient pas tous les nœuds de  $G$  faire
    for  $v$  voisin de  $\text{dernier\_ajout}$  do     On met à jour les distances des sommets voisins de celui ajouté
        if  $\text{dist\_finale}(\text{dernier\_ajout}) + \omega(\text{dernier\_ajout}, v) < \text{dist\_prov}(v)$ 
            then
                |  $\text{dist\_prov}(v) := \text{dist\_finale}(\text{dernier\_ajout}) + \omega(\text{dernier\_ajout}, v)$ 
                |  $\text{pere}(v) := \text{dernier\_ajout}$ 
            end
    end
end

```

La nouvelle distance du sommet voisin est le minimum entre la distance existante et celle obtenue en ajoutant le poids de l'arc entre sommet voisin et sommet ajouté à la distance du sommet ajouté.

Selectionner $v \notin V(T)$ tel que dist_prov est minimum
Ajouter $(\text{pere}(v), v)$ à T
 $\text{dist_finale}(v) := \text{dist_prov}(v)$
 $\text{dernier_ajout} := v$

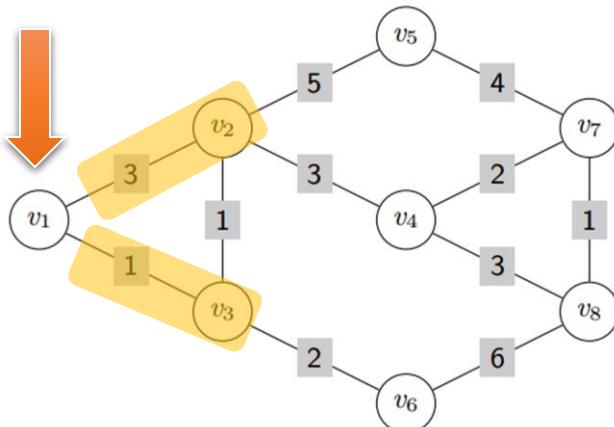
Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe

end **Output :** Un arbre couvrant T et la distance $D(v) = d_G(u, v)$ pour tout v

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 .

Algorithme de Dijkstra: exemple

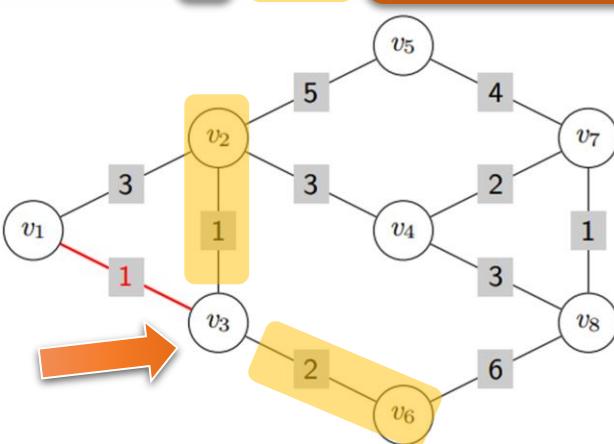
Au départ, on considère que les distances de chaque sommet au sommet de départ sont infinies, sauf pour le sommet de départ pour lequel la distance est nulle.



On met à jour les distances des sommets voisins de celui ajouté

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

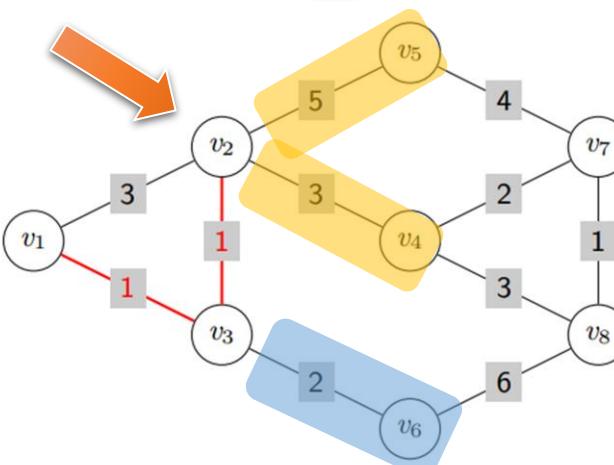
Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe



On met à jour les distances des sommets voisins de celui ajouté

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	∞	∞

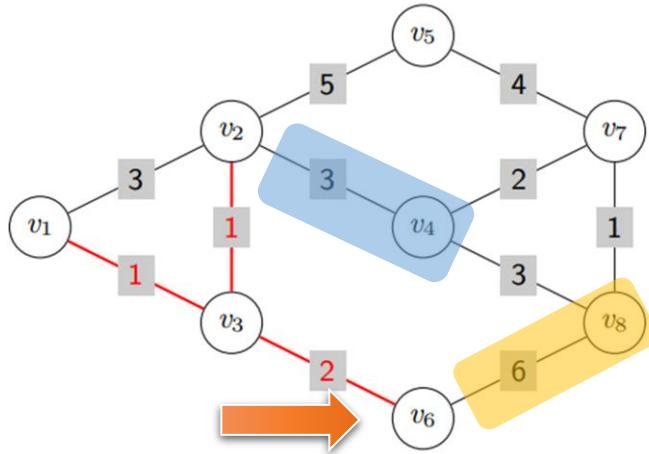
Au cours de chaque itération, on choisit en dehors du sous-graphe un sommet de distance minimale et on l'ajoute au sous-graphe



On met à jour les distances des sommets voisins de celui ajouté

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	∞	∞

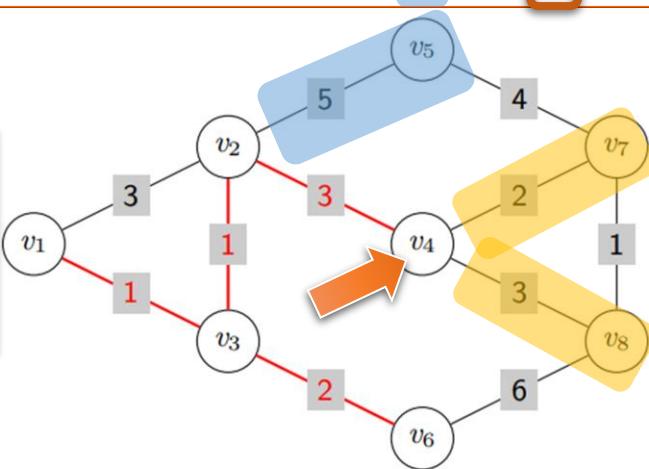
**Au cours de chaque itération,
on choisit en dehors du sous-
graphe un sommet de
distance minimale et on
l'ajoute au sous-graphe**



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

**On met à
jour les
distances des
sommets
voisins de
celui ajouté**

**Au cours de chaque itération,
on choisit en dehors du sous-
graphe un sommet de
distance minimale et on
l'ajoute au sous-graphe**

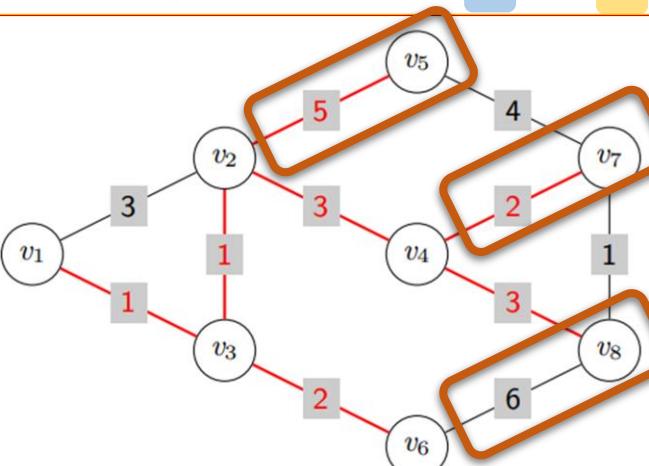


Sommet	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Distance (rouge si finale)	0	2	1	5	7	3	7	8

**On met à jour les
distances des
sommets voisins
de celui ajouté**

**La nouvelle distance du
sommet voisin est le
minimum entre la
distance existante et celle
obtenue en ajoutant le
poids de l'arc entre
sommel voisin et sommet
ajouté à la distance du
sommel ajouté.**

**Au cours de chaque itération,
on choisit en dehors du sous-
graphe un sommet de
distance minimale et on
l'ajoute au sous-graphe**



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

