Algorithmique et complexité

Manipuler des structures de données avancées : arbres et graphes

Déroulement du cours

- 20 sessions de cours de 1h30
- Contrôle continu : implémentations dans le langage de programmation de votre choix des algorithmes mis en place en TD
- Examen final sous forme d’un projet : comparatif des méthodes de parcours de graphe pour estimer des vitesses de parcours sur un réseau de type autoroutier
- Cours puis exercices d’applications
- Les implémentations proposés ne répondent à aucun langage de programmation actuel. L’objectif est de transmettre la compréhension de l’algorithme à mettre en œuvre de manière rigoureuse (typage de chaque variable, opérations sur les structures usuelles de données informatiques).

Projet – Examen final

<u>Sujet :</u> Vous construirez un graphe représentant un réseau de transport existant constitué d'environ 10 à 20 destinations, présentant une structure de type graphe (avec au moins un cycle).

Vous créerez un outil permettant à un utilisateur de choisir un point de départ, un point d'arrivé et de lui indiquer l'itinéraire à suivre. Cet itinéraire sera calculée via au moins 2 manières différentes.

Vous comparerez les résultats en temps de transport prévu pour l'utilisateur de ces différentes méthodes et présenterez la complexité en temps que ces algorithmes ont engendrés dans leur calcul.

Sommaire

Sommaire

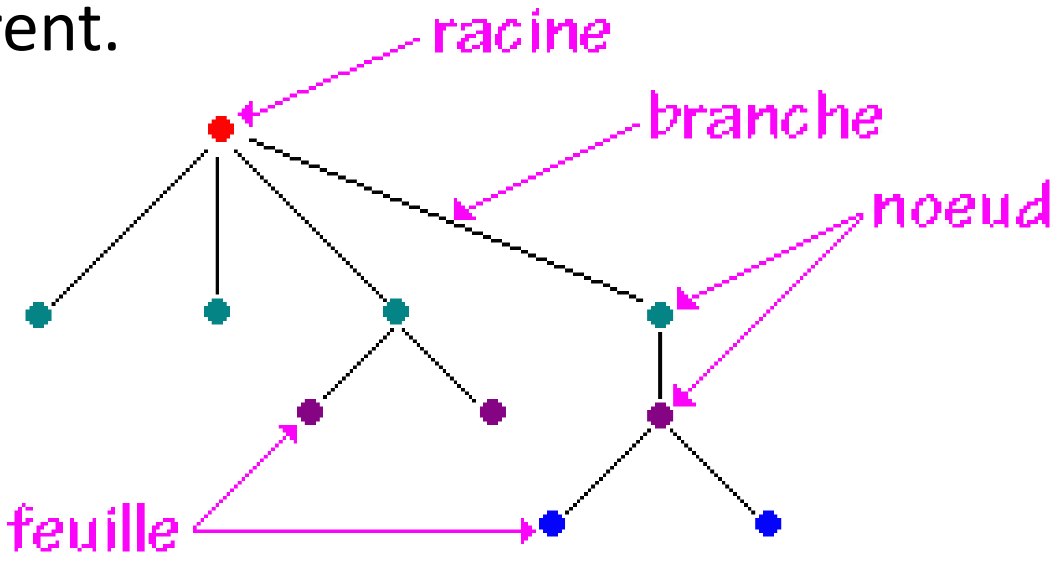
2. Graphe
    1. Définitions et représentations
    2. Parcours en largeur
    3. Parcours en profondeur
    4. Fermeture transitive et symétrique
    5. Chemin et existence de chemin
    6. Tri topologique
    7. Composante connexes/fortement connexe
    8. Recherche du plus court chemin (Algorithme de Dijkstra)
    9. Arbre couvrant de poids minimal :
        1. Algorithme de Prim
        2. Algorithme de Kruskal

Partie 1

Les Arbres

1. Arbres - Définitions

Arbre (appliqué à l'informatique) : Un arbre est une structure de données constituées de nœuds, où chaque nœud peut avoir plusieurs nœuds enfants. Un arbre de contient pas de cycle : un nœud ne peux avoir qu'il seul parent.

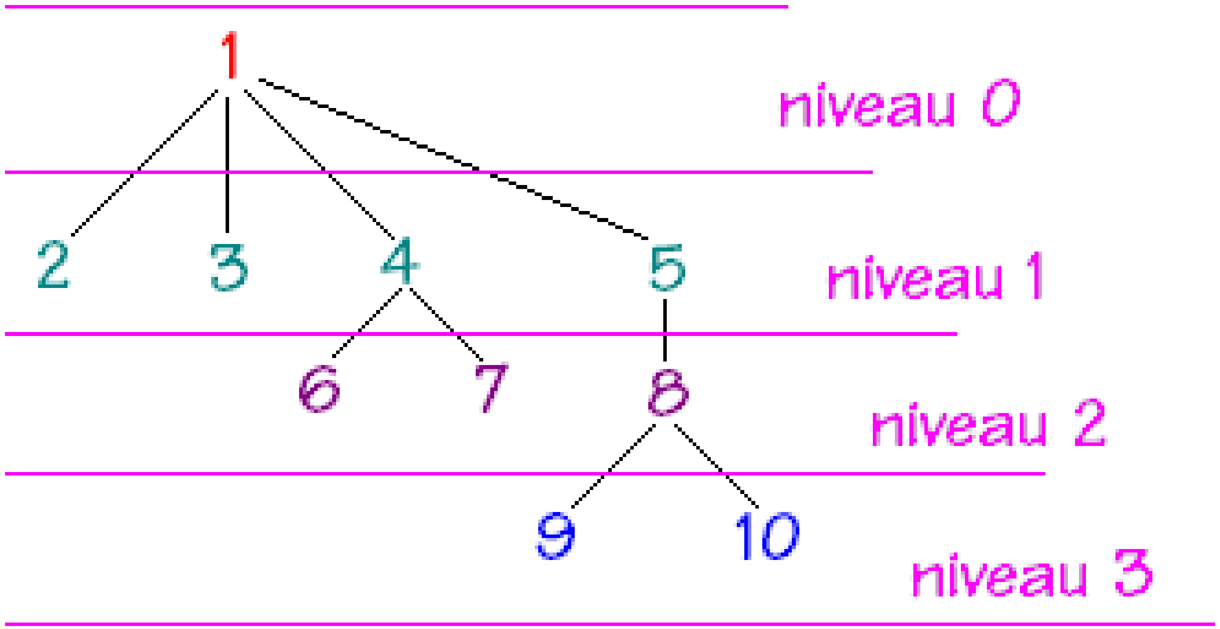


1. Arbres - Définitions

- Racine d’un arbre : C’est le seul nœud qui n’est pas enfant d’un autre nœud. Pour permettre de mieux identifier un nœud, on lui donne un identifiant. On peux alors y attacher toute objet.
- Branche : Elément représentant la filiation entre 2 nœuds
- Feuille : Nœud ne possédant pas de nœuds enfants
- Chemin: Suite des nœuds par lesquels il faut passer pour aller d’un nœud de départ à un nœud de destination. Cela suppose qu’une branche existe entre chaque nœud consécutif de cette liste.
- Degré d’un nœud : nombre de nœuds enfant du nœud concerné
- Degré d’un arbre : plus grand degré pouvant être calculé sur un arbre

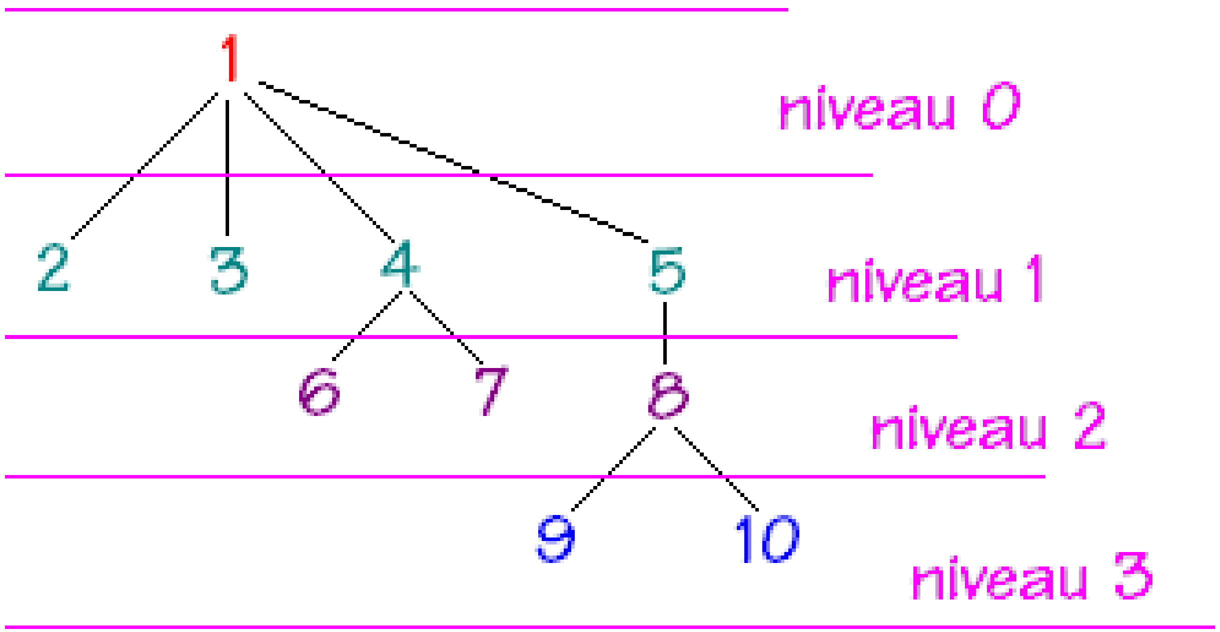
1. Arbres - Définitions

- On représente un arbre en disposant les nœud de même niveau sur une même ligne



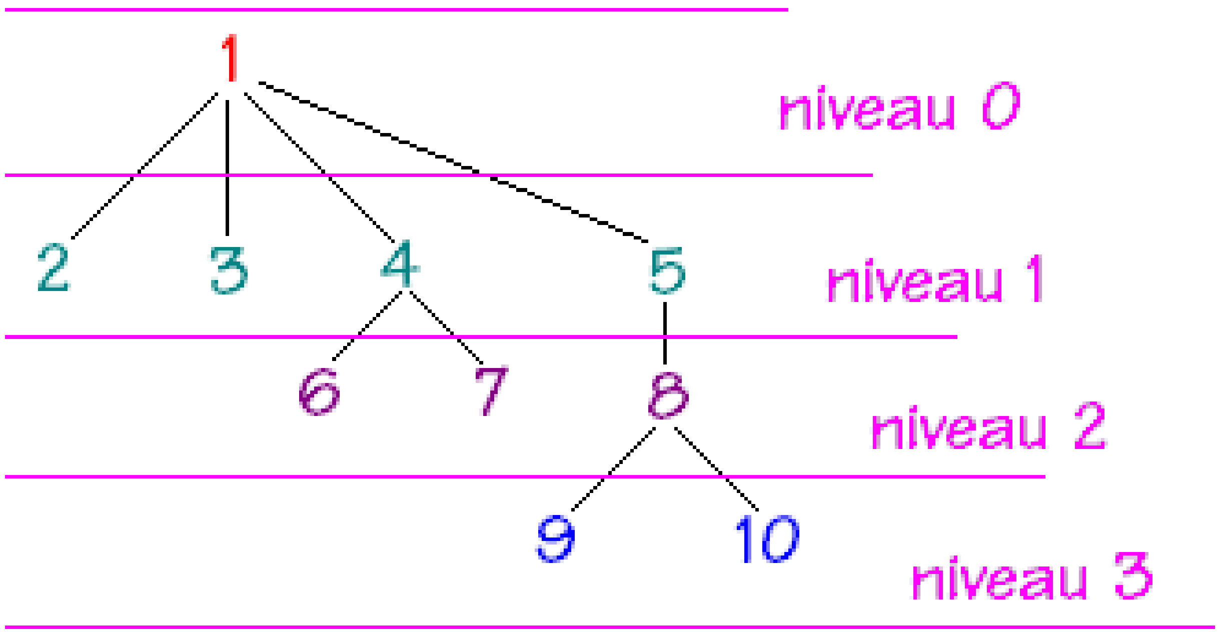
1. Arbres - Définitions

- Profondeur (ou niveau) d’un nœud : nombres de nœuds compris dans le chemin entre la racine de l’arbre et ce nœud.



1. Arbres - Définitions

- Hauteur d’un arbre : plus grande profondeur pouvant être calculé sur cette arbre



1. Arbres – exemple d’implémentation

```
Class Tree {
	int id;
	Tree[] children;
	Tree* parent;
}
```

2. Arbres – Parcourir un arbre

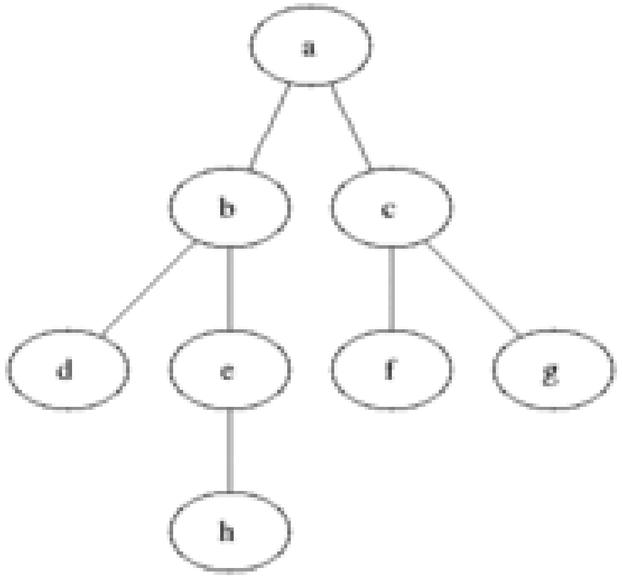
Différentes méthodes courantes de parcours d’un arbres

Chaque méthode a ses avantages/inconvénients en fonction du type d’action à effectuer

Tout parcours d’arbre permet d’extraire une liste ordonnées de nœuds

2. Arbres – Parcours en largeur

On explore les nœud par niveau, toujours dans le même sens (usuellement de gauche à droite), en commençant par la racine.



2. Arbres – Parcours en largeur - Implémentation

```
function breadthFirstSearch(Tree T) {
        Queue currentNodes = new Queue();
        currentNodes.push(T); //We add an element to the queue
        Tree[] result = []; //We create a dynamic list of elements
        While(!currentNodes.isEmpty()) {
                Tree currentNode = currentNodes.pop(); //We make one item go out from
                                                         the queue
                result.push(currentNode);
                foreach(currentNode.children as child) {
                        currentNodes.push(child);
        }
}
```

3. Arbres – Parcours en profondeur

Il existe trois type de parcours en profondeur sur les arbres :

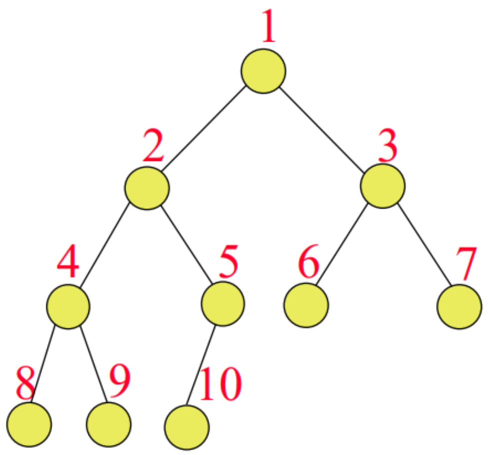
- Parcours en profondeur préfixe
- Parcours en profondeur suffixe
- Parcours en profondeur infixe

3. Arbres – Parcours en profondeur préfixe

On explore le nœud courant

On extrait ce nœud

On applique ce même traitement à chaque nœud enfant, toujours dans le même ordre (de gauche à droite)



donne 1, 2, 4, 8, 9, 5, 10, 3, 6, 7

3. Arbres – Parcours en profondeur préfixe - implémentation

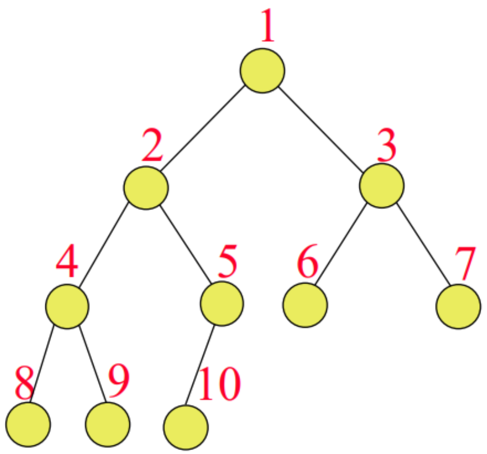
```
Function depthFirstSearch(Tree T, Tree[] *result ) {
	result.push(T);
	foreach(T.children as child) {
		depthFirstSearch(child, result);
	}
	return result;
}
```

3. Arbres – Parcours en profondeur suffixe

On explore le nœud courant

Si ce nœud n'a pas d'enfant, on l'extrait

Sinon on applique ce même traitement à chaque nœud enfant, toujours dans le même ordre (de gauche à droite), puis on extrait le nœud courant



donne 8, 9, 4, 10, 5, 2, 6, 7, 3, 1

3. Arbres – Parcours en profondeur suffixe - implémentation

```
Function suffixSearch(Tree T, Tree[] *result ) {
        foreach(T.children as child) {
                suffixSearch(child, result);
        }
        result.push(T);
        return result;
}
```

3. Arbres – Parcours en profondeur infixe

Ce type de parcours envisageable uniquement pour un arbre binaire.

Un arbre binaire est un arbre de degrés au plus 2.

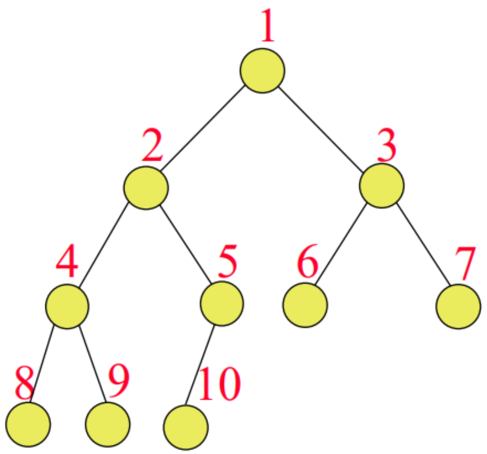
3. Arbres – Parcours en profondeur infixe

On explore le nœud courant

Si ce nœud dispose d'un nœud gauche, on applique ce même traitement au nœud gauche.

On extrait le nœud courant

Si ce nœud dispose d'un nœud droit, on applique ce même traitement.



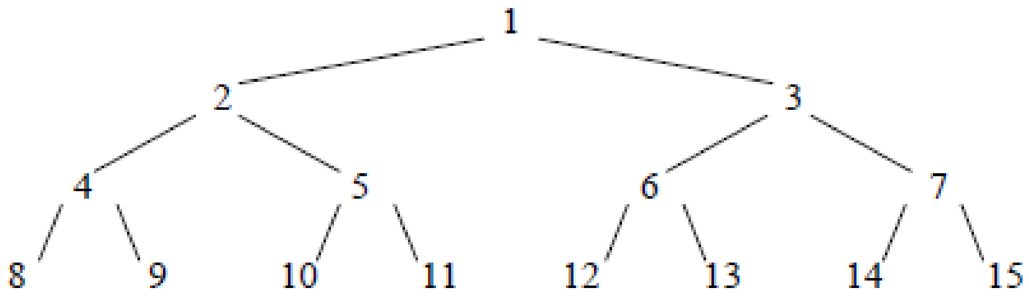
donne 8, 4, 9, 2, 10, 5, 1, 6, 3, 7

->TD1

4. Tas

Quelques définitions supplémentaires :

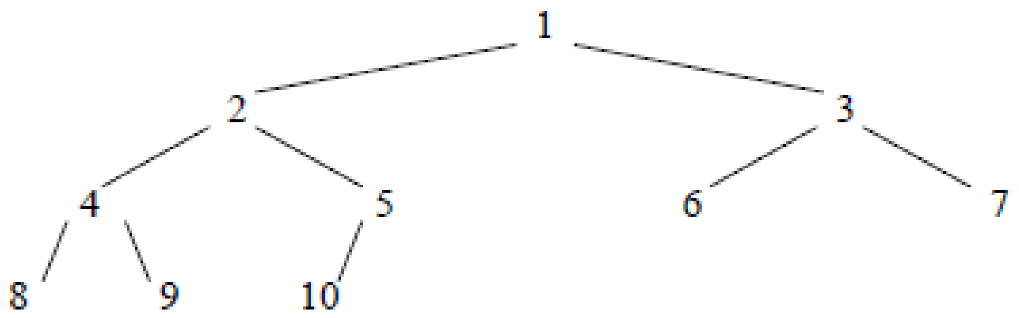
- Arbre binaire entier : tout les nœuds possèdent 0 ou 2 nœuds enfants
- Arbre binaire complet : arbre binaire entier où toutes les feuilles sont à la même profondeur



4. Tas

Quelques définitions supplémentaires :

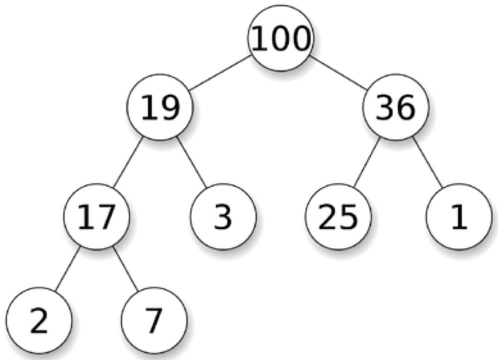
- Arbre binaire presque complet : arbre binaire où tous les niveau sont pleins, sauf éventuellement le dernier niveau où les nœuds sont remplis de manière consécutive en partant de la gauche.
- Etiquette : valeur associée à un nœud pouvant être comparé



4. Tas

Un tas est un arbre binaire presque complet vérifiant une des propriétés suivante :

- Chaque nœud a une étiquette supérieure aux étiquettes de tous ses nœuds enfants
- Chaque nœud a une étiquette inférieure aux étiquettes de tous ses nœuds enfants



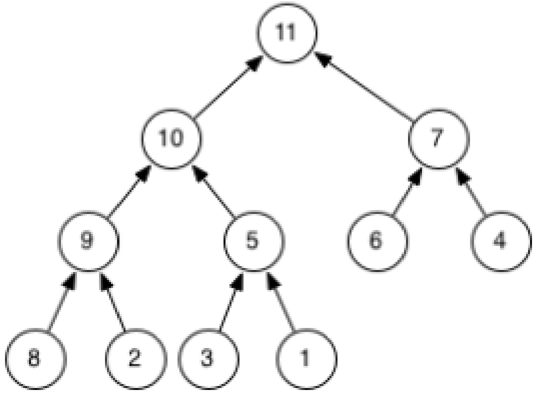
4. Tas – représentation possible en tableau

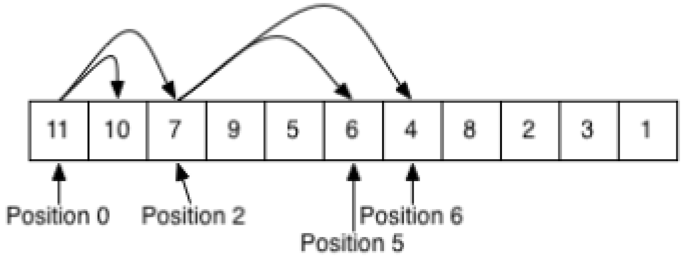
Un tas est équivalent, au niveau de sa structure de données, à un tableau.

En effet, en plaçant la racine de l'arbre à l'indice 0 du tableau, pour tout nœud à l'indice i :

- Chaque fils gauche est à l'indice 2*i + 1
- Chaque fils droit est à l'indice 2*i + 2
- Le père de ce nœud est à l'indice PartieEntière($\frac{i-1}{2}$)

Cette propriété est en fait valable pour tout arbre binaire. En revanche, dans le cas d'arbre binaire qui ne sont pas presque complet, de la mémoire peut être utilisé de manière inutile.





4. Tas – insertion

- Tamisage : Opération permettant de construire un tas à partir d’un arbre binaire quelconque. Cela s’effectue par déplacement et inversion de nœuds.
- Insertion : On insère un élément à une place libre sur le dernier niveau ou on créer un nouveau niveau. On tamise le tas. Complexité : O(log(n))
- Supprimer : On se limite à supprimer la racine dans un tas - complexité O(log(n)). On remplace la racine par le dernier nœud du tas puis on tamise.
  En cas de suppression d’un autre nœud, on reconstruit un nouveau tas en omettant l’élément à supprimer. Complexité : O(n)

5. Tri par tas

Algorithme de tri : permet d’ordonner une succession d’éléments

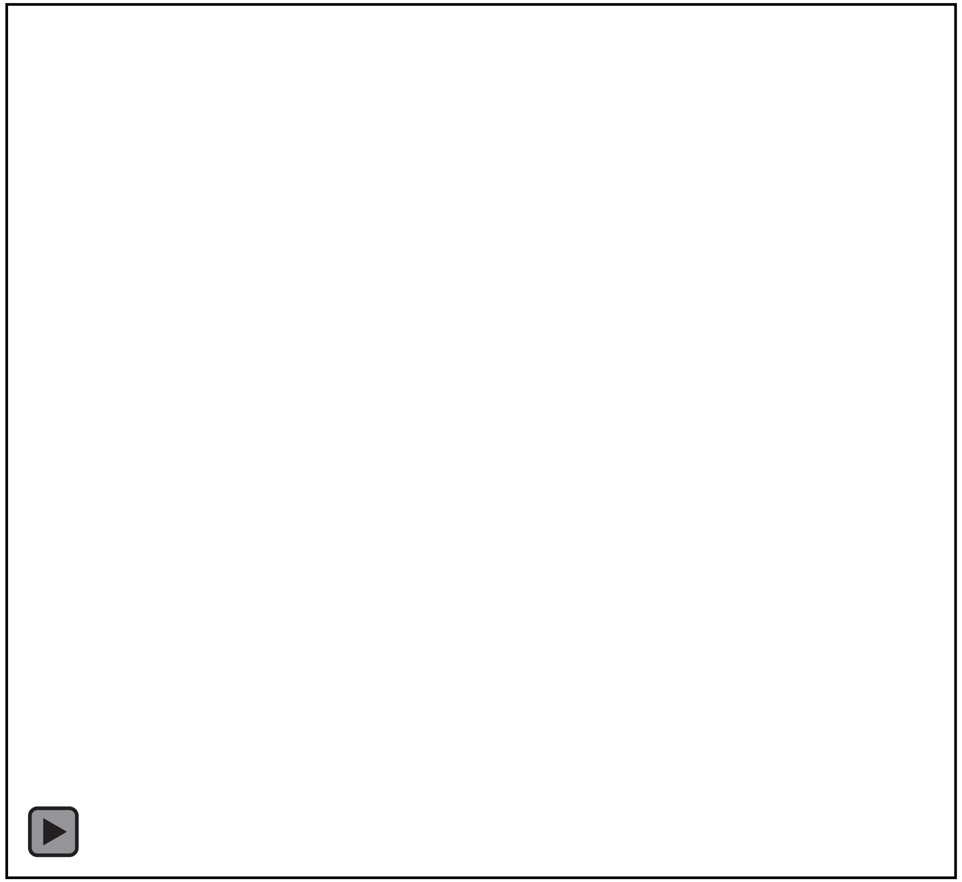
Le tri par tas s’appuis sur la notion de tas pour effectuer un tri. En effet, la racine d’un tas est toujours le plus petit/grand. Un algorithme peut être mis en œuvre autour de cette propriété

5. Tri par tas

Algorithme :

- On créé un arbre binaire à partir du tableau d’éléments à trier
- Tant que l’arbre contient plus de 1 nœud :
  - On tamise l’arbre et on reporte les inversions de nœud par des inversions de valeurs dans le tableau
  - On inverse le dernier et premier éléments du tableau
  - On supprime de l’arbre le dernier élément du tableau

Complexité : O(nlog(n))

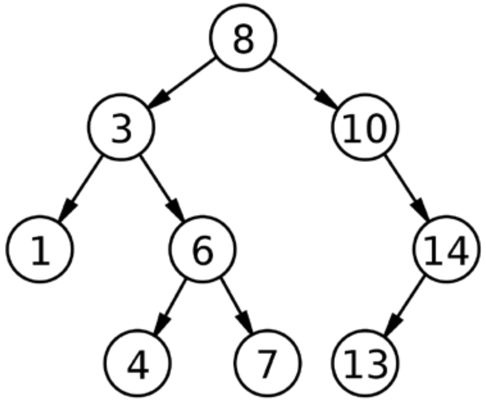


->TD2

6. Arbre binaire de recherche

Un arbre binaire de recherche est un arbre binaire qui respecte la propriété suivante :

- Chaque nœud gauche a une étiquette inférieure à l’étiquette de son nœud parent
- Chaque nœud droit a une étiquette supérieure à l’étiquette de son nœud parent

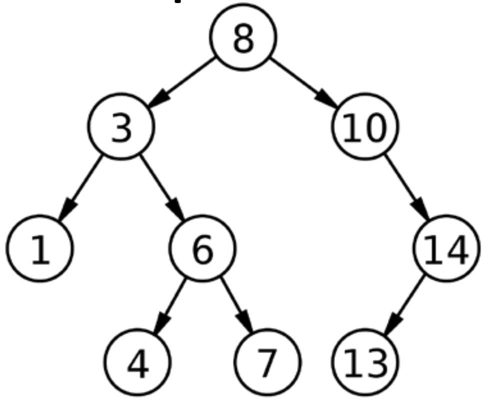


6. Arbre binaire de recherche

Propriétés intéressantes :

- L’insertion et la recherche d’une clef dans le tableau a une complexité en O(log(n)) en moyenne et en O(n) au maximum
- La suppression d’un élément est en O(n) au maximum

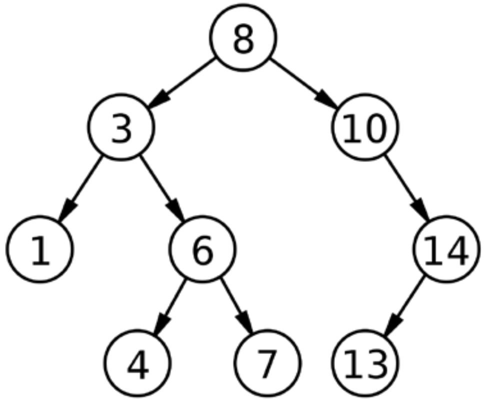
Les complexités sont réduite lorsque les arbres sont équilibrés.



6. Arbre binaire de recherche

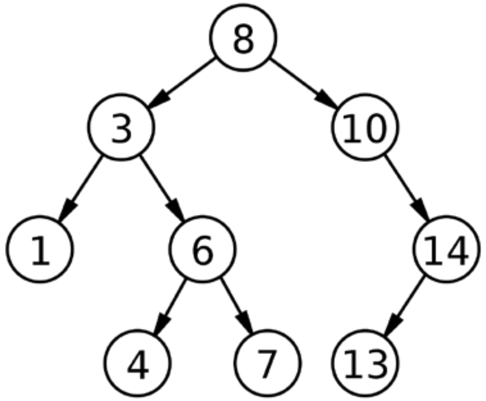
- Recherche de l'élément 7
  - On se place à la racine. 7 < 8 donc on se rend au fils gauche
  - 3 < 7 donc on se déplace vers le fils droit
  - 6 < 7 donc on se déplace vers le fil droit
  - On a trouvé l'élément

En cas de recherche d'un élément non présent dans l'arbre de recherche, on se retrouve sur une feuille de l'arbre sans avoir trouvé la valeur.



6. Arbre binaire de recherche

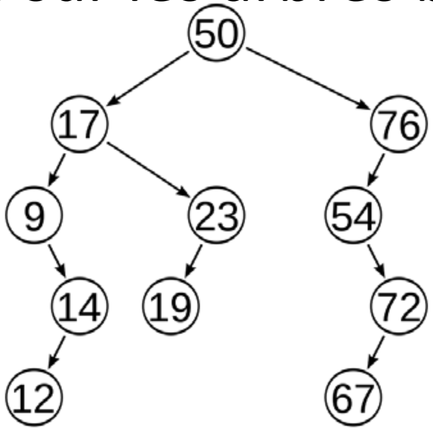
- Insertion d’un élément
  - On effectue tout d’abord une recherche de la clef associé à l’élément
  - Lorsque l’on arrive à une feuille :
    - Si l’étiquette de l’élément inséré est plus grand que l’étiquette de la feuille, on insère l’élément à droite
    - Sinon on insère l’élément à gauche

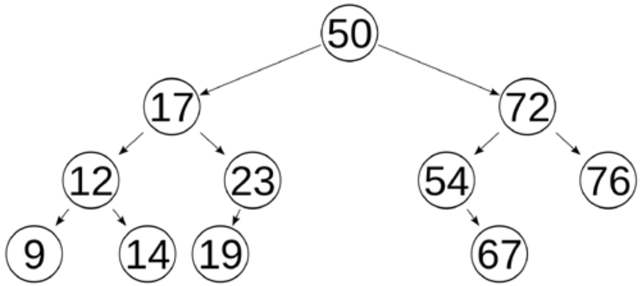


->TD3

7. Arbre équilibré

- Arbre dont les profondeurs de l’ensemble des feuilles diffère de maximum 1 et est minimisé
- Eviter de construire des « arbres peignes »
- Permet de réduire le nombre de déplacements nécessaire sur les nœuds lors d’opération sur un arbre
- Exemple d’arbre binaire de recherche dégénéré / équilibré
- Surtout utilisé sur les arbres binaires



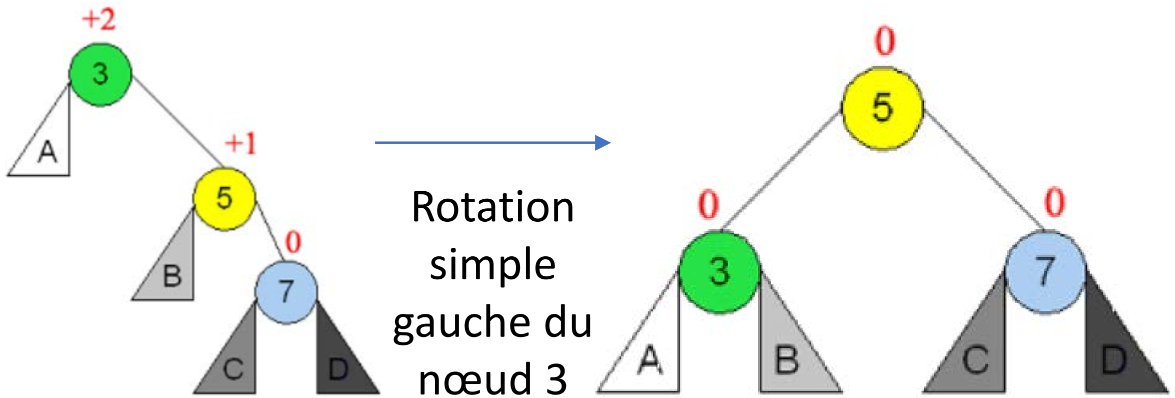


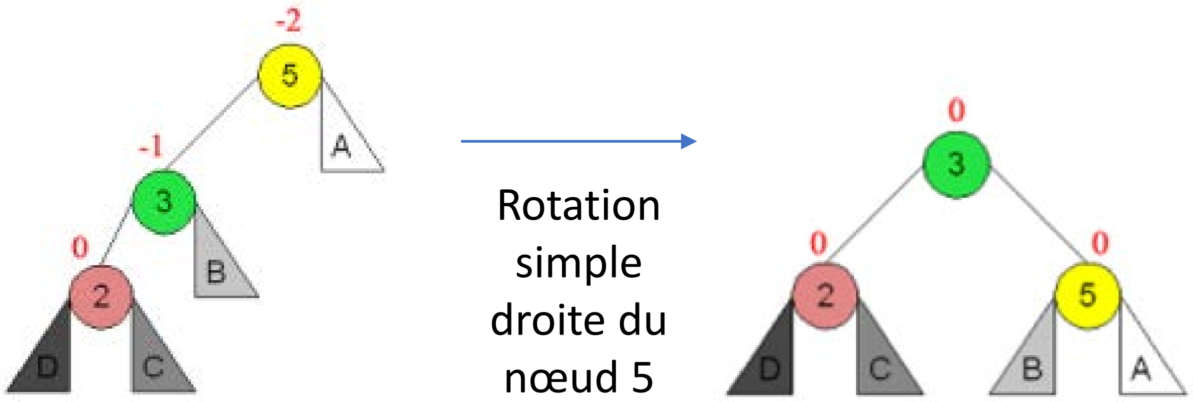
7. Arbre équilibré

- Facteur d’équilibrage d’un nœud : différence de hauteur entre le sous-arbres droit et le sous-arbre gauche du nœud considéré
- En calculant le facteur d’équilibrage d’un nœud, on peux effectuer des rotations simple droite ou des rotations simples gauches pour équilibrer les sous-arbres de ce nœud.

7. Arbre équilibré - Rotations

- Rotation simple gauche : le nœud devient le fils gauche de son fils droit
- Rotation simple droite : le nœud devient le fils droit de son fils gauche
- Ces rotations transforment un ABR en un autre ABR





->TD4

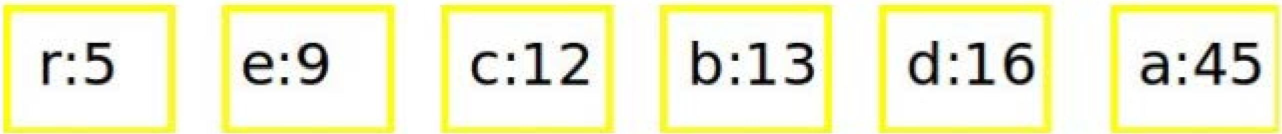
8. Algorithme d’Huffman

- Algorithme de compression de données sans perte
- Utilise à code à longueur variable :
  - Les symboles les plus fréquent ont les codes avec les longueurs les plus courtes
  - L’algorithme assure que le signal encodé soit déchiffrable
- La constitution du codage des symboles est effectués à l’aide d’un arbre binaire
- Des implémentations permettent de se rapprocher de manière très proche de l’entropie de la source d’information. La compression mis en œuvre peut donc être très efficace.
- Limitation de ce système de codage :
  - Le codage des symboles doit être transmit avec le message
  - Il faut établir une analyse fréquentielle de l’intégralité du message à coder pour constituer l’arbre de codage des symboles
  - Des implémentations adaptatives existent pour ne pas faire une analyse complète d’un message ou compresser des données obtenu en temps réel (streaming). Cela implique une mise à jour régulière de l’arbre de codage et donc de lourd calcul.

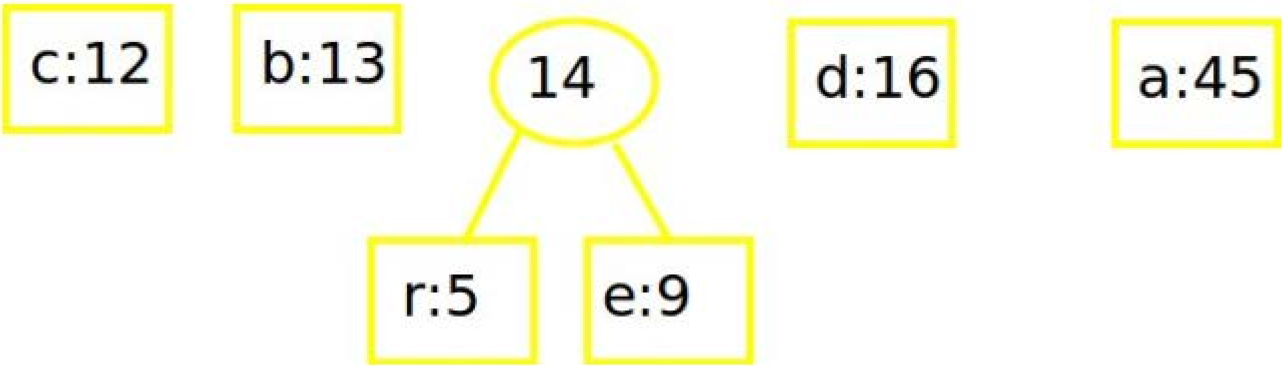
8. Algorithme d'Huffman – mise en œuvre

- Faire une analyse fréquentielle des symboles du message à compresser
- Etablir une liste de fréquence / symbole
- On construit une arbre binaire de recherche avec l'algorithme suivant:
    - On prend les 2 nœuds de la liste courante avec les plus petites étiquettes
    - On les places tout les deux sous un nouveau nœud parent avec une étiquette égale à la somme des étiquettes des nœuds sélectionnés.
    - On supprimes ces 2 nœuds de la liste courante et on y insère le nœud parent ainsi constitué
- On arrête l'algorithme quand il n'y a plus de nœud à placer dans la liste courante
- Par convention, on place un zéro sur la branche gauche de chaque nœud et un 1 sur la branche droite de chaque nœud
- Le codage de chaque symbole est obtenu en concaténant les valeurs traversé sur chaque branche en effectuant le chemin entre la racine de l'arbre et le nœud correspondant à chaque symbole

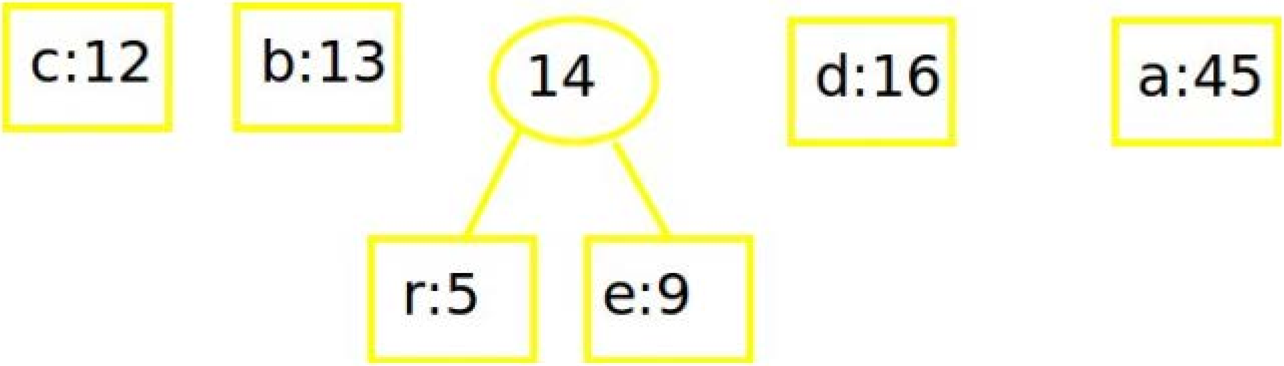
8. Algorithme d’Huffman – exemple



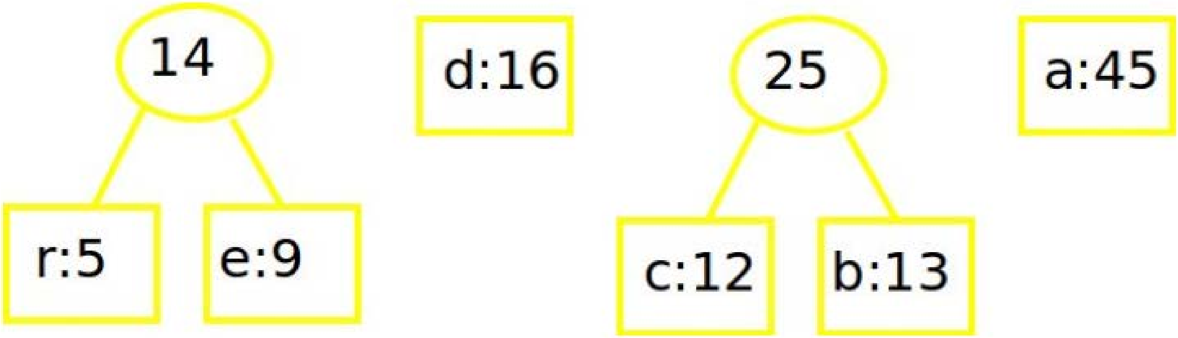
r et e sont les 2 symboles avec la plus petite étiquette



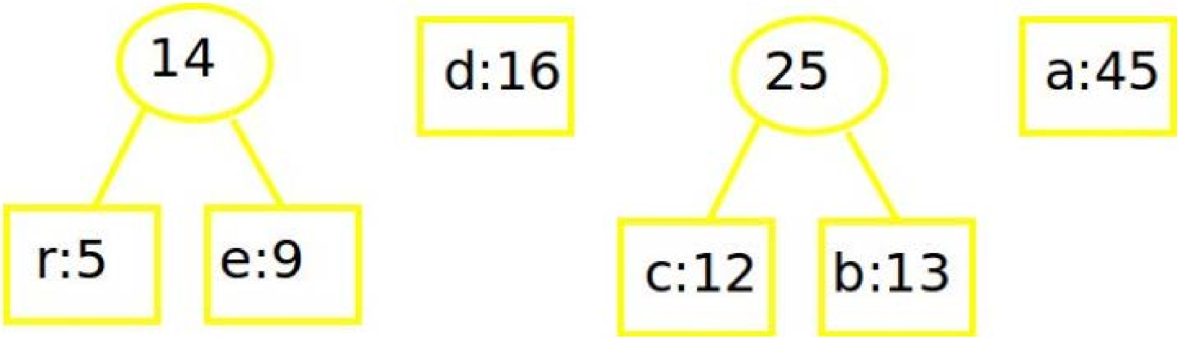
8. Algorithme d’Huffman – exemple



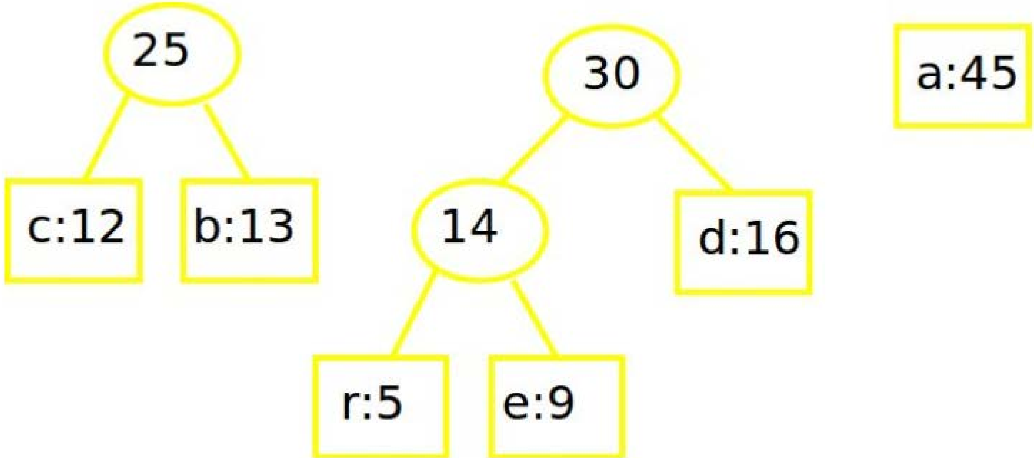
c et b sont les 2 symboles avec la plus petite étiquette



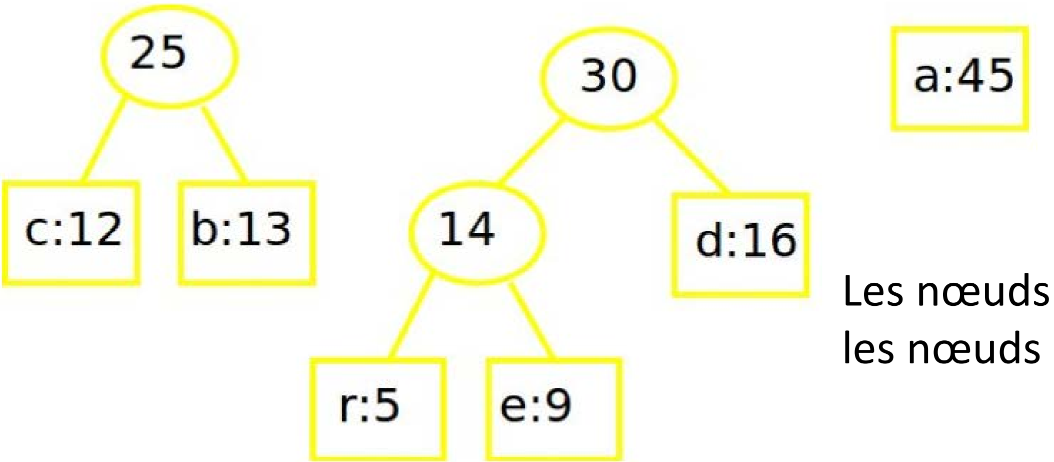
8. Algorithme d’Huffman – exemple



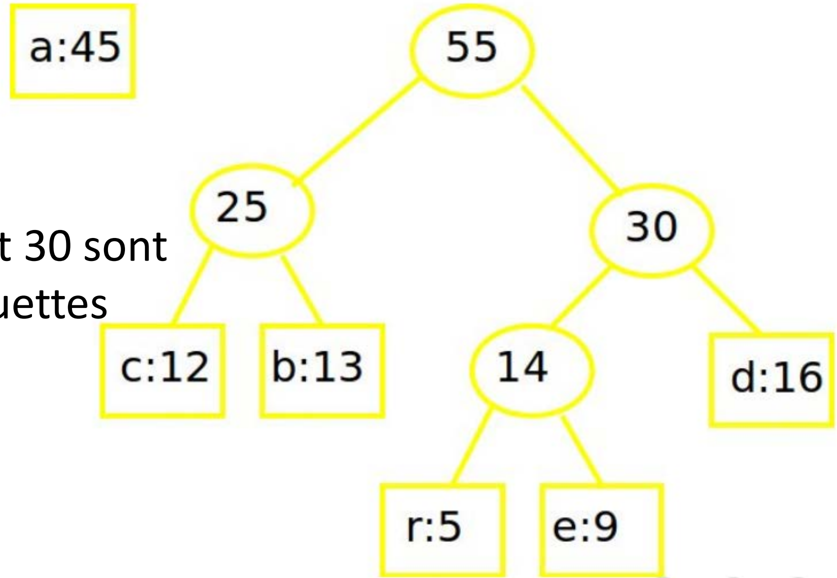
Le nœud avec pour étiquette 14 et le symbole d ont les plus petites étiquette



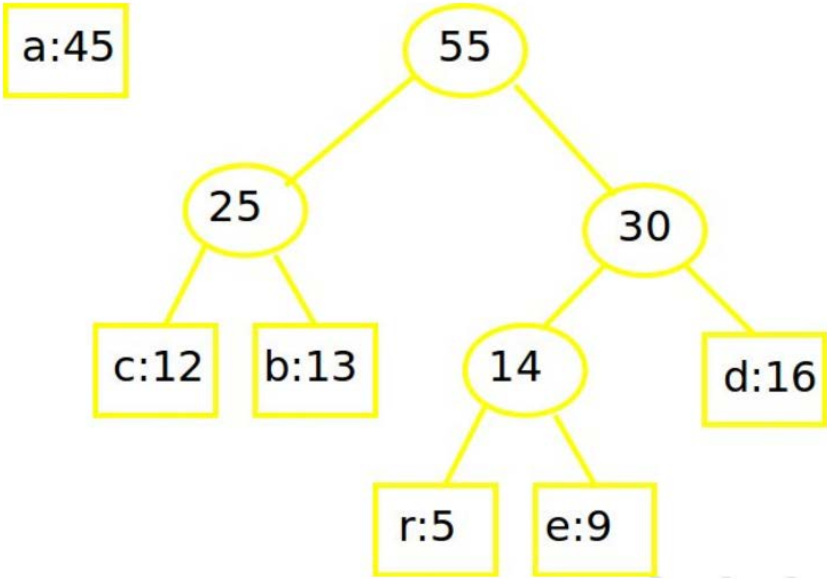
8. Algorithme d’Huffman – exemple



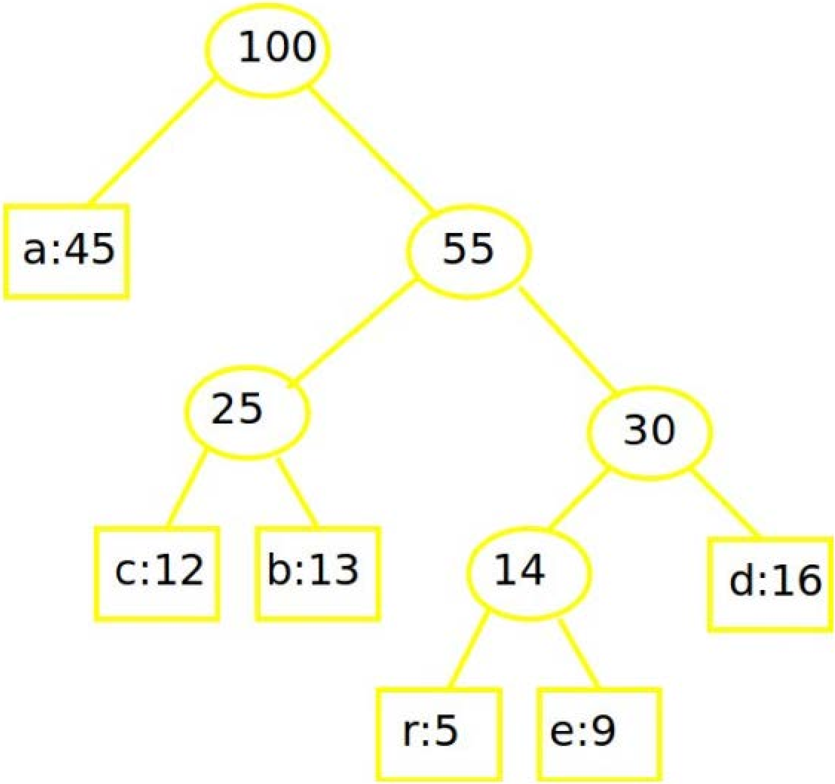
Les nœuds avec pour étiquette 25 et 30 sont
les nœuds avec les plus petites étiquettes



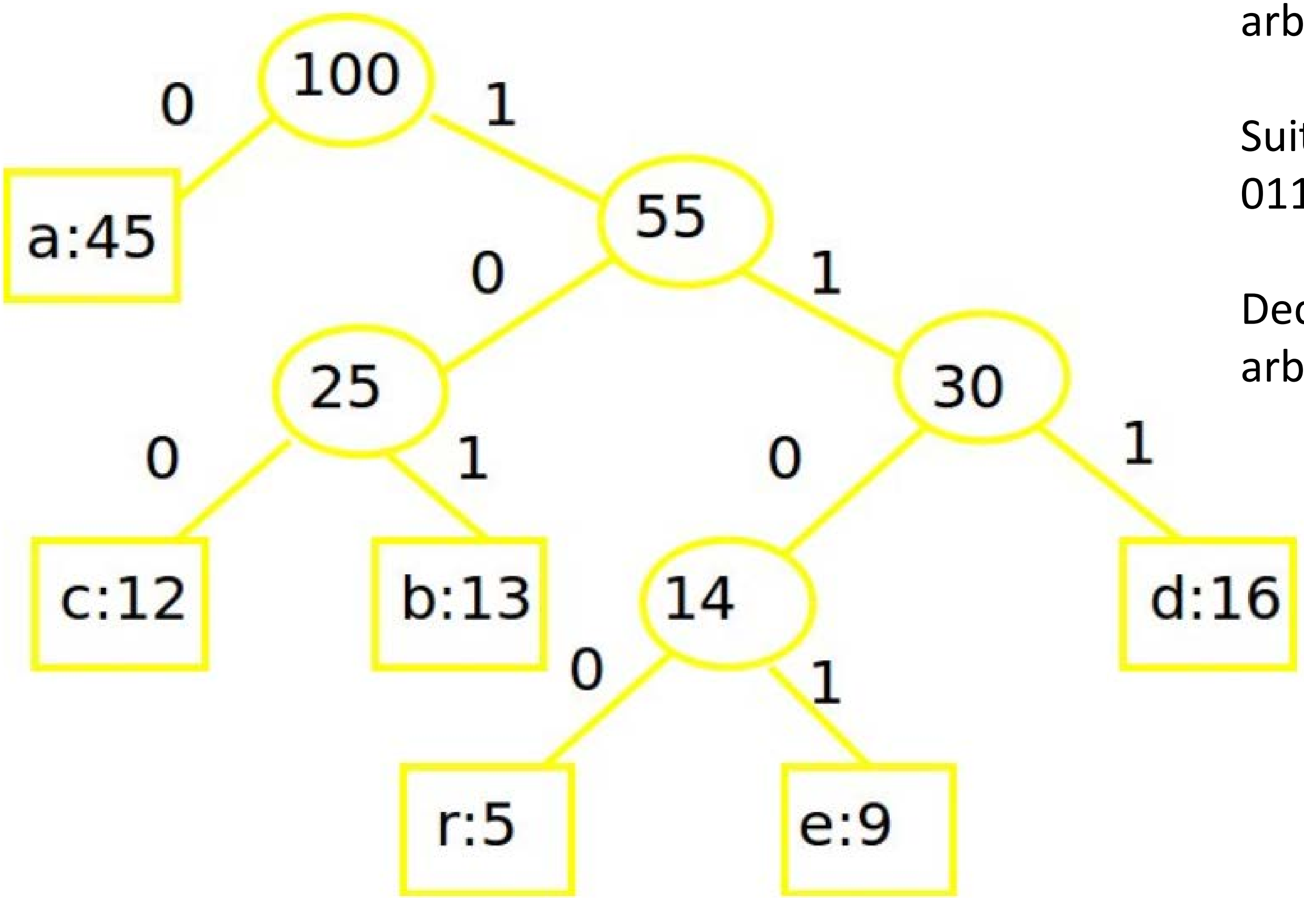
8. Algorithme d’Huffman – exemple



Le nœud avec l’étiquette 55 et le nœud « a » sont les nœuds avec les plus petites étiquettes (et les derniers)



8. Algorithme d'Huffman – exemple



arbre

Suite de 5 entiers codé sur 8bits => 40 bits
0110010111001101

Dechiffrage :
arbre

Symbole	Code
a	0
b	101
c	100
d	111
e	1101
r	1100

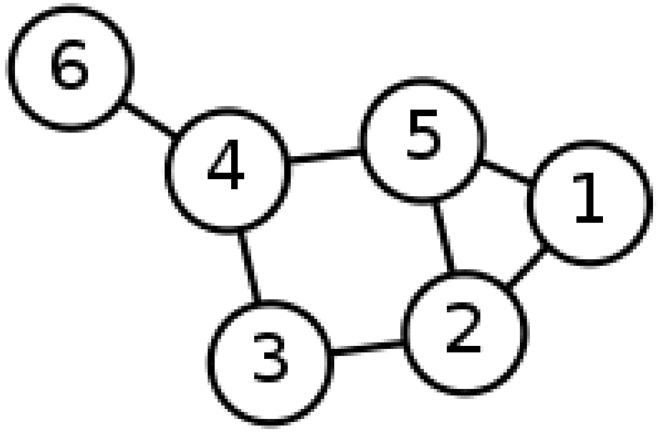
->TD5

Partie 2

Les Graphes

1. Graphe – Définitions et représentations

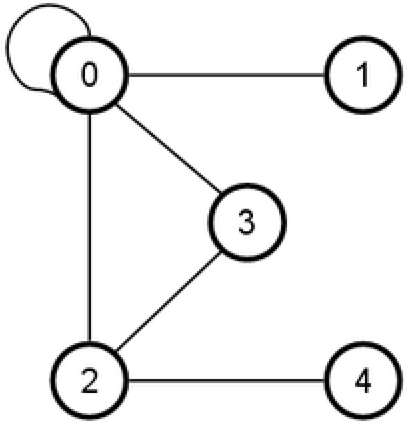
- Un graphe est un ensemble composé de 2 type d’éléments :
  - Un ensemble de sommet
  - Un ensemble d’arrêtes constituée de paire de sommets



Ensemble des sommets : {1,2,3,4,5,6}
Ensemble des arrêtes : {(1,2), (1,5), (2,5), (2,3), (3,4), (4,5),(4,6)}

1. Graphe – Définitions et représentations

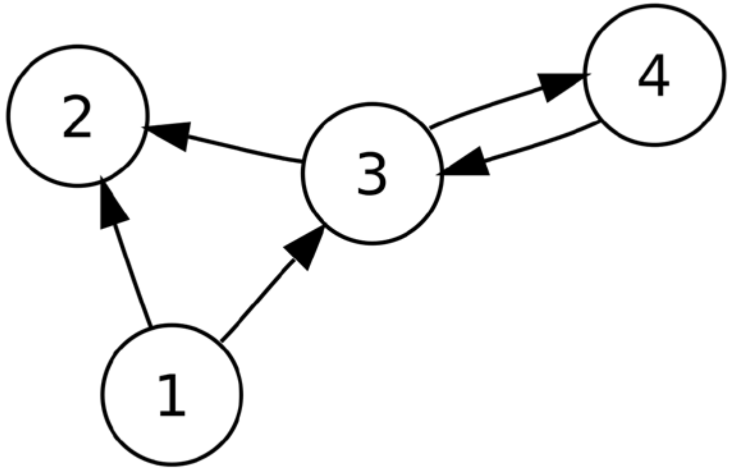
- Un autre exemple



Ensemble des sommets : {0,1,2,3,4}
Ensemble des arrêtes : {(0,0), (0,1), (0,2), (0,3), (2,3), (2,4)}

1. Graphe – Définitions et représentations

- Graphes orientés

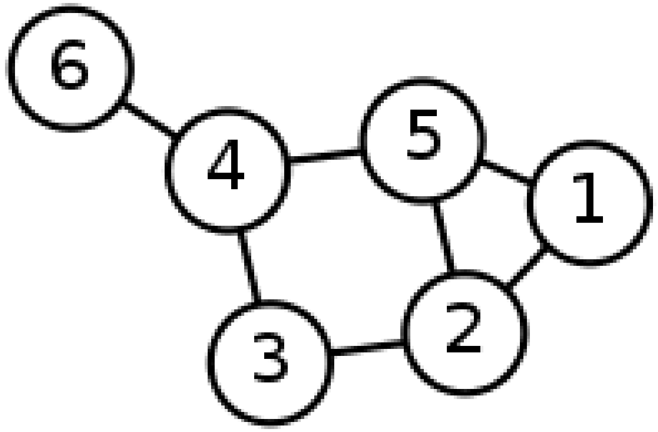


En se déplaçant de sommet en sommet le long des arrêtes, celles-ci ne peuvent être emprunté que dans un seul sens

Ensemble des sommets : {1,2,3,4}
Ensemble des arrêtes : {(1,2), (1,3), (3,2), (3,4), (4,3)}

1. Graphe – Définitions et représentations

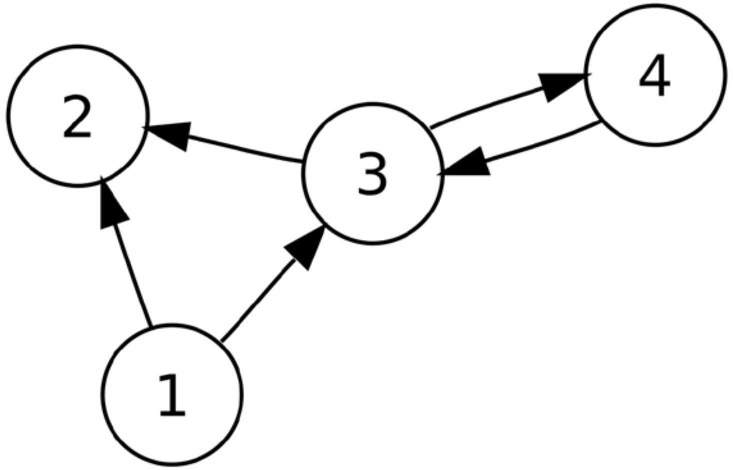
- Représentation par liste d'adjacence



Représentation par liste d'adjacence	
Sommet	Sommets adjacents
1	2,5
2	1,3,5
3	2,4
4	3,5,6
5	1,2,4
6	4

1. Graphe – Définitions et représentations

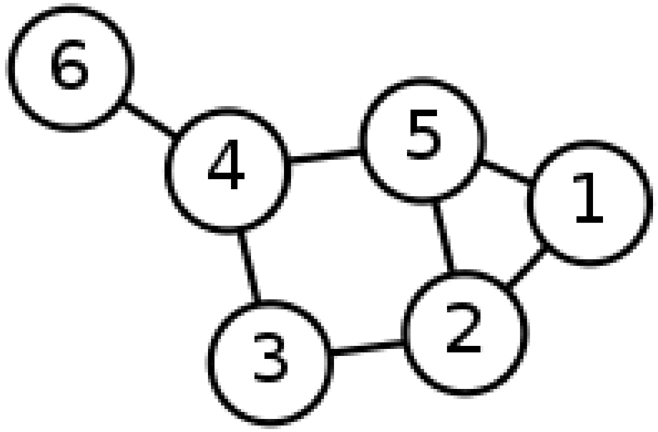
- Représentation par liste d'adjacence – graphe orienté



Représentation par liste d'adjacence	
Sommet	Sommets adjacents
1	2,3
2	
3	2,4
4	3

1. Graphe – Définitions et représentations

- Matrice d'adjacence
- Matrice symétrique (car non orienté)



Sommet d'arrivée

Sommet de départ

$$
\begin{array}{c}
 \\
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6
\end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$

1. Graphe – Définitions et représentations

- Matrice d’adjacence

Représentation par liste d’adjacence	
Sommet	Sommets adjacents
1	2,5
2	1,3,5
3	2,4
4	3,5,6
5	1,2,4
6	4

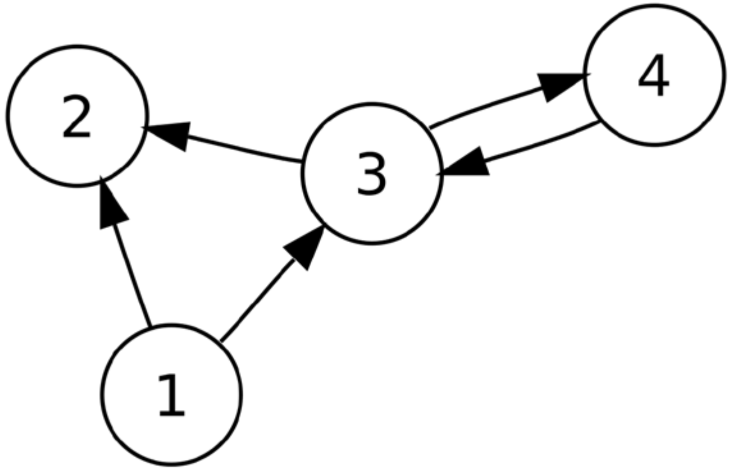
Sommet d’arrivée

Sommet de départ

$$\begin{array}{cc} & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

1. Graphe – Définitions et représentations

- Matrice d’adjacence – graphe orienté



Sommet d’arrivée

Sommet de départ

$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 0 \\
2 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 1 \\
4 & 0 & 0 & 1 & 0
\end{array}
$$

1. Graphe – Définitions et représentations

- Matrice d'adjacence

Représentation par liste d'adjacence	
Sommet	Sommets atteignable
1	2,3
2	
3	2,4
4	3

Sommet d'arrivée

Sommet de départ

$$\begin{array}{cc} & \begin{array}{cccc}1 & 2 & 3 & 4\end{array} \\ \begin{array}{c}1\\2\\3\\4\end{array} & \begin{pmatrix}0 & 1 & 1 & 0\\0 & 0 & 0 & 0\\0 & 1 & 0 & 1\\0 & 0 & 1 & 0\end{pmatrix}\end{array}$$

1. Graphe – Définitions et représentations

- Degrés d’un sommet : nombre de sommets atteignable pour ce sommet
- Chemin entre 2 sommet : ensemble d’arrêtes permettant de se rendre du sommet de départ au sommet d’arrivé
- Longueur d’un chemin : nombre d’arrêtes contenu dans ce chemin

1. Graphe – exemple implémentation

```
Class Graph {
        int id;
        Graph[] destinations;
}
```

->TD6

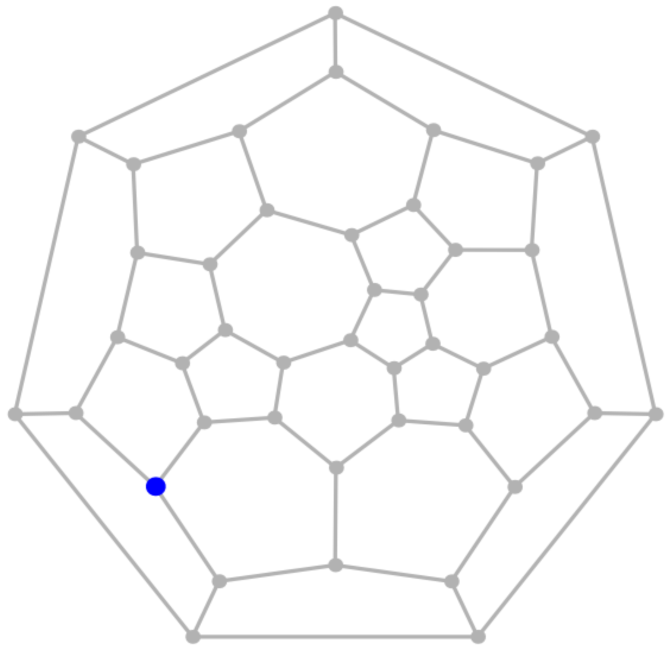
2. Graphe – Parcours en largeur

Principe de fonctionnement :

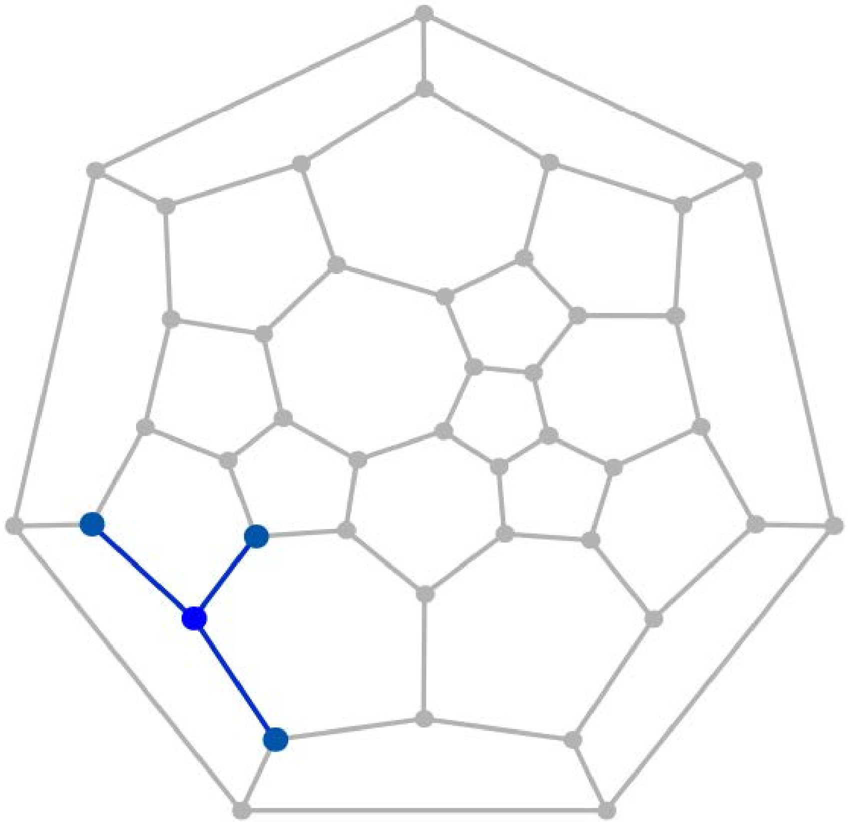
Un sommet source émet un message à tous les sommets qui lui sont accessible. Dès qu’un sommet reçoit un message, il l’envoi à tous ses voisins qui ne l’ont pas encore reçu.

Dès qu’un sommet reçoit le message, on l’extrait pour obtenir une liste de sommet découvert par un parcours en largeur.

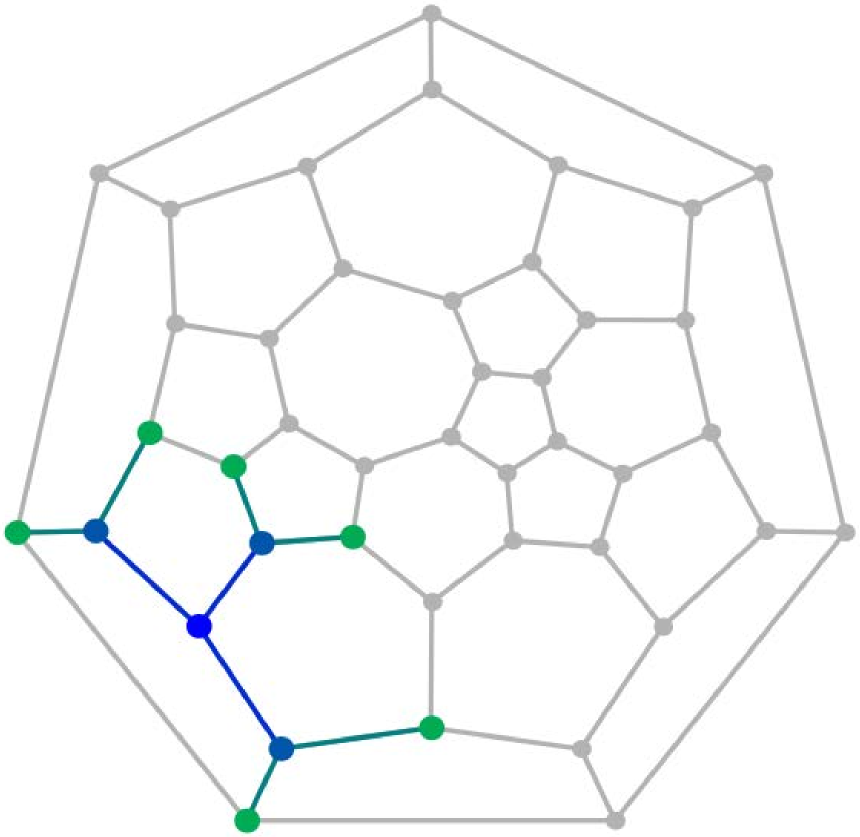
2. Graphe – Parcours en largeur



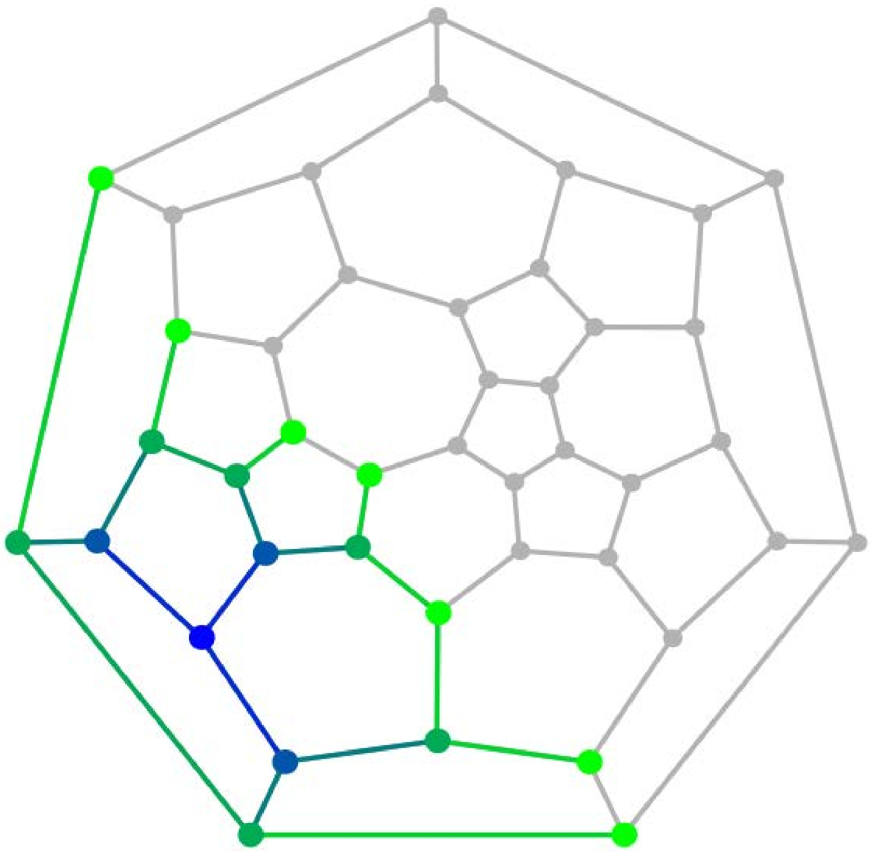
2. Graphe – Parcours en largeur



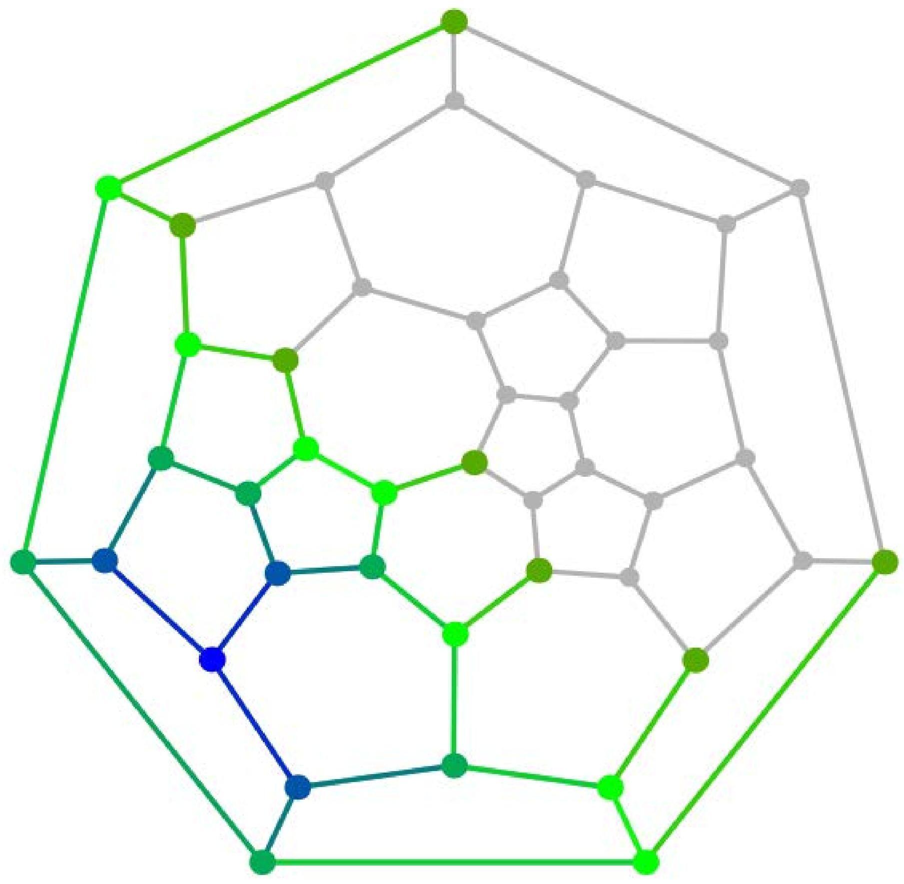
2. Graphe – Parcours en largeur



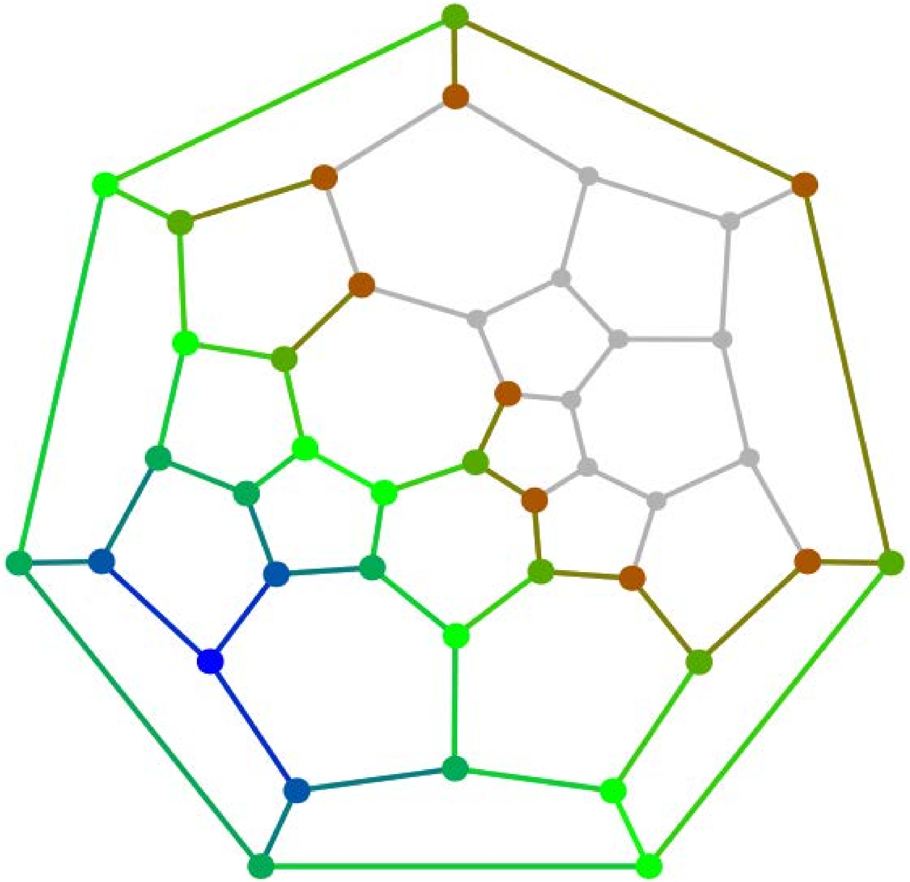
2. Graphe – Parcours en largeur



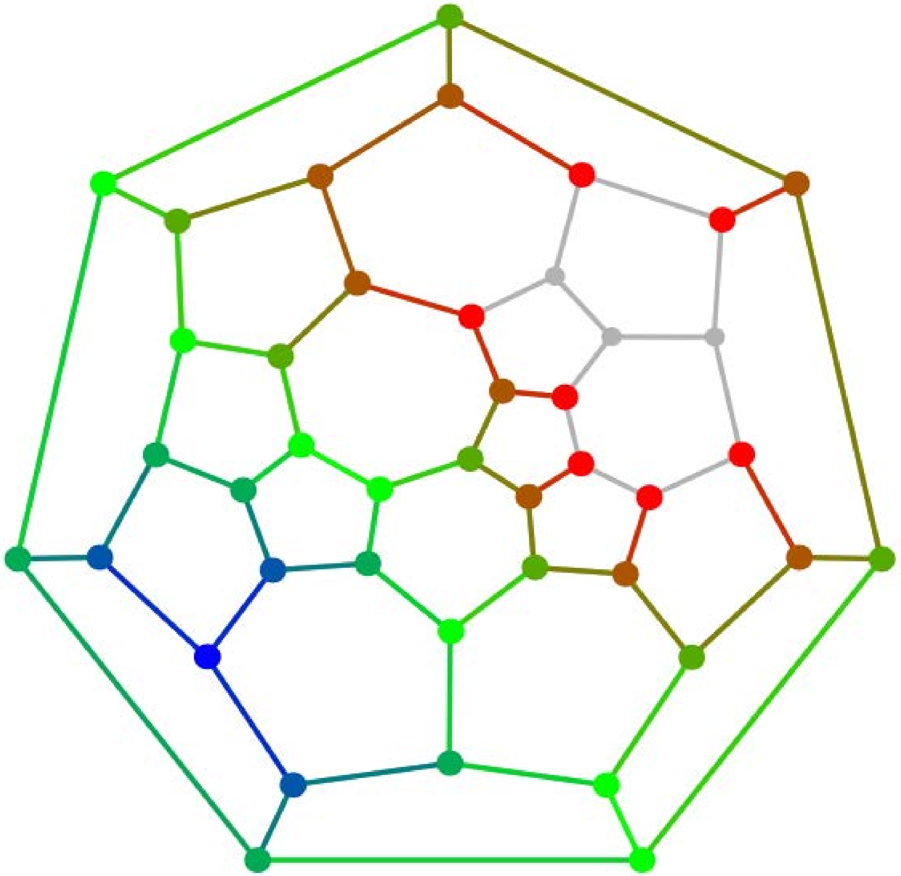
2. Graphe – Parcours en largeur



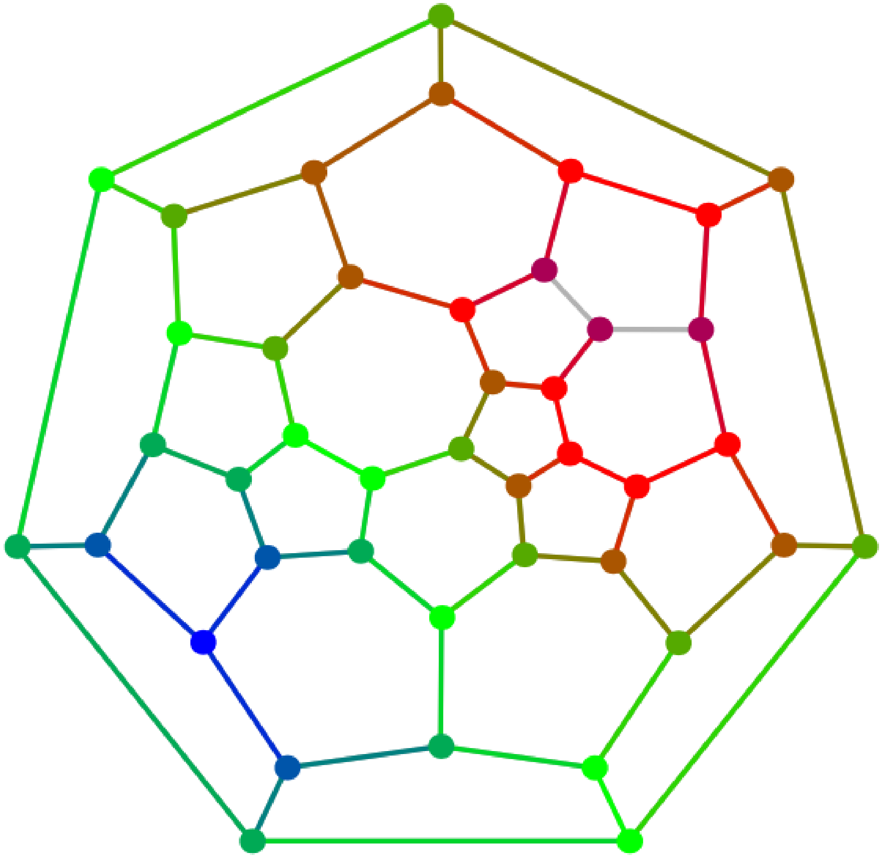
2. Graphe – Parcours en largeur



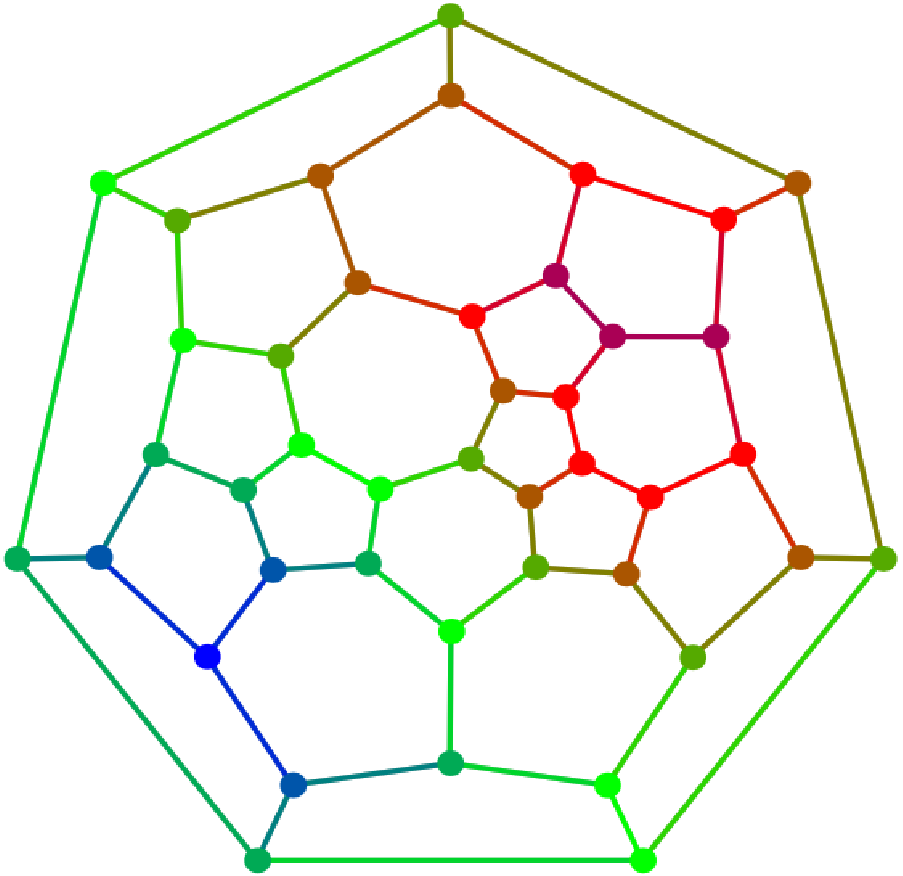
2. Graphe – Parcours en largeur



2. Graphe – Parcours en largeur



2. Graphe – Parcours en largeur



2. Graphe – Parcours en largeur

L'implémentation est presque identique au parcours en largeur d'un arbre. Une simple gestion des boucles que peux présenter un graphe est gérer pour éviter de parcourir 2 fois un même sommet

```
function breadthFirstSearch(Graph T) {
		Queue currentPoints = new Queue();
		 currentPoints.push(T); //We add an element to the queue, sure to do it by reference
		 Graph[] result = []; //We create a dynamic list of elements
		While(!Queue.isEmpty()) {
				Tree currentPoint = currentPoints.pop(); //We make one item go out from the queue, sure to do it
				                                         //by reference
				if(!currentPoint.alreadyDone) {
						result.push(currentPoint);
						foreach(currentPoint.destinations as point) {
								currentPoints.push(point);
						}
						currentPoint. alreadyDone = true;
				}
}
```

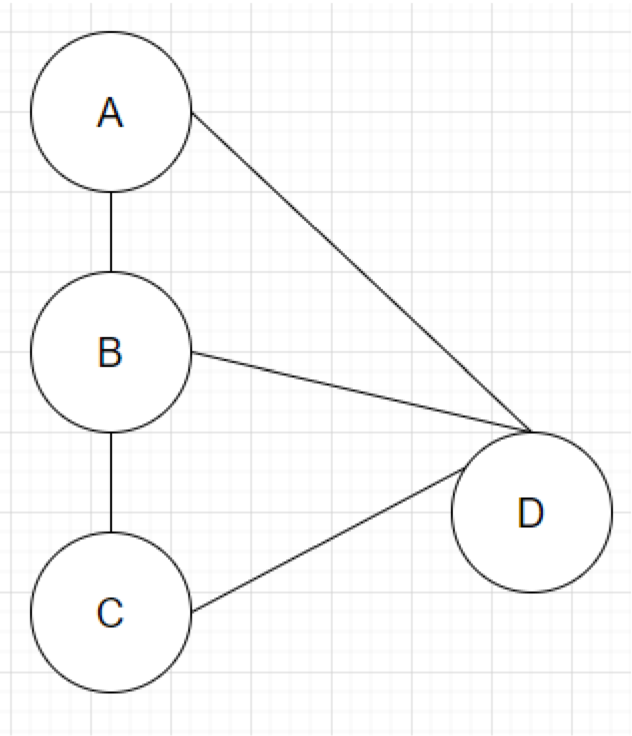
3. Graphe – Parcours en profondeur - Principe

- Si le sommet courant est marqué comme visité, on ne fais rien et on quitte ce traitement
- Si le sommet courant n’est pas marqué comme visité, on l’extrait et on le marque comme visité
- On se rend auprès de chaque sommet accessible depuis le sommet courant en y effectuant ce traitement

3. Graphe – Parcours en profondeur - Exemple

On dispose d’un graphe avec les listes d’adjacence suivante :

A : B, D
B: A, C, D
C: B, D
D : A, B, C



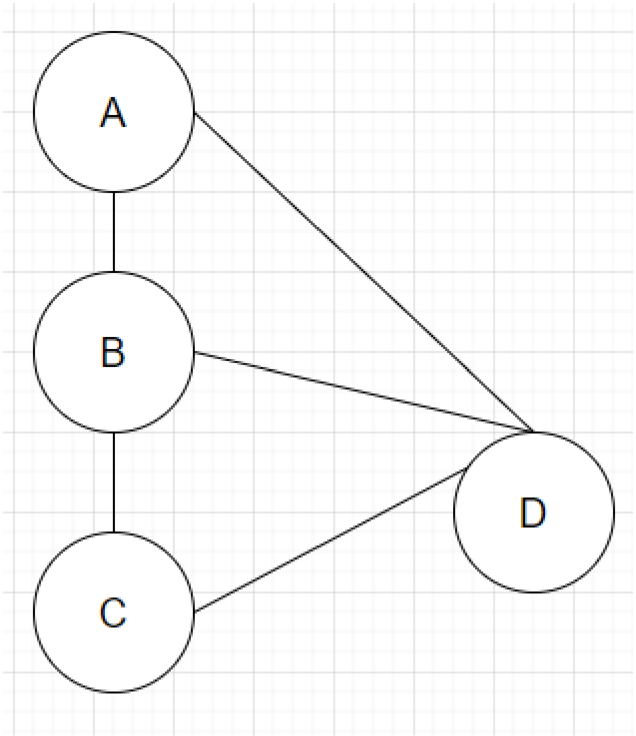
On va effectuer le parcours en profondeur à partir de A

3. Graphe – Parcours en profondeur - Exemple

Nœud courant : A
Nœuds en attente de fin d'exécution de l'algorithme :
Nœuds déjà visités :
Résultats :

Nœud courant : A
Nœuds en attente de fin d'exécution de l'algorithme : A
Nœuds déjà visités : A
Résultats : A

On va aux destinations disponible de A

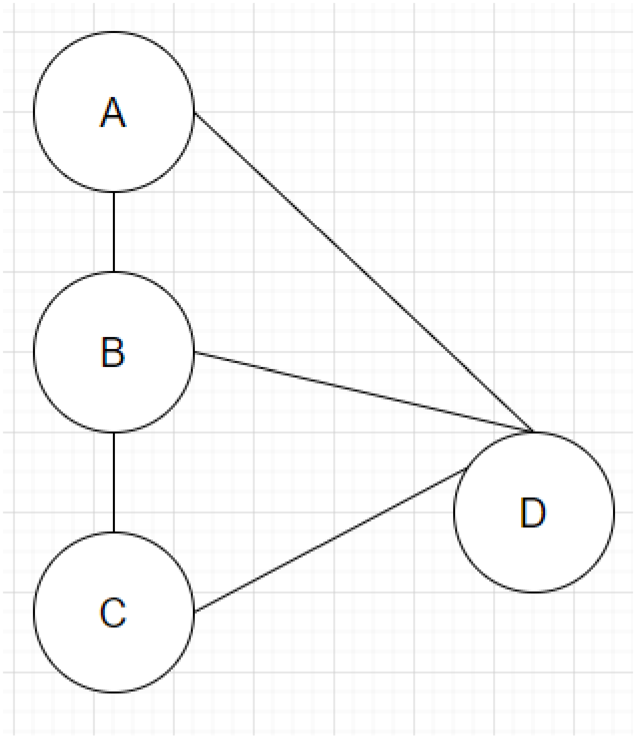


3. Graphe – Parcours en profondeur - Exemple

Nœud courant : B
Nœuds en attente de fin d’exécution de l’algorithme : A
Nœuds déjà visités : A
Résultats : A

Nœud courant : B
Nœuds en attente de fin d’exécution de l’algorithme : A, B
Nœuds déjà visités : A, B
Résultats : A,B

On va aux destinations disponible de B

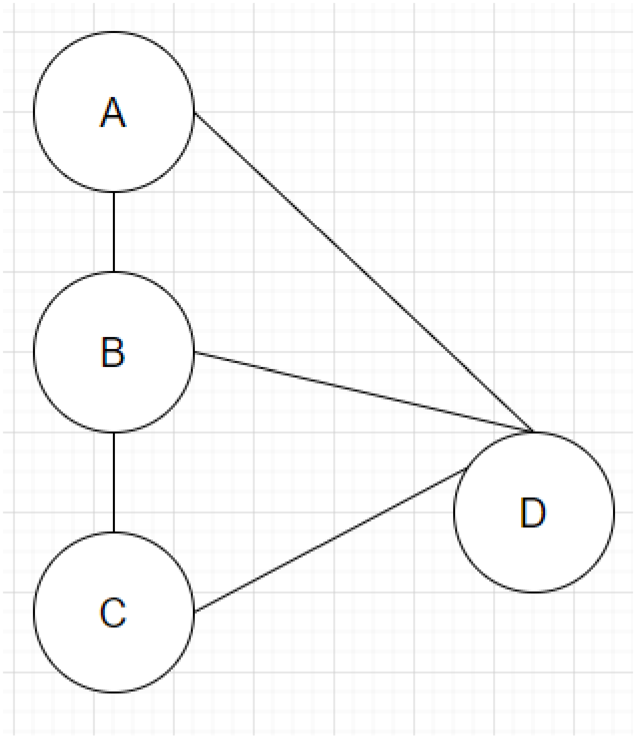


3. Graphe – Parcours en profondeur - Exemple

Nœud courant : C
Nœuds en attente de fin d’exécution de l’algorithme : A, B
Nœuds déjà visités : A, B
Résultats : A,B

Nœud courant : C
Nœuds en attente de fin d’exécution de l’algorithme : A, B, C
Nœuds déjà visités : A, B, C
Résultats : A,B, C

On va aux destinations disponible de C

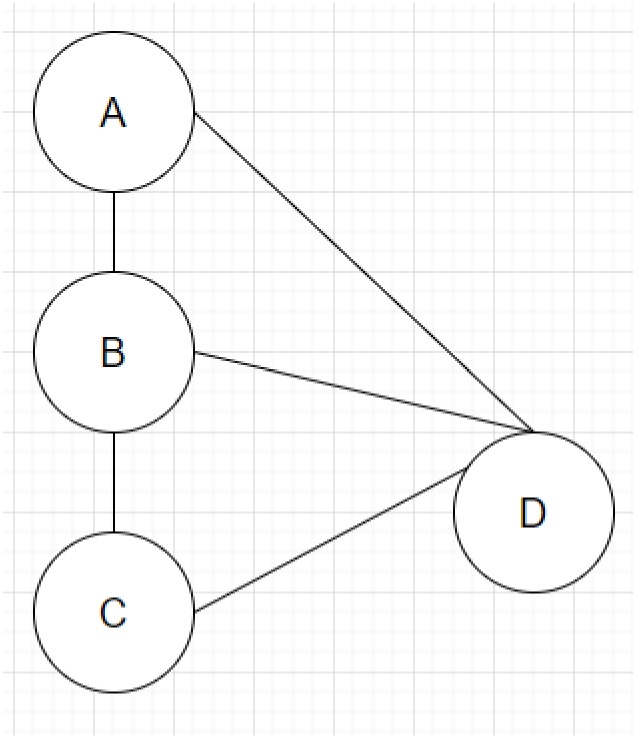


3. Graphe – Parcours en profondeur - Exemple

Nœud courant : D
Nœuds en attente de fin d’exécution de l’algorithme : A, B, C
Nœuds déjà visités : A, B, C
Résultats : A,B, C

Nœud courant : D
Nœuds en attente de fin d’exécution de l’algorithme : A, B, C, D
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

On va aux destinations disponible de D

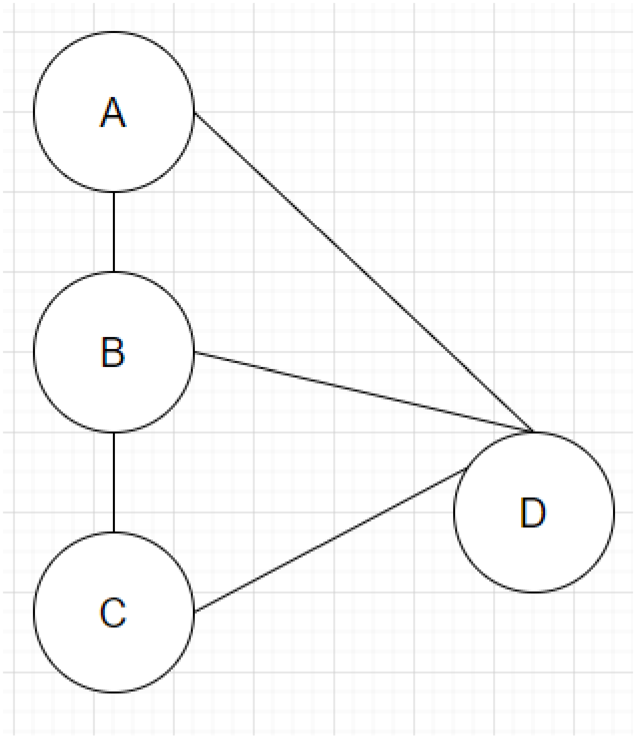


3. Graphe – Parcours en profondeur - Exemple

Nœud courant : A
Nœuds en attente de fin d'exécution de l'algorithme : A, B, C, D
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

Nœud courant : B
Nœuds en attente de fin d'exécution de l'algorithme : A, B, C, D
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

Nœud courant : C
Nœuds en attente de fin d'exécution de l'algorithme : A, B, C, D
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

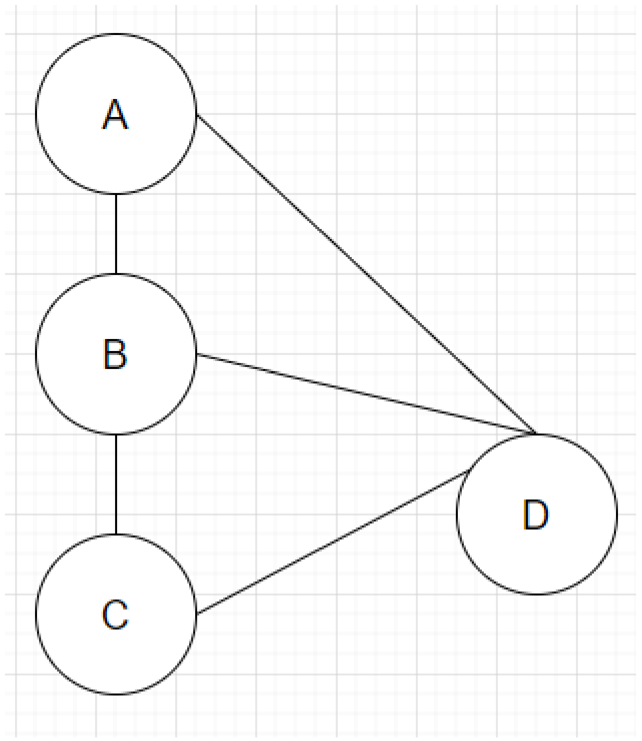


On vient de terminer l'exécution de l'algorithme sur le sommet D, on retourne à l'exécution sur le sommet C

3. Graphe – Parcours en profondeur - Exemple

Nœud courant : B
Nœuds en attente de fin d'exécution de l'algorithme : A, B, C
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

On vient de terminer l'exécution de l'algorithme sur le sommet C, on retourne à l'exécution sur le sommet B

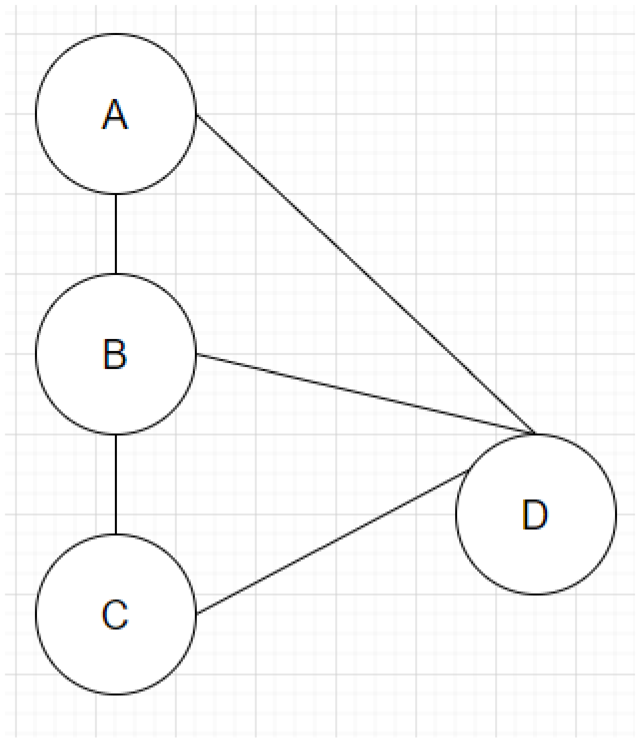


3. Graphe – Parcours en profondeur - Exemple

Nœud courant : A
Nœuds en attente de fin d’exécution de l’algorithme : A, B
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

Nœud courant : D
Nœuds en attente de fin d’exécution de l’algorithme : A, B
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

On vient de terminer l’exécution de l’algorithme sur le sommet B, on retourne à l’exécution sur le sommet A

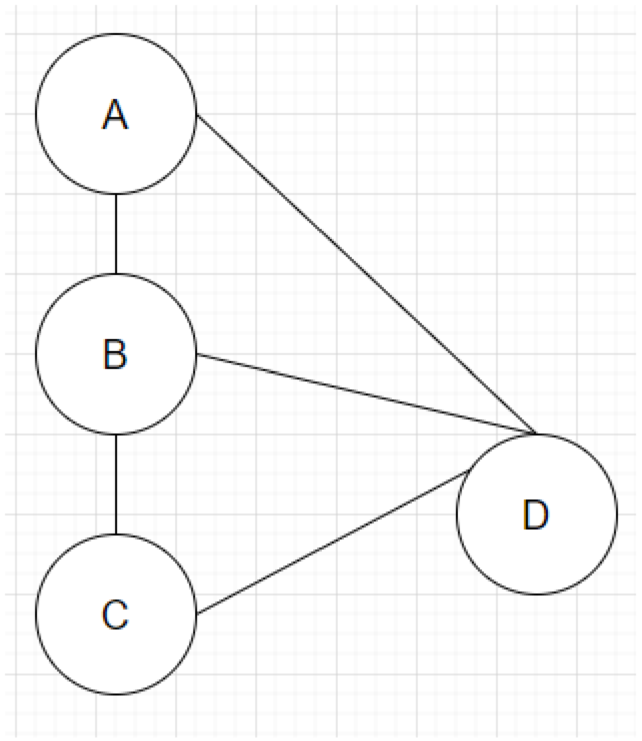


3. Graphe – Parcours en profondeur - Exemple

Nœud courant : D
Nœuds en attente de fin d'exécution de l'algorithme : A
Nœuds déjà visités : A, B, C, D
Résultats : A,B, C, D

On vient de terminer l'exécution de l'algorithme sur le sommet A

Il n'y a plus d'exécution en attente, et la dernière exécution récursive vient de s'achever. Le parcours en profondeur est achevé



3. Graphe – Parcours en profondeur

L'implémentation est presque identique au parcours en profondeur d'un arbre. Une simple gestion des boucles que peux présenter un graphe est gérer pour éviter de parcourir 2 fois un même sommet

```
Function depthFirstSearch(Graph T, Graph[] *result ) {
        If(!T.alreadyDone) {
                result.push(T);
                T.alreadyDone = true;
                foreach(T.destinations as point) {
                        depthFirstSearch(point, result);
                }
        return result;
        }
}
```

->TD7

4. Fermeture transitive et symétrique

<u>Fermeture transitive :</u> A partir du graphe original, c’est une opération consistant à ajouter des arrêtes entre chaque sommet entre lesquels il existe un chemin.

→ On ajoute des arrêtes permettant de se rendre directement aux sommets qui auraient pu être accessible à travers un voyage au travers de plusieurs sommet

4. Fermeture transitive et symétrique

<u>Propriété mathématique sur la matrice d'adjacence</u>

- M représente les chemins possibles existants entre 2 sommets de longueur 1
- M^2 représente les chemins possibles existants entre 2 sommets de longueur 2
- M^3 représente les chemins possibles existant entre 2 sommets de longueur 3

…Etc

→ M^k représente les chemins possibles existant entre 2 sommets de longueur k

Sur la matrice M^k, chaque valeur de $M^k_{i,j}$ représente le nombre de chemin contenant k arrêtes du sommet i au sommet j

4. Fermeture transitive et symétrique

Pour un graphe représenté par la matrice carré M de taille n, la fermeture transitive de ce graphe est donc représenté par la matrice suivante :

$$\sum_{\mathrm{p}=1}^{n-1} M^p$$

qu'il suffit donc de calculer

Attention : La somme mise en œuvre est un somme logique. Chaque valeur final de l'addition ne peux être égal qu'à 0 (faux) ou 1 (vrai)

4. Fermeture transitive et symétrique

On peux également effectuer, depuis chaque nœud du graphe :

- Un parcours en largeur
- Un parcours en profondeur

Et créer les arrêtes manquantes entre le sommet de départ et tout les sommets découverts au travers de ces parcours

→ Cela revient à créer la liste d’adjacence du graphe représentant la fermeture transitive du graphe original

4. Fermeture transitive et symétrique

Graphe symétrique : Pour tout arrête permettant d'aller d'un sommet A à un sommet B, il existe une arrête permettant d'aller du sommet B au sommet A

Fermeture symétrique : Opération consistant à ajouter des arrêtes afin de rendre un graphe symétrique

→ S'applique uniquement à un graphe orienté
→ Transforme un graphe orienté en un graphe non orienté

4. Fermeture transitive et symétrique

Fermeture symétrique : Opération consistant à ajouter des arrêtes afin de rendre un graphe symétrique

→ Peux s'effectuer de manière aisée avec un calcul sur la matrice d'adjacence que l'on note M de taille n
→ En notant ${}^{t}M$ la matrice transposé de M, la matrice d'adjacence R du graphe correspondant à la fermeture symétrique du graphe originale est constitué en effectuant une opération terme à terme sur les matrice M et ${}^{t}M$ de la manière suivante :

$$R = M + {}^{t}M$$

Attention : la somme ici présenté est une somme logique des éléments

->TD8

5. Chemin et existence de chemin

Objectif : déterminer l'existence et proposer un chemin chemin entre 2 sommets à l'intérieur d'un graphe.

Pour déterminer l'existence d'un chemin entre 2 sommets, on peut :

- Réaliser la fermeture transitive du graphe et en vérifier la présence d'une arrête entre le sommet de départ et le sommet d'arrivée dans cette dernière
- Faire un parcours de graphe en partant du sommet de départ à la recherche du sommet d'arrivée

5. Chemin et existence de chemin

<u>Objectif :</u> déterminer le chemin entre 2 sommets

Pour déterminer le chemin entre 2 sommets, la méthode la plus aisée est d’effectuer en parcours du graphe en enregistrant, pour chaque sommet découvert, l’arrête par laquelle nous avons découvert ce dernier.

-> Une fois le sommet d’arrivée trouvée, cela permet de remonter le chemin effectuée jusqu’au sommet originel

5. Chemin et existence de chemin

Objectif : déterminer le chemin entre 2 sommets – exemple d'algorithme – dérivé du parcours en profondeur de graphe

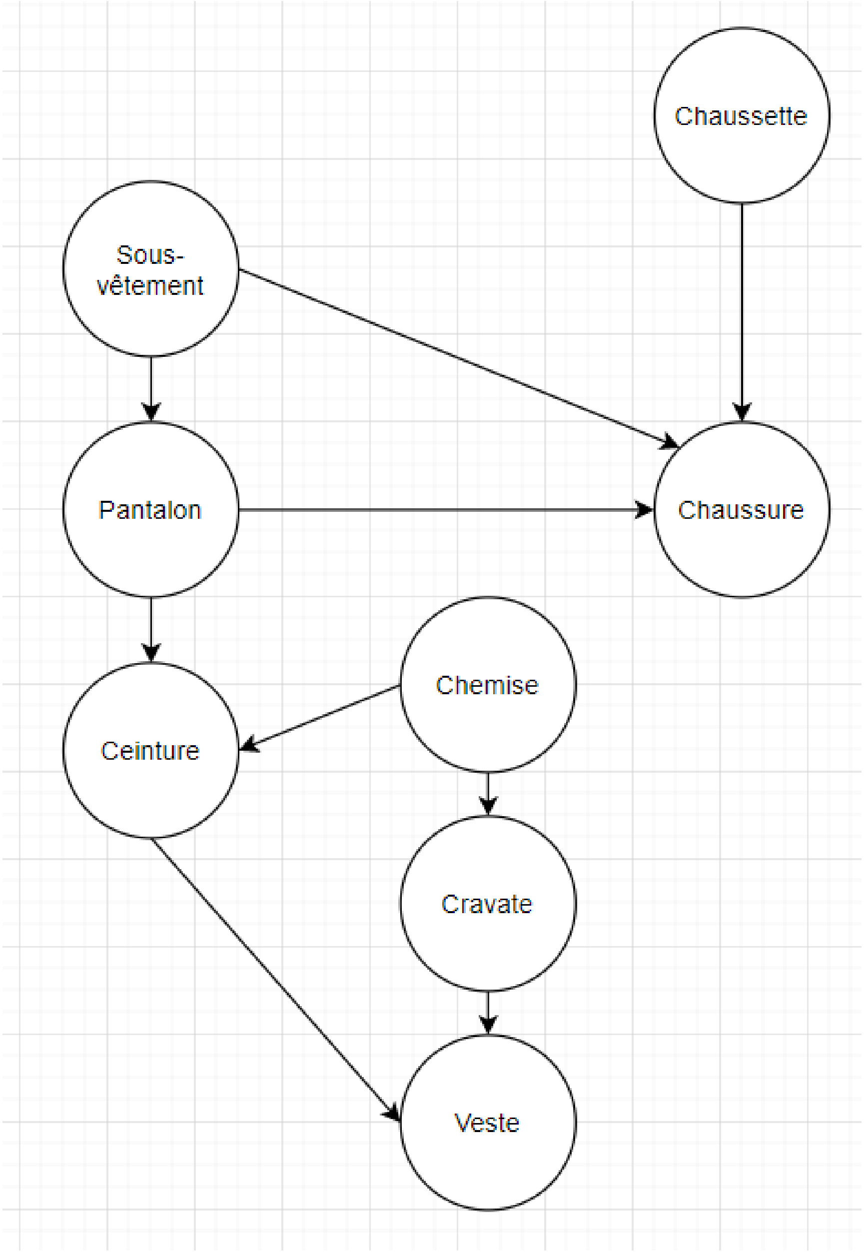
```
Function depthFirstSearchPoint(Graph T, Graph[] *invertedPath, Graph search) {
	if(T.id == search.id) {
		return true
	}
	else if(!T.alreadyDone) {
		T.alreadyDone = true;
		foreach(T.destinations as point) {
			if(depthFirstSearch(point, path, search) == true) {
				invertedPath.push(T);
				return true
			}
		}
	}
	return false;
}
```

->TD9

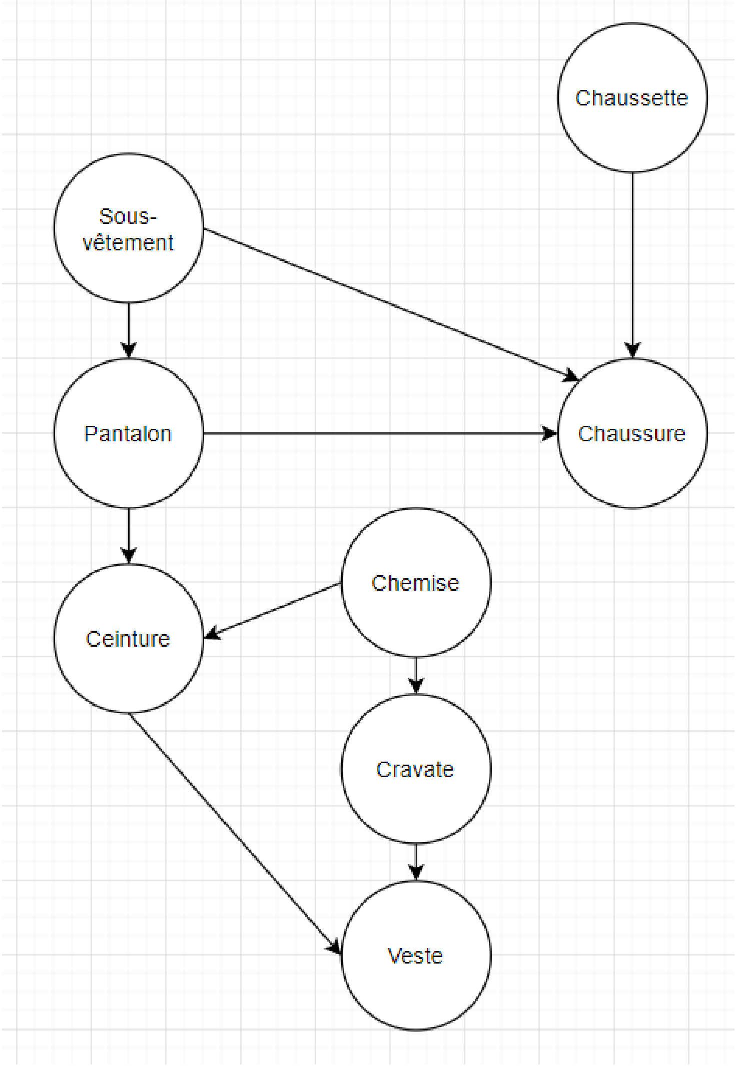
6. Tri topologique

S'utilise dans un graphe acyclique
Permet de résoudre des problématiques d'ordonnancement

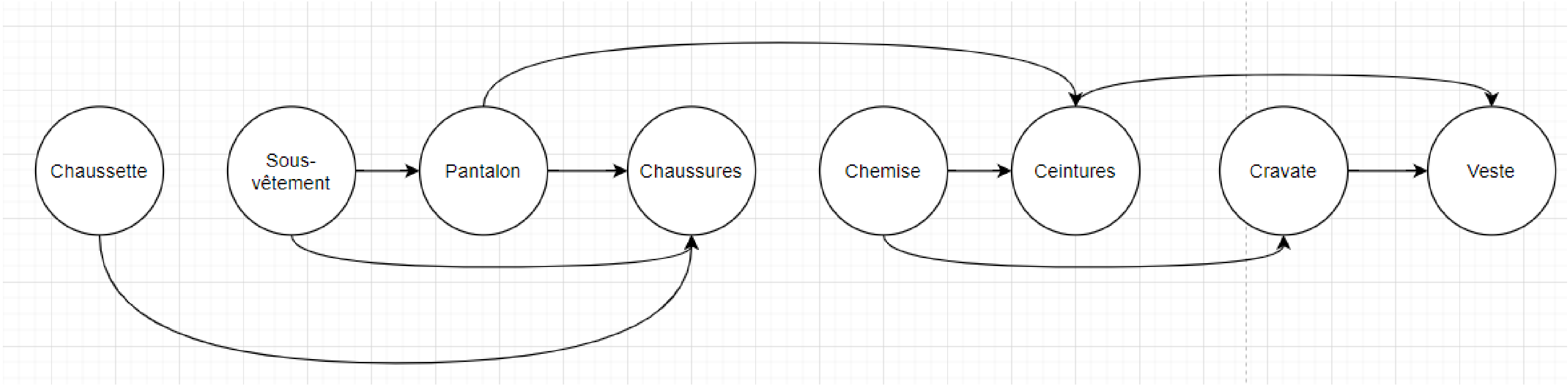
Graphe de dépendance d'une problématique d'habillement



6. Tri topologique



Donne, une fois triée topologiquement, la représentation suivante :



Le graphe n'est pas modifié, seule un ordre est calculé
Cet ordre permet de représenter le graphe avec toutes les flèches dans le même sens

6. Tri topologique

Prérequis

=> s'assurer que le graphe est acyclique

=> Pour tous les sommets de ce graphes, il n'existe aucun chemin permettant d'aller de ce sommet à lui même

Conséquence : le graphe est nécessairement orienté et non symétrique

6. Tri topologique – Mise en œuvre algorithme

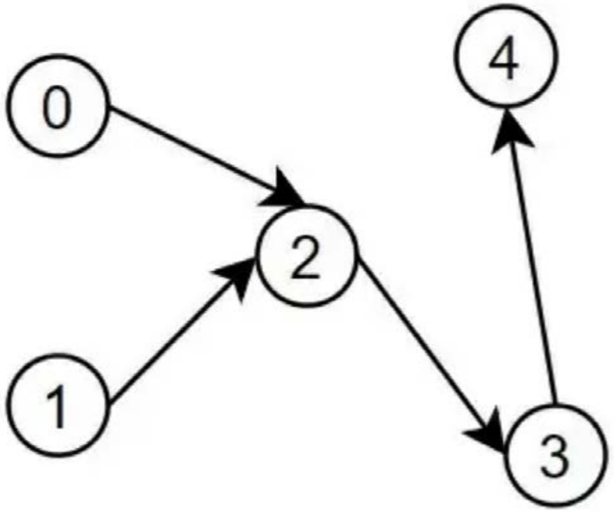
On se base sur le parcours en profondeur.
On construit la liste des sommets de ce graphe
Tant qu'il reste des sommets non visités:

- On effectue un parcours en profondeur à partir du premier sommet non visité trouvée. Dès qu'un traitement sur un sommet à l'intérieur de ce parcours en profondeur est terminé, on extrait ce sommet

Une fois que tout les sommets ont été visités, on inverse la liste des sommets extraits.
Cette ordre est un ordre topologique sur ce graphe

6. Tri topologique – Mise en œuvre algorithme

Exemple n°1

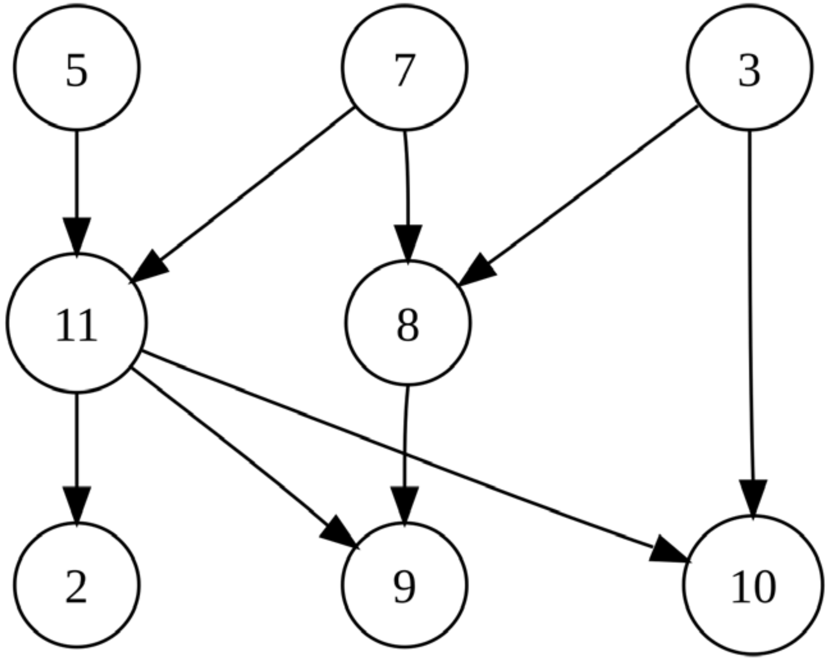


Sommet courant	Sommet en attente de fin de traitement	Sommets marqués comme visités	Liste de résultats inversés
4	4	4	4
2	2	2,4	4
3	2	2,3,4	4,3
2		2,3,4	4,3,2
0		0,2,3,4	4,3,2,0
1		0,1,2,3,4	4,3,2,0,1

On obtient donc le tri topologique : 1,0,2,3,4

6. Tri topologique – Mise en œuvre algorithme

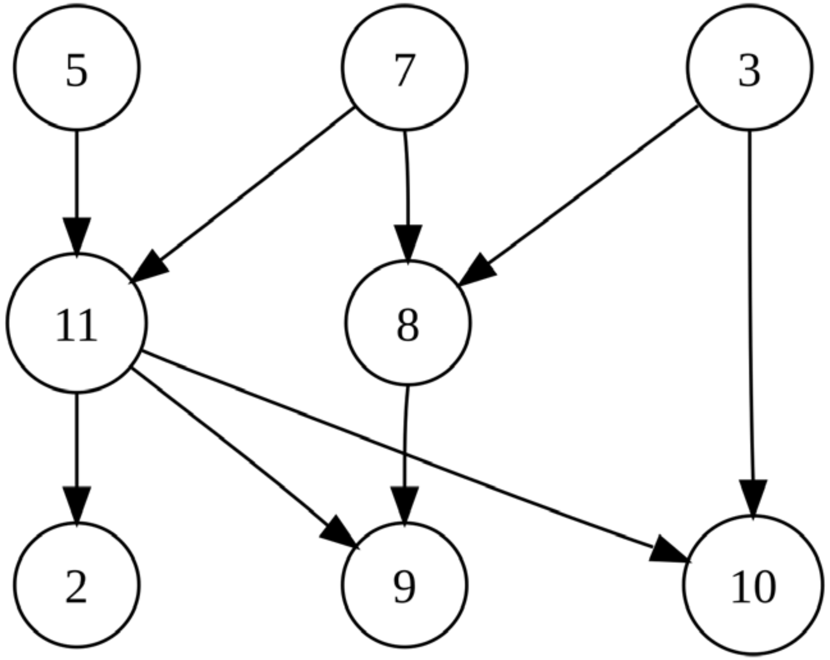
Exemple n°2



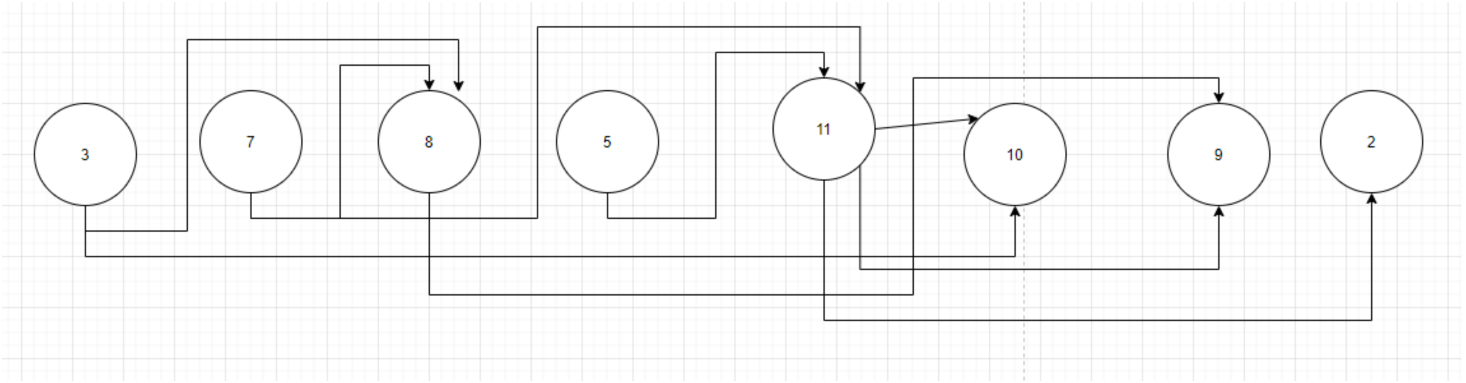
Sommet courant	Sommet en attente de fin de traitement	Sommets marqués comme visités	Liste de résultats inversés
5	5	5	
11	5,11	5,11	
2	5,11,2	5,11,2	
11	5,11	5,11,2	2
9	5,11,9	2,5,11,9	2
11	5,11	2,5,11,9	2,9
10	5,11,10	2,5,10,11,9	2,9
11	5,11	2,5,10,11,9	2,9,10
5	5	2,5,10,11,9	2,9,10,11
		2,5,10,11,9	2,9,10,11,5
7	7	2,5,10,11,9,7	2,9,10,11,5
8	7,8	2,5,10,11,9,7,8	2,9,10,11,5
7	7	2,5,10,11,9,7,8	2,9,10,11,5,8
		2,5,10,11,9,7,8	2,9,10,11,5,8,7
3		2,3,5,10,11,9,7,8	2,9,10,11,5,8,7,3

6. Tri topologique – Mise en œuvre algorithme

Exemple n°2



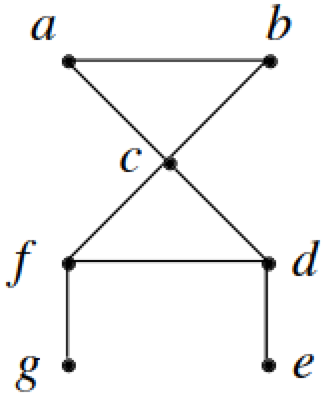
On obtient le tri topologique
3,7,8,5,11,10,9,2



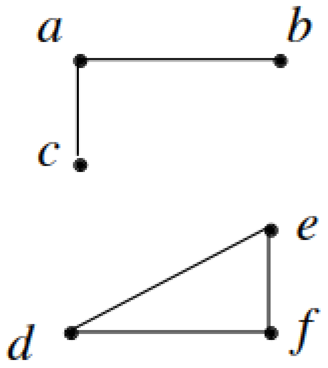
->TD10

7. Composante Connexe / Fortement Connexe

Graphe Connexe : Un graphe non orienté est connexe s'il existe un chemin entre n'importe quelle paire de sommet du graphe



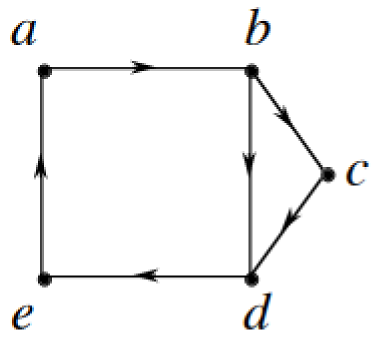
Graphe Connexe



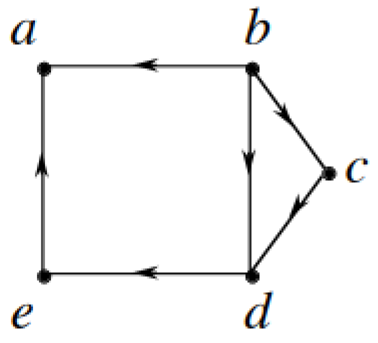
Graphe Non Connexe

7. Composante Connexe / Fortement Connexe

Graphe Fortement Connexe : Un graphe orienté est fortement connexe si, pour toute paire (a,b) de sommets, il existe un chemin de a vers b et un chemin de b vers a



Graphe Fortement Connexe

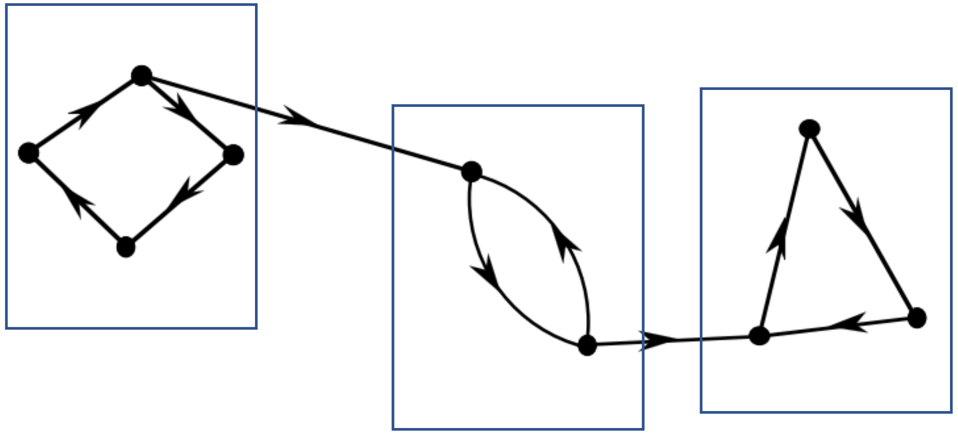


Graphe Non Fortement Connexe (pas de chemin entre a et b par exemple)

7. Composante Connexe / Fortement Connexe

Composante Connexe d'un graphe non orienté : Sous-graphe du graphe non orienté considéré qui est un graphe connexe

Composante Fortement Connexe d'un graphe orienté : Sous-graphe du graphe orienté considéré qui est un graphe fortement connexe



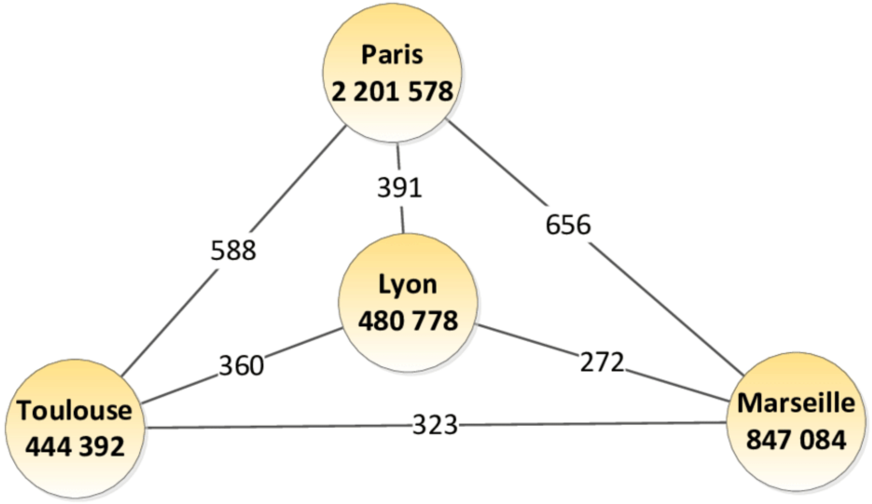
Ce graphe contient 3 composantes fortement connexes

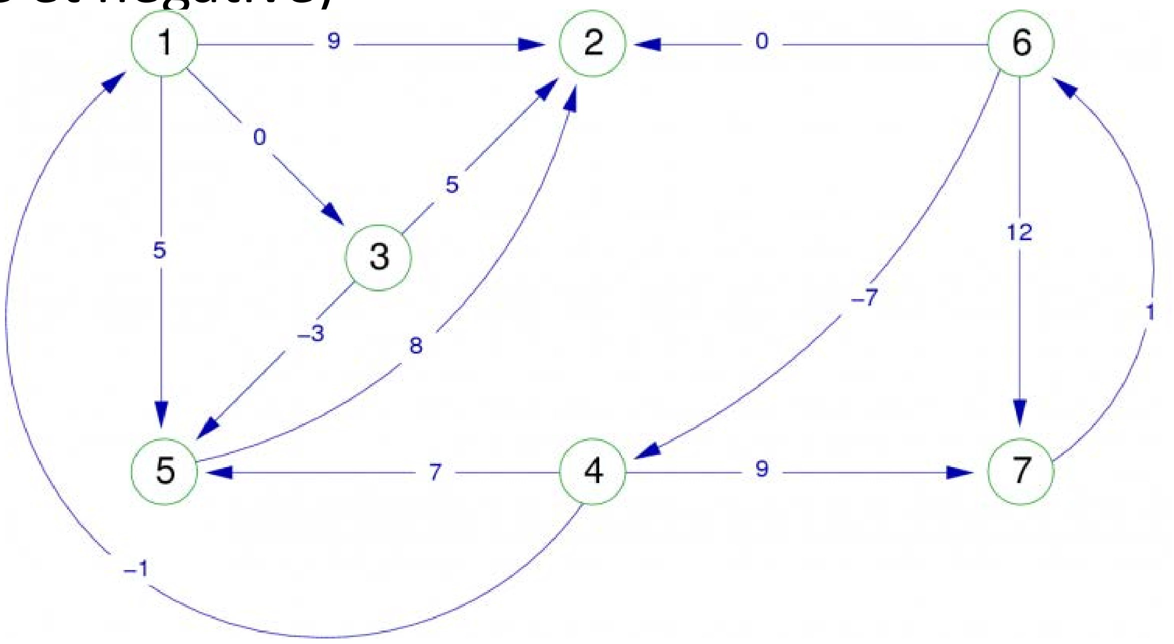
->TD10

8. Recherche du plus court chemin – notion de coût / distance

Un graphe peut être valué/pondéré : on indique un cout à chaque arrête. Cette information représente le coût/distance à dépenser/parcourir pour se déplacer d'un sommet à un autre

Cette valeur est un nombre réel (positive et négative)



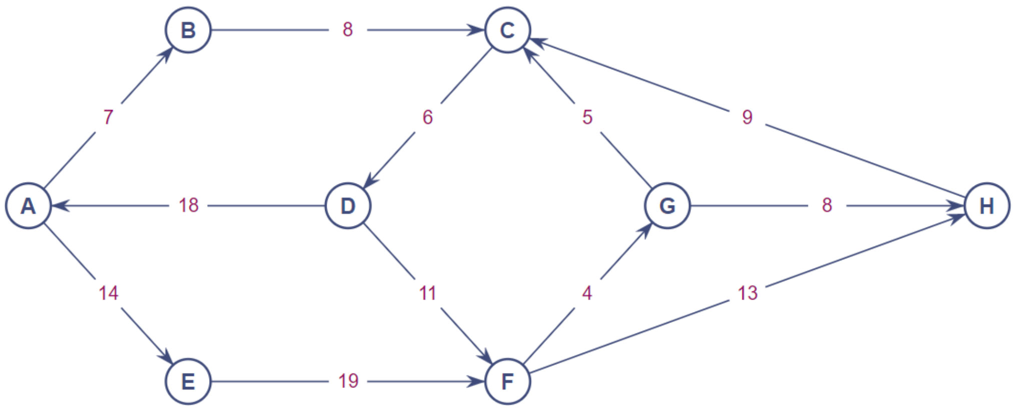


8. Recherche du plus court chemin – Matrice des poids

Quand une arrête existe, on précise le poids de cette arc pour aller du sommet affectée à la ligne vers le sommet affectée à la colonne

Quand une arrête n’existe pas, on affecte la valeur infini

On considère un poids nul pour aller d’un sommet à lui-même si le graphe ne dispose pas d’une arrête sur lui même



$$P = \begin{pmatrix} 0 & 7 & \infty & \infty & 14 & \infty & \infty & \infty \\ \infty & 0 & 8 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 6 & \infty & \infty & \infty & \infty \\ 18 & \infty & \infty & 0 & \infty & 11 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & 19 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 4 & 13 \\ \infty & \infty & 5 & \infty & \infty & \infty & 0 & 9 \\ \infty & \infty & 9 & \infty & \infty & \infty & \infty & 0 \end{pmatrix}$$

8. Recherche du plus court chemin – Itinéraire

Un chemin entre 2 sommets se voir alors affecté un cout en plus du nombre d’arrête qu’il contient.

Cela permet de représenté de manière plus approfondie, par exemple, une structure de données de :

- Itinéraire entre 2 villes
- Un réseau informatique
- Un réseau de transport

On peux affecter autant de cout différent à une même arrête afin de mieux caractériser une données. Par exemple, le temps de latence et le débit pour chaque lien informatique dans un graphe représentant un réseau

8. Recherche du plus court chemin – Itinéraire

Comment déterminer le plus court chemin entre 2 sommets ?

On peux essayer naïvement de de trouver tout les itinéraires possibles entre 2 sommets, calculer le cout de chacun puis indiquer celui avec le plus petit cout

→ Non optimale en complexité, assez lourd comme fonctionnement

→ Utilisation de l'algorithme de Dijkstra

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Prérequis :</u>

- Le graphe ne contient que des poids positifs (risque de non convergence/résultats incohérent)

→ Donne le poids du plus petit chemin du graphe vers chaque sommet accessible depuis le sommet de départ

→ Permet de calculer l'itinéraire avec ce plus petit poids vers chaque sommet accessible

Attention : il est fourni ici une explication proche d'une proposition d'implémentation de l'algorithme de Dijkstra, pas une version couramment expliqué permettant de le faire « à la main »

8. Recherche du plus court chemin – Algorithme de Dijkstra

```
Class Node {
		int id;
		 Branch[] children;
}
Class Branch {
		double weight
		Node destination
}
Class Dijkstra {
		Node node;
		Boolean visited;
		distanceFromSource double;
		bestParentFromSource: Node
}
```

Tout les propriétés de ces classes doivent être passés par référence pour la suite des algorithme

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Initialisation technique:</u>
On construit une liste d'instance de la classe Dijkstra à partir de la liste des sommets de notre graphe.

On initialise sur tout les éléments de cette liste les propriétés avec les valeurs suivante :

- visited : false
- distanceFromSource : +infinite
- bestParentForSource : null

8. Recherche du plus court chemin – Algorithme de Dijkstra

Initialisation algorithmique :

Placer la valeur 0 à la propriété distanceFromSource du nœud de départ

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Algorithme à mettre en œuvre :</u>

Dans la liste d'instance de Dijkstra, trouver l'instance avec la plus petite valeur à la propriété distanceFromSource et ayant une valeur false à la propriété visited.
Placer cette instance dans une variable currentNode, ou placer la valeur null si cette instance ne peux être trouvé

```
Tant Que currentNode != NULL {
		currentNode.visited = true
		foreach(currentNode.children as branch) {
				Si(branch.node.distanceFromSource > currentNode. distanceFromSource + branch.weight) {
				//On a trouvé un chemin plus optimal, on le sauvegarde
				branch.node.distanceFromSource = currentNode. distanceFromSource + branch.weight
				branch.node. bestParentFromSource = currentNode
				}
		}
		Affecter à currentNode l'instance avec la plus petite valeur à la propriété distanceFromSource et
ayant une valeur false à la propriété visited dans la liste d'instance de Dijkstra ou Null
}
```

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Algorithme à mettre en œuvre :</u>

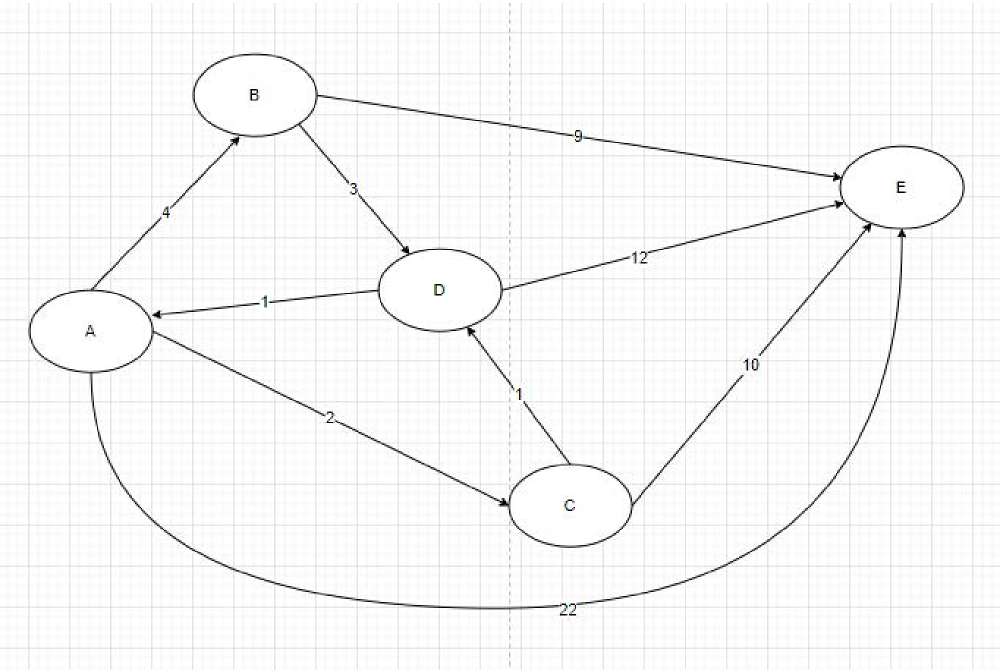
A la fin de l'algorithme, dans la listes des instances de Dijkstra, on dispose pour chaque élément :

- De la poids du chemin entre cet élément et le nœud source dans distanceFromSource
- Du nœud parent vers lequel il faut se diriger (propriété bestParentFromSource) depuis cet élément pour aller vers le nœud source de manière optimale

    → On peux donc retracer, si cette propriété est différente de null (et donc qu'un chemin existe), l'itinéraire entre le nœud de départ et cette élément

8. Recherche du plus court chemin – Algorithme de Dijkstra

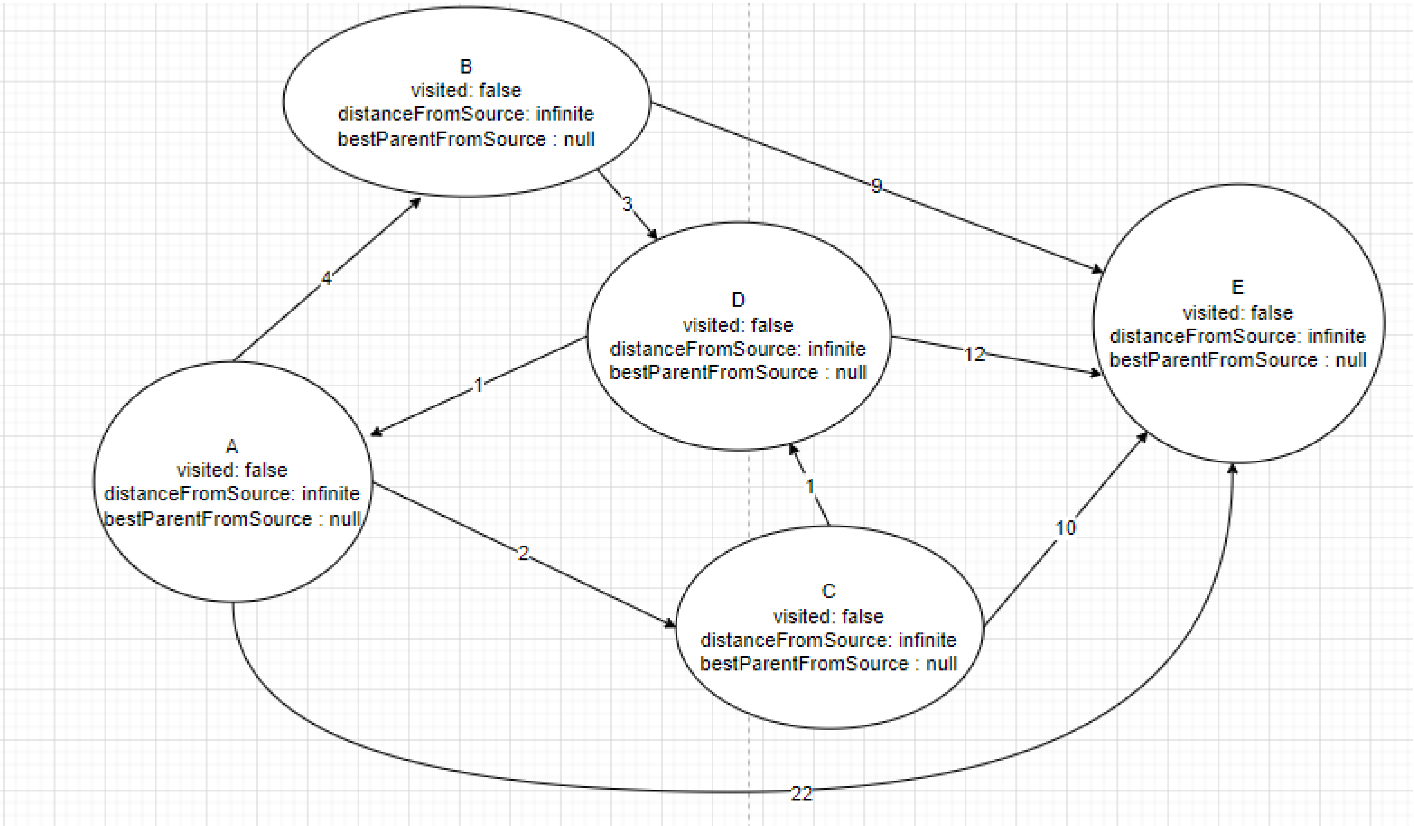
Exemple de mise en œuvre :



Meilleur chemin pour aller de A à E

8. Recherche du plus court chemin – Algorithme de Dijkstra

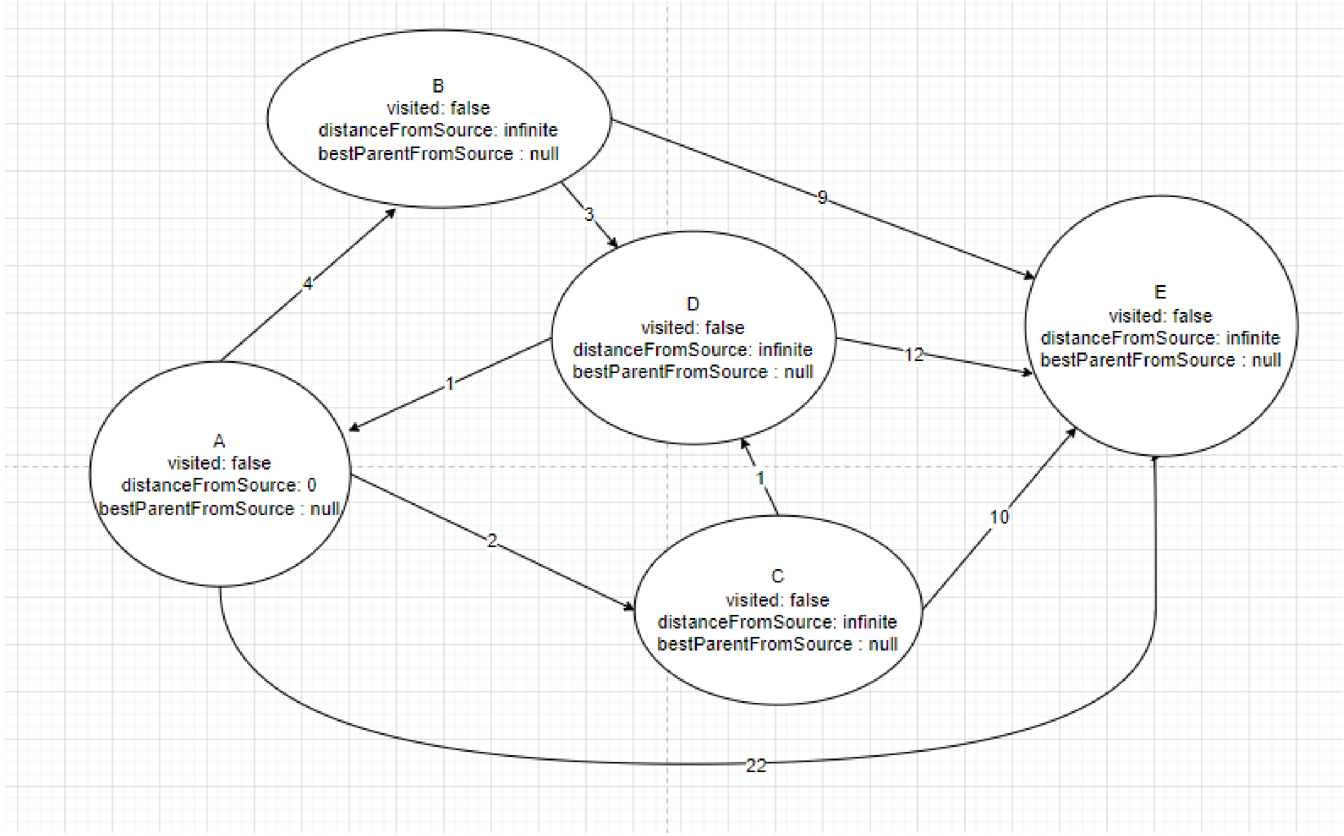
Exemple de mise en œuvre :



On construit les instance Dijkstra

8. Recherche du plus court chemin – Algorithme de Dijkstra

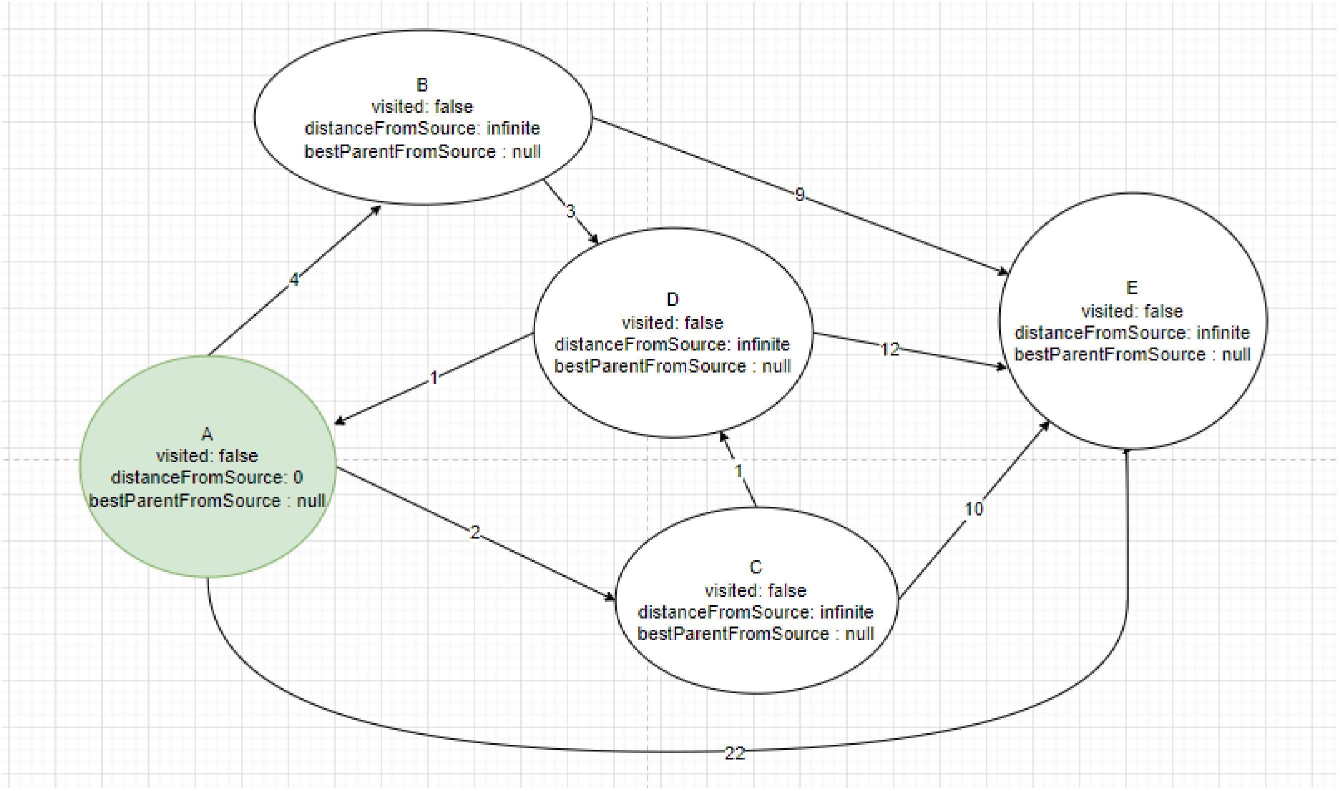
Exemple de mise en œuvre :



On initialise le nœud de départ A en placant distanceFromSource à zéro

8. Recherche du plus court chemin – Algorithme de Dijkstra

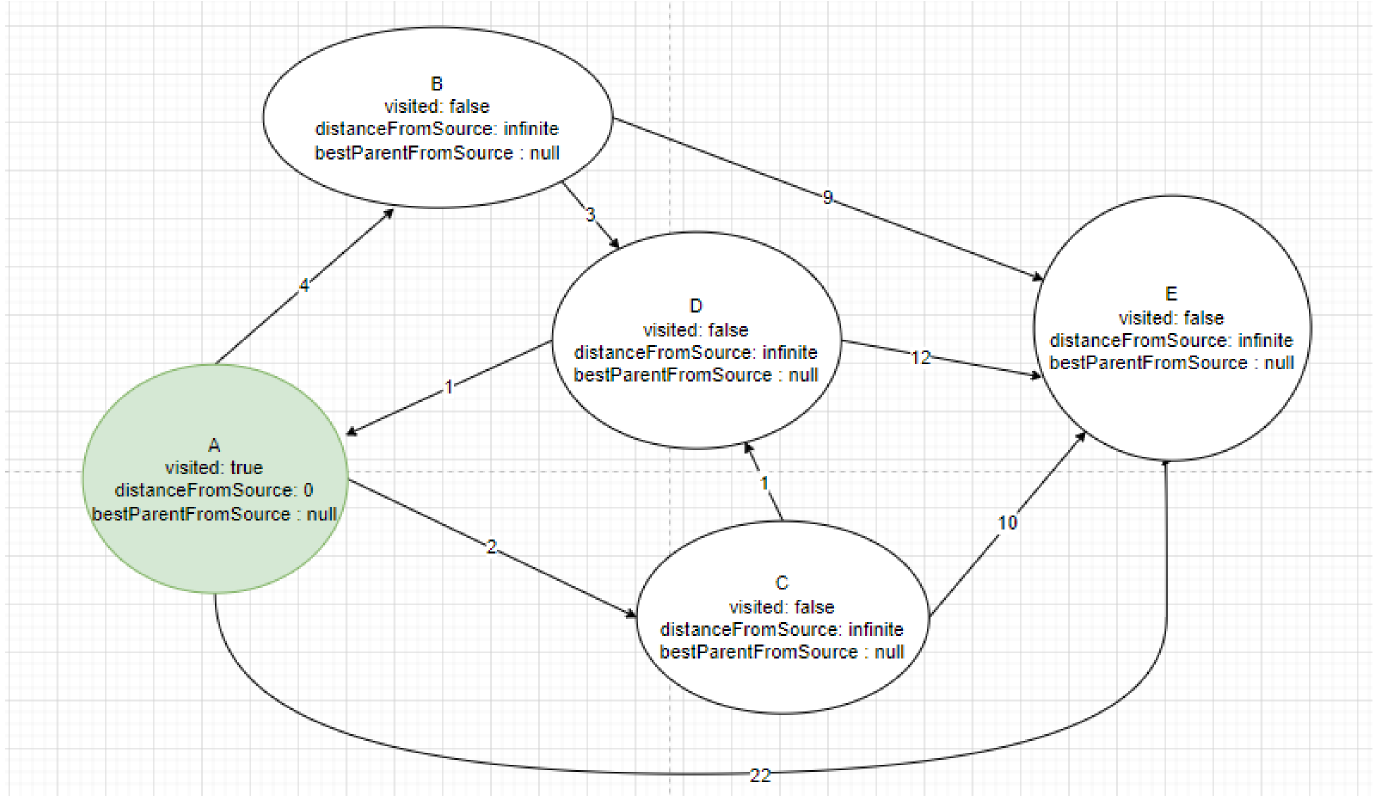
Exemple de mise en œuvre :



On initialise l'objet courant (en vert)

8. Recherche du plus court chemin – Algorithme de Dijkstra

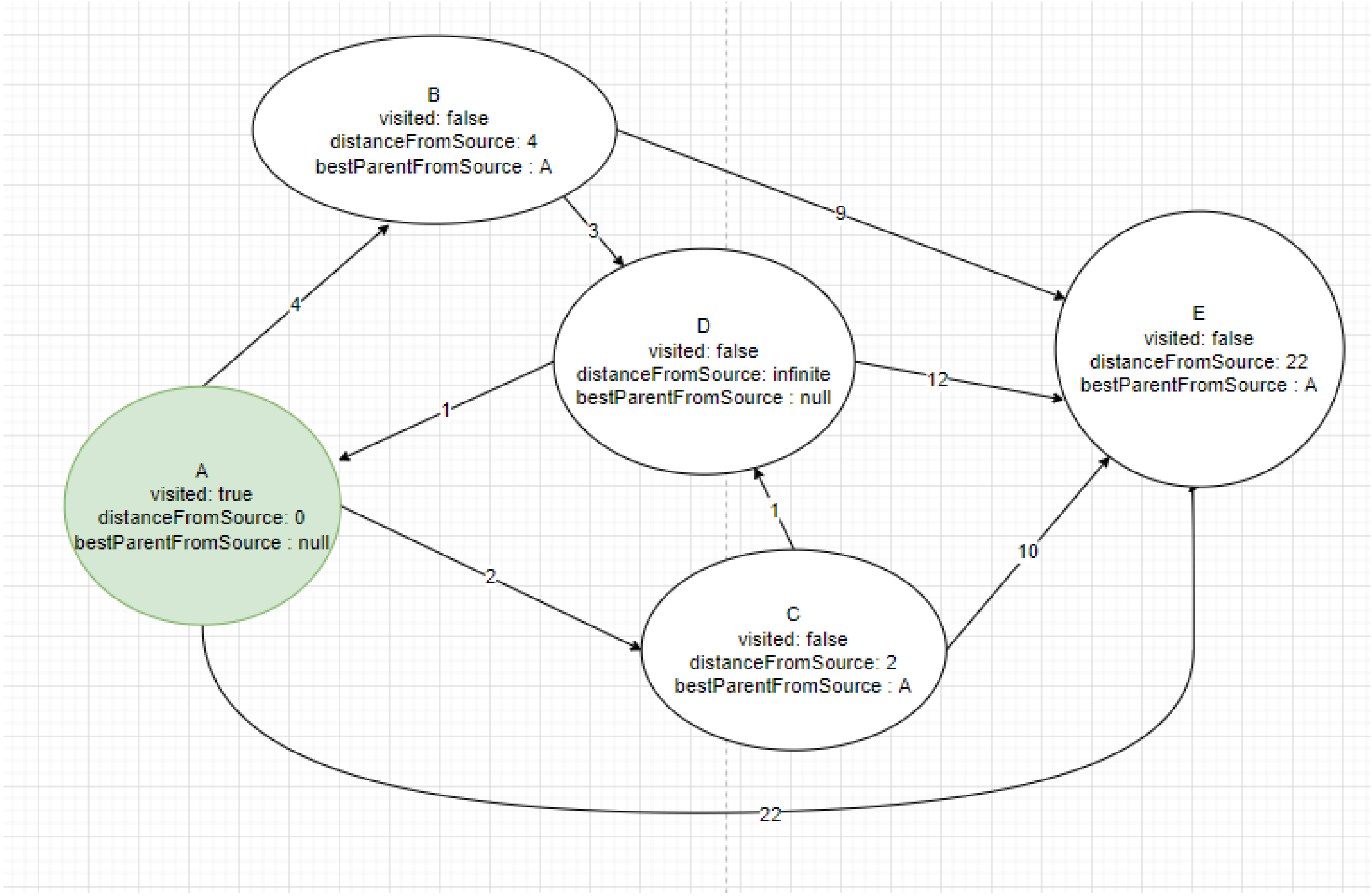
<u>Exemple de mise en œuvre :</u>



On place visited à true dans l’objet courant

8. Recherche du plus court chemin – Algorithme de Dijkstra

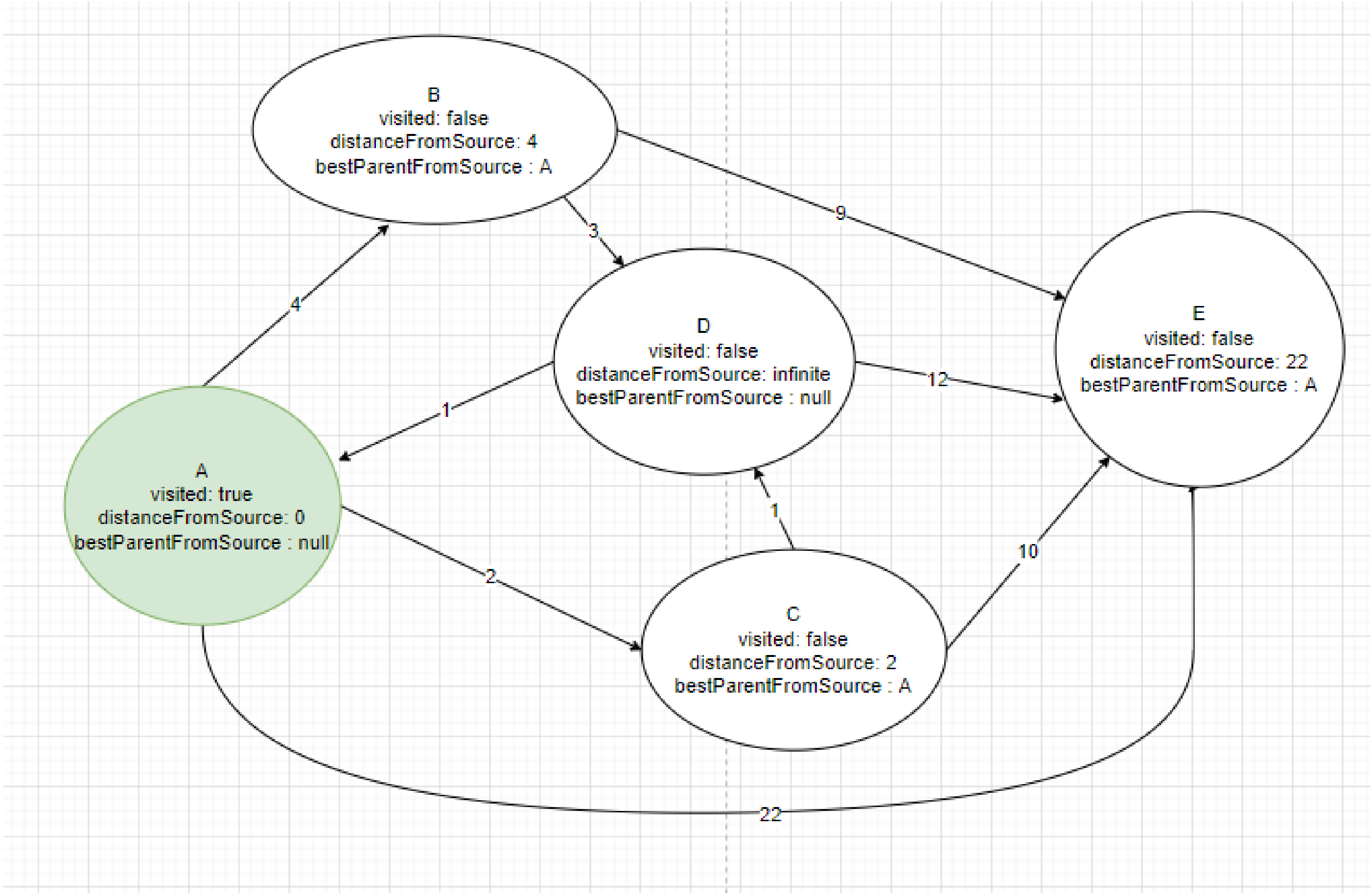
<u>Exemple de mise en œuvre :</u>



On parcours les nœud enfants
Les valeurs de distance dans les enfants B, C et E sont infinite.
On met donc dans distanceFromSource les valeurs des branches entre A et chaque nœud et on place le nœud A dans bestParentFromSource

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :



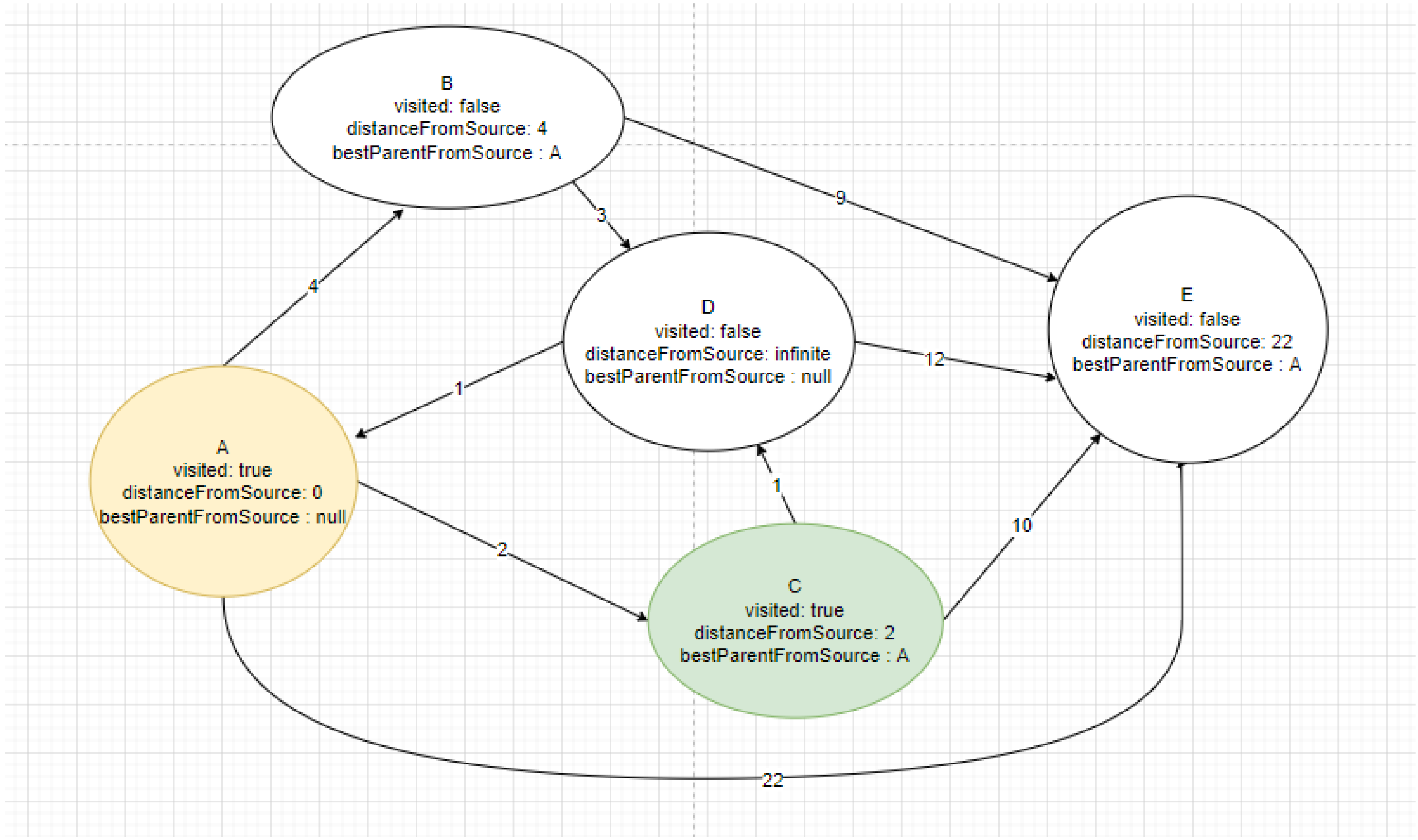
On recherche le nouveau objet courant suivant ces 2 conditions :

- L'élement a la valeur visited à false
- Plus petite valeur dans la propriété distanceFromSource

→ Le prochain objet courant est donc le nœud C

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :

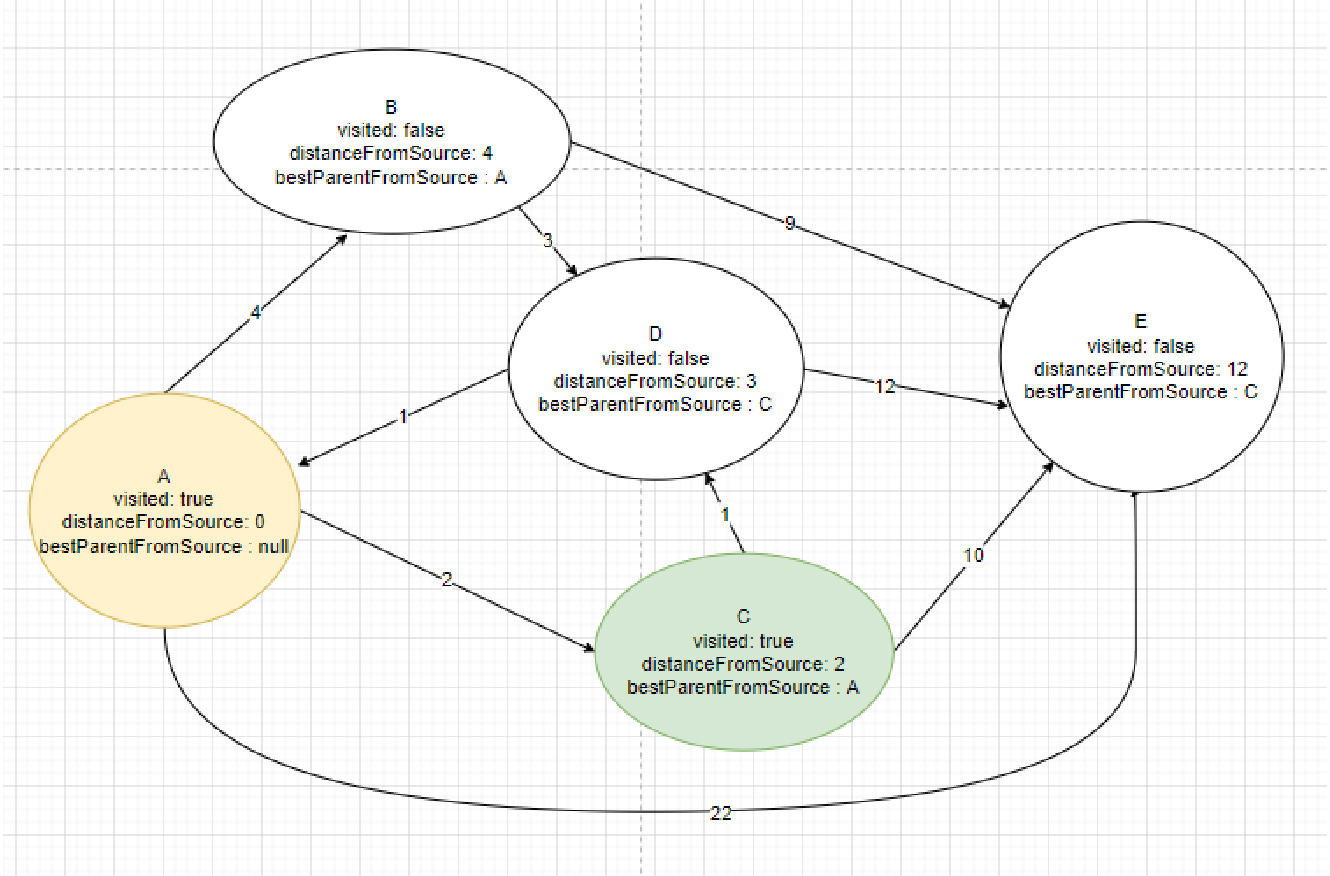


C est le nœud courant

On place la valeur visited à true dans le nœud courant

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :



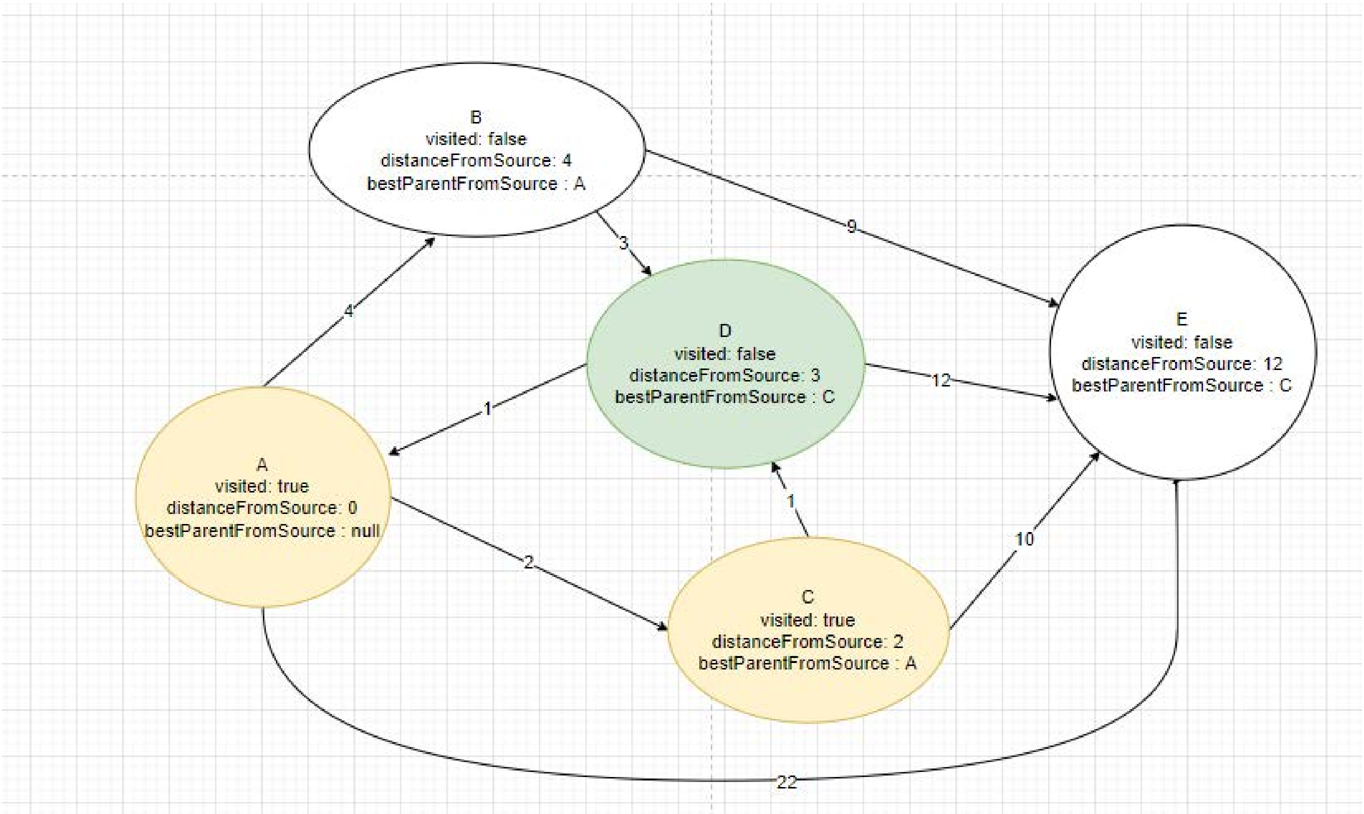
On effectue le traitement à chaque enfant du nœud C

- Pour le nœud D, distanceFromSource est à infinite. On va donc mettre à jour les informations sur ce nœud et y placer C dans bestParentFromSource et 2 + 1 = 3 dans distanceFromSource

- Pour le nœud E, la distance pour aller de A à E en passant par C est de 2 + 10 = 12, soit meilleure que la distance précédamment calculé de 22. On met donc à jour le nœud E en mettant dans distanceFromSource 12 et bestParentFromSource C

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Exemple de mise en œuvre :</u>



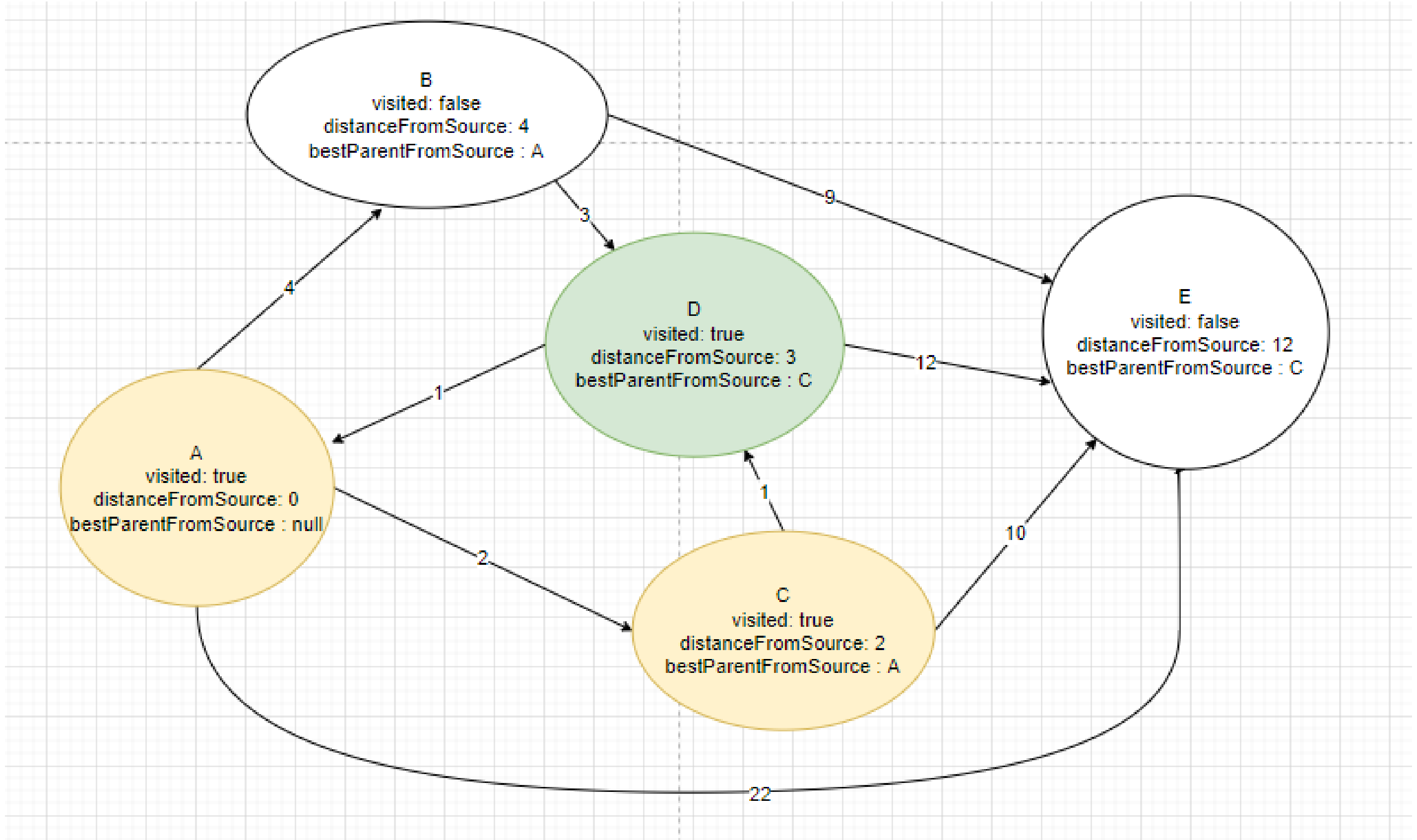
On recherche le nouveau objet courant suivant ces 2 conditions :

- L'élement a la valeur visited à false
- Plus petite valeur dans la propriété distanceFromSource

→ Le prochain objet courant est donc le nœud D

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Exemple de mise en œuvre :</u>

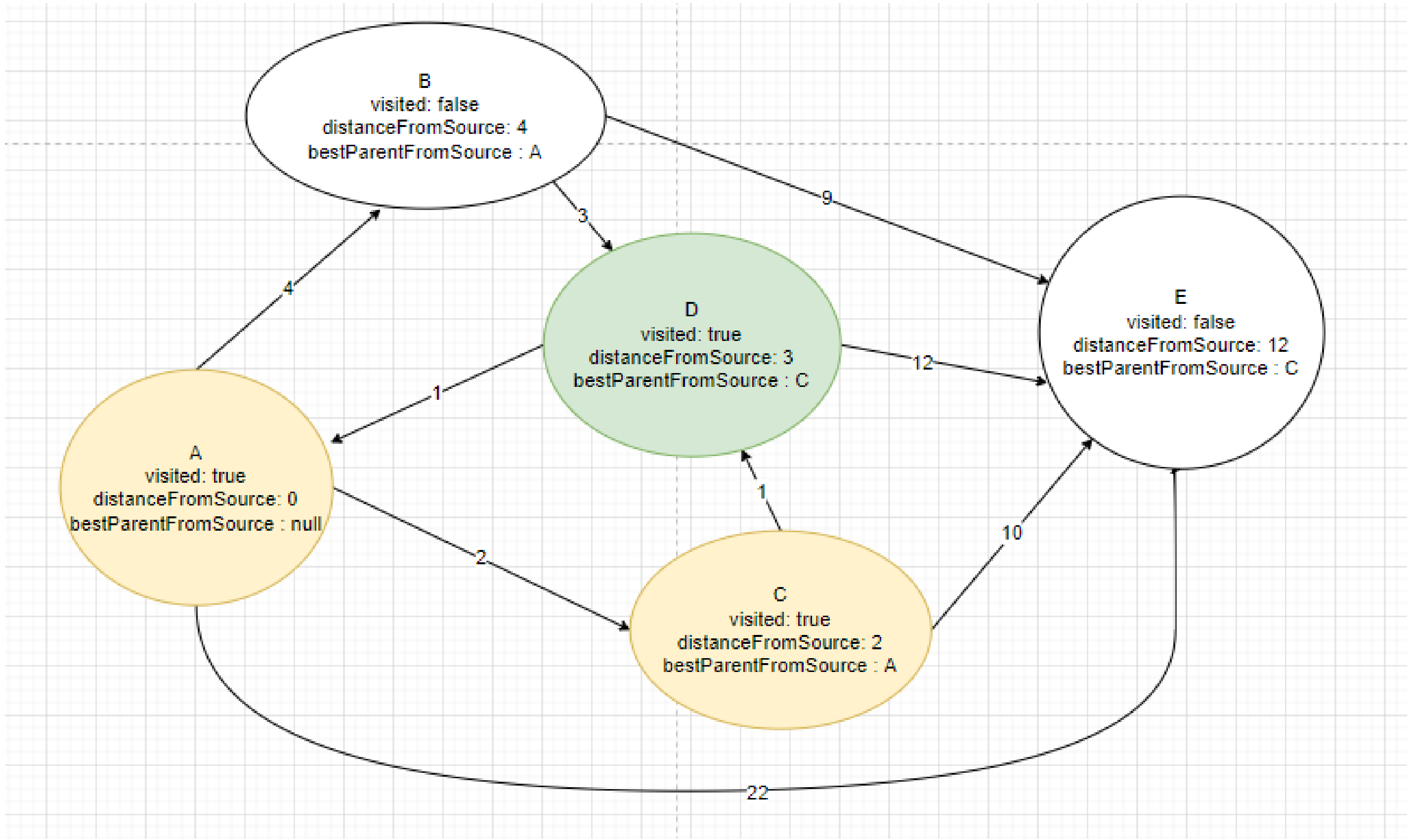


D est le nœud courant

On place visited à true dans l'objet courant

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Exemple de mise en œuvre :</u>



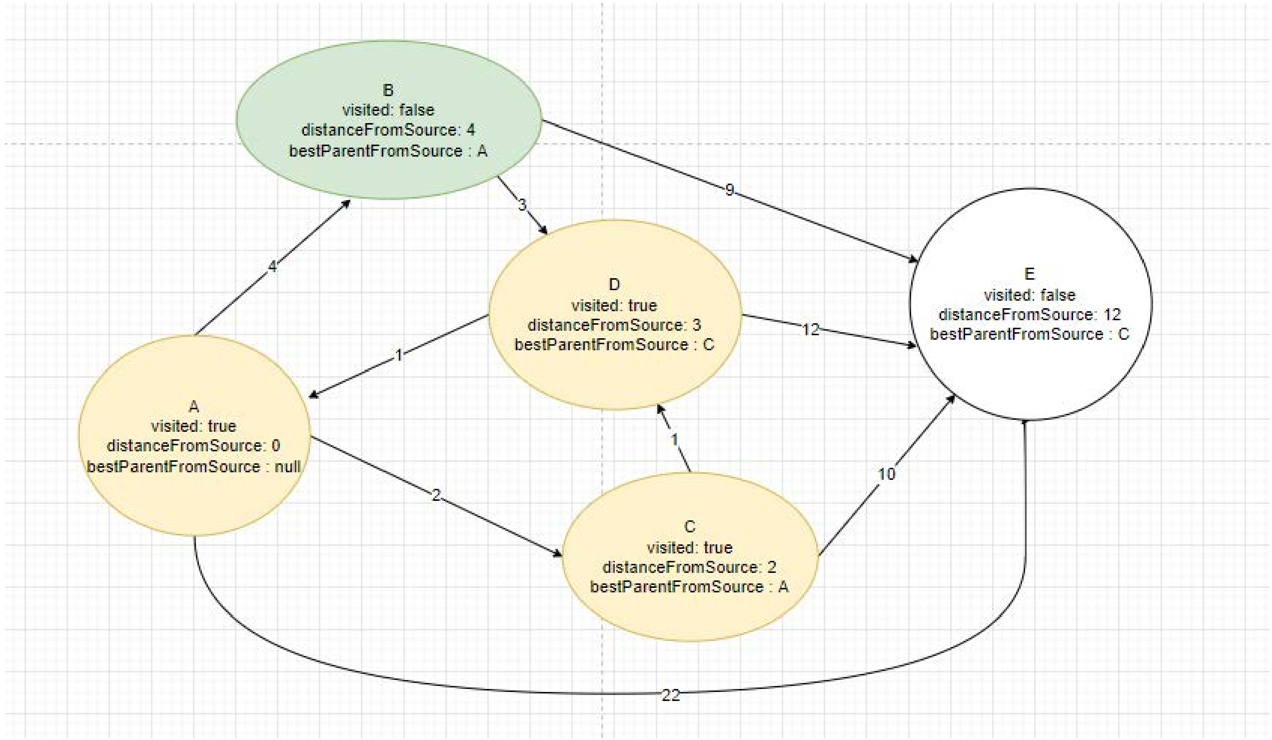
On effectue le traitement à chaque enfant du nœud D

- Pour le nœud A, distanceFromSource est à 0, difficile de faire mieux. On ne fait rien

- Pour le nœud E, la distance pour aller de A à E en passant par D est de 3 + 12 = 15, soit moins bon que la distance précédamment calculé de 12. On ne fait donc rien

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :



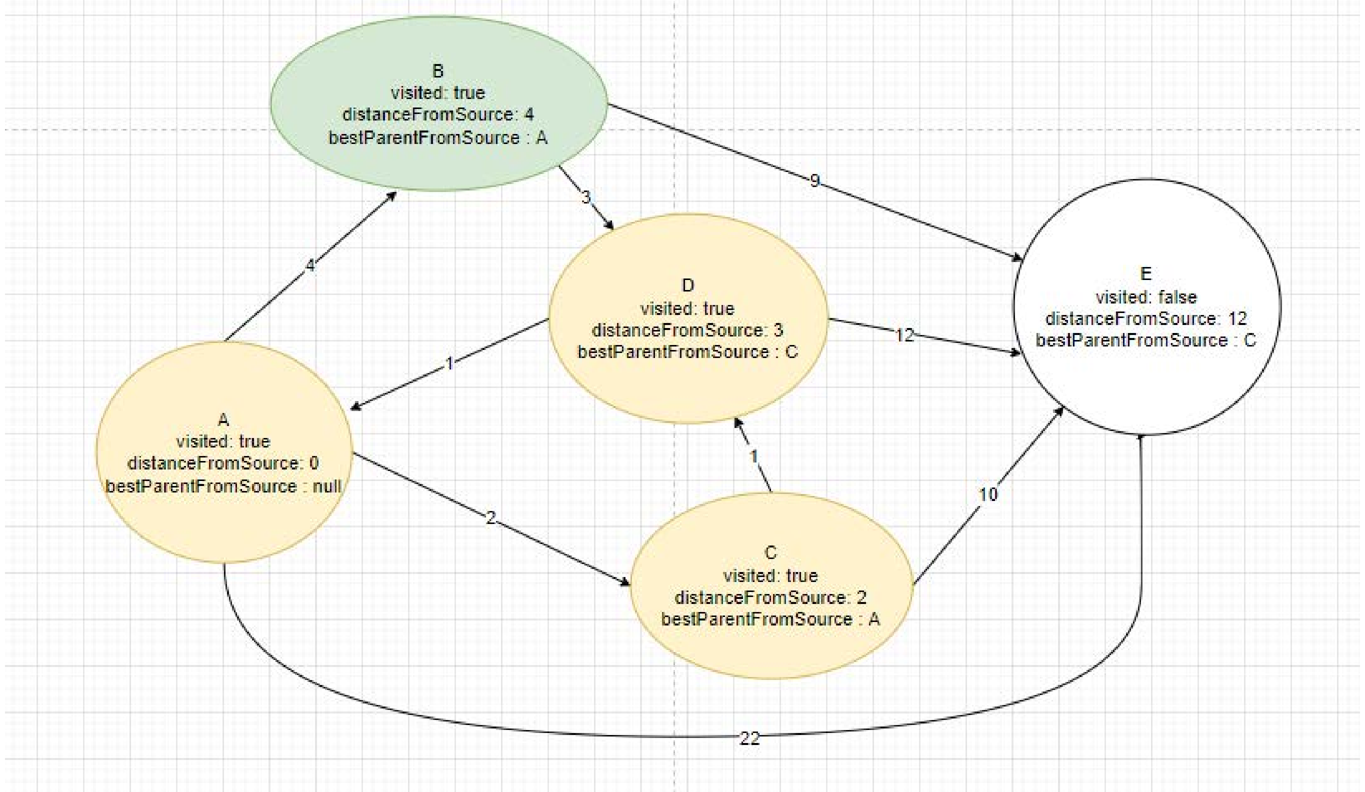
On recherche le nouveau objet courant suivant ces 2 conditions :

- L’élement a la valeur visited à false
- Plus petite valeur dans la propriété distanceFromSource

→ Le prochain objet courant est donc le nœud B

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :

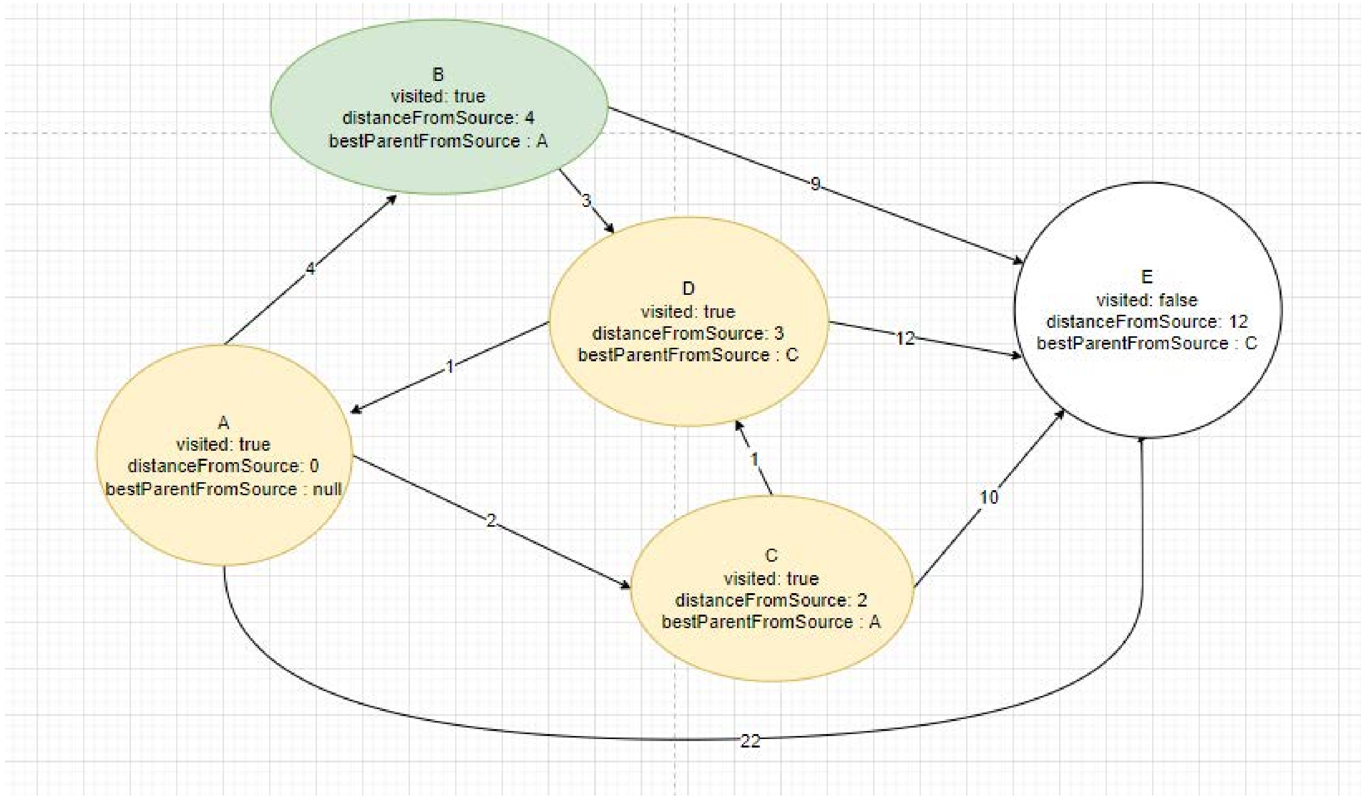


B est le nœud courant

On place visited à true dans l'objet courant

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :



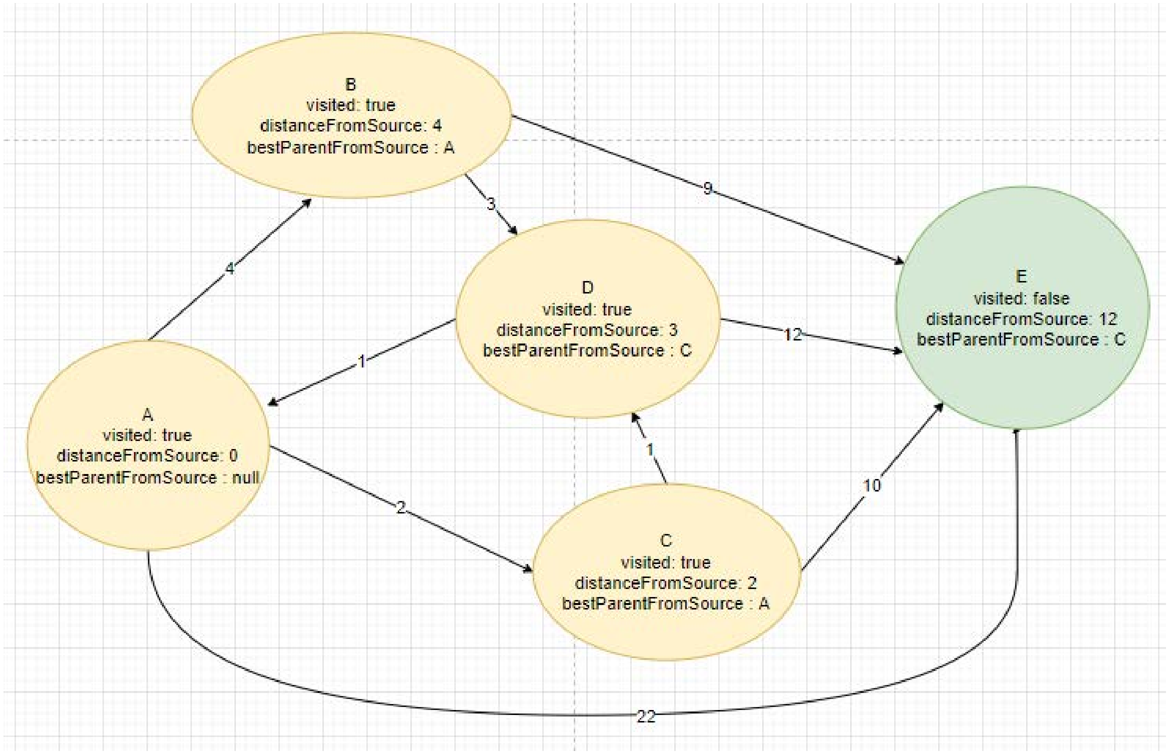
On effectue le traitement à chaque enfant du nœud D

- Pour le nœud D, la distance pour aller de A à D en passant par B est de 7, moins bon que le chemin actuellement enregistré (3) -> on ne fait rien

- Pour le nœud E, la distance pour aller de A à E en passant par B est de 4 + 9 = 13, soit moins bon que la distance précédemment calculé de 12. On ne fait donc rien

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :



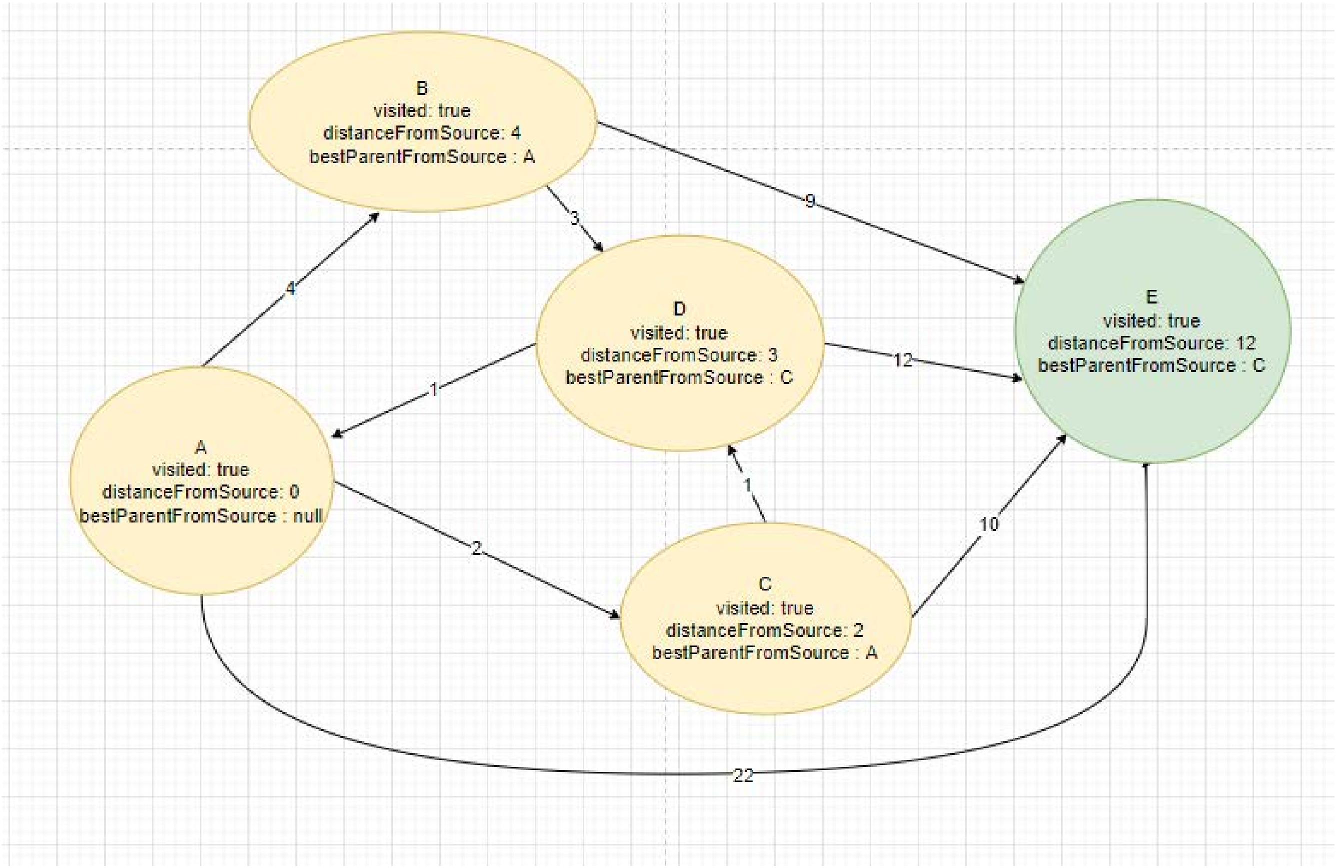
On recherche le nouveau objet courant suivant ces 2 conditions :

- L'élement a la valeur visited à false
- Plus petite valeur dans la propriété distanceFromSource

→ Le prochain objet courant est donc le nœud E

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Exemple de mise en œuvre :</u>

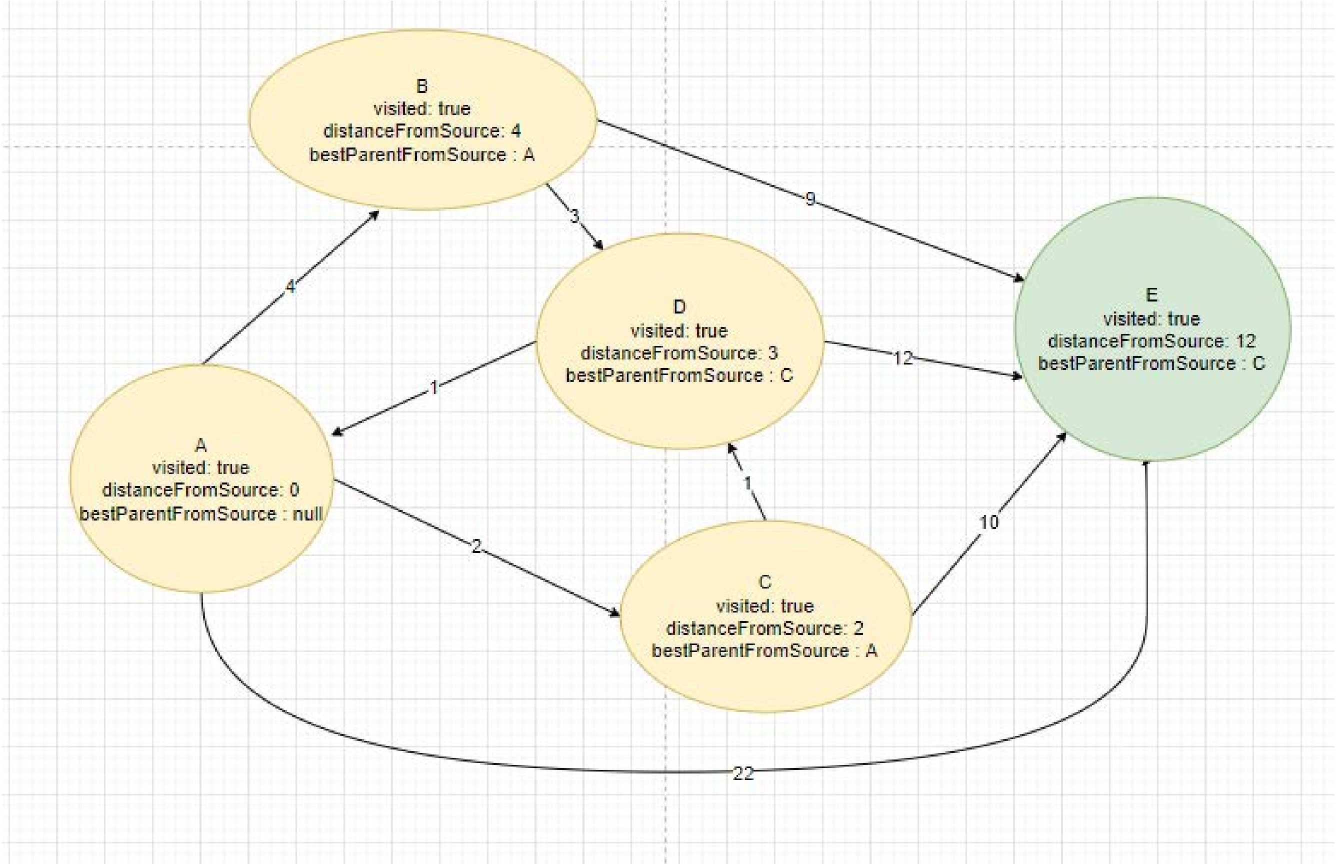


E est le nœud courant

On place visited à true dans l'objet courant

8. Recherche du plus court chemin – Algorithme de Dijkstra

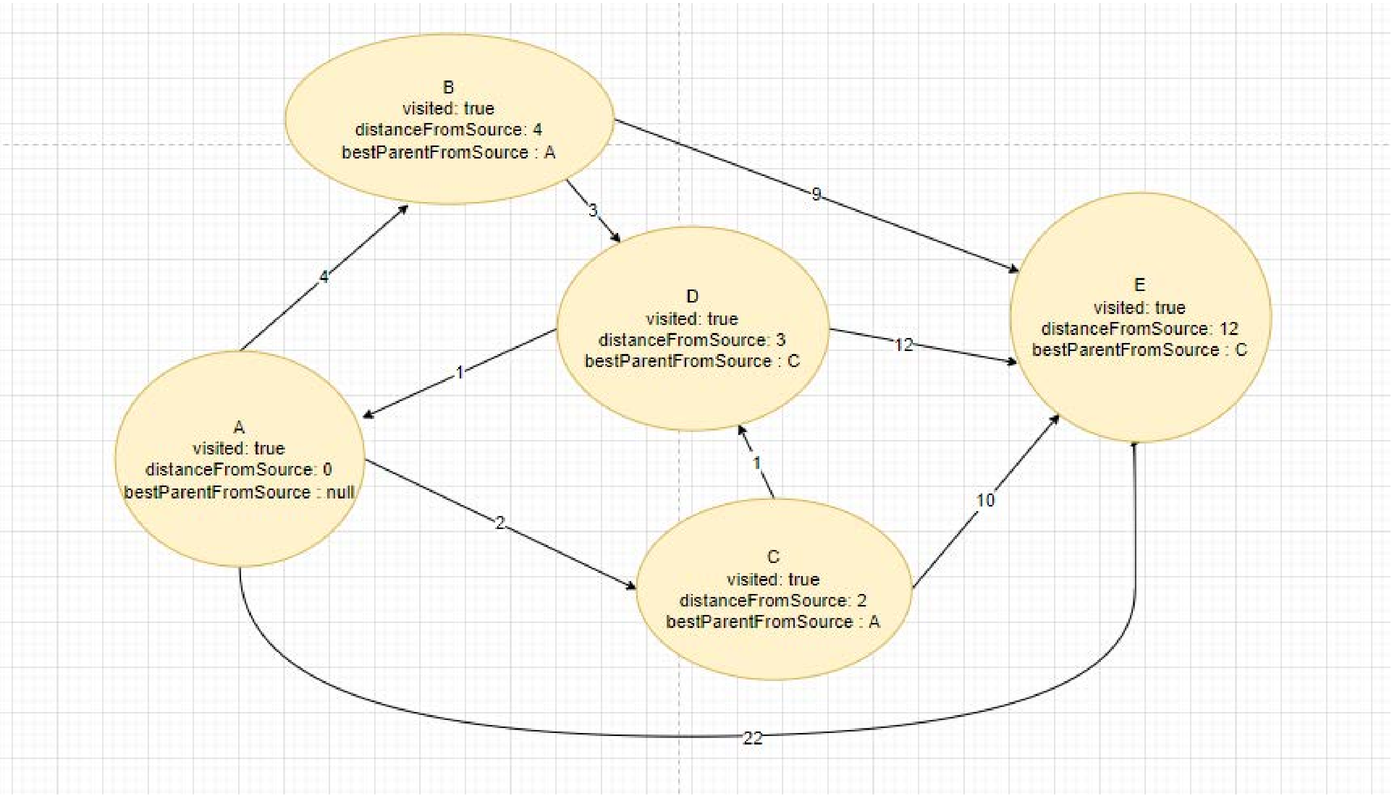
Exemple de mise en œuvre :



On effectue le traitement à chaque enfant du nœud E.
E n'a pas d'enfant, on ne fait donc rien

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Exemple de mise en œuvre :</u>



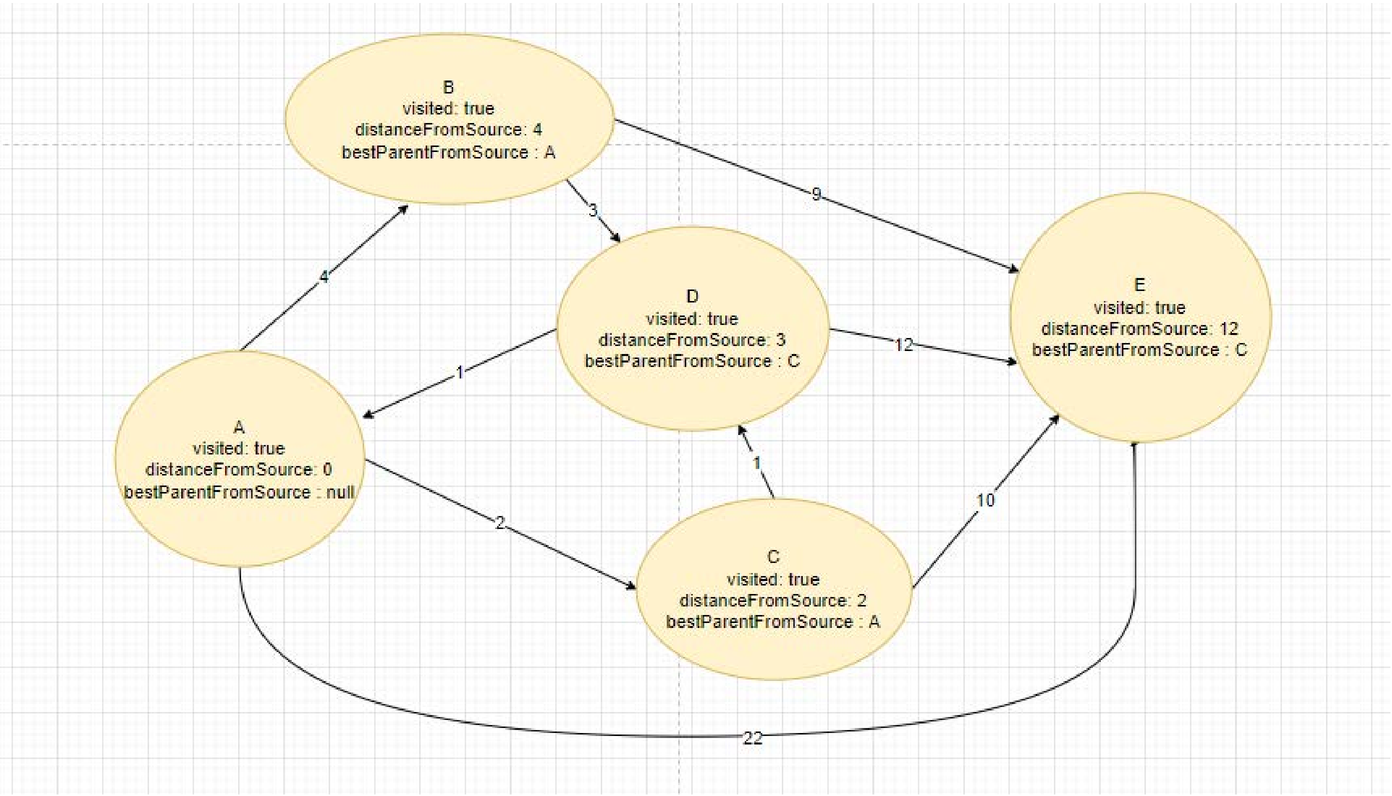
On recherche le nouveau objet courant suivant ces 2 conditions :

- L'élement a la valeur visited à false
- Plus petite valeur dans la propriété distanceFromSource

→ Le prochain objet courant est donc null
→ L'algorithme est donc terminé

8. Recherche du plus court chemin – Algorithme de Dijkstra

<u>Exemple de mise en œuvre :</u>



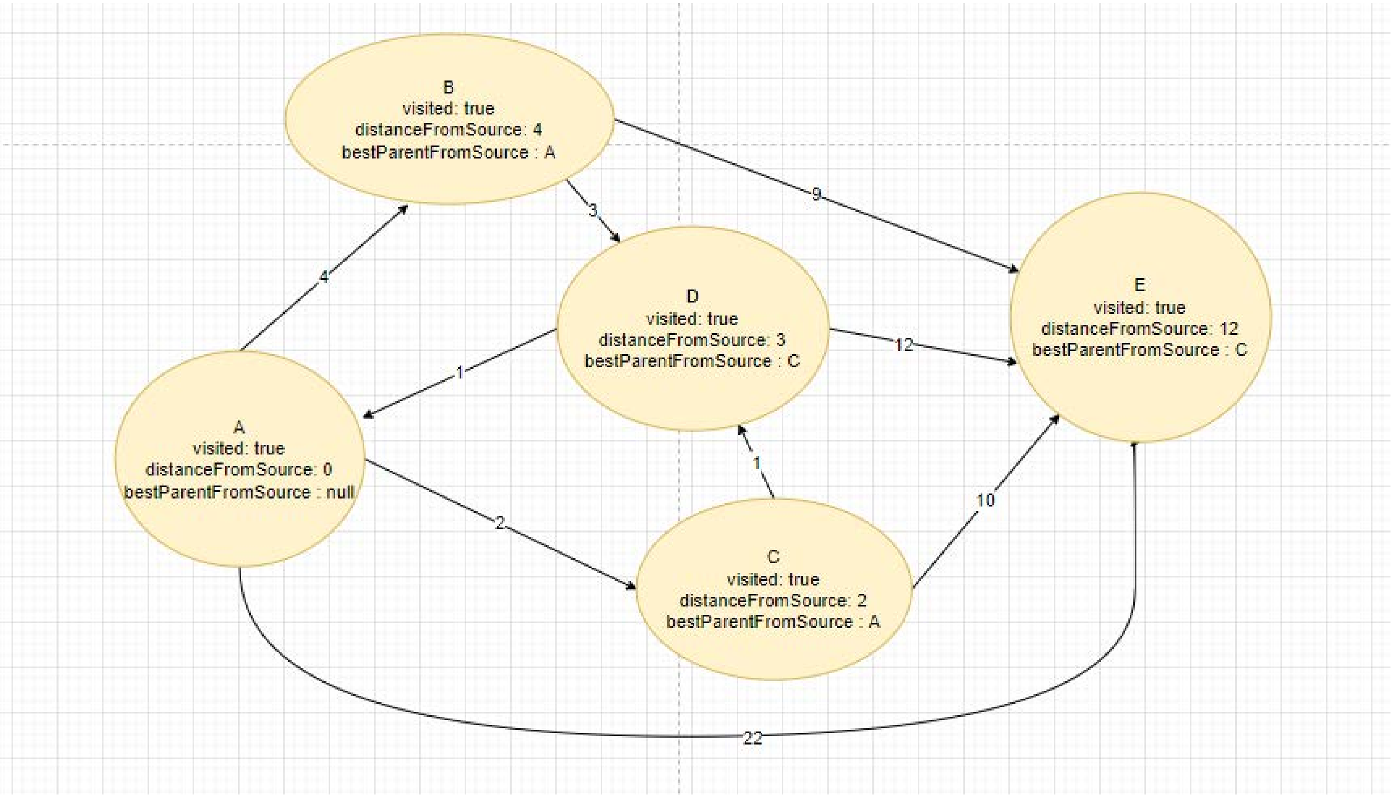
Un résumé des informations obtenu par l'algorithme

Distance minimale depuis la nœud A			
B	C	D	E
4	2	3	12

Pour atteindre ce nœud dans un chemin de distance minimale depuis A, il faut que le nœud précédent soit			
B	C	D	E
A	A	C	C

8. Recherche du plus court chemin – Algorithme de Dijkstra

Exemple de mise en œuvre :



Un résumé des informations obtenu par l'algorithme

Pour atteindre ce nœud dans un chemin de distance minimale depuis A, il faut que le nœud précédent soit			
B	C	D	E
A	A	C	C

Ce qui permet de déduire, pour aller de A vers E avec une distance minimale, de suivre le chemin A-C-E

->TD12

9. Arbre couvrant de poids minimal

- S'utilise uniquement sur un graphe non orienté connexe

- Graphe acyclique non orienté: Un graphe non orienté est acyclique s'il ne contient pas de cycle. C'est-à-dire que pour tous les sommets il n'existe pas de chemin de ce sommet vers lui-même contenant 2 fois la même arrête

- Généralisation de la définition d'un arbre : Un arbre est un graphe acyclique connexe

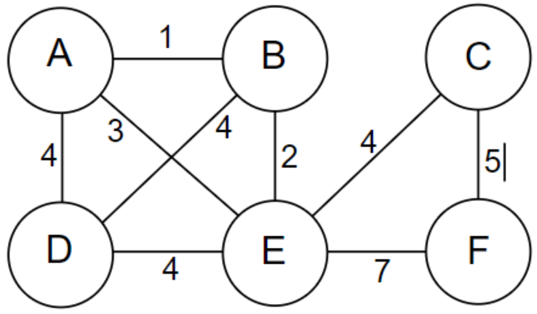
- Arbre couvrant : Dans un graphe non orienté connexe, un arbre couvrant est un sous-graphe acyclique connexe (un arbre) qui contient tous les sommets du graphe

- Arbre couvrant de poids minimal : arbre couvrant d'un graphe minimisant la somme des poids ses branches

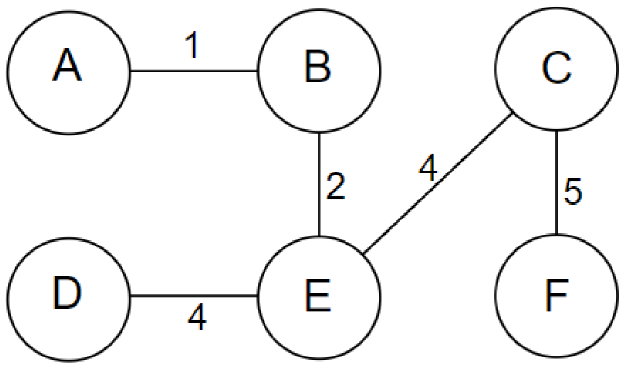
- Exemple d'usage : dans le cas de conception de réseau électrique/informatique, si le poids représente le coup de construction d'une liaison, calculer l'arbre courant de poids minimale permet de déterminer le réseau à mettre en œuvre pour minimiser le cout de ce réseau.
- Aussi utilisé dans les switchs au niveau 2 (spanning tree protocol)

9. Arbre couvrant de poids minimal

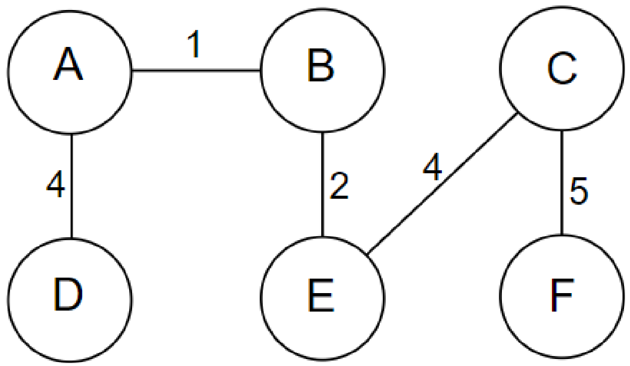
- Exemple sur le graphe suivant



- On peux par exemple extraire les 2 arbres couvrant de poids minimal suivants :



Sommes des poids : 16



Sommes des poids : 16

9. Arbre couvrant de poids minimal

<u>Comment calculer un arbre couvrant de poids minimal ?</u>

- Naïvement / de manière empirique -> complexité risquée comprise entre 1 et n^{n-2} si n est le nombre de cycle du graphe
- Utilisation de l'algorithme de Prim (complexité maximale en v^2) (v est le nombre d'arrête)
- Utilisation de l'algorithme de Kruskal (complexité maximale en $v\log(v)$) (v est le nombre d'arrête)

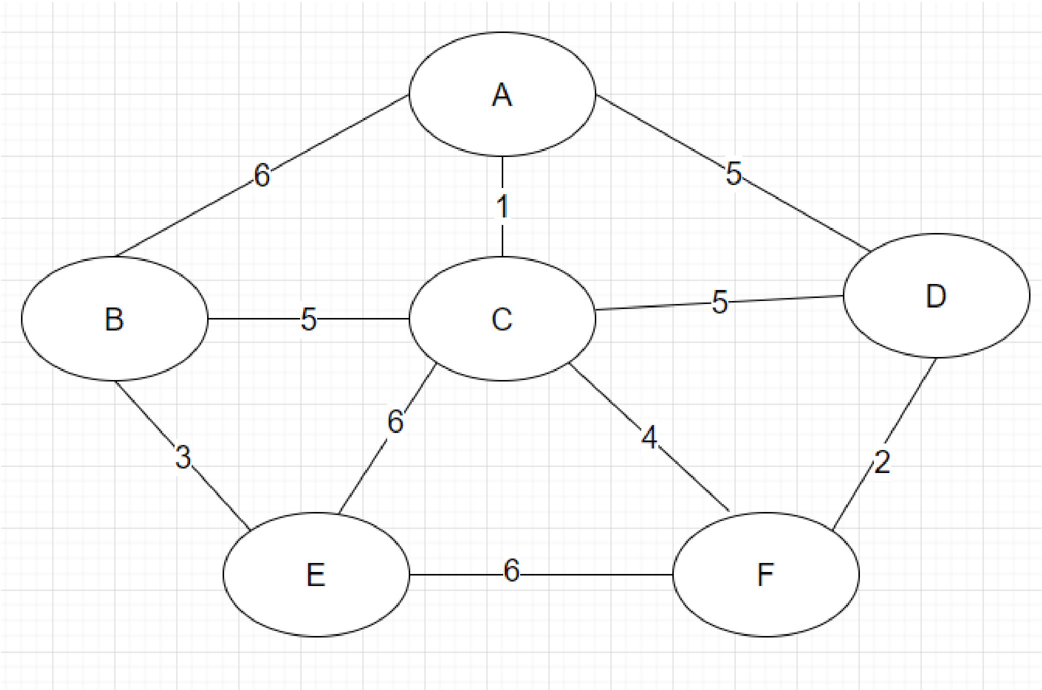
9.1 Algorithme de Prim

<u>Algorithme de Prim :</u>

- Choisir un sommet de départ et le placer dans l’arbre de résultat
- Tant qu’il ne reste pas de sommet à placer dans l’arbre de résultat :
    - Identifier les arrêtes adjacentes à notre arbre de résultat dont le sommet de destination n’est pas un sommer déjà présent dans l’arbre
    - Ajouter à notre arbre l’arrête avec le plus petit poids

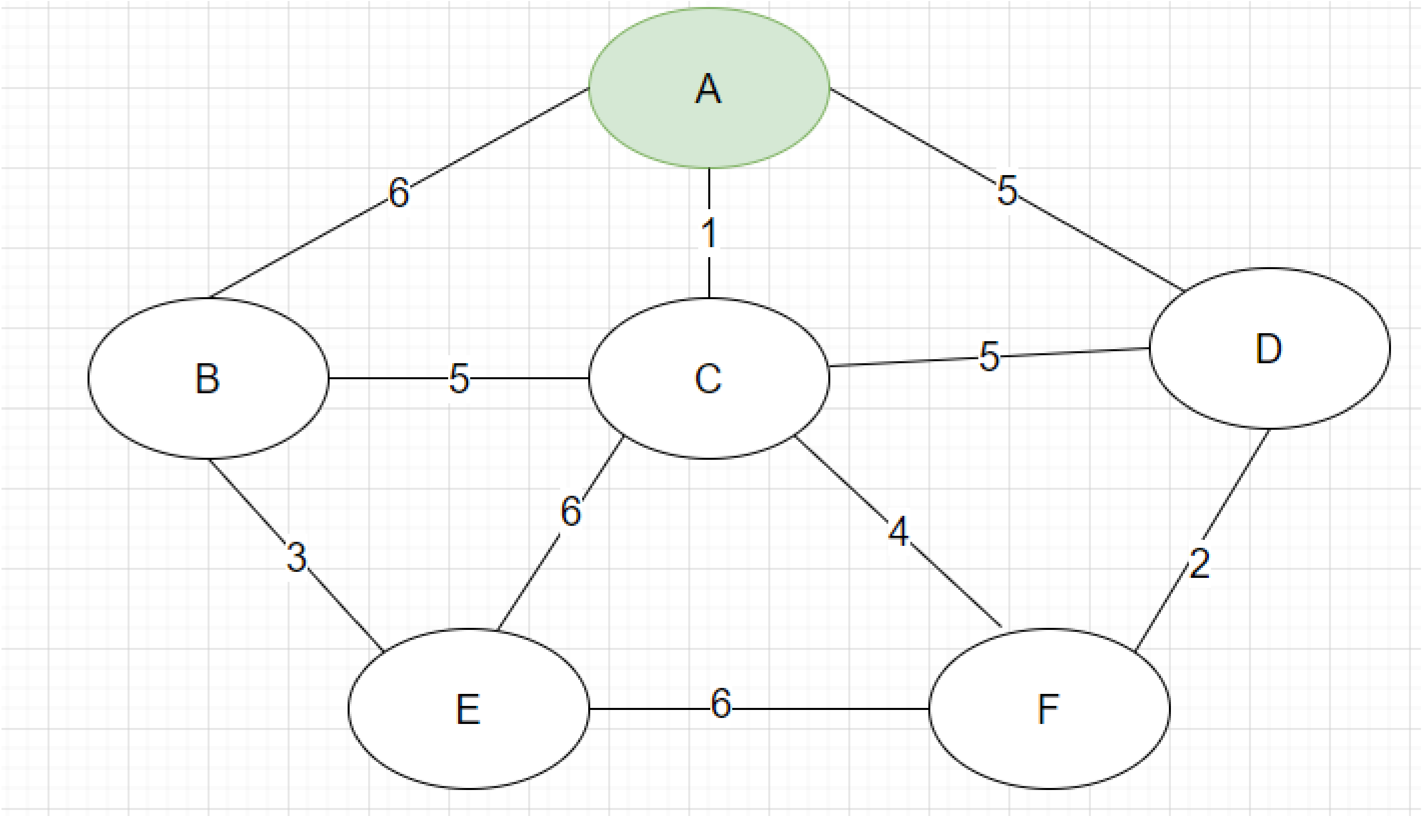
9.1 Algorithme de Prim

<u>Exemple de mise en œuvre sur ce graphe</u>



9.1 Algorithme de Prim

Exemple de mise en œuvre sur ce graphe



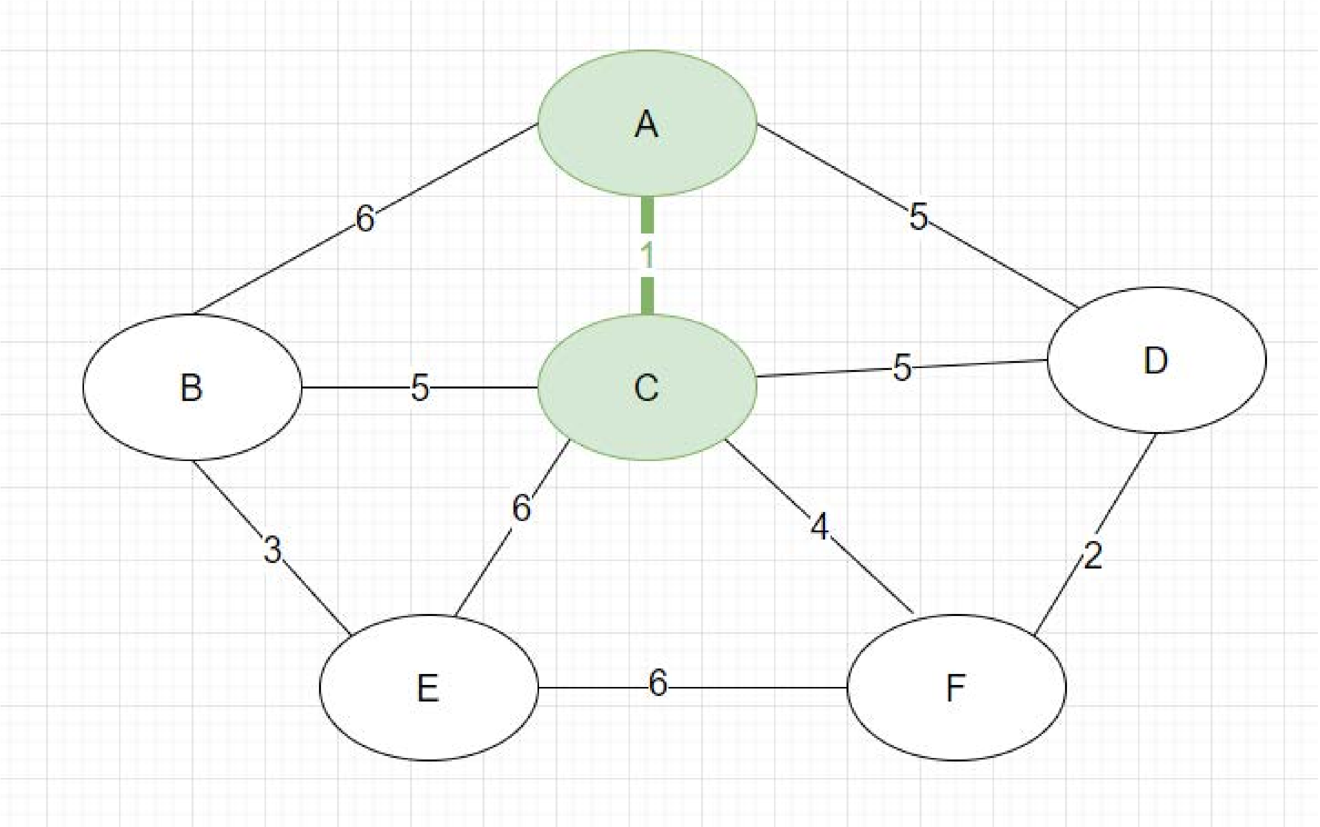
Nous partons arbitrairement du nœud A et le plaçons dans l'arbre résultat
Les arrêtes adjacentes à l'arbre en cours de constructions dont le sommet de destination n'est pas un sommet de l'arbre sont :

- A-B de poids 6
- A-C de poids 1
- A-D de poids 5

L'arrête de plus petit poids est donc l'arrête A-C

9.1 Algorithme de Prim

<u>Exemple de mise en œuvre sur ce graphe</u>



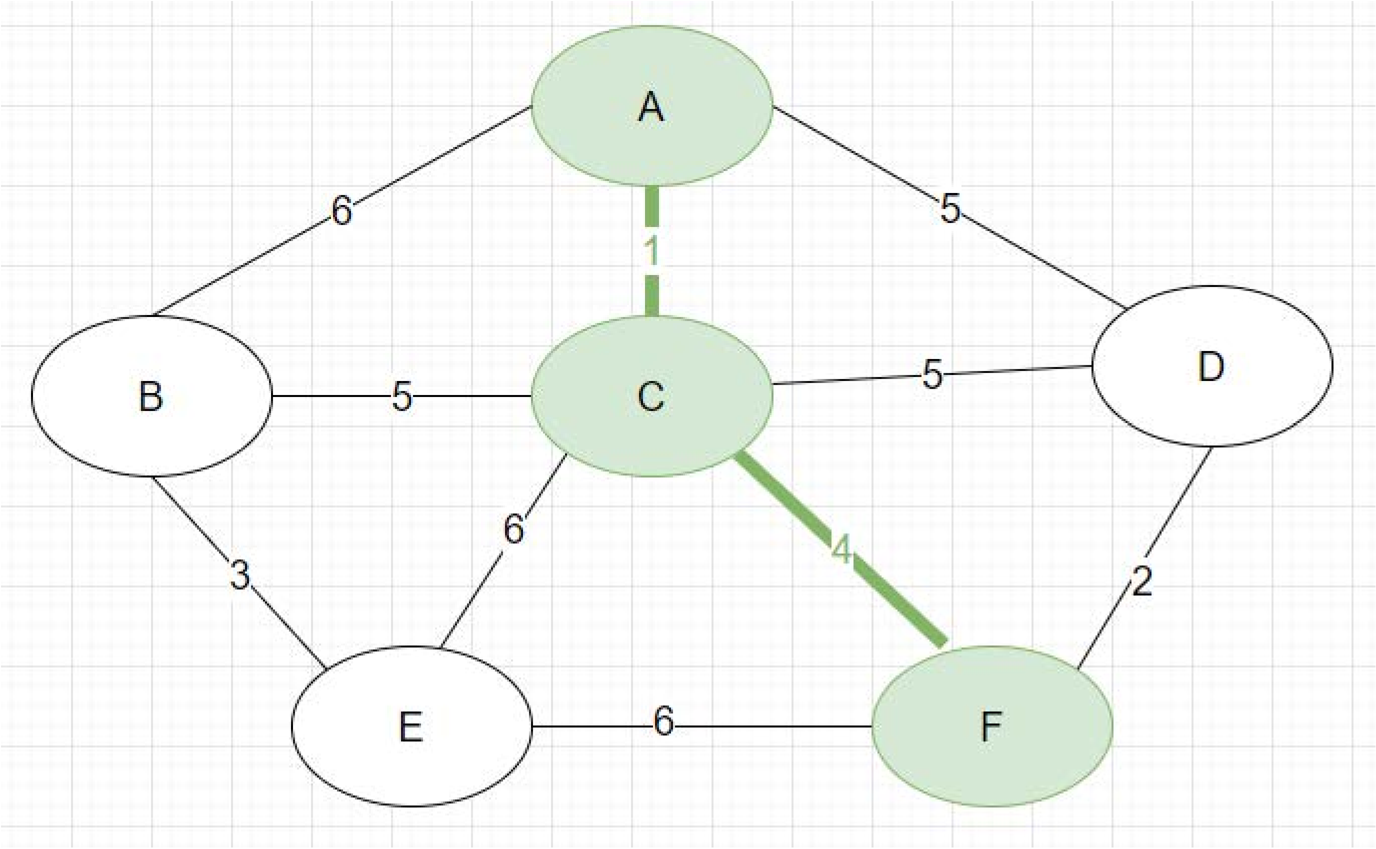
On ajoute donc l'arrête A-C de poids 1 avec le sommet associé
Les arrêtes adjacentes à l'arbre en cours de constructions dont le sommet de destination n'est pas un sommet de l'arbre sont :

- A-B de poids 6
- C-B de poids 5
- C-E de poids 6
- C-F de poids 4
- C-D de poids 5
- A-D de poids 5

L'arrête avec le plus petit poids est donc l'arrête C-F avec un poids de 4

9.1 Algorithme de Prim

<u>Exemple de mise en œuvre sur ce graphe</u>



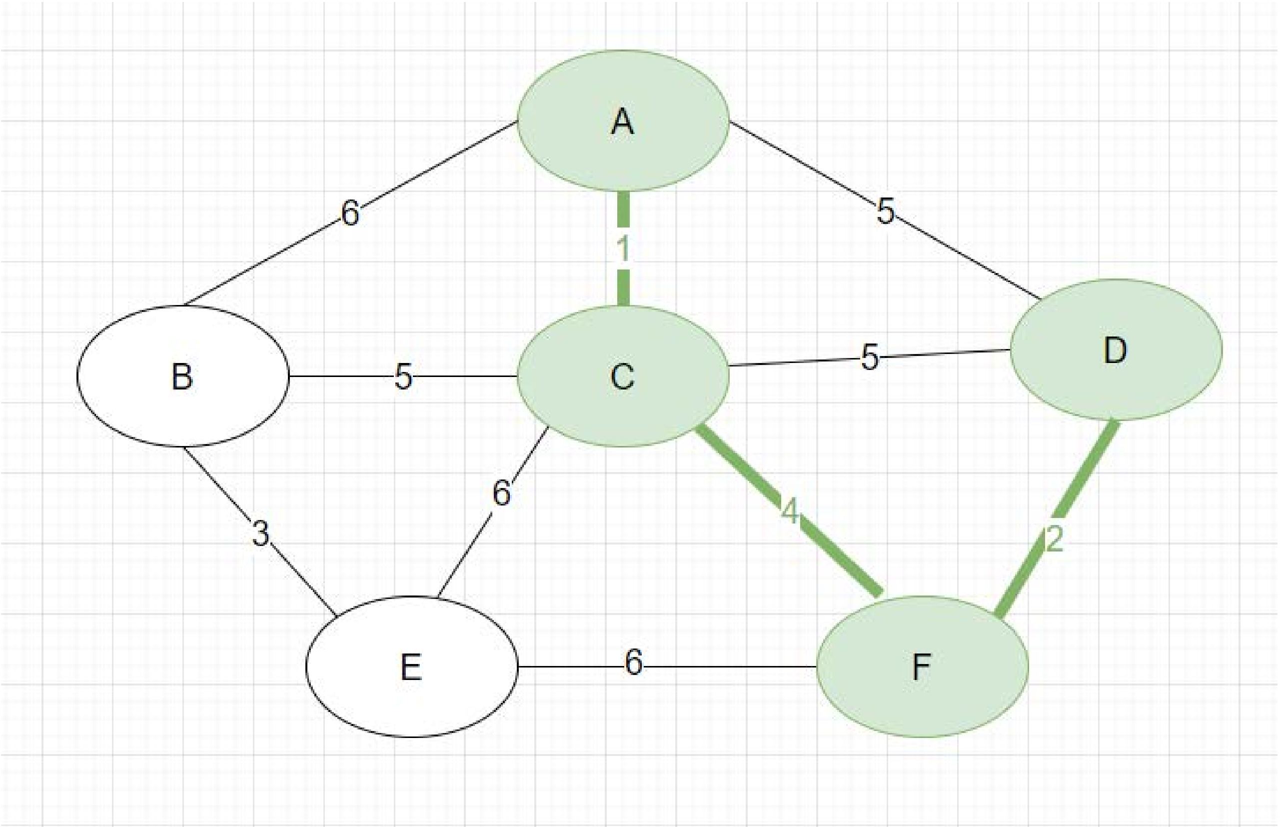
On ajoute donc l'arrête C-F de poids 4 avec le sommet associé
Les arrêtes adjacentes à l'arbre en cours de constructions dont le sommet de destination n'est pas un sommet de l'arbre sont :

- A-B de poids 6
- C-B de poids 5
- C-E de poids 6
- F-E de poids 6
- F-D de poids 2
- C-D de poids 5
- A-D de poids 5

L'arrête avec le plus petit poids est donc l'arrête F-D avec un poids de 2

9.1 Algorithme de Prim

<u>Exemple de mise en œuvre sur ce graphe</u>

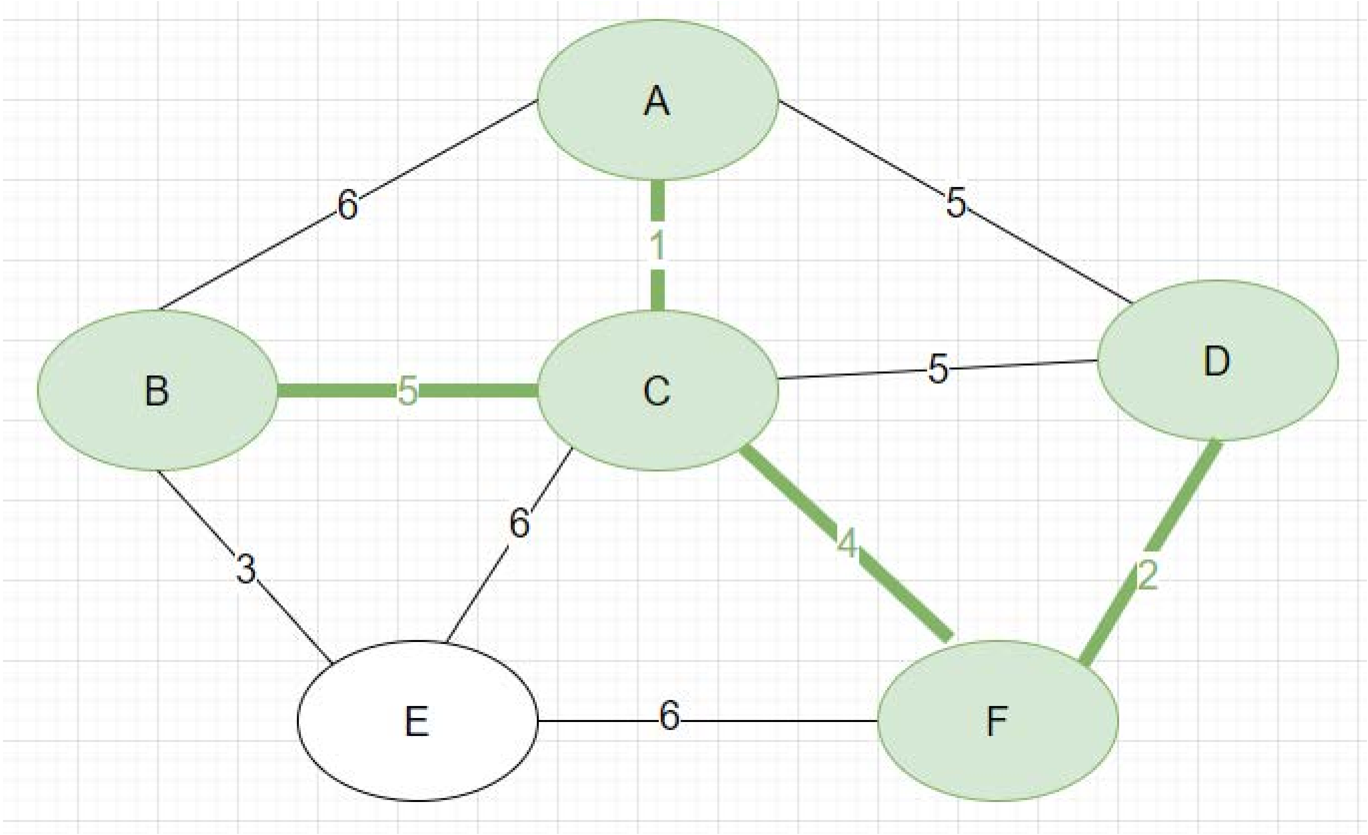


On ajoute donc l'arrête F-D de poids 2 avec le sommet associé
Les arrêtes adjacentes à l'arbre en cours de constructions dont le sommet de destination n'est pas un sommet de l'arbre sont :
- A-B de poids 6
- C-B de poids 5
- C-E de poids 6
- F-E de poids 6

L'arrête avec le plus petit poids est donc l'arrête C-B avec un poids de 5

9.1 Algorithme de Prim

<u>Exemple de mise en œuvre sur ce graphe</u>



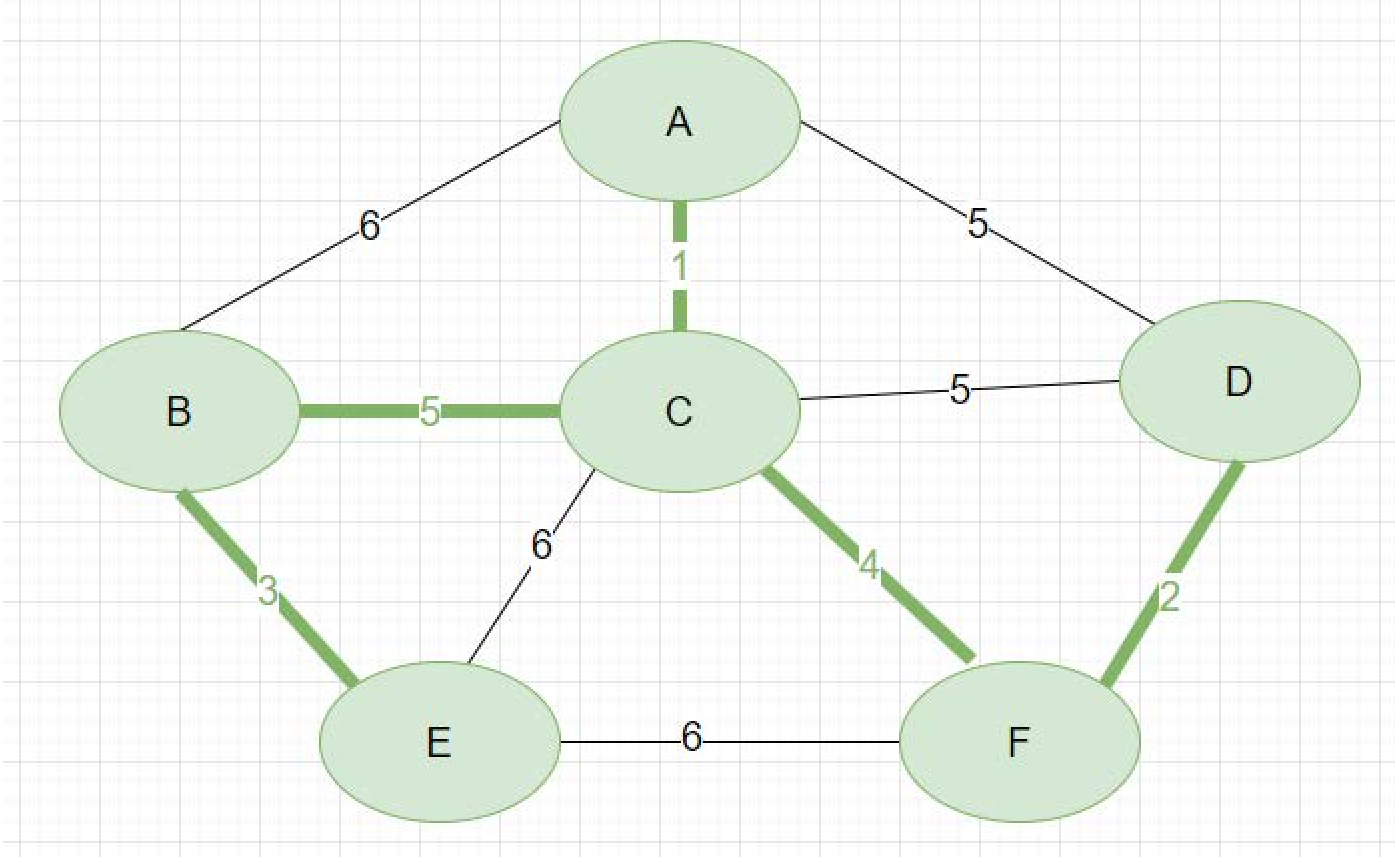
On ajoute donc l'arrête C-B de poids 5 avec le sommet associé
Les arrêtes adjacentes à l'arbre en cours de constructions dont le sommet de destination n'est pas un sommet de l'arbre sont :

- B-E de poids 3
- C-E de poids 6
- F-E de poids 6

L'arrête avec le plus petit poids est donc l'arrête B-E avec un poids de 3

9.1 Algorithme de Prim

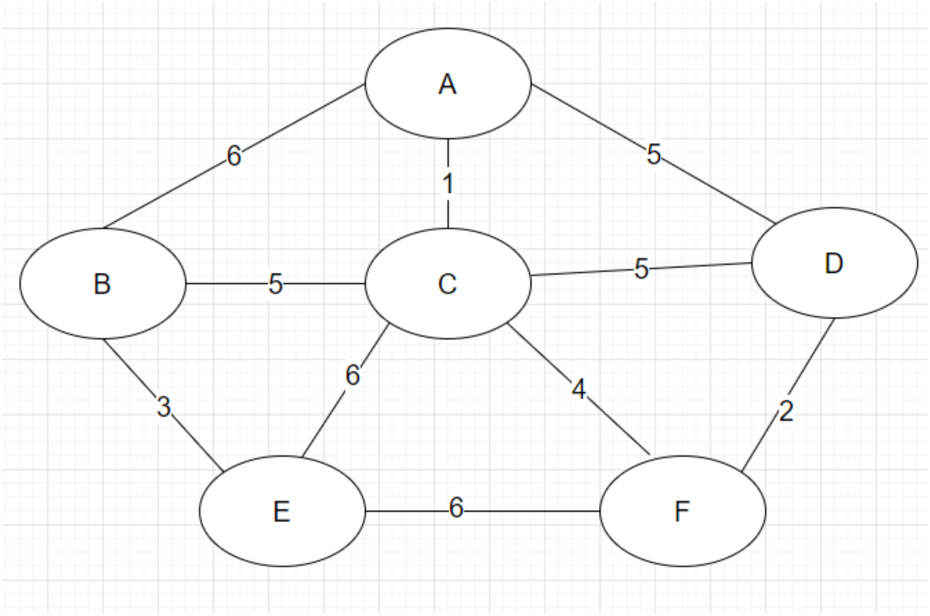
<u>Exemple de mise en œuvre sur ce graphe</u>



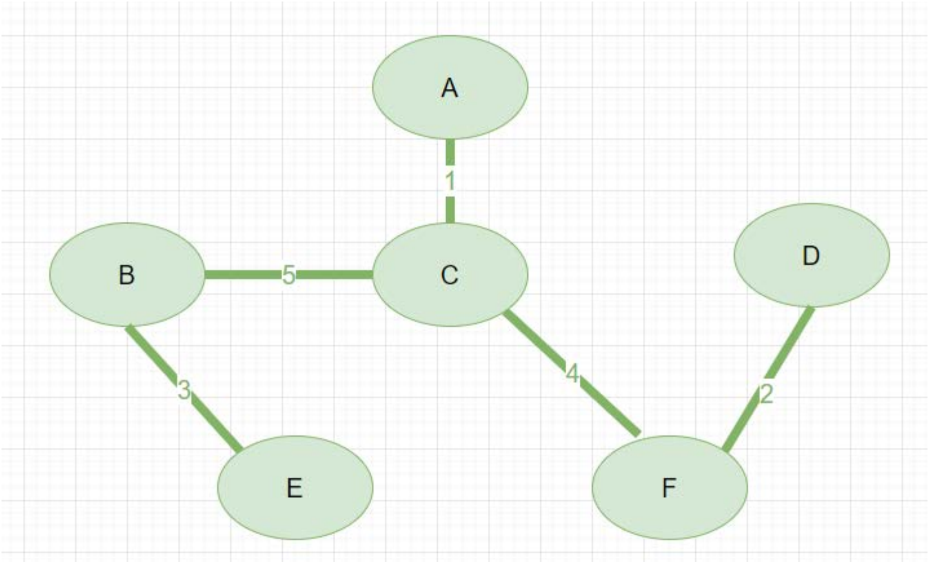
On ajoute donc l'arrête B-E de poids 3 avec le sommet associé
Il n'y a plus de sommets à ajouter, l'algorithme est donc terminé

9.1 Algorithme de Prim

<u>Exemple de mise en œuvre sur ce graphe</u>



On obtient donc cet arbre couvrant minimal



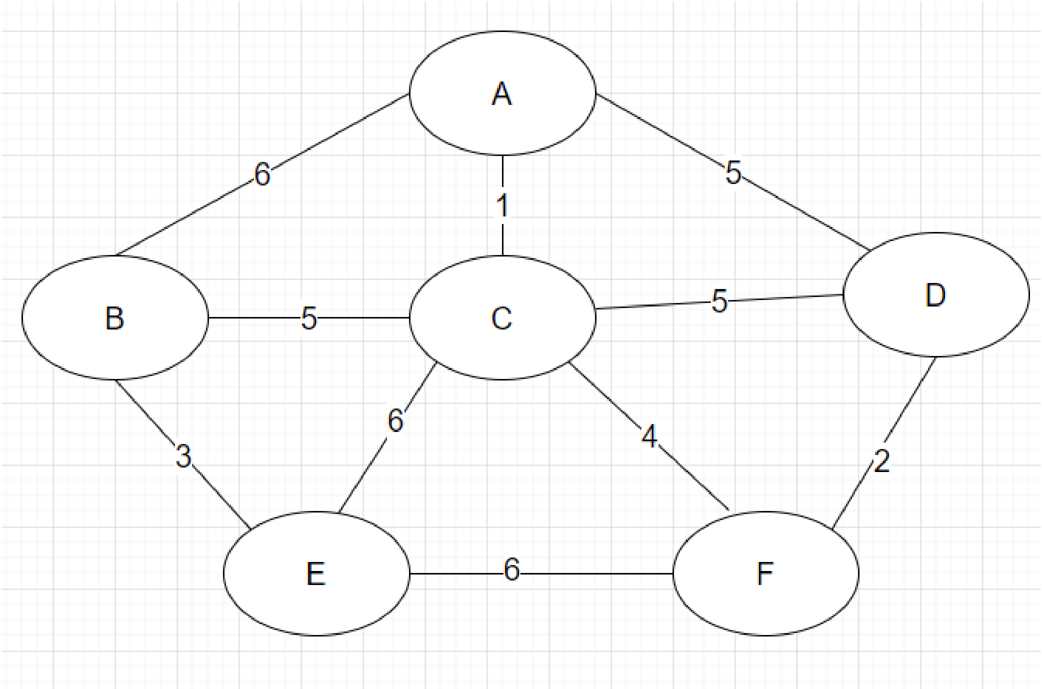
9.2 Algorithme de Kruskal

<u>Algorithme de Kruskal :</u>

- Faire l’inventaire des arrêtes du graphe avec leur poids associés
- Trier cette liste d’arrête par ordre croissant de poids
- Parcourir chaque arrête de cette liste et effectuer le traitement suivant :
    - Si l’ajout de l’arrête et de ses sommets associés manquant ne créé pas de cycles dans l’arbre couvrant ainsi constitué, l’ajouter

9.2 Algorithme de Kruskal

<u>Exemple de mise en oeuvre:</u>

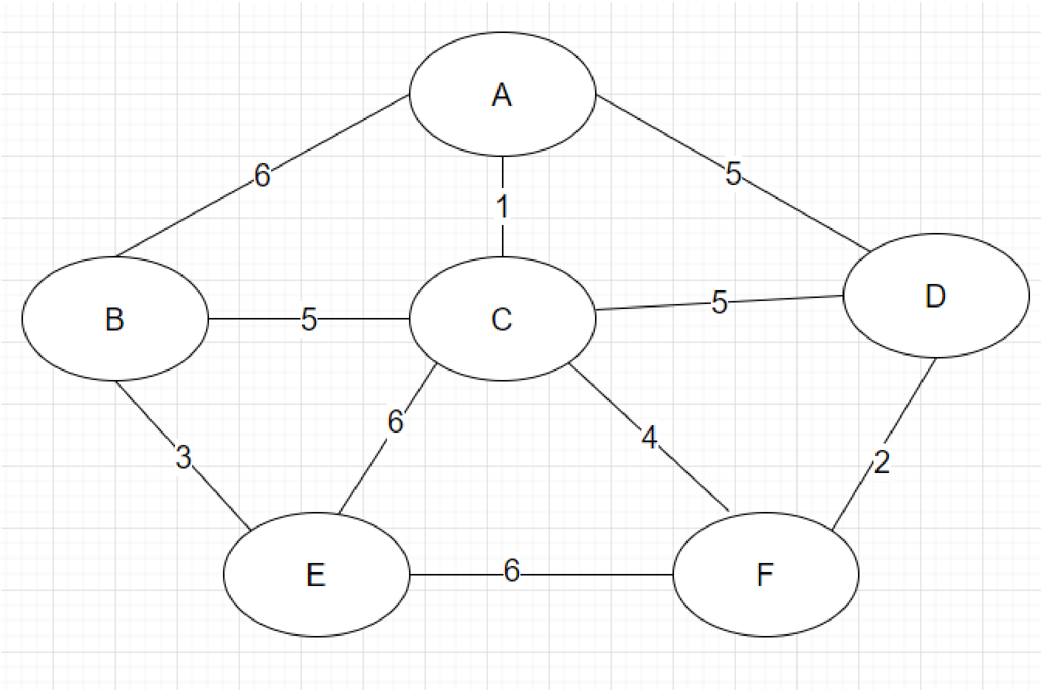


On liste les arrêtes de ce graphes avec leur poids

- A-B : 6
- A-C : 1
- A-D : 5
- B-C : 5
- C-D : 5
- B-E : 3
- C-E : 6
- C-F : 4
- D-F : 2
- E-F : 6

9.2 Algorithme de Kruskal

<u>Exemple de mise en oeuvre:</u>

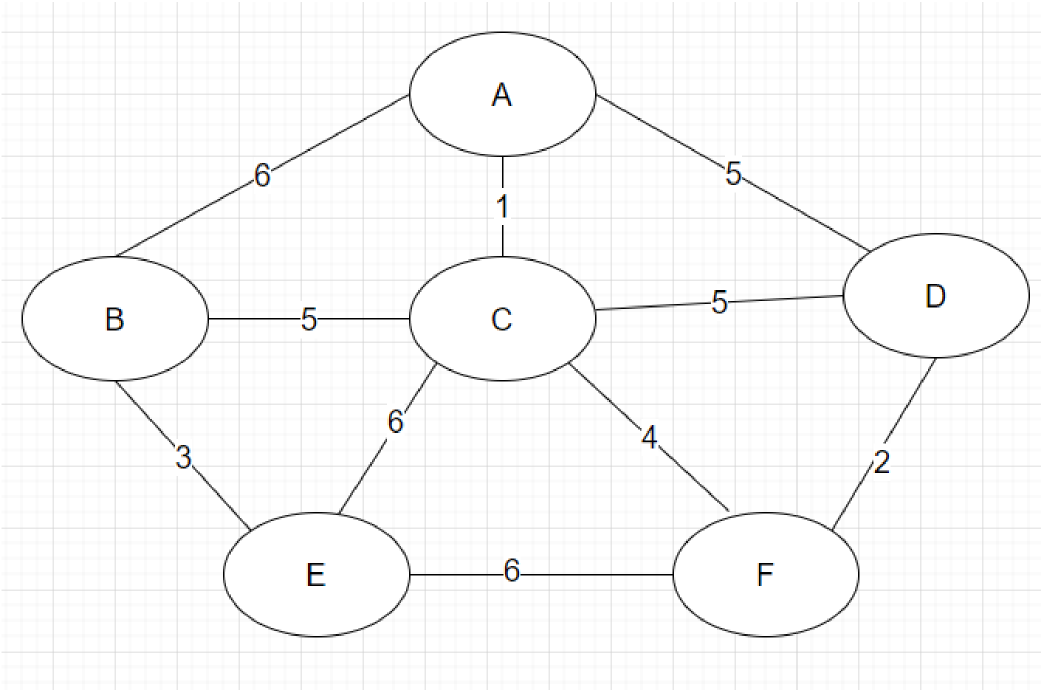


On les trie par ordre croissant

- A-C : 1
- D-F : 2
- B-E : 3
- C-F : 4
- A-D : 5
- B-C : 5
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

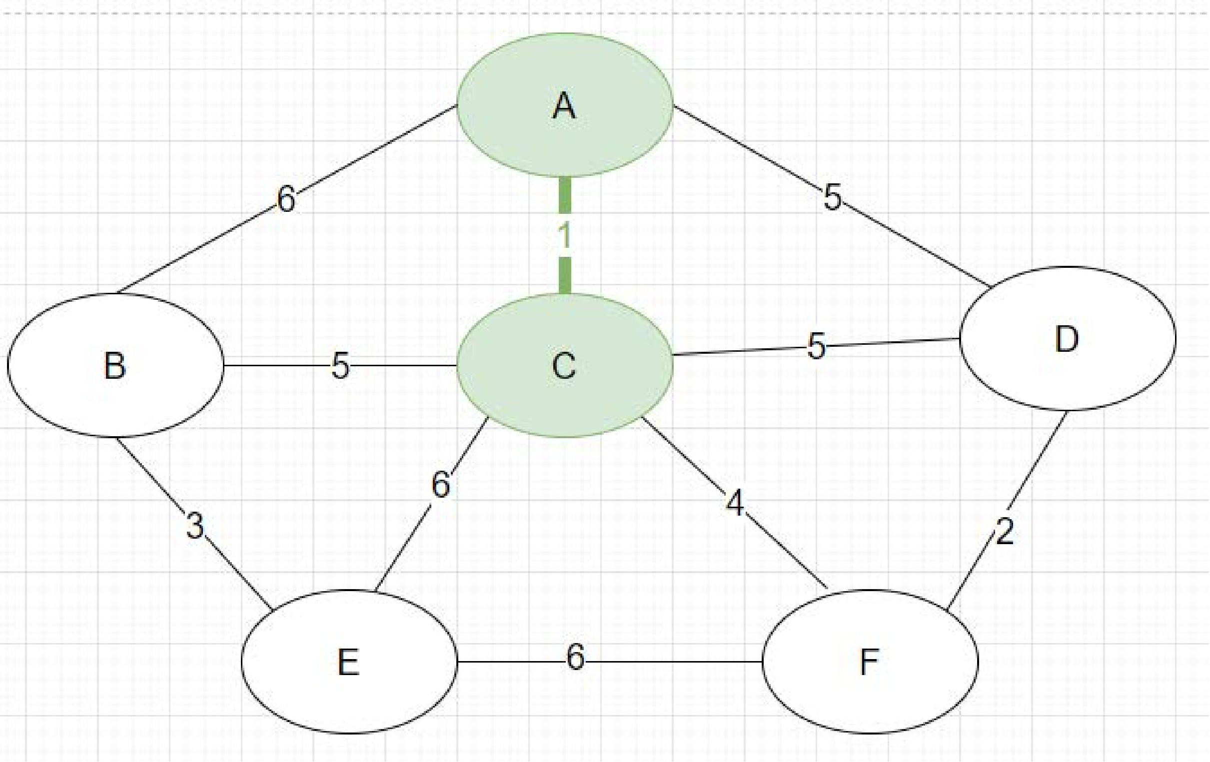


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- A-C : 1
- D-F : 2
- B-E : 3
- C-F : 4
- A-D : 5
- B-C : 5
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

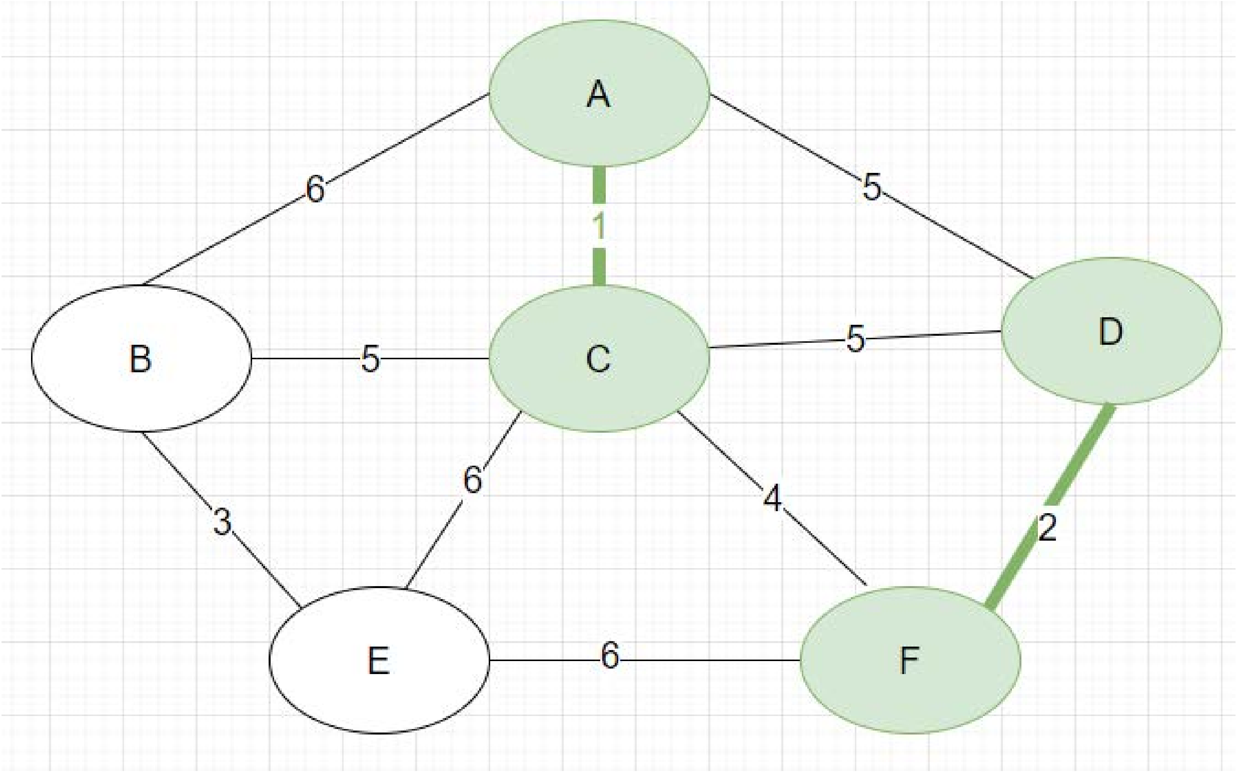


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- ~~A-C : 1~~
- D-F : 2
- B-E : 3
- C-F : 4
- A-D : 5
- B-C : 5
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

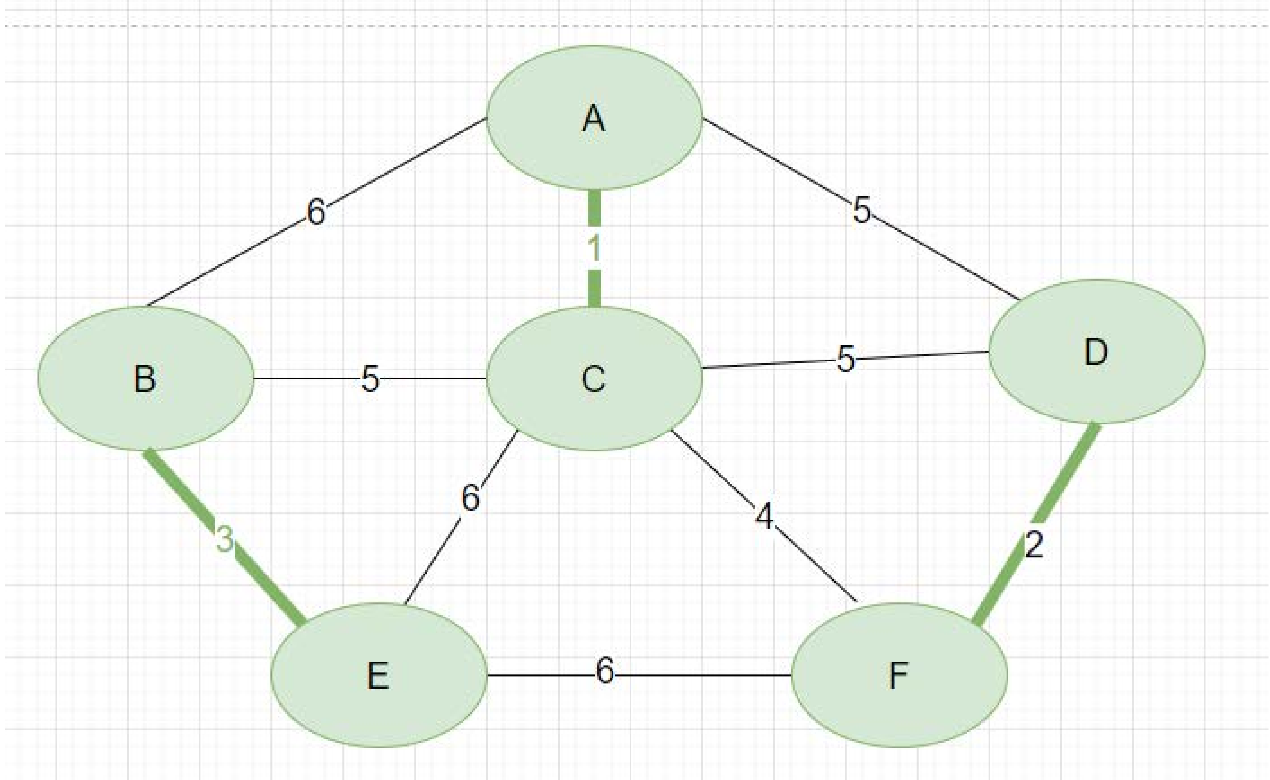


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l’ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- B-E : 3
- C-F : 4
- A-D : 5
- B-C : 5
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

<u>Exemple de mise en oeuvre:</u>

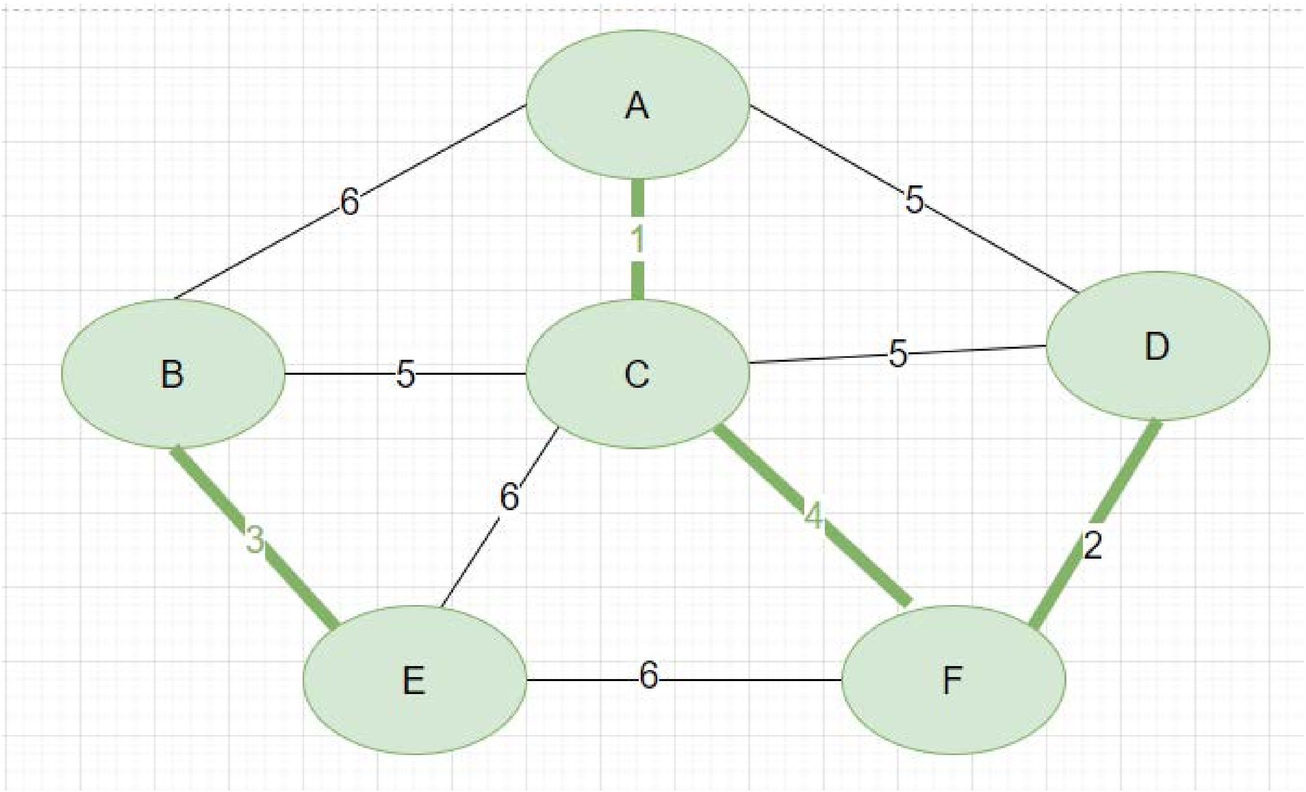


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- C-F : 4
- A-D : 5
- B-C : 5
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

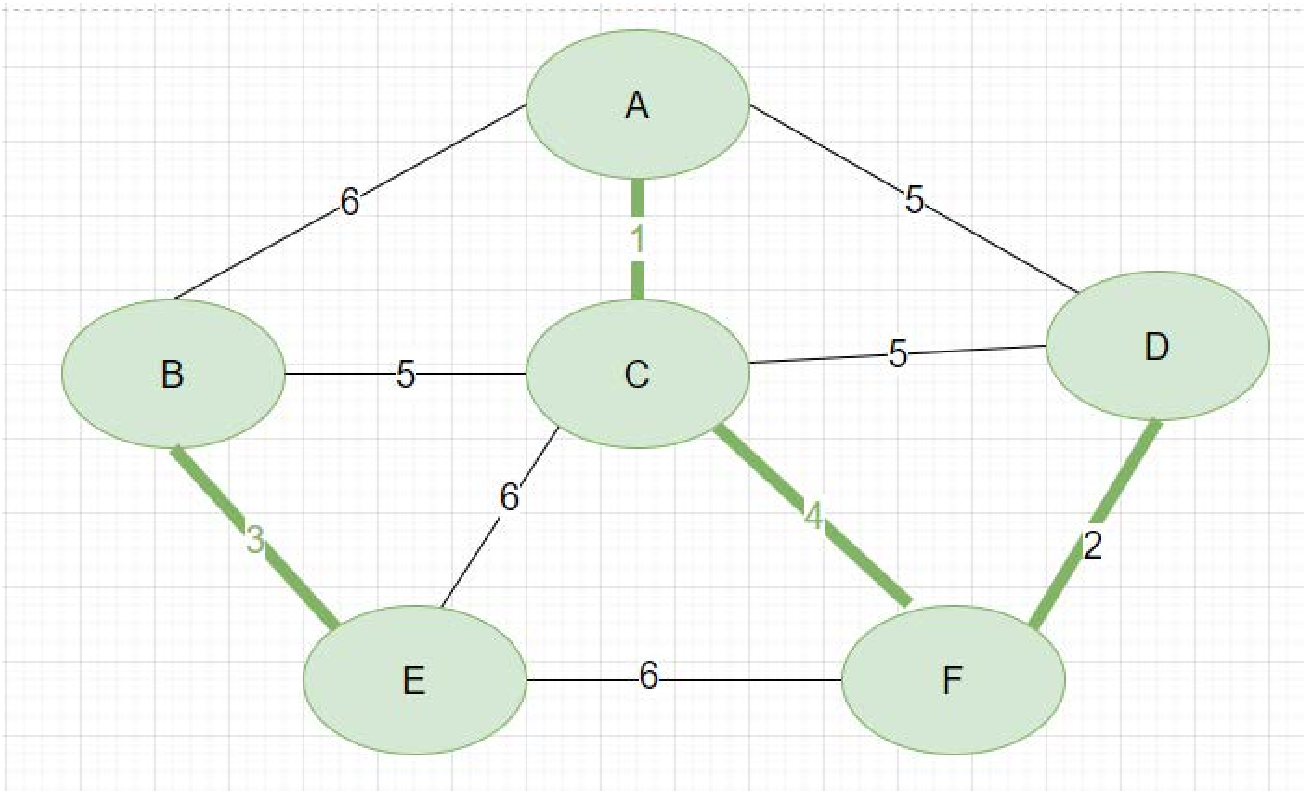


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- ~~C-F : 4~~
- A-D : 5
- B-C : 5
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

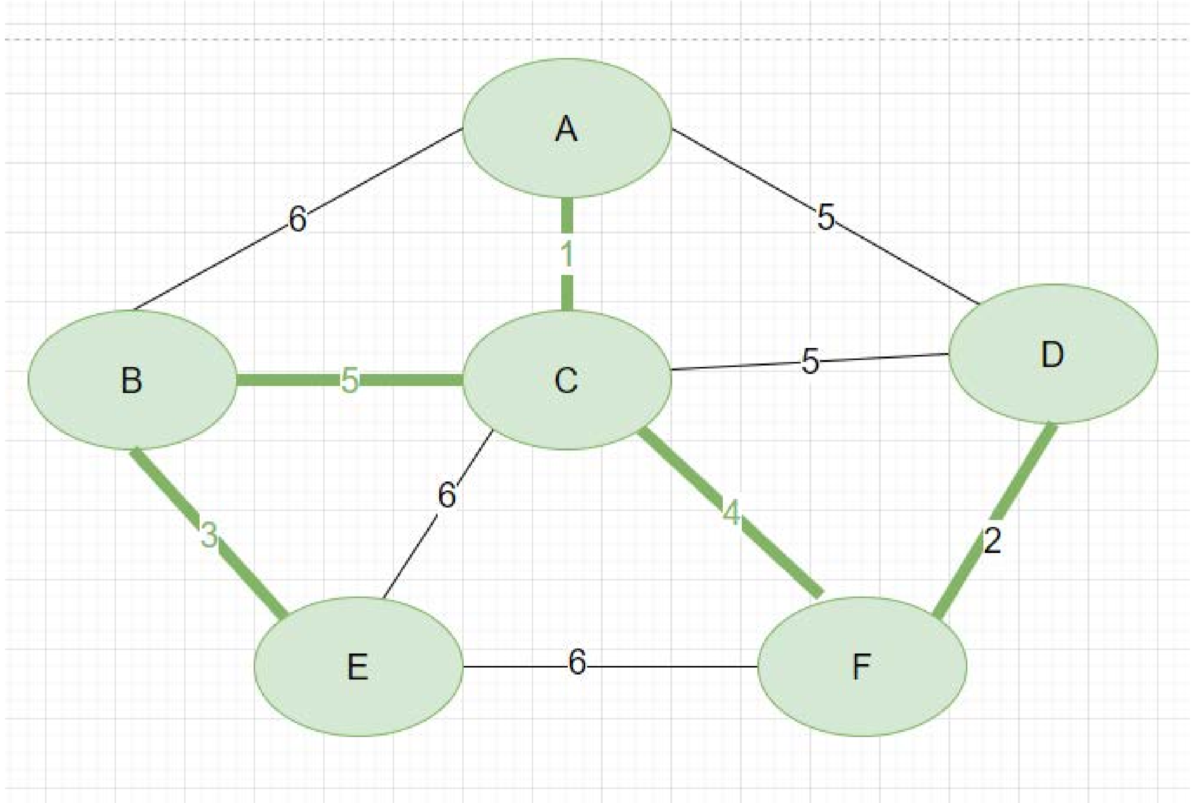


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- ~~C-F : 4~~
- A-D : 5 – non ajouté car créer un cycle si ajouter
- B-C : 5
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

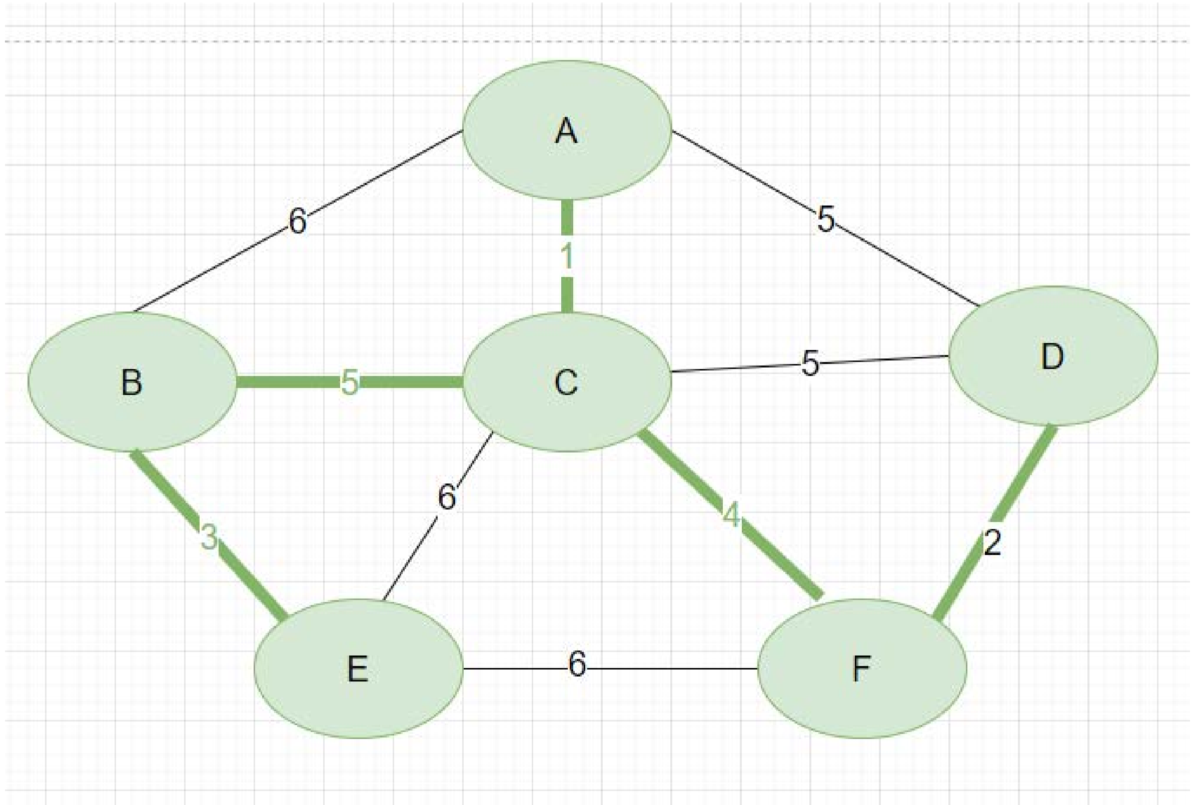


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- ~~C-F : 4~~
- A-D : 5 – non ajouté car créer un cycle si ajouter
- ~~B-C : 5~~
- C-D : 5
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

<u>Exemple de mise en oeuvre:</u>

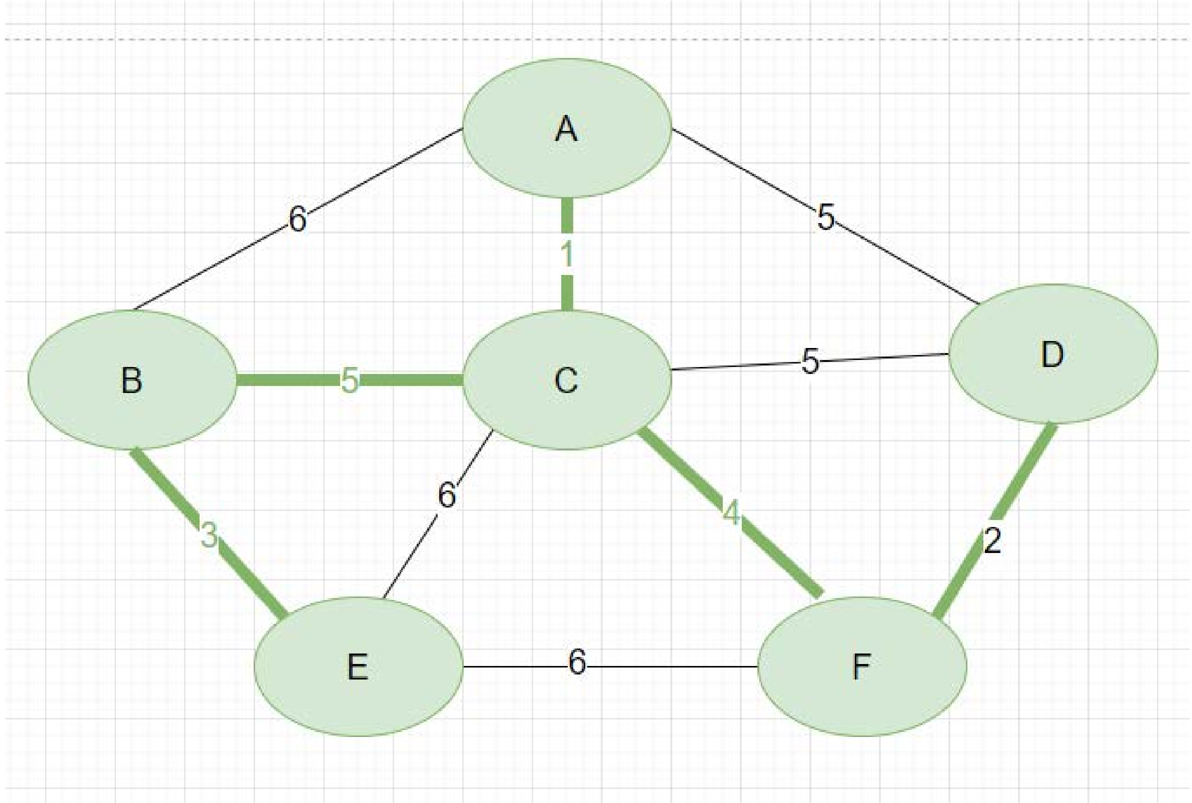


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l’ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- ~~C-F : 4~~
- A-D : 5 – non ajouté car créer un cycle si ajouter
- ~~B-C : 5~~
- C-D : 5 – non ajouté car créer un cycle si ajouter
- C-E : 6
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

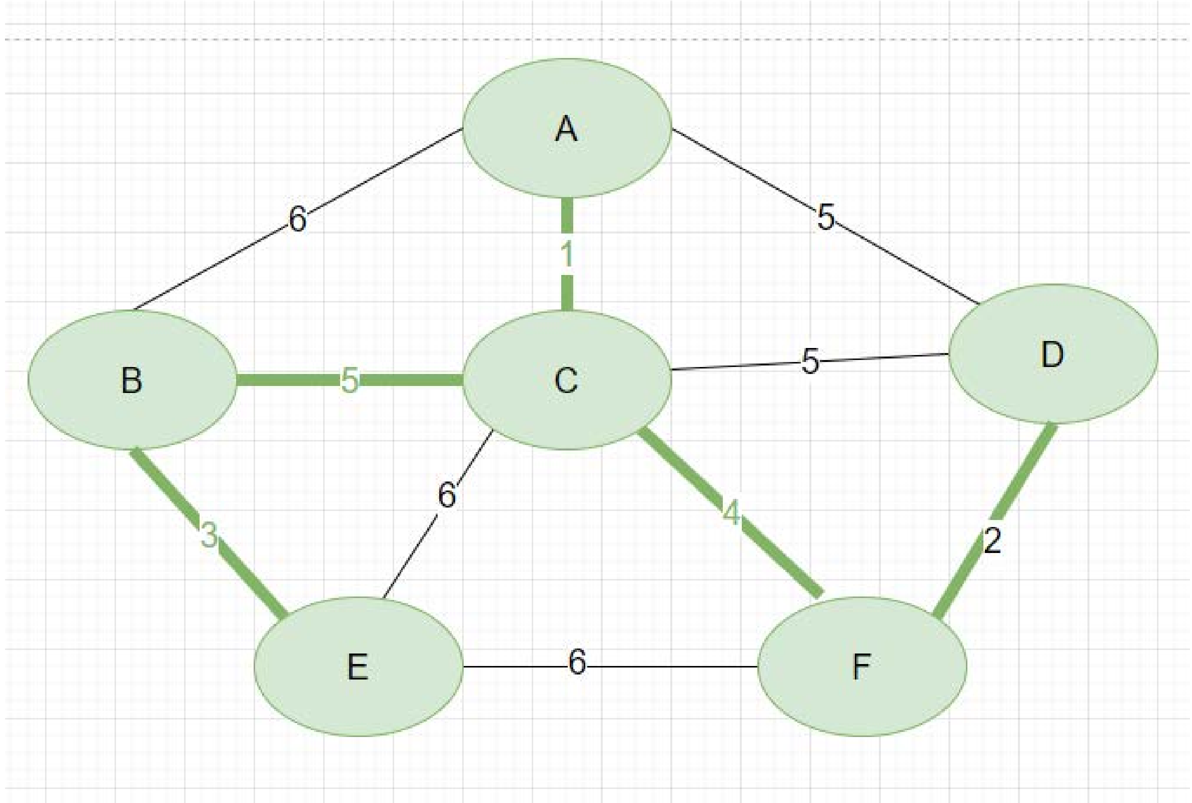


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- ~~C-F : 4~~
- A-D : 5 – non ajouté car créer un cycle si ajouter
- ~~B-C : 5~~
- C-D : 5 – non ajouté car créer un cycle si ajouter
- C-E : 6 – non ajouté car créer un cycle si ajouter
- E-F : 6
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

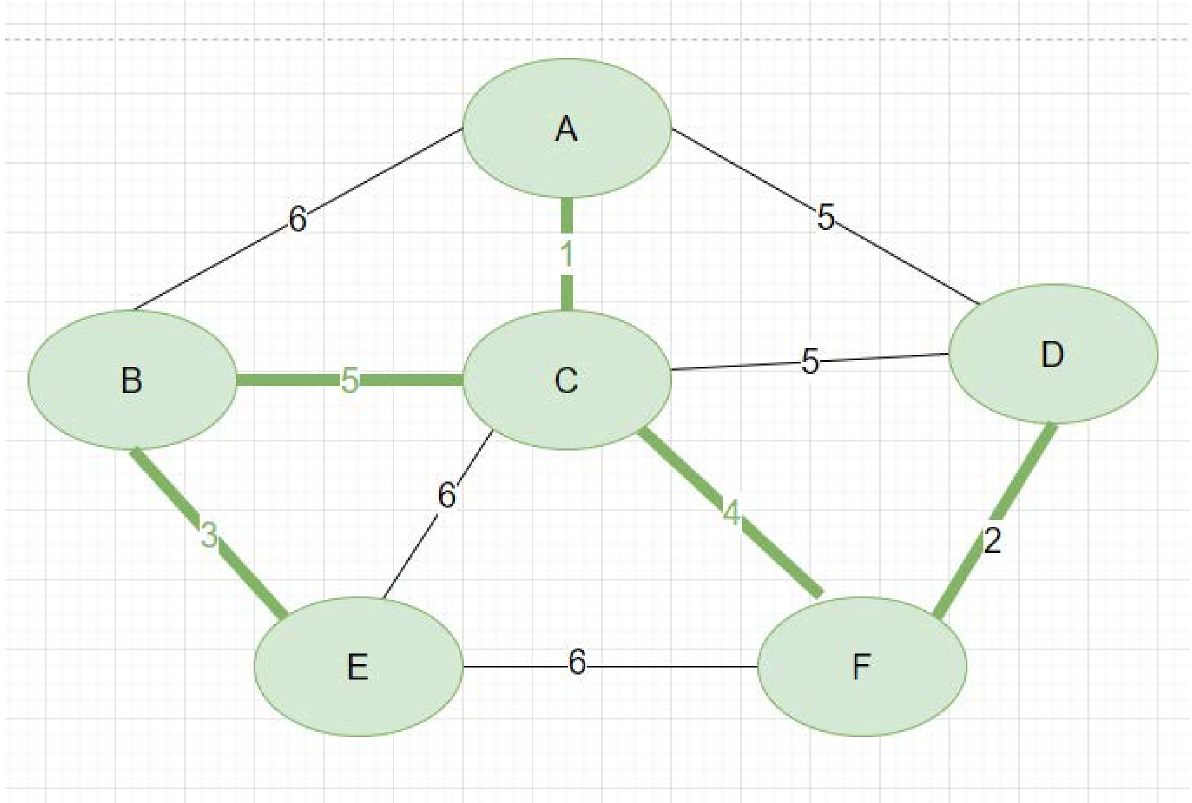


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l'ordre ci-dessous

- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- ~~C-F : 4~~
- A-D : 5 – non ajouté car créer un cycle si ajouter
- ~~B-C : 5~~
- C-D : 5 – non ajouté car créer un cycle si ajouter
- C-E : 6 – non ajouté car créer un cycle si ajouter
- E-F : 6 – non ajouté car créer un cycle si ajouter
- A-B : 6

9.2 Algorithme de Kruskal

Exemple de mise en oeuvre:

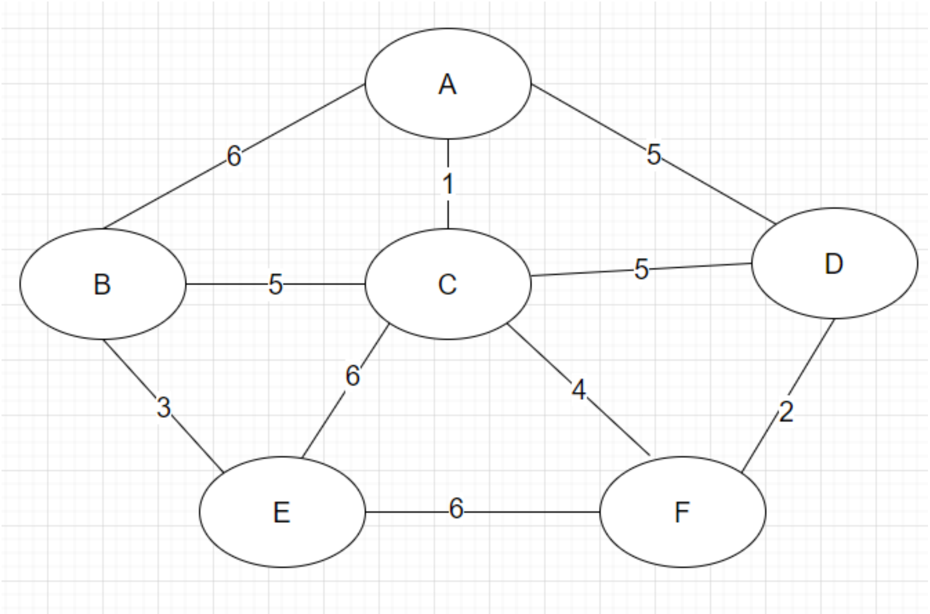


On ajoute, si cela ne créé pas de cycle, les arrêtes les unes après les autres dans l’ordre ci-dessous

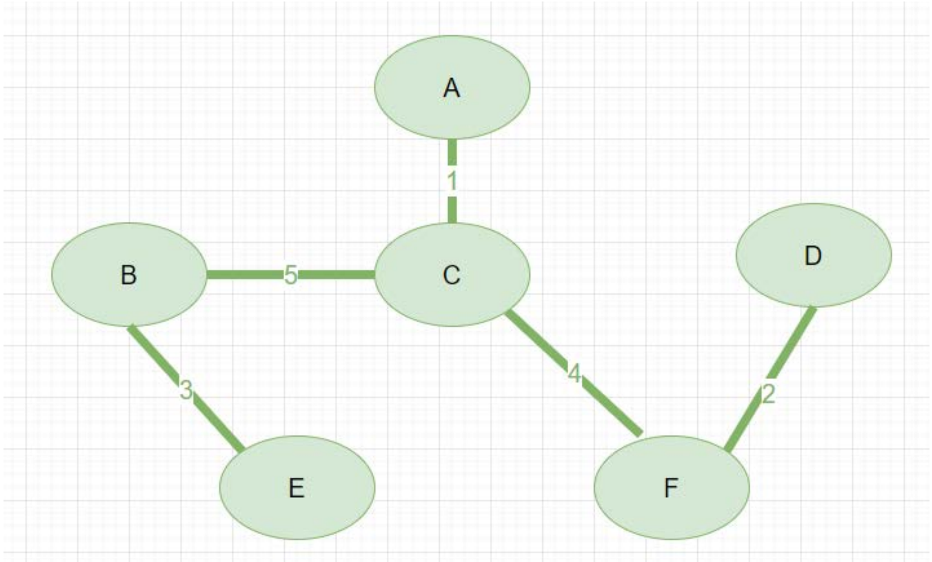
- ~~A-C : 1~~
- ~~D-F : 2~~
- ~~B-E : 3~~
- ~~C-F : 4~~
- A-D : 5 – non ajouté car créer un cycle si ajouter
- ~~B-C : 5~~
- C-D : 5 – non ajouté car créer un cycle si ajouter
- C-E : 6 – non ajouté car créer un cycle si ajouter
- E-F : 6 – non ajouté car créer un cycle si ajouter
- A-B : 6 – non ajouté car créer un cycle si ajouter

9.2 Algorithme de Kruskal

Exemple de mise en œuvre sur ce graphe



On obtient donc cet arbre couvrant minimal



->TD13