République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A. Mira de Béjaia

Faculté des Sciences Exactes

Département de Recherche Opérationnelle

Mémoire préparé

En Vue de l'Obtention du Diplôme de Master en

Recherche Opérationnelle

Spécialité : Modélisation Mathématique et Evaluation de Performances Des Réseaux

Thème

Arbres Binaires et Application dans les Réseaux

Présenté par :

M[r] SAIDI Wassim

M[r] GACI Yacine

Soutenu devant le jury :

M[r] KABYL	Kamel	M.C.B	Encadreur
M[r] TALEM	Djamel	M.A.A	Président
M[r] SOUFIT	Massinissa	Doctorant	Examinateur

Année universitaire 2016/2017

Remerciments

Nous souhaitons remercier notre promoteur, monsieur : KABYL Kamel pour son encadrement et ses conseil précieux tout au long de notre travail, avec sa grande patience intarissable durant la réalisation de ce travail de fin d 'études.

Nous souhaitons exprimer notre gratitude aux membres du jury pour avoir bien voulu consacrer une partie de leur temps à notre travail. on remercie Mr. TALEM Djamel et Mr. SOUFIT Massinissa d'avoir accepté de faire partie du jury de notre thèse.

Nous voudrions accorder une place d'honneur dans nos remerciements à nos famille, plus particulièrement, nos parents et nos sœurs et frères. Leur soutien et encouragement n'a jamais failli au cours de ce long cursus universitaire.

En fin, on remercie tous nos amis pour nous avoir soutenu durant cette période de thèse.

GACI Yacine

SAIDI Wassim

Dédicace

Dédicace

On dédie ce modeste travail aux :

Personnes les plus chères à nos yeux, nos parents nos frères et sœurs

aux familles GACI et SAIDI.

Nos amis et tous les gens qui nous aiment.

Tous ceux qui sont proches de nos cœurs et dont on a pas a citer les noms.

Table des matières

Table des figures

Liste des tableaux

Introduction

La théorie des graphes constitue aujourd'hui un corpus de connaissances trés important, historiquement elle est née en 1736 avec la communication d'Euler $(1707 - 1783)$ dans laquelle il proposait une solution au célébre probléme des ponts de Königsberg (*Euler*, 1736). "Les habitants de Konigsberg se demandaient s'il était possible, en partant d'un quartier quelconque de la ville, de traverser tous les ponts sans passer deux fois par le même et de revenir à leur point de départ." **Euler** d'emontra que ce probléme n'a pas de solution. Le problème des ponts de Konigsberg est identique à celui consistant à tracer une figure géométrique sans lever le crayon et sans repasser plusieurs fois sur un même trait.
La théorie des graphes est un trés vaste domaine, en évolution constante tant du point de vue des recherches fondamentales que de celuis des applications d'autre part, les applications sont trés nombreuses. elles justifient une recherche importante en algorithmique, implémenter quelques structures de données pour vérifier si en pratique nous obtenons des résultats aussi intéressants qu'en théorie. En effet, il se peut que pour des structures trop complexes les résultats obtenus en pratique soient moins bons que ceux auxquels nous aurions pu nous attendre à avoir par la théorie.

Nous nous intéressons dans notre travail à la définition ainsi que la programmation de quelques techniques d'optimisation dans les graphes, dont le but est de résoudre un probléme réel en utilisant la théorie des graphes et plus particuliérement l'optimisation, l'étude présenté dans ce document porte essentielement sur les arbres binaire et leur application dans les réseaux.
Et pour cela, on a réparti notre mémoire en quatre chapitres, une conclusion et une introduction :
- Dans le premier chapitre, nous allons rappeler briévement quelques notion de base sur la théorie des graphes et certaines définition essentielles.
- Dans le deuxième chapitre nous nous sommes intèresser aux arbres binaires et nous avons cité quelques techniques de parcours avec des exemples.
- Dans le troisième chapitre, nous avons expliqué le principe des différents algorithmes que nous allons utiliser par la suit dans le dernier chapitre lors de notre application sur un exemple réel

- Dans le quatrième chapitre, nous nous serviront des algorithmes cités dans le chapitre précédent afin de résoudre un problème de télécommunication et un problème de tri.

1

Notations et notions de bases

On donne dans ce chapitre, les définitions classiques de la théorie des graphes ainsi que quelques notations, qui permettent á ce document d'être *auto-contenu*. On adoptera la terminologie de Berge[3]

1.1 Graphes

Définition 1.1.1 Un *graphe non orienté* G (ou plus simplement graphe) est un couple d'ensembles finis (V(G),E(G)) oú E(G) est constitué de paires (non ordonnées) de V(G). Les éléments de V(G) sont appelés *sommets* de G et ceux de E(G) *arêtes* de G. si aucune confusion n'est possible on note V au lieu de V(G) et E au lieu de E(G). On utilisera les notations simplifiées UV ou VU pour l'arête $\{u, v\}$. Une arête de type uu est appelée boucle de G. L'ordre d'un graphe est le cardinal de son ensemble de sommets, noté $|V(G)|$. Si $e = uv$ est une arête de G on dit que u et v sont voisins dans G et qui'il forment les extrémités de e. Deux arêtes sont dites adjacentes si elles ont une extrémité en commune.
On définit le *voisinage* d'un sommet u dans un graphe G comme l'ensemble de ses voisins, on le note $N_G(u)$ ou N(u) s'il n'existe pas d'ambiguité sur le graphe considéré. Le degré d'un sommet u dans G, noté $d_G(u)$ est le cardinal de $N_G(u)$, avec le maximum des degrés des sommets de G est noté par $\Delta(G)$ et le minimum des degrés des sommets de G est noté par $\delta(G)$. Un sommet de degré 0 est dit sommet pendant. Si tous les sommets ont le même degré, on dira que le graphe G est *régulier* et si $\Delta(G) = \delta(G)$, alors G est dit $\Delta(G)$-*régulier*. Un exemple d'un graphe á 5 sommets et 4 arêtes est montré dans la Figure 1.1.

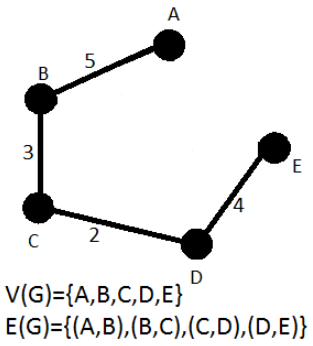


FIGURE 1.1 – Graphe valué à 5 sommets et 4 arêtes

1.1.1 Graphes comlémentaires, graphes partiels et sous-graphes

Le graphe complémentaire de G, noté $\overline{G}$, est le graphe dont l'ensemble de sommets est V(G) et deux sommets distincts de $\overline{G}$ sont adjacents si et seulement s'ils ne sont pas adjacents dans G. On dit q'un graphe H=(W(H),F(H)) est un sous-graphe, noté G[W], ayant W(H) comme ensemble de sommets et toutes les arêtes de G contenues dans W(H) comme ensemble d'arêtes est appelé sous-graphe de G *induit* par W(H). Dans le cas oú W(H)=V(G) et F(H)$\subseteq$ E(G), H=(W(H),F(H)) est appelé graphe partiel de G. (H peut être vu comme étant le graphe obtenu en supprimant certaines arêtes de G).

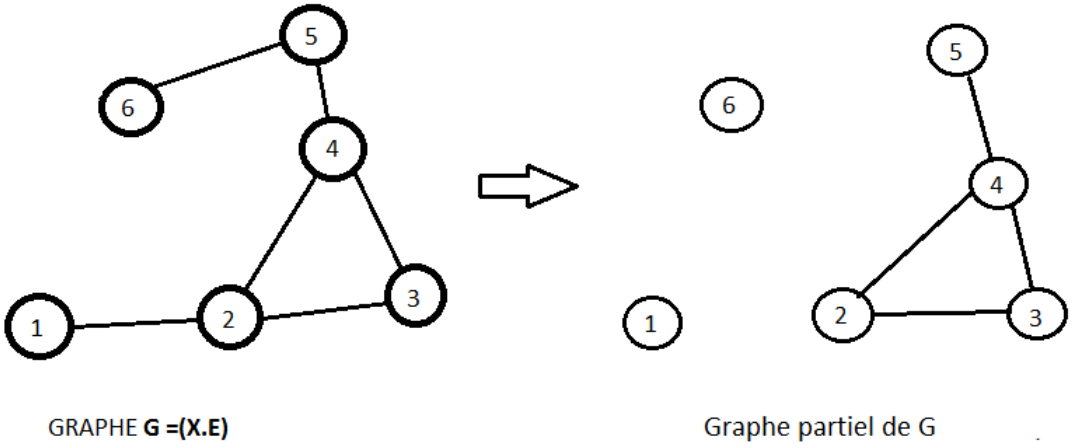


FIGURE 1.2 – Un exemples d'un graphe partiel

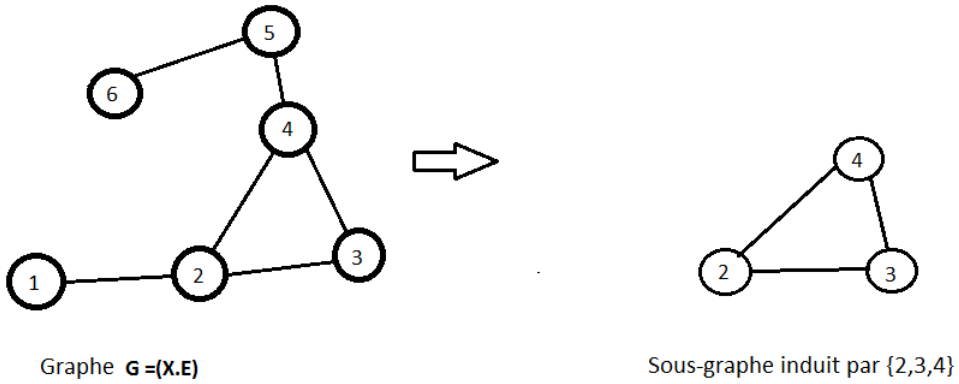


FIGURE 1.3 – Un exemples d'un sous graphe

1.1.2 Clique

Une clique est un sous-ensemble des sommets de ce graphe dont le sous-graphe induit est complet, c'est-à-dire que deux sommets quelconques de la clique sont toujours adjacents.

Le sous-graphe induit par l'ensemble $\{2, 3, 4\} montr dans la figure 1.3 est une clique.$

1.1.3 Stable

Un stable, appelé aussi ensemble indépendant ou (independent set en anglais), est un ensemble de sommets deux à deux non adjacents. La taille d'un stable est égale au nombre de sommets qu'il contient.
le graphe de la figure 1.5 admet deux stables tel que $S_1 = (1, 3, 6, 8)$ et $S_2 = (2, 5, 7, 4)$

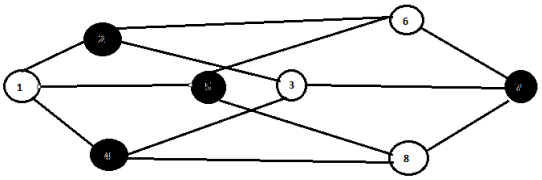


FIGURE 1.4 – Un exemples d'un stable

1.2 Connexité dans les graphes

1.2.1 Connexité

Un graphe $G = (V(G), E(G))$ est dit connexe si pour toute paire de sommets (u, v) de $V(G) \times V(G)$, il existe une chaîne reliant u et v. Si G n'est pas connexe, alors il va avoir des sous-graphes connexes, ces derniers sont appelés composantes connexes de G.

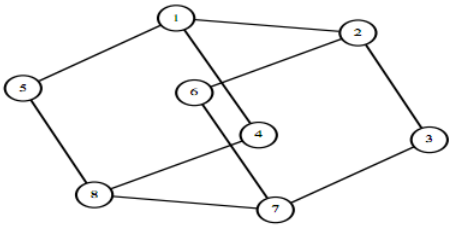


FIGURE 1.5 – Exemple d'un graphe connexe

1.2.2 Forte connexité

Définittion 1.2.1 [3] : Un graphe G est dit fortement connexe si $\forall$(x, y) $\in X^2$ il existe un chemin de x à y et un autre chemin de y à $x^{\cdot}$
Le graphe de la Figure 1.7 contient 2 composantes fortement connexes : la premiére est le sous-graphe défini par les sommets $a, b, c, d\}$, et la seconde est le sous-graphe défini par les sommets $\{e, f, g\}$.

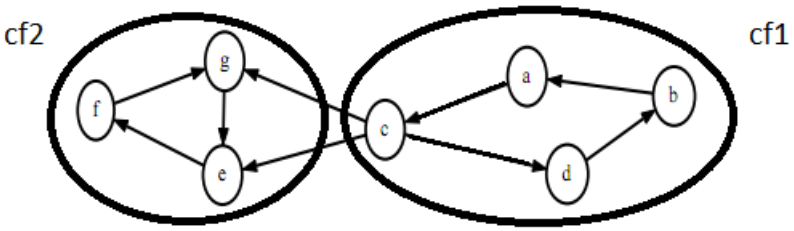


FIGURE 1.6 – Deux composantes fortement connexes

1.2.3 Graphe réduit

On appelle le graphe réduit du graphe $G = (X, E)$ le graphe $G_r = (X_r, E_r)$ tel que :

les sommets de G_r sont les composantes fortement connexes de G. S'il existe un arc (x, y) dans le graphe G avec : $x \in c.f.c_i$ et $y \in c.f.c_j$ alors il existera un arc de $c.f.c_i$ à $c.f.c_j$ dans le graphe G_r $i, j \in (1...n)$. . Le graphe de la Figure 1.8 représente le graphe réduit de l'exemple précédent

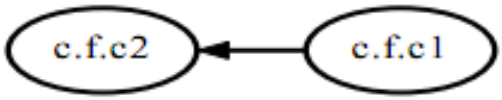


FIGURE 1.7 – Exemple d'un graphe réduit

1.2.4 Chaine et cycle

Dans un graphe non orienté, on appelle chaîne entre deux sommets u et v d'un graphe G, une suite de sommets $u_1, ..., u_k$ dont deux consécutifs sont adjacents, avec $u_1 = u$ et $u_k = v$. Une chaîne qui n'utilise pas deux fois le même sommet est dite élémentaire .
Une chaîne simple dont les éxtrémités initiale et finale sont confondues est appelée cycle.

1.2.5 Chemin et circuit

Un chemin conduisant du sommet u au sommet v est une suite ayant pour éléments alternativement des sommets et des arcs, commençant et se terminant par un sommet, et telle que chaque arc est encadré à gauche par son sommet origine et à droite par son sommet destination Une chemin dont les éxtrémités initiale et finale sont confondues est appelée circuit.

1.2.6 Graphe Eulérien

On appelle cycle eulérien d'un graphe G un cycle passant une et une seule fois par chacune des arêtes de G. Un graphe est dit eulérien s'il possède un cycle eu-

lérien. On appelle chaîne eulérienne d'un graphe G une chaîne passant une et une seule fois par chacune des arêtes de G. Un graphe ne possèdant que des chaînes eulêriennes est semieulérien. Plus simplement, on peut dire qu'un graphe est eulérien (ou semi-eulérien) s'il est possible de dessiner le graphe sans lever le crayon et sans passer deux fois sur la même arête.

1.2.7 Graphe Hamiltonien

On appelle cycle hamiltonien d'un graphe G un cycle passant une et une seule fois par chacun des sommets de G. Un graphe est dit hamiltonien s'il possède un cycle hamiltonien. On appelle chaîne hamiltonienne d'un graphe G une chaîne passant une et une seule fois par chacun des sommets de G. Un graphe ne possèdant que des chaînes hamiltoniennes est semi-hamiltonien. Contrairement aux graphes eulériens, il n'existe pas de caractérisation simple des graphes (semi-)hamiltoniens. On peut énoncer quelques propriétés et conditions suffisantes :
- un graphe possèdant un sommet de degré 1 ne peut pas être hamiltonien ;
- si un sommet dans un graphe est de degré 2, alors les deux arêtes incidentes à ce sommet doivent faire partie du cycle hamiltonien.

1.3 Opérations sur les graphes

1.3.1 Somme cartésienne de deux graphes

Étant donné deux graphes $G = (V(G), E(G))$ et $H = (V(H), E(H))$, la *somme Cartésienne* de G avec H, notée $G\square H$, est le graphe définit sur l'ensemble de sommets $V(G) \times V(H)$ tel que deux sommets (u, u') et (v, v') sont adjacents si et seulement si $(u = v \; et \; u'v' \in E(H))$ ou $(u' = v' \; et \; uv \in E(G))$. La Figure 1.9 montre la somme Cartésienne $H = C_4\square K_2$.

Il est á noter que le graphe $G\square H$ posséde $|V(G)| \times |V(H)|$ sommets et $|V(G)| \times |E(H)| \; + \; |V(H)| \times |E(G)|$ arêtes.

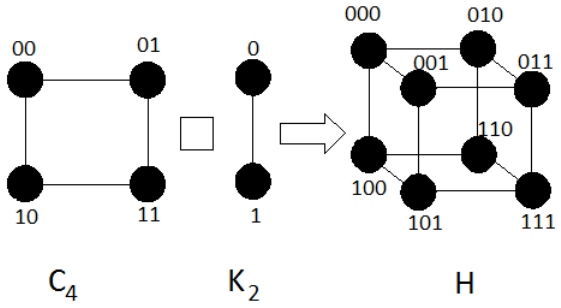


FIGURE 1.8 – Somme Cartésienne $H = C_4 \square K_2$

1.3.2 Produit Cartésien de deux graphes

Étant donné deux graphes G = (V(G),E(G)) et H = (V(H),E(H)), la *produit Cartésien* de G avec H, notée $G \times H$, est le graphe définit sur l'ensemble de sommets $V(G) \times V(H)$ tel que seux sommets (u,u') et (v,v') sont adjacents si et seulement si $(uv \in E(G))$ et $(u'v' \in E(H))$. La Figure 1.10 montre le produit Cartésien $K = G_1 \times G_2$.

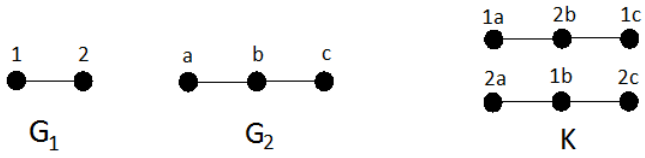


FIGURE 1.9 – Produit Cartésien $K = G_1 \times G_2$

1.3.3 Subdivisions de graphes

En remplaçant les arêtes d'un graphe $G = (V(G), E(G))$ par des chaînes disjointes intérieurement, on obtient une subdivision de G. Une subdivision d'une

arête uv du graphe G est le remplacement de cette arête par une chaîne $u = u_0, u_1, ..., u_p - 1, u_p = v$ où les u_i sont les nouveaux sommets insérés sur l'arête uv pour tout $i \in \{1, 2, ..., p-1\}$.

1.3.4 Morphismes de graphes

Soient $G = (V(G), E(G))$ et $H = (V(H), E(H))$ deux graphes. Un homomorphisme de G dans H est une application$f$ de $V(G)$ dans $V(H)$, telle que pour toute arête uv de G on a $f(u)f(v)$ est une arête de H, c'est á dire $(\forall (u, v) \in V(G) \times V(G), uv \in E(G) \Rightarrow f(u)f(v) \in E(H))$. La Figure 1.11 montre l'homomorphisme de G dans H.

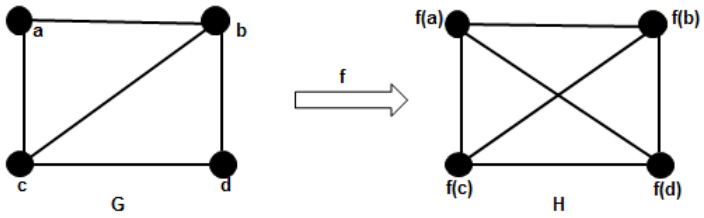


FIGURE 1.10 – Homomorphisme de G dans H

1.3.5 Isomorphismes de graphes

Deux graphe $G = (V(G), E(G))$ et $H = (V(H), E(H))$ sont dits isomorphes si et seulement si il existe une application bijective $\varphi : V(G) \longrightarrow V(H)$ qui vérifie la condition suivante :

$$uv \in E(G) \Leftrightarrow \varphi(u)\varphi(v) \in E(H).$$

Ceci signifie aussi que φ est un morphisme et φ^{-1} est un morphisme. un isomorphisme de G dans lui même est appelé isomorphisme intérieur ou bien automorphisme. On voit facilement dans la Figure 1.12(a), que l'application f de $V(G)$

dans $V(H)$, telles que $f(1) = b, f(2) = a$ et $f(3) = c.f$ présente bien un homomorphisme, mais on ne peut pas trouver un homomorphisme g de $V(G_2)$ dans $V(G_1)$. Donc$f$ n'est pas un isomorphisme, par contre dans la Figure 1.12(b), l'application t de $V(G)$dans $V(H)$, telles que $t(a) = 1, t(b) = 6, t(c) = 8, t(d) = 3, t(e) = 5, t(f) = 2, t(g) = 4$, et $t(h) = 7$ montre que G et H sont isomorphes.

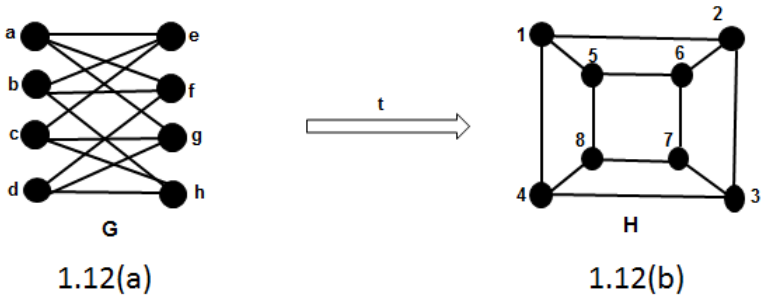


FIGURE 1.11 – Isomorphisme de G dans H

1.3.6 Distances dans les graphes

Étant donnée deux sommets u et v d'un graphe G = (V(G) , E(G)). On appelle distance entre u et v, la longueur d'une plus courte (u,v)-chaîne (en terme de nombre d'arêtes) qui les relies et on la note par $d_g(u, v)$ (ou $d(u, v)$ s'il n'y a pas de confusion). Une telle chaîne s'appelle *géodésique*.

- L'excentricité d'un sommet u notée $e_G(u)$ (ou $e(u)$ s'il n'y a pas de confusion) est le nombre suivent :
$$e_G(u) = \max_{u \in V(G)} d_G(u, v).$$
- Le diamétre de G noté D(G) (ou D s'il n'y pas de confusion) est la plus grande excentricité :
$$D(G) = \max_{u \in V(G)} [e(u)].$$
Un sommet $u \in V(G)$ est dit diamétral de $v \in V(G)$ si $d_G(u, v) = D(G)$. Si chaque
sommet de G admet un unique sommet diamétral, on dira que G est diamétral.

- Le rayon de G noté R(G) (ou R s'il n'y pas de confusion) est la plus petite excentricité :
$$R(G) = \min_{u \in V(G)} [e(u)]$$
- Le centre de G est l'ensemble des sommets de G dont l'excentricité est égale au rayon.

- La distance entre deux arêtes u_1v_1 et u_2, v_2 dans un graphe G = (V(G), E(G)) est définie par :
$$d_G(u_1v_1, u_2v_2) = min\{d_G(u_1, u_2), d_G(u_1, v_2), d_G(v_1, u_2), d_G(v_1, v_2)\}.$$

1.3.7 Intervalles dans les graphes

L'intervalle $I_G(u, v)$ (ou I(u,v) s'il n'y pas de confusion) c'est l'ensemble des sommets de G appartenant aux plus courtes (u,v)-chaîne. $I_G(u, v) = \{w \in V(G)\}$, avec w est sur une plus courte (u, v)-chaîne. Trivialement, un sommet $w \in I(u, v)$ si et seulement si :

$$d_G(u, w) + d_G(w, v) = d_G(u, v).$$

Proposition 1.3.1. [10]
Soient u et v deux sommets d'un graphe G alors :

1. $u, v \in I_G(u, v)$.
2. $I_G(u, v) = I_G(u, v)$.
3. *Si* $w \in I_G(u, v)$, *alors* $I_G(u, w) \in I_G(u, v)$.
4. *Si* $w \in I_G(u, v)$, *alors* $I_G(u, w) \cap I_G(w, v) = \{w\}$.
5. *Si* $w \in I_G(u, v)$ *et* $z \in I_G(u, w)$,*alors* $w \in I_G(z, v)$.

Proposition 1.3.2. [10]
Pour tout triplet u, v, w *d'un graphe G il existe un somet* $z \in I_G(u, w)_G(u, v)$,*tel que*

$$I_G(z, w) \cap I_G(z, v) = \{z\}.$$

Proposition 1.3.3. [10]
Soient u, v, w *et* z *quatre sommets d'un graphe G.* z *est l'unique sommets de* $I_G(u, w) \cap I_G(u, v)$, *tel que* $I_G(z, v) \cap I_G(z, w) = \{z\}$ *si est seulement si* $I_G(u, w) \cap I_G(u, v) = I_G(u, z)$.

1.3.8 Matrice d'adjacence

On peut représenter un digraphe par une matrice d'adjacences. Une matrice $(n \times n)$ est un tableau de n lignes et m colonnes. (i, j) désigne l'intersection de la ligne i et de la colonne j . Dans une matrice d'adjacences, les lignes et les colonnes représentent les sommets du graphe. Un (1) à la position (i, j) signifie qu'un arc part de i peut rejoindre j.

Cette matrice a plusieurs caractéristiques :
1. Elle est carrée : il y a autant de lignes que de colonnes.
2. Il n'y a que des zéros sur la diagonale. Un (1) sur la diagonale indiquerait une boucle.
3. Contrairement à celle d'un graphe non orienté, elle n'est pas symétrique.
4. Une fois que l'on fixe l'ordre des sommets, il existe une matrice d'adjacences unique pour chaque graphe. Celle-ci n'est la matrice d'adjacences d'aucun autre graphe.

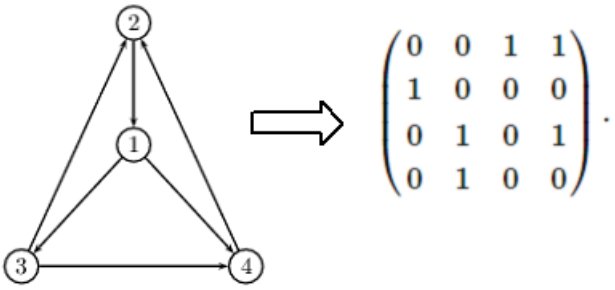
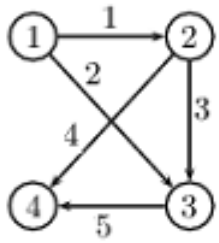


FIGURE 1.12 – Exemple d'une matrice d'adjacence

1.3.9 Matrice d'incidence

Soit G un graphe orienté qui possède n sommets numérotés de 1 à n, et m arcs numérotés de 1 à m. On appelle matrice d'incidence du graphe la matrice $A = (a_{i,j})$ comportant n lignes et m colonnes telle que :
$a_{i,j}$ vaut $+1$, si l'arc numéroté j admet le sommet i comme origine ;
$a_{i,j}$ vaut -1, si l'arc numéroté j admet le sommet i comme arrivée ;
$a_{i,j}$ vaut 0 dans les autres cas. Voici un exemple de graphe, et la matrice d'incidence associée :



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 \\ 0 & -1 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 \end{pmatrix}.$$

FIGURE 1.13 – Exemple d'une matrice d'incidence

On peut aussi définir la matrice d'incidence d'un graphe non-orienté. Dans un tel graphe, il n'y a plus de notion d'origine et d'arrivée d'une arête. On met donc $+1$ là où auparavant on mettait $+1$ ou -1, et on met 0 ailleurs.

1.4 Quelques types de graphes

1.4.1 Graphes Bipartis

Un graphe est biparti si ses sommets peuvent être divisés en deux ensembles X et Y, de sorte que toutes les arêtes du graphe relient un sommet dans X à un sommet dans Y (dans l'exemple ci-dessous, on a $X = \{2, 4\}$ et $Y = \{1, 3, 5\}$, ou vice versa.

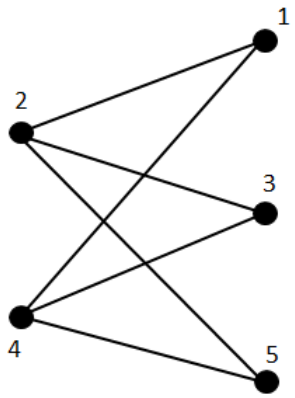


FIGURE 1.14 – Exemple de graphe biparti complet $K_{2,3}$.

Un graphe biparti est dit équilibré si chaque sous ensemble de la bipartition contient le même nombre de sommets,
les cycles de longueurs pairs sont des graphes bipartis équilibrés, les graphes bipartis complets où $|X| = |Y|$, sont aussi de tels graphes.

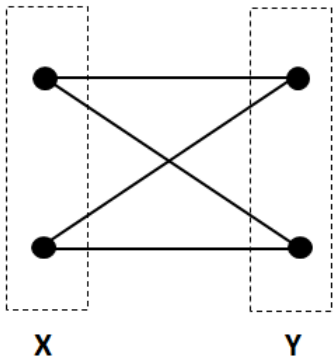


FIGURE 1.15 – Exemple de graphe biparti équilibré.

1.4.2 Graphes Hamming

Définition 1.4.1. Soient $a_1, a_2, ..., a_n$ des entiers positifs, Le graphe de Hamming $H_{a_1,a_2,...,a_n}$ est le graphe dont l'ensemble des sommets est $\Pi_{i=1}^{n}\{0, 1, ..., a_i - 1\}$ et dans lequel deux sommets sont adjacents si et seulement si leur vecteur correspon-

dant diffère exactement d'une seule composante.
On peut également le définir, comme étant la somme Cartésienne de n graphes complets :

$\mathrm{H}_{a_1,a_2,...,a_n} = K_{a_1} \square K_{a_2} \square ... \square K_{a_n}$

1.4.3 Graphes Distance Monotone

Un graphe simple connexe G=(V(G),E(G)) est dit distance monotone si, pour tout intervalle $I_G(u,v)$ et pour tout sommet $w \in I_G(u,v)$, il existe $w' \in I_G(u,v)$ tel que $d_G(w,w') > d_G(u,v)$.

1.4.4 Graphes d'Intervalles

On construit un graphe G à partir des intervalles de la droite réelle $I_1, ..., I_n$, où les sommets de G sont numérotés de1 à n. Dans un graphe d'intervalles, il existe une arête entre les sommets i et j , $i \neq j$, si et seulement si $I_i \cap I_j \neq \emptyset$.

Autrement dit, deux sommets sont reliés si et seulement si les deux intervalles correspondants se chevauchent.

2
Arbres Binaires

2.1 Arbres

Définition 2.1 Un arbre est une structure composée de sommets et de feuilles (sommets pendant) relées par des branches. On le représente généralement en mettant la racine en haut et les feuilles en bas (contrairement á un arbre réel).

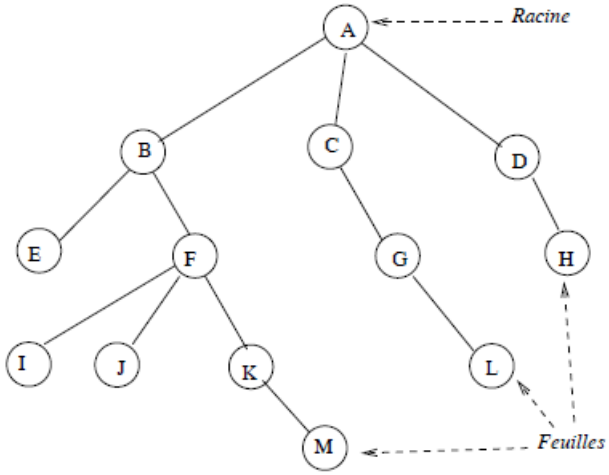


FIGURE 2.1 – Exemple d'arbre.

Théorème 2.1 [2] *Soit G un graphe d'ordre n. les propriétés suivantes sont équivalents.*

1. *G est sans cycle et connexe .*
2. *G est sans cycle et possède $n-1$ arêtes.*
3. *G est connexe et possède $n-1$ arêtes.*
4. *G est sans cycle et maximal pour cette propriété(lorsque on lui ajoute une arête on cré e un cycle et un seul).*
5. *G est connexe et maximal pour cette propriété(lorsque on lui supprime une arête quelconque on le déconnecte).*
6. *Tout couple de sommet (u, v) est relié par une chaine et une seule .*

Un graphe G=(V(G),E(G)) vérifiant au moins l'une des propriétés ci-dessus est un arbre d'ordre n.

Théorème 2.2. [2] *Tout arbre d'ordre $n \geq 2$, admet au moins deux sommets pendants.*

2.1.1 Vocabulaire employé sur les arbres

[
Le sommet A est la racine de l'arbre.
- Les sommets E, I, J, M, L et H sont des feuilles.
- Les sommets B, C, D, F, G et K sont des sommets intermédiaires.
- Si une branche relie un sommet n_i à un sommet n_j situé plus bas, on dit que n_i est un ancêtre de n_j.
- Dans un arbre, un sommet n'a qu'un seul père (ancêtre direct).
- Un sommet peut contenir une ou plusieurs valeurs.
- La hauteur (ou profondeur) d'un sommet est la longueur du chemin qui le lie á la racine.
- La taille d'un arbre est le nombre de ses sommets

Étiquette Un arbre dont tous les sommets sont nommés est dit étiqueté. L'étiquette (ou nom du sommet) représente la "valeur" du sommet ou bien l'information associée au noeud. Ci-dessous un arbre étiqueté dans les entiers entre 1 et 10 :

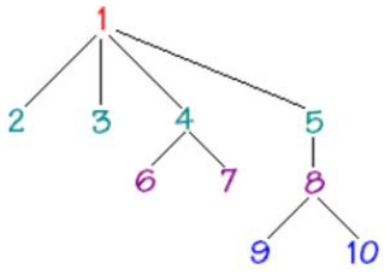


FIGURE 2.2 – Exemple d'arbre étiqueté.

Hauteur, profondeur ou niveau d'un sommet Nous conviendrons de définir la hauteur(ou profondeur ou niveau) d'un sommet x comme égale au nombre de sommets à partir de la racine pour aller jusqu'au sommet x. En reprenant l'arbre précédant et en notant h la fonction hauteur d'un sommet :
La figure ci-dessous montre les différent niveaux de l'arbre précédent :

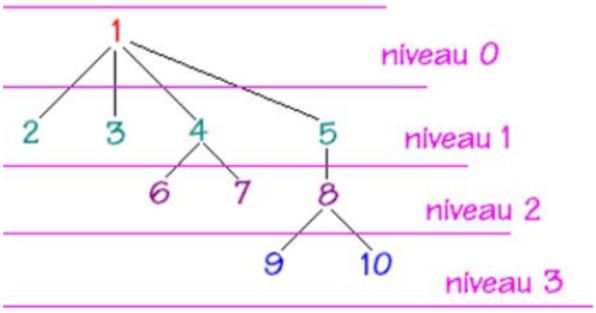


FIGURE 2.3 – Niveaux d'un arbre.

Par définition, la hauteur de la racine est égal à 1.
$h(racine) = 1$ (pour tout arbre non vide)

Remarque : (Certains auteurs adoptent une autre convention pour calculer la hauteur d'un sommet : la racine a pour hauteur 0 et donc n'est pa comptée dans le nombre de sommets, ce qui donne une hauteur inférieure d'une unité à notre définition).

Nous pouvons définir récursivement la hauteur h d'un sommet X à partir de celle de son parent :

$h(racine) = 1$;
$h(X) = 1 + h(parent(X))$

Degré d'un sommet Par définition le degré d'un sommet est égal au nombre de ses descendants (enfants).

Hauteur ou profondeur d'un arbre Par définition c'est le nombre de sommets du chemin le plus long dans l'arbre. La hauteur h d'un arbre correspond donc au nombre de niveau maximum :
h(Arbre) = max h (X) / " X, X sommet de Arbre
si Arbre est vide, alors h(Arbre) = 0

Degré d'un arbre Le degré d'un arbre est égal au plus grand des degrés de ses sommets :
$d°(Arbre) = maxd°(X)/"$ **X, X sommet de Arbre**

2.2 Arbres binaires

Définition 2.2.1 Un arbre binaire est un arbre tel que les sommets ont au plus deux fils (gauche et droit).

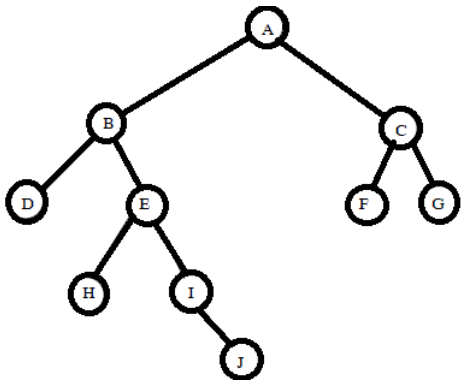


FIGURE 2.4 – Exemple d'arbre binaire.

2.2.1 Arbres binaires particuliers

a- Arbres binaires dégénérés Un arbre binaire est dit dégénéré ou filiforme si tous les sommets n'ont qu'un seul fils. ils ne sont constitués que de points simples à gauche ou à droite, il s'agit alors d'une chaine.

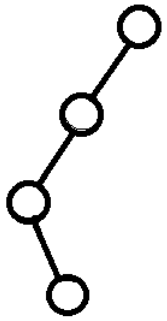


FIGURE 2.5 – Arbres binaires dégénérés ou filiformes.

b- Arbres binaires complets Un arbre binaire est dit complet si chaque niveau est complètement rempli. Le nombre total de sommets est alors : $2^{h+1} - 1$ (où h est la hauteur de l'arbre).

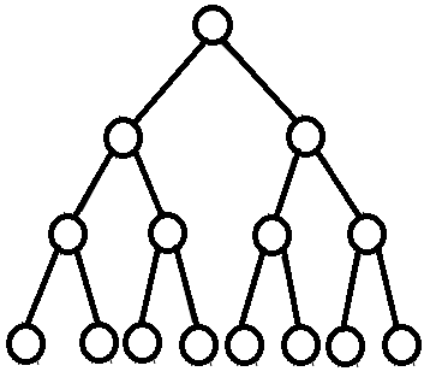


FIGURE 2.6 – Arbres binaires complets.

c- Arbres binaires localement complets Les arbres localement complets ne sont constitués que de points doubles et de feuilles. il existe des arbres localement complets particuliers comme le peigne droit (peigne gauche) dont tous les fils gauches (droits) sont des feuilles.

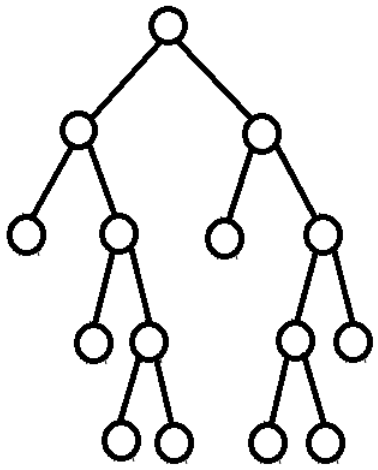


FIGURE 2.7 – Arbres binaires localement complets.

d- Arbres binaires parfaits Les arbres parfaits voient tous leurs niveaux remplis excepté le dernier qui est rempli de gauche à droite.

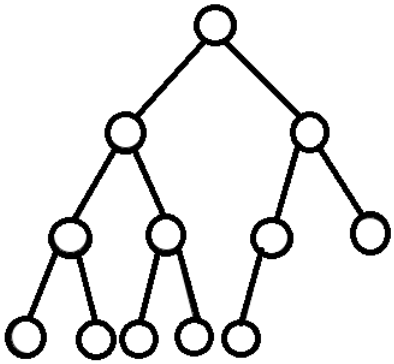


FIGURE 2.8 – Arbres binaires parfaits.

e- Arbres binaires partiellement ordonnés :

Définition 2.2.1 C'est un arbre étiqueté dont les sommets appartiennent à un ensemble muni d'une relation d'ordre total (les nombres entiers, réels etc... en sont des exemples) tel que pour un sommet donné tous ses fils ont une valeur supérieure ou égale à celle de leur pére.

Exemple d'un arbre partiellement ordonné sur l'ensemble $(20, 27, 29, 30, 32, 38, 45, 45, 50, 51, 67, 85)$ d'entiers naturels :

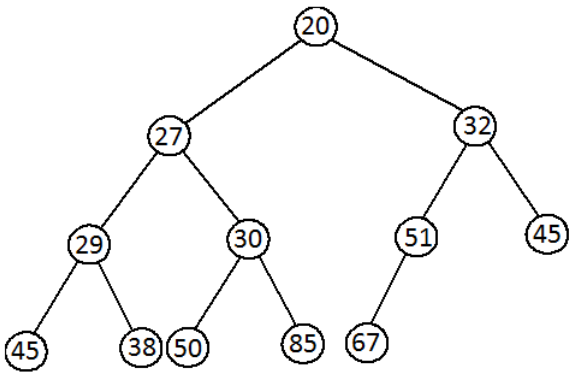


FIGURE 2.9 – Exemple arbre binaire partiellement ordonné

Nous remarquons que la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum (le plus petit élément de l'ensemble), car la valeur de ce sommet par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses descendants c'est le minimum. Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre.

2.3 Quelques utilisation d'un arbre binaire

2.3.1 La file de priorité

Le probléme à résoudre : Un certain nombre d'élements (processus, clients, requêtes dans une base de données,...) se présentent à différents moments pour utiliser une ressource. Quand celle-ci n'est pas libre, ils doivent attendre, et quand elle se libére on doit choisir l'élément le plus prioritaire. Comment tenir à jour la liste des éléments en attente ?
Structure de donnée : file de priorité =
un arbre binaire chaque sommet contient un nombre (une priorité) et tel que tout sommet a une priorité supérieure ou égale à celle de chacun de ses fils.

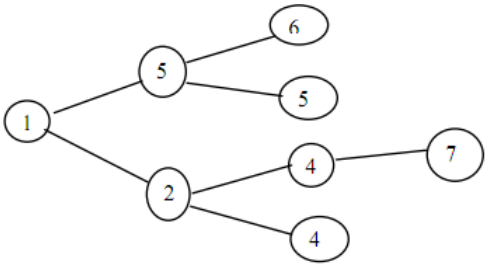


FIGURE 2.10 – Une file de priorité.

Opérations :
a- Prélever le sommet le plus prioritaire (la racine) , il faut reconstruire l'arbre.

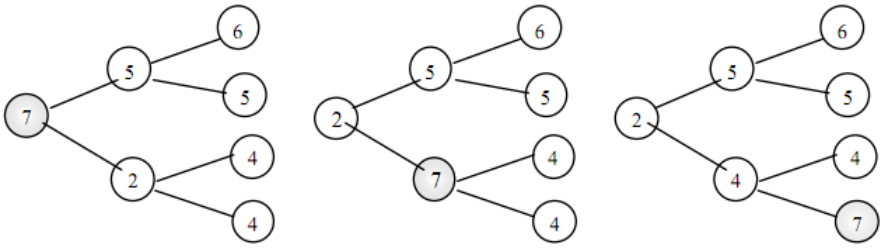


FIGURE 2.11 – Reconstruction de l'arbre.

b- Insérer un nouveau sommet.

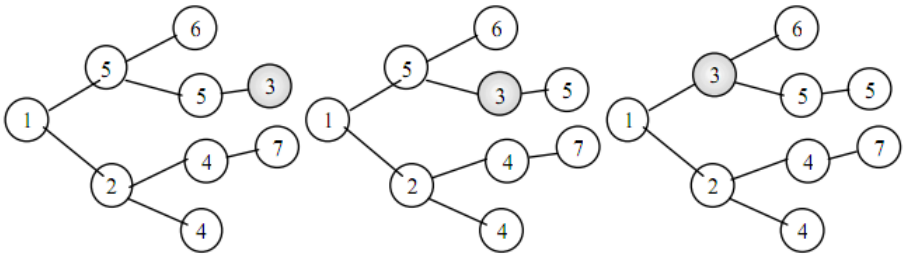


FIGURE 2.12 – Insértion d'un nouveau sommet.

2.4 Arbres binaires de recherche

On considère que l'ensemble des éléments admet un ordre total. Un arbre binaire de recherche est un arbre binaire où les éléments sont triés de gauche à droite. On a vraiment l'image du tri rapide en tête : la racine est le pivot e et les sous-arbres correspondent respectivement aux éléments plus petits ou égaux à e et aux éléments plus grands strictement que e.

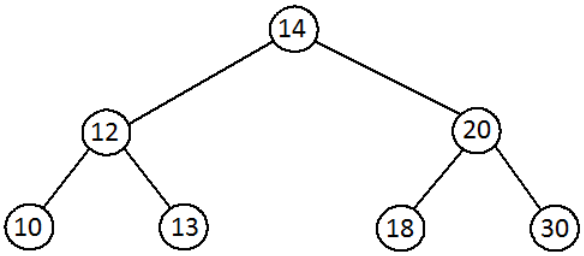


FIGURE 2.13 – Exemple d'arbres binaires de recherche.

Définition 2.4.1 : Un **arbre binaire de recherche** est un arbre binaire tel que pour tout sommet de cet arbre, pour tout $n \in G$, $n \leq x$ et pour tout $n \in D$, $n \geq x$
tels que G est le sous-arbre gauche et D est le sous-arbre droit.

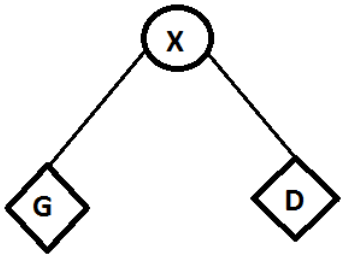


FIGURE 2.14 – Arbres binaires de recherche.

Opérations

Pour chercher i dans l'arbre,on cherche i dans G si $i \leq x$ et dans D si $i \geq x$
Pour ajouter i dans l'arbre s'il y est , dans ce cas on ne fait rien . si non on l'ajoute à G si $i \leq x$ et l'ajoute à D si non .
pour supprimer i de l'arbre on le supprime dans G si $i \leq x$, dans D si i $\geq x$ et si $i = x$, on renvoie $N(G', y, D)$ avec $y = max\ G$ et $G' = G \setminus (y)$.
Ces trois opérations en $O(h)$ avec h la hauteur de l'arbre.
La suppression est moins triviale. On recherche l'élément e à supprimer dans T . Une fois trouvé, nous avons un arbre de racine e avec deux sous-arbres. Si un des sous-arbres est vide...

c'est pratique ! On remplace l'arbre courant par ce sous-arbre. Sinon, on remplace la racine par le maximum du sous-arbre de gauche tout en supprimant l'élément maximal du sous-arbre gauche comme le montre la figure 2.15 :

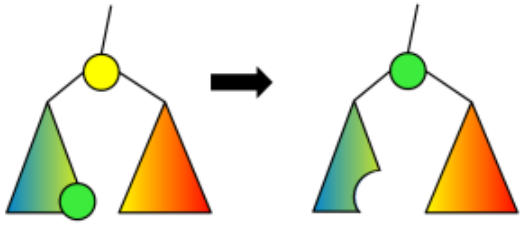


FIGURE 2.15 – Arbres binaires de recherche.

Pour trouver le maximum d'un arbre, on file tout droit à droite ! ! Dans le pire des cas, toutes ces opérations sont réalisées en O(h) où h est la hauteur de l'arbre. Dans le pire des cas, l'arbre peut être linéaire et la hauteur est alors égale à n, le nombre d'élément dans l'ensemble.

2.4.1 Intérêt des Arbres Binaires de Recherche

-Les Arbres Binaires de Recherche sont une structure de données qui permet de représenter correctement à un ensemble ordonné de clés.
-Les alternatives (tableaux ou listes) ont de bien moins bons comportements dans les operations usuelles : ajout / suppression / recherche.
-Tous les algorithmes des arbres binaires de recherches , le temps de calcul est proportionnel à la hauteur de l'arbre (et non au nombre de clés comme pour les autres alternatives).
-Toutefois, dans le pire des cas, la hauteur de l'arbre est le nombre de clés dans l'arbre.
-En moyenne, les ABRs sont quand même souvent interessants.
-Pour éviter les mauvais cas : **équilibrage des arbres**.

3

Méthodes appliquées sur les arbres

3.1 Parcours d'un arbre binaire

Objectif Les arbres sont des structures de données. Les informations sont contenues dans les sommets de l'arbre, afin de construire des algorithmes effectuant des opérations sur ces informations (ajout, suppression, modification,...) il nous faut pouvoir examiner tous les sommets d'un arbre. Nous devons avoir à notre disposition un moyen de parcourir ou traverser chaque sommet de l'arbre et d'appliquer un traitement à la donnée rattachée à chaque sommet.
Un parcours est une énumération des sommets de l'arbre, chaque parcours définit un ordre sur les sommets.
Cette opération consiste à retrouver systématiquement tous les sommets d'un arbre et d'y appliquer un même traitement, Il existe deux types principaux de parcours.

3.1.1 Parcours en largeur où hiérarchique

Un algorithme classique consiste à explorer chaque sommet d'un niveau donné de gauche à droite, puis de passer au niveau suivant. On part de la racine et on énumère l'ensemble de ses descendants directs, Puis on réitère le processus sur les descendants eux-mêmes. On dénomme cette stratégie le parcours en largeur de l'arbre.

3.1.2 Parcours en profondeur

La stratégie consiste à descendre le plus profondément soit jusqu'aux feuilles d'un sommet de l'arbre, puis lorsque toutes les feuilles du sommet ont été visitées, l'algorithme "remonte" au sommet plus haut dont les feuilles n'ont pas encore été visitées. Notons que ce parcours peut s'effectuer systématiquement en commencant par le fils gauche, puis en examinant le fils droit ou bien l'inverse.
À partir de ce contour, on définit trois parcours des sommets de l'arbre :

1- L'ordre préfixe :
La règle de parcours est la suivante :

- Traiter la racine
- Parcourir le sous-arbre gauche
- Parcourir le sous-arbre droit

On liste chaque sommet la première fois qu'on le rencontre dans la balade.

Exemple : Prenons l'exemple de l'arbre binaire de calcul ci-dessous.

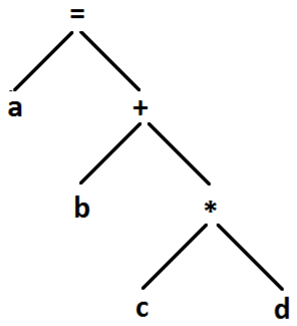


FIGURE 3.1 – Arbre binaire de calcul.

Le parcours préfixe donne sur cet exemple :$(= a + b * cd)$

2- L'ordre postfixe :

La règle de parcours est la suivante :

- Parcourir le sous-arbre gauche
- Parcourir le sous-arbre droit
- Traiter la racine

On liste chaque sommet la dernière fois qu'on le rencontre.

L'application du parcours postfixe sur l'exemple precedant donne :$(abcd * + =)$

3- L'ordre infixe :

La règle de parcours est la suivante :

- Parcourir le sous-arbre gauche
- Traiter la racine
- Parcourir le sous-arbre droit

On liste chaque sommet ayant un fils gauche la seconde fois qu'on le voit et chaque sommet sans fils gauche la première fois qu'on le voit.

sur le même exemple, le parcours infixe donne :$a = (b + (c * d))$

Voici les algorithme récursifs des trois parcours précedants :

1. Parcours préfixe.

Pseudo-code

```
ParcoursPréfixe(Arbre binaire T de racine r)
        Afficher clef[r]
        ParcoursPréfixe(Arbre de racine fils_gauche[r])
        ParcoursPréfixe(Arbre de racine fils_droit[r])
```

2. Parcours postfixe.

Pseudo-code

```
ParcoursPostfixe(Arbre binaire T de racine r)
        ParcoursPostfixe(Arbre de racine fils_gauche[r])
        ParcoursPostfixe(Arbre de racine fils_droit[r])
        Afficher clef[r]
```

3. Parcours infixe.

Pseudo-code

```
ParcoursInfixe(Arbre binaire T de racine r)
        ParcoursInfixe(Arbre de racine fils_gauche[r])
        Afficher clef[r]
        ParcoursInfixe(Arbre de racine fils_droit[r])
```

Exemple. Les trois parcours du graphe suivant sont donnés ci-dessous

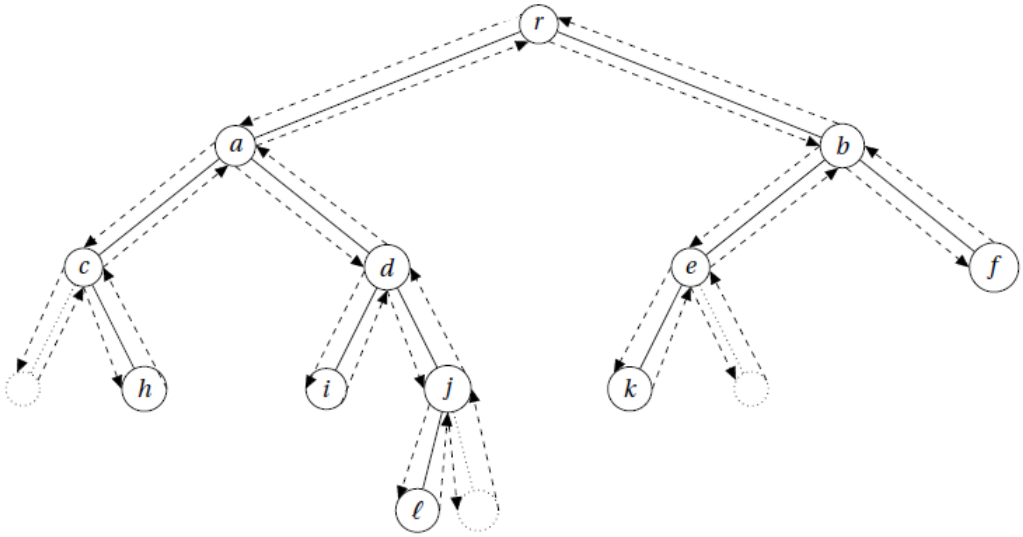


FIGURE 3.2 – Parcours d'arbre binaire .

– Ordre préfixe : r,a,c,h,d,i,j,l,b,e,k,f .
– Ordre postfixe : h,c,i,l,j,d,a,k,e,f,b,r .
– Ordre infixe : c,h,a,i,d,l,j,r,k,e,b,f .

On peut ainsi considérer qu'on passe une fois à gauche de chaque sommet (en descendant), une fois en dessous de chaque sommet, une fois à droite de chaque sommet (en remontant). Vérifier, sur l'exemple , que chacun des ordres préfixe, infixe, postfixe est obtenu en listant tous les mots :

– Soit lorsqu'on passe à leur gauche,
– Soit lorsqu'on passe à leur droite,
– Soit lorsqu'on passe en-dessous.

1- A gauche : préfixe.
2- A droite : postfixe.
3- En-dessous : infixe.

3.2 Implémentation

Nous proposons de représenter un arbre binaire étiqueté selon deux spécifications différentes classiques :
1- Une implantation fondée sur une structure de tableau en **allocation de mémoire statique**, nécessitant de connaître au préalable le nombre maximal de sommets de l'arbre (ou encore sa taille).

2- Une implantation fondée sur une structure **d'allocation de mémoire dynamique** implémentée soit par des pointeurs (variables dynamiques) soit par des références (objets) .

3.2.1 Implantation dans un tableau statique

Facile à implémenter mais :
- Taille fixée à la compilation.
- Insertion/ suppression au milieu de la structure coûteuse (séquence), aux deux extrémités astucieuses (queue, séquence).

Spécification concréte Un sommet est une structure statique contenant 3 éléments :
- L'information du sommet.
- Le fils gauche.
- Le fils droit.

Pour un arbre binaire de taille = n, chaque sommet de l'arbre binaire est stocké dans une cellule d'un tableau de dimension 1 à n cellules. Donc chaque sommet est repéré dans le tableau par un indice (celui de la cellule le contenant).
Le champ fils gauche du sommet sera l'indice de la cellule contenant le descendant gauche, et le champ fils droit vaudra l'indice de la cellule contenant le descendant droit.

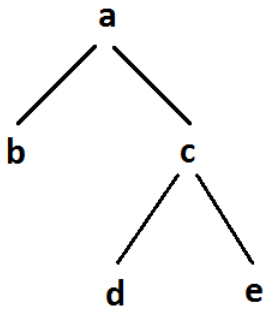


FIGURE 3.3 – Exemple

Selon l'implantation choisie, par hypothése de départ, la racine $< a$, vers b, vers $c >$ est contenue dans la cellule d'indice 2 du tableau, les autres sommets sont supposés être rangés dans les cellules $1, 3, 4, 5$:

racine = table[2]
table[1] = $<d, 0, 0>$
table[2] = $<a, 4, 5>$
table[3] = $< e, 0, 0>$
table[4] = $< b, 0, 0>$
table[5] = $< c, 1, 3>$

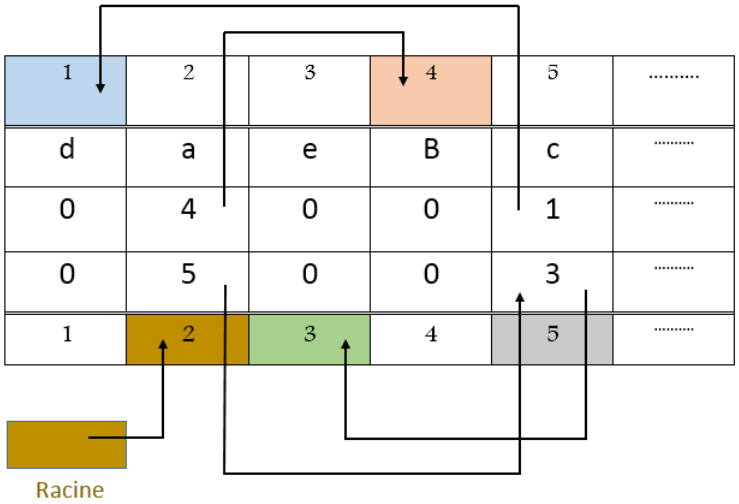


FIGURE 3.4 – Représentation par tableau .

-L'insertion ou la suppression d'un sommet dans l'arbre ainsi représenté s'effectue directement dans une cellule du tableau. Il faudra donc posséder une structure (de liste, de pile ou de file par exemple) permettant de connaître les cellules libres ou de ranger une cellule nouvellement libérée. Une telle structure se dénomme "espace libre".

– L'insertion se fera dans la première cellule libre, l'espace libre diminuant d'une unité.

– La suppression rajoutera une nouvelle cellule dans l'espace libre qui augmentera d'une unité.

3.2.2 Implantation avec des variables dynamiques

On peut définir une représentation des arbres en utilisant les pointeurs. Dans ce cas un arbre binaire peut être vide (pointeur égal à NIL)
voici les regles a suivre lors de l'implantation :

- La taille de la variable est connue
- Insertion/suppression au milieu de la structure
- Insertion/suppression aux deux extrémités (à condition d'avoir une référence directe au début et à la fin de liste)
- L'accès au i-ème élément demande le parcours séquentiel de la liste depuis le début
- Le calcul du nombre de pointeurs de chaque noeud est nécessaire.

Spécification concrète Le sommet reste une structure statique contenant 3 éléments dont 2 sont dynamiques :

- L'information du sommet
- Une référence vers le fils gauche.
- Une rèférence vers le fils droit.

Exemple

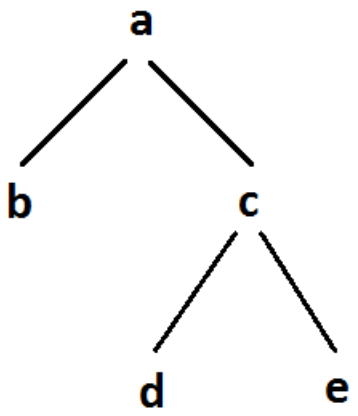


FIGURE 3.5 – Exemple

Selon l'implantation choisie, par hypothèse de départ, la référence vers la racine pointe vers la structure statique (le sommet) $< a$, ref vers b, ref vers $c >$

ref racine $< a$, ref vers b, ref vers $c >$
ref vers $b < b$, null, null>
ref vers $c < a$, ref vers d, ref vers $e >$
ref vers $d < d$, null, null $>$
ref vers $e < e$, null, null $>$

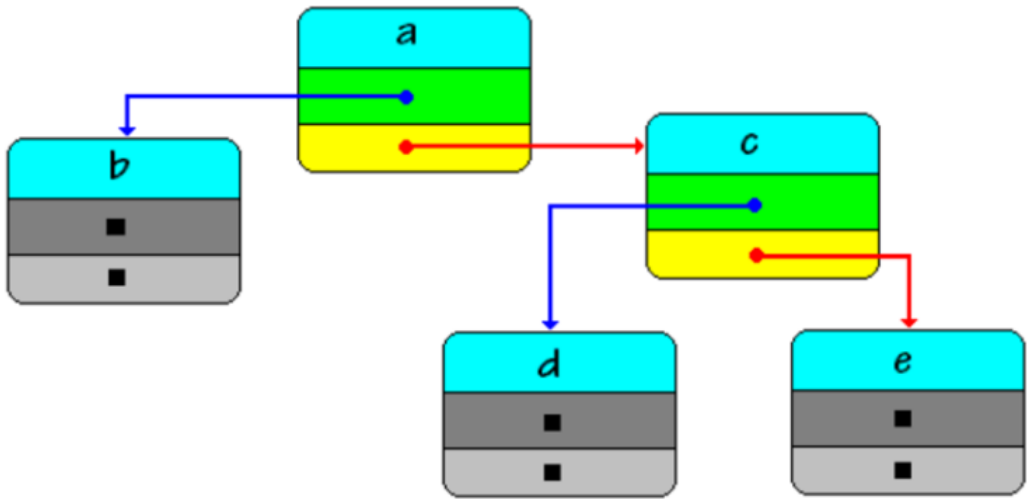


FIGURE 3.6 – Ilustration des pointeur.

Nous noterons une simplification notable des écritures dans cette implantation par rappoprt à l'implantation dans un tableau statique. Ceci provient du fait que la structure d'arbre est définie rècursivement et que la notion de variable dynamique permet une définition récursive donc plus proche de la structure.

3.3 Problème de l'arbre couvrant de poids minimum

3.3.1 Algorithme de Kruskal

L'algorithme de *Kruskal* permet de déterminer l'arbre couvrant de poids minimale dans un graphe pendéré.
L'ensemble des arêtes gardées forme par construction un arbre couvrant, de plus on a gardé les plus petites arêtes nécessaires à la couverture du graphe : c'est un arbre couvrant minimale.
L'algorithme de kruskal est utilisé dans la théorie des graphes pour construire une arborescence de poids minimale.

3.3.2 Principe de l'algorithme de Kruskal

L'idée de l'algorithme de Kruskal est tout d'abord de numéroter les arcs par ordre de poids croissants. En suit de construire progressivement l'arbre A en rajoutant dans leur ordre, les arcs un par un. Un arc est ajouté seulement si son adjonction à A ne detérmine pas un cycle, c'est á dire si A ne prend pas sa notion d'arbre sinon on pass à l'arc suivant dans l'ordre de le numérotation.

3.3.3 Énonce de l'algorithme de Kruskal :

Données : Soit $G = (X, E, F)$ un graphe,
(0) **Initialisation :** Numéroter les arcs de G dans l'ordre des poids croissants :
$P(u_1) \leq$P(u_2) $\leq$P(u_m) soit $W = \oslash$,i=1:
(1)**Si**(X , W $\cup U_i$) contient un cycle aller en (3) sinon aller en (2) ;
(2) On pose $W = W \cup U_i$, aller en (3) :
(3) Si $i = m$ termine
$A = (X, W)$ est l 'arbre de poids minimaum $P(W) = \sum P(u_i)$ pour $u_i \in W$
Sinon , $i = i + 1$ aller en (1).
Le critère d'arrêt : L'algorithme s'arrête lorsque le nombre d'arcs retenus est

égale à $n-1$
Résultat : Trouver un arbre couvrant de poids minimum G .

3.3.4 L'algorithme de Prim

L'algorithme de Prim est un algorithme qui permet de trouver un arbre couvrant minimal dans un graphe connexe valué. En d'autres termes, cet algorithme trouve un sous-ensemble d'arêtes formant un arbre sur l'ensemble des sommets du graphe initiale, et Si le graphe n'est pas connexe, alors l'algorithme ne déterminera que l'arbre couvrant minimal d'une composantes connexe du graphe, il a été concu en 1957 par *C. Prim*

3.3.5 Principe de l'algorithme de Prim

L'algorithme consiste à choisir arbitrairement un sommet et à faire croître un arbre à partire de ce sommet. Chaque augmentation se fait de la manière la plus économique possible.

3.3.6 Énonce de l'algorithme de Prim

données : soit $G = (X, E, P)$ un graphe probabiliste.
soit I l'ensemble de sommets déja inclus dans l'arbre et NI l'ensemble de sommets non encors inclus,
(0) **Initialisation :** $I = \oslash$, $NI =$ l'ensemble de tout les sommets,
(1) Mettre le sommet de départ dans I ,
(2) SI l'arbre est connexe aller à (5)
(3) Trouver un sommet de NI dont la distance à un sommet a quelquonque de I est minimal.
On relie ce sommet a au sommet b et faire $I = I + b$, $NI = NI - b$
(4) Aller à (3).
(5) FIN.
Le critère d'arrêt : L'algorithme s'arrête lorsque le nombre d'arcs retenus est égale à $n-1$.
Résultat : Trouver un arbre de poids minimum de G.

3.4 Tri par tas

C'est un tri également appelé tri par tas (heapsort, en anglais). Il utilise une structure de données temporaire dénommée "tas" comme mémoire de travail. C'est une variante de méthode de tri par sélection où l'on parcourt le tableau des éléments

en sélectionnant et conservant les minimas successifs (plus petits éléments partiels) dans un arbre parfait partiellement ordonné.

Le tas : On appelle tas un tableau représentant un arbre parfait partiellement ordonné

3.4.1 Principe du tri par tas

A) Spécification abstraite

Soit une liste $(a_1, a_2, ..., a_n)$ d'éléments appartenant à un ensemble totalement ordonné (entiers, chaînes de caractères,...). Le principe est de parcourir la liste $(a_1, a_2, ..., a_n)$ en ajoutant chaque élément a_k dans un arbre parfait partiellement ordonné.

-L'insertion d'un nouvel élément a_k dans l'arbre a lieu **dans la dernière feuille vide de l'arbre à partir de la gauche** (ou bien si le niveau est complet en recommencant un nouveau niveau par sa feuille la plus à gauche) et, en effectuant des échanges tant que la valeur de a_k est inférieure à celle de son pére. Lorsque tous les éléments de la liste seront placés dans l'arbre, l'élément minimum a_i de la liste $(a_1, a_2, ..., a_n)$ se retrouve à la racine de l'arbre qui est alors partiellement ordonné.
-On recommence le travail sur la nouvelle liste $(a_1, a_2, ..., a_n) - ai$ (la précédente privée de son minimum), pour cela on supprime l'élément minimum a_i de l'arbre pour le mettre dans la liste triée puis, **on prend l'élèment de la derniére feuille du dernier niveau et on le place à la racine**. On effectue ensuite des échanges de contenu avec le fils dont le contenu est inférieur, en partant de la racine, et en descendant vers le fils avec lequel on a fait un échange, ceci tant qu'il n'a pas un contenu inférieur à ceux de ses deux fils (ou de son seul fils) ou tant qu'il n'est pas à une feuille.
-On recommence l'opŕation de suppression et d'échanges éventuels **jusqu'à ce que l'arbre ne contienne plus aucun élément**

B) Spécification concrète

La suite $(a_1, a_2, ..., a_n)$ est rangée dans un tableau $T[...]$ correspondant au tableau d'initialisation. Puis les éléments de ce tableau sont ajouter et traiter un par un dans un arbre avant d'être ajoutés dans un tableau trié en ordre décroissant ou croissant, selon le choix de l'utilisateur.
Rappelons qu'un arbre binaire parfait se représente classiquement par un tableau

Si T est ce tableau, on a les régles suivantes :
$T[1]$ est la racine :

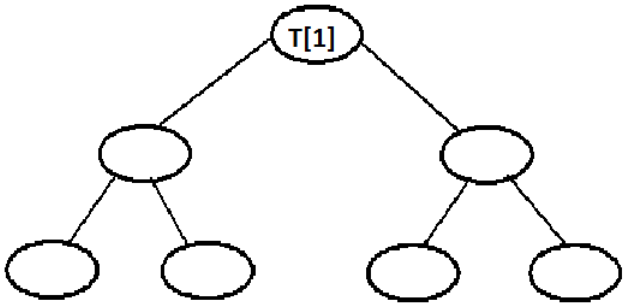


FIGURE 3.7 – Représentation de la racine

- $T[idiv2]$ est le père de $T[i]$ pour $i \geq 1$:

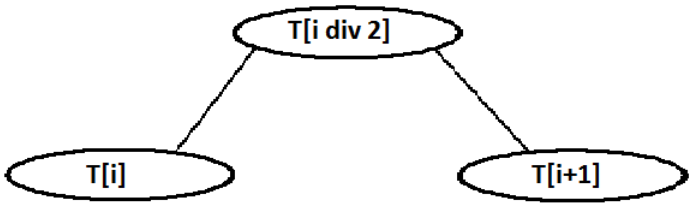


FIGURE 3.8 – Représentation du pére

- $T[2 * i]$ et $T[2 * i + 1]$ sont les deux fils, s'ils existent, de $T[i]$.

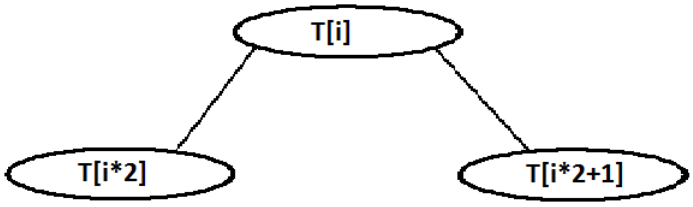


FIGURE 3.9 – Représentation des fils

- Si P est le nombre de sommets de l'arbre et si $2 * i = p$, $T[i]$ n'a qu'un fils,
- Si i est supérieur à $p\ div2$, $T[i]$ est une feuille

implementation

On en déduit l'algorithme ci dessous composé de 2 sous algorithmes Ajouter pour la première étape, et Supprimer pour la seconde :
voici l'algorithme de l'ajout :

Algorithme Ajouter
Entrée P : entier ; X : entier ;// *P nombre d'éléments dans le tas ; X : élément à ajouter.*
Tas[1...max] : tableau d'entiers ;//*le tas*
Sortie : P : entier ;
Tas[1...max] :// *le tas*
Local : j, temp : entiers ;
début :
P ← P-1 ; //*incrémentation du nombre d'éléments du tas.*
j ← p ; //*initialisation de j à la longueur du tas(position de la dernière feuille)*
Tas[p] ← x; //*ajout de l'élément x à la dernière feuille dans le tas.*
Tantque(j ≻ 1) et (Tas[j] ≺ Tas[j div 2]) faire ;
//*tant que l'on est pas arrivé à la racine, et le fils est inférieur à son père, on permute les 2 valeu*
temp← Tas[j] ;
Tas[j] ← Tas[j div 2] ;
Tas[j div 2] ← temp ;
j ← j div 2 ;

Fin tantque
Fin Ajouter

Et voici l'algorithme de suppression :

Algorithme Supprimer
Entré : P : entier ; //*P nombre d'éléments contenus dans l'arbre.*
Tas[1...max] : tableau d'entiers ; //*le tas.*
Sortie : P : entier ; //*P nombre d'éléments contenus dans l'arbre.*
Tas[1...max] : tableau d'entiers ; //*le tas.*
Lemini : entier ; //*le plus petit de tous les éléments du tas.*
Local i, j, temp, : entiers ;
dédut
Lemini ← Tas[1] ; //*retourne la racine(minimum actuel) pour stockage éventuel.*
Tas[1] ← Tas[p] ; //*la racine prend le valeur de la dernière feuille de l'arbre.*
P ← P-1 ;
j ← 1 ;

Tantque j $\prec=$ (P div 2) faire
//recherche de l'indice i du plus petit des descendants de Tas[j]
Si (2*j = p) ou (Tas[2*j] $\prec$ Tas[2*j+1])
alors i $\leftarrow$ 2*j ;
sinon i $\leftarrow$ 2*j+1 ;
FinSi
//Echange éventuel entre Tas[j] et son fils Tas[i].
Si Tas[j] $\succ$ Tas[i] alors
temp $\leftarrow$ Tas[j] ;
Tas[j] $\leftarrow$ Tas[i] ;
Tas[i] $\leftarrow$ temp ;
j $\leftarrow$ i ;
Sinon Sortir
Finsi
Fin Tantque
Fin Supprimer

Nous avons l'algorithme de tri par arbre qui fait appel aux deux algorithme(Ajouter et Supprimer), et renvoie un tableau triée par ordre croissant.

L'algorithme de tri par arbre : .
Algorithme Tri-par-arbre
Entrée : Tas[1...max]*//le tas.*
TabInit[1...max]*//les données initiales.*
Sortie : TabTrie[1...max] : tableau d'entiers. *//tableau une fois trié.*
Local P : entier ; *//P le nombre d'éléments à trier.*
Lemin : entier ; *//le plus petit de tous les éléments du tas*
début
P $\leftarrow$ 0 ;
Tantque (P $\prec$ max) faire
Ajouter(P, Tas, TabInit[p+1]) ; *//appel de l'algorithme Ajouter.*
Fin Tantque
Tantque (P $\succ=$ 1), faire
Supprimer(P, Tas, Lemin ; *//appel a l'algorithme Supprimer.*
TabTrie[max - P] $\leftarrow$ Lemin ; *//stockage du minimum dans le nouveau tableau.*
Fin Tantque
Fin Trie-par-arbre

L'extraction successive des racines peut alors se faire directement en place.

Rappelons nous que pour extraire la racine d'un tas (c'est à dire son plus petit élément), nous l'avons échangé avec le dernier élément du tas (ce qui la place donc en dernière position), puis nous l'avons supprimée (ce qui revient à diminuer de 1 la longueur de travail) et enfin nous avons effectué une percolation de l'étiquette du premier élément du tableau. En itérant cette méthode, nous allons donc obtenir un tableau dans lequel le plus grand élément sera en derniére position puis l'élément juste un peu plus petit sera en avant derniére position, et ainsi de suite. Autrement dit, notre tableau sera trié en ordre croissant.

4
Applications

4.1 Problème de tri

4.1.1 Algorithme de tri par tas

Problème : Prenons l'exemple de la suite de p chiffres qu'on veut trier par ordre croissant, pour ca nous allons utiliser les arbres binaire dans un algorithme de tri par arbre (tri par tas).

La suite de chiffres que nous allons traiter est la suivante :
$N = \{50, 38, 70, 12, 43, 15, 45, 63, 08, 19, 95, 68, 75, 23, 35, 47, 59, 61, 32, 41\}$.

et $|N| = 20$.
1**-Algorithme del'Ajout** :

▶ Ajout du premier élément 50 comme racine de l'arbre :

▶ Ajout du second élement 38 comme fils gauche de la racine :
Comparer 38 à son pére ;
On a 38 est plus petit que 50 ;
⇒ Permuter entre 38 et 50 ;

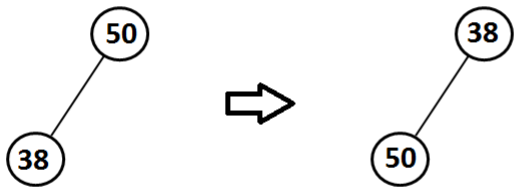


FIGURE 4.1 – Ajout de 50 et 38

▶ Ajout du troisiéme élement 70 comme fils droit de la racine :
Comparer 70 à son pére ;
On a 70 est plus grand que 50 ;

Passer au prochain élement.

▶ Ajout du quatrième élement 12 dans un nouveau niveau sur l'extrème gauche (fils gauche de 50) :
Comparer 12 à son pére ;
On a : 12 est plus petit que ; 50
⇒ Permuter entre 12 et 50 ;

ensuite comparer 12 à son nouveau pére ;
On a : 12 est plus petit que 38 ;

⇒ Permuter entre 12 et 38 ;

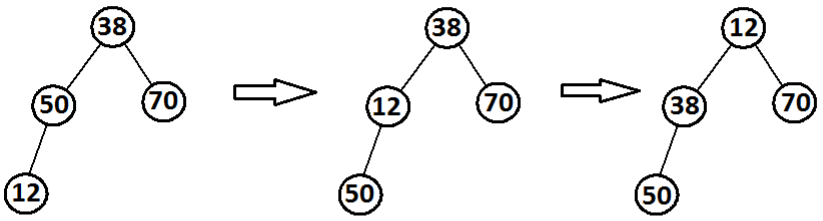


FIGURE 4.2 – Ajout de 12

► Ajout de 43 dans la dernière feuille de l'arbre :
Comparer 43 à son pére ;

43 est plus grand que 38 ;
Passer au prochain élement ;

On continue de la même manière pour ajouter tout les noeud de la suite de chiffres N. nous obtenons ainsi **l'arbre parfait partiellement ordonné** qui est montré dans la figure ci dessous :

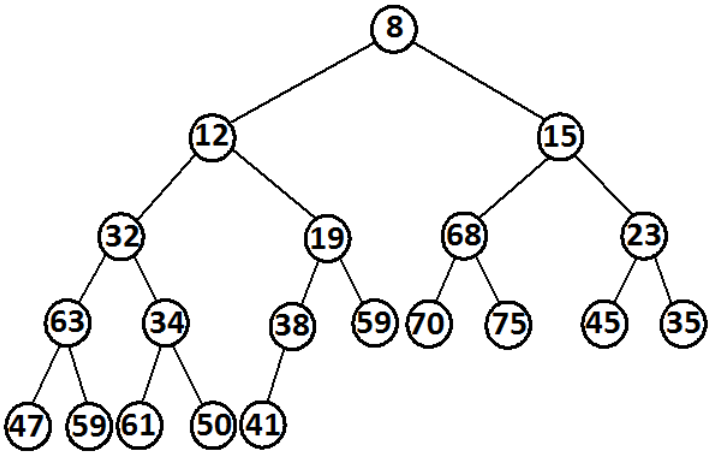


FIGURE 4.3 – Premier tas

2-**Algorithme de la Suppréssion** :

Itération 1 **:**

Supprimer la racine 8 de l'arbre qui est le minimum ; et on pose $T = \{8\}$;
$N = N - \{8\} = \{12, 15, 32, 19, 68, 23, 63, 34, 38, 59, 70, 75, 45, 35, 47, 59, 61, 50, 41\}$;

41 devient la racine du nouvel arbre ;

$min\{12, 15\} = 12$;
Comparer 12 à son pére 41 ;
12 est plus petit que 41 ;
Permuter entre 12 et 41 ;

$min\{32, 19\} = 19$;
Comparer 19 à son pére 41 ;
19 est plus petit que 41 ;
Permuter entre 19 et 41 ;

$min\{38, 59\} = 38$;
Comparer 38 à son pére 41 ;
38 est plus petit que 41 ;
Permuter entre 38 et 41 ;

La figure ci dessous montre les différentes étapes de l'itération 1.

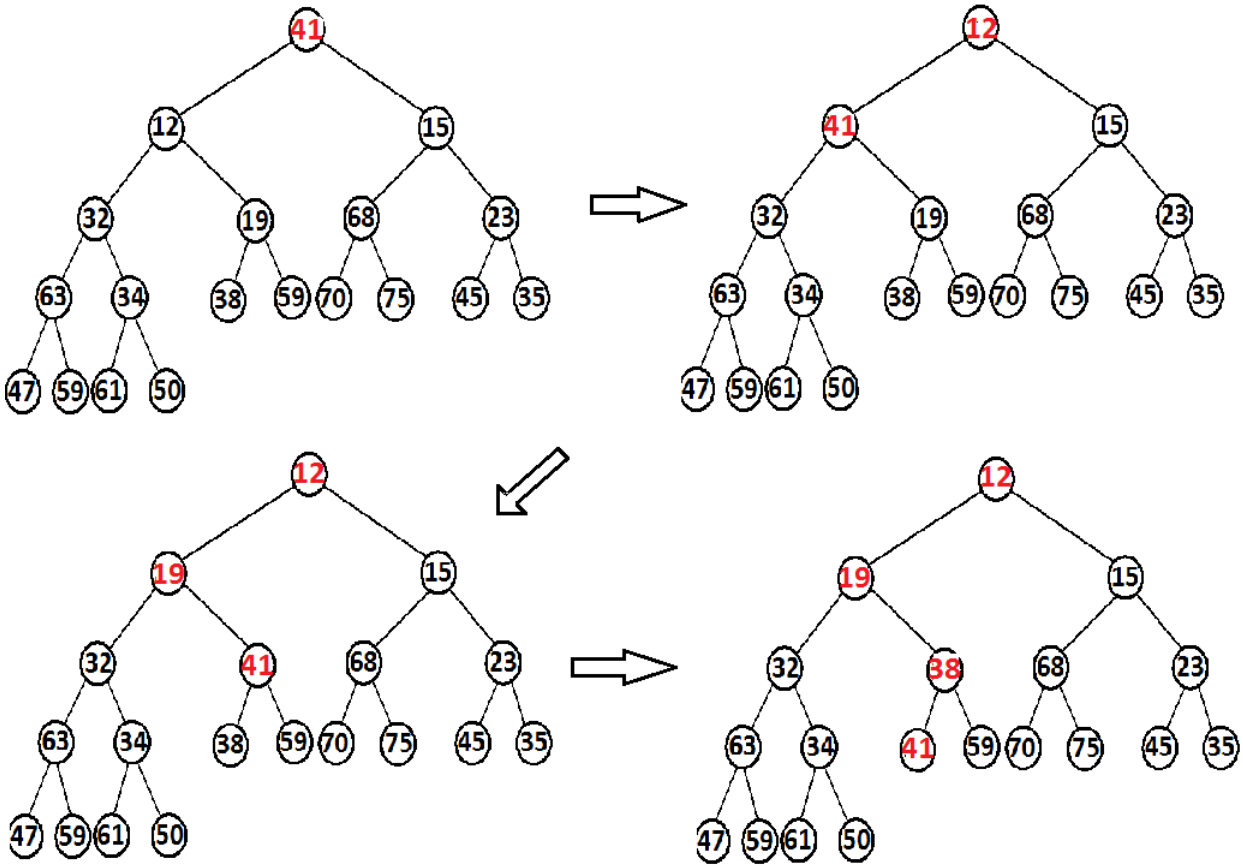


FIGURE 4.4 – Première itération

Itération 2

Supprimer la racine 12 de l'arbre qui est le minimum ; et on pose $T = \{8, 12\}$;
$N = N - \{12\} = \{15, 32, 19, 68, 23, 63, 34, 38, 59, 70, 75, 45, 35, 47, 59, 61, 50, 41\}$;
50 devient la racine du nouvel arbre ;

$min\{15, 19\} = 15$;
Comparer 15 à son pére 50 ;
15 est plus petit que 50 ;
Permuter entre 15 et 50 ;

$min\{23, 68\} = 23$;
Comparer 23 à son pére 50 ;
23 est plus petit que 50 ;

Permuter entre 23 et 50 ;

$min\{35, 45\} = 35$;
Comparer 35 à son pére 50 ;
35 est plus petit que 50 ;
Permuter entre 35 et 50 ;

La figure ci dessous montre les différentes étapes de l'itération 2.

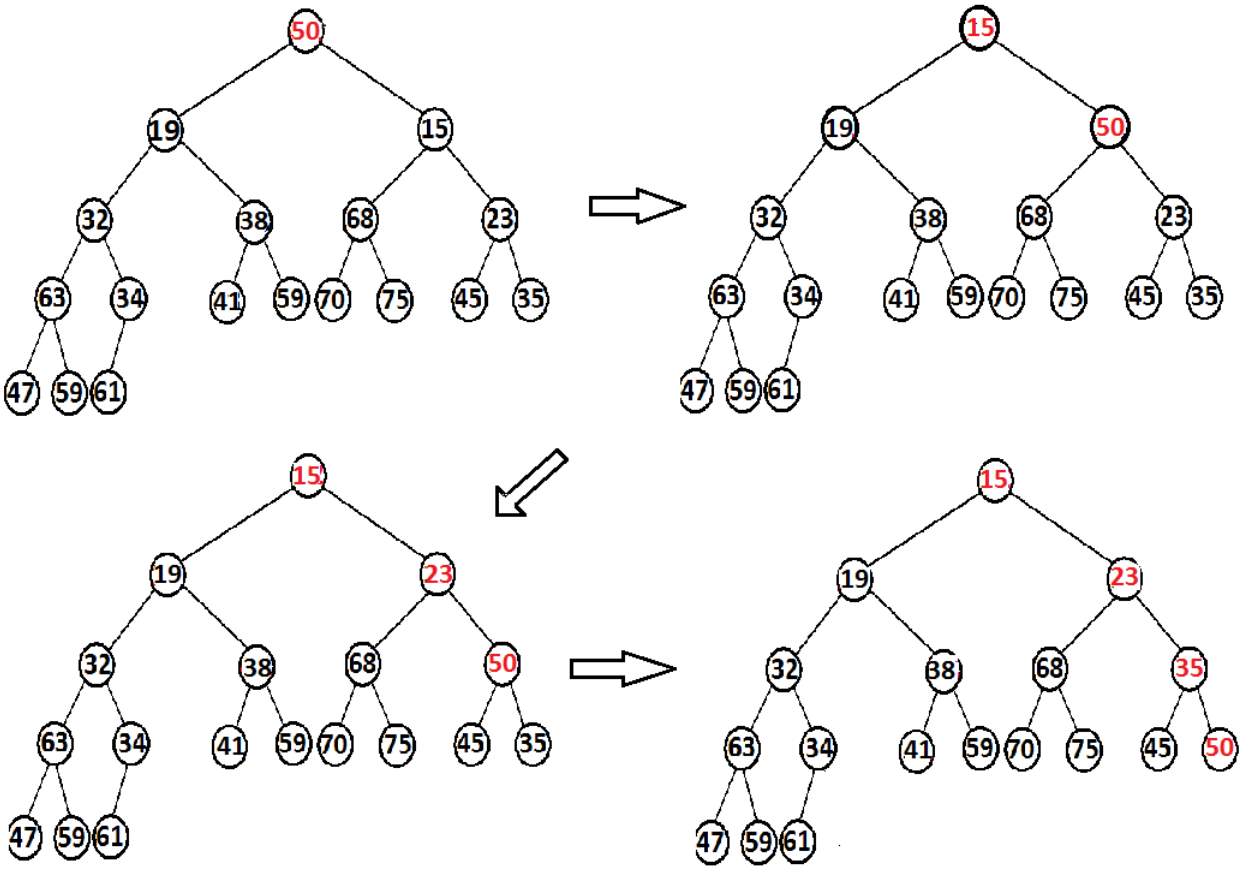


FIGURE 4.5 – Deuxième itération

Itération 3
Supprimer la racine 15 de l'arbre qui est le minimum ; et on pose $T = \{8, 12, 15\}$;
$N = N - \{15\} = \{32, 19, 68, 23, 63, 34, 38, 59, 70, 75, 45, 35, 47, 59, 61, 50, 41\}$;
61 devient la racine du nouvel arbre ;

$min\{19, 23\} = 19$;
Comparer 19 à son pére 61 ;
19 est plus petit que 61 ;
Permuter entre 19 et 61 ;

$min\{32, 38\} = 32$;
Comparer 32 à son pére 61 ;
32 est plus petit que 61 ;
Permuter entre 32 et 61 ;

$min\{63, 43\} = 43$;
Comparer 43 à son pére 61 ;
43 est plus petit que 61 ;
Permuter entre 43 et 61 ;

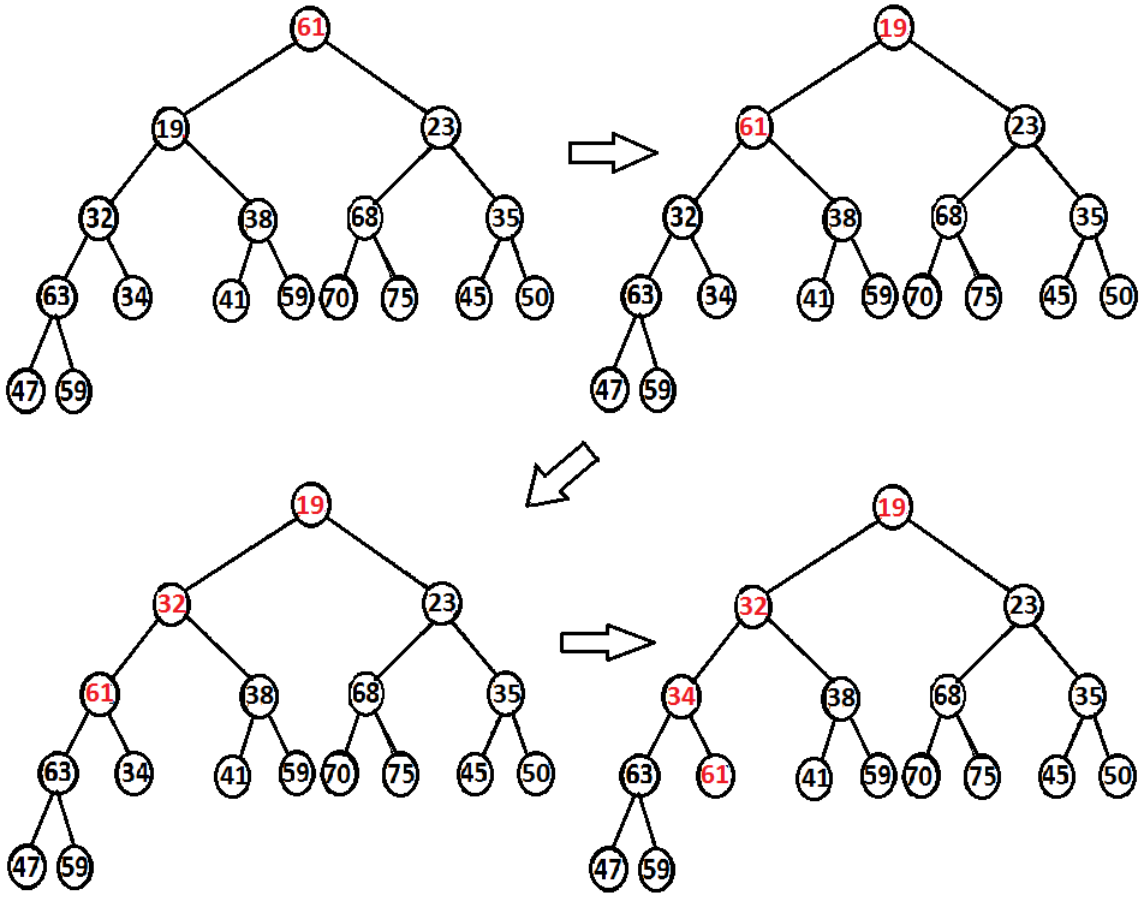


FIGURE 4.6 – Troisième itération

Itération 4
Supprimer la racine 19 de l'arbre qui est le minimum ; et on pose $T = \{8, 12, 15, 19\}$;
$N = N - \{19\} = \{32, 68, 23, 63, 34, 38, 59, 70, 75, 45, 35, 47, 59, 61, 50, 41\}$;
59 devient la racine du nouvel arbre ;
$min\{23, 32\} = 23$;
Comparer 23 à son pére 59 ;
23 est plus petit que 59 ;
Permuter entre 23 et 59 ;

$min\{68, 35\} = 35$;
Comparer 35 à son pére 59 ;
35 est plus petit que 59 ;
Permuter entre 35 et 59 ;

$min\{50, 45\} = 45$;
Comparer 45 à son pére 59 ;
45 est plus petit que 59 ;
Permuter entre 45 et 59 ;

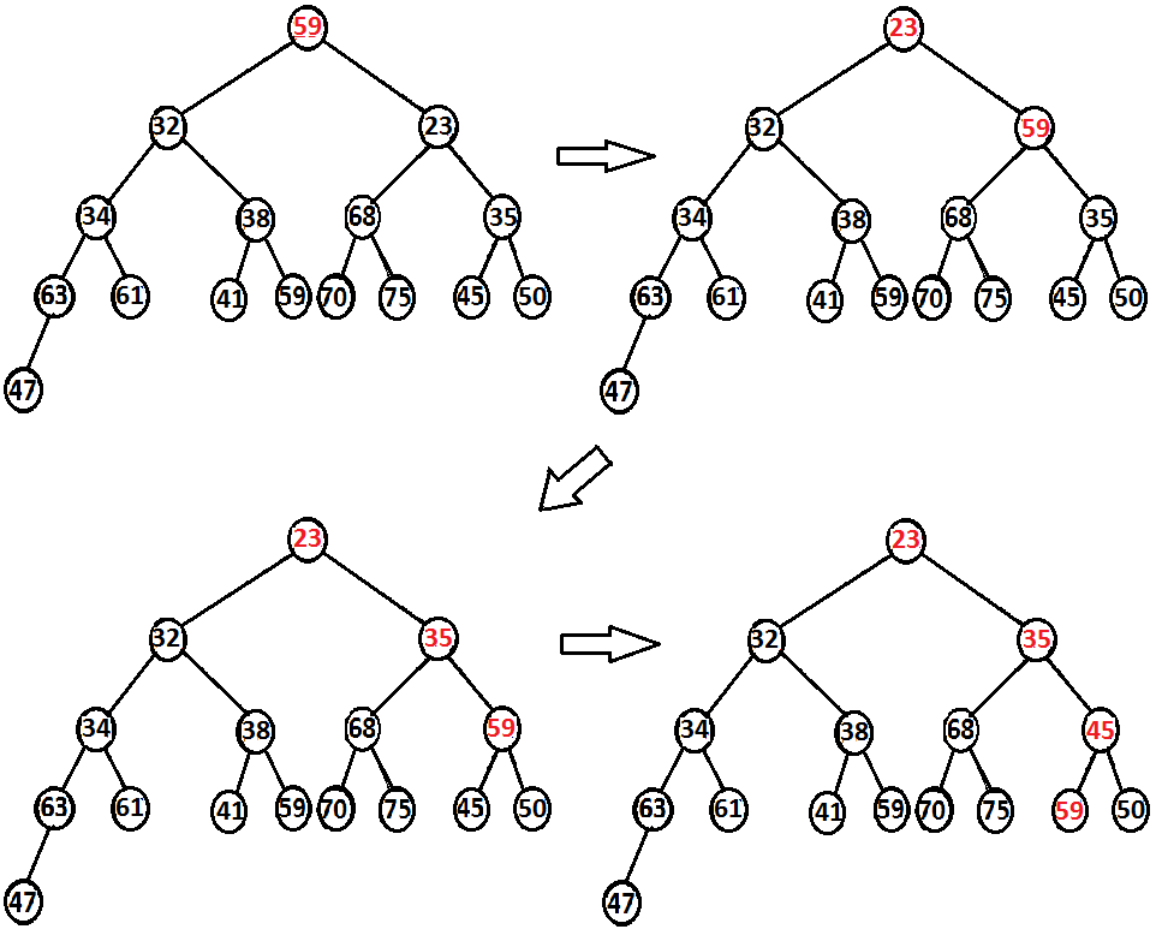


FIGURE 4.7 – Quatrième itération

On continue de la même manière jusqu'à ce que l'arbre ne contienne plus aucun élément.
$N = \emptyset$.
résultat : on obtient le tableau trié par ordre croissant
$T = \{8, 12, 15, 19, 23, 32, 35, 38, 41, 43, 45, 47, 50, 59, 61, 63, 68, 70, 75, 95\}$

4.1.2 Application en Java.

Présentation de Java
Le langage Java est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton, employés de Sun Microsystems, avec le soutien de Bill Joy (cofondateur de Sun Microsystems en 1982), présenté officiellement le 23 mai 1995 au SunWorld.
La société Sun a été ensuite rachetée en 2009 par la société Oracle qui détient et maintient désormais Java.
La particularité et l'objectif central de Java est que les logiciels écrits dans ce langage doivent être trés facilement portables sur plusieurs systèmes d'exploitation tels que UNIX, Windows, Mac OS ou GNU/Linux, avec peu ou pas de modifications. Pour cela, divers plateformes et frameworks associés visent à guider, sinon garantir, cette portabilité des applications développées en Java.

Le langage Java reprend en grande partie la syntaxe du langage C++, trés utilisé par les informaticiens. Néanmoins, Java a été épuré des concepts les plus subtils du C++ et à la fois les plus déroutants, tels que les pointeurs et références, ou l'héritage multiple contourné par l'implémentation des interfaces. Les concepteurs ont privilégié l'approche orientée objet de sorte qu'en Java, tout est objet à l'exception des types primitifs (nombres entiers, nombres à virgule flottante, etc.). Java a donné naissance à un système d'exploitation (JavaOS), à des environnements de développement (eclipse/JDK), des machines virtuelles (MSJVM (en), JRE) applicatives multiplate-forme (JVM), une déclinaison pour les périphériques mobiles/embarqués (J2ME), une bibliothèque de conception d'interface graphique (AWT/Swing), des applications lourdes (Jude, Oracle SQL Worksheet, etc.), des technologies web (servlets, applets) et une déclinaison pour l'entreprise (J2EE). La portabilité du bytecode Java est assurée par la machine virtuelle Java, et éventuellement par des bibliothèques standard incluses dans un JRE. Cette machine virtuelle peut interpréter le bytecode ou le compiler à la volée en langage machine. La portabilité est dépendante de la qualité de portage des JVM sur chaque OS.

Le Java Development Kit (JDK) désigne un ensemble de bibliothèques logicielles de base du langage de programmation Java, ainsi que les outils avec lesquels le code Java peut être compilé, transformé en bytecode destiné à la machine virtuelle Java.

Compilation Lors de l'ouverture de *Java*, la première étape et de créer un nouveau *Java Project*, en cliquant sur :
file⇒ *new* ⇒ *JavaProject*.

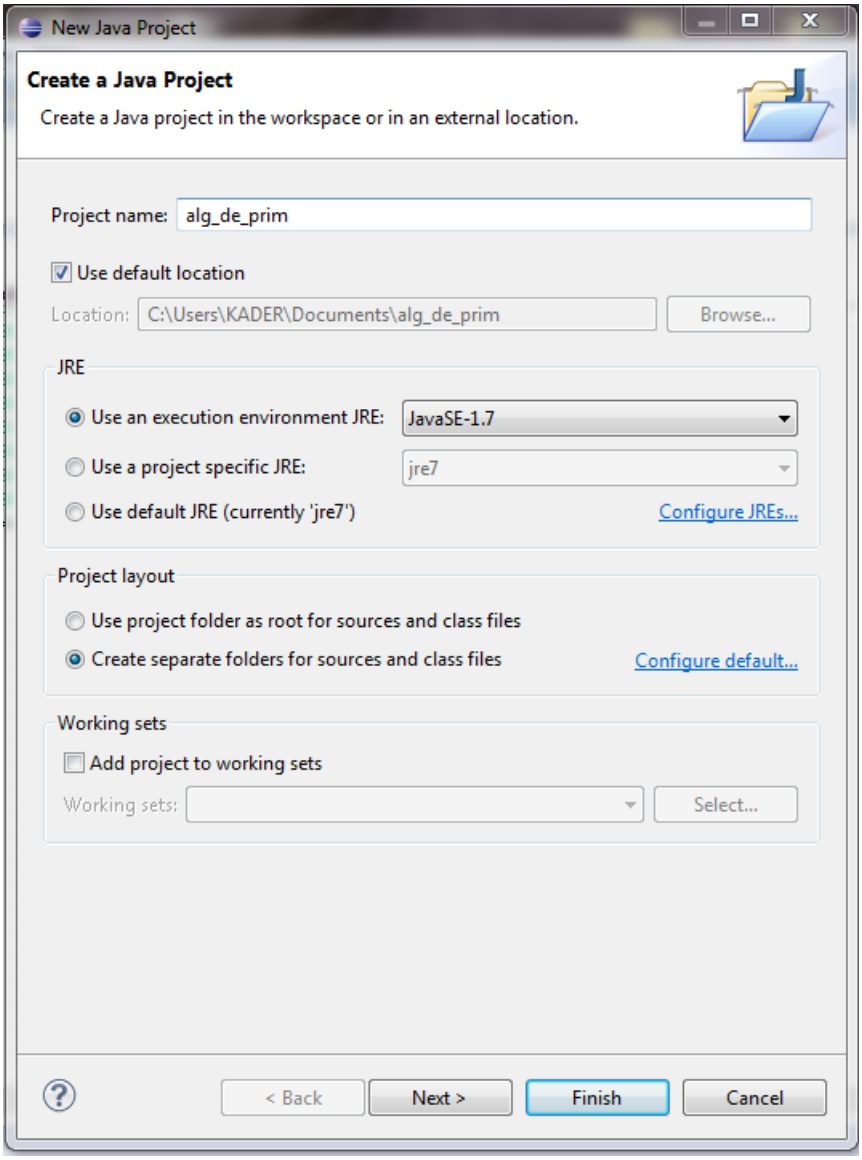


FIGURE 4.8 – Création d'un java project.

La prochaine étape et de créer une classe dans le java project précédement créé. file⇒ *new* ⇒ *class*.

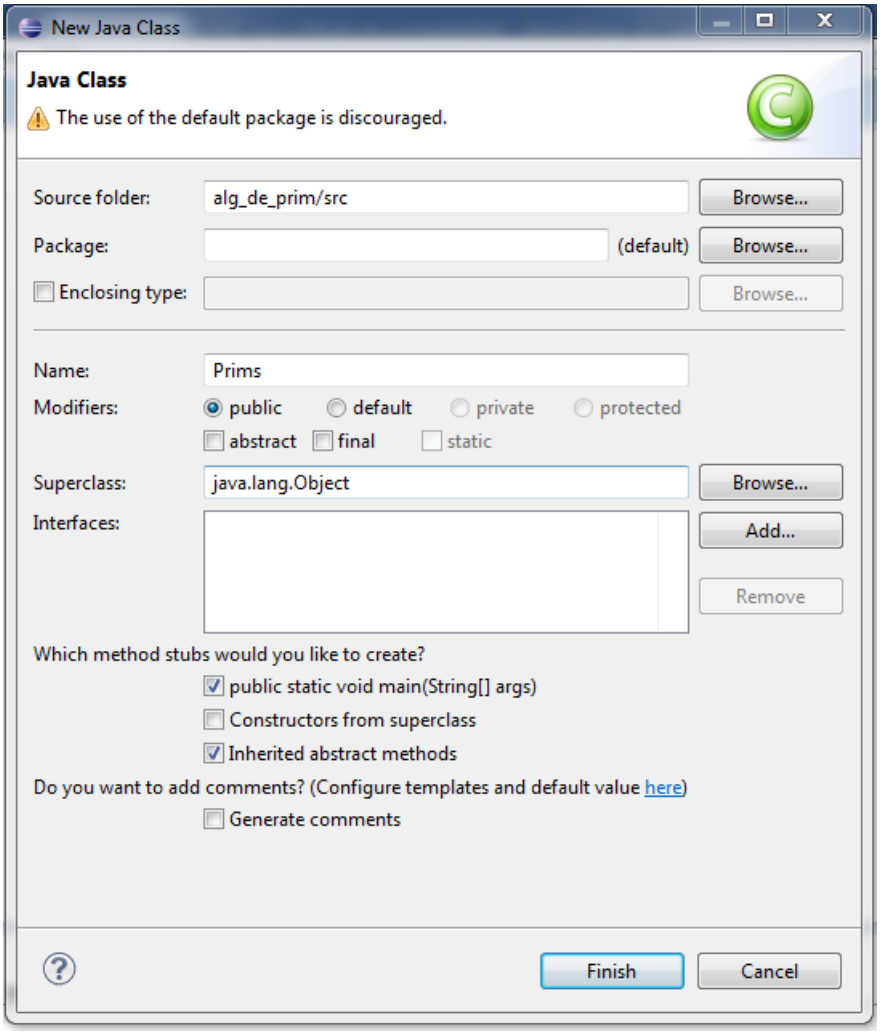


FIGURE 4.9 – Création d'une nouvelle classe.

La figure suivante montre l'interface du language Java.

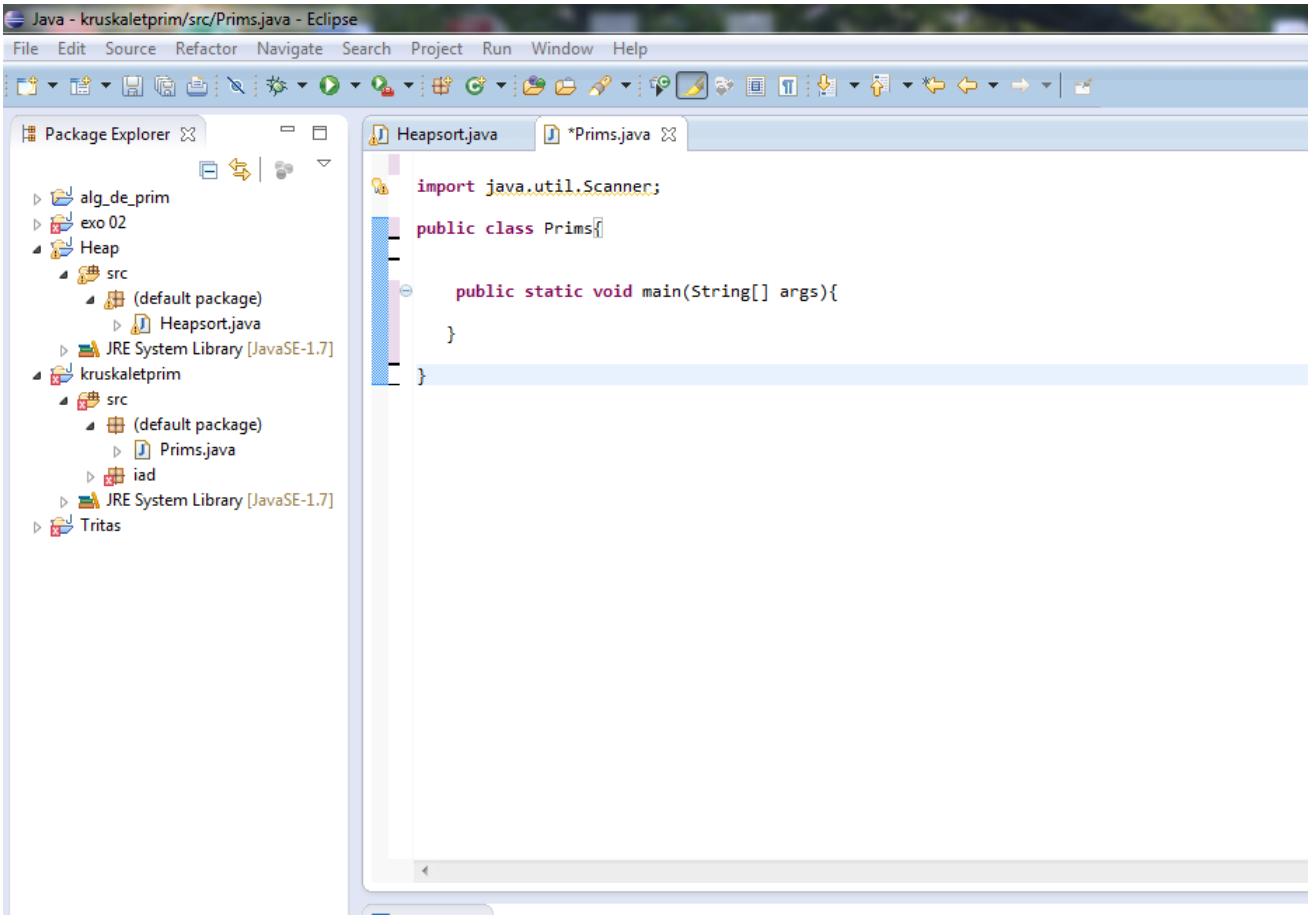


FIGURE 4.10 – Interface de Java

Compilation de l'algorithme de tri par tas Lors de la compilation de l'algorithme de tri par tas (tri par arbre) dans le language java, nous allons introduire le nombre de sommets (noeuds) que contient la suite de chiffres (23 dans cet exemple) ainsi que la suite qu'on veut trier (comme paramétres d'entrés), et le résultat retourné est la suite de 20 chiffres triée par ordre croissant. La figure ci-dessous montre le résultat retourné par le compilateur de Java.

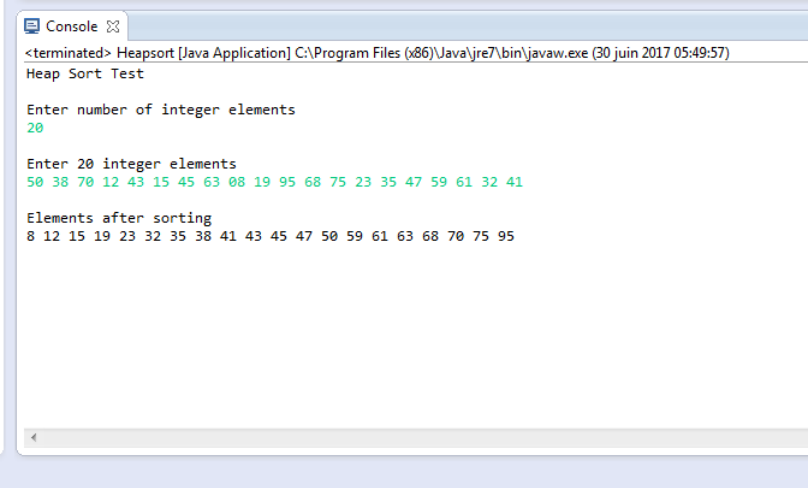


FIGURE 4.11 – Compilation de l'algorithme de Tri Par Tas

4.2 Problème de l'arbre couvrant de poids minimum

Problème Considérons n personnes qui se communiquent entre elles dans un réseau tels que la probabilité qu'un message confidentiel entre la personne i et la personne j soit intercepté par une personne étrangère k est p_{ij}.
L'objectif est de trouver une communication aussi sure que possible, autrement dit une communication qui minimise les probabilités d'interception du message.
Nous allons utiliser les algorithmes de **Kruskal** et **Prim** sur le graphe montrée dans figure ci-dessous pour trouver les arbres couvrants de poids minimums.

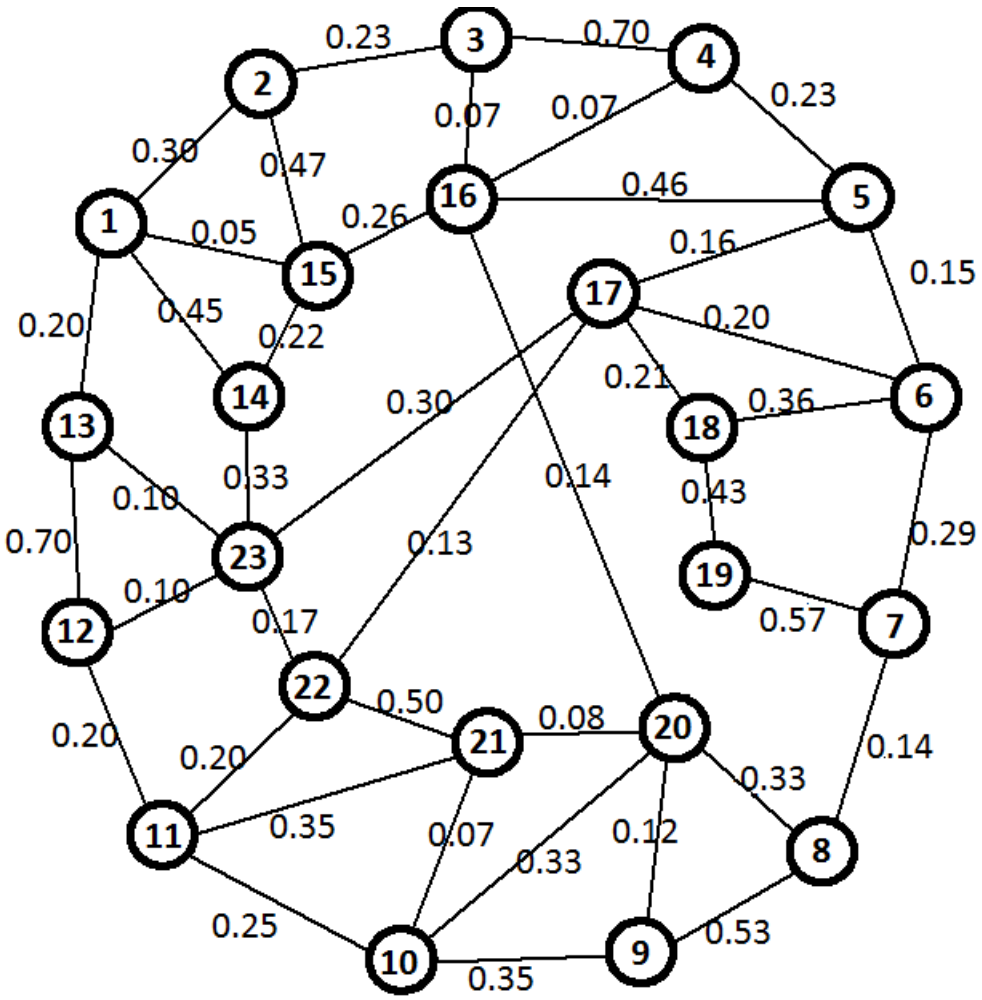


FIGURE 4.12 – Réseau de communication téléphonique .

4.2.1 Algorithme de Kruskal

Le tableau suivant représente l'ordonnancement croissant des arêtes du graphe selon les probabilités d'interception :

i	1	2	3	4	5	6	7	8	9
a_i	$(1,15)$	$(3,16)$	$(4,16)$	$(10,21)$	$(20,21)$	$(13,23)$	$(12,23)$	$(9,20)$	$(17,22)$
$P(a_i)$	(0.05)	(0.07)	(0.07)	(0.07)	(0.08)	(0.10)	(0.10)	(0.12)	(0.13)
i	10	11	12	13	14	15	16	17	18
a_i	$(16,20)$	$(7,8)$	$(5,6)$	$(5,17)$	$(22,23)$	$(6,17)$	$(11,12)$	$(11,22)$	$(1,13)$
$P(a_i)$	(0.14)	(0.14)	(0.15)	(0.16)	(0.17)	(0.20)	(0.20)	(0.20)	(0.20)
i	19	20	21	22	23	24	25	26	27
a_i	$(17,18)$	$(14,15)$	$(2,3)$	$(4,5)$	$(10,11)$	$(15,16)$	$(6,7)$	$(1,2)$	$(17,23)$
$P(a_i)$	(0.21)	(0.22)	(0.23)	(0.23)	(0.25)	(0.26)	(0.29)	(0.30)	(0.30)
i	28	29	30	31	32	33	34	35	36
a_i	$(8,20)$	$(10,20)$	$(14,23)$	$(9,10)$	$(11,21)$	$(6,18)$	$(18,19)$	$(1,14)$	$(5,16)$
$P(a_i)$	(0.33)	(0.33)	(0.33)	(0.35)	(0.35)	(0.36)	(0.43)	(0.45)	(0.46)
i	37	38	39	40	41	42			
a_i	$(2,15)$	$(21,22)$	$(8,9)$	$(7,19)$	$(3,4)$	$(12,13)$			
$P(a_i)$	(0.47)	(0.50)	(0.53)	(0.57)	(0.70)	(0.70)			

Initialisation : Soit : W $= \emptyset; i = 1; m = 42$.

1^{re} **itération :**
On a : $a_1 = (1,15)$;
Soit : $W = W \cup a_1$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1,15)\}, et|W| = 1$.
On a : $i \neq m$, alors on pose $i = i + 1 = 2$.

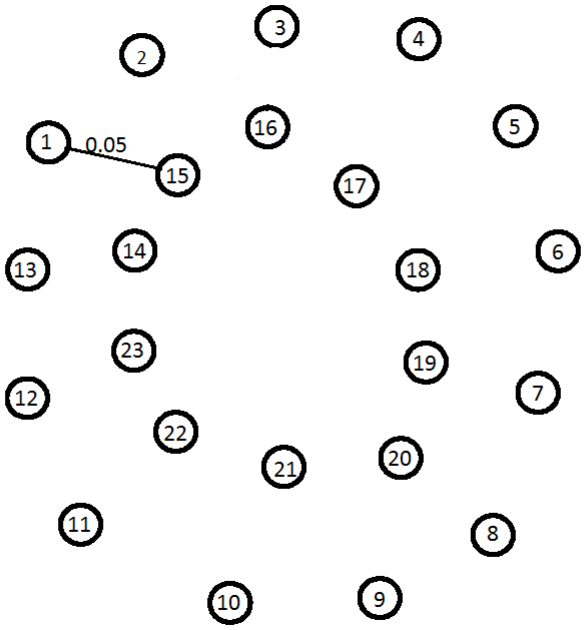


FIGURE 4.13 – Ajout de l'arête : (1,15).

2^{me} **itération :**

On a : $a_2 = (3, 16)$;
Soit : $W = W \cup a_2$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1, 15), (3, 16)\}, et|W| = 2$.
On a : $i \neq m$, alors on pose $i = i + 1 = 3$.

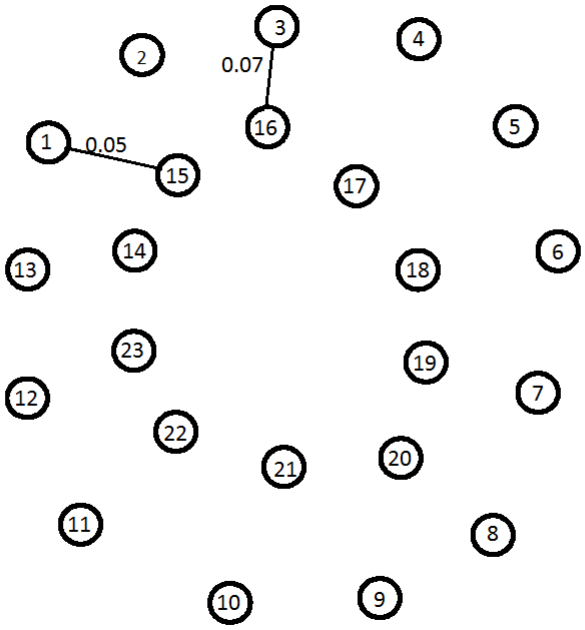


FIGURE 4.14 – Ajout de l'arête : (3,16).

3^{me} **itération :**
On a : $a_3 = (4, 16)$;
Soit : $W = W \cup a_3$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1, 15), (3, 16), (4, 16)\}, et|W| = 3$.
On a : $i \neq m$, alors on pose $i = i + 1 = 4$.

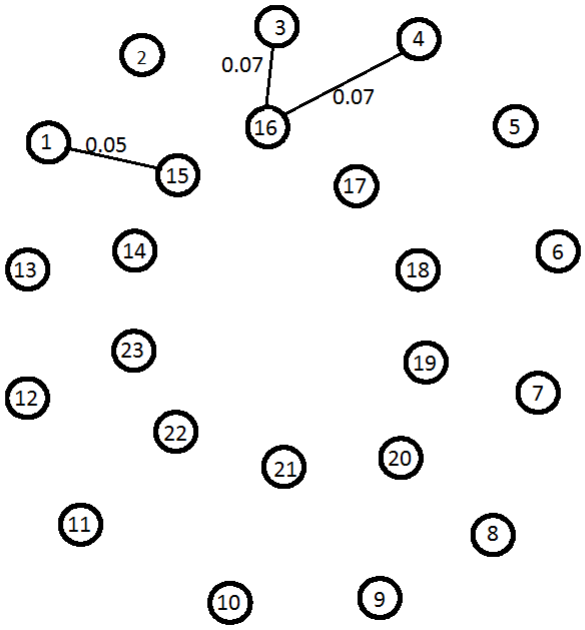


FIGURE 4.15 – Ajout de l'arête : (4,16).

4^{me} **itération :**
On a : $a_4 = (10, 21)$;
Soit : $W = W \cup a_4$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1, 15), (3, 16), (4, 16), (10, 21)\}, et|W| = 4$.
On a : $i \neq m$, alors on pose $i = i + 1 = 5$.

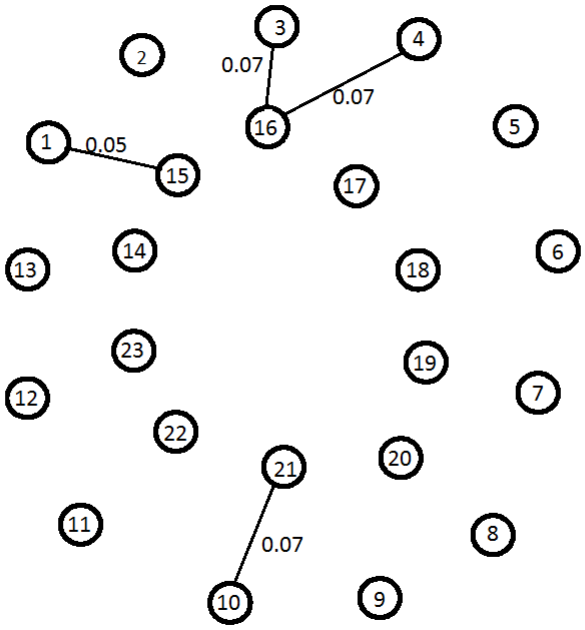


FIGURE 4.16 – Ajout de l'arête : (10,21).

5^{me} **itération :**
On a : $a_5 = (20, 21)$;
Soit : $W = W \cup a_5$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1, 15), (3, 16), (4, 16), (10, 21), (20, 21)\}, et|W| = 5.$
On a : $i \neq m$, alors on pose $i = i + 1 = 6$.

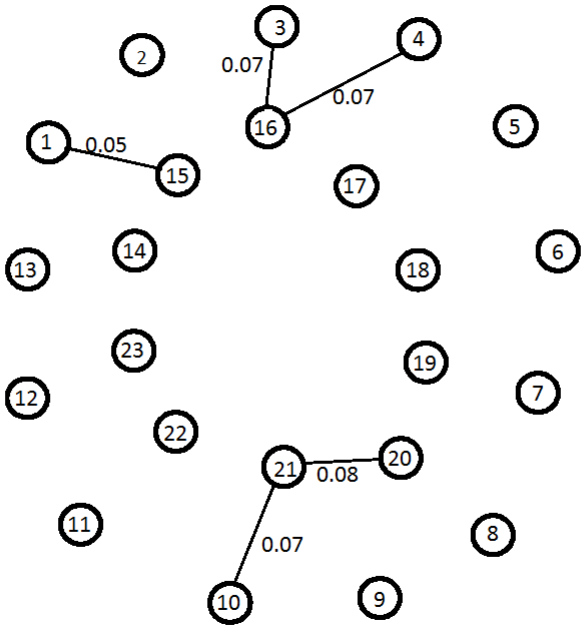


FIGURE 4.17 – Ajout de l'arête : (20,21).

6^{me} **itération :**
On a : $a_6 = (13, 23)$;
Soit : $W = W \cup a_6$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1, 15), (3, 16), (4, 16), (10, 21), (20, 21), (13, 23)\}, et |W| = 6$.
On a : $i \neq m$, alors on pose $i = i + 1 = 7$.

7^{me} **itération :**
On a : $a_7 = (12, 23)$;
Soit : $W = W \cup a_7$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1, 15), (3, 16), (4, 16), (10, 21), (20, 21), (13, 23), (12, 23)\}, et |W| = 7$.
On a : $i \neq m$, alors on pose $i = i + 1 = 8$.

8^{me} **itération :**
On a : $a_8 = (9, 20)$;
Soit : $W = W \cup a_8$;
Le graphe ne contient pas de cycle, alors :

On pose : $W = \{(1,15),(3,16),(4,16),(10,21),(20,21),(13,23),(12,23),(9,20)\}$,
et $|W| = 8$.
On a : $i \neq m$, alors on pose $i = i + 1 = 9$.

On continue de la même facon jusqu'a la 26^{me} itération.

Itération 26 **:**

On a : $a_{26} = (18,19)$;
Soit : $W = W \cup a_{26}$;
Le graphe ne contient pas de cycle, alors :
On pose : $W = \{(1,15),(3,16),(4,16),(10,21),(20,21),(13,23),(12,23),(9,20),(17,22),(16,20),$
$,(7,8),(5,6),(5,17),(22,23),(11,12),(1,13),(17,18),(14,15),(2,3),(4,5),(6,7),(18,19)\}$
On a $|W| = n - 1 = 23 - 1 = 22$; terminer.

L'arbre trouvé dans cette dernière itération est un arbre couvrant de poids minimum. Il est montré dans la Figure ci-dessous.

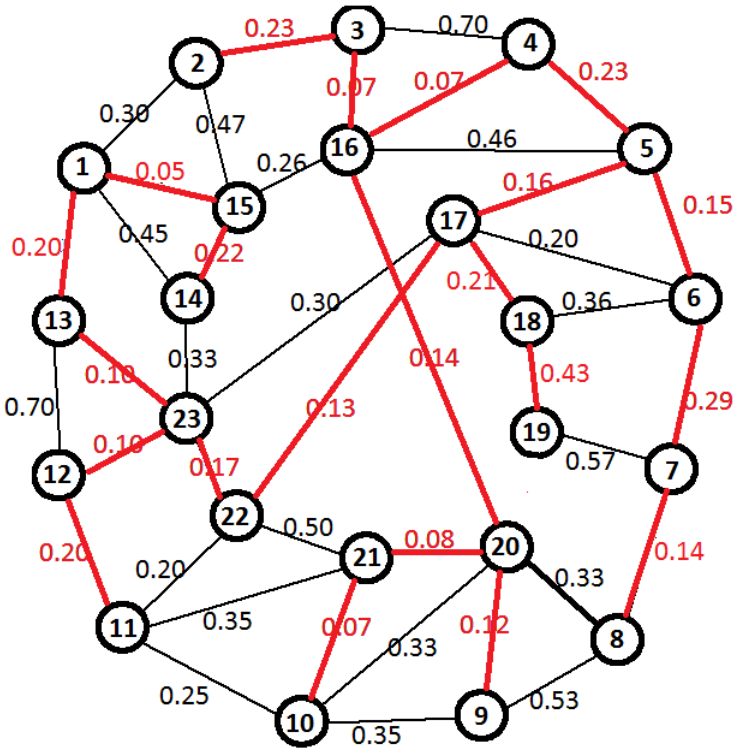


FIGURE 4.18 – Arbre de poids minimum

.

La probabilité minimale est :
$P(W) = \prod_1^{22}(1 - p_{ij})$
$P(W) = [(1 - 0.05) \times (1 - 0.07) \times (1 - 0.07)\ times(1 - 0.07)\ times(1 - 0.08) \times (1 - 0.10) \times (1 - 0.10) \times (1 - 0.12) \times (1 - 0.13) \times (1 - 0.14) \times (1 - 0.14) \times (1 - 0.15) \times (1 - 0.16) \times (1 - 0.17) \times (1 - 0.20) \times (1 - 0.20) \times (1 - 0.21) \times (1 - 0.22) \times (1 - 0.23) \times (1 - 0.23) \times (1 - 0.29) \times (1 - 0.43)] = 0.0181$

4.2.2 Algorithme de Prim

Prenons l'exemple précédant.
Initialisation
$I = \emptyset$.
$NI = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23\}$.

1^{re} **itération :**

On prend le sommet 1 comme un sommet de départ.
$min\{P(1,2), P(1,13), P(1,14), P(1,15)\} = P(1,15)$.
Donc on ajoute l'arête (1,15) au graphe.
$I = I \cup 15 = \{1, 15\}$.
$NI = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23\}$

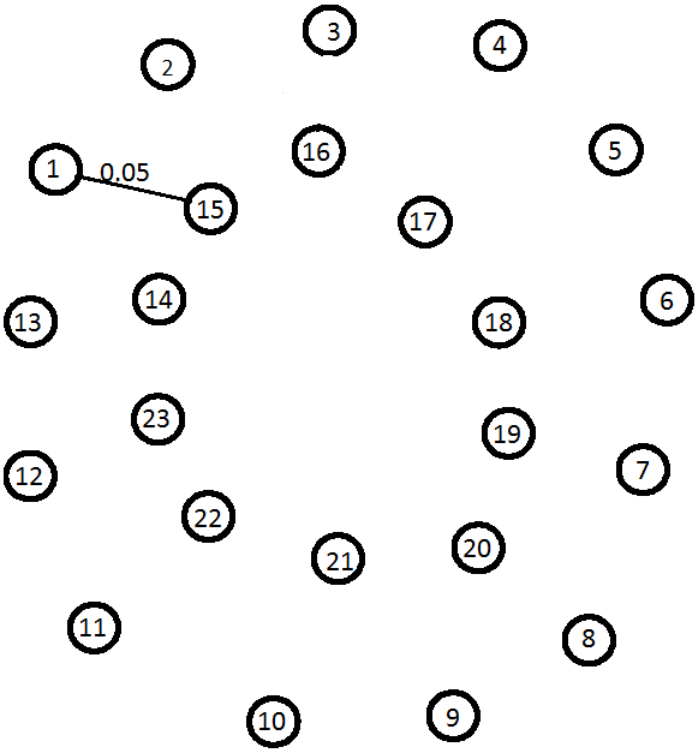


FIGURE 4.19 – Ajout de l'arête : (1,15).

2^{me} itération :

$min\{P(1,2), P(1,13), P(1,14), P(15,2), P(15,14), P(15,16)\} = P(1,13)$.
Donc on ajoute l'arête (1,13) au graphe.
$I = I \cup 13 = \{1, 15, 13\}$.
$NI = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 17, 18, 19, 20, 21, 22, 23\}$

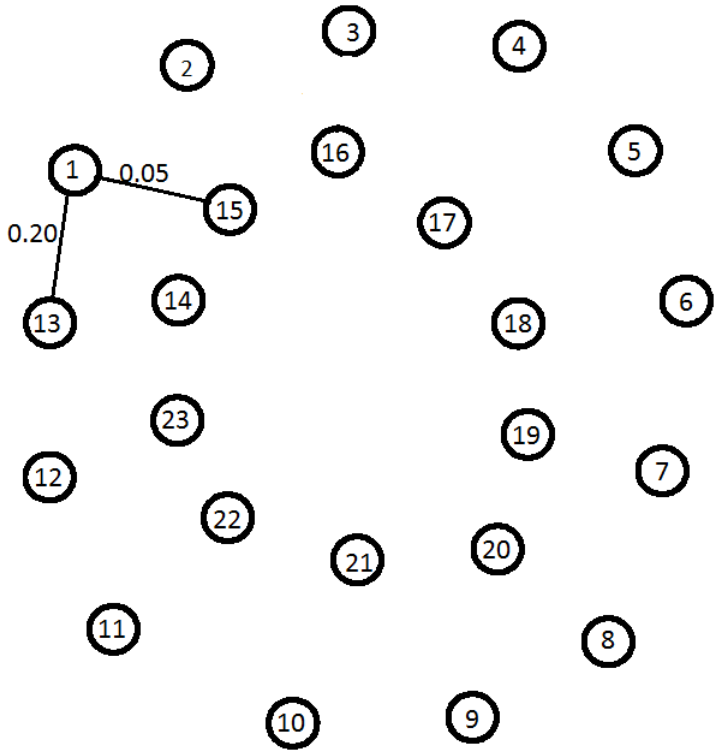


FIGURE 4.20 – Ajout de l'arête : (1,13).

3^{re} itération :

$min\{P(1,2), P(1,14), P(15,2), P(15,14), P(15,16), P(13,12), P(13,23\} = P(13,23)$.
Donc on ajoute l'arête (13,23) au graphe.
$I = I \cup 23 = \{1, 15, 13, 23\}$.
$NI = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 17, 18, 19, 20, 21, 22, \}$

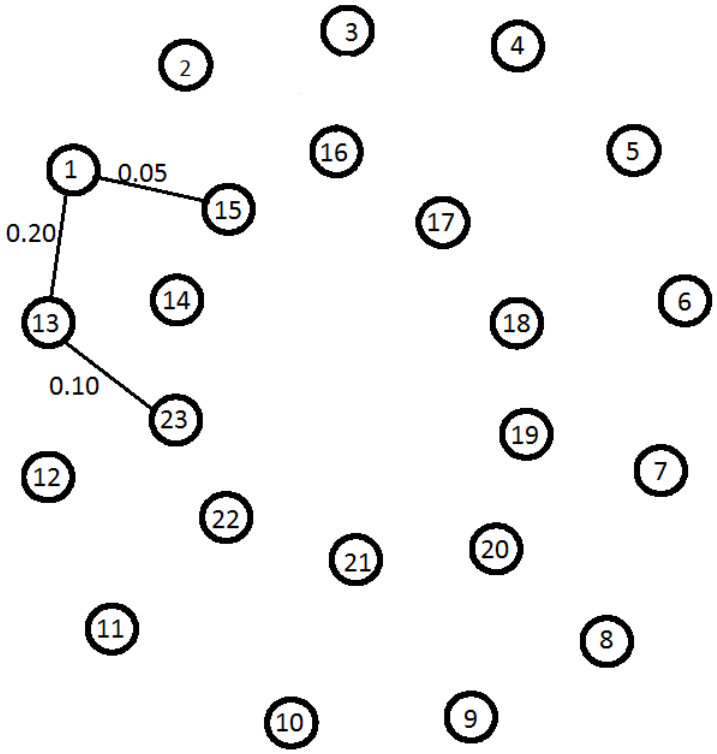


FIGURE 4.21 – Ajout de l'arête : (13,23).

4^{me} **itération :**

$min\{P(1,2), P(1,14), P(15,2), P(15,14), P(15,16), P(23,14), P(23,17), P(23,22), P(23,12)\} = P(23,12)$.
Donc on ajoute l'arête (23,12) au graphe.
$I = I \cup 12 = \{1, 15, 13, 23, 12\}$.
$NI = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 16, 17, 18, 19, 20, 21, 22, \}$

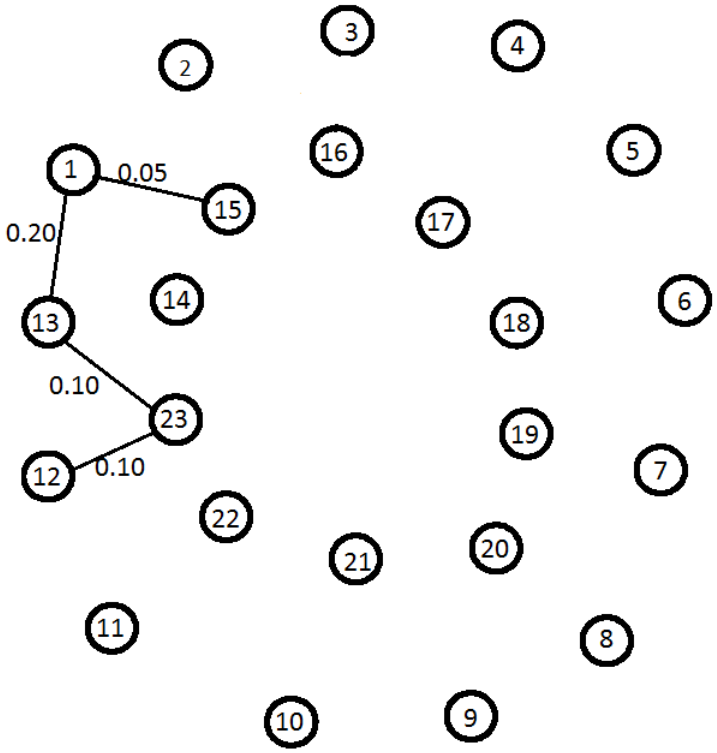


FIGURE 4.22 – Ajout de l'arête : (23,12).

5^{me} **itération :**

$min\{P(1,2), P(1,14), P(15,2), P(15,14), P(15,16), P(23,14), P(23,17), P(23,22), P(12,11)\} = P(23,22)$.
Donc on ajoute l'arête (23,22) au graphe.
$I = I \cup 22 = \{1, 15, 13, 23, 12, 22\}$.
$NI = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 16, 17, 18, 19, 20, 21, \}$

6^{me} **itération :**

$min\{P(1,2), P(1,14), P(15,2), P(15,14), P(15,16), P(23,14), P(23,17), P(12,11)$
$P(22,17), P(22,21), P(22,11)\} = P(22,17)$.
Donc on ajoute l'arête (22,17) au graphe.
$I = I \cup 17 = \{1, 15, 13, 23, 12, 22, 17\}$.
$NI = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 14, 16, 18, 19, 20, 21, \}$

7^{me} **itération :**

$min\{P(1,2), P(1,14), P(15,2), P(15,14), P(15,16), P(23,14), P(12,11), P(22,21)$,
$P(22,11), P(17,5), P(17,6), P(17,18)\} = P(17,5)$.
Donc on ajoute l'arête (17,5) au graphe.
$I = I \cup 5 = \{1, 15, 13, 23, 12, 22, 17, 5\}$.
$NI = \{1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 14, 16, 18, 19, 20, 21, \}$

8^{me} **itération :**

$min\{P(1,2), P(1,14), P(15,2), P(15,14), P(15,16), P(23,14), P(12,11), P(22,21)$,
$P(22,11), P(17,6), P(17,18), P(5,6), P(5,4), P(5,16)\} = P(5,6)$.
Donc on ajoute l'arête (5,6) au graphe.
$I = I \cup 6 = \{1, 15, 13, 23, 12, 22, 17, 5, 6\}$.
$NI = \{1, 2, 3, 4, 7, 8, 9, 10, 11, 14, 16, 18, 19, 20, 21, \}$

9^{me} **itération :**

$min\{P(1,2), P(1,14), P(15,2), P(15,14), P(15,16), P(23,14), P(12,11), P(22,21)$,
$P(22,11), P(17,18), P(5,4), P(5,16), P(6,18), P(6,7)\} = P(12,11)$.
Donc on ajoute l'arête (12,11) au graphe.
$I = I \cup 11 = \{1, 15, 13, 23, 12, 22, 17, 5, 6, 11\}$.
$NI = \{1, 2, 3, 4, 7, 8, 9, 10, 14, 16, 18, 19, 20, 21, \}$

on continue de la même facon jusqu'à l'itération 22 :

Itération 22 :

$min\{P(18,19)\} = P(18,19)$.
Donc on ajoute l'arête (18,19) au graphe.
$I = I \cup 19 = \{1, 15, 13, 23, 12, 22, 17, 5, 6, 11, 18, 14, 4, 16, 3, 20, 21, 10, 9, 2, 7, 8, 19\}$.
$NI = \emptyset$

Terminer.
L'arbre minimal obtenus est montré dans la figure ci-dessous :

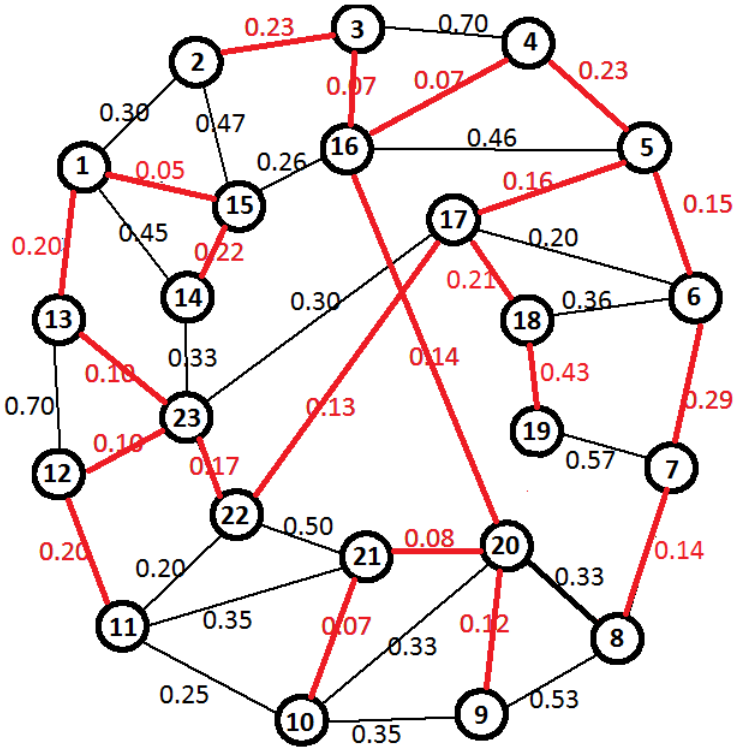


FIGURE 4.23 – Arbre couvrant minimale.

L'arbre trouvé dans cette dernière itération est un arbre couvrant de poids minimum.
La probabilité minimale est :
$P(W) = 1 - \prod_1^{22}(1 - p_{ij})$
$P(W) = 1 - [(1 - 0.05) \times (1 - 0.07) \times (1 - 0.07)\ times(1 - 0.07)\ times(1 - 0.08) \times (1 - 0.10) \times (1 - 0.10) \times (1 - 0.12) \times (1 - 0.13) \times (1 - 0.14) \times (1 - 0.14) \times (1 - 0.15) \times (1 - 0.16) \times (1 - 0.17) \times (1 - 0.20) \times (1 - 0.20) \times (1 - 0.21) \times (1 - 0.22) \times (1 - 0.23) \times (1 - 0.23) \times (1 - 0.29) \times (1 - 0.43)] = 0.0181$

Compilation de l'algorithme de Prim Lors de la compilation de l'algorithme de Prim, la matrice d'adjacence du réseau (graphe) représente le paramètre d'entré, et le programme renvoie l'arbre minimale comme résultat (les arêtes de l'arbre). La figure ci-dessous montre la matrice d'adjacence de l'exemple de communication téléphonique vu précédement tels que les poids sur les arcs sont les pourcentages d'interception des messages téléphoniques.

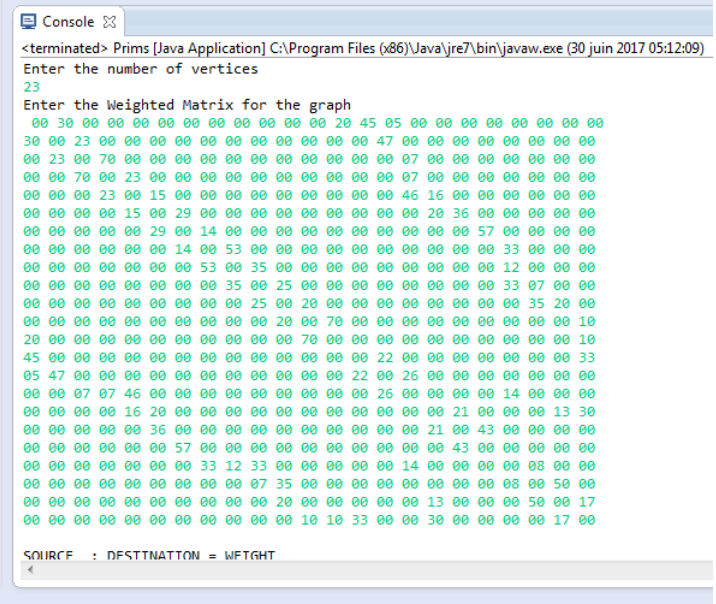


FIGURE 4.24 – Matrice d'adjacence du graphe.

Enfin, l'arbre couvrant de poids minimum, calculé à l'aide de l'algorithme de Prim est définit ci-dessous par ses arêtes, de telle sorte qu'une arête est d'̀efinie par sa source, sa déstinnation et son poids (weight).

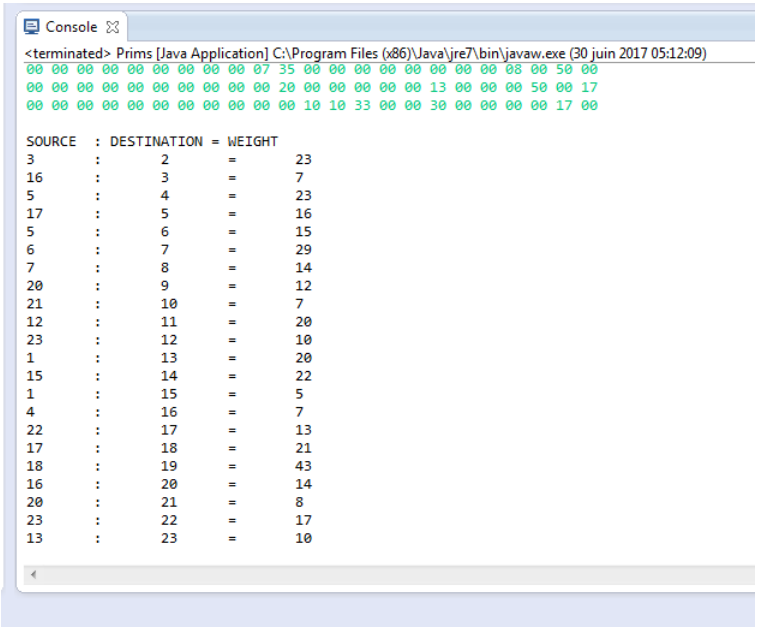


FIGURE 4.25 – Arbre couvrant de poids minimums.

Conclusion

Les graphes constituent donc une méthode de pensée qui permet de modéliser une grande variété de problèmes concrets en se ramenant à l'étude de sommets et d'arcs.
La théorie des graphes permet de générer des circuits optimisés et de gérer des réseaux (routiers, de communication, d'eau de gaz,..), d'ordonnancer des tâches et de gérer des plannings. Elle est la clé de l'intelligence artificielle avec la notion du (plus court chemin).
Ces nombreuses applications font de la théorie des graphes un outil appréciable d'aide à la décision (en recherche opérationnelle).
Notre travail consiste a donné aux lecteurs quelques techniques d'optimisation dans les graphes en générale, utilisable pour résoudre les problèmes rencontré. et l'objectif exacte de notre travail est de résoudre des problèmes en utilisant ces techniques particulièrement dans un graphe probabiliste et un arbre binaire.

le premier consiste à résoudre un problème de télécommunication entre vingt trois personnes en prenant compte de la fiabilité de communication c'est à dire qu'il y a une probabilité qu'un message entre deux personnes soit intercepté par une personnes étrangére, en utilisant l'algorithme de Kruskal et Prim.
Le deuxième consiste aà faire un tri déléments désordonnés. Et les réordonner de tel sorte quelle soit en ordre croissant afin de faciliter la recherche d'un élément parmi tant d'autres.

Bibliographie

[1] Arbres de recherche, Sylvie Hamel, Université de Montréal, 2009.

[2] C. Berge. Graphs and hypergraphs. Dunod, Paris, 1973.

[3] C. Berge. Théories des graphes et applications, 1 er édition, paris, lavoisier2008

[4] Cours: arbres binaires et de recherche Christophe Gonzales – Pierre-Henri Wuillemin -– Université Paris 6, p16.

[5] Cours Andrea G. B. Tettamanzi.Université de Nice Sophia Antipolis.Département Informatique.2012.

[6] Cours .Algorithmes et structures de données génériques, Divay M.,2004, Dunod

[7] Cour: Les Arbres. Renaud Dumont, Ulg 2009 2010.

[8] Cours 06 d'algorithmique. F. Lévy .UT De Villetaneuse .Dép informatique-2003.3004, (p1p3)

[9] D. WEST, Introduction to graph theory, 2nd, Prentice HALL, 2001.

[10] H.M. Mulder. The interval function of a graph. Mathematical centre tracts 132, Mathemaisch centrum, Amesterdam, (1980).

[11] J. Michel Hélary, algorithmes des graphes, 2004.

[12] Livre Didier Müller: Introduction à la théorie des graphes ,cahier N6 .CRM 2012(p2-page16).

[13] Livre théorie des graphe J.A. Bondy et U.S.R. Murty Traduit de l'anglais par F. Havet, 2008.

[14] Mémoire. François Schwarzentruber . ALGO1 - ENS Rennes p28,p30.

[15] M. GONDRON, M, MINOUX, Graphes et algorithmes,4éme édition, lavoisier, 2009.

[16] M. GONDRON, M, MINOUX, Graphes et algorithmes, Eyrolles, Paris, 1984

[17] Ph. Lacomme, M. Prins, M. Sevaux, Algorithmes de graphes, Eyrolles, 2003.

[18] Thèse de Wajdi Tekara, problème d'arbres couvrant de poids minimum probabiliste, université de Toulouse.

Resumé

L'objectif de ce travail est de montrer l'utilité de la "Théorie des graphes "en optimisation, ce mémoire contribue également à montrer l'importance de l'utilisation des arbres , (plus particulièrement les arbres binaires) . Dans la résolution de certains problèmes de la recherches opérationnelles , et cela en cherchant quelques problèmes d'optimisation pour les quelles nous donnons quelques algorithmes de résolution pour chaque problèmes, suivie d'une résolution, en faisant appel à un programme réalisé sous JAVA .

Mots-clés : Arbre binaire, Optimisation, Théorie des graphes, JAVA

Abstract

The objective of this work is to show the utility of the "theory of graphs" in optimization, this memory also helps to show the importance of the use of trees (especially binary trees) in solving some problems Of the operational research, and that while looking for some optimization problems for which we give some algorithms of resolution for each problem, followed by a resolution, using a program realized under JAVA.

Keyword : binary trees, theory of graphs, Optimization, JAVA.