République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

---

CENTRE UNIVERSITAIRE DE MILA

INSTITUT DES SCIENCES ET DE LA TECHNOLOGIE

Mémoire de fin d'étude
Présenté pour l'obtention du diplôme de

LICENCE ACADEMIQUE

Domaine : Mathématiques et Informatique
Filière : Informatique
Spécialité : Informatique Académie

Thème

L'implémentation des algorithmes de traitement des graphes

Présenté par: -Belbel Abderahime
- Bousmina Zaid

Sous la direction de : GUETTICHE Mourad

Année universitaire 2010-2011

Remerciement

C'est avec l'aide de Dieu qu'a vu le jour ce présent travail.

Ensuite, il n'aurait pas pu être achevé sans le soutien, les conseils et les encouragements de certaines personnes auxquelles nous tenons ici à exprimer nos sincères remerciements.

En premier lieu, nous exprimons toute notre gratitude pour Notre Promoteur, Monsieur M. GUETTICHE pour ses précieux conseils, sa disponibilité, la confiance qu'il nous a toujours témoigné et la sollicitude dont il nous a entouré, et ce tout au long de l'élaboration du présent travail.

Nous n'oublions pas non plus Nos Enseignants, qui tout au long du cycle d'études au centre universitaire de MILA, nous ont transmis leur savoir.

Nous adressons une pensée particulièrement affective à Nos Amis de CUM qui ont rendu agréables nos longues années d'études.

Nous tenons enfin à remercier tous ceux qui ont collaborés de près ou de loin à l'élaboration de ce travail. Qu'ils acceptant nos humble remerciements.

Belbel & *Bousmina*

Dédicace

À mes très chers parents à qui j`ai transmis mon stress et anxiété, pour leur affection, leur patience, leur soutien et leurs encouragements qui m` ont permis d'arriver au bout de ce travail.

À mes frères que je les aime énormément.

À mon binôme Bousmina que j`estime beaucoup.

À toute ma famille, à tous mes amis surtout Djaafer et Yasser.

Abderahime

A Ma Mère.

A Mon Père.

A mes chers frères et sœurs Amine et Kaiss et Alia

et Islam.

A Tous les Membres de Ma Famille.

À mon binôme Belbel que j`estime beaucoup.

A tous mes Amis Djaafer et Yasser et les Collègues de Promotion.

Je dédie ce modeste travail.

Zaid

Table des matières

Chapitre 3 : *Description de l'application*

Liste des figures

Introduction générale

La représentation d'un problème par un dessin, un plan, une esquisse contribue souvent à sa compréhension. Le langage des graphes est construit, à l'origine, sur ce principe. Nombres de méthodes, de propriétés de procédures ont été pensées ou trouvées a partir d'un schéma pour être ensuite formalisées et développées. Chacun d'entre nous a, au moins une fois, vu ou utilise un plan de métro, une carte de lignes ferroviaires, un plan électrique, un arbre généalogique ou un organigramme d'entreprise ; ainsi tout le monde sait plus ou moins intuitivement ce qu'est un graphe. Toutefois, entre cette notion vague ou des points, représentant des individus, des objets, des lieux ou des situations, sont relies par des flèches, il y a une longue élaboration des concepts. La première difficulté à laquelle on peut être confronté concerne la terminologie (très abondante en théorie des graphes) [1].

Notre principale objectif est de générer automatiquement la solution des problèmes du monde réel après la modélisation mathématique et l'implémentation algorithmique, nous proposons donc une petite application permet à nous d'introduire un graphe et faisons certains traitements sur le graphe saisit.

Afin de réaliser ça, nous avons divisé notre mémoire en trois parties principales, on peut les citer comme suite :

-Le premier chapitre concerne les notions fondamentales sur la théorie des graphes (Généralités sur les graphes) et ses applications dans l'informatique.

-La deuxième chapitre explique notre choix pour quoi le langage java et pour quoi l'environnement NetBeans IDE exact, ainsi que la structure des graphes en java.

- Le dernier chapitre on parle de notre travail réel, on explique notre application et son mode de fonctionnement avec des illustrations.

Chapitre 1:

Notions fondamentales de la théorie des graphes

I. Bref historique de la théorie des graphes

La théorie des graphes est un outil puissant de modélisation et de résolution des problèmes concrets et aussi est une des piliers de la recherche opérationnelle qui est constitué une discipline carrefour ou se rencontre l'économie, les mathématiques et l'informatique, et leurs objectifs est de trouver les solutions optimales pour des problèmes économiques en utilisant des méthodes mathématiques capable d'être programmées par l'ordinateur [2].

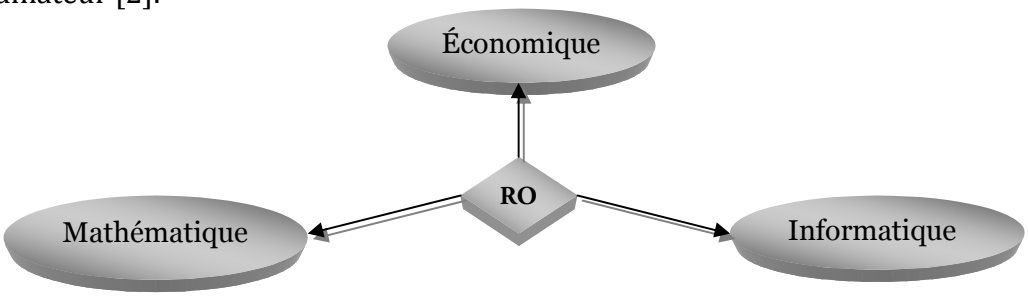

FIG .1- *Illustration de la recherche opérationnelle*

Le premier problème connu d'utilisation d'un graphe pour résoudre un problème est celui des « 7 ponts de Königsberg », résolu en 1735 par le mathématicien suisse Leonhard Euler : La ville de Königsberg (Prusse orientale) comptait 7 pont, l'histoire veut que Léonard Euler, en visite dans cette ville, ait eu à résoudre le problème qui préoccupait fortement ses habitants : Est-il possible de trouver un circuit qui emprunte une fois et une seule chacun des sept ponts de la ville ?

Pour cela, l'idée est de commencer par traduire l'énoncé du problème par un schéma : Chaque lieu d e la ville est repéré par sa position géographique : N pour le nord de la ville ; S pour le sud de la ville, O pour l'ouest et I pour île. Chaque pont sera alors représenté par un « trait » reliant ces lieux entre eux .Cette modélisation s'appelle un graphe [3].

II. Définition des concepts de base

II.1. Définition d'un graphe

On appelle graphe G le couple (S, A) constitué d'un ensemble S de sommets et d'un ensemble A d'arcs. Chaque arc de A relie deux sommets de S.

Un graphe peut être orienté ou non selon le sens que l'on donne aux arcs, les deux sommets qu'ils relient pouvant être considérés comme ordonné ou non. Et peut être valuée ou non valuée.

- ***Graphes orientés* :**

En donnant un sens aux arêtes d'un graphe, on obtient un graphe orienté.

Un graphe orienté fini $G = (V, E)$ est défini par l'ensemble fini $V = \{v_1, v_2, ..., v_n\}$ ($|V| = n$) dont les éléments sont appelés **sommets**, et par l'ensemble fini $E = \{e_1, e_2, ..., e_m\}$ ($|E| = m$) dont les éléments sont appelés arcs.

Un **arc** e de l'ensemble E est défini par une paire **ordonnée** de sommets. Lorsque $e = (u, v)$, on dira que l'arc e va de u à v. On dit aussi que u est l'extrémité initiale et v l'extrémité finale de e.

- ***Graphes non orientés* :**

Un **graphe** fini $G = (V, E)$ est défini par l'ensemble fini $V = \{v1, v2, ..., vn\}$ ($|V| = n$) dont les éléments sont appelés **sommets**, et par l'ensemble fini $E = \{e1, e2, ..., em\}$ ($|E| = m$) dont les éléments sont appelés **arêtes**.

Une arête e de l'ensemble E est définie par une paire **non-ordonnée** de sommets, appelés les extrémités de e. Si l'arête e relie les sommets a et b, on dira que ces sommets sont **adjacents**, ou **incidents** avec e, ou encore que l'arête e est **incidente** avec les sommets a et b.

On appelle **ordre d'un graphe** le nombre de sommets (n) de ce graphe.

- ***Graphe valué (pondéré)* :**

On appelle un graphe valué, un graphe, orienté ou non, dont les arêtes (ou les arcs) possèdent un poid. C'est-à-dire que les arêtes possèdent un nombre qui identifie le coût

de passage d'un sommet à un autre .cette représentation est utilisée par exemple dans la représentation d'un réseau routier pour indiquer le temps ou la distance joignant deux villes.

II.2. Modes de représentation d'un graphe

Un certain nombre de représentations existent pour décrire un graphe. En particulier, elles ne sont pas équivalentes du point de vue de l'efficacité des algorithmes.

On distingue principalement la représentation par matrice d'adjacence, par matrice d'incidence sommets-arcs (ou le sommets-arêtes dans le cas non orienté) et par listes des successeurs.

- ***Matrice d'adjacence*** **:**

La matrice d'adjacence fait correspondre les sommets origine des arcs (placés en ligne dans la matrice) aux sommets destination (placés en colonne). Dans le formalisme *matrice booléenne*, l'existence d'un arc (x_i, x_j) se traduit par la présence d'un 1 à l'intersection de la ligne *xi* et de la colonne x_j ; l'absence d'arc par la présence d'un 0.

- ***Matrice d'incidence sommets-arcs*** **:**

Un graphe peut être représenté par une matrice n x m (n = |X| et m = |U|), dite d'incidence, pouvant contenir uniquement les valeurs 0, 1, -1. Chaque ligne de la matrice est associée à un nœud et chaque colonne à un arc. Ainsi, une case indique la relation qu'il existe entre un nœud et un arc.

_ (0) signifie que le nœud et l'arc ne sont pas adjacents,

_ (1) signifie que le nœud est l'extrémité initiale de l'arc,

_ (-1) signifie que le nœud est l'extrémité terminale de l'arc.

- ***Listes d'adjacence (des successeurs)*** **:**

L'avantage de la représentation par listes d'adjacence, par rapport à celle par matrice d'adjacence, est le gain obtenu en place mémoire ; ce type de représentation est donc mieux adapté pour une implémentation. Le but est de représenter chaque arc par son extrémité finale, son extrémité initiale étant définie implicitement. Tous les arcs émanant d'un même sommet sont liés entre eux dans une liste. A chaque arc sont donc associés le nœud destination et le pointeur au prochain sommet dans la liste.

On crée deux tableaux *LP* (tête de listes) de dimension *(n + 1)* et *LS* (liste de Successeurs) de dimension *m* (cas orienté) ou 2*m* (cas non orienté). Pour tout *i*, la liste des successeurs de *i* est dans le tableau *LS* à partir de la case numéro *LP*(*i*) [4].

II.3. Connexité dans les graphes

- ***Chaîne – Cycle*** **:**

Une *chaîne* est une séquence d'arcs telle que chaque arc ait une extrémité commune avec le suivant. Un *cycle* est une chaîne qui contient au moins une arête, telle que toutes les arêtes de la séquence sont différentes et dont les extrémités coïncident.

- ***Chemin – Circuit*** **:**

Ce sont les mêmes définitions que les précédentes mais en considérant des concepts orientés.

Le terme de *parcours* regroupe les chemins, les chaînes, les circuits et les cycles. Un parcours est :

– *élémentaire* : si tous les sommets qui le composent sont tous distincts ;

– *simple* : si tous les arcs qui le composent sont tous distincts ;

– *hamiltonien* : passe une fois et une seule par chaque sommet du graphe ;

– *eulérien* : passe une fois et une seule par chaque arc du graphe ;

- ***Graphe connexe*** **:**

Un graphe $G = (X, U)$ est *connexe* si $\forall\, i, j \in X$, il existe une Chaîne entre *i* et *j*.

On appelle *composante connexe* le sous-ensemble de sommets tels qu'il existe une chaîne entre deux sommets quelconques.

Un graphe est connexe s'il comporte une composante connexe maximale et une seule.

Chaque composante connexe est un graphe connexe.

- ***Graphe fortement connexe*** **:**

Un graphe $G = (X, U)$ est *fortement connexe* si $\forall\, i, j \in X$, il existe un chemin entre *i* et *j*.

Une *composante fortement connexe* (CFC) est un sous-ensemble de sommets tel qu'il existe un chemin entre deux sommets quelconques. Une *CFC maximale* (CFCM) est un ensemble maximal de CFC. Les différentes CFCM définissent une partition de *X*.

Un graphe est fortement connexe s'il comporte une seule CFCM.

II.4. Cocycles et cocircuit

Soit $\mathcal{A} \subset \mathcal{X}$, l'ensemble des arcs incidents à l'ensemble A est noté :

$$\mathcal{W}(\mathcal{A}) = \mathcal{W}^{+}(\mathcal{A}) \cup \mathcal{W}^{-}(\mathcal{A}).$$

Un *cocycle* est un ensemble d'arcs de la forme $w(\mathcal{A})$ non vide et partitionné en deux classes $\mathcal{W}^{+}(\mathcal{A})$ et $\mathcal{W}^{-}(\mathcal{A})$.

Un *cocircuit* est un cocycle, $\mathcal{W}(\mathcal{A}) = \mathcal{W}^{+}(\mathcal{A})$ *ou* $\mathcal{W}(\mathcal{A}) = \mathcal{W}^{-}(\mathcal{A})$, dans lequel tous les arcs sont orientés dans le même sens soit vers l'extérieur de $\mathcal{A}$, soit vers l'intérieur [4].

II.5. Graphe particuliers

Il existe plusieurs types de graphes on peut citer quelque types :

–*Graphe partiel* $G = (X, U)$ et $U_p \subset U$. $G_p = (X, U_p)$ est un *graphe partiel* de G. (On peut ainsi obtenir des sommets isolés...).

–*Sous-graphe* $G = (X, U)$ et $X_s \subset X$. $G_s = (X_s, V)$ est un *sous-graphe* de G, où V est la restriction de la fonction caractéristique de U à X_s. $V = \{(x, y)/(x, y) \in U \cap X_s \times X_s\}$. $\forall x_i \in X_s$, $\Gamma_s(x_i) = \Gamma(x_i) \wedge X_s$.

– *Sous-graphe partiel* Combine les deux définitions précédentes.

– Graphe *réflexif* : $\forall x_i \in X$, $(x_i, x_i) \in U$ (*un graphe possédant une boucle sur chaque sommet*).

– Graphe *irréflexif* : $\forall x_i \in X$, $(x_i, x_i) \notin U$.

– Graphe *symétrique* : $\forall x_i, x_j \in X$, $(x_i, x_j) \in U \Longrightarrow (x_j, x_i) \in U$.

–Graphe *asymétrique* : $\forall x_i, x_j \in X$, $(x_i, x_j) \in U \Longrightarrow (x_j, x_i) \notin U$ (si G est asymétrique, G est irréflexif).

–Graphe *antisymétrique* : $\forall x_i, x_j \in X$, $(x_i, x_j) \in U$ et $(x_j, x_i) \in U \Longrightarrow x_i = x_j$ (si *G est* asymétrique, G est aussi antisymétrique).

–Graphe *transitif* : $\forall x_i, x_j \in X$, $(x_i, x_j) \in U$, $(x_j, x_k) \in U \Longrightarrow (x_i, x_k) \in U$.

–Graphe *complet* : $\forall x_i, x_j \in X$, $(x_i, x_j) \notin U \Longrightarrow (x_j, x_i) \in U$.

–Graphe *biparti* : s'il est possible de partitionner X l'ensemble des sommets en deux sous ensembles X_1 et X_2 tels que chaque arc ait exactement une extrémité dans X_1 et une extrémité dans X_2.

– une *clique* *est aussi est un graphe particulier est définit comme la suite* :

Est un ensemble des sommets d'un sous-graphe complet. Soit $C \subset X$ une clique de G non orienté : $\forall (x_i, x_j) \in C$, $(x_i, x_j) \in U$ (2 sommets distincts de C sont toujours adjacents).

Notons qu'un graphe complet et antisymétrique s'appelle un « tournoi », car il symbolise le résultat d'un tournoi où chaque joueur est opposé une fois à chacun des autres joueurs.

-Il existe d'autre type de graphes particuliers on parle de graphes sans circuit et *graphe planaire et les graphes dual……etc*

II.6. Noyau d'un graphe

Stabilité

On dit qu'un sous ensemble de l'ensemble des sommets est stable s'il ne contient pas de paire de sommets adjacents.

Absorbance

Un sous-ensemble A de sommets est dit absorbant si tout sommet b n'appartenant pas à A possède au moins un successeur dans A.

Un sous-ensemble A de sommets est appelé noyau du graphe s'il est à la fois stable et absorbant.

En bref, le noyau d'un graphe…

- n'admet pas de boucle
- contient tout sommet x pour lequel $\Gamma(x) = \emptyset$
- Est l'ensemble stable avec le plus d'éléments.

II.7. Arbre et arborescence

-Définition d'un arbre

Un arbre est un graphe connexe sans cycle. Il a donc $(n-1)$ arcs. On peut donc dire qu'un arbre est un graphe qui connecte tous les nœuds entre eux avec un minimum d'arcs.

Un graphe sans cycle qui n'est pas connexe est appelé une forêt (chaque composante connexe est un arbre).

Soit un graphe non orienté G = (S, A). Un arbre T = (S, B) ou B ∈ A s'appelle un arbre recouvrant. L'existence d'un arbre recouvrant implique que le graphe considéré est connexe. Le plus souvent, on choisit un sommet s, on explore le graphe à partir de ce sommet et on construit une arborescence dont la racine est s. On parle néanmoins < Traditionnellement > d'arbre recouvrant et non d'arborescence recouvrante.

Arborescence : est un graphe orienté admettant un sommet, appelé la racine, tel que pour tout sommet il existe un unique chemin de la racine vers ce sommet. Clairement, tout sommet autre que la racine, admet un unique prédécesseur appeler son père.

III. Problèmes classiques de la théorie des graphes

III.1. Classification des problèmes mathématiques

Les problèmes sont répartis en deux classes selon leurs difficultés. Un problème est de:

Classe P

La classe de complexité P contient l'ensemble des problèmes pour lesquels il existe un algorithme déterministe qui s'exécute au pire des cas en temps polynomial pour les résoudre ;

Classe NP

Un problème de décision est un problème pour lequel la réponse est soit <<oui>>, soit <<non>>. La classe de complexité NP contient l'ensemble des problèmes de décision qui permet de le résoudre en temps polynomial ; ou encore pour lesquels il existe un algorithme polynomial qui permet de vérifier une solution (certificat) au problème.

Il est évident que P⊆NP ; et la grande question ouvertes en informatique est de savoir si P=NP ou si P≠NP.

Après que nous avons introduit les principaux concepts de la théorie des graphes, On passe maintenant au traitement des problèmes les plus courants de la théorie des graphes. Qui sont des problèmes très rencontrés dans la pratique et ils présentent une solution incontournable pour nombreux problèmes, parmi eux on trouve :

III.2. Exploration de graphes, Composantes connexe et Bipartisme

III.2.1. Construction des listes de successeurs

La liste des successeurs s d'un sommet x s'obtient simplement en parcourant la ligne x, et celle des prédécesseurs en parcourant la colonne x en $O(n)$. Cependant, si le graphe est peu dense, le parcours de la matrice d'adjacence est très encombrant. Le recours à la liste d'adjacence peut s'avérer ainsi avantageux, car il ne nécessite que O(d(x)), et dans un graphe peu dense, d(x) peut être beaucoup plus petit que n.

Cependant, l'utilisation de la liste pour l'énumération des prédécesseurs n'est guerre efficace. En effet, dans ce cas, il faut parcourait tout le graphe !

III.2.2. Décomposition d'un graphe en niveaux

La décomposition d'un graphe consiste à associer un niveau à chaque sommet de telle sorte à ce qu'il soit de niveau inférieur à tous ses successeurs. Elle permet d'optimiser certain algorithmes en leurs faisant traiter tout sommet avant ses successeurs.

Une façon très simple pour obtenir une telle décomposition (appelée aussi tri topologique) est d'associer un niveau à chaque sommet n'ayant pas de prédécesseur, supprimer ce sommet et augmenter la valeur du niveau tant qu'il reste des sommets sans niveau. Notons que cet algorithme ne fonctionne que sur des graphes sans circuits.

III.2.3. Exploration de graphes

L'exploration d'un graphe G à partir d'un sommet s est une opération fondamentale utilisée dans de nombreux algorithmes de graphes. Partant d'un sommet x, explorer (on dit aussi parcourir ou visiter) un graphe G, c'est déterminer l'ensemble des descendants de x, c'est-à-dire l'ensemble des sommets situés sur des chemins d'origine x.

Le principe de l'algorithme d'exploration est le suivant : au début, on marque le seul sommet x. en suite, on marque tout sommet y tel qu'il existe un arc (x, y) avec x marqué, jusqu'à ce qu'il ne reste aucun sommet non marqué.

La recherche en profondeur consomme généralement beaucoup moins d'espace que la recherche en largeur.

Par contre si le graphe est très grand (ou infini) ou si l'on veut plutôt chercher la solution la plus proche du sommet initial(en nombre d'arc traversés), la recherche en largeur est alors nécessaire.

III.2.4. Composantes connexes

Une exploration en largeur ou en profondeur à partir d'un sommet x de départ fournit les descendants de x, c'est-à-dire les sommets reliés à x par une chaine. Ces sommets forment donc la composante connexe de x.

Pour obtenir toutes les composantes, on initialise seulement une fois les marques au début, puis on lance l'exploration à partir d'un sommet quelconque x, ce qui donne la première composante connexe. Tant qu'il reste des sommets non marqués, on relance une exploration à partir de l'un d'eux, ce qui donne les composantes connexes suivantes.

Cet algorithme a la même complexité $O(M)$ qu'une seule exploration. Ce résultat est intuitif, car le parcours du graphe est réparti sur les explorations successives.

III.2.5. Composantes fortement connexes

Partant d'un sommet x, on peut trouver la liste des sommets avec lesquels il forme une composante connexe. Un sommet y appartient à la composante contenant x s'il existe un chemin de x vers y, et un autre de y vers x. On obtient simplement les sommets accessibles par un chemin d'origine x par une exploration en largeur ou en profondeur, coutant $O(m)$. Trouver les ancêtres de x revient à faire une exploration en $O(m)$ dans le graphe inverse. A la fin, les sommets atteints par les deux explorations forment la composante fortement connexe à laquelle appartient x. Le calcul de la composante fortement connexe de x est donc en $O(m)$ [6].

Pour trouver toutes les composantes, il faut itérer l'algorithme précédent pour chaque sommet n'appartenant pas déjà à une composante fortement connexe. On obtient en fin une complexité totale de $O(N.M)$.

III.2.6. Test de bipartisme

On dit qu'un graphe est biparti s'il existe une partition de l'ensemble des sommets en deux sous ensembles, telle que toutes arête joint un sommet du premier ensemble à un sommet de second.

Les graphes bipartis présentent deux propriétés importantes. Les algorithmes de tests de bipartisme sont généralement basés sur ces deux propriétés. Un graphe est donc biparti si et seulement s'il est 2-colorable ou qu'il n'a aucun cycle impair.

L'algorithme de test se base sur la détection d'un cycle impair. On utilise la procédure de marquage suivante : un sommet est mis à 0 s'il n'est pas encore visité, à 1 s'il est atteint par une chaine impaire et à 2 s'il est atteint par une chaine paire. Les successeurs d'un sommet marqué 1 sont marqués 2, et vise versa.

Au début, on initialise un sommet x à 2, et à partir de celui-ci on explore le graphe en appliquant la procédure de marquage. Un cycle impair est détecté lorsqu'un sommet y a un successeur z déjà marqué et de la même marque. En effet, une chaine de l'exploration joint x à y et une autre x à z. Comme y et z ont tout les deux soit un nombre pair soit un nombre impair d'arêtes, la chaine reliant y à z est paire. Avec (y, z), on forme un cycle impair.

III.3. Le problème des chemins optimaux

Le problème de la recherche du plus court chemin dans un graphe se rencontre dans de nombreuses applications. On peut citer entre autres :

– les problèmes de tournées,

– certains problèmes d'investissement et de gestion de stocks,

– les problèmes de programmation dynamique à états discrets et temps discret,

– les problèmes d'optimisation de réseaux (routiers, télécommunications),

– certaines méthodes de traitement numérique du signal, de codage et de décodage de l'information,

– les problèmes de labyrinthe et de récréations mathématiques.

III.3.1. Grands types de problèmes

On distingue trois types de problèmes :

Problème A

Etant donnés deux sommets i et j sera de trouver un chemin μ (i, j) de i et j dont la longueur totale $l(\mu) = \sum a \in \mu (i, j)\ l(a)$ soit minimum.

Problème B

Etant donné un sommet de départ s, trouver un plus court chemin de s vers tout autre sommet.

Problème C

Trouver un plus court chemin entre tout couple de sommets, c'est-à-dire calculer une matrice **N ×**N appelé **Distancier**.

Ces problèmes sont liés. En effet, un algorithme pour A peut être appliqué pour résoudre B ou C, ainsi que pour B et C. A part certains algorithmes conçus pour un type donné de problèmes.

III.3.2. Les deux familles d'algorithmes

Certain algorithme traitent définitivement un sommet à chaque itération : ils sélectionnent un sommet x et la valeur définitive du plus court chemin.ces algorithmes sont dits à fixation d'étiquettes. D'autres algorithmes peuvent affiner jusqu'à la dernière itération l'étiquette de chaque sommet. On les appelle algorithmes à correction d'étiquettes.

III.3.3. Le principe d'optimalité de Bellman

Ce principe n'énonce simplement qu'un plus court chemin est formé de plus courts chemins.

Si C est un plus court chemin de s à t et si u appartient à ce plus court chemin, alors les sous-chemins de s à u et de u à t sont également des plus courts chemins.

III.3.4. Les principaux algorithmes

Problème A

Le problème A est généralement résolu par un algorithme à fixation d'étiquettes pour le problème B ; qu'on stoppe dès que le sommet de destination choisi est traité définitivement. Il existe cependant un algorithme conçu spécifiquement pour le problème A, celui de Sedgwick et Vitter. Cet algorithme en $O(M.logN)$ est valable pour les graphes dit euclidiens.

Problème B

Soit W une fonction associé à tout arc qui représente le poid d'un arc ;

Cas W=constante :

Le problème revient à trouver des plus courts chemins en nombre d'arcs. Il peut se résoudre par une exploration du graphe en largeur, implémentable en $O(M)$.

Cas W≥0 :

Il est résoluble en $O(N^2)$ par un algorithme à fixation d'étiquettes dû de Dijkstra. Avec une structure de tas, on obtient une variante en $O(M.logN)$, intéressante si G est peu dense.

Cas W quelconque :

L'algorithme à correction d'étiquettes de Bellman est une méthode très connue, résolvant ce cas général en $O(N.M)$. Il existe une implémentation connue sous le nom de FIFO et basée sur une file de sommets.

Cas W quelconque, G sans circuit :

L'algorithme de Bellman peut alors se simplifier grâce à une décomposition de G en niveaux. Sa complexité chute alors à $O(M)$ et il devient aussi valable pour calculer des plus longs chemins.

Problème C

Il existe un algorithme à correction d'étiquettes très simple dû à FLOYD, calculant en $O(N^2)$ un distancier. Nécessitant une représentation matricielle du graphe, il consomme cependant trop de temps de calcul et de mémoire sur des graphes de simple densité.

III.3.5. Algorithmes à fixations d'étiquettes

- **Algorithme de Sedgwick et Vitter :**

Cet algorithme a été conçu pour des graphes euclidiens, c'est-à-dire des graphes non orientés dont les sommets sont des points d'un espace, et les arêtes des segments entre points valués par la longueur euclidienne du segment. Comme il a été déjà dit, cet algorithme est conçu pour le problème A, d'une complexité de $O(M.logN)$.

Pour chaque sommet x, l'algorithme maintient une évaluation par défaut du cout d'un plus court chemin de s à t passants par x. La structure générale est similaire à celle de Dijkstra, sauf qu'on traite définitivement à chaque itération le sommet d'évaluation par défaut [7].

- **Algorithme de Dijkstra :**

Cet algorithme détermine **les plus courts chemins d'un point s à tous les autres points d'un réseau** R = (X; U; d). Il suppose que **les longueurs sur les arcs sont positives ou nulles.** L'idée de cet algorithme est de partager les nœuds en deux groupes: ceux dont on connaît la distance la plus courte au point s (ensemble S) et ceux dont on ne connaît pas cette distance (ensemble S'). On part avec tous les nœuds dans S'. Tous les nœuds ont une distance infinie ($p(x) = +\infty$ avec le point s, excepté le point s lui-même qui a une distance nulle ($p(s) = 0$). A chaque itération, on choisit le nœud x qui a la plus petite distance au point s. Ce nœud est déplacé dans S. Ensuite, pour chaque successeur y de x, on regarde si

la distance la plus courte connue entre s et y ne peut pas être améliorée en passant par x. Si c'est le cas, p(y) est modifié. Ensuite, on recommence avec un autre nœud.

III.3.6. Algorithmes à correction d'étiquettes

- **Algorithme de Bellman :**

Cet algorithme a été conçu dans les années 50 par Bellman et Moore. Contrairement à l'algorithme de Dijkstra, celui-ci est prévu pour des valuations quelconques en $O(\mathcal{M}.\mathcal{N})$. Il peut être adapté pour détecter des circuits négatifs. Il utilise une méthode d'optimisation récursive, décrite par la relation de récurrence suivante :

$$\begin{cases} V_0(S) = 0, \\ V_0(Y) = +\infty, \quad y \neq s; \\ V_K(Y) = min\{V_{K-1}(X) + W(X,Y), K > 0\} \end{cases}$$

$V_k(x)$ désigne la valeur des plus courts chemins d'au plus k arcs entre le sommet s et le sommet x. Les deux premières relations servent à stopper la récursion. Le sommet s peut être considéré comme un chemin de 0 arc et de coût nul. La troisième relation signifie qu'un chemin optimal de k arcs de s à y s'obtient à partir des chemins optimaux de k-1 arcs de s vers tous les prédécesseurs x de y (principe d'optimalité de Bellman).

Pour détecter un circuit négatif, on fait une N-énième itération : si alors $V_n \neq V_{n-1}$, il y a un circuit. En fait l'algorithme ne peut détecter un circuit que dans la descendance de s. Pour trouver un circuit négatif quelconque, tout sommet doit être accessible au départ de s [20].

- **Algorithme FIFO :**

Cet algorithme examine les sommets dans l'ordre FIFO grâce à une file de sommets. Au début, seul s est dans la fille. Une itération principale traite tous les sommets présents dans la file au début de l'itération, on parcours leurs successeurs et place les successeurs d'étiquette améliorée à la fin de la file.

L'algorithme se termine quand la file est vide.

Pour détecter un circuit négatif, il suffit de compter le nombre d'itérations : il y a un circuit négatif si la file n'est pas vide à la fin des N-énièmes itérations. Sa complexité est même que celle de Bellman. En fait-il en est un dérivé [6].

- **Algorithme de FLOYD :**

L'algorithme de FLOYD calcule un distancier N×N donnant les valeurs des plus courts chemins entre tout couple de sommets (problème C). Au début, on initialise la matrice par les valeurs des arcs, les boucles étant sans effets et elles sont initialisés à 0, enfin, les arcs qui n'appartiennent pas au graphe sont mis à l'infini.

Soit V^0 la matrice de départ, on calcule V^1 par la formule suivante :

$$V_{ij}^1 = \min (\ V_{ij}^0 , V_{i1}^0, V_{1j}^0)$$

La matrice V^1 obtenue contient les couts des plus courts chemins ayant le seul sommet 1 comme sommet intermédiaire. De la même façon, on peut construire une suite de matrices :

$$V_{ij}^k = \min (V_{ij}^{k-1}, V_{i1}^{k-1}, V_{1j}^{k-1})$$

V_K donne les couts des plus chemins dont tous les sommets intermédiaires sont dans l'ensemble {1,2, . . ., k}. La matrice V^N fournit donc le distancier désiré qui va coûter $O(N^3)$ [7].

III.4. Le problème de Coloriage

III.4.1. Introduction

En théorie des graphes, **colorer un graphe** signifie attribuer une couleur à chacun de ses sommets de manière à ce que deux sommets reliés par une arête soient de couleur différente. Est souvent recherchée l'utilisation d'un nombre minimal de couleurs, dit **nombre chromatique**. Ce problème peut être complexifié en ne cherchant plus une mais plusieurs couleurs par sommets et en associant des coûts à chacune des couleurs. Le champ d'applications de la coloration de graphe couvre notamment le problème de l'allocation de fréquences dans les télécommunications ou la conception de puces électroniques.

III.4.2. Définition

La notion de coloration n'est définie que pour les graphes sans boucle, et la multiplicité des arêtes ne joue aucun rôle. Donc, soit *G* un graphe simple (sans boucle ni arête multiple), un *stable* de *G* est un sous-ensemble de sommets deux-à-deux non-adjacents, et une *coloration* de *G* est une partition de son ensemble de sommets en stables [8].

La définition originale de la coloration est la suivante:

Une coloration de G est une fonction associant à tout sommet de G une couleur, généralement un élément de l'ensemble d'indices des couleurs {1,2,..., n}, telle que deux sommets adjacents n'ont pas la même couleur (où n est le nombre de sommets du graphe). On lui préfère généralement la définition que nous avons donnée en premier, car, pour la plupart des questions liées au concept de coloration, on ne souhaite pas différencier les colorations qui ne sont distinctes qu'à permutation des indices de couleurs près (ce qui est le cas pour le problème central lié à la coloration, celui de déterminer le nombre minimum de couleur dans une coloration de G, c'est-à-dire son **nombre chromatique**). Par-exemple, si G est constitué de deux sommets u et v et d'une arête les reliant, alors les deux fonctions f et g avec $f(u)=1$, $f(v)=2$, et $g(u)=2$, $g(v)=1$ sont des colorations différentes pour la deuxième définition mais équivalentes pour la première.

Dans ses différents ouvrages (en français ou en anglais), Berge (Mathématicien français. Fondateur de la théorie des graphes) a utilisé les deux notations $\gamma(G)$ et $\chi(G)$ pour désigner le nombre chromatique de G. De nos jours, la plupart des ouvrages adoptent le symbole $\chi(G)$ (tandis que $\gamma(G)$ concerne plutôt un invariant lié au concept de cycle).

Enfin, dans certains contextes, on parle aussi de coloration pour désigner une fonction associant une couleur aux sommets d'un graphe mais satisfaisant d'autres propriétés (dans le contexte de l'optimisation de la génération de code sur une machine comportant un grand nombre de registres) [9].

III.4.3. Algorithme de Welsh et Powell

Voici un exemple qui, en plus, fournit une preuve constructive du théorème de Vizing, qui établit la formule : $X(G) \leq \Delta(G) + 1$, où $\Delta(G)$ représente le degré maximum d'un sommet de G (le degré d'un sommet étant le nombre des arêtes qui lui sont incidentes). Pour s'en convaincre, appliquons l'algorithme suivant (Welsh et Powell) :

1. Repérer le degré de chaque sommet.
2. Ranger les sommets par ordre de degrés décroissants (dans certains cas plusieurs possibilités).
3. Attribuer au premier sommet (A) de la liste une couleur.
4. Suivre la liste en attribuant la même couleur au premier sommet (B) qui ne soit pas adjacent à (A).

5. Suivre (si possible) la liste jusqu'au prochain sommet (C) qui ne soit adjacent ni à A ni à B.

6. Continuer jusqu'à ce que la liste soit finie.

7. Prendre une deuxième couleur pour le premier sommet (D) non encore colorié de la liste.

8. Répéter les opérations 4 à 6.

9. Continuer jusqu'à avoir coloré tous les sommets.

Remarquons que cette méthode peut aboutir à la pire des colorations possibles, par-exemple si le graphe G a la structure de couronne à n sommets, son nombre chromatique est 2 tandis que Welsh-Powell donne dans certains cas (selon l'ordre dans lequel sont rangés les sommets) une coloration utilisant n/2 couleurs. L'heuristique DESATUR permet d'éviter ce problème et donne une coloration moins mauvaise dans le pire cas [20].

III.4.4. Algorithme DESATUR

On considère un graphe *G= (V, E)* simple connexe et non-orienté. Pour chaque sommet *v* de *V*, on calcule le degré de saturation *DSAT(v)* de la manière suivante:

Si aucun voisin de v n'est coloré alors :

DSAT(v)=degré(v)

Sinon

DSAT(v)= le nombre de couleurs différentes utilisées dans le premier voisinage de v

L'algorithme DSATUR est un algorithme de coloration séquentiel, au sens où il colorie un seul sommet à la fois et tel que:

Au départ le graphe n'est pas coloré

On colorie un sommet non déjà coloré

On stoppe DSATUR quand tous les sommets de *G* sont colorés.

Dans un premier temps on voit bien d'une part que la preuve de terminaison est triviale et d'autre part que l'algorithme est séquentiel. Dans le détail l'algorithme est le suivant:

1. Ordonner les sommets par ordre décroissant de degré.
2. Colorer un des sommets de degré maximum avec la couleur 1.

3. Choisir un sommet non coloré avec DSAT maximum. Si un sommet voisin a la même couleur (cad conflit), choisir un sommet parmi ceux de degré maximum.

4. Colorer ce sommet par la plus petite couleur possible.

5. Si tous les sommets sont colorés alors stop .Sinon aller en 3 [20].

III.5. Problèmes de flots et de couplages

Nous allons présenter ici deux concepts importants en optimisation : les flots et les couplages. Etant donné un réseau aux arcs valués par des capacités, le problème du flot maximal consiste à faire véhiculer le plus grand flot possible entre deux nœuds *s* et *t*, sans excéder les capacités des arcs.

Si on définit une deuxième valuation sur les arcs, représentant des couts par unité de flot traversant l'arc, on peut chercher un flot de cout minimal parmi les flots maximaux, ou encore acheminer un flot de débit fixé à un cout minimal.

Enfin, pour un graphe simple, on appelle couplage un ensemble d'arêtes telles que deux quelconques d'entre elles n'aient aucun sommet commun [11].

III.5.1. Problème du flot maximal

Définitions

-Un réseau de transport est un graphe sans boucle, où chaque arc est associé à un nombre c(u), appelé "capacité de l'arc u". En outre, un tel réseau vérifie les hypothèses suivantes.

_ Il existe un seul nœud s qui n'a pas de prédécesseurs, tous les autres en ont au moins un. Ce nœud est appelé l'entrée du réseau, ou la source.

_ Il existe également un seul nœud p qui n'a pas de successeurs, tous les autres en ont au moins un. Ce nœud est appelé la sortie du réseau, ou le puits.

-Un flot f dans un réseau de transport est une fonction qui associe à chaque arc u une quantité f(u) qui représente la quantité de flot qui passe par cet arc, en provenance de la source et en destination du puits.

Un flot doit respecter la règle suivante: la somme des quantités de flot sur les arcs entrants dans un nœud (autre que s et p) doit être égale à la somme des quantités de flot sur les arcs sortants de ce même nœud. En d'autres termes, la quantité totale de flot qui entre dans un nœud est égale à la quantité totale de flot qui en sort [12].

Le problème de flot maximum :

Connaissant les capacités des arcs d'un réseau de transport, le problème du flot maximum consiste à trouver quelle est la quantité maximum de flot qui peut circuler de la source au puits. L'algorithme le plus connu pour résoudre ce problème est celui de Ford et Fulkerson. Il existe deux approches pour cette méthode. La première est basée sur la recherche d'une chaîne dans le réseau alors que la seconde construit un graphe "d'écart" dans lequel on recherche un chemin.

Cas particuliers

-Cas ou G n'est pas orienté

On peut remplacer toute arête par un couple d'arcs de capacité identique.

-Cas où il y a plusieurs sources ou puits

Éventuellement avec des disponibilités et demandes limités. On crée alors une super source qu'on relie à toute source par un arc de capacité égale à la disponibilité de cette source, et on relie tout puits à un super puits par un arc de capacité égale à la demande du puits.

-Cas de capacité sur les nœuds

On remplace tout nœud x par deux nœuds x' et x'' reliés par l'arc (x', x'') de même capacité que x.

Coupe et graphe d'écart

Une coupe S de G est un sous ensemble de nœuds qui inclut la source s, mais pas le puits t. Si l'on note par T l'ensemble (X - S), les arcs entrant sont ceux orientés de S vers T, les arcs sortant sont ceux orientés de T vers S. On appelle capacité d'une coupe la somme des capacités des arcs sortant.

Pour un flot ϕ, le graphe d'écart $G^e(\phi)=(X, A(\phi))$ décrit les possibilités d'augmentation de flot. Il aux mêmes sommets que G, et ses arcs sont défini comme suit :

-Tout arc non saturé de A est reporté dans $A(\phi)$.

-Pour tout arc (i, j) de flot non nul dans A, on crée un arc (i, j) dans $A(\phi)$.

Ainsi, on peut augmenter le flot dans G s'il existe un chemin de s à t dans le graphe d'écart $G^e(\phi)$. A ce chemin correspond une chaine améliorante u de s à t dans $G^e(\phi)$. On appelle

capacité résiduelle d'un arc (i, j) de u la quantité C_{ij} - ϕ_{ij} pour un arc avant, et ϕ_{ij} pour un arc arrière.

Propriétés

-Le débit de tout flot est inférieur ou égal à la capacité de toute coupe.

-Un graphe G contient une chaine améliorante si et seulement si le graphe d'écart $G^e(\phi)$ contient un chemin de s à t.

Lemme des chaines améliorantes :

S'il existe pour le flot ϕ une chaine améliorante reliant s à t, alors il existe un flot ϕ' supérieur à ϕ.

-Un flot est maximal si et seulement s'il n'existe pas de chaine améliorante.

Algorithme de Ford-Fulkerson :

On part d'un flot compatible. Le plus évident est le flot nul, i.e. pour tout arc u, f(u) = 0. Ensuite, on cherche une chaîne reliant la source au puits telle que son flot peut être augmenté.

Si on n'en trouve pas, le problème est résolu. Sinon, on augmente le flot sur cette chaîne. Ensuite, on recommence à chercher une chaîne augmentante et ainsi de suite. Une chaîne augmentante, i.e. une chaîne pour laquelle le flot peut être augmenté est une chaîne pour laquelle les arcs dans le sens direct n'ont pas atteint leur limite maximum et les arcs en sens indirect ont un flot non nul qui les traverse. L'augmentation de flot maximum pour une chaîne est le minimum des écarts entre le flot courant et le flot maximal pour les arcs directs ou le flot courant pour les arcs indirects. Autrement dit, une chaîne C

_ Pour tout arc u direct, f(u) < c(u),

_ Pour tout arc u indirect, f(u) > 0.

Le flot sur cette chaîne C peut être augmenté de:

Min ({c(u) - f(u) | u, C et u sens direct} {f(u) | u, C et u sens indirect})

Le deuxième algorithme et le suivant :

Comme pour la méthode précédente, on part d'un flot compatible. Ensuite, on construit un graphe d'écart à partir de ce flot. Ce graphe d'écart représente les modifications de flot possibles sur chaque arc. Sur ce graphe, les nœuds ont exactement la même signification que dans le réseau de transport. Par contre, un arc indiquera de combien il est possible

d'augmenter le flot entre deux nœuds. Ainsi, pour un arc u = (x;y), on créera dans le graphe d'écart:

- Un arc de x à y de capacité c ' ((x;y)) = c(u) - f(u) si c(u) ≠ f(u),
- Un arc de y à x de capacité c' ((y;x)) = f(u) si f(u) ≠ 0.

Ensuite, dans ce graphe d'écart, on cherchera un chemin de la source au puits. Si on n'en trouve pas, le problème est résolu. Sinon, on augmente le flot sur ce chemin, qui correspond en fait à une chaîne si l'on se ramène à la première méthode. Le flot sera augmenté de la plus petite capacité des arcs du chemin. Autrement dit, le chemin C sera augmenté de: min {c'(u) | u ∈C}.

Algorithme des distances estimées au puits :

Pour un réseau de transport G=(X, A, C, s, t) et un flot ϕ, on appelle distance estimée au puits une fonction Dist de X dans N vérifiant :

-Dist(t)=0

-Dist(i) <dist(j) +1, pour tout arc du graphe d'écart.

Un tel arc (i, j) de $G^e(\phi)$ est dit admissible. Un chemin s à t dans $G^e(\phi)$ est dit admissible si ses arcs sont admissibles.

Principe

Cet algorithme dû à Ahoja et Orlin, peut être vu comme une variante de l'algorithme de Ford-Fulkerson avec la recherche de plus courtes chaines améliorantes en nombre d'arcs (augmentantes).

Alors qu'une recherche en largeur rendait l'algorithme de Ford-Fulkerson en $O(N.M^2)$, l'algorithme d'Ahoja et Orlin atteint une complexité de $0(N^2.M)$ avec un système astucieux d'estimation de la distance des nœuds au puits dans le graphe d'écart.

Il commence par calculer les distances exactes au puits en faisant une exploration en largeur à partir de t dans le graphe inverse G^{-1}. En suite, il réalise des augmentations de flux le long de chemins admissibles. Il construit un tel chemin arc par arc. Si le chemin partiel est parvenu en un nœud i, l'algorithme cherche un arc admissible du graphe d'écart. Si on trouve, on pose $i = j$ et on recommence.

Si l'on parvient en t, on a trouvé une chaine améliorante. On procède à l'augmentation de flots, et on repart de $i = s$.

Dans le cas où il n' y a aucun arc admissible en parvenant en un nœud i, l'algorithme tente de créer de nouveaux arcs admissibles en remplaçant sa distance par le minimum des distances incrémenté de 1 de tout successeur de i dans le graphe d'écart.

Quand la distance de s dépasse $\mathcal{N}$, On sait alors qu'il n'existe plus de chemins admissibles : l'algorithme se termine.

Problème de flot compatible

Dans certains cas, on est obligé de considérer une valeur minimale de flux qui doit être présente en chaque nœud.

Le problème est de trouver un flot compatible avec les capacités minimales pour chaque arc.

La solution est de transformer un tel réseau en un réseau classique (capacités minimales nulles) pour le ramener à un problème de flot maximal.

Notons ici, que la complexité de l'algorithme de flot n'est pas affectée par la transformation du réseau [6].

III.5.2. Problèmes de couplages

Définition

Le problème posé à l'aviation anglaise durant la bataille d'Angleterre était de pouvoir former un nombre maximum de couples de pilotes parmi un ensemble de pilotes venant des quatre coins du monde. On confiait à deux pilotes les commandes d'un avion à l'unique condition qu'ils partageaient une même langue. Un grand nombre de pilotes parlaient plusieurs langues.

Ce problème se formalise aisément en un problème de graphes, qui a pour nom le problème du couplage maximum.

Un *couplage C* d'un graphe simple $G = (X, E)$ est un sous-ensemble d'arêtes deux à deux sans extrémité commune. Le cardinal d'un couplage est le nombre d'arêtes composant le couplage.

Un sommet $x \in X$ est dit *saturé* par un couplage C si x est l'extrémité d'une des arêtes de C. Dans le cas contraire, le sommet est dit *insaturé*.

Un couplage est *parfait* si ses arêtes contiennent tous les sommets du graphe (en d'autres termes, un couplage sature tous les sommets de X).

Un couplage *maximal* de *G* est un couplage de cardinal maximal. Un couplage parfait est un couplage maximal. Un *problème d'affectation* désigne la recherche d'un couplage maximal dans un graphe biparti.

Soit *C* un couplage de *G*. Une chaîne est dite *alternée* relativement au couplage *C* si elle emprunte alternativement des arêtes de *C* et des arêtes de $E \backslash C$. Une arête unique est une chaîne alternée de longueur 1.

Une chaîne alternée *augmentante* ou *améliorante* (CAA) joint deux sommets insaturés par *C*.

Conversion en problème de flot

Pour un graphe biparti $G = (X, Y, E)$, on peut convertir les problèmes de couplage en problèmes de flot dans un réseau $R = (X, Y, U, C, s, t)$ avec :

– on oriente les arêtes de *G* en arcs de *X* vers *Y* et on leur attribue une capacité infinie;

– on ajoute une entrée *s* reliée à tout sommet de *X* par un arc de capacité 1 et une sortie *t* à laquelle tout sommet de *Y* est relié par un arc de capacité 1.

On peut notamment ramener le problème du couplage maximal en un problème de flot maximal dans un réseau de transport (résolu par exemple par l'algorithme de Busacker-Gowen).

On fait passer une unité de flot sur les arêtes du couplage et sur les arêtes correspondantes vers *s* et *t*.

Principaux généraux des algorithmes de couplage

Le problème de couplage maximal dans un graphe biparti ou non peut être résolu grâce aux deux propriétés suivantes dues à Berge :

Soit un graphe simple G=(X, U), un couplage C de G, et une CAA u pour C. Alors un transfert sur u donne un nouveau couplage C' avec |C'|=|C|+1.

Un couplage est maximal si et seulement s'il n'admet aucune CAA.

Dans un graphe biparti, nous avons de plus le théorème de Hall suivant(en notant N(S) l'ensemble des voisins de S dans G) :

Un graphe biparti (X, U, E) admet un couplage saturant X si et seulement si pour tout S de X on a ||N(S) || ≥ ||S ||.

III.6. Arbres et arborescences

-Problèmes de l'arbre de cout minimum /maximum

Imaginons que l'on associe une valeur, un poids, à chaque arc d'un graphe G. Le problème de l'arbre de coût maximum consiste à trouver un arbre, graphe partiel de G, dont la somme des poids des arcs est maximum. En pratique, les problèmes se ramène plutôt à trouver l'arbre de coût minimum, ce qui ne change pas fondamentalement le principe des algorithmes proposés ici.

Par exemple, minimiser le coût d'installation de lignes électriques entre des maisons peut être modélisé par la recherche d'un arbre de coût minimum. En effet, on veut connecter toutes les maisons entres elles sans avoir de lignes inutiles (d'où la recherche d'un arbre). Ensuite, on veut utiliser le moins de câble possible, aussi on associera à chaque possibilité de connexion la longueur de câble nécessaire et on cherchera à minimiser la longueur totale de câble utilisée.

Il existe deux algorithmes dans la littérature. L'efficacité de chacun d'eux dépend du choix de représentation du graphe et de la structure même du graphe.

Théorème d'optimalité des Arbres Recouvrant des Poids Minimaux :

Les algorithmes ARPM exploitent tous plus ou moins la propriété suivante : soit $F= ((X_1, T_1), (X_2, T_2), \ldots, (X_k, T_k))$ une forêt de G, et [u, v] la plus petite arête ayant une extrémité dans X_1. Alors, parmi tous les arbres recouvrant incluant cette forêt, le meilleur contient [u, v].

- **Algorithme de Kruskal (1956) :**

Le principe de l'algorithme de Kruskal pour trouver un arbre de poids minimum dans un graphe G est tout d'abord de trier les arcs par ordre croissant de leur poids. Ensuite, dans cet ordre, les arcs sont ajoutés un par un dans un graphe G' pour construire progressivement l'arbre. Un arc est ajouté seulement si son ajout dans G' n'introduit pas de cycle, autrement dit, si G' reste un arbre. Sinon, on passe à l'arc suivant dans l'ordre du tri.

- **Algorithme de Prim (1957) :**

Le principe de l'algorithme de Prim pour trouver un arbre de poids minimum dans un graphe G est de fusionner, deux par deux les nœuds de G pour obtenir finalement un arbre. Par fusionner, on entend remplacer deux nœuds par un seul. Tous les arcs adjacents à l'un

ou l'autre des anciens nœuds deviennent adjacents au nouveau nœud. Le choix des nœuds que l'on fusionne est fait en choisissant au hasard un nœud et en cherchant un arc adjacent (autre qu'une boucle) qui a le coût le plus faible. Ce qui fournit le deuxième nœud de la fusion [10].

-Problèmes de l'arborescence de poids minimal

On considère un graphe orienté valué G=(X, U, W) dans lequel on cherche une arborescence recouvrante de poids minimal et de racine s fixée. Nous précisons d'abord une définition d'une arborescence couvrante A qui sera exploitée par l'algorithme d'Edmonds.

-A est un arbre couvrant de G (si on néglige l'orientation).

-Tout sommet sauf la racine a un seul prédécesseur.

III.7. Tournée eulérienne et hamiltonienne

L'étude des problèmes– eulériens ou hamiltoniens – (recherche d'une chaîne ou d'un cycle passant exactement une fois par chaque arête – ou par chaque sommet –) remonte aux origines de la théorie des graphes.

L'intérêt porté aujourd'hui à ces problèmes s'explique par leurs nombreuses applications : tournées de distribution, tracé automatique sur ordinateur, problèmes d'ordonnancement d'atelier, etc [14].

IV. L'intérêt des graphes dans le monde réel

Les graphes (et par conséquent la théorie des graphes) sont utilisés dans de nombreux domaines. On peut donner quelques exemples :

-les réseaux de communication : réseaux de routes représentés par une carte routière, réseaux de chemin de fer, de téléphone, de relais de télévision, réseaux électriques, réseaux des informations dans une organisation, etc... ;

-La gestion de la production : graphes potentiels-étapes plus connu sous le nom de graphes PERT ["Programme Evaluation and Research Task" ou "Programme Evaluation Review Technique"] ;

-L'étude des circuits électriques : Kirchhof, qui a étudié les réseaux électriques, peut être considéré comme un des précurseurs de cette théorie ;

-La chimie, la sociologie et l'économie: la notion de clique est un exemple de l'implication de la théorie des graphes dans ces disciplines.

Les graphes sont énormément utilisés en informatique. Outre leur efficacité dans la modélisation programmatique de structures de données complexes, on les rencontre par exemple pour :

- **Les bases de données**

Un modèle relationnel de données est représentable par un graphe orienté regroupant des relations (sommets du graphe) et des dépendances (arcs du graphe).On parlent notamment de graphe sémantique normalisé pour désigner un schéma de données relationnel résultant du processus de normalisation.

- **Système d'exploitation**

-Parallélisme :

Les techniques d'optimisation d'algorithmes ou de détermination d'ordre d'exécution cohérente dans ce domaine prennent souvent en entrée des graphes de dépendance de flots d'instructions ou de données, ou les sommets sont respectivement des instructions (du code à exécuter) et des données (initiales ou calculées) et les arcs des relations de dépendance temporelle (telle instruction doit s'exécuter après telle autre; telle donnée doit être calculée avant telle autre).

-Le problème de l'interbloquage :

Est un problème récurant. Parfois, il existe des situations ou un processus demande une ressource pour laquelle il est propriétaire et au même temps il utilise une autre ressource pour laquelle il ne l'est pas.

Pour illustrer cette situation, supposant que l'on dispose de trois ressources.

Le processus P1 utilise la ressource R1 et demande la ressource R2 qui est utilisée par P2.

Le processus P2 détient R2 et demande R3 qui est détenue par P3.

Le processus P3 détient R3 et demande R1.

Le problème de l'interbloquage se ramène à la recherche de circuit dans le graphe des attentes [15].

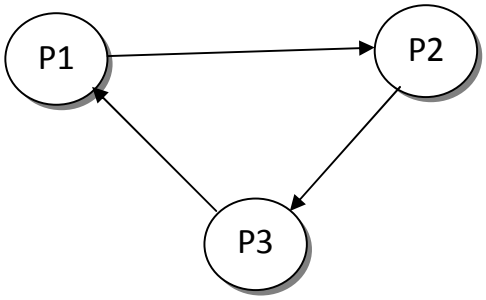

FIG.2- *Représentation de l'interbloquage par un graphe*

-Il existe plusieurs structures de répertoires, parmi elles on trouve :

Répertoires à graphes acycliques :

Dans certains cas les utilisateurs travaillent en équipe et désirent travailler sur les mêmes fichiers ou répertoires.

Ceux-ci doivent être partagés. En d'autres termes, ils doivent exister en deux endroits (où plus), sans autant qu'il ait deux copies du fichier ou répertoire partager.

Comme le partage n'est pas possible dans une structure à répertoire en arbre, la structure à graphe acyclique a été définie. Cette structure est une généralisation de la structure arborescente et sous forme d'un graphe sans cycle, ou un fichier ou répertoire partagé se retrouve en deux répertoires différents.

Répertoire de graphes généraux :

Lorsqu'une structure de répertoire arborescente est utilisée, il est possible que les utilisateurs ajoutent des liens à un répertoire pour se partager ce répertoire. Avec l'ajout de ces liens, la structure arborescente est perdue et il est possible d'obtenir un graphe contenant des cycles, ce qui est appelé répertoire de graphes généraux. Toutefois, cette structure rend difficile la gestion de certains problèmes, tels que la recherche des références ou liens vers un fichier(ou répertoire) et la suppression d'un fichier [16].

Graphe du Web

Aujourd'hui, le Web est composé de milliards de pages écrites par des personnes indépendantes. Ces pages sont interconnectées par des liens hypertextes que les internautes utilisent pour se déplacer d'une page à l'autre.

Chaque personne créant une page Web choisit librement les pages vers lesquelles pointer et peut changer d'avis au cours du temps, sans qu'il y ait un contrôle global. Lorsqu'on veut étudier le Web, on se retrouve donc face à un immense ensemble d'éléments interconnectés pour lesquels aucune autorité centrale n'existe.

Dans ce cas, obtenir à un moment donné une vue d'ensemble des pages disponibles et de leurs interconnexions par les liens hypertextes devient très difficile.

La structure du Web se modélise naturellement à l'aide d'un graphe orienté dont les sommets sont les pages Web et les arcs sont les liens hypertextes entre ces différentes pages. Un certain nombre de phénomènes peut être compris grâce à l'étude de cette structure. Citons en particulier l'optimisation des moteurs de recherche ou encore la détection des communautés dans un but commercial par exemple [17].

Graphe de l'internet

Un des réseaux les plus connus est le réseau Internet, plusieurs applications utilisent ce réseau pour transmettre de l'information.

Internet peut être vu comme un graphe à plusieurs niveaux : Interfaces (adresse IP), routeurs (machines physiques) et systèmes autonomes (institutions) [18].

Le graphe d'Internet est un graphe dont les sommets sont des ordinateurs ou des routeurs et les arêtes représentent des liaisons physiques entre les ordinateurs.

V. Conclusion

L'étude présentée dans ce chapitre, nous a permis de mieux comprendre les notions de bases de la théorie des graphes et leurs applications en Informatique en abordant les divers problèmes classiques.

Chapitre 2 :

Les graphes sous java

I. Introduction

Dans le but de bien réaliser notre application qui intègre tous les algorithmes déjà vus dans le chapitre précédent, mais aussi afin de bien les implémenter on avait besoin d'utiliser un langage à la fois simple à manipuler, qu'il soit un langage orienté objet, qu'il puisse effectuer toutes les taches d'un langage de haut niveau (bureautique, graphique, multimédia, base de donné, environnement de développement, etc.).

On a choisit l'environnement de développement NetBeans.

II. Pour quoi java ?

On a préfère ce langage de programmation afin de :

- Maitriser un nouveau langage de programmation et comprendre bien le concept orienté objet.
- Et les exceptions qui distingue ce langage par rapport les autres langages comme la facilité (leur syntaxe est une extension de langage C)…etc

III. Présentation générale

Apparu fin 1995 début 1996 et développé par Sun Microsystems Java s'est très rapidement taillé une place importante en particulier dans le domaine de l'internet et des applications client-serveur.

Destiné au départ à la programmation de centraux téléphoniques sous l'appellation de langage "oak", la société Sun a eu l'idée de le recentrer sur les applications de l'internet et des réseaux. C'est un langage en évolution permanente Java 2 est la version stabilisée de java fondée sur la version initiale 1.2.2 du JDK (Java Development Kit de Sun : http://java.sun.com).

Les objectifs de java sont d'être multi-plateformes et d'assurer la sécurité aussi bien pendant le développement que pendant l'utilisation d'un programme java. Il est en passe de détrôner le langage C++ dont il hérite partiellement la syntaxe mais non ses défauts (l'héritage multiple, les pointeurs, la surcharge des opérateurs). Comme C++ et Delphi, java est algorithmique et orienté objet à ce titre il peut effectuer comme ses compagnons, toutes les tâches d'un tel langage (bureautiques, graphiques, multimédias, bases de données, environnement de développement, etc...). Son point fort qui le démarque des autres est sa portabilité due (en théorie) à ses bibliothèques de classes indépendantes de la plate-forme, ce qui est le point essentiel de la programmation sur internet ou plusieurs machines dissemblables sont interconnectées. La réalisation multi-plateformes dépend en fait du système d'exploitation et de sa capacité à posséder des outils de compilation et d'interprétation de la machine virtuelle Java.

Actuellement ceci est totalement réalisé d'une manière correcte sur les plates-formes Windows et Solaris, un peu moins bien sur les autres semble-t-il.

En bref on dit que java est le premier langage du troisième millénium actuel.

III.1. Caractéristiques du langage java

Java possède un certain nombre de caractéristiques qui ont largement contribué à son énorme succès :

- ***Simplicité*** **:**

Nous avons voulu créer un système qui puisse être programmé simplement, sans nécessiter un apprentissage ésotérique, et qui tire parti de l'expérience standard actuelle. En conséquence, même si nous pensions que C++ ne convenait pas, Java a été conçue de façon relativement proche de ce langage dans le dessein de faciliter la compréhension du système. De nombreuses fonctions compliquées, mal comprises, rarement utilisées, de C++, qui nous semblaient par expérience apporter plus d'inconvénients que d'avantages, ont été supprimées de Java.

- ***Orienté objet*** **:**

Pour rester simples, disons que la conception orientée objet est une technique de programmation qui se concentre sur les données (les objets) et sur les interfaces avec ces objets. Pour faire une analogie avec la menuiserie, on pourrait dire qu'un menuisier "orienté objet" s'intéresse essentiellement à la chaise (l'objet) qu'il fabrique et ensuite aux outils

utilisés pour la fabriquer. Par opposition, le menuisier "non orienté objet" penserait d'abord à ses outils. Les facilités orientées objet de Java sont essentiellement celles de C++...

- ***Fiabilité*** **:**

Java a été conçu pour que les programmes qui l'utilisent soient fiables sous différents aspects. Sa conception encourage le programmeur à traquer préventivement les éventuels problèmes, à lancer des vérifications dynamiques en cours d'exécution et à éliminer les situations génératrices d'erreurs...

La seule et unique grosse différence entre C/C++ et Java réside dans le fait que ce dernier intègre un modèle de pointeur qui écarte les risques d'écrasement de la mémoire et d'endommagement des données.

- ***Java assure la gestion de la mémoire*** **:**

L'allocation de la mémoire pour un objet est automatique à sa création et Java récupère automatiquement la mémoire inutilisée grâce au garbage collector qui restitue les zones de mémoire laissées libres suite à la destruction des objets.

- ***Architecture neutre*** **:**

Le compilateur génère un format de fichier objet dont l'architecture est neutre — le code compilé est exécutable sur de nombreux processeurs, à partir du moment où le système d'exécution de Java est présent. Pour ce faire, le compilateur Java génère des instructions en bytecodes (ou pseudo-code) qui n'ont de lien avec aucune architecture d'ordinateur particulière. Au contraire, ces instructions ont été conçues pour être à la fois faciles à interpréter, quelle que soit la machine, et faciles à traduire à la volée en code machine natif.

- ***Portabilité*** **:**(*il est indépendant de toute plate-forme*)

Il n'y a pas de compilation spécifique pour chaque plate forme. Le code reste indépendant de la machine sur laquelle il s'exécute. Il est possible d'exécuter des programmes Java sur tous les environnements qui possèdent une Java Virtual Machine. Cette indépendance est assurée au niveau du code source grâce à Unicode et au niveau du byte code.

- ***Interprété*** **:**

La source est compilé en pseudo code ou byte code puis exécuté par un interpréteur Java : la Java Virtual Machine (JVM). Ce concept est à la base du slogan de Sun pour Java : WORA (Write Once, Run Anywhere : écrire une fois, exécuter partout). En effet, le byte code, s'il ne

contient pas de code spécifique à une plate-*forme particulière peut être exécuté et obtenir quasiment les mêmes résultats sur toutes les machines disposant d'une JVM.(on peut dit que java possède un semi compilateur et un semi interpréteur pour quoi on dit comme ça parce que la compilation extraire le programme en code machine et le cas de nous on obtient pas le code machine on obtient le code en octet aussi le semi interprétation parce que on n'interprète pas depuis le code sourc , on interprète depuis le code en octet).*

- ***Performances élevées* :**

En règle générale, les performances des bytecodes interprétés sont tout à fait suffisantes ; il existe toute fois des situations dans lesquelles des performances plus élevées sont nécessaires. Les bytecodes peuvent être traduits à la volée (en cours d'exécution) en code machine pour l'unité centrale destinée à accueillir l'application.

- ***Multithread* :**

Il permet à nous l'utilisation de threads qui sont des unités d'exécution isolées. La JVM, elle même, utilise plusieurs threads. Les avantages du multithread sont une meilleure interréactivité et un meilleur comportement en temps réel.

- ***Java, langage dynamique* :**

Sur plusieurs points, Java est un langage plus dynamique que C ou C++. Il a été conçu pour s'adapter à un environnement en évolution constante. Les bibliothèques peuvent ajouter librement de nouvelles méthodes et variables sans pour autant affecter leurs clients. La recherche des informations de type exécution dans Java est simple [19].

IV. Pour quoi NetBeans ?

On a distingué l'environnement de développement NetBeans parce que il est faciles à comprendre et à utiliser, très riche, et disponible gratuitement, pour compiler et exécuter les programmes Java, les environnements de développement intégrés peuvent nécessiter des ressources importantes et être lourds à utiliser pour de petits programmes. Comme solution intermédiaire, vous disposez des éditeurs de texte appelant le compilateur Java et exécutant les programmes Java. Lorsque vous aurez maîtrisé les techniques présentées.

V. Présentation général de NetBeans

V.1. Définition

*NetBeans IDE est un environnement de développement intégrer (*On peut y écrire, compiler et exécuter les programmes. Un IDE fournit aussi un utilitaire d'*Aide* qui décrit tous les éléments du langage et te permet de trouver et de corriger plus facilement les erreurs dans tes programmes.*)* Est à l'origine un EDI Java. NetBeans fut développé à l'origine par une équipe d'étudiants à Prague, racheté ensuite par Sun Microsystems. Quelque part en 2002, Sun a décidé de rendre NetBeans open-source. Mais NetBeans n'est pas uniquement un EDI Java. C'est également une plateforme, vous permettant d'écrire vos propres applications Swing. Sa conception est complètement modulaire : Tout est module, même la plateforme. Ce qui fait de NetBeans une boite à outils facilement améliorable ou modifiable. La licence de NetBeans permet de l'utiliser gratuitement à des fins commerciales ou non. Elle permet de développer tous types d'applications basées sur la plateforme NetBeans. Les modules que vous pourriez écrire peuvent être open source comme ils peuvent être closed-source, Ils peuvent être gratuits, comme ils peuvent être payants.

V.2. Caractéristiques

- L'environnement de développement peut être étendu par un ensemble de plugins que la communauté des contributeurs développe.
- NetBeans est un moteur de graphique servant de base logicielle pour le développement de RAD (Rich Application Desktop). Il offre un framework RCP (Rich Client Programming) pour développer tout type d'application graphique en utilisant l'environnement graphique Swing de java.
- **Interopérabilité**

  Il est possible d'importer des projets Eclipse vers NetBeans avec NetBeans IDE 4.1 ou supérieur, il suffit d'installer 'Eclipse Projet Importer'.

- **Modularité**

  -NetBeans possède un module de facilitation et de hiérarchie classloader.
  -Un module dépendance influence runtime classpath.
  -Un module peut tourner sur d'autre module.

- **Coopération de modules**

  -Un module possède une fenêtre, un autre une action.

  -Windows peut avoir des associés contextes.

  -Sélection fenêtre définit le contexte mondial.

  -Eléments de l'interface utilisateur mise à jour en fonction de son classloader.

  -Le menu, la barre d'outils des éléments obtenus fusionnés par le cadre de NetBeans.

  -Windows Layout.

- **Il intègre**

  -Ruby Support.

  -Swing GUI développement.

V.3. Avantages de NetBeans

NetBeans fournit un environnement riche pour le règlement de problèmes et l'optimisation des applications.

.Intégration de :

-Swing.

-La plut part des outils java.

-Window System.

-Docking cadre extensions autour de Swing.

-Module Système.

.Mise à jour automatique.

.Exécuter des morceaux de code bien spécifiques à un moment donné (en utilisant les points d'arrêt comme délimiteurs).

.Suspendre l'exécution lorsqu'une condition spécifiée est rencontrée (comme par exemple quand un itérateur atteint une certaine valeur).

.Suspendre l'exécution lors d'une exception, soit à la ligne de code qui a provoqué l'exception, ou dans l'exception elle-même.

.Pister la valeur d'une variable ou d'une expression.

.pister l'objet référencé par une variable (fixed watched).

.Fixer le code à la volée et continuer la session de débogage.

.Suspendre les threads individuellement ou collectivement.

.Revenir au début d'une méthode appelée précédemment (pop a call) dans le call stack actuelle.

.Exécuter de multiples sessions de débogage en même temps. Par exemple, on pourrait recourir à cette capacité pour déboguer une application client/serveur

VI. Les graphes sous java

Un graphe est implémenté classiquement soit par la matrice d'adjacence, soit par une liste d'adjacence. Le choix de la représentation d'un graphe sera guidé par sa densité (nombre d'arcs), mais aussi par les opérations qui sont appliquées. D'une façon générale, plus le graphe est dense plus la matrice d'adjacence conviendra et avec cette représentation, la complexité en espace mémoire est O (n^2).

Au contraire, pour des graphes creux, les listes d'adjacences seront adaptées : c'est-à-dire lorsque le nombre d'arcs est plus petit par rapport au nombre de sommet. Un graphe orienté de n sommets et p arcs nécessite n + p éléments de liste, alors qu'un graphe non orienté en nécessite *n+2p*.

VI.1. L'implémentation en java

Le JDK (Java Developpement Kit) actuel ne contient pas un package ou une classe qui implémente les graphes. Cependant, beaucoup d'ouvrages ont traité les graphes sous java chacun d'une manière différente de l'autre.

Dans le cadre de notre projet, nous avons crée une structure plus adaptée à nos besoins et cela en utilisant les deux représentations d'un graphe en mémoire mais tout en se basant sur la liste d'adjacence.

Nous supposons que chaque sommet est associé à un nombre entier positif ou nul, son numéro, qui est le moyen par lequel l'utilisateur l'identifie et le manipule. Deux sommets différents ont des numéros différents.

Pour mettre en évidence l'ensemble des opérations sur les graphes nécessaires à la programmation de l'application envisagée, On définit trois classes et interface.

VI.1.1. Interface Graphe

public int nombreSommets ( ) renvoie le nombre de sommets du graphe.

public int nombreArêtes ( ) renvoi le nombre d'arête.

public Sommet sommet (int numéro) renvoi le sommet ayant le numéro indiqué, ou null si un tel sommet n'existe pas.

public Sommet insererSommet (int numéro) crée, insère dans le graphe et renvoie comme résultat un sommet portant le numéro indiqué.

public void supprimerSommet (Sommet sommet) supprime le sommet indiqué et tous les arcs dont il est une extrémité.

public void supprimeArc (Sommet x, Sommet y) supprime l'arc indiquée.

public void insreArc (Sommet x, Sommet y) insère l'arc indiqué.

VI.1.2. Classe Sommet

Attributs

private byte marque : c'est le champ de marquage.

public LinkedList listSucc : c'est la liste des successeurs d'un sommet.

Méthodes

public Sommet (int numéro) construit un sommet initialisé avec son numéro.

public int numero ( ) renvoi le numéro du sommet en question.

public long marque ( ) renvoi la marque du sommet en question.

public void marquer (long marque) définit la marque du sommet en question.

VI.1.3. Classe Successeur

Attributs

private int num : c'est le numéro du successeur.

private double val : c'est le poids de l'arc.

Méthode

public int getNum ( ) : retourne le numéro du successeur.

public void setVal (double val) : permet de valuer un arc.

public double getVal ( ) : permet de retourner la valuation de l'arc en question.

VI.1.4. Classe Graph

Attributs

private LinkedList listSommet : c'est la liste d'adjacence.

Méthode

Cette classe implémente toutes les méthodes de l'interface Graphe. On peut ajouter à la classe Graph quelques méthodes auxiliaires utiles :

public void insererArete (int i, int j) : crée et insère dans le graphe les sommets portant les numéros i et j, s'ils n'existent pas, et l'arête qui a ces sommets pour l'extrémité.

public void getMat ( ) : renvoi la matrice d'adjacence.

VII. Conclusion

Durant ce chapitre, on a justifié et argumenter notre choix du langage java et IDE NetBeans et nous avons défini et présenté notre propre structure des graphes sous java, vu que ce langage ne contient pas une structure de données spécifique pour les graphes.

Chapitre 3 :

Description de l'application

I. Introduction

Nous allons aborder maintenant à la troisième partie de notre travail, on donnant tous les détails sur l'application, ses ingrédients et ses méthodes de fonctionnements. **Graphe projet** qui éclaire un point précis du concept abordé, inclure quelques boutons qui sont chacune représente une solution d'un certain problème ou une manipulation de base sur un graphe, nous allons décrire en détail ça dans ce chapitre.

II. Structure de l'application

La fenêtre principale se compose de quatre parties principales comme la suite, une partie contient une liste des boutons qui permets de commencer (nouveau, ouvrir…), et la deuxième partie pour les applications sur les graphes, et deux autres parties l'une représente un Editeur et l'autre Résultat d'exécution.

La première partie permet de commencer avec nouveau projet, ouvrir et enregistrer des fichiers graphe. Et la deuxième partie se compose aussi de quatre d'autres parties la première permet des manipulations de base relatives, l'ajout et la suppression de composants du graphe. La seconde propose les différents modes d'affichage d'un graphe que permet notre application. La troisième propose quelque types de graphes particuliers pouvant être extraire du graphe saisit et la quatrième présente l'ensemble des solutions algorithmiques traitées par l'application.

Enfin il reste la troisième et la quatrième partie qui sont occupées par un Editeur dans lequel seront visionnée la matrice d'adjacence du graphe de façon et par Résultat d'exécution qui permet de voir toutes les résultats concernant le graphe. Dans ce qui suit, nous allons présenter en détail les différentes parties évoquées dans cette section.

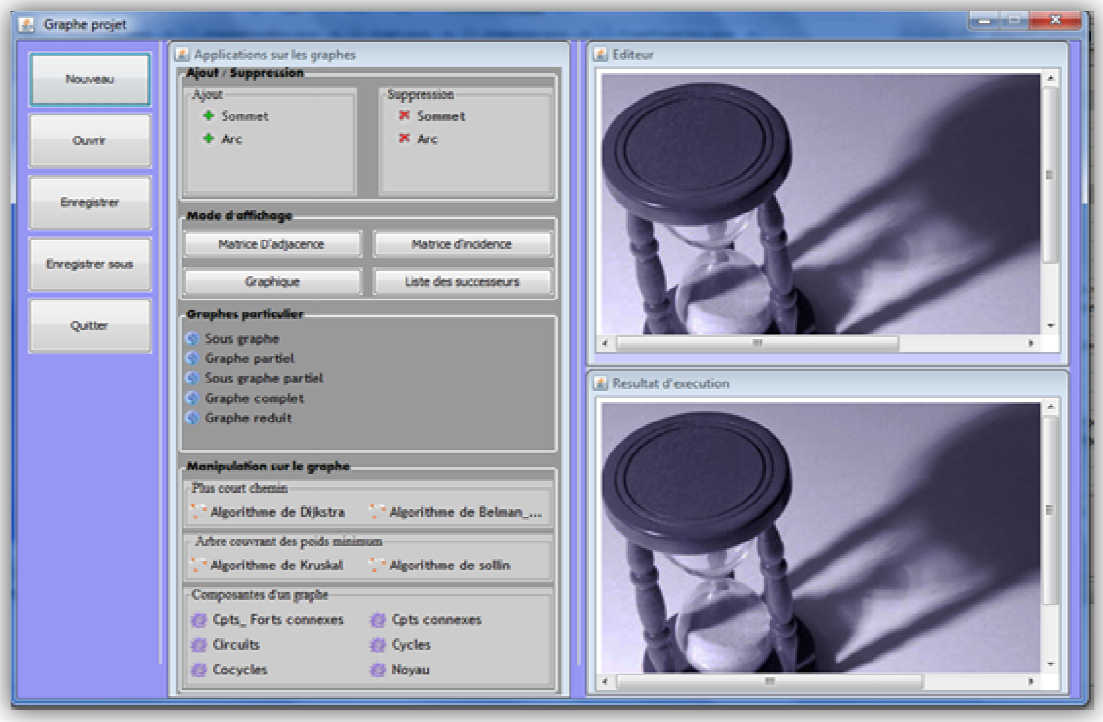

FIG. 3-*Interface graphique principale*

III. Débuter avec graphe projet

Au lancement de l'application, seules les commandes Nouveau et Ouvrir seront actives et bien sûre Quitter et les autres commandes seront désactivés. Après l'insertion d'un graphe tous les autres commandes qui n'est pas actives elles deviennent actives.

Pour commencer notre travail, le premier pas que doit être faire évidement c'est introduire un graphe. Avec **graphe projet** on a deux possibilités pour introduire un graphe, La première consiste à insérer le graphe de façon interactive, en saisissant sa matrice d'adjacence. La seconde possibilité est d'ouvrir directement un fichier graphe enregistré au préalable par l'application.

Conclusion générale

Nous avons abordé dans notre travail les grandes lignes de la théorie des graphes, qui prend une place importante dans la modélisation de nombreux problèmes réels. Cette étude nous a permis d'approfondir notre maitrise du langage JAVA et de la programmation orientée objet. Ainsi, nous avons pu tirer profit des différents avantages qu'offre ce langage.

En implémentant la structure des graphes sous JAVA, un langage fiable et performant, nous avons pu entreprendre une étude détaillée des graphes. Nous avons développé, dans une application sous JAVA en se basant sur les concepts généraux de la théorie des graphes. Dans la réalité, les graphes obtenus par la modélisation peuvent être d'une très grande taille, il devient alors plus aisé de les traiter en les programmant.

Cependant, l'application que nous proposons demeure un chantier ouvert à d'éventuels perfectionnements, de par la présence de nombreux problèmes ouverts tel le problème de coloriage, le problème de flots, . . .

Bibliographie

[1] Sébastien Cabot, cours de *Théorie des graphes*, Université de Montréal, 2008.

[2] cours de *théorie des graphes*, université de MILA.

[3] Y.Dantal and C.Haug. *Théorie des graphes*. Soluscience, 2003.

[4] Éric Sopéna, *cours de Théorie des graphes*, Université de bordeaux, 2001.

[5] Eric Sigward. *Introduction à la théorie des graphes*, Mars 2003.

[6] C. Prins. *Algorithmes de graphes*. Eyrolles, 2003.

[7] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. INTRODUCTION A L'ALGORITHMIQUE Cours et exercices –DUNOD- 2003.

[8] C. Berge. *Graphes et Hypergraphes*. Bordas, 1973.

[9] M. Gondran and M.Minoux. *Graphes et algorithmes*. Eyrolles, 1995.

[10] Roseau. *Exercices Et problèmes Résolus de Recherche Opérationnelle*. Masson, 1995.

[11] Didier Maquin .*Eléments de Théorie des Graphes*. INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, 2003

[12] R. Cabane, *Théorie des graphes*, Techniques de l'Ingénieur, Traité Sciences fondamentales, document AF205.

[14] J-C. Fournier. *Théorie des graphes et applications*. Lavoisier, 2006.

[15] cours de *théorie des graphes*, université de LIEGE.

[16] N.Salmi. *Principe des systèmes d'exploitation*. Pages Bleues, 2007.

[17] M.Lyckova, I. Charon, L.Denoeud, O.Hudry, and A. Lobstein. *Rapport sur le projet* WEB-MOPT : *Optimisation et modélisation du graphe du web*. Ecole Nationale Supérieure des Télécommunications, 2005.

[18] J-L .Guillaume et M.Latapy. *Grands Réseaux d'interactions*. Université Paris 7.

[19] Cay S. Horstmann and Gary Cornell. *Au cœur de Java™*, 2008