Introduction

Vue d'ensemble du programme

Le programme contient :

- Une interface GrapheOriente qui définit tous les prototypes de méthodes qui devront exister dans les deux implémentations du graphe.
- Une classe abstrait dans laquelle apparaissent les attributs communs aux deux implémentations : le nombre de nœud, le nombre d'arcs, et un idGenerator qui servira lors de la création d'Arcs.
- Les deux classes MatGraph et ListGraph qui sont respectivement les implémentations du graphe par une matrice et par une liste.

Choix de représentation

Pour maintenir la cohésion entre les deux implémentations, il a fallu faire un choix quant à la représentation de nos nœuds et arcs dans le graphe.

Un nœud sera un simple entier et ne stockera pas d'information supplémentaire.

Un arc n'a concrètement pas d'existence dans le graphe. Sa représentation dépend uniquement de l'implémentation. Cependant pour permettre l'utilisation du graphe, il a été nécessaire de définir une forme de représentation ; en particulier pour des méthodes qui nécessitent la liste des Arcs. Lorsque le graphe devra renvoyer un Arc, il le renverra sous la forme d'une classe stockant la valeur du nœud de départ , la valeur du nœud d'arrivée ainsi que son identifiant.

Il est important de décrire la classe Encapsulateur puisque celle-ci va structurer nos graphes. Un Encapsulateur est composé d'une valeur, d'une liste d'identifiants et d'un entier *nbArcs*. La valeur sera un booléen pour la matrice d'adjacence(qui correspond à l'existence on non d'un arc entre deux nœuds) et un entier pour la liste d'adjacence(qui correspond à la valeur du nœud d'arrivée d'un arc). La liste des identifiants contient tous les identifiants d'arc qui existent entre deux même nœuds. L'entier *nbArcs* permet de stocker une information différentes selon l'utilisation de l'Encapsulateur.

Courte précision sur le calcul de la complexité

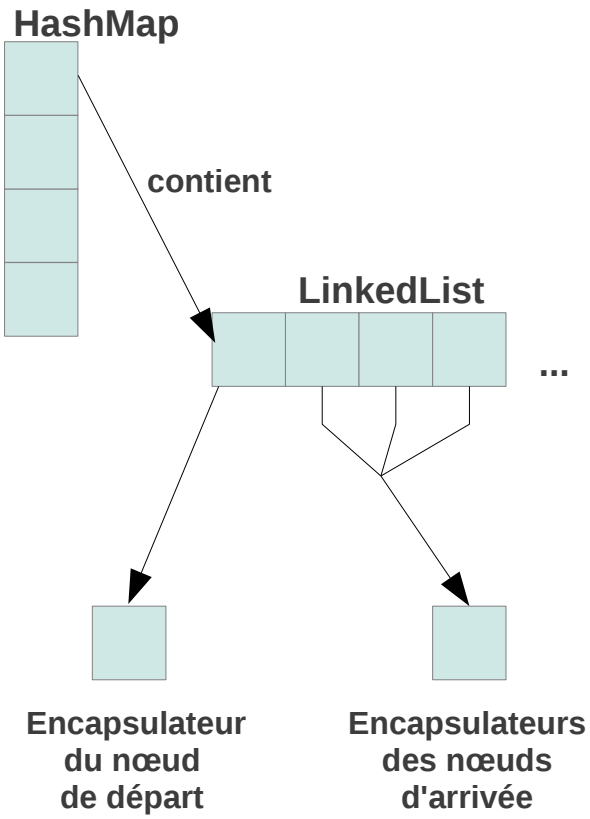
Étant donné que nous n'avons pas accès à l'implémentation du garbage collector, ce paramètre sera donc ignoré. Lors des calcul de complexité, la suppression d'un élément dont l'accès est direct sera considéré comme s'effectuant en temps constant.

Quant aux benchmarks, ils ont été conçu pour isoler le plus possible le coût du garbage collector. Ainsi les tests de suppression seront généralement effectués le plus tard possible. Et chaque test sur un graphe est fait avec une mémoire vide pour ne pas avoir un impact issu d'un autre test.

Liste d'adjacence

Structure de la liste d'adjacence

La liste d'adjacence est représentée par HashMap de LinkedList d'Encapsulateur :



La première implémentation était réalisée à l'aide d'une ArrayList à la place de la HashMap. Cependant comme nous utilisons des entiers pour représenter les nœuds, il était difficile de différencier la valeur d'un nœud et sa position dans la liste. Il aurait été nécessaire de maintenir une structure qui associe la valeur du nœud à son indice.

L'implémentation en ArrayList a donc été modifié au profit de la HashMap. Cette dernière est très simple puisque c'est la valeur du nœud qui va servir de clé pour localiser la LinkedList associée. Le premier élément de la LinkedList est toujours le nœud de départ.

Précision sur l'utilisation des Encapsulateurs

Il parait important de décrire l'utilisation des Encapsulateurs en fonction de leurs rôles.

Lorsqu'un Encapsulateur représente un nœuds d'arrivée

- la valeur de l'Encapsulateur est alors celle du nœud d'arrivée
- la liste des identifiants des Arc contient les identifiants des Arcs entre le nœud de départ et d'arrivée
- le champ *nbArcs* correspond au nombre d'arcs contenus dans l'Encapsulateur.

Lorsque l'Encapsulateur représente un nœud de départ :

- la valeur de l'Encapsulateur est alors celle du nœud de départ
- la liste des identifiants des Arcs est alors vide
- le champs *nbArcs* contient le nombre d'Arc sortants de ce nœud.

L'avantage de stocker le nombre d'arcs sortant pour un nœud de départ se joue au moment de la suppression d'un nœud. En effet pour mettre à jour le nombre d'arc, il faut savoir combien d'arcs associés seront à supprimer. Or la taille de la LinkedList ne correspond pas au nombre d'arcs (étant donné qu'un Encapsulateur peut contenir plusieurs Arcs). Stocker cette information va donc nous éviter un parcours de la liste.

Évaluation sur la complexité des méthodes

Il faut noter que le code n'a pas été prévu dans une optique de vérification. Il n'y a donc aucune vérification de l'existence d'un nœud/arc lors de la suppression/ajout. Un tel contrôle aurait nécessairement jouer sur la complexité.

Ajout d'un nœud

L'ajout d'un nœud correspond à l'ajout d'un couple clé/valeur dans la HashMap, on peut donc prétendre à un O(1).

Ajout d'un arc

Étant donné notre structure de donné et que nous sommes dans le cas d'un graphe non simple, la complexité en O(1) est perdue. En effet la structure impose que dans une LinkedList, il n'y ait pas deux Encapsulateurs portant la même valeur. Il est donc nécessaire de parcourir la liste pour vérifier si un Encapsulateur contient déjà des arcs entre les deux nœuds ou l'on souhaite faire l'ajout;ou en créé un nouveau si aucun arc n'existe

La complexité devient donc un O(|A|). Ce coût aurait pu être perdu si une liste contenait plusieurs fois un même nœud d'arrivée mais différencié par l'id de l'arc pointant dessus.

La structure choisie n'est donc pas optimale, et celle-ci a été conservé car une modification aurait pris trop temps. Ce choix va d'ailleurs coûter très cher au niveau des benchmarks.

Suppression d'un nœud

La suppression d'un nœud correspond à la complexité vue en cours, c'est à dire un parcours de tous les arcs.

Suppression d'un arc

Là aussi la complexité est plus grande que celle vue en cours puisque l'on passe d'un O(1) en O(|A|). Pour avoir accès à l'arc il est nécessaire de parcourir une liste de nœuds d'arrivée. Il est plutôt ardu de retrouver la complexité vue en cours. Cela nécessiterait d'accéder directement à un arc, or la Linked List ne va pas le permettre.

Lister les nœuds

La liste des nœuds s'obtient en parcourant la HashMap, soit une complexité en O(|N|).

Lister les arcs

La liste des arcs nécessite un parcours de la structure. Cependant la complexité n'est pas en O(|A|). Dans notre cas, il est nécessaire de parcourir chaque nœud pour vérifier s'il a des arcs qui en sortent, la complexité monte alors en O(|N+A|).

Il aurait été possible de retrouver un temps standard en maintenant une structure telle qu'une ArrayList qui fournis la liste des nœuds possédant des Arcs.

Lister les successeurs d'un nœud

Pour obtenir la liste des successeurs, il suffit de parcourir la LinkedList associée au nœud. On a donc une complexité en O(|A|). On peut noter que dans le cas d'un graphe non simple, tous les arcs ne seront pas parcourus.

Liste des arcs sortants

Idem que pour la liste des successeurs.

Lister les prédécesseurs d'un nœud

L'algorithme revient à parcourir tous les arcs. La complexité revient donc à un O(|N+A|) pour les mêmes raisons que pour la liste des arcs.

Liste des arcs entrants

Idem que pour la liste des prédécesseurs.

Tests et parcours sur le graphe.

L'implémentation des algorithmes suivants sont les mêmes pour la matrice d'adjacence et la liste d'adjacence. Les méthodes utilisées sont des méthodes définies dans l'interface graphe. Il sera donc intéressant de voir les temps de calcul pour ces tests pour chacun des graphes.

Parcours DFS et BFS

Il n'y a pas grand chose à dire sur ces algorithmes. Ceux-ci suivent l'algorithme de cours quasiment à la lettre. La différence provient essentiellement de la quantité d'information que l'algorithme fournit.

Par exemple dans DFS : la table f qui indique le moment ou un nœud et ces descendants sont parcourus(noir) n'est pas calculée.

L'algorithme renvoie en gros la suite des nœuds parcourus, dans quel ordre et par quel nœud l'on est passé pour arriver au suivant.

Test de simplicité d'un graphe

L'algorithme consiste à demander au graphe la liste des arcs et de voir si celle ci contient deux fois un même arc. L'algorithme nécessite de maintenir une structure à coté pour vérifier l'existence d'un doublon. Dans le pire des cas le temps est de $a^2/2 + a + n$ soit un $O(|A|^2 + |N|)$.

Test de connexité d'un graphe

Le test de connexité consiste à effectuer le parcours du graphe à partir de chaque nœud et vérifier à chaque fois que le nombre d'éléments parcourus correspond au nombre de nœud dans le graphe.

La complexité de l'algorithme est $O(|N|^2 + |A| * |N|)$.

Test d'acyclicité

Le test d'acyclicité est sûrement le test le plus coûteux en terme d'espace et de temps. Pour réaliser cette algorithme nous avons essayer de trouver une solution sans rechercher sur internet.

Le test consiste donc, à partir de la liste des arcs , à trouver l'ensemble des nœuds qui sont accessible à partir d'un autre. Si un nœud a accès à lui-même alors le graphe est cyclique.

Pour cette algorithme on va parcourir la liste pour supprimer les boucles et ignorer les arcs multiples. Puis arc par arc, nous allons ajouter dans une HashMap de TreeSet notre ensemble nœud départ et nœud d'arrivée. Ainsi la HashMap est indicée par la valeur du nœud de départ et le TreeSet associé contient l'ensemble des nœuds qui lui sont accessibles au premier degré (ses fils directs en somme). Puis on parcourt cette structure et pour chaque nœud accessible *nA* on ajoute au nœud de départ *nD* la liste des nœuds accessibles de *nA*. L'algorithme s'arrête lorsque la structure n'évolue plus ou lorsque l'on a trouvé un cycle.

Le problème de cet algorithme est la difficulté à évaluer la complexité. Le coût temporel va énormément dépendre de la forme du graphe. Ainsi dans le cas d'un graphe linéaire (1->2, …, n-1->n), la complexité est de $n^2 + \sum_{0}^{n-1} U_i$ où $U_i = U_{i-1} + i - 1 = \frac{i^2 - i}{2}$ avec $U_0 = 0$ donc une suite qui avoisine la somme des carrés successifs $\sum_{0}^{n-1} U_i \approx \frac{n(n+1)(2n+1)}{6}$ soit un $O(n^3)$. Alors que dans le cas du graphe utilisé dans les benchmarks (voir la partie concernée pour plus d'information sur le graphe) la complexité est de $O(|A| + |N|)$.