1 Matrice d'adjacence

1.1 Choix de classes

Cette forme de structure de données concrète présente les particularités suivantes :

- Propose des complexités optimales (O(1)) pour les opérations élémentaires (ajout/suppression de noeuds/arcs).
- Donne aussi les différentes listes (arcs entrant/sortant et nœuds successeurs/prédécesseurs) avec une bonne compléxité (O(n)).

Le but de cette réflexion est d'aboutir à des choix classes répondant au mieux à ces caractéristiques tout en offrant un minimum d'ergonomie pour l'utilisateur (choix d'un label, opacité de la classe, etc).

1.2 Opérations

Dans un premier temps notre réflexion s'est basée sur les moyens de répondre au premier point décrit ci-dessus. Cette bonne performance nécessite donc un accès direct aux différents éléments de la structure dans le but de modifier ou supprimer un élément de la matrice dans le cas d'une opération sur un arc. Cet accès direct par **indice** primordial, a orienté notre choix vers ArrayList plutôt que LinkedList. En effet cet aspect de cette structure est très performant notamment pour les opérations d'ajouts et de suppressions. Même si ici seul l'atout de la performance d'ajout en fin de ligne nous intéresse (on ajoutera toujours en fin de tableau mais on peut supprimer n'importe où). Le choix d'une HashSet aurait été alors plus judicieux. Cependant cette collection ne tolère ni doublons, ni valeur nulle ni accès à un élément donné.

1.3 Allocation et occupation mémoire

Il faut cependant que l'allocation mémoire soit dynamique et de complexité optimale dans le cas d'ajouts ou de suppressions de noeuds. Une linkedlist occupe un espace mémoire plus grand qu'une arrayList. En effet la linked List a besoin d'un instance de "Noeud" par élément. Une ArrayList est gourmande en occupation mémoire. Elle alloue une taille deux fois supérieure à son contenu. De la même manière si seulement un tiers de l'ArrayList est plein, elle désallouera la deuxième moitié. Cela permet en revanche lors d'ajouts ou de suppressions de ne pas détériorer la complexité par des algorithmes de ré allocation. Dans le cadre de matrice d'adjacence, nous utilisations des graph raisonnablement petits. L'ArrayList présente donc un nouvel argument pour son utilisation.

1.4 Utilisation

L'idéal pour l'utilisateur, serait de pouvoir de pouvoir manipuler un noeud ou un arc par un nom ou label qu'il aurait déterminé lui même. Cependant cette fonctionnalité, requière le stockage de ces information et par conséquences un coût supplémentaire en temps et espace. On peut éviter ces surcoûts en supprimant ces «droits» à l'utilisateur. Il pourrait seulement ajouter **un** nœud/arc. Celui ci serait seulement référencé par son étiquette unique. L'étiquette représenterait

l'indice du nœud dans la structure et donnerait un temps d'accès performant (O(1) pour une ArrayList par exemple). Même si cette manière de procéder est séduisante par sa simplicité d'écriture et sa performance, elle rend l'utilisation des instances très peu intuitive. En effet, lors d'une suppression d'un élément d'indice d'une ArrayList, tous les éléments d'indices $>$ n seront décalés d'un «cran» à droite. La seul moyen pour effectuer une opération sur un élément d'un point de vue utilisateur, est d'utiliser la position actuelle de l'élément dans la structure.

Exemple Si un utilisateur insère trois nœuds dans le graph. Leurs étiquettes uniques seront respectivement 1,2 et 3. Après la suppression du nœud 2, si l'utilisateur veut agir sur l'élément 3, il devra cependant renseigner l'indice 2 puisqu'il s'agit de la position du noeud d'étiquette unique 3.

On constate, que cette manière de procéder rend l'utilisation tellement difficile que stocker les noms des nœuds semble raisonnable. Cela rajoute un coût en O(n) pour le parcours d'un tableau contenant les n nœuds. La mise à jour de ce tableau est en revanche en O(1) si on utilise une ArrayList (il ne s'agit que de suppressions ou d'ajouts) Le cas des arcs n'est pas aussi simple à cause de la structure même d'une matrice d'adjacence. En effet chaque éléments de la matrice contient une liste d'arcs (nulle si la valeur de la case est «false»). Ainsi la plupart des algorithmes sur les arcs (lister arcs entrants/sortants par exemple) demanderait de multiples parcours d'un tableau référençant les noms des arcs. Mais l'opération la plus gourmande serait la mise a jour du graph en cas de n'importe quelle modification impactant sur les arcs. Il faudrait alors systématiquement parcourir le tableau référençant les noms des arcs. On serait alors très loin des complexité en O(1) données par le cours.

1.5 Conclusions

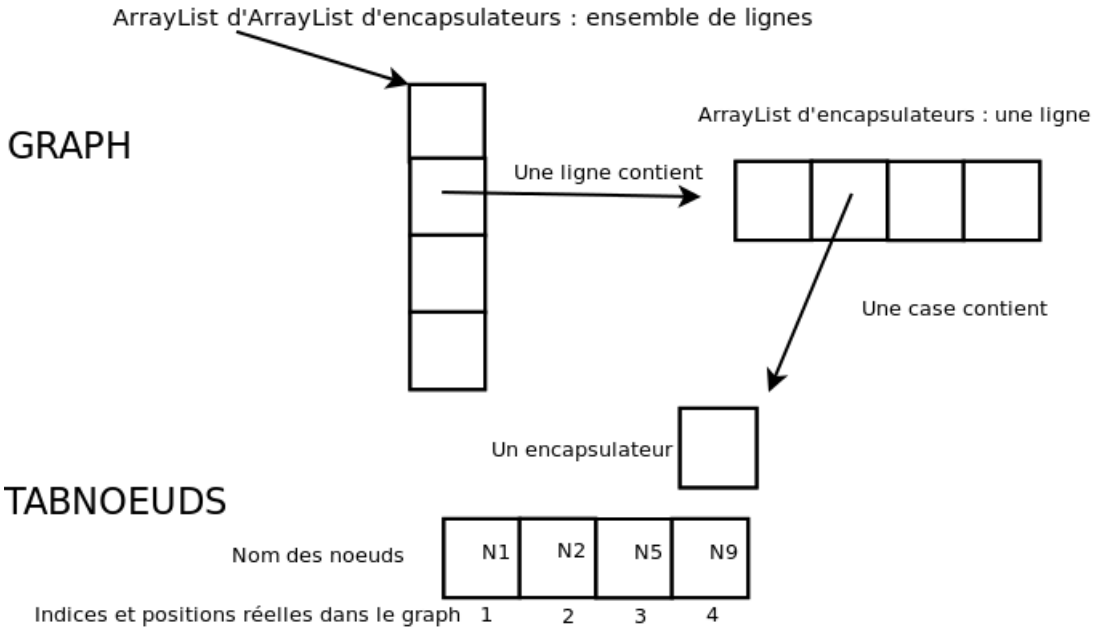
La classe ArrayList semble celle qui semble la plus appropriée pour répondre aux attentes de notre classe de graph que l'on rappelle ci-après:

- Présenter des complexités proches de celles vues en cours.
- Trouver un compromis entre efficacité et facilité d'utilisation.

Une grande partie des opérations sur une matrice résidant sur l'accés des valeurs indexées, il était primordial que ces accès soient faciles. De plus la gestion de l'allocation et la réallocation mémoire de ArrayList est relativement pratique et efficace dans le cadre de notre problème(cf : 1.2). Nous choisissons de permettre la manipulation des nœuds par des noms/labels étant donnée que cela ne détériore que très peu la complexité. En revanche nous ne permettons pas celle des arcs. L'utilisateur devra pour opérer sur un arc avoir connaissance de son l'indice de l'arc dans la structure (cf : 1.3 Exemple). Nous avons conscience que cette méthode rend notre peu fonctionnelle. Une méthode permettant d'éviter une telle concession sans dégrader grandement nos complexité existe certainement. Nous n'avons pas cependant réussi à la trouver dans le temps imparti et nous avons opté pour cette solution.

1.6 Présentation des structures

Une case de la matrice sera un encapsulateur(cf :I). Une ligne de la matrice est une collection ordonnée de cases. Une ligne est donc une ArrayList d'encapsulateurs. Une Matrice est une collection ordonnée de lignes. On obtient donc une ArrayList d'ArrayList d'encapsulateurs. Une ArrayList supplémentaire est em-



ployée pour référencer le nom des nœuds. L'indice d'un nœud représente sa position dans la structure du graph.

Remarque : La manipulation systématique de deux ArrayList pour écrire les différentes méthodes aurait été fastidieuse et illicite. A la place, nous avons crée une classe ArrayList2D permettant de manipuler de facilement cette imbrication.