Master 1 ALMA
Université de Nantes
2010-2011

Projet de TP n°1
Structures complexes et algorithmes

MARGUERITE Alain
RINCE Romain

Université de Nantes
2 rue de la Houssinière,
BP92208, F-44322 Nantes cedex 03, FRANCE

Encadrant : Christophe JERMANN

Table des matières

1 Introduction

1.1 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.

1.2 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.

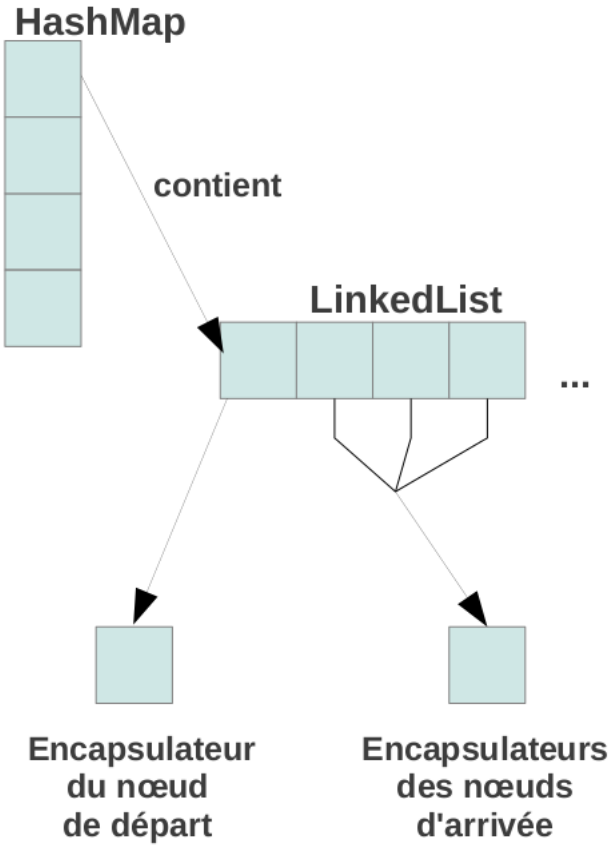
1.3 Courte précision sur le calcul de la complexité

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.

2 Liste d'adjacence

2.1 Structure de la liste d'adjacence

La liste d'adjacence est représentée par une 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.

2.2 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.

2.3 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 joué sur la complexité.

2.3.1 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).

2.3.2 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.

2.3.3 Suppression d'un nœud

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

2.3.4 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.

2.3.5 Lister les noeuds

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

2.3.6 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.

2.3.7 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.

2.3.8 Liste des arcs sortants

Idem que pour la liste des successeurs.

2.3.9 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.

2.3.10 Liste des arcs sortants

Idem que pour la liste des prédécesseurs.

2.3.11 Lister les successeurs 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.

2.3.12 Liste des arcs entrants

Idem que pour la liste des prédécesseurs.

3 Matrice d'adjacence

3.1 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).

3.2 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.

3.3 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.

3.4 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.

3.5 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 :

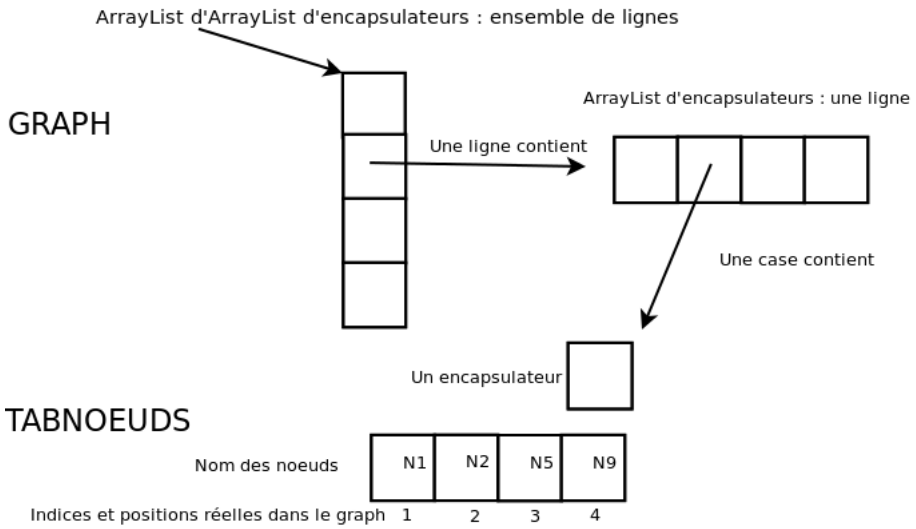
– Présenter des complexités proches de celles vues en cours.

– 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.

3.6 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 employée 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.

4 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.

4.1 Parcours DFS et BFS

Il n'y a pas grand chose à dire sur ces deux implémentations. 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.

4.2 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 $\frac{a^2}{2} + a + n$ soit un O(|A|2 + |N|) .

4.3 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|).

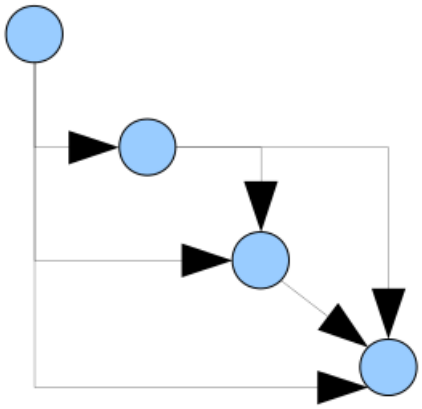
4.4 Test d'acyclicité

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 $n^2 + \sum_0^{n-1}(u_i)$ 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(n3). Le cas de complexité O(n3) semble être le pire cas mais il est malheureusement assez difficile de la prouver. De plus cette complexité ne prend pas en compte une éventuelle copie des nœuds accessibles. Surtout que dans le cas d'un TreeSet la structure doit vérifier si l'élément est déjà présent. Un algorithme plus simple aurait été d'utiliser un des parcours du graphe pour voir si un nœud retrouvait un chemin vers lui-même et itérer sur chaque nœud. La complexité serait alors en O(|N|2 +|A|*|N|) avec notre implémentation. Encore une fois l'algorithme n'a pas été changé par manque de temps.

5 Benchmark

5.1 Type de graphe utilisé pour les tests de performances

Les graphes contiendront 100, 500, 1000, 1500 et 2000 nœuds. Les graphes ont tous une structure commune. Si l'on numérote nos nœuds de 1 à n alors le graphe est défini tel que :
$\forall i \in Noeuds, \forall j \in Noeuds, i < j \Leftrightarrow \exists (i \Rightarrow j) \in Arcs$ Voici un exemple sur 4 nœuds :



Il y aura donc au total $\frac{n^2-n}{2}$ arcs. Ce graphe a l'avantage d'être un cas assez long à traiter pour certains de nos tests. Nous nous sommes limiter à 2000 nœuds car au dessus de ça, la mémoire d'Eclipse déborde. Les tests ont d'ailleurs tous été effectués sur la même machine ; le Heap Space de Java étant fixé à 768Mo.

5.2 Benchmark

Le tableau des résultats ainsi que les graphe de comparaison des temps entre les deux implémentations sont placés en annexe à la fin de ce rapport. Il faut préciser que les tests effectuant la création/suppression des nœuds/arcs correspond à la création/suppression de l'ensemble des nœuds/arcs du graphe. La complexité de l'algorithme est alors multiplié par un facteur qui correspond au

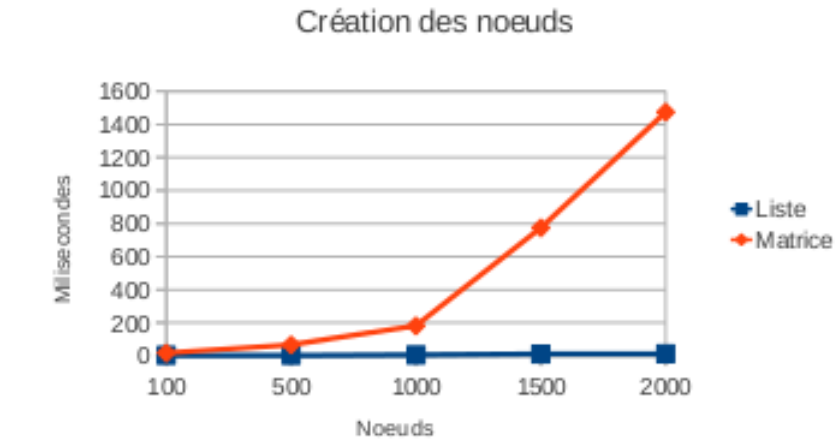
nombre de nœuds/arcs. De plus le nombre d'arcs est quadratique par rapport au nombre de nœuds. Ces résultats mettent très clairement en évidence les algorithmes qui mériteraient d'être modifiés tel que l'acyclicité ou l'ajoute d'arc pour l'implémentation en liste. Mais ces problèmes ont déjà été évoqués. Ce que l'on constate par ailleurs, c'est la rapidité d'accès de la matrice par rapport à la liste. L'accès à un arc, la suppression d'un arc, d'un nœud ou l'accès au prédécesseurs est très faible voire négligeable sur la matrice ; en contrepartie celle-ci va consommer un espace extrêmement important en mémoire. Et chaque ajout de nœud coûtera de plus en plus cher en mémoire. L'implémentation en liste à l'avantage d'être plus modulable

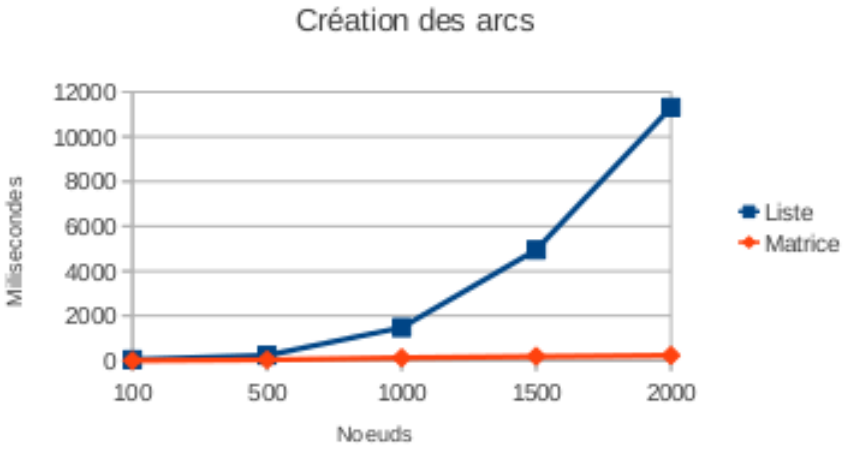
6 Annexes

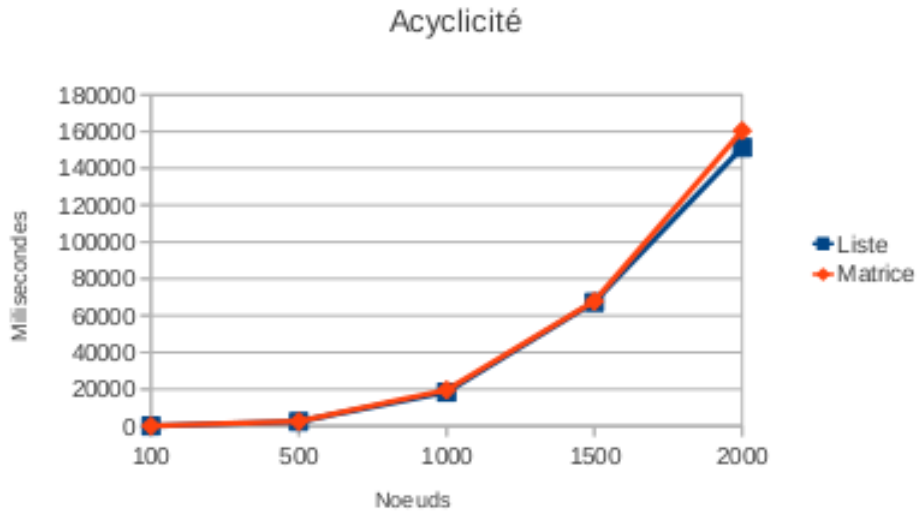
6.1 Annexe 1 :Tableau des résultats

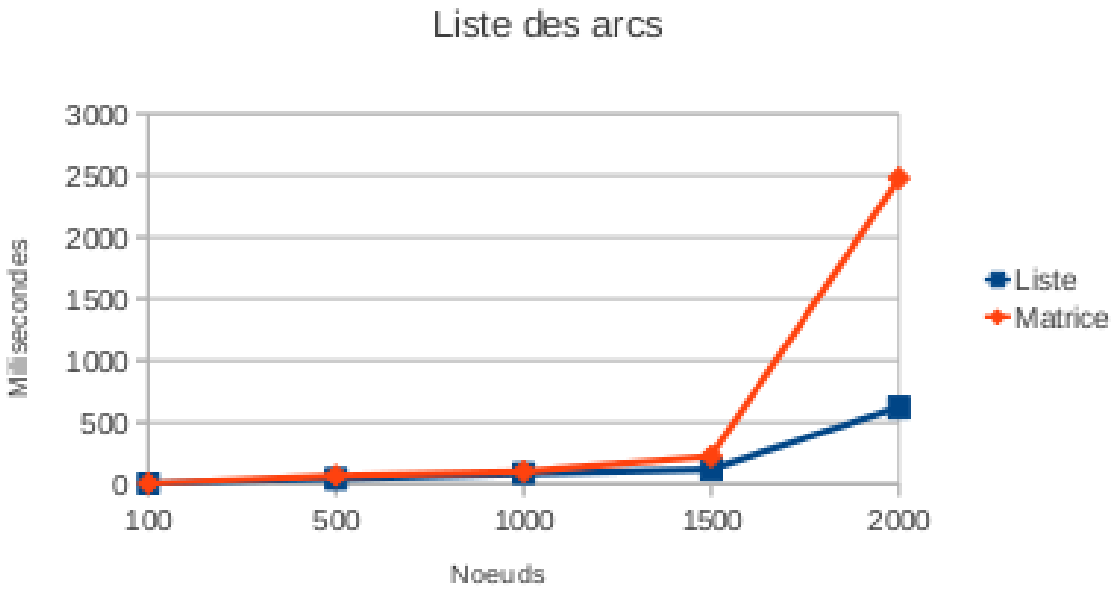
	Liste				
	100	500	1000	1500	2000
Création des nœuds	3	3	5	11	12
Création des arcs	46	235	1481	4944	11303
Liste des nœuds	0	1	1	2	1
Liste des arcs	5	42	86	120	620
Acyclicité	73	2446	18434	67283	151267
Simplicité	11	80	152	438	3872
Connexité	7	74	190	387	835
Successeurs	0	0	0	0	1
Prédécesseur	1	25	32	55	89
Liste des arcs sortants	0	0	0	1	1
Liste des arcs entrants	1	25	37	64	98
DFS	14	86	83	137	435
BFS	6	90	119	138	1940
Suppression des arcs	33	192	1119	3756	9583
Suppression des nœuds	58	1243	7737	24674	60565
	Matrice				
	100	500	1000	1500	2000
Création des nœuds	17	66	181	774	1474
Création des arcs	9	42	128	194	234
Liste des nœuds	0	0	1	1	1
Liste des arcs	6	69	102	223	2476
Acyclicité	82	2531	19594	67823	160414
Simplicité	11	87	171	1912	10397
Connexité	11	89	217	403	2882
Successeurs	0	0	0	0	0
Prédécesseur	0	0	0	1	2
Liste des arcs sortants	0	0	1	0	1
Liste des arcs entrants	0	0	0	2	1
DFS	9	34	77	137	458
BFS	12	67	128	168	307
Suppression des arcs	3	26	57	64	151
Suppression des nœuds	6	44	532	2138	5214

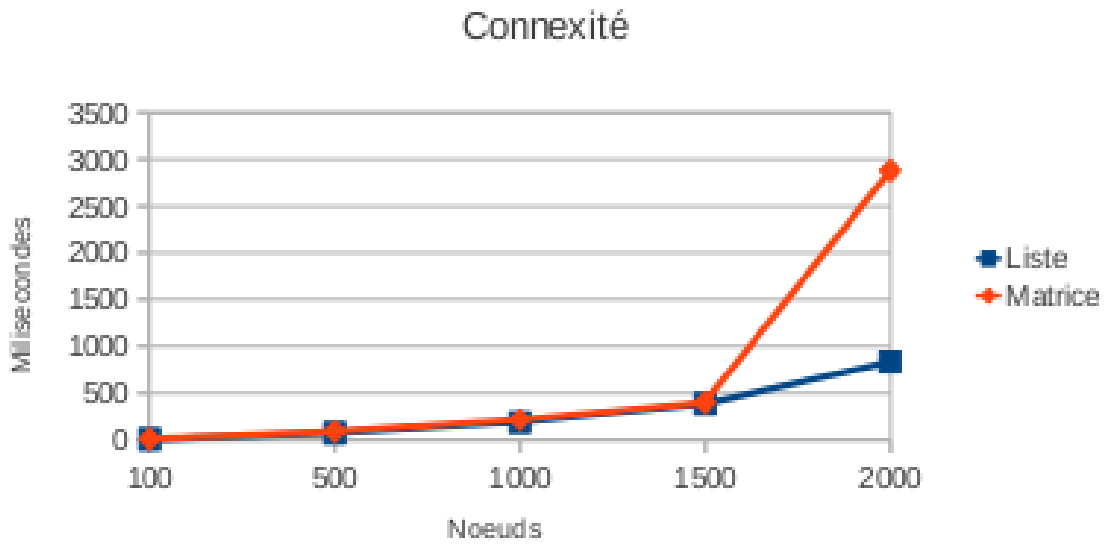
6.2 Annexe 2 : Graphes de comparaison

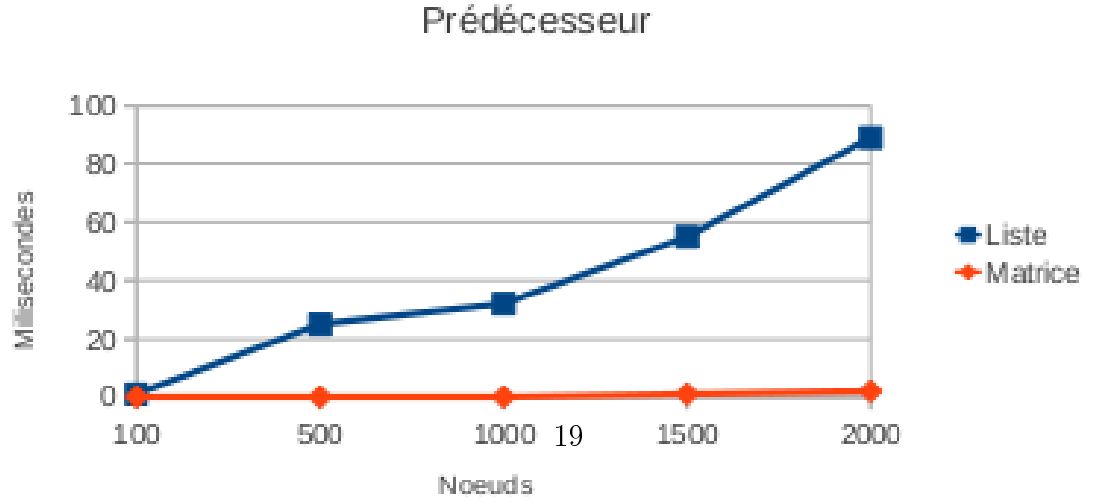


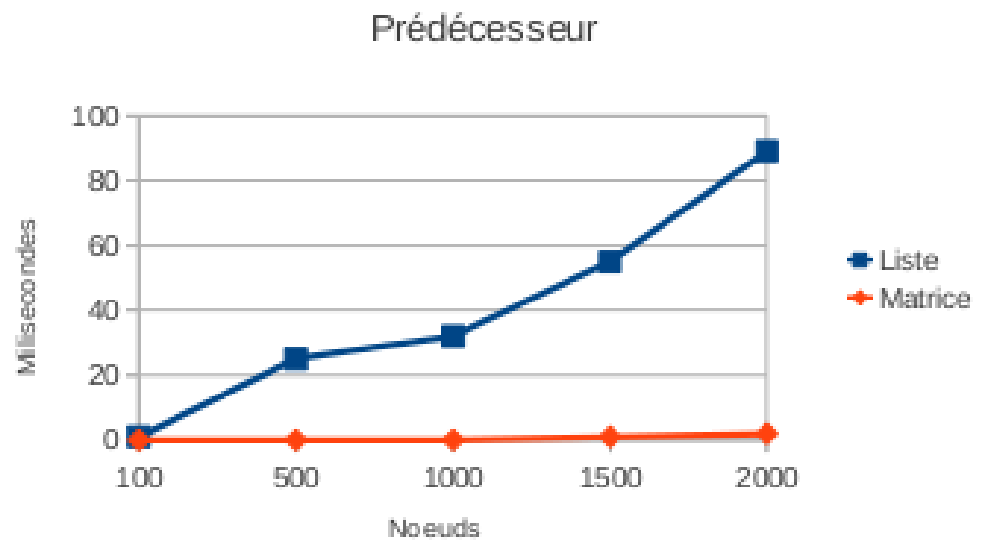


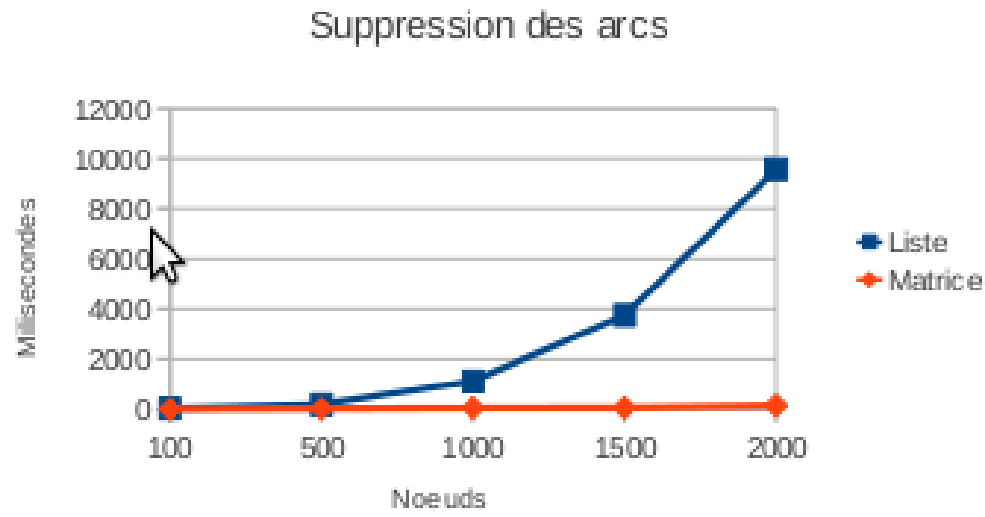


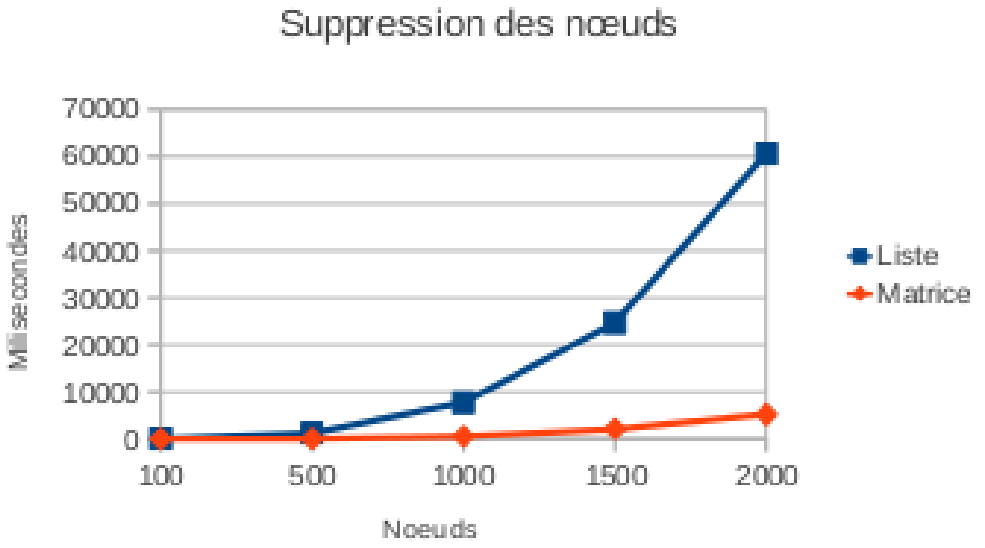

Liste des arcs
3000
2500
2000
1500
1000
500
0
Millisecondes
100
500
1000
1500
2000
Noeuds
Liste
Matrice

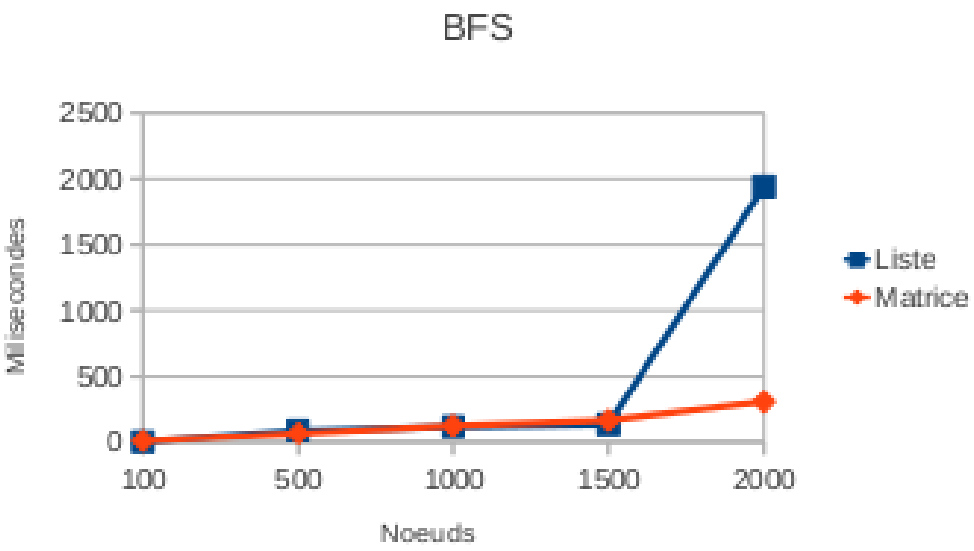

Connexité
3500
3000
2500
2000
1500
1000
500
0
Millisecondes
100
500
1000
1500
2000
Noeuds
Liste
Matrice


Prédécesseur
100
80
60
40
20
0
Millisecondes
100
500
1000
1500
2000
Noeuds
Liste
Matrice


Prédécesseur
100
80
60
40
20
0
Millisecondes
100
500
1000
1500
2000
Noeuds
Liste
Matrice


Suppression des arcs
12000
10000
8000
6000
4000
2000
0
Millisecondes
100
500
1000
1500
2000
Noeuds
Liste
Matrice


Suppression des nœuds
70000
60000
50000
40000
30000
20000
10000
0
Millisecondes
100
500
1000
1500
2000
Noeuds
Liste
Matrice


BFS
2500
2000
1500
1000
500
0
Millisecondes
100
500
1000
1500
2000
Noeuds
Liste
Matrice