

# Table des matières

I.Introduction.....	2
A)Vue d'ensemble du programme.....	2
B)Choix de représentation.....	2
C)Courte précision sur le calcul de la complexité.....	2
II.Liste d'adjacence.....	3
A)Structure de la liste d'adjacence.....	3
B)Précision sur l'utilisation des Encapsulateurs.....	3
C)Évaluation sur la complexité des méthodes.....	4
a)Ajout d'un nœud.....	4
b)Ajout d'un arc.....	4
c)Suppression d'un nœud.....	4
d)Suppression d'un arc.....	4
e)Lister les nœuds.....	4
f)Lister les arcs.....	5
g)Lister les successeurs d'un nœud.....	5
h)Lister les arcs sortants.....	5
i)Lister les prédécesseurs d'un nœud.....	5
j)Lister les arcs entrants.....	5
III.Tests et parcours sur le graphe.....	6
A)Parcours DFS et BFS.....	6
B) Test de simplicité d'un graphe.....	6
C)Test de connexité d'un graphe.....	6
D)Test d'acyclicité.....	6
IV.Benchmarks.....	7
A)Type de graphe utilisé pour les tests de performances.....	7
B)Benchmarks.....	7
V.Annexe.....	8
A)Annexe 1 :Tableau des résultats.....	8
B)Annexe 2 : Graphes de comparaison.....	9

# I. Introduction

## A) *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 abstraite dans laquelle apparaissent les attributs communs aux deux implémentations : le nombre de nœuds, 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.

## B) *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 ou 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 mêmes nœuds. L'entier `nbArcs` permet de stocker une information différente selon l'utilisation de l'`Encapsulateur`.

## C) *Courte précision sur le calcul de la complexité*

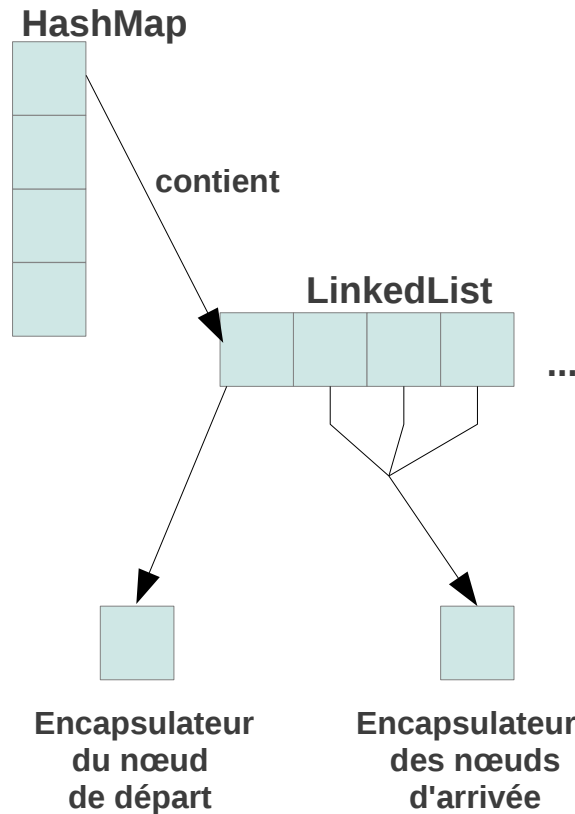
Étant donné que nous n'avons pas accès à l'implémentation du garbage collector, ce paramètre sera donc ignoré. Lors du calcul de complexité, la suppression d'un élément dont l'accès est direct sera considéré comme s'effectuant en temps constant.

Quant aux benchmarks, ils ont été conçus pour isoler le plus possible le coût du garbage collector. Ainsi les tests de suppression seront généralement effectués le plus tard possible.

## II. Liste d'adjacence

### A) 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 modifiée 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.

### B) Précision sur l'utilisation des Encapsulateurs

Il paraît 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.

### **C) 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.

#### **a) 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  $O(1)$ .

#### **b) Ajout d'un arc**

Étant donné notre structure de données 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 créer 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 conservée car une modification aurait pris trop de temps. Ce choix va d'ailleurs coûter très cher au niveau des benchmarks.

#### **c) 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.

#### **d) 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.

#### **e) Lister les nœuds**

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

#### **f) 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.

#### **g) 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.

#### **h) Lister les arcs sortants**

Idem que pour la liste des successeurs.

#### **i) 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.

#### **j) Lister les arcs entrants**

Idem que pour la liste des prédécesseurs.

### III. 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.

#### A) *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 où un nœud et ses 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.

#### B) *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 côté pour vérifier l'existence d'un doublon. Dans le pire des cas le temps est de  $\frac{a^2}{2} + a + n$  soit un  $O(|A|^2 + |N|)$ .

#### C) *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  $O(|N|^2 + |A| * |N|)$ .

#### D) *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 essayé 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 accessibles à partir de chaque nœud. Si un nœud a accès à lui-même alors le graphe est cyclique.

Pour cette algorithme on va parcourir la liste des arcs 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. La HashMap sera indexée 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 à partir 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  $n^2 + \sum_{i=0}^{n-1} U_i$  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  $\sum_{i=0}^{n-1} U_i \approx \frac{n(n+1)(2n+1)}{6}$  soit un  $O(n^3)$ . Le cas de complexité  $O(n^3)$  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.

## IV. Benchmarks

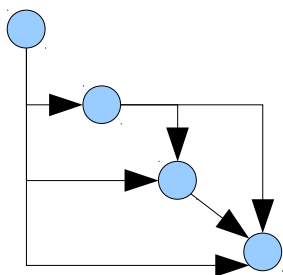
### A) Type de graphe utilisé pour les tests de performances

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

$$\forall i \in \text{Noeuds}, \forall j \in \text{Noeuds}, i < j \Leftrightarrow \{\exists (i \rightarrow j) \in \text{Arcs}\}$$

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 limités 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.

### B) Benchmarks

Le tableau des résultats ainsi que le 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 correspondent à la création/suppression de l'ensemble des nœuds/arcs du graphe. La complexité de l'algorithme est alors multipliée 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 a l'avantage d'être plus modulable.

## V. Annexe

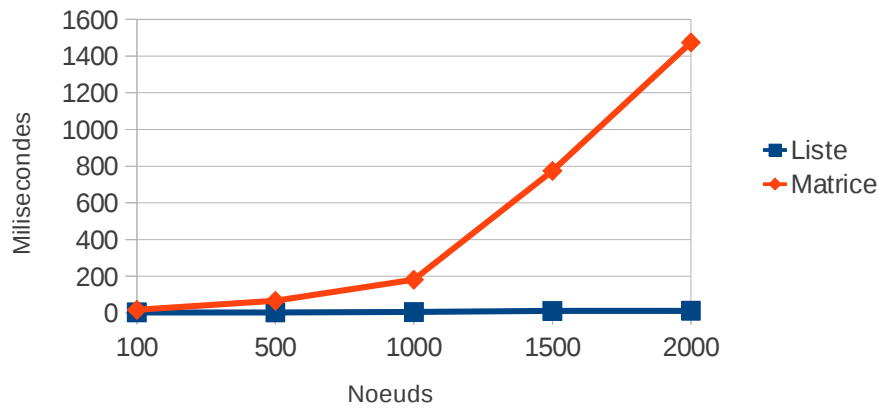
### A) Annexe 1 :Tableau des résultats

	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

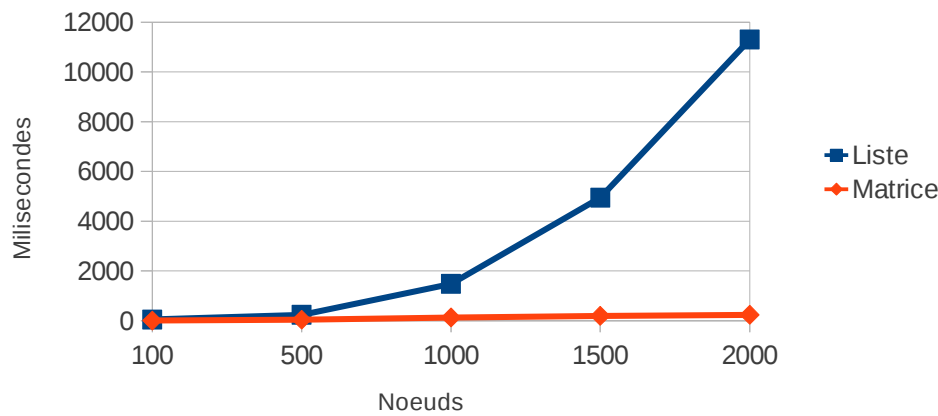


**B) Annexe 2 : Graphes de comparaison**

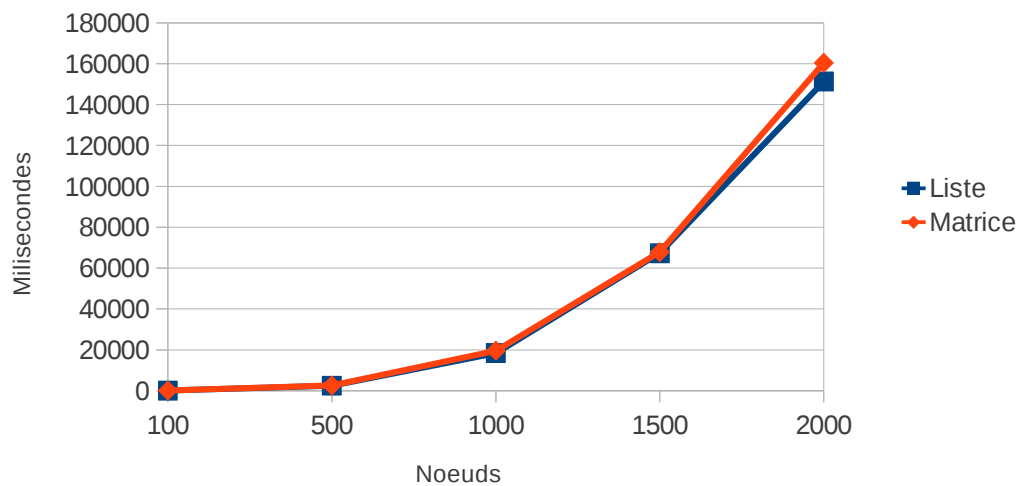
Création des noeuds



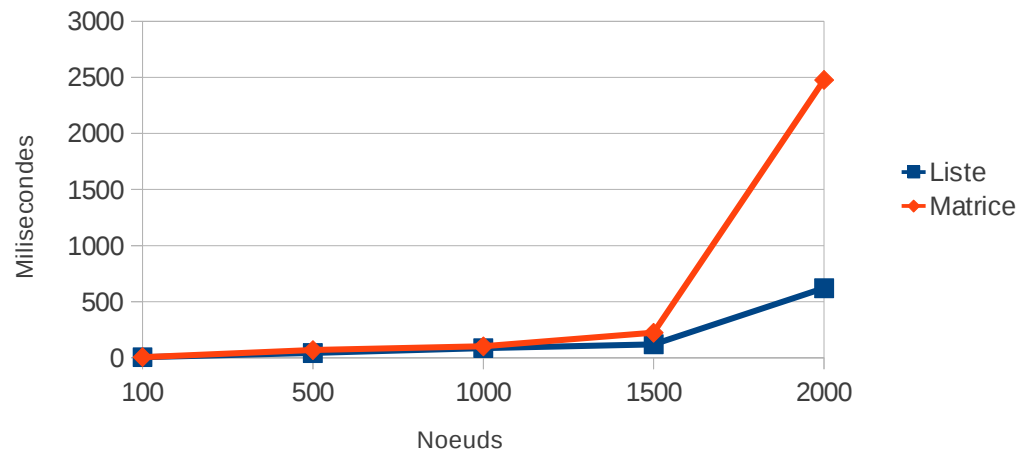
Création des arcs



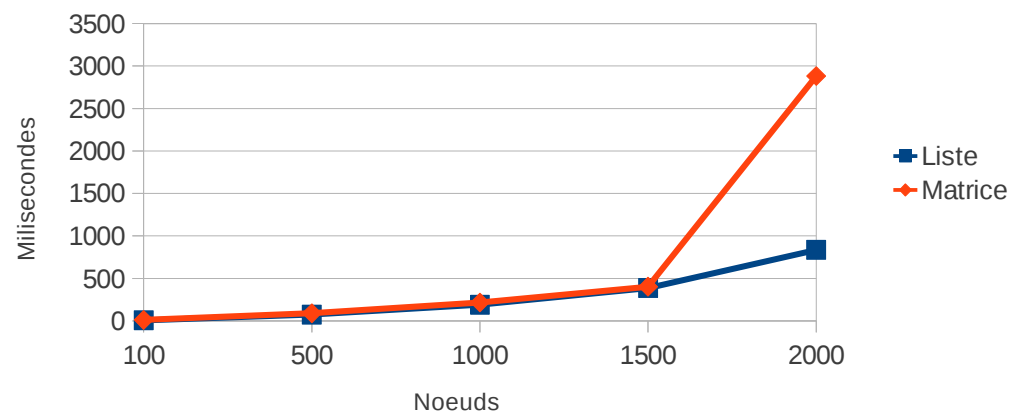
Acyclicité



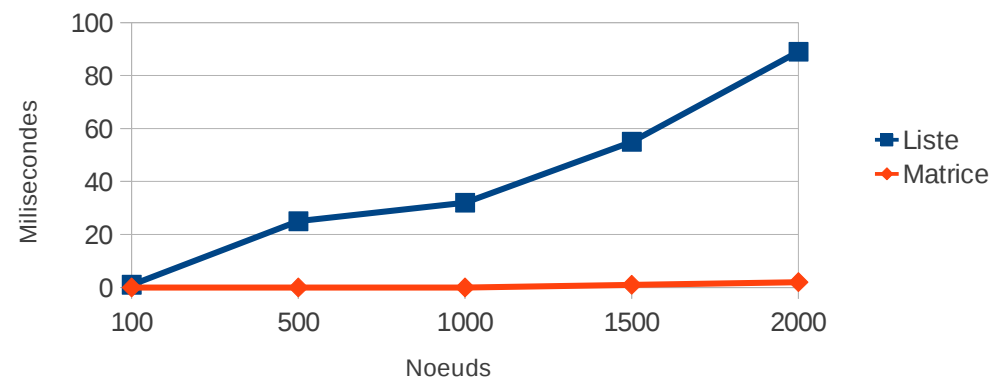
Liste des arcs



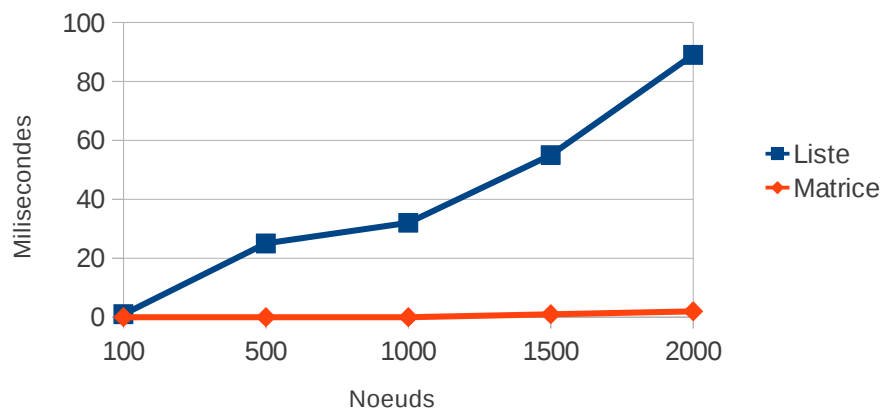
Connexité



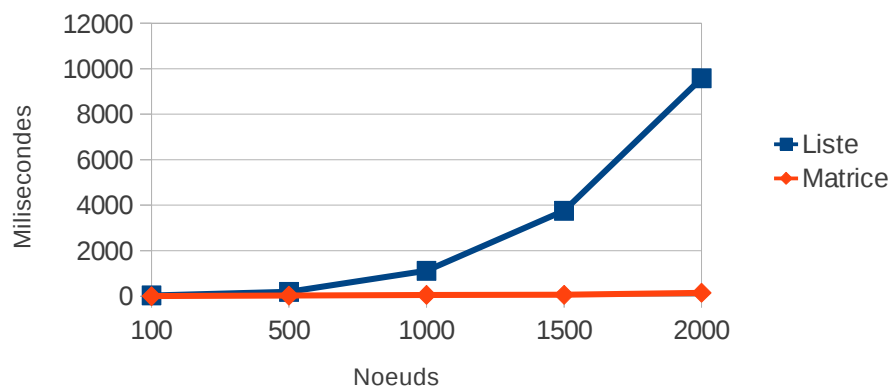
Prédécesseur



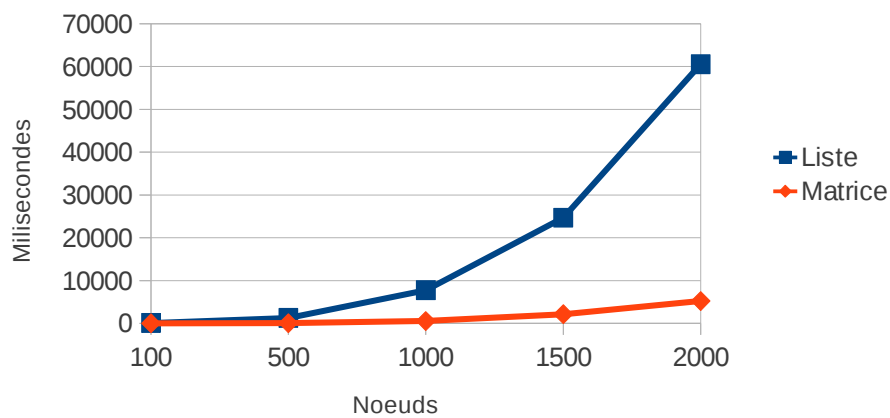
### Prédécesseur



### Suppression des arcs



### Suppression des nœuds



## BFS

