FORNALI Damien

Master I IFI - 2017-2018

Université Nice-Sophia-Antipolis

Strongly Connected Components

Tarjan, Kosaraju, Gabow

Le but de ce projet est d'implémenter trois algorithmes de recherche de composantes fortement connexes. Nous comparerons ensuite leurs performances sur des graphes de tailles et densités différentes.

J'ai choisi **Java** comme langage car il est celui sur lequel j'ai le plus d'aisance et celui qui me semble le plus adéquat pour ce projet.

Méthodes de résolution *Pr. REGIN Jean-Charles*

I. Architecture de code

Le package ***app*** contient le point d'entrée du projet.

Le package ***core*** contient les trois algorithmes ***TarjanSCC***, ***KosarajuSCC*** et ***GabowSCC***. Ils implémentent tous l'interface ***IGraphProcessor*** (***GP***). Un ***GP*** doit pouvoir traiter un ***Graph*** et afficher les résultats de son exécution.

Chacun des ***GP*** possède un ***singleton***. Les structures de données utilisées pour le parcours du graphe sont générées lors de l'appel de traitement du graphe.

Chaque singleton possède un attribut ***SCC*** qui est modifié au cours du traitement de l'algorithme. C'est une représentation des composantes fortement connexes résultantes.

Le package ***model*** contient ma modélisation du graphe (nombre de sommets et leur adjacence) ainsi que celle du ***SCC*** résultat. L'objet graphe permet facilement d'ajouter une arête dirigée ou bien d'itérer sur les sommets adjacents d'un sommet.

Le package ***utils*** quant à lui contient des utilitaires. ***Constants*** contient une valeur entière initiale utilisée dans les trois algorithmes et ***IntRef*** est un objet utilisé afin de donner une valeur aux sommets lors de leur découverte (référence en paramètre de méthode afin d'éviter d'utiliser une variable statique).

De plus, ***Timer*** permet de connaître le temps de traitement d'un algorithme et ***GraphBuilder*** génère un graphe de tailles données en plaçant les arêtes de manière aléatoire entre tous les sommets (sans répétitions).

II. Utilisation

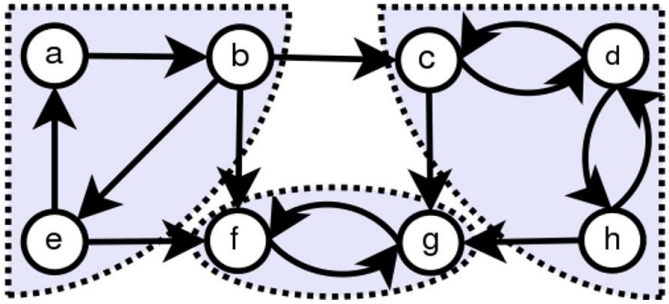
I. Génération du graphe

- *Automatique*

```
Graph graph = GraphBuilder.get().generate(100, 125);
```

- *Manuel*

```
// of(verticesCount)
Graph graph = Graph.of(8);

graph.addEdge(0, 1); // a -> b
graph.addEdge(1, 4); // b -> e
graph.addEdge(4, 0); // e -> a
graph.addEdge(1, 2); // b -> c
graph.addEdge(2, 3); // c -> d
graph.addEdge(2, 6); // c -> g
graph.addEdge(3, 2); // d -> c
graph.addEdge(3, 7); // d -> h
graph.addEdge(7, 3); // h -> d
graph.addEdge(7, 6); // h -> g
graph.addEdge(5, 6); // f -> g
graph.addEdge(6, 5); // g -> f
```



II. Lancement du traitement

- *Les trois algorithmes se lancent de la même manière*

```
TarjanSCC.get().process(graph);

KosarajuSCC.get().process(graph);

GabowSCC.get().process(graph);
```

III. Affichage du résultat du traitement

- Les trois algorithmes affichent leurs résultats de la même façon

```
TarjanSCC.get().output();

KosarajuSCC.get().output();

GabowSCC.get().output();
```

IV. Utilisation du Timer

- Démarrage du timer

```
Timer.start();
```

- Arrêt

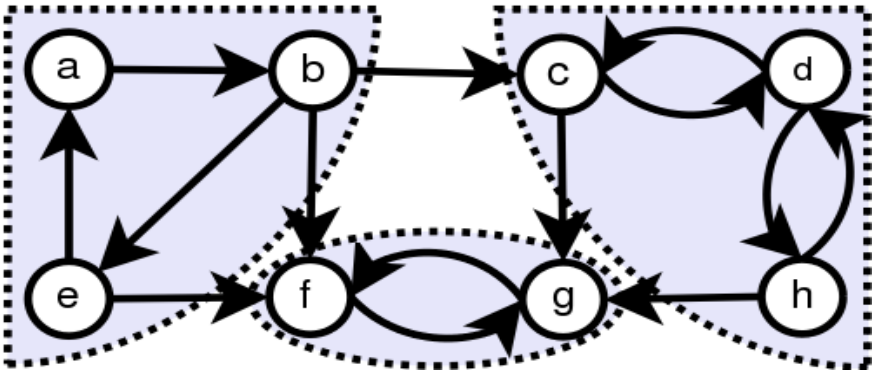
```
Timer.end();
```

- Affichage du temps réel écoulé entre le démarrage et l'arrêt, la granularité peut-être choisie. La granularité standard est **MILLISECONDS** si le temps écoulé est supérieur à 0 millisecondes, **NANOSECONDS** sinon.

```
Timer.output();
Timer.output(Timer.MILLISECONDS);
Timer.output(Timer.NANOSECONDS);
```

V. Exemple complet et affichage

- *Lancement du traitement des trois processeurs sur le graphe ci-dessous et affichages*




```
Graph graph = Graph.of(8); // 8 vertices

graph.addEdge(0, 1); // a -> b
graph.addEdge(1, 4); // b -> e
graph.addEdge(4, 0); // e -> a
graph.addEdge(1, 2); // b -> c
graph.addEdge(2, 3); // c -> d
graph.addEdge(2, 6); // c -> g
graph.addEdge(3, 2); // d -> c
graph.addEdge(3, 7); // d -> h
graph.addEdge(7, 3); // h -> d
graph.addEdge(7, 6); // h -> g
graph.addEdge(5, 6); // f -> g
graph.addEdge(6, 5); // g -> f

Timer.start();
TarjanSCC.get().process(graph);
Timer.end();
TarjanSCC.get().output();
Timer.output();

Timer.start();
KosarajuSCC.get().process(graph);
Timer.end();
KosarajuSCC.get().output();
Timer.output();

Timer.start();
GabowSCC.get().process(graph);
Timer.end();
GabowSCC.get().output();
Timer.output();
```

Même SCC (organisées différemment) avec des temps de traitement similaires

```
>> Tarjan
SCC: [[5,6], [7,3,2], [4,1,0]]
SCC: [[2], [3], [3]]
Time: 2 ms.
>> Kosaraju
SCC: [[0,4,1], [2,3,7], [6,5]]
SCC: [[3], [3], [2]]
Time: 1 ms.
>> Gabow
SCC: [[5,6], [7,3,2], [4,1,0]]
SCC: [[2], [3], [3]]
Time: 629717 ns.
```

III. Performances

J'ai généré des graphes de taille croissante mais de densité fixe (nombre d'arêtes / nombre de sommets).

J'ai choisi de tester les algorithmes sur trois densités différentes : 0.5, 1.25, et 2.0. Nous verrons pour chaque densité quel est l'algorithme qui semble le plus performant.

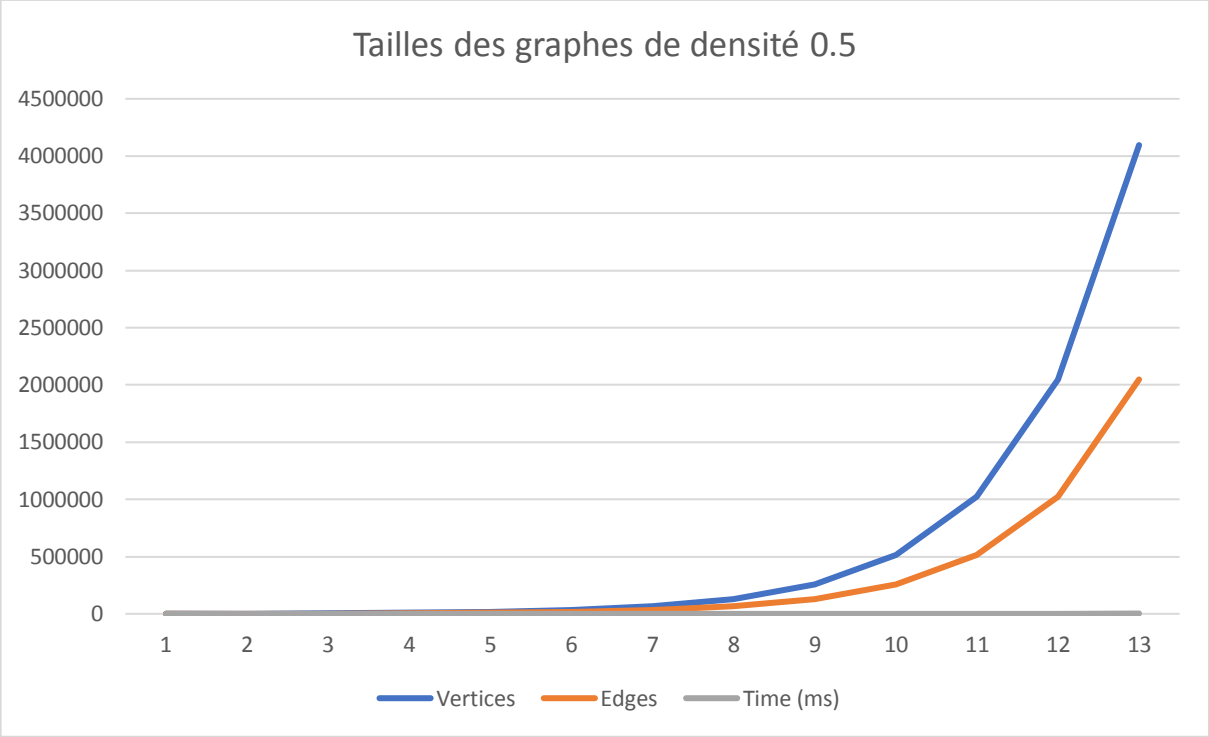
Pour chaque densité et jeu de graphes, les temps de traitement semblent bien croître linéairement.

I. Densité de 0.5

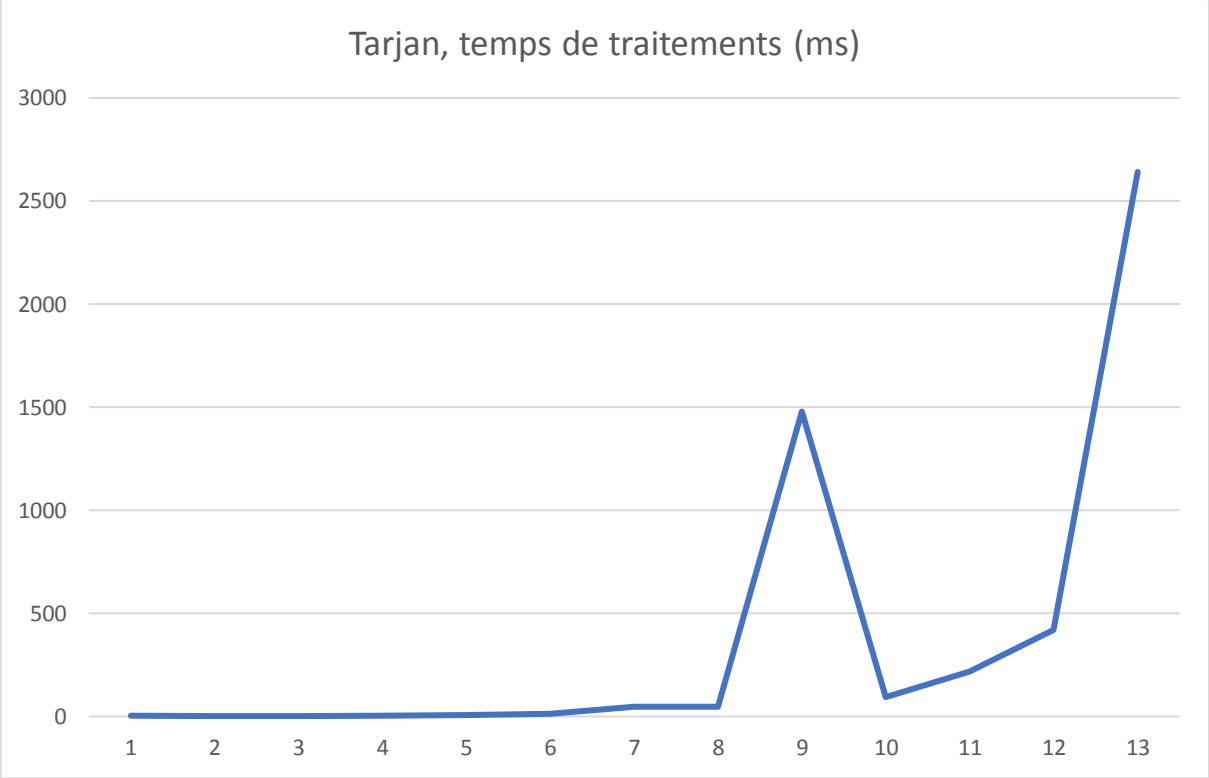
Ici, le nombre de sommets et d'arêtes doublent de graphe en graphe, les temps de traitement doublent donc aussi.

```
GraphBuilder gb = GraphBuilder.get();
Graph graph1 = gb.generate(1000, 500);
Graph graph2 = gb.generate(2000, 1000);
Graph graph3 = gb.generate(4000, 2000);
Graph graph4 = gb.generate(8000, 4000);
Graph graph5 = gb.generate(16000, 8000);
Graph graph6 = gb.generate(32000, 16000);
Graph graph7 = gb.generate(64000, 32000);
Graph graph8 = gb.generate(128000, 64000);
Graph graph9 = gb.generate(256000, 128000);
Graph graph10 = gb.generate(512000, 256000);
Graph graph11 = gb.generate(1024000, 512000);
Graph graph12 = gb.generate(2048000, 1024000);
Graph graph13 = gb.generate(4096000, 2048000);
```

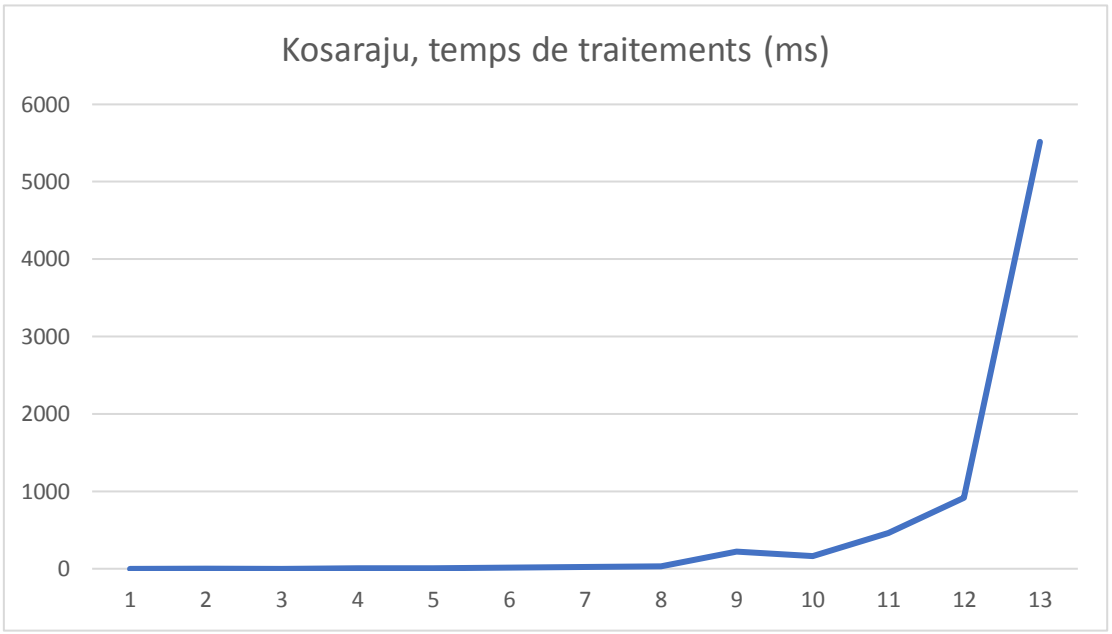
Jeu de graphes de densité 0.5


Tailles des graphes de densité 0.5
4500000
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
Vertices
Edges
Time (ms)

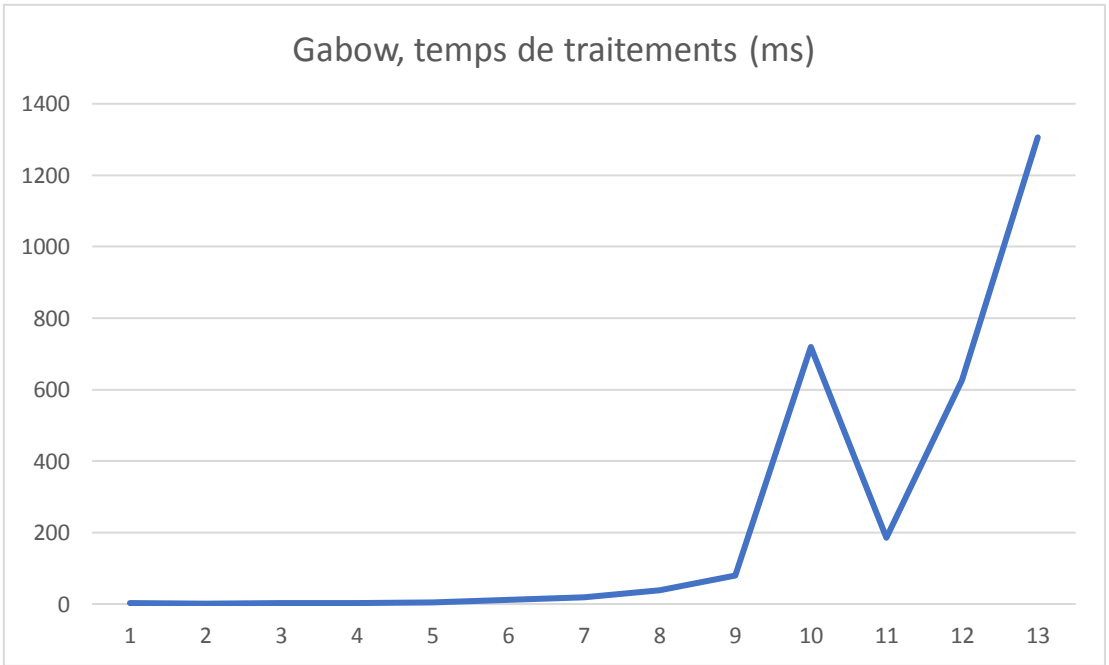
Tarjan 0.5


Tarjan, temps de traitements (ms)
3000
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8
9
10
11
12
13

Kosaraju 0.5



Gabow 0.5



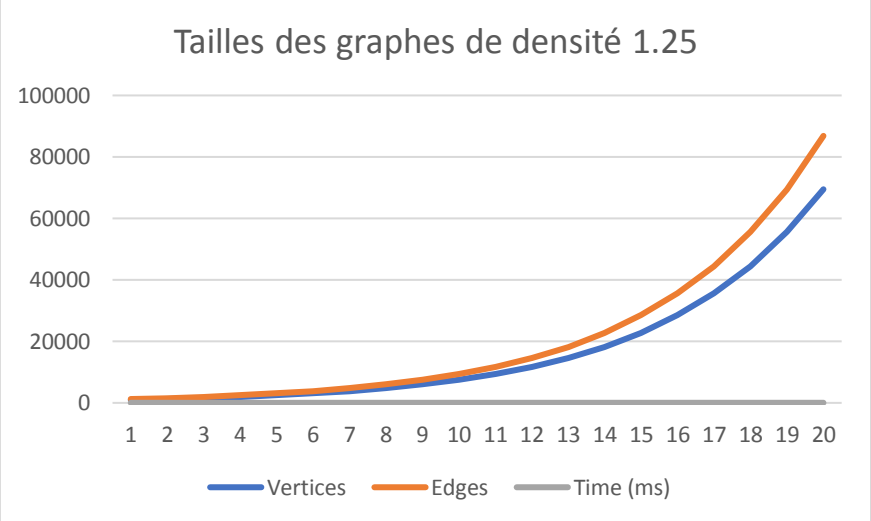
Bien qu'il y ait une irrégularité au graphe 10 pour **GabowSCC** et au graphe 9 pour **TarjanSCC**, ils semblent être, dans l'ordre, les plus performants.

II. Densité de 1.25

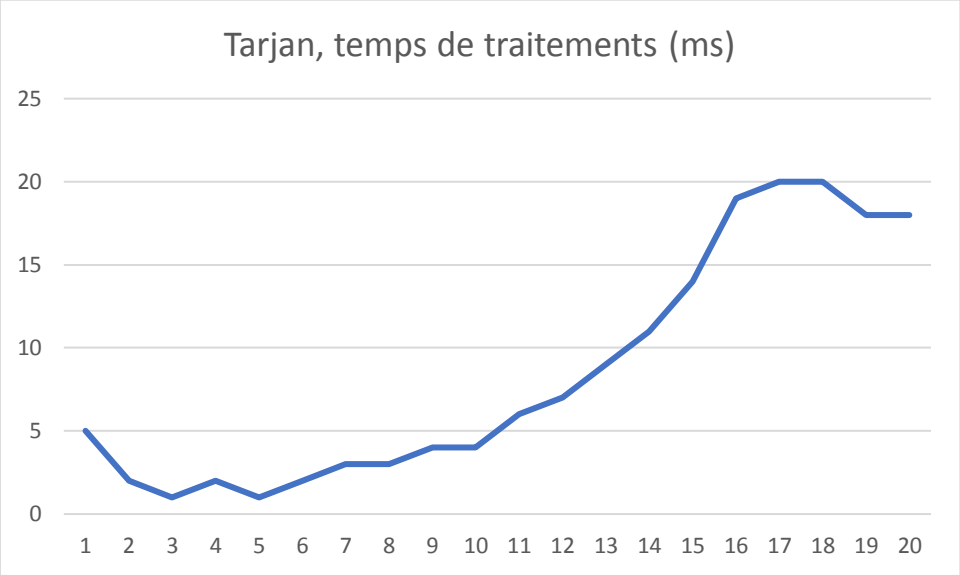
Ici, le nombre de sommets et d'arêtes augmentent progressivement. Les temps de traitements augmentent linéairement.

Jeu de graphes de densité 1.25

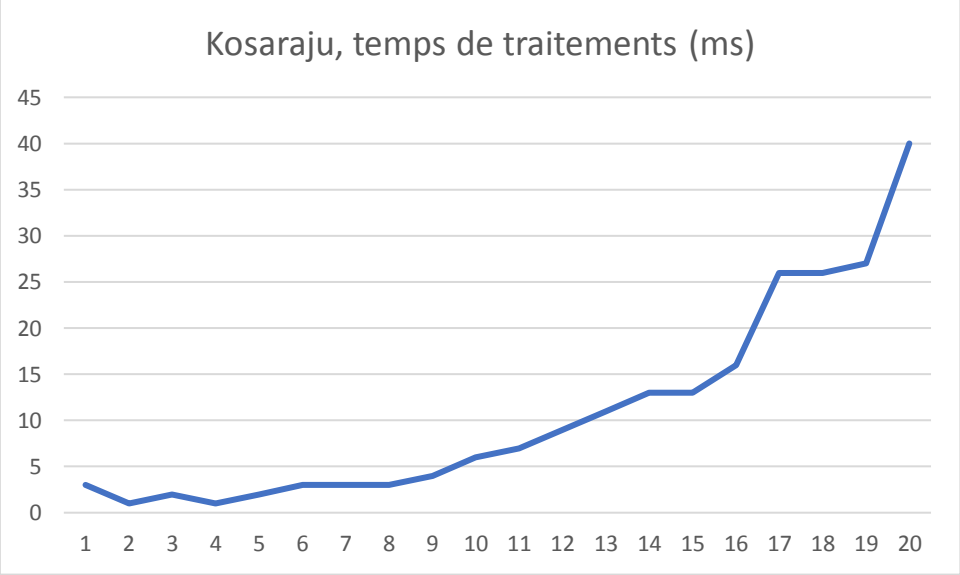
```
GraphBuilder gb = GraphBuilder.get();

Graph graph1 = gb.generate(1000, 1250);
Graph graph2 = gb.generate(1500, 1563);
Graph graph3 = gb.generate(1563, 1954);
Graph graph4 = gb.generate(1954, 2443);
Graph graph5 = gb.generate(2443, 3054);
Graph graph6 = gb.generate(3054, 3818);
Graph graph7 = gb.generate(3818, 4773);
Graph graph8 = gb.generate(4773, 5967);
Graph graph9 = gb.generate(5967, 7459);
Graph graph10 = gb.generate(7459, 9324);
Graph graph11 = gb.generate(9324, 11655);
Graph graph12 = gb.generate(11655, 14569);
Graph graph13 = gb.generate(14569, 18212);
Graph graph14 = gb.generate(18212, 22765);
Graph graph15 = gb.generate(22765, 28456);
Graph graph16 = gb.generate(28456, 35570);
Graph graph17 = gb.generate(35570, 44463);
Graph graph18 = gb.generate(44463, 55579);
Graph graph19 = gb.generate(55579, 69474);
Graph graph20 = gb.generate(69474, 86843);
```



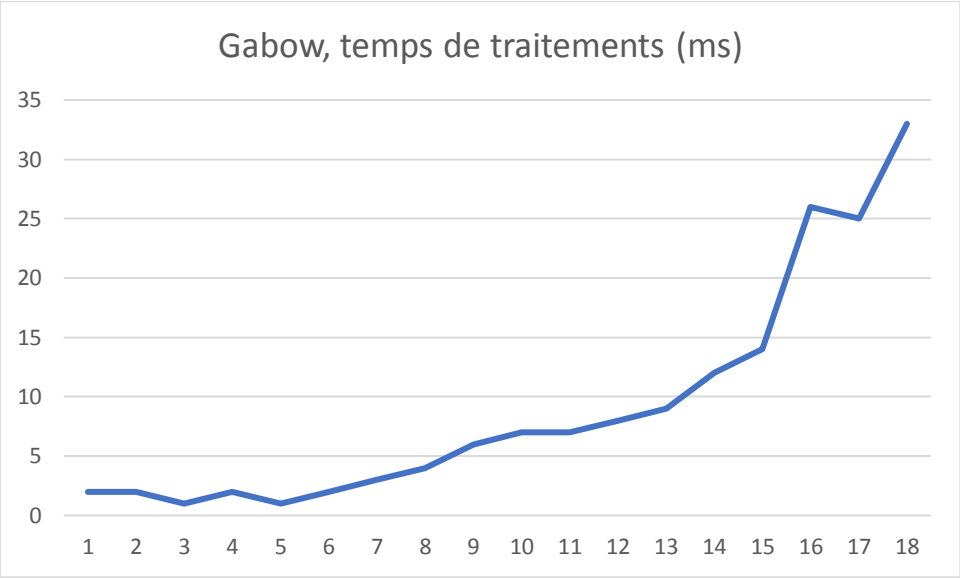
Tarjan



Kosaraju



Gabow



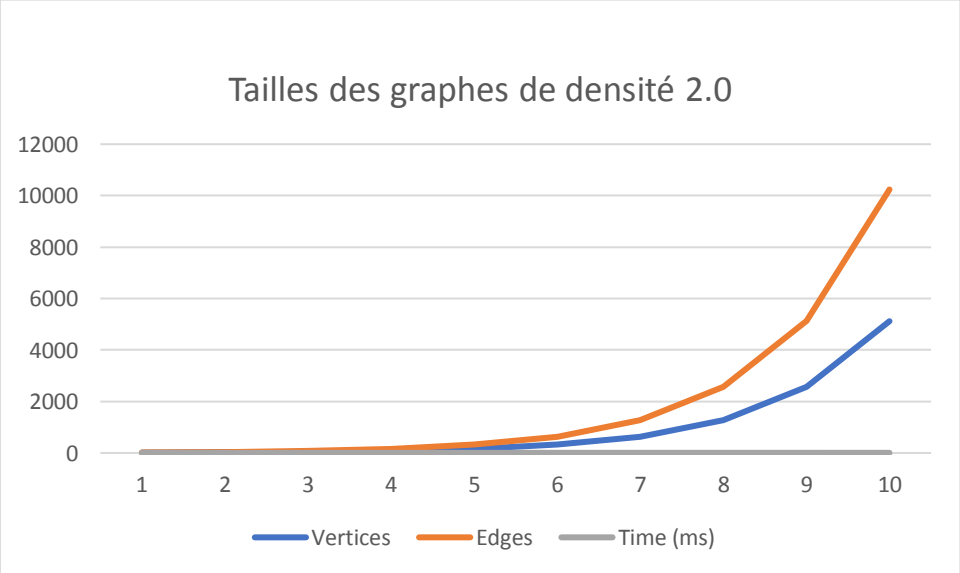
GabowSCC ne peut pas traiter les graphes 19 et 20.

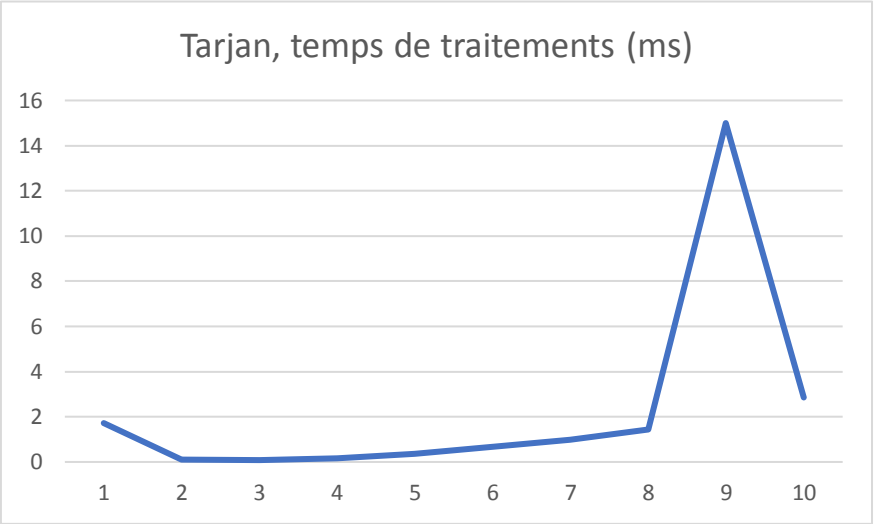
Les trois algorithmes ont des performances similaires mais **TarjanSCC** semble être le plus performant bien qu'il soit le second à lancer une **StackOverFlow** après **GabowSCC** (graphe 19). **KosarajuSCC** semble être celui qui résiste le mieux à l'exception.

III. Densité de 2.0

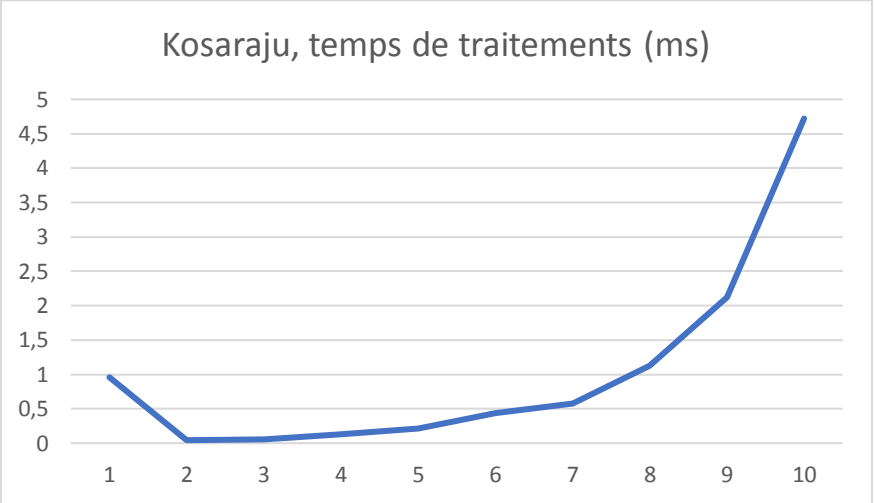
*Ici, le nombre de sommets et d'arêtes croît rapidement. Les temps de traitements augmentent en adéquation et le **StackOverFlow** est vite atteint. Les temps obtenus sont similaires et **GabowSCC** est le premier à obtenir l'exception.*

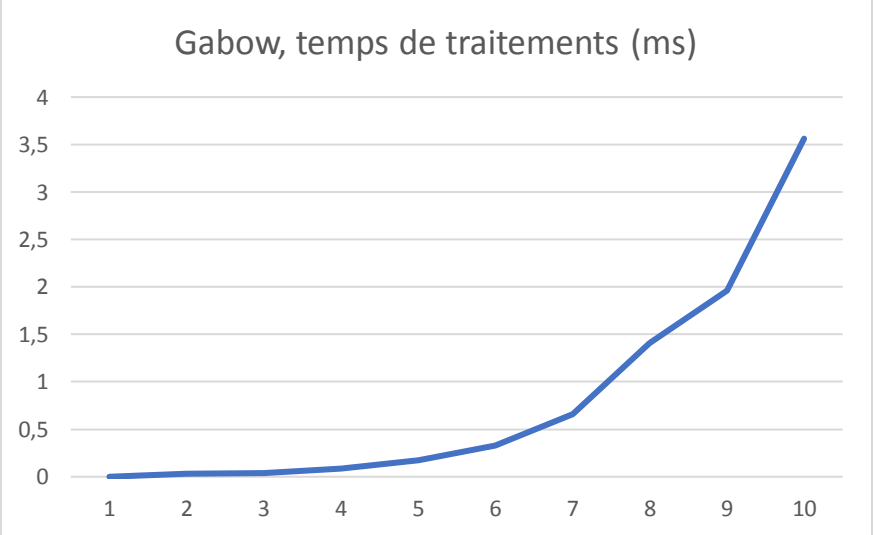
```
GraphBuilder gb = GraphBuilder.get();
Graph graph1 = gb.generate(10, 20);
Graph graph2 = gb.generate(20, 40);
Graph graph3 = gb.generate(40, 80);
Graph graph4 = gb.generate(80, 160);
Graph graph5 = gb.generate(160, 320);
Graph graph6 = gb.generate(320, 640);
Graph graph7 = gb.generate(640, 1280);
Graph graph8 = gb.generate(1280, 2560);
Graph graph9 = gb.generate(2560, 5120);
Graph graph10 = gb.generate(5120, 10240);
```





On observe une irrégularité au graphe 9 pour Tarjan





IV. Avis / conclusion

Les codes des trois algorithmes sont plutôt similaires. La difficulté résidait plus dans le choix des structures de données et leur utilisation que dans l'implémentation de l'algorithme lui-même.

Une fois **TarjanSCC** fait, les suivants furent assez rapides à implémenter.

Conceptuellement, je préfère l'algorithme de **Kosaraju**, j'aime bien l'idée saugrenue de devoir transposer le graphe.

Au niveau de l'implémentation celui que j'ai trouvé le plus facile à implémenter est **GabowSCC**, le code est court et concis.

Les algorithmes étant récursifs, en Java et dans mon implémentation, la **StackOverFlow** est vite arrivée.

Elle est généralement liée à trop de push au sein de la / des piles. On pourrait peut-être tenter de palier à ce souci précis en utilisant plus d'objets. Les performances s'en verraient cependant amoindries et l'exception apparaîtrait probablement toujours en raison du nombre d'appels à la méthode de traitement principale.

Finalement, de manière générale et dans mon implémentation, l'algorithme le plus performant semble être celui de **Tarjan** bien qu'il soit plutôt limité à cause d'un trop grand nombre d'appels récursifs. **Kosaraju** quant à lui semble être le plus solide face à une éventuelle **StackOverFlow** exception.