

Compte Rendu Algo TP6-9

Julien D'aboville - 22107316

2023

Sommaire :

Sommaire :	2
TP6 : Visualisation de Graphes	3
TP7 et TP8 : TD ACM (MST in English) et PCC Arbre couvrant minimum et plus courts chemins TP Dijkstra et A*	3
Analyse des Résultats.....	7
Comparaison des Temps d'Exécution.....	7
Nombre de Nœuds Explorés.....	7
Qualité des Chemins Trouvés.....	7
Utilisation dans des Scénarios Réels.....	7
Algorithme de Dijkstra.....	7
Algorithme A*	7
Réflexion.....	8
Conclusion.....	8
TP9 Comparaison performance	9
Conclusion :	9
Bonus :	10
Explication :	10

TP6 : Visualisation de Graphes

Lors de l'exploration des capacités de Java pour la visualisation de graphes, on constate que des bibliothèques comme JavaFX peuvent être limitées pour les grands graphes.

Neo4j, en tant que base de données orientée graphe, surpasse ces limitations en offrant des outils avancés pour la visualisation et la manipulation de graphes complexes.

Sa capacité à gérer de grands ensembles de données et son interface utilisateur intuitive facilitent l'exploration graphique.

De plus, son caractère multiplateforme permet une intégration aisée avec divers environnements de développement.

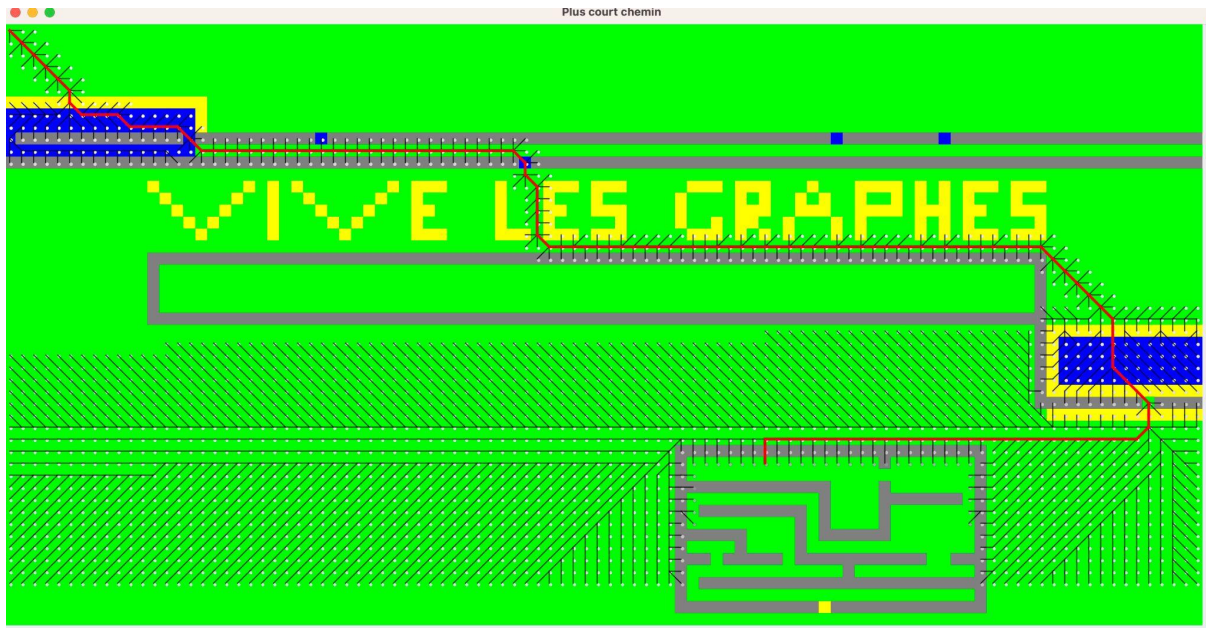
Ainsi, Neo4j se présente comme une solution robuste pour les projets nécessitant une visualisation de graphe avancée et une manipulation de données complexes.

TP7 et TP8 : TD ACM (MST in English) et PCC Arbre couvrant minimum et plus courts chemins TP Dijkstra et A*

En raison de contraintes de temps, il n'a pas été possible d'implémenter un menu interactif qui aurait permis aux utilisateurs de choisir facilement entre les heuristiques de Manhattan et Euclidienne, ainsi que les ajustements correspondants du graphe, dans la version actuelle du projet. Actuellement, il est donc seulement possible de tester avec l'heuristique Euclidienne pour l'algorithme A* ou d'utiliser l'algorithme de Dijkstra.

Néanmoins, j'ai effectué des tests en utilisant l'heuristique de Manhattan, en laissant en commentaires les parties nécessaires pour cette mise en œuvre. Ces tests incluent des images illustrant les chemins trouvés, ainsi que des détails sur le temps d'exécution et le poids des chemins.

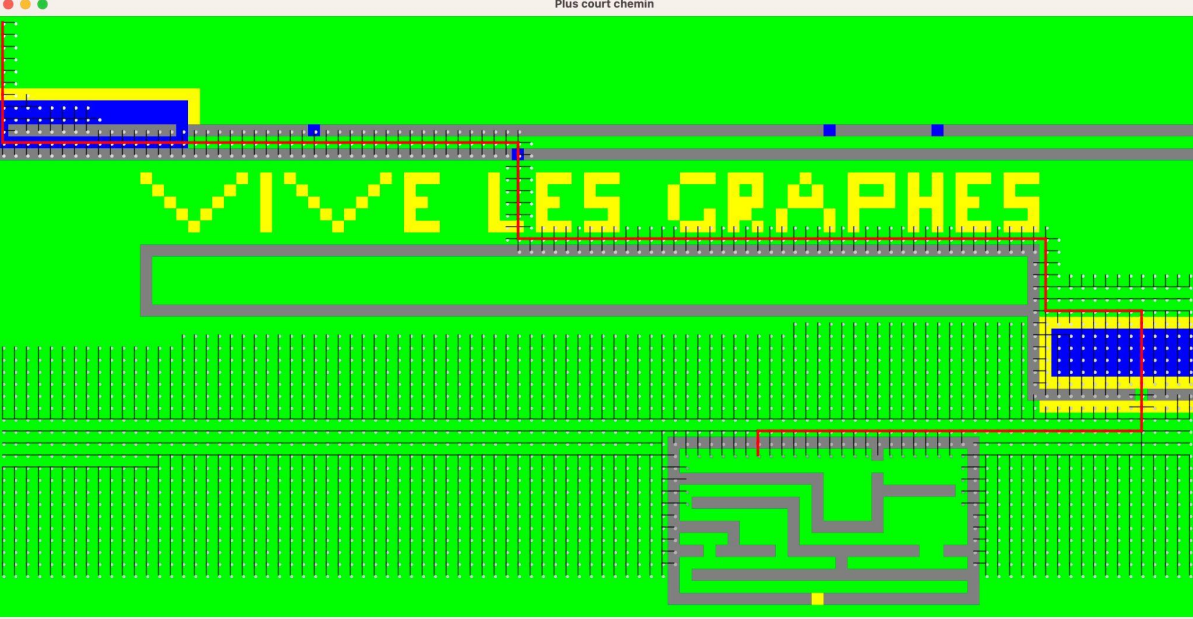
Plus court chemin avec A* (heuristique :Division Euclidienne)



Problems @ Javadoc Declaration Console X

```
<terminated> App (1) [Java Application] /Library/Java/JavaVirtualMachine
nb lignes :50
nb colonnes :ncol=100
(0) Dijkstra ou (1) A*
1
Done! Using A*:
    Number of nodes explored: 1822
    Total time of the path: 1638.8416664540925
A* - Temps d'exécution: 22.837598917 secondes
Poids total du chemin : 1638.8416664540925
```

Plus court chemin avec A* (heuristique :Manhattan)



Plus court chemin

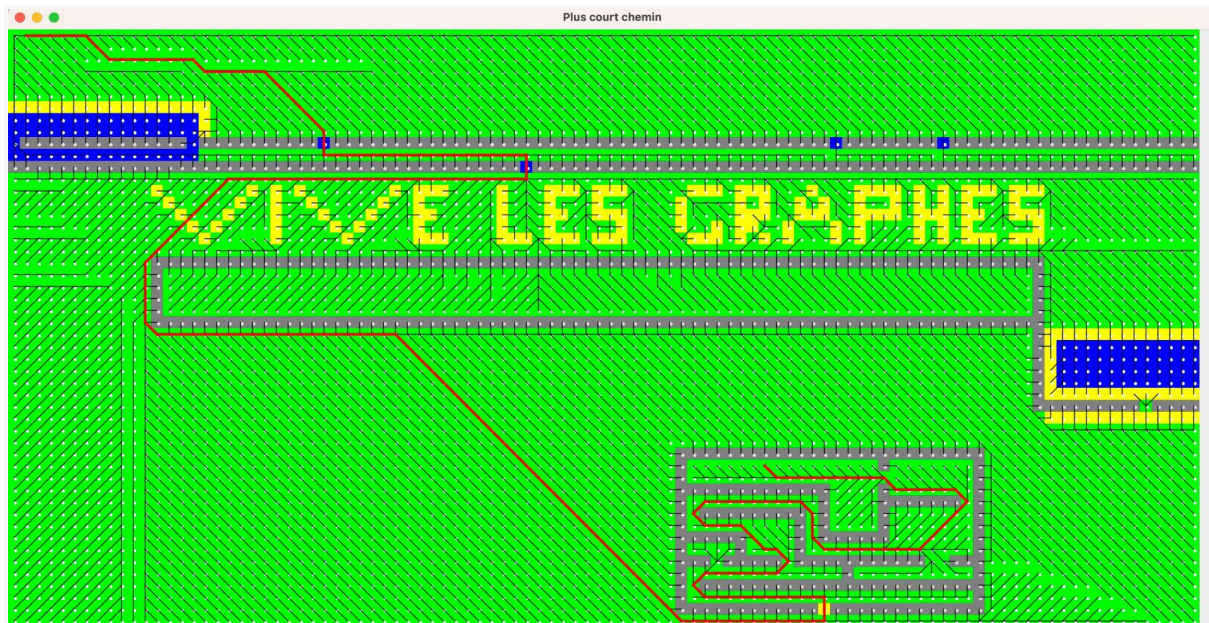
VIVE LES GRAPHES

Problems Javadoc Declaration Console X

App (1) [Java Application] /Library/Java/JavaVirtualMachines/temurin-17.

nb lignes :50
nb colonnes :ncol=100
(0) Dijkstra ou (1) A*
1
Done! Using A*:
 Number of nodes explored: 1838
 Total time of the path: 1955.0
A* – Temps d'exécution: 22.858054292 secondes
Poids total du chemin : 1955.0

Plus court chemin avec Dijkstra (avec prise en compte des mouvements diagonaux)



nb lignes :50

nb colonnes :ncol=100

(0) Dijkstra ou (1) A*

0

Done! Using Dijkstra:

Number of nodes explored: 5000

Dijkstra - Temps d'exécution: 62.294441042 secondes

Poids total du chemin : 302.6101730552669

Analyse des Résultats

Comparaison des Temps d'Exécution

- **Dijkstra** : Plus lent, avec un temps d'exécution moyen significativement plus élevé que celui de A*.
- **A (Manhattan)*** : Plus rapide, mais avec un chemin qui n'est pas toujours le plus court.
- **A (Euclidienne)*** : Également rapide, et potentiellement plus précis sur des graphes où les déplacements diagonaux sont permis.

Nombre de Nœuds Explorés

- **Dijkstra** : Explore tous les nœuds, ce qui augmente le temps de calcul.
- **A*** : Explore moins de nœuds grâce à l'heuristique qui guide la recherche, réduisant ainsi le temps d'exécution.

Qualité des Chemins Trouvés

- **Dijkstra** : Trouve toujours le chemin le plus court comme il l'explore exhaustivement, poids total du chemin trouvé était de 302.61 pour l'exemple donné.
- **A (Manhattan)*** : Peut ne pas trouver le chemin le plus court à cause de l'heuristique, poids total du chemin trouvé était de 1955 pour l'exemple donné.
- **A (Euclidienne)*** : A une chance plus élevée de trouver le chemin le plus court sur des graphes avec des mouvements diagonaux, poids total du chemin trouvé était de 1638.84 pour l'exemple donné.

Utilisation dans des Scénarios Réels

Algorithme de Dijkstra

- **Réseaux de Télécommunications ou de Distribution d'Électricité** : Où le chemin le plus court et le plus fiable est crucial.
- **Graphes de Petite à Moyenne Taille** : Où l'efficacité computationnelle est gérable.

Algorithme A*

- **Grands Graphes avec Coûts Hétérogènes** : Comme dans les jeux vidéo ou la robotique, où des décisions rapides sont nécessaires.
- **Simulation et Jeux** : Où des chemins doivent être trouvés en temps réel dans des environnements complexes.

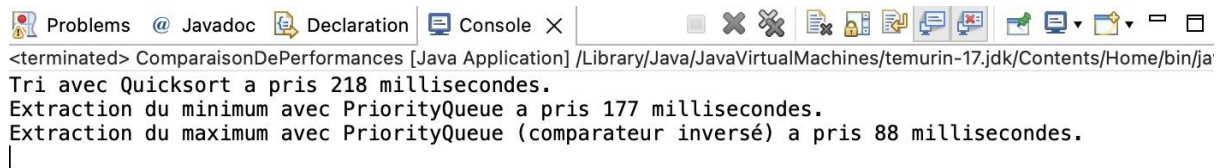
Réflexion

- **Heuristique** : L'heuristique joue un rôle déterminant dans l'efficacité de l'algorithme A*. La distance de Manhattan convient mieux aux grilles restreintes pour des mouvements horizontaux et verticaux, tandis que l'heuristique euclidienne est adaptée aux grilles permettant des mouvements dans toutes les directions.
- **Équilibre entre Vitesse et Précision** : L'équilibre entre la vitesse de recherche et la précision du chemin trouvé est essentiel. A* peut être optimisé avec une heuristique adéquate pour trouver rapidement le plus court chemin possible, tandis que Dijkstra reste le choix le plus fiable lorsque la précision est incontournable.

Conclusion

La sélection entre Dijkstra et A* doit être faite en considérant à la fois la taille du graphe et l'importance de la précision du chemin par rapport à la vitesse de calcul. A* est préférable pour les applications en temps réel où une réponse rapide est nécessaire, tandis que Dijkstra est idéal lorsque le chemin le plus court est la priorité absolue, sans contraintes de temps d'exécution.

TP9 Comparaison performance



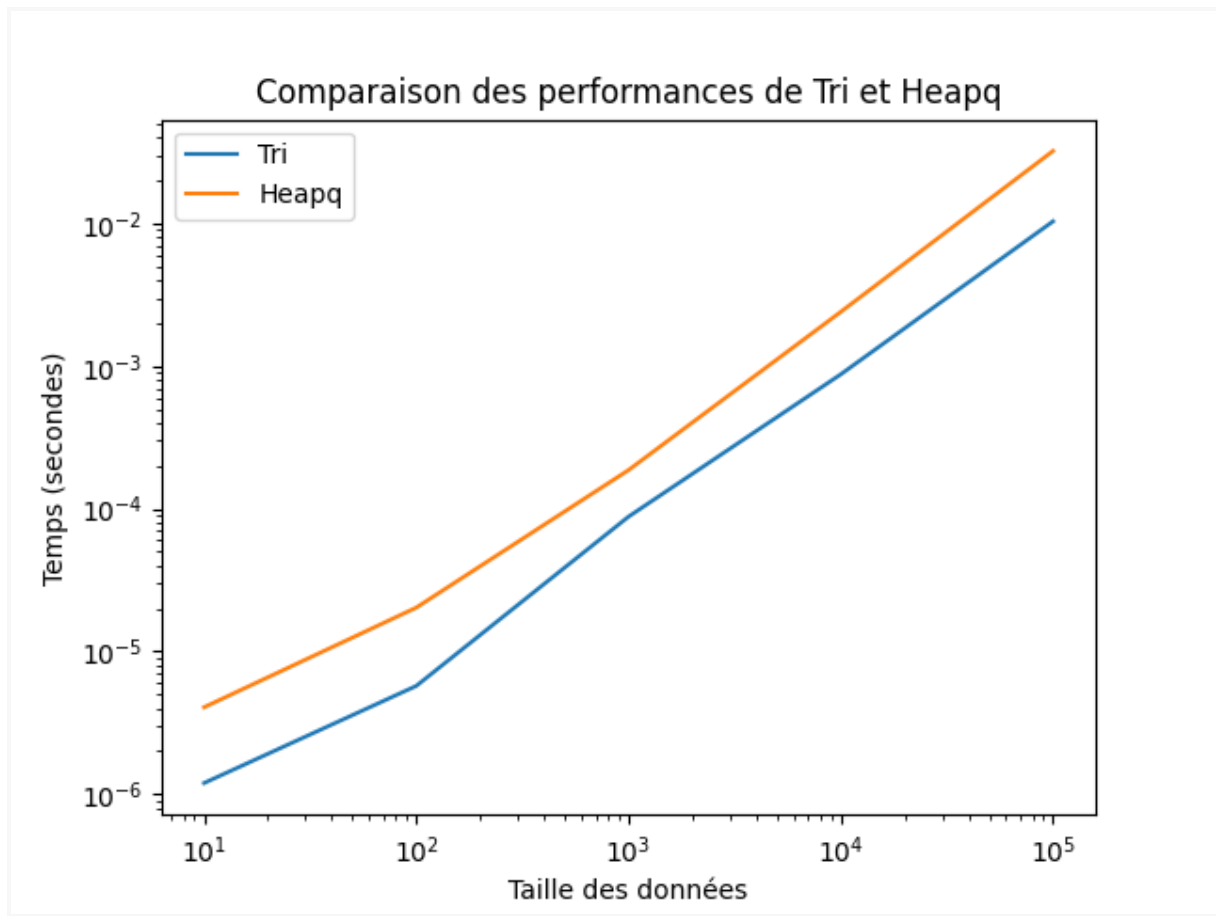
The screenshot shows an IDE interface with a 'Console' tab selected. The console output displays the results of a Java application named 'ComparaisonDePerformances'. The output text is as follows:

```
<terminated> ComparaisonDePerformances [Java Application] /Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home/bin/ja  
Tri avec Quicksort a pris 218 millisecondes.  
Extraction du minimum avec PriorityQueue a pris 177 millisecondes.  
Extraction du maximum avec PriorityQueue (comparateur inversé) a pris 88 millisecondes.  
|
```

Conclusion :

Suite à de multiples exécutions de l'algorithme, il est clairement observable que l'utilisation de `PriorityQueue` pour les opérations de tri et d'extraction de minimum et maximum est nettement plus efficace que l'utilisation du tri Quicksort sur un ensemble de nombres aléatoires.

Bonus :



Explication :

Les résultats montrent que l'utilisation de tas (heapq) est systématiquement plus rapide que le tri standard (sorted) pour toutes les tailles de données testées.

Cette tendance est constante même lorsque la taille des données augmente, indiquant une efficacité supérieure de heapq dans la manipulation des ensembles de données.