Rapport AG51

Etienne Gartner

14 janvier 2018

Table des matières

1 | TP1 : Algorithmes de Tri

1.1 Objectifs du TP

L'objectif de ce premier TP est d'implémenter plusieurs algorithmes de Tri (tri par insertion, tri par sélection, tri par permutation, tri rapide, tri shell, tri fusion et tri par tas), puis de comparer leurs temps d'exécution sur des tableaux de tailles différentes. Le but est de confronter leur complexité théorique et leur coût d'exécution.

1.2 Mesures et Comparaisons

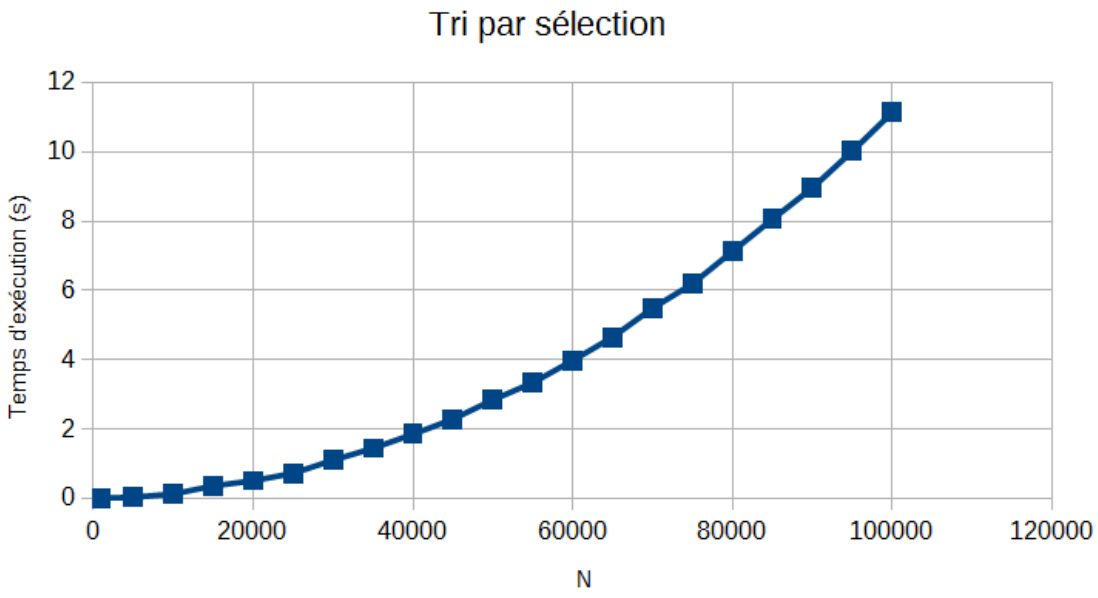

FIGURE1.1 – Coût expérimental de l'algorithme du tri par sélection

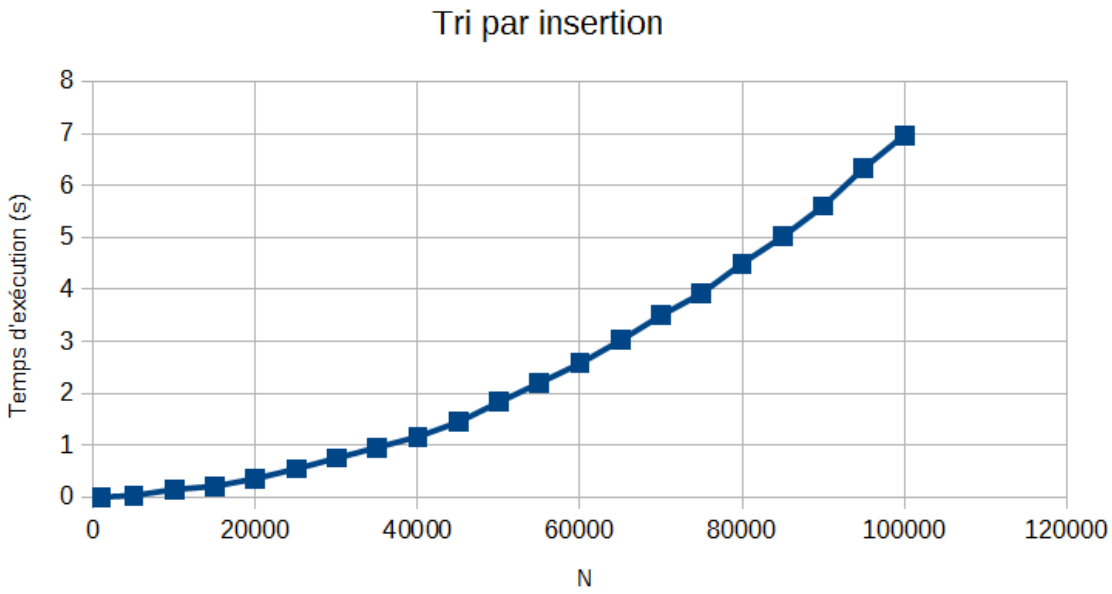

FIGURE1.2 – Coût expérimental de l'algorithme du tri par insertion

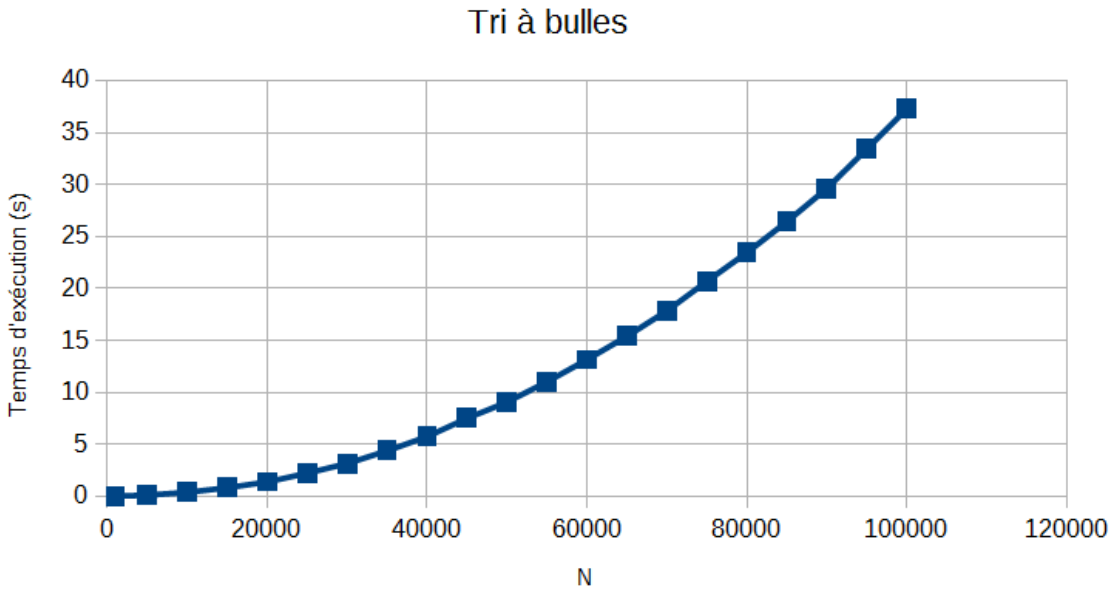

FIGURE1.3 – Coût expérimental de l'algorithme du tri à bulles

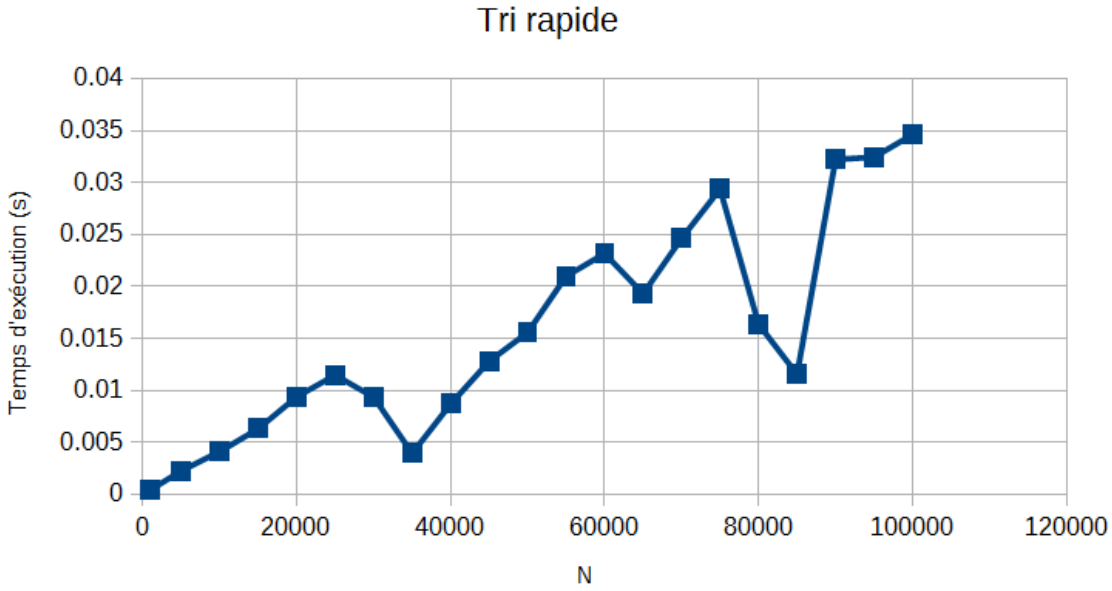

FIGURE1.4 – Coût expérimental de l'algorithme du tri rapide

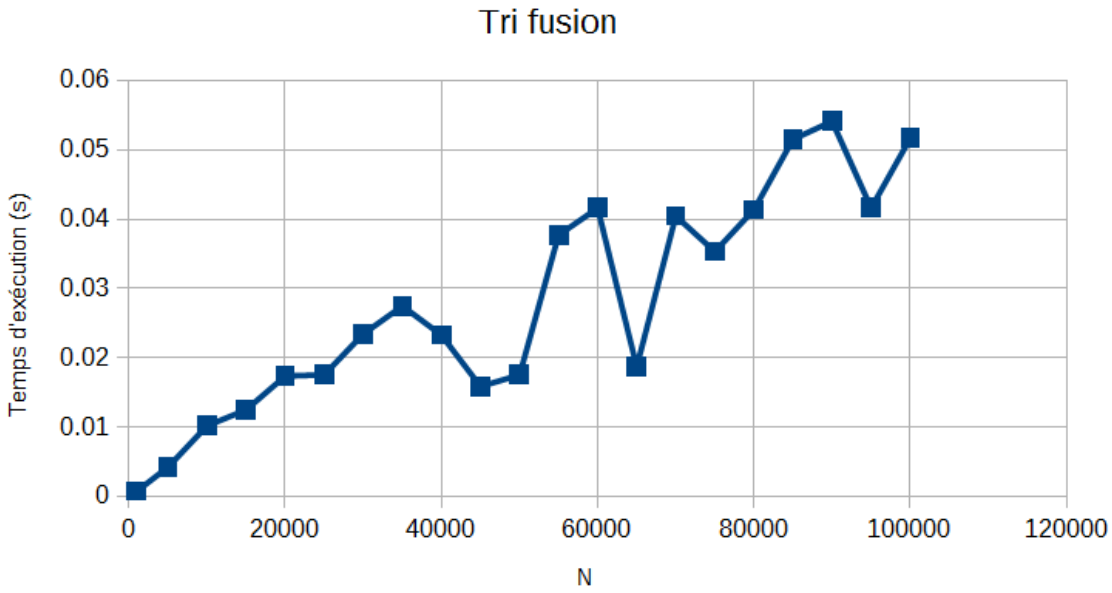

FIGURE1.5 – Coût expérimental de l'algorithme du tri fusion

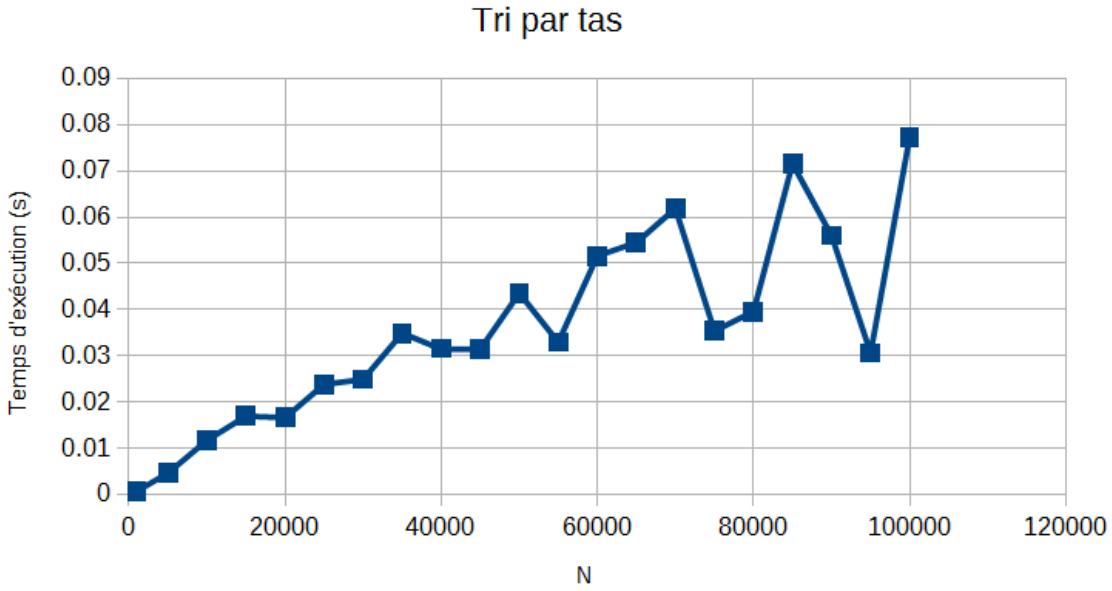

FIGURE1.6 – Coût expérimental de l'algorithme du tri par tas

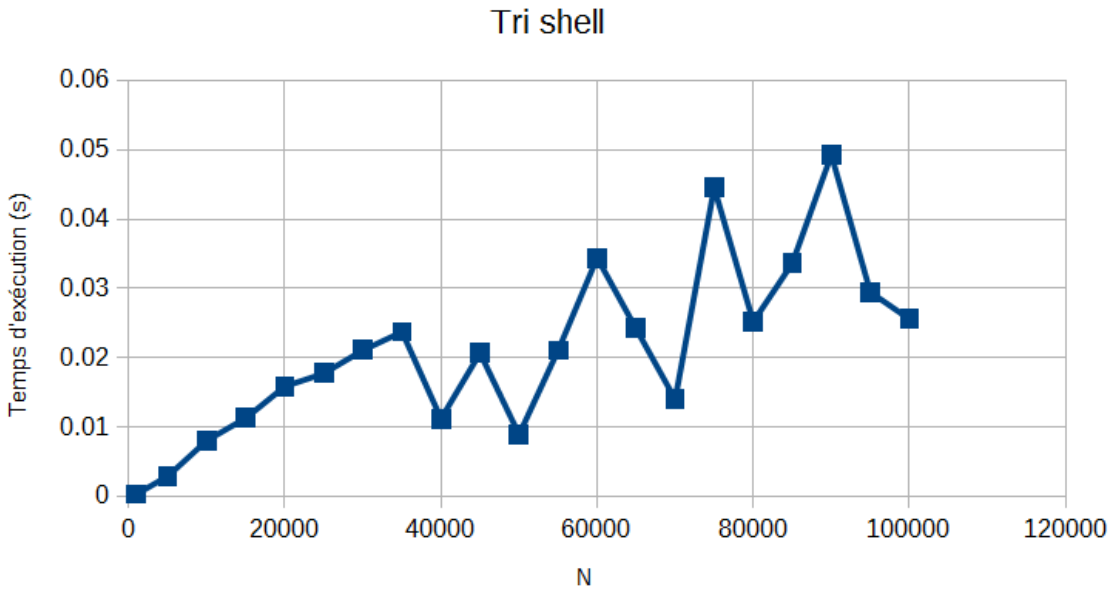

FIGURE1.7 – Coût expérimental de l'algorithme du tri shell

1.3 Conclusion

On remarque que les tris par insertion, sélection et à bulle ont une courbe qui s'appparrente à une fonction quadratique. En réalité, ce résultat est tout à fait cohérent avec la complexité de ces algorithmes qui est de O(n^2). Dans ces trois algorithmes, le pire cas a une complexité similaire au cas moyen. Le meilleur cas est O(n) et il insinue que le tableau est déjà intégralement trié : en règle général ce cas a peu d'importance (tout comme le meilleur cas), notamment pour les algorithmes de tri.

Les autres tris quant à eux présentent un coût expérimental plus proche de Nlog(N). C'est en effet

le cas pour la complexité théorique du tri rapide, du tri par tas et du tri fusion qui partagent une complexité moyenne de O(Nlog(N)) dans le cas moyen et le pire cas. En ce qui concerne le tri Shell, son coût théorique moyen est de $O(Nlog(N)^2)$.

On explique l'irrégularité des courbes par la disposition des éléments dans le tableau initial : en effet certains éléments peuvent déjà être triés, ce qui réduit le temps d'exécution de l'algorithme. Pour obtenir des courbes plus exactes, il faudrait effectuer un plus grand nombre de mesures et en faire la moyenne.

Pour finir, on remarque qu'expérimentalement, les algorithmes de complexité logarithmique sont bien plus efficaces que ceux de complexité quadratique. Par exemple, pour une instance où N = 100000, les algorithmes en $O(n^2)$ donnent un résultats sous plusieurs secondes voire plusieurs dizaines de secondes ; les algorithmes en O(Nlog(N)) quant à eux donnent un résultats en quelques dizaines de millisecondes.

2 | TP2 : Tas Binaire et File de Priorité

2.1 Objectifs du TP

L'objectif de ce TP est d'implémenter la structure de données du tas binaire. Cette implémentation servira de base pour mettre en oeuvre une file de priorité. Pour finir, nous comparerons les performances de la file de priorité basée sur un tas binaire face à celles d'une file de priorité basée sur un tableau linéaire.

2.2 Mesures et Comparaisons

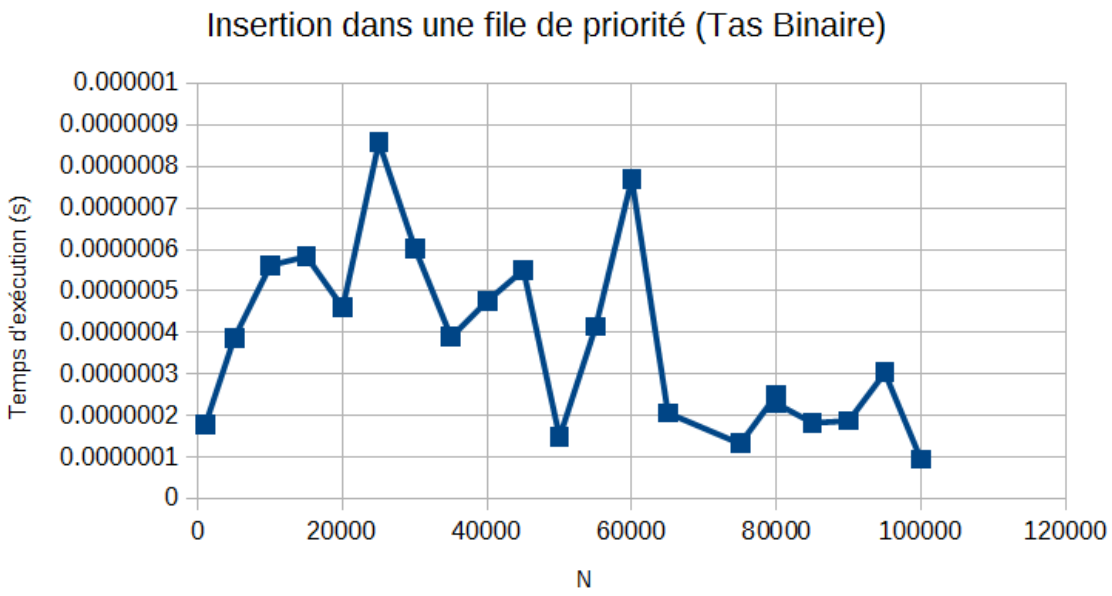

FIGURE2.1 – Coût expérimental d'une insertion dans une file de priorité implémentée à partir d'un tas

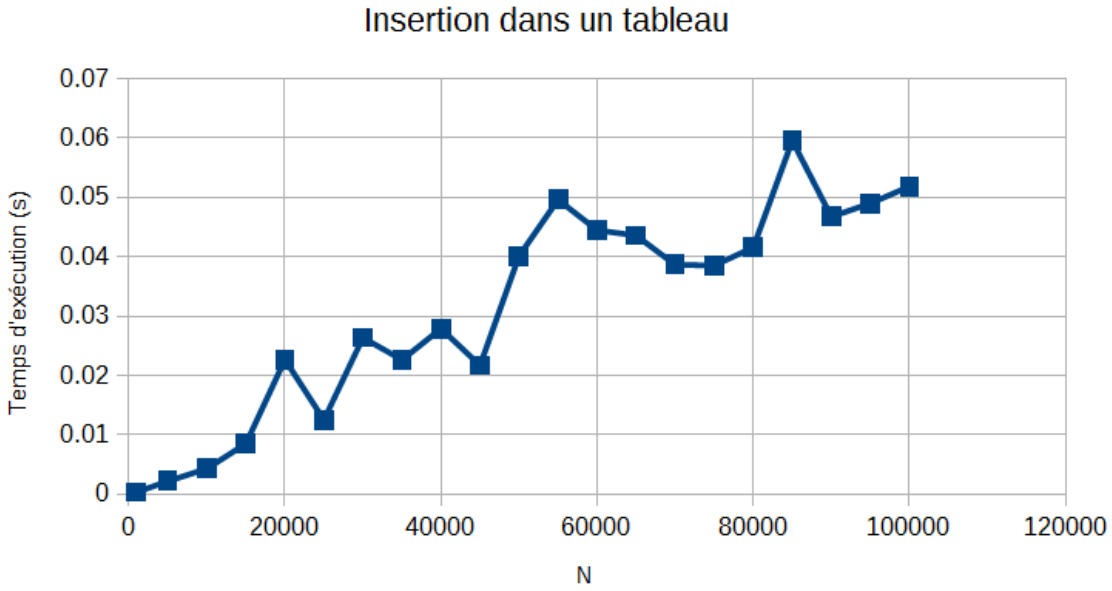

FIGURE2.2 – Coût expérimental d'une insertion dans une file de priorité implémentée à partir d'un tableau

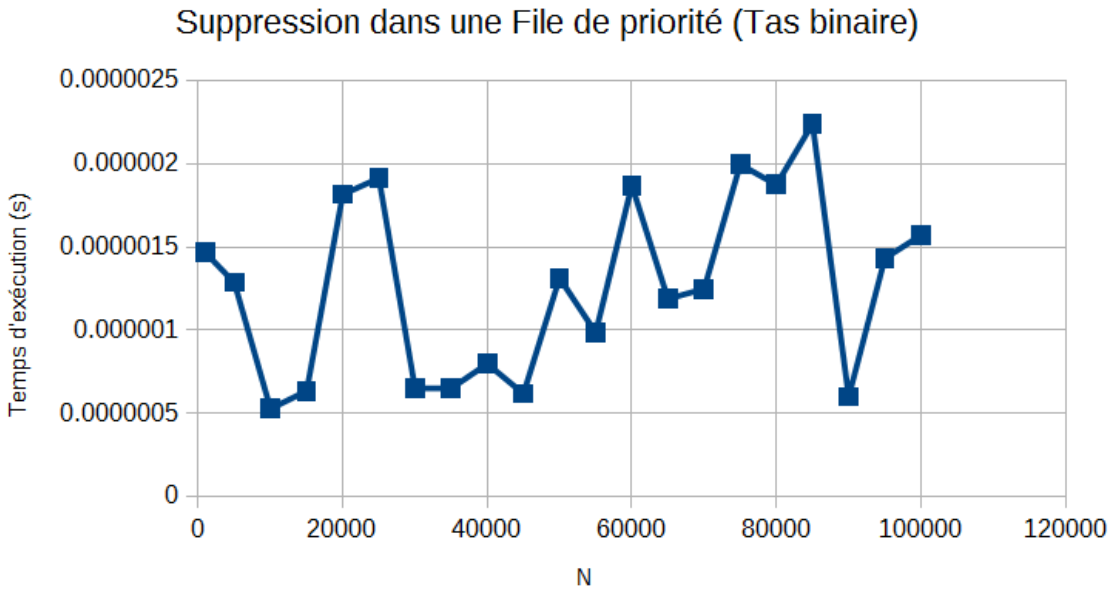

FIGURE2.3 – Coût expérimental d'une suppression dans une file de priorité implémentée à partir d'un tas

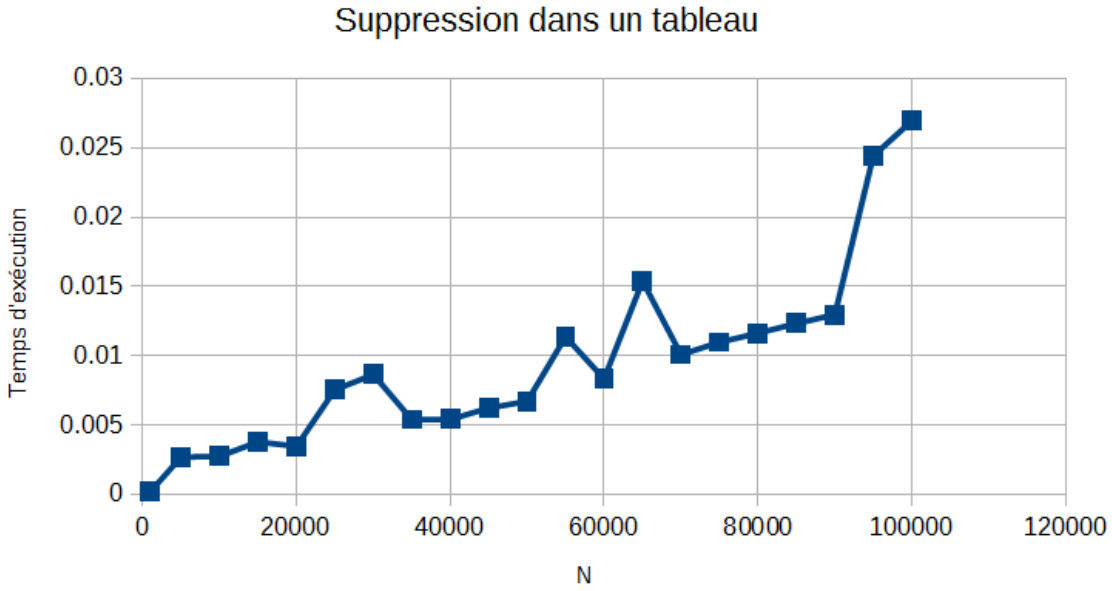

FIGURE2.4 – Coût expérimental d'une suppression dans une file de priorité implémentée à partir d'un tableau

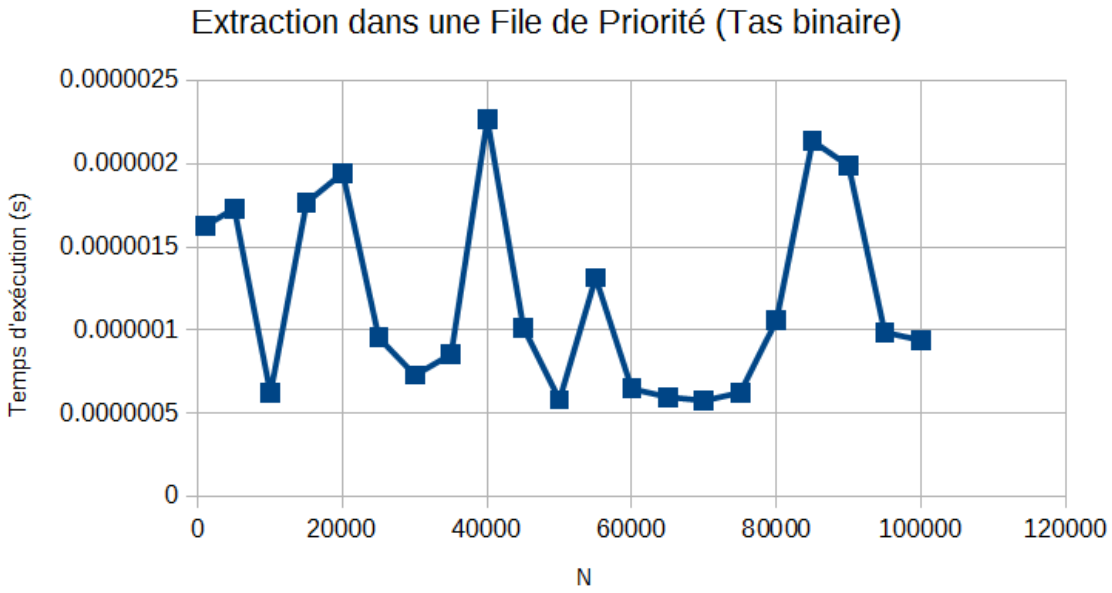

FIGURE2.5 – Coût expérimental d'un accès à l'élément minimum dans une file de priorité implémentée à partir d'un tas

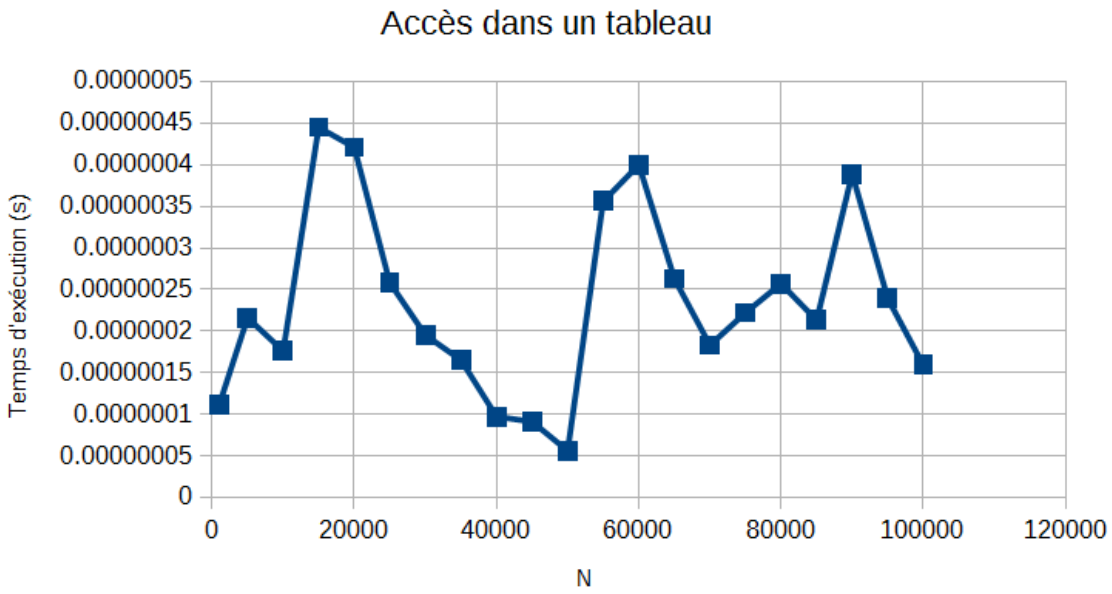

FIGURE2.6 – Coût expérimental d'un accès à l'élément minimum dans une file de priorité implémentée à partir d'un tas

2.3 Conclusion

Pour ce TP, j'ai choisi d'implémenter un tas binaire minimal (en réalité, il y a peu de différences entre un tas min et un tas max et les deux peuvent être utilisés pour implémenter une file de priorité selon la stratégie désirée). J'ai ensuite comparé ses performances à celles d'un tableau trié.

Sur les mesures ci-dessus, on remarque que le tas binaire offre des temps quasimment constant pour les trois opérations testées. En théorie, la complexité est constante pour accèder à l'élément minimum (il s'agît d'accèder à la racine), mais elle est de O(log(N)) pour l'insertion et la suppression.

En ce qui concerne l'implémentation à partir d'un tableau trié, son coût expérimental semble être O(n) pour l'insertion et la suppression, et un temps constant O(1) pour l'accès à l'élément minimum. Cela est donc cohérent avec les coûts théoriques. En effet, pour insérer ou supprimer un élément, il faut parcourir tout le tableau jusqu'à la position de l'élément.

Ci-dessous, un tableau récapitulatif des coûts des opérations d'une file de priorité selon la structure de données utilisée :

	Insertion	Suppression	Accès à l'élément min
Tableau trié	O(n)	O(n)	O(1)
Tableau non trié	O(1)	O(1)	O(n)
Tas binaire	O(log(N))	O(log(N))	O(1)

3 | TP3 : Tables de Hachage

3.1 Objectifs du TP

L'objectif de ce TP est d'implémenter une fonction de hachage, puis de comparer ses coûts expérimentaux et théorique face à une structure de donnée linéaire de stockage ainsi qu'à la structure de donnée prédéfinie du language utilisé. Pour finir, il faudra contrôler l'accès à cette table lorsqu'elle est utilisée par des threads concurrents.

3.2 Mesures et Comparaisons

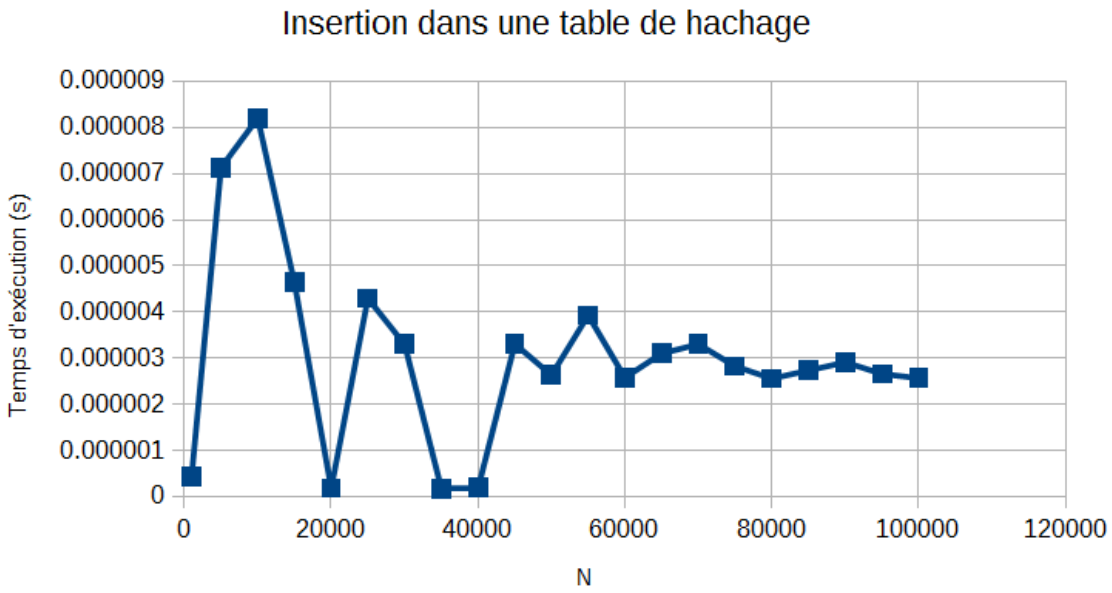

FIGURE3.1 – Coût expérimental d'une insertion dans la table de hachage

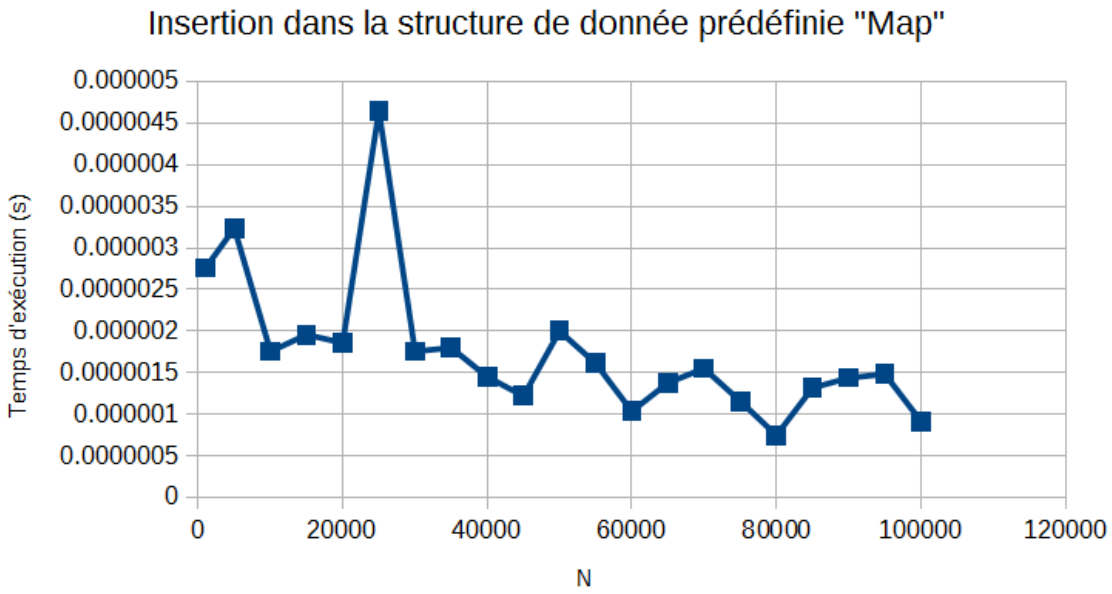

FIGURE3.2 – Coût expérimental d'une insertion dans la structure de donnée prédéfinie en C++

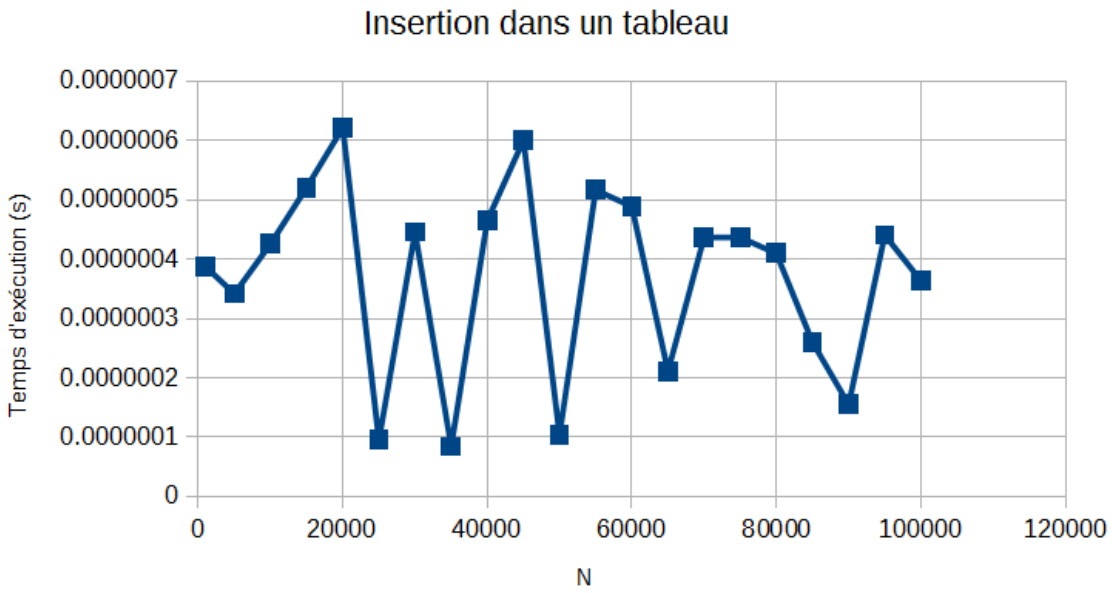

FIGURE3.3 – Coût expérimental d'une insertion dans un tableau

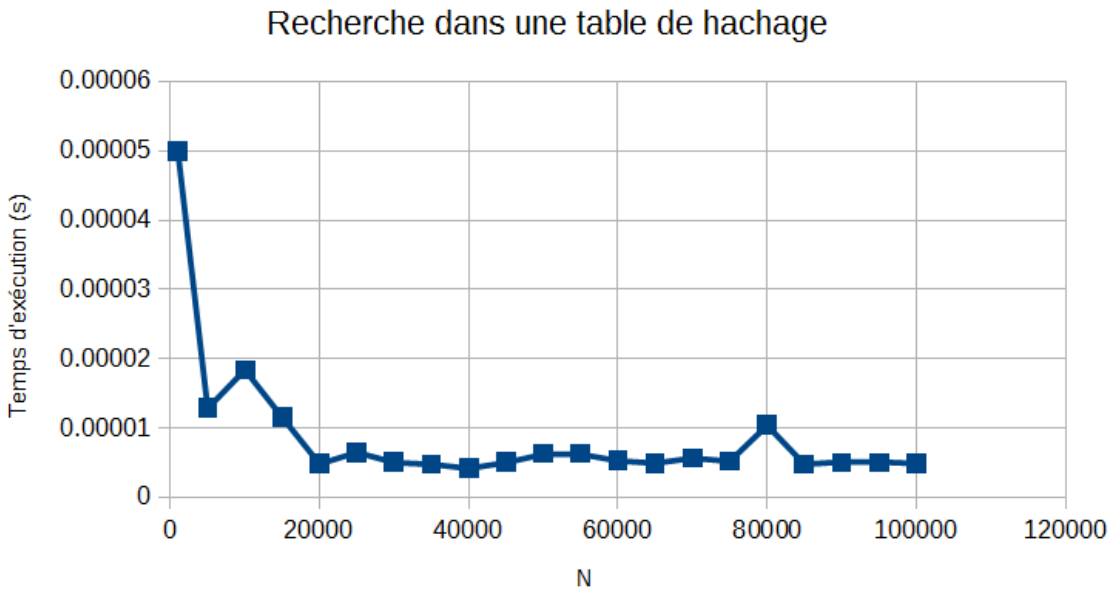

FIGURE3.4 – Coût expérimental d'une recherche dans une table de hachage

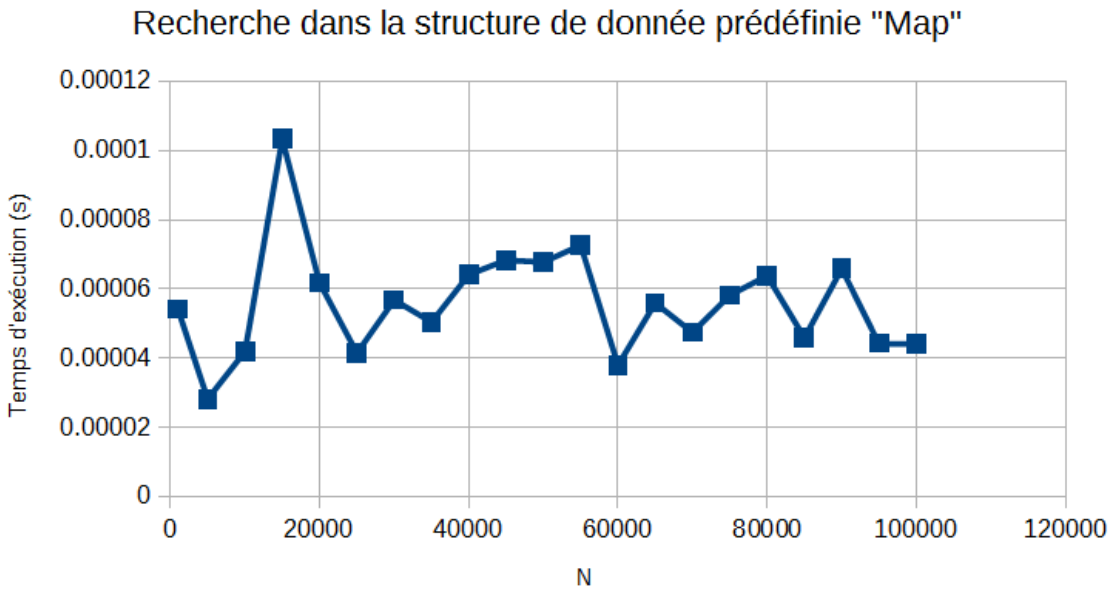

FIGURE3.5 – Coût expérimental d'une recherche dans la structure de donnée prédéfinie en C++

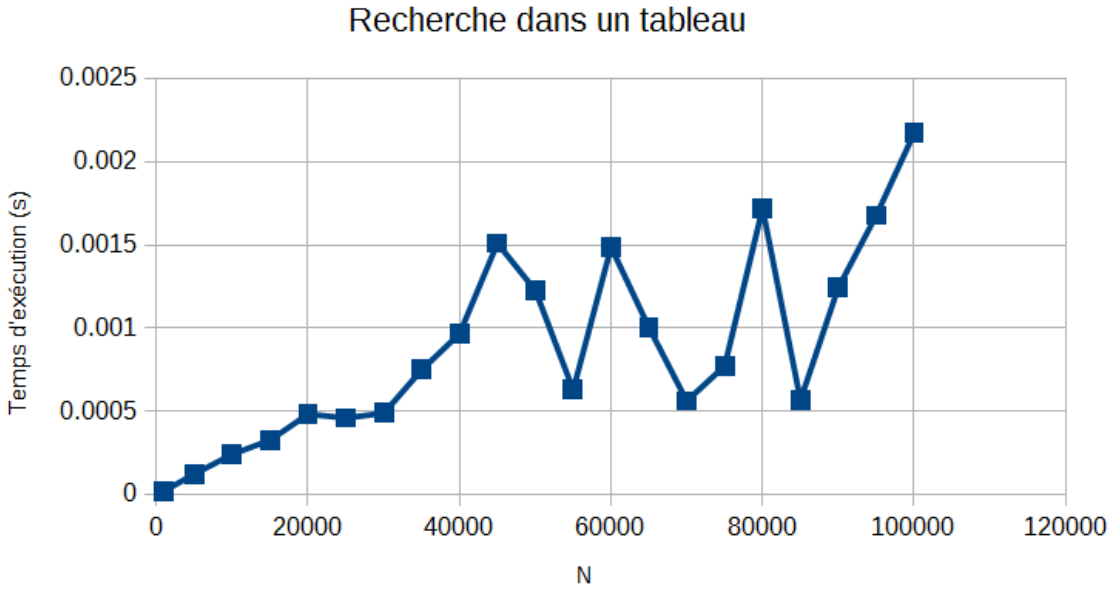

FIGURE3.6 – Coût expérimental d'une recherche dans un tableau

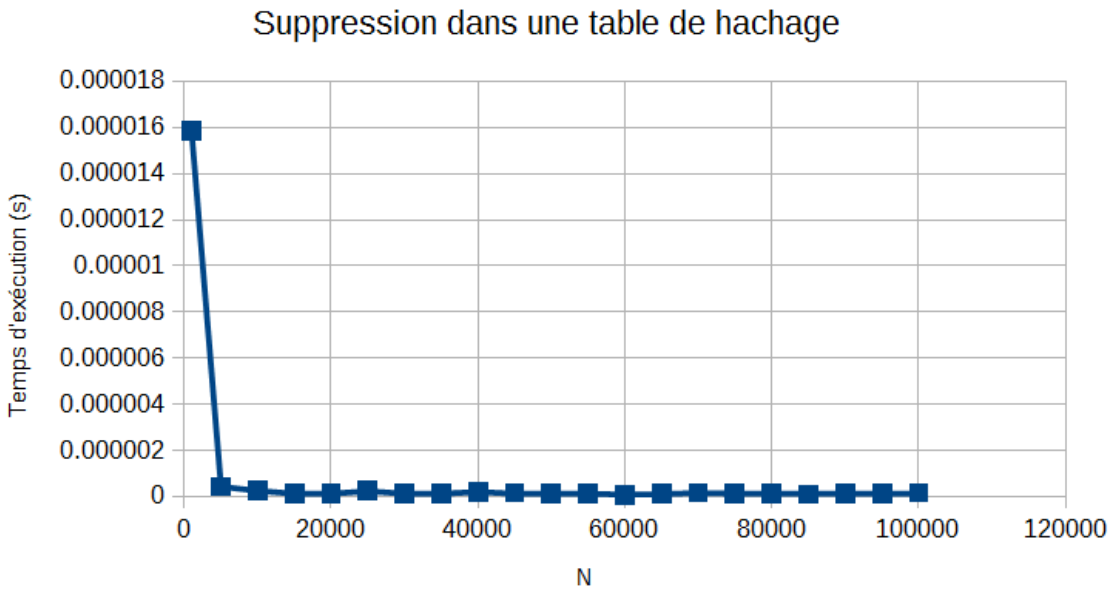

FIGURE3.7 – Coût expérimental d'une suppression dans une table de hachage

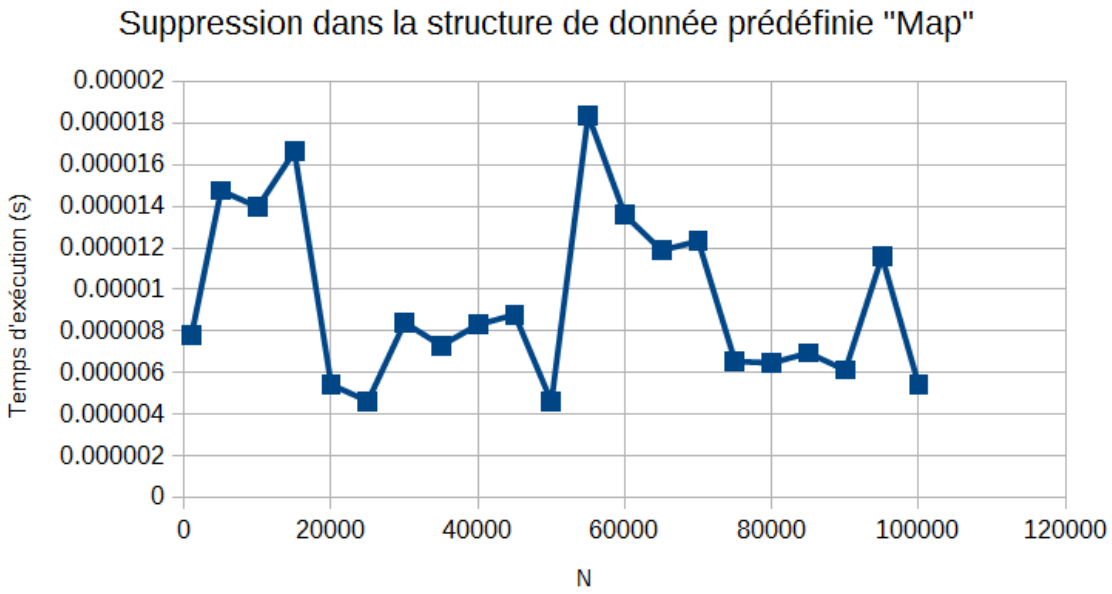

FIGURE3.8 – Coût expérimental d'une suppression dans la structure de donnée prédéfinie en C++

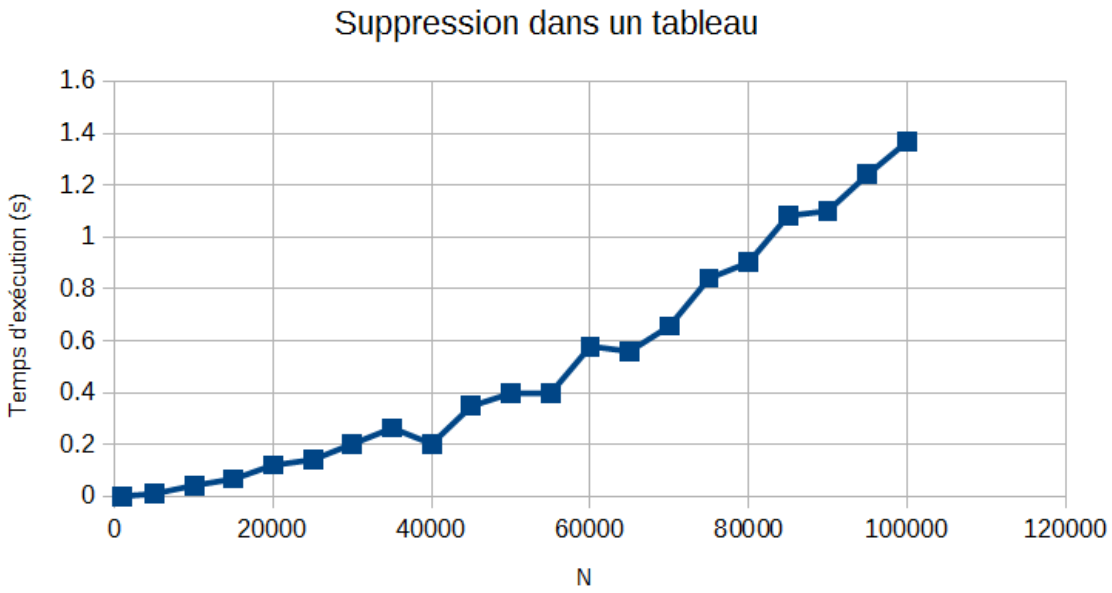

FIGURE3.9 – Coût expérimental d'une suppression dans un tableau

3.3 Conclusion

J'ai décidé d'implémenter une table de hachage avec adressage ouvert et sondage quadratique. Les coûts théoriques d'une table de hachage avec adressage ouvert sont les suivants :

	Insertion	Recherche	Suppression
Opération fructueuse	$\frac{1}{\alpha}ln(\frac{1}{1-\alpha})$	$\frac{1}{\alpha}ln(\frac{1}{1-\alpha})$	$\frac{1}{\alpha}ln(\frac{1}{1-\alpha})$
Opération infructueuse	$\frac{1}{(1-\alpha)}$	$\frac{1}{(1-\alpha)}$	$\frac{1}{(1-\alpha)}$

Les mesures ont été réalisé sur des opérations fructueuses. Elles sont donc cohérentes avec les coûts théoriques car on observe un temps d'exécution relativement constant.

J'ai ensuite comparé les performances de la table de hachage face à celles d'un tableau et d'une structure de donnée de la bibliothèque standard C++. En ce qui concerne le tableau, les temps d'exécutions observés sont cohérents avec les coûts théoriques. En effet, la recherche et la suppression ont une complexité de O(n) car l'algorithme parcours tout le tableau jusqu'à trouver la bonne position. En revanche, l'insertion est O(1) car le tableau a été implémenté à l'aide d'un "Vector" de la librairie standard C++ qui offre un accès constant au dernier élément.

Pour la structure de donnée de la bibliothèque standard, j'ai choisi "unordered map". En regardant ses coûts théoriques, on peut deviner qu'il s'agît d'une table de hachage car elle présente des coûts d'exécution constants. En effet, cette structure de donnée utilise également un adressage ouvert.

Pour conclure, les tables de hachages constituent en pratique un moyen bien plus efficace de stocker l'information que les tableaux.

4 | TP4 : Algorithmes pour les Graphes

4.1 Objectifs du TP

L'objectif de ce TP est d'implémenter les principaux algorithmes pour les graphes. Dans un premier temps, on implémentera les algorithmes de parcours (Depth First Search et Breadth First Search) afin de comparer leurs coûts expérimentaux. Dans une deuxième partie, on implémentera différentes version de l'algorithme de construction d'un arbre de recouvrement minimal pour également comparer leurs coûts expérimentaux.

4.2 Mesures et Comparaisons

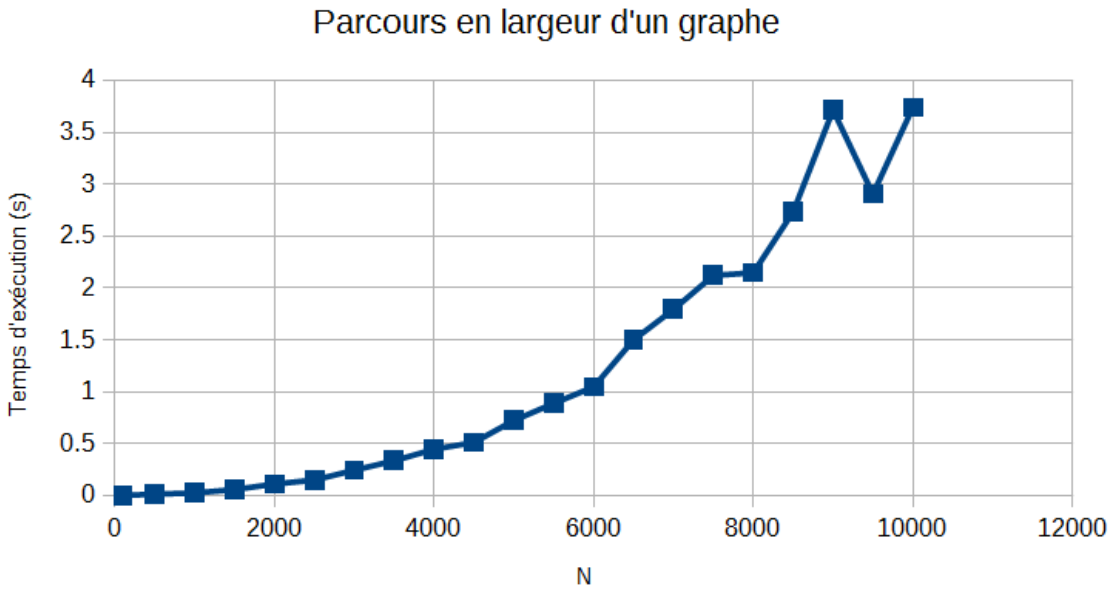

FIGURE4.1 – Coût expérimental de l'algorithme Breadth First Search

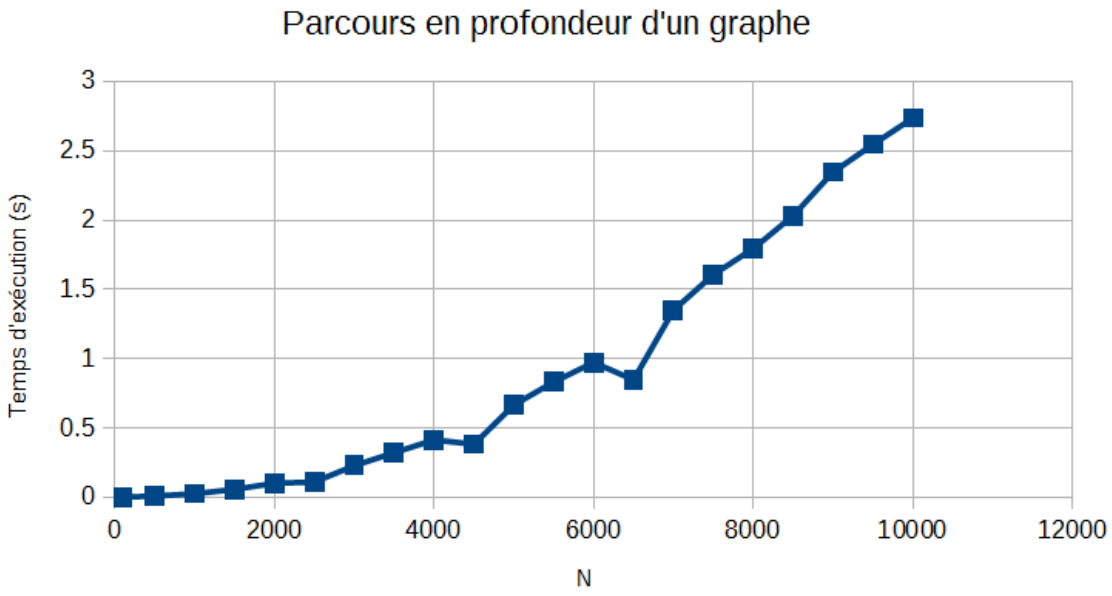

FIGURE4.2 – Coût expérimental de l'algorithme Depth First Search

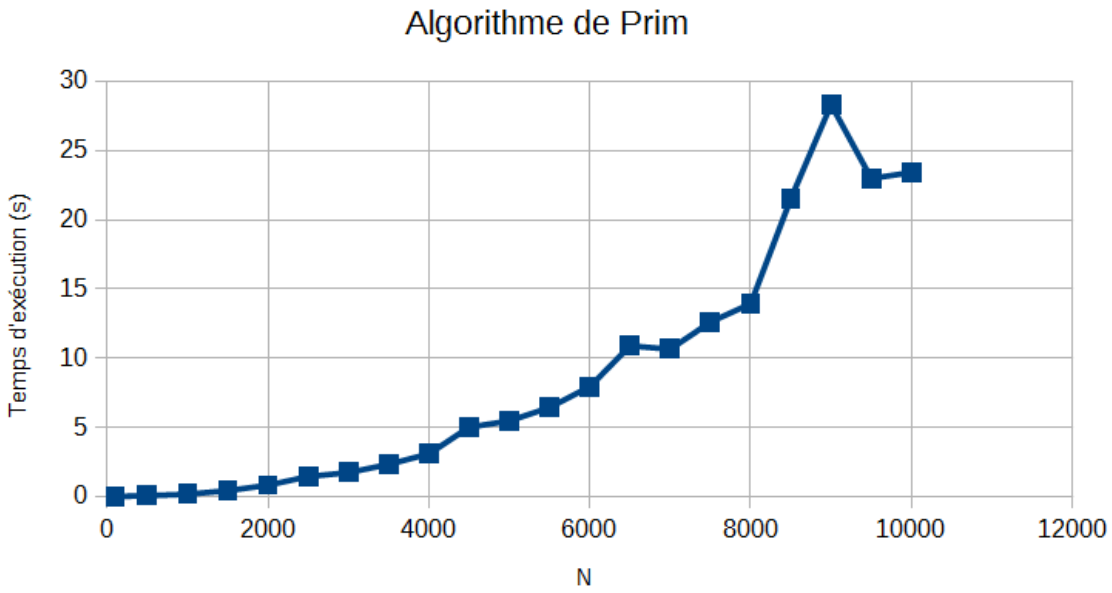

FIGURE4.3 – Coût expérimental de l'algorithme de Prim

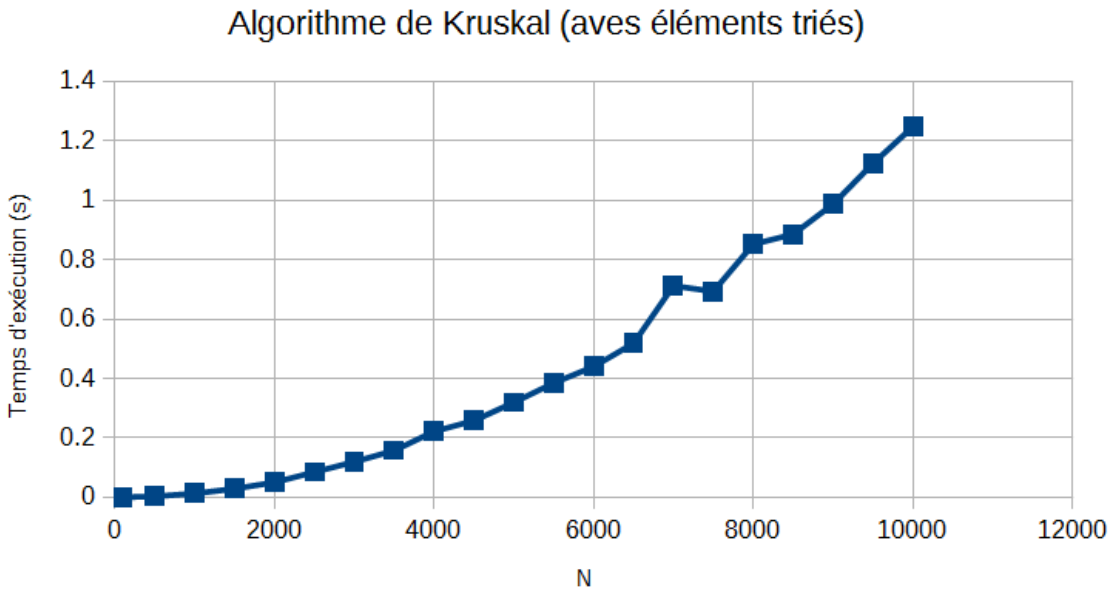

FIGURE4.4 – Coût expérimental de l'algorithme de Kruskal sans le tri des arrêtes

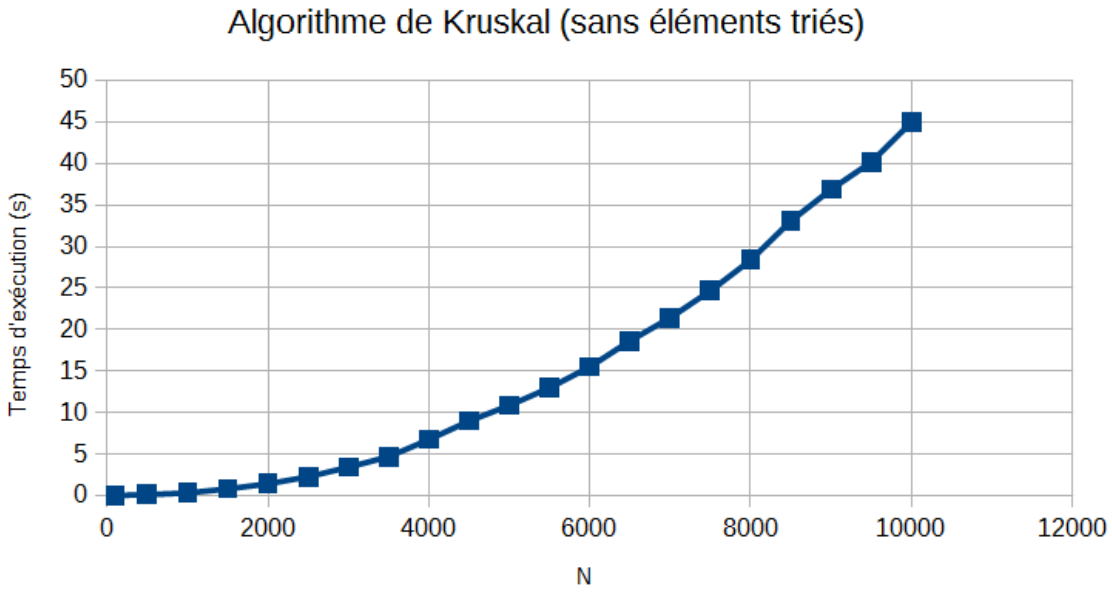

FIGURE4.5 – Coût expérimental de l'algorithme de Kruskal avec le tri des arrêtes

4.3 Conclusion

4.3.1 Algorithmes de parcours de graphes

L'algorithme BFS (Breadth First Search) est l'algorithme de parcours en largeur d'un graphe. Il s'agît d'explorer tous les voisins directs d'un sommet, puis, d'explorer leurs voisins à leurs tour, etc. Soit un graphe possédant N sommets et M arrêtes, alors la complexité de l'algorithme BFS est de O(N + M) : il s'agît d'un temps linéaire, ce qui est cohérent avec les mesures obtenues.

L'algorithme DFS (Depth First Search) est l'algorithme de parcours en profondeur d'un graphe. Il

s'agît d'explorer un voisin d'un sommet, puis d'explorer le plus loin possible en parcourant le graphe voisin par voisin. Au bout de ce chemin, l'algorithme va rencontrer un sommet qui n'aura plus de voisins explorables (il est interdit de réexplorer des sommets déjà connus) : à ce moment, il reviendra sur ses pas jusqu'à trouver un sommet explorable. La complexité de cet algorithme donné N sommets et M arrêtes est également O(N + M). En effet, les mesures effectués pour le DFS sont très similaires à celles obtenues sur le BFS.

4.3.2 Algorithmes de recouvrement minimal

Un arbre de recouvrement minimal (ou Minimum Spanning Tree), est un arbre contenant chacun des sommets d'un graphe. Sa particularité est que la somme des poids de ses arcs est minimale. Il existe de nombreux algorithmes pour construire de tels arbres. J'ai choisi d'implémenter deux versions populaires : l'algorithme de Prim et celui de Kruskal.

L'algorithme de Kruskal se base sur des ensembles disjoints. Au départ, chaque sommet représente un ensemble, c'est à dire un arbre indépendant. Ensuite, les arêtes sont triés selon leurs poids, puis sélectionnées de manière croissante. A chaque fois qu'une arête est sélectionnée, l'algorithme vérifie si les deux sommets qu'elle relie appartiennent déjà à un même arbre. Si c'est le cas, on ne peut pas sélectionner l'arrête au risque de créer un cycle. Dans le cas contraire, on ajoute l'arête aux ensembles disjoints : on fusionne les arbres contenant les deux sommets reliés.

La complexité de l'algorithme de Kruskal dépend de la structure de donnée utilisée pour implémenter les ensembles disjoints et du tri des arrêtes. Pour implémenter les sets disjoints, j'ai utilisé des listes chaînées ainsi que les heuristiques d'union par rang et de compression de chemin (comme proposé dans le livre "Introductions to Algorithms" par Thomas H. Cormen et al.). Ces heuristiques ont pour but d'améliorer l'efficacité de la structure de donnée et permettent un coût théorique linéaire : soit une séquence de M opérations de création d'ensemble, d'union et de recherche d'ensemble dont N opérations sont la création d'ensemble, alors la complexité dans le pire des cas est de O(M α (N)) (ici, α est inférieur à 4).

Soit V le nombre de sommets et E le nombre d'arrêtes. L'algorithme de Kruskal effectue E fois les opérations de recherche d'ensemble et d'union et V fois l'opération de création d'ensemble. Cela résulte en une complexité de $O((E+V)\alpha(V))$. Comme $E > V-1$ et que $\alpha(V) = O(logV) = O(logE)$, alors on peut définir le coût de l'algorithme de Kruskal en tant que O(E log V). A cela, on rajoute la complexité de l'algorithme de tri utilisé : dans notre cas, il s'agît du tri proposé par la bibliothèque standard de C++ qui a une complexité de O(N log N).

L'algorithme de Kruskal a donc une comlpexité linéaire, ce qui est cohérent avec les mesures obtenues ci-dessus.

Le second algorithme implémenté est l'algorithme de Prim. Il commence à partir d'un sommet choisi de manière aléatoire, puis ajoute à l'arbre l'arrête minimale qui relie un sommet existant de l'arbre et un sommet non existant. Dans l'implémentation, on utilise une file de priorité pour choisir les arrêtes à ajouter. La complexité de l'algorithme dépend donc de la structure de donnée utilisée pour cette file de priorité. J'utilise la file de priorité proposée par la librairie standard C++ qui est implémentée à partir d'un tas binaire. Cela signifie qu'une opération d'extraction a pour complexité O(log(V)) tout comme l'opération de réduction d'une clé. Comme l'opération d'extraction est appellée V fois et que la réduction est appellée O(E) fois, alors la complexité de l'algorithme de Prim est de O(V log(V) + E log(V)) = O(E log V).

Cette complexité est donc linéaire et est similaire à celle de l'algorithme de Kruskal. Ceci est également cohérent avec les résultats expérimentaux.