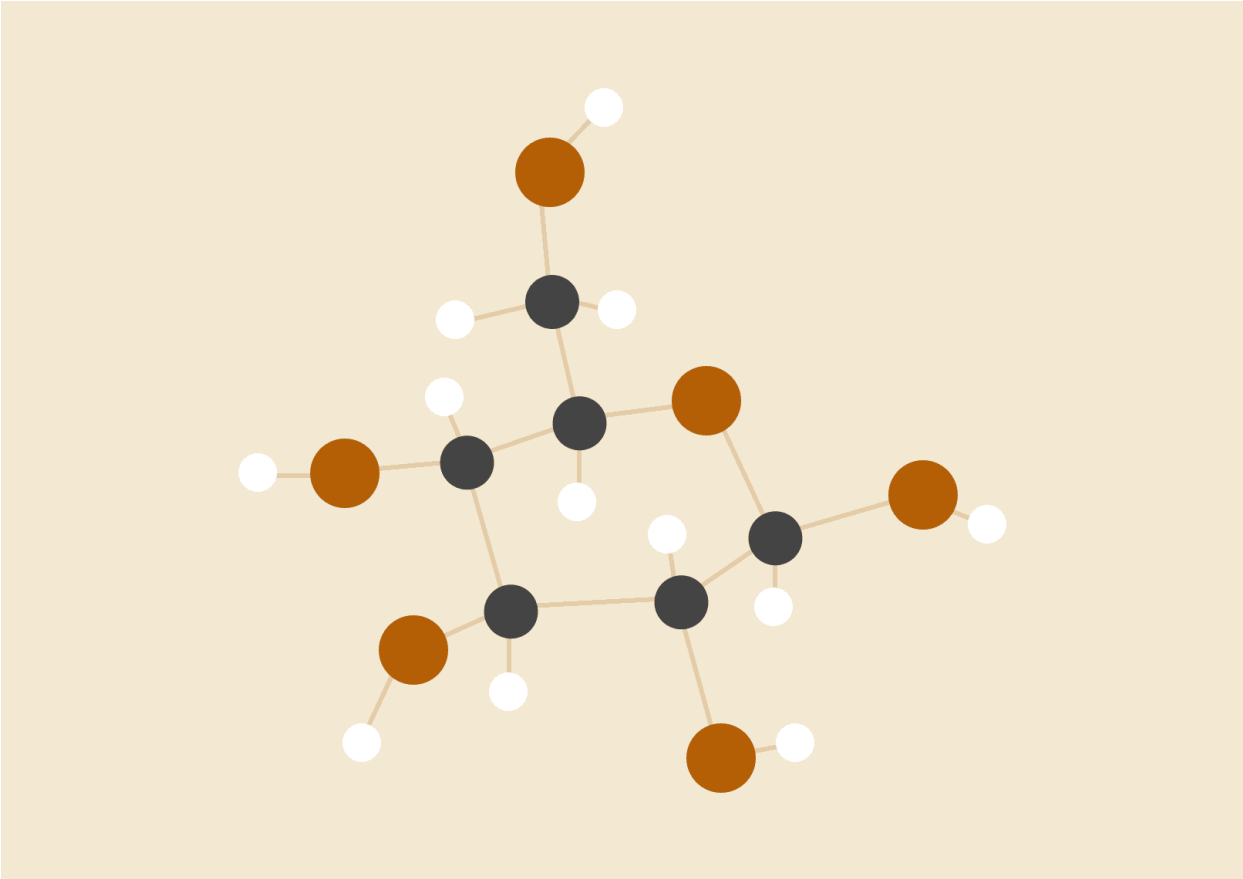
Algorithme de Dijkstra

Rapport de projet, une comparaison des différentes méthode d'implémentation



Ernesto de Menibus

19.03.2020

INTRODUCTION

L’Algorithme de Dijkstra a été développé par le mathématicien et informaticien Edsger W. Dijkstra en 1956. C’est un algorithme qui permet de trouver efficacement le meilleur chemin entre deux sommets d’un graphe orienté et non orienté avec des arêtes a poids différents. Ses applications varient : Google Maps, protocole OSPF , …

PROCÉDURE

L’Algorithme suit une méthode simple;

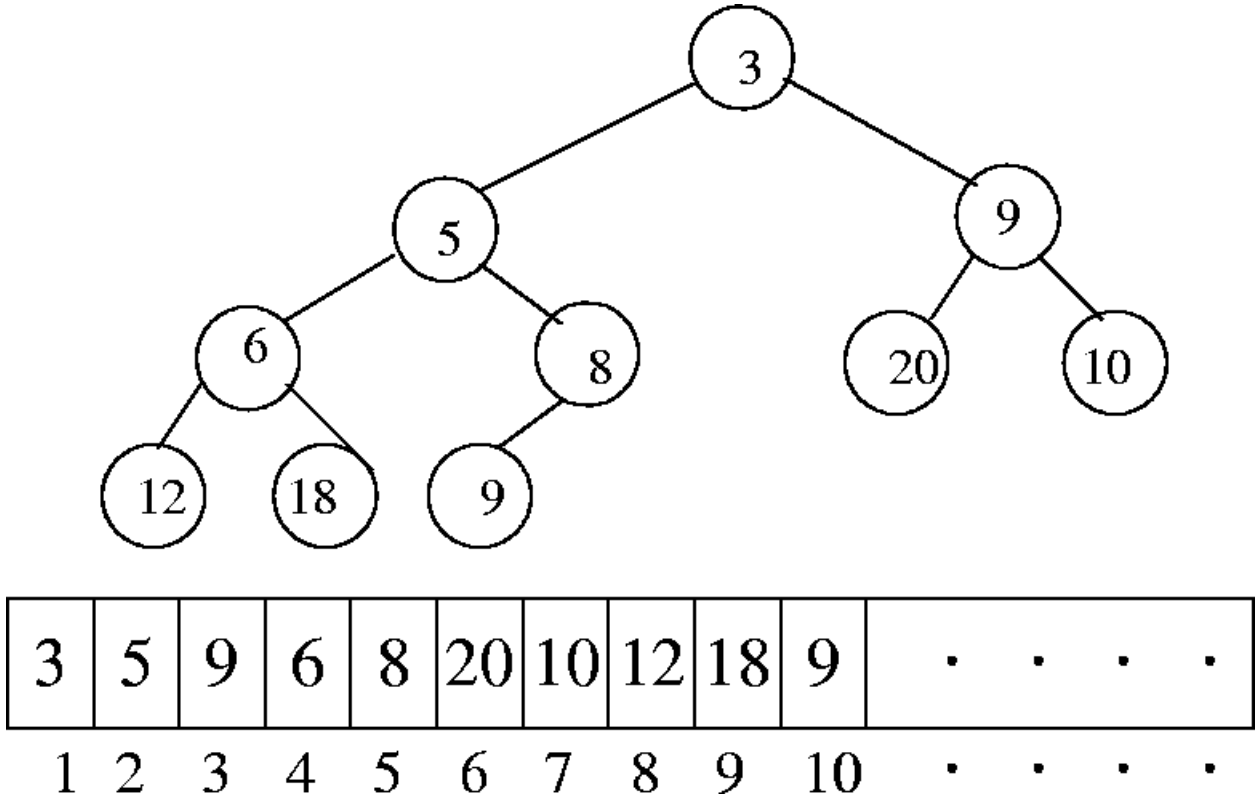
- Trouver le chemin ayant le poid le moins important qu'on a exploré
- Explorer celui-ci
- Mettre à jour le poid du chemin
- Répéter ces étapes

HYPOTHÈSES

Néanmoins, l'étape peut être faite de différentes manières et celle de trouver le chemin ayant le poids le moins important.

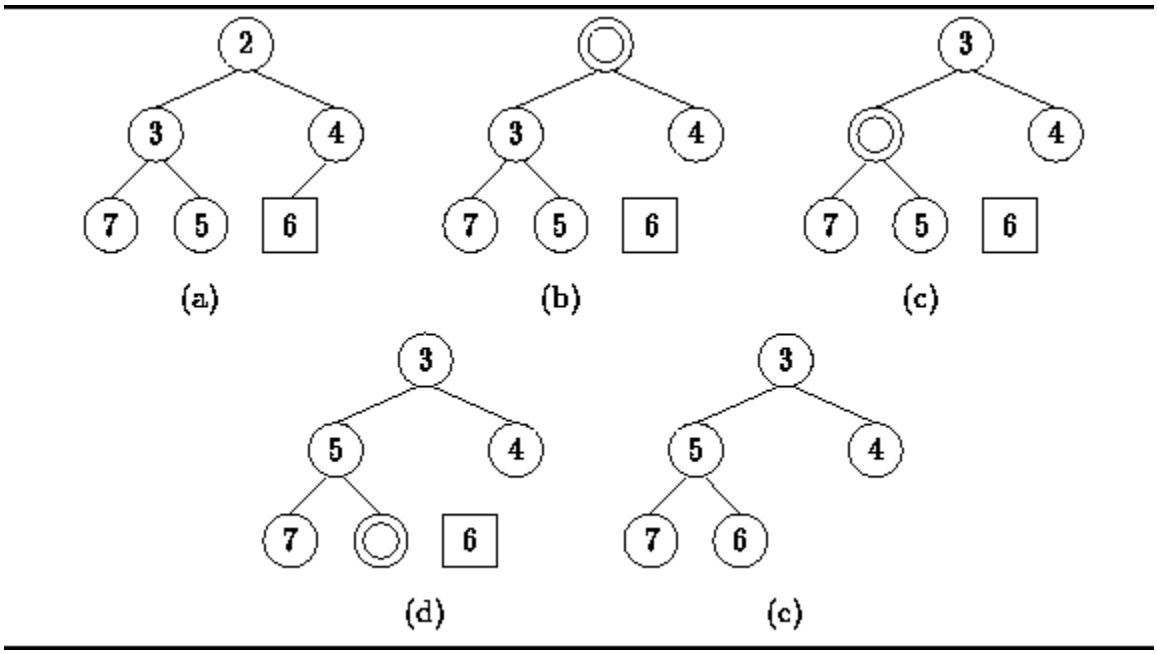
En effet, une méthode simple serait de parcourir tous les poids, enregistrer l’indice du plus petit et l’enlever. Cela a une complexité de O(n) car on regarde chaque élément une fois. Or ce processus peut sembler lent à reproduire sur un grand graph → soit beaucoup de chemins différents. Il existe donc deux autres manières de faire cela, grâce à l'utilisation d’un **Binary Heap** ou bien d’un **Fibonacci Heap** la complexité peut être réduite à O(log(n))

Binary Heap



Voici un exemple de Binary heap. On peut voir que l'élément le plus haut, soit le premier élément, est le plus petit. Donc le temps de recherche du plus petit est de O(1), temps constant. Néanmoins comme dit précédemment, la complexité de l'enlever est de O(log(n)).

Voici la procédure à prendre pour enlever celui-ci ;



Donc si on élève un sommet d'un binary heap ayant pour auteur H on a :

Complexité de changer le sommet parent avec le sommet enfant : O(1)

Donc complexité de changer le sommets dans une branche (down heapify): O(H)

(Dans le pire des cas on fait H echange de O(1))

Donc complexité total: O(1) + O(H) = O(H)

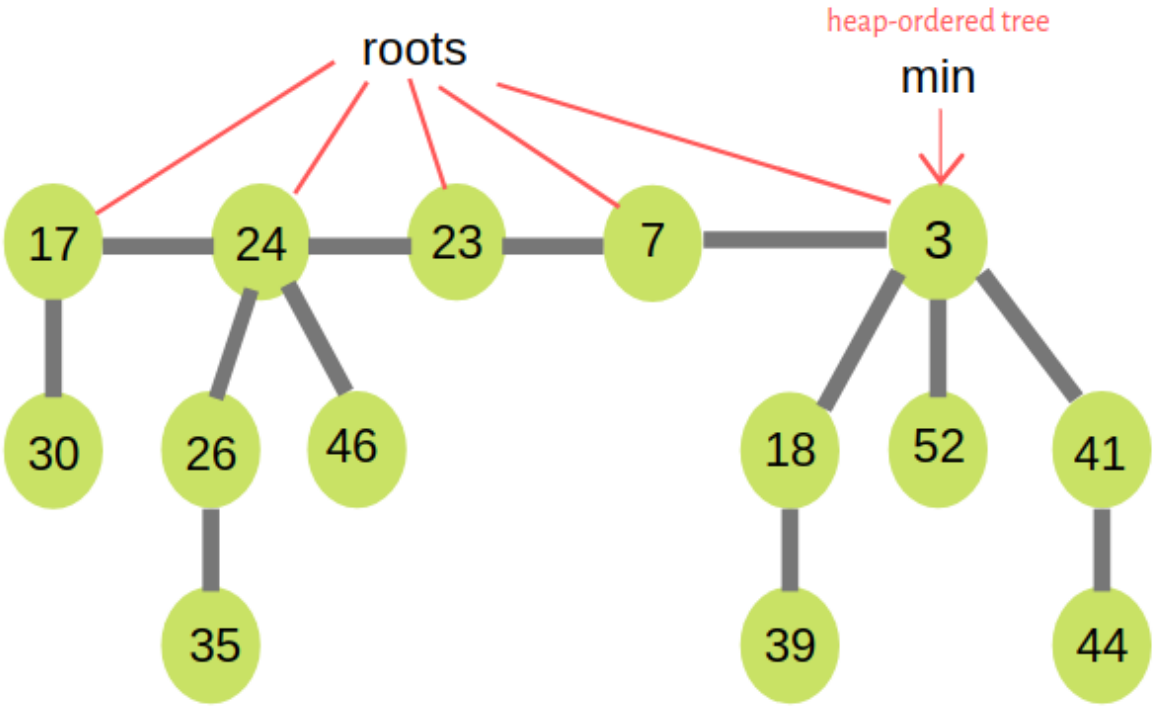
Dans un arbre binaire de hauteur H, complexité = O(log(n)) avec N le nombre de sommets.

Donc la complexité d'enlever le minimum est de O(log(n))

Ajouter un sommet

Afin d'ajouter un nouveau sommet on fait une recherche binaire pour trouver son emplacement dans l'arbre. Ayant donc une complexité de O(log(n)).

Fibonacci Heap

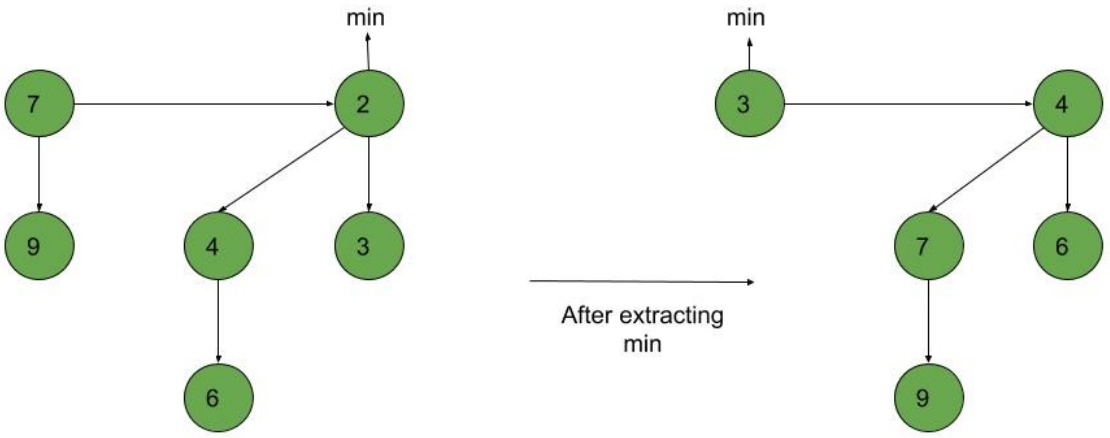


Fibonacci Heaps: Structure

Voici un exemple d'un Fibonacci Heap, une structure de données qui nous permet aussi de rapidement avoir le minimum d'un ensemble de nombres.

Etant donné que cette méthode est plus complexe je ferai de mon mieux pour expliquer comment elle fonctionne.

Extraire le minimum



Afin d'extraire le minimum on performe ses étapes:

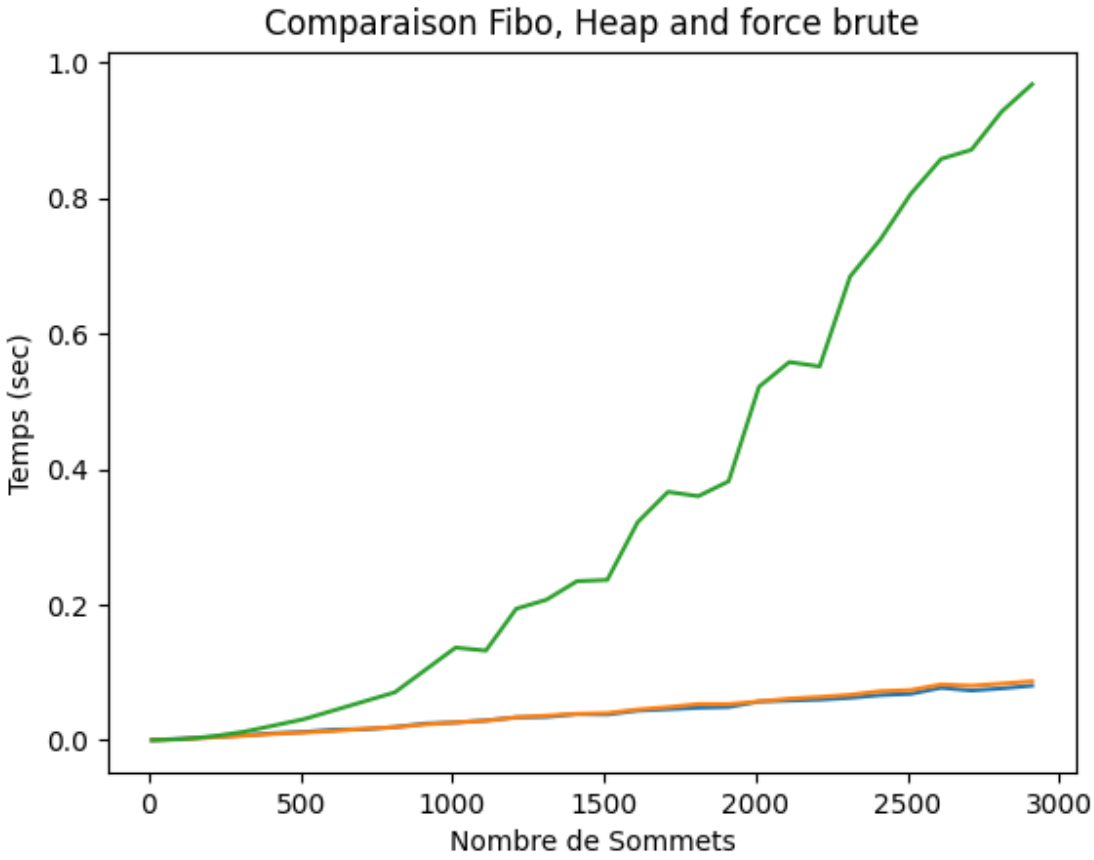
- On efface le minimum
- On rend le prochain minimum le nouveau minimum et on ajoute tous les sous arbres du nouveau minimum à une liste que l'on traitera.
- Ensuite, on replace les sommets qui sont dans la liste en vérifiant si la place leur convient ou pas.

La complexité de ce processus est de **O(log(n))** rendant cette partie de l'algorithme aussi efficace que celui du Binary Heap.

Néanmoins, leurs différences est que la complexité pour ajouter un nouveau sommet dans le Fibonacci Heap est de **O(1), signifiant que théoriquement l'utilisation d'un Fibonacci Heap rendra Dijkstra's plus rapide.**

Résultat de Recherche

Après avoir codé les 3 différentes implémentations en python, j'ai écrit un programme pour les comparer avec des graphes de différentes tailles. Voici les résultat:



Bleu = Fibonacci Heap

Orange = Binary Heap

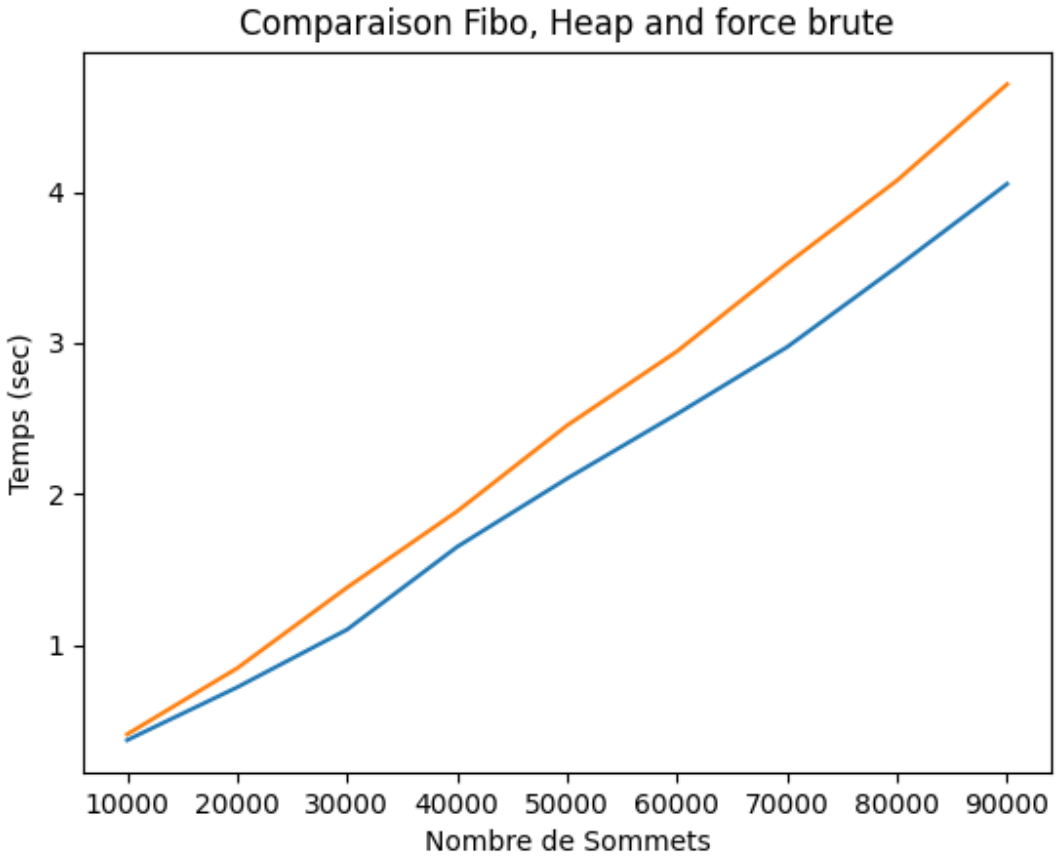
Vert = Méthode naïve

Les résultats sont très intéressants et se montrent attendus. Ceux-ci ont été pris avec de multiples essais afin de créer une moyenne crédible.

On peut donc voir que la méthode naïve à une pente beaucoup plus importante et est donc significativement plus lente.

On peut voir que la resultats ressemble bien a ce qui était attendue (rouge : y = x, bleu : y = log2(x))

Maintenant en comparant Fib et Bin on trouve :



On peut voir que l'utilisation du Fibonacci Heap rend l'algorithme légèrement plus efficace grâce a sa meilleur complexité pour ajouter un nouveau sommet (O(1) vs O(logn))

Complexité de l'Algorithme

Naive : O((V+E)*N) = O(V^2)

Binary Heap : O((V+E)*log(N))

Fibonacci Heap : O(E + V* log(N))

CE QUE J'AI APPRIS ET PROBLÈME RENCONTRÉ

Durant ce projet j'ai beaucoup appris sur les structures de données des Heap que je ne connaissais pas. Leur utilisation et la façon de résoudre des problèmes avec elles. De plus, j'ai appris à comparer la complexité de différents algorithmes et de graphé les résultats avec python afin d'automatiser les tests et comparaisons.

J'ai eu quelques soucis notamment avec la création de graphes. En effet ma méthode qui consiste à utiliser un aspect aléatoire forme des graphes qui n'étaient pas forcément connexes. J'ai donc utilisé un parcours en profondeur afin de déterminer si le graphe est connexe ou pas, et j'ai créé les liens nécessaires pour le rendre connexe. De plus, j'ai aussi eu un problème avec l'implémentation du Binary Heap et Fibonacci Heap car un module existe pour le binary heap mais qui est construit sur du C++ le rendant plus rapide et donc pas comparable à une implémentation Fibonacci que je code en python. Afin de résoudre cela j'ai implémenté le binary heap en python.

CONCLUSION

En conclusion on peut voir que certains algorithmes peuvent être optimisés avec les structures de données qu'on utilise ou d'autres choses. La méthode naïve semble être bien plus lente ne permettant pas de calculer le chemin le plus court dans un graphe avec plus de 10000 sommets. Tandis qu'avec les Heaps on peut calculer au-delà de 100000 sommets (dans un temps raisonnable).

On peut donc dire que le Fibonacci Heap est la structure qui rend Dijkstra's le plus rapide.