Atelier d'approfondissement en informatique: Graphes et Algorithmes

Quentin Garrido

21 avril 2019

Table des matières

1 Introduction

1.1 Objectifs

L'objectif de ce projet est d'implémenter des algorithmes de recherche de plus courts chemins dans un graphe, et plus particulièrement dans un graphe représentant le réseau de métro de Paris.
Nous implémenterons tout d'abord l'algorithme de Dijkstra, puis le modifierons pour qu'il utilise la stratégie A* et nous optimiserons son temps d'éxécution grâce à des structures de données adaptées.

1.2 Utilisation

Tout le code source est disponible à l'adresse suivante : *https ://github.com/garridoq/metro-shortest-path*.

Tout les éxécutables devraient vous être fournis dans le mail et devraient fonctionner sans devoir les recompiler. Dans le cas contraire voici la démarche à suivre :

Un makefile est fourni pour la compilation, il servira à compiler les bibliothèques et les tests. Une fois le code source obtenu il faudra exécuter la commande suivante pour compiler les bibliothèques :

```
> make
```

Vous pourrez alors compiler tous les fichiers de tests de la manière suivante :

```
> make NOM.exe
```

Où NOM est le nom du fichier de test (pour test_heap.c, il faudra entrer make test_heap.exe).

Voici la liste des fichiers de test et leur utilisation :
— test_heap :./test_heap.exe , ce fichier permet de tester l'implémentation du tas binaire et de ses opérations primaires.
— test_dijkstra :./test_dijkstra.exe GRAPH DEBUT FIN , ce fichier va récupérer le graphe dans le fichier GRAPH, puis calculer le plus court chemin de DEBUT à FIN en utilisant l'agorithme de Dijkstra, l'A* et A* avec une file de priorité.
Un fichier EPS sera créé pour chaque algorithme.

Afin de trouver le numéro de station associé à un nom vous pouvez utiliser la commande suivante :

```
> cat GRAPHE | grep -iF NOM
```

La recherche n'est pas sensible à la casse mais l'est aux accents. Vous obtiendrez alors toutes les stations contenant NOM et leur numéro associé.

À titre de référence, voici sur la filgure 1 à quoi ressemble le graphe entier du réseau de métro, que nous utiliserons par la suite :
Les stations sont réprésentées par les sommets et nous avons une arc entre deux sommets si ils sont reliés par le métro.

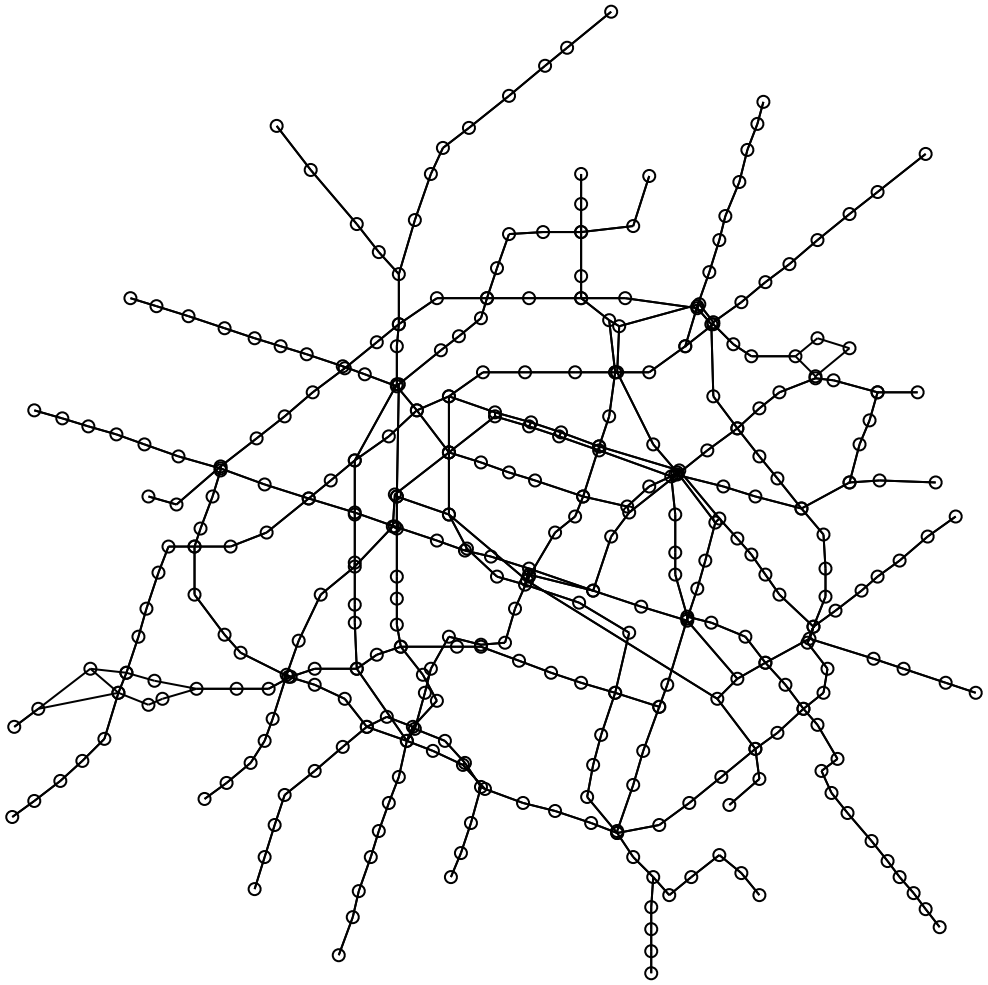

FIGURE 1 – Graphe de référence du métro

2 Algorithme de Dijkstra

2.1 Implémentation

Pour calculer le chemin le plus court d'un point D à A nous allons utiliser la version suivante de l'algorithme de Dijkstra, adaptée depuis le cours de l'unité Graphes et Algorithmes.

Ici nous n'avons pas besoin de calculer les chemins de notre sommet de départ vers tous les autres sommets du graphe et nous nous arrêterons donc dès que nous atteignons notre sommet d'arrivée.

Algorithm 1 Algorithme de Dijkstra

```
1:  procedure DIJKSTRA(E, Γ, l, d ∈ E, a ∈ E)
2:      S = {d}, π(d) = 0, k = 1, x_1 = d
3:      for all x ∈ E\{d} do
4:          π(x) = ∞
5:      end for
6:      while k < n et π(x_k) < ∞ do
7:          for all y ∈ Γ(x_k) tel que y ∉ S do
8:              π(y) = min[π(y), π(x_k) + l(x_k, y)]
9:          end for
10:         Extraire x ∉ S tel que π(x)=min{π(y), y ∉ S}
11:         k = k + 1, x_k = x, S = S ⋃{x_k}
12:         if x_k = a then
13:             break
14:         end if
15:     end while
16:     return π, S
17: end procedure
```

Nous implémentons S avec un tableau de $n = |E|$ éléments, correspondants aux sommets de notre graphe.
Ainsi nous l'initialiserons entièrement à 0 et S = S $\bigcup\{x_k\}$ correspondra à faire S[k] = 1.
Bien que cette implémentation soit plus coûteuse en mémoire qu'une liste chaînée elle permettra d'implémenter l'appartenance à S en temps constant, opération très utilisée aux lignes 7 et 10.
De plus l'ajout d'un élément sera aussi simplifié car nous n'aurons pas à vérifier l'appartenance avant de l'insérer ou non.

2.2 Récupération du plus court chemin

Soit c notre chemin de coût minimum, pour tout $u = (x, y) \in \vec{\Gamma}$, $u \in c$ si $l(x, y) = \pi(y) - \pi(x)$ avec $\pi(x)$ le coût d'un plus court chemin de i (notre point de départ) à x.

Nous allons donc utiliser cette propriété pour trouver notre plus court chemin grâce à l'algorithme suivant :

Algorithm 2 Plus court chemin

```
1: procedure PCC(π, Γ^{-1}, l, d ∈ E, a ∈ E)
2:     x = a, c = ∅
3:     while x ≠ d do
4:         for all y ∈ Γ^{-1}(x) do
5:             if π(x) − π(y) = l(y, x) then
6:                 c = c ⋃ {(y, x)}
7:                 x = y
8:                 break
9:             end if
10:        end for
11:    end while
12:    return c
13: end procedure
```

2.3 Résultats

Considérons un trajets des stations Alexandre Dumas (1) à Porte Dauphine (256). Nous trouvons alors le chemin le plus court suivant :

Alexandre Dumas − > Philippe-Auguste − > Père Lachaise − > Ménilmontant − > Couronnes − > Belleville − > Colonel Fabien − > Jaurès − > Stalingrad − > La Chapelle − > Barbès Rochechouart − > Anvers − > Pigalle − > Blanche − > Place de Clichy − > Rome − > Villiers − > Monceau − > Courcelles − > Ternes − > Charles de Gaulle, Étoile − > Victor Hugo − > Porte Dauphine

Nous pouvons observer ce chemin sur la figure 2. Les arcs forment le plus court chemin et les sommets sont uniquement ceux visités. Nous en avons visité 329 sur 376.

Comme nous pouvons le voir nous avons parcouru des sommets qui nous éloignaient grandement du résultat uniquement car le coût pour y aller était plus faible (cf ligne 10 de l'algorithme).
Cet effet est exacerbé ici car le trajet que nous avons choisi est particulièrement long.
Ce constat reste le même sur tous les trajets, sauf sur ceux très courts.

Bien que le chemin que nous trouvions ne paraisse pas optimal, nous l'avons vérifié via le service Vianavigo de la RATP, où nous trouvons le même chemin. Cela est dû au fait que les deux stations sont sur la même ligne et donc que nous avons aucun changement à faire, d'où la plus courte durée du trajet.

Nous avons vérifié plusieurs autres trajets de la même manière avec le même résultat à chaque fois, ainsi notre implémentation semble être correcte, à condition que l'algorithme utilisé par la RATP pour Vianavigo soit correct, ce que nous pouvons affirmer être vrai.

FIGURE 2 – Résultat de l'algorithme de Dijkstra pour aller des sommets 1 à 256

3 Stratégie A*

Pour utiliser la stratégie A* nous allons appliquer une heuristique lors du choix du prochain sommet à étudier dans l'algorithme de Dijkstra, ce qui correspond à la ligne 10 de l'algorithme.
En effet nous avons pu voir que choisir toujours le somemt avec le plus court chemin vers lui se révèle sous optimal car nous allons partir dans des directions qui nous éloignent de l'arrivée.
Nous allons donc essayer de choisir une heuristique qui va nous permettre d'explorer moins de sommets.

Le principe de la stratégie A* est le suivant :
On considère $f(n)$ une estimation du coût d'un plus court chemin de notre départ d à notre arrviée a passant par n. On a alors :

$$f(n) = g(n) + h(n)$$

g(n) est le coût d'un chemin optimal de d à n, dans notre cas $\pi(n)$.
h(n) est notre heuristique qui estimera le coût d'un chemin optimal de n à a.

Nous utiliserons alors cette fonction f pour comparer deux sommet et savoir lequel nous allons étudier ensuite.

3.1 Choix de l'heuristique

Dans notre cas, la durée de trajet entre deux stations correspond au poids de l'arc. Cette valeur est calculée pour un métro se déplaçant en moyenne à 10 $m.s^{-1}$ et une unité dans notre graphe correspond

à 25.7 m.

L'heuristique que nous devons choisir doit estimer le plus court chemin du sommet courant à l'arrivée. Ainsi nous allons utiliser la distance euclidienne entre notre sommet actuel et l'arrivée.
Nous possédons les coordonnées de tous les sommet mais il faut les convertir en mètres en multipliant par 25.7 puis diviser par 10 pour obtenir une valeur comparable aux valeurs des arcs, et par conséquent des longueurs de chemins.
Cette heuristique devrait s'avérer bonne car très simple à calculer et elle représente le trajet à vol d'oiseau entre nos stations, ce qui est le résultat optimal peu importe le moyen de transport utilisé, aucun trajet ne peut être plus court que cela.

Notre heuristique est alors :

Algorithm 3 Heuristique

```
1: procedure HEURISTIQUE(E, i ∈ E, a ∈ E)        ▷ a est notre point d'arrivée
2:     return distance_euclidienne(i, a) × 25.7/10
3: end procedure
```

Une seule question subsiste, il s'agit de l'optimalité de la solution. En effet si notre heuristique est trop 'forte' par rapport aux valeurs des chemins nous n'explorerons jamais certains sommets qui nous donneraient un plus court chemin.
L'algorithme A* trouvera une solution optimale si l'heuristique est admissible, c'est à dire si l'heuristique ne surestime jamais le coût pour atteindre l'arrivée.
Dans notre cas, la distance euclidienne étant toujours la distance optimale (en supposant que nous sommes dans un plan, en réalité ce n'est pas le cas à cause de la courbure de la Terre, mais l'approximation est acceptable dans notre cas, Paris n'étant pas assez vaste pour que la courbure aie une grande influence) nous pouvons garantir l'optimalité de la solution trouvée.

Nous pouvons bien voir qu'avec cette condition d'optimalité, l'algorithme de Dijkstra trouve une solution optimale car il correspond à avoir une heuristique toujorus égale à 0, or dans un réseau avec des arcs à valeur positive, un chemin est toujours plus long que 0.

3.2 Implémentation de l'algorithme

Pour implémenter la stratégie A* seules de très légères modifications de l'algorithme de Dijkstra sont nécessaires, il nous suffit de modifier le choix du prochain sommet à étudier à chaque étape. Nous obtenons alors :

Algorithm 4 Algorithme A*

```
1: procedure A*(E, Γ, l, d ∈ E, a ∈ E)
2:     S = {d}, π(d) = 0, k = 1, x_1 = d
3:     for all x ∈ E\{d} do
4:         π(x) = ∞
5:     end for
6:     while k < n et π(x_k) < ∞ do
7:         for all y ∈ Γ(x_k) tel que y ∉ S do
8:             π(y) = min[π(y), π(x_k) + l(x_k, y)]
9:         end for
10:        Extraire x ∉ S tel que π(x)=min{π(y)+heuristique(y), y ∉ S}
11:        k = k + 1, x_k = x, S = S ⋃{x_k}
12:        if x_k = a then
13:            break
14:        end if
15:    end while
16:    return π, S
17: end procedure
```

3.3 Résultats

Afin de pouvoir comparer la stratégie A* et Dijkstra de manière la plus juste possible nous allons réutiliser le même chemin que précédemment.

Nous obtenons désormais la figure suivante :

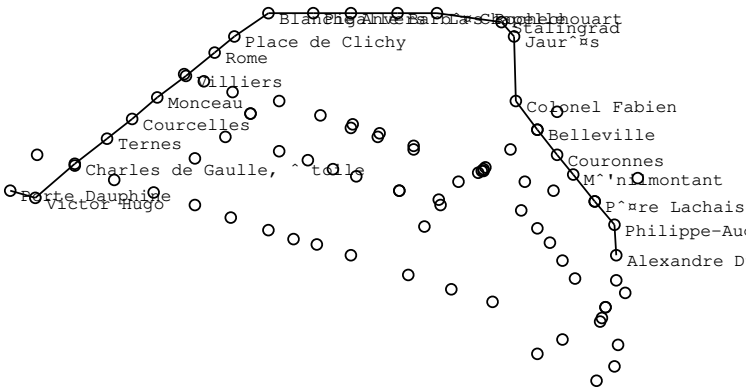

FIGURE 3 – Résultat de la stratégie A* pour aller des sommets 1 à 256

Comme nous pouvons le voir, nous explorons bien moins de sommet, avec la stratégie A* nous en explorons uniquement 89 contre 329 auparavant.
Cela se traduit au niveau du temps d'éxécution qui passe de $550\mu s$ à $241\mu s$ ce qui est une amélioration

très importante, et qui le sera de plus en plus en augmentant la taille du graphe. Comme attendu, nous priorisons les sommets nous rapprochant de l'arrivée et donc il ne nous arrive plus de partir dans des directions opposées. Cet exemple est un des pire cas possible pour la stratégie A* dans notre problème car le chemin le plus court est tout sauf direct pour l'arrivée, ce qui rend moins efficace notre heuristique.

Afin de mieux se rendre compte des différences entre nos stratégies voici d'autres exemples d'éxécution :

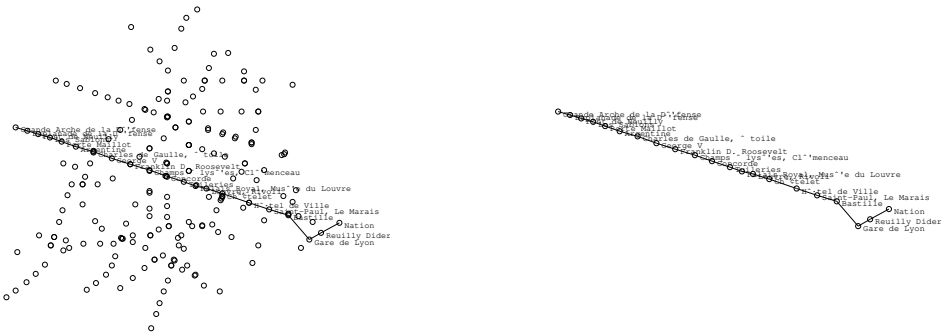

FIGURE 4 – Comparaison entre Dijkstra(à gauche) et A*(à droite) de 130 à 212

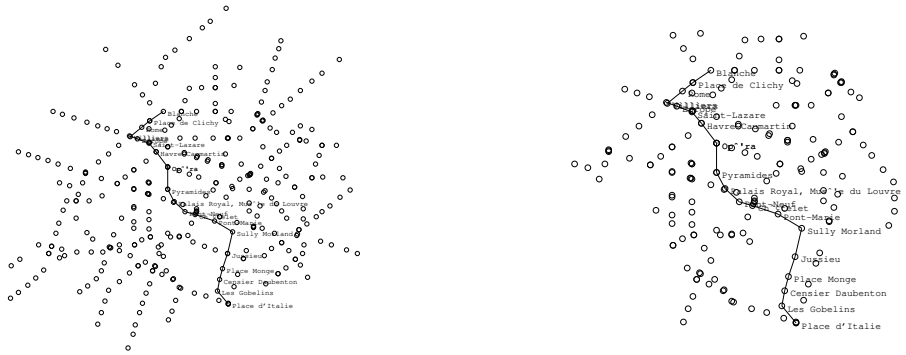

FIGURE 5 – Comparaison entre Dijkstra(à gauche) et A*(à droite) de 27 à 244

FIGURE 6 – Comparaison entre Dijkstra(à gauche) et A*(à droite) de 200 à 201

FIGURE 7 – Comparaison entre Dijkstra(à gauche) et A*(à droite) de 146 à 212

Voici la comparaison des temps d'éxécution sur ces exemples :

Départ	Arrivée	Temps avec Dijkstra (en μs)	Temps avec A*(en μs)	rapport
130	212	350	110	3.18
27	244	500	420	1.19
200	201	600	530	1.13
146	212	230	170	1.35

Nous pouvons voir que comme sur notre autre exemple nous parcourons en général bien moins de sommet mais le temps d'éxécution ne varie pas toujours autant que précédemment.
Mais même dans le pire de nos exemples, nous avons quand même un gain de 13%, ce qui reste intéressant.

4 Tas Binaire

Que ce soit dans l'algorithme de Dijkstra ou dans la stratégie A*, une des opérations les plus coûteuses reste la ligne 10, lorsque que nous calculons un minimum, cette opération s'effectue en $O(n)$.
Cependant avec des structures de données adaptées cette opération peut devenir en $O(1)$ (ou $O(log(n))$ si nous souhaitons extraire ce minimum) tout en gardant une compléxité en $O(log(n))$ pour les autres opérations. Une des structures permettant cela est le tas binaire, qui est relativement simple à implémenter.

Un tas binaire est un arbre binaire qui possède à sa racine son élément le plus petit (tas minimun) ou le plus grand (tas maximum).
De plus pour chaque noeud dans l'arbre tous ses 'enfants' son plus petits que lui (tas minimum) ou plus grands (tas maximum).

4.1 Implémentation classique

Tout d'abord nous avons implémenté un tas binaire sans qu'il serve de file de priorité.
Afin de l'implémenter nous avons utilisée un struct contenant un tableau (les éléments), sa capacité maximale, et sa capacité actuelle.
Par la suite nous parlerons uniquement d'un tas binaire minimum car c'est celui ci que nous souhaitons utiliser pour notre problème.

En effet un tas binaire peut être implémenté avec un tableau uniquement, comme nous pouvons le voir sur le schéma suivant :

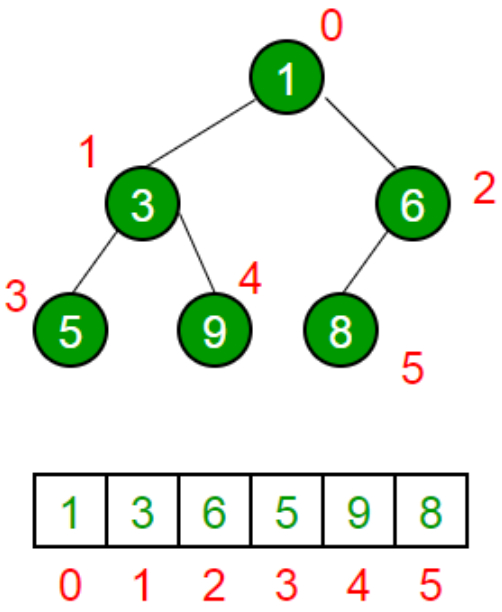

FIGURE 8 – Représentation en mémoire d'un tas binairei - Source : geeksforgeeks.org

Pour chaque noeud (indicé i), nous pouvons obtenir son parent et ses enfants de la manière suivante :
— parent(i) = $\lfloor \frac{i-1}{2} \rfloor$
— gauche(i) = $i \times 2 + 1$

— droite(i) = $i \times 2 + 2$

Le tas binaire doit supporter les opérations suivantes, dont nos implémentations proviennent de *Introduction à l'algorithmique* :
— minHeapify(t, i) : cette méthode va rétablir les propriétés du sous tas t à l'indice i
— insert(x) : cette méthode va insérer un élément dans notre tas via minHeapify
— buildMinHeap(tab) : cette méthode va construire un tas à partir d'un tableau non trié

Vous pourrez trouver un implémentation d'un tas maximum dans le fichier *heap.c* et les tests via le fichier *test_heap.c*.

4.2 Implémentation pour notre problème

Afin de pouvoir utiliser un tas binaire comme file de priorité il va falloir y apporter quelques modifications liées à notre problème.

Nous allons utiliser deux tableaux, *elements* qui contient les sommets de notre graphe et *keys* qui contient les valeurs utilisées pour le tri dans le tas. Dans *keys* nous stockerons le coût d'un plus court chemin vers notre sommet plus son heuristique.

Un troisième tableau à n éléments, *indice* sera utilisé afin de savoir si un sommet est déjà présent dans le tas et si il y est où se trouve-t-il, si il est absent nous utiliserons la valeur -1. Nous avons alors indice[i] = argument de i dans *elements*.
Ce tableau nous permettra de savoir si un sommet est dans le tas binaire en temps constant, ce qui nous sera utile pour savoir si nous allons devoir insérer un sommet ou juste mettre sa clef à jour. Nous pourrions toujours réinsérer un sommet peut importe qu'il y soit ou non mais cela nous forcerait a utiliser un tas de taille maximale dynamique, et aurait aussi pour conséquence de ralentir l'ajout d'éléments dans la file de priorité car le tas contiendrait plus d'éléments.

Nous aurons aussi besoin des nouvelles fonctions suivantes :
— minInsert(elt, key) : il s'agit d'une variante de *insert* qui insère un élément et sa clef associée dans notre file de priorité
— getMin(t) : cette fonction retourne l'élément minimum dans la file de priorité (l'élément à l'indice 0)
— extractMin(t) : cette fonction va enlever l'élément minimal de la file de priorité et le retourner.
— decreaseKey(elt, key) : cette fonction va baisser la valeur de la clef de l'élément *elt* pour y mettre la valeur *key*. Elle s'effectue en temps logarithmique grâce à notre tableau *indices*

Nous allons alors modifier notre algorithme pour utiliser cette file de priorité afin d'obtenir plus rapidement l'élément minimum :

Algorithm 5 Algorithme A* avec file de priorité

```
1: procedure A*(E, Γ, l, d ∈ E, a ∈ E)
2:     S = {d}, π(d) = 0, k = 1, x_1 = d, file = ∅
3:     for all x ∈ E\{d} do
4:         π(x) = ∞
5:     end for
6:     while k < n et π(x_k) < ∞ do
7:         for all y ∈ Γ(x_k) tel que y ∉ S do
8:             π(y) = min[π(y), π(x_k) + l(x_k, y)]
9:             if y ∈ file then
10:                decreaseKey(file, y, pi[y]) ;
11:            else
12:                minInsert(file, y, pi[y]) ;
13:            end if
14:        end for
15:        x = extractMin(file)
16:        k = k + 1, x_k = x, S = S ⋃{x_k}
17:        if x_k = a then
18:            break
19:        end if
20:    end while
21:    return π, S
22: end procedure
```

En comparant à précédemment nous avons désormais decreaseKey qui est en $O(log(n))$ minInsert qui est en $O(log(n))$ et extractMin en $O(1)$.
Nous avions avant un calcul de minimum en $O(n)$.
Malgré l'ajout de deux opérations en temps logarithmique, nous devrions voir de grandes améliorations de temps d'éxécution car nous avons transformé une opération en temps linéaire en temps constant.

4.3 Résultats

En rajoutant cette file de priorité nous trouvons toujours les bons chemins les plus courts, cependant bien plus rapidement, voici un tableau récapitulatif sur certains trajets :

Départ	Arrivée	Temps sans file (en μs)	Temps avec file (en μs)	rapport
130	212	650	530	1.22
146	212	1050	500	2.1
1	256	1090	550	1.98
27	244	2000	800	2.5
200	201	2500	750	3.33

Ces résultats ont été obtenus en réalisant 3 fois chaque mesure puis en prenant la moyenne des trois essais.

Comme nous pouvons le voir le temps est toujours meilleur avec la file de priorité, en moyenne nous sommes environ deux fois plus rapidesi avec que sans, avec des cas extrèmes où nous sommes à peine plus rapides et d'autres où nous sommes plus de trois fois plus rapides.

4.4 Autre structures de données possibles

Le tas binaire n'est pas la seule manière d'implémenter une file de priorité, et d'autres types de tas permettent d'obtenir des meilleures complexités sur certaines opération, voici un tableau récapitulatif pour certains types de tas :

Tas	getMin	extractMin	insert	decreaseKey	merge
Binaire	$\Theta(1)$	$\Theta(log(n))$	$O(log(n))$	$O(log(n))$	$\Theta(n)$
Binomial	$\Theta(log(n))$	$\Theta(log(n))$	$\Theta(1)*$	$\Theta(log(n))$	$\Theta(log(n))$
Fibonnaci	$\Theta(1)$	$O(log(n))$	$\Theta(1)$	$\Theta(1)*$	$\Theta(1)$

* : Temps ammorti

Comme nous pouvons le voir, le tas binaire est largement battu théoriquement par les tas binomiaux ou de Fibonnaci, notamment pour l'opération merge.
Cependant dans notre cas nous allons surtout nous intéresser aux opérations extractMin, insert et decreaseKey qui sont celles que nous allons utiliser pour notre problème.
ExtractMin garde la même complexité peu importe la structure de données choisie.
Insert passe en temps constant pour les tas binomiaux et de Fibonnaci, cependant cela ne sera rentable en pratique que si les constantes, masquées par l'étude asymptotique de la complexité, sont similaires. Or le tas binaire est très simple à implémenter, notamment face au tas de Fibonnaci qui lui est assez complexe à implémenter en pratique et où les cosntantes deviennent trop grande pour une utilisation pratique.
Peu importe la taille, passer d'un temps linéaire à logarithmique est rentable, mais passer d'un temps logarithmique à linéaire ne l'est pas forcément, car le logarithme croît très lentement ($log(10^n) = n$). C'est là que la difficulté d'implémentation peut rentre cette amélioration théorique inutilisable en pratique.

Le constat est le même pour decreaseKey, qui passe d'un temps logarithmique pour le tas binaire ou binomial à un temps constant pour le tas de Fibonnaci.

Nous pouvons alors voir que dans notre cas, le tas de binaire est le bon compromis entre difficulté d'implémentation et complexité, cependant si nous avions dû fusionner des tas, il aurait été intéressant d'implémenter un tas binomial ou de Fibonnaci.

5 Conclusion

Nous avons pu voir qu'en partant de l'algorithme de Dijkstra, qui est l'algorithme de référence pour trouver un plus court chemin dans un réseau à longueurs positives, nous avons réussi à arriver à un résultat bien plus rapide grâce à la stratégie A* et à une file de priorité implémentée grâce à un tas binaire.
Nous avons aussi pu nous rendre compte de la dualité qu'il existe entre mémoire et temps d'éxécution, en effet nous avons pu accélérer certaines opérations (appartenance à un tas binaire) en gardant des informations supplémentaires en mémoire. Et de manière analogue nous avons pu gagner en mémoire (taille maximale de notre tas binaire) en ayant un peu plus d'opérations à effectuer (appartenance au tas binaire).

Ainsi beaucoup de solutions que nous avons apportées pour notre problème sont généralisables mais certaines moins que d'autres, car ici nous connaissions certaines informations très utiles (position dans l'espace de nos sommets pour notre heuristique) qui nous ont permis d'adapter une solution plus générale à notre problème. Ainsi avec d'autres informations nous aurions pu adapter encore différemment la méthode pour qu'elle fonctionne mieux sur un certain problème.