UVSQ université PARIS-SACLAY | ISTY Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

Rapport du Projet d'Analyse d'algorithmes et Validation de programmes

Problème de plus court chemin

Réalisé par :

GHANNOUM Jihad IATIC4

Mr. Devan SOHIER : Encadrant
Projet IATIC4 effectué du 05/02/2022 au 03/03/2022

TABLE DES MATIÈRES

Executive Summary

Dans ce rapport, nous présenterons le problème de plus court chemin dans un graphe pondéré. Ce dernier a de nombreuses applications, notamment dans les réseaux routiers. Ainsi après avoir présenté le problème en détail, nous verrons plusieurs algorithmes le résolvant de manière plus ou moins efficace. Nous les comparerons ensuite afin de voir si leurs performances empiriques sont les mêmes que celles auxquelles on peut s'attendre théoriquement.

Introduction

Le problème de plus court chemin dans un graphe est l'un des problèmes les plus courants en optimisation combinatoire. Ce problème se présente comme suit : étant donné un graphe et une fonction coût sur les arcs, le problème consiste à trouver le chemin le moins coûteux d'un sommet choisi à un autre. La définition du chemin requiert que les sommets consécutifs soient connectés par une arête commune.
Il se résout aisément grâce à de nombreux algorithmes (Dijkstra, Bellman, etc.), ils ne seront pas tous abordés. Nous allons ici nous intéresser aux différentes façons d'implémenter l'algorithme de Dijkstra.

Partie 1 : Structures de données et primitives

Dans cette partie, nous allons présenter les différentes structures de données utilisées pour résoudre le problème choisi.

Soit $G = (S, A)$ un graphe pondéré, les sommets sont stockés par des entiers de 1 à N. G est stocké avec des listes de successeurs. Les valuations sont rangées dans une variable de type "double" pour chaque arête de type *LISTE_ADJACENCE_NOEUD* (cette structure permet de représenter la liste des arcs sortant d'un nœud).

Nous utilisons des structures de tas pour avoir rapidement le sommet d'étiquette minimale. Un tas permet d'enlever le sommet d'étiquette minimale, de modifier l'étiquette d'un sommet interne, et d'ajouter un nouveau sommet. Voici les primitives qui nous sont utiles :

- Création d'un tas : $O(|S|)$
- Opérations (extraire_min,...) : $O(S.log(S))$
- Tas_est_vide : $O(1)$

Le tas de Fibonacci est une structure plus compliquée qui améliore l'opération *INSERT*, le coût au pire est toujours $O(logN)$, mais la complexité moyennée sur la pire suite d'appels est seulement $O(1)$ grâce à des effets de compensation.

Nous utilisons aussi une structure *ETAT* permettant de stocker l'état d'un nœud dans un tableau à une dimension.

Les états possibles :

- NON-VISITE -> "0"
- EN_COURS -> "1"
- DEJA_VISITE -> "2"

Partie 2 : L'algorithme de Dijkstra avec un tas

2.1 Présentation théorique :

L'algorithme de Dijkstra est sans doute l'un des algorithmes de plus court chemin le plus connu. Ce dernier se base sur le parcours en largeur afin de trouver le plus court chemin dans un graphe pondéré positivement. Nous savons qu'un parcours en largeur permet de trouver le plus court chemin sur un graphe non pondéré et qu'il utilise une file pour stocker les éléments afin de les parcourir profondeur par profondeur.
Puisque tous les arcs n'ont pas la même pondération, l'algorithme de Dijkstra n'utilise pas une file simple, mais une file à priorité (tas). Cette dernière permet toujours d'avoir le nœud avec la distance minimale par rapport au départ en tête de file, et garantie alors que nous avons trouvé le plus court chemin lorsqu'on atteint le nœud de sortie.
La file à priorité a aussi l'avantage d'être une structure de données très efficace et rapide, parce qu'il permet l'insertion de nouveau nœuds en temps logarithmique et un simple accès au plus grand élément, ce qui rend notre algorithme d'autant plus intéressant

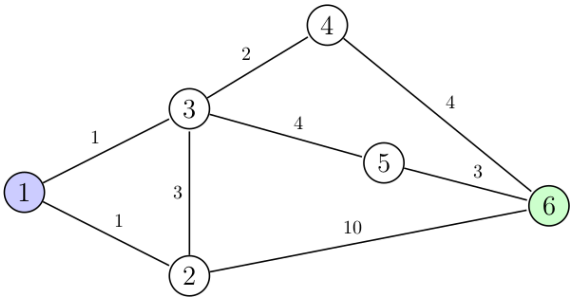
Exemple :

Soit $G = (S, A)$ est un graphe pondéré avec S : l'ensemble des sommets et A : l'ensemble des arêtes. ($|S| = n$ et $|A| = m$)
Soit $p : A \rightarrow \Re$ une fonction de pondération des arcs.

Le poids d'un chemin $c =< v_0, v_1, ..., v_k >$ est $p(c) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$

Le poids d'un chemin le plus court entre u et v :

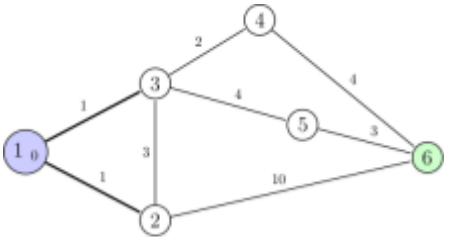
- S'il existe un chemin entre u et v, $\delta(u, v) = min\{p(c) : u \rightarrow v\}$
- Sinon $\delta(u, v) = \infty$



Exemple d'un graphe pondéré

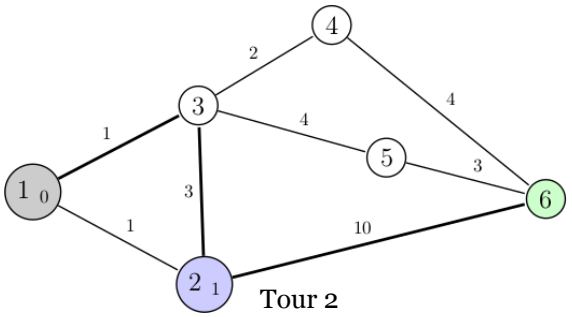
Nous souhaitons trouver le plus court chemin pour aller du nœud de départ "en bleu" au nœud d'arrivée "en vert".

L'algorithme de Dijkstra commence à examiner les voisins du nœud de départ "1".



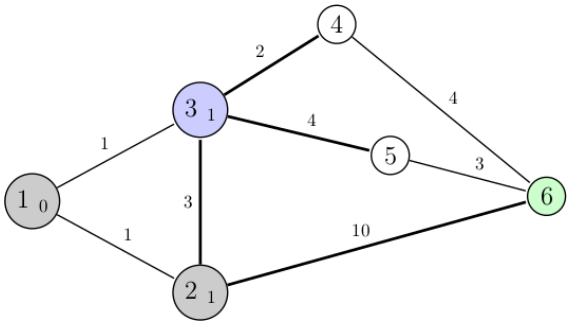
Tour 1

Le noeud “1” représente désormais le nœud en tête de notre file à priorité, et nous avons aussi indiqué la distance parcourue depuis le noeud initial en indice. Et comme le nœud en bleu possède deux arcs, nous avons deux possibilités, soit d’aller du nœud 1 → nœud 2, soit du nœud 1 → nœud 3.
L’algorithme prend en compte la distance depuis le nœud de départ ainsi que le poids des arcs afin de choisir le prochain nœud à visiter.
Maintenant, nous avons les combinaisons suivantes : 0+1 “nœud 1 → nœud 2” et 0+1 “nœud 1 → nœud 3”. Dans notre cas, les deux nœuds sont équivalents en termes d’efficacité.
Nous imaginerons que l’on visite le nœud 2 (même si le nœud 3 nous conduira au plus court chemin). Après avoir retiré le nœud 1, notre fil contient les nœuds 2 et 3, mais le “2” apparaît avant, c'est donc celui-ci que nous allons visiter au prochain tour :



Tour 2

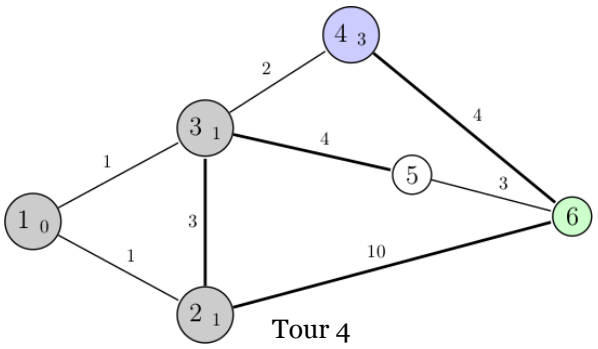
L’algorithme nous mène au nœud 2, avec une distance totale 1. Lors du dernier tour, nous avions deux choix possibles d’arcs à emprunter, en choisissant l’un des deux, il est important de conserver l’autre au cas où justement ce n’est pas le chemin le plus optimal que nous venons de prendre.
La file à priorité se chargera de faire remonter automatiquement le nœud 3 en tête si ce dernier devient finalement un choix plus intéressant.
Après avoir ajouté les voisins du nœud 2 à notre file, nous devons choisir entre 0+1 “nœud 1 → nœud 3), 1+3 “nœud 2 → nœud 3” ou 1+10 “nœud 2 → nœud 6”. Nous voyons que le minimum est atteint pour 0+1, soit le nœud 1 vers le nœud 3.
Notre file ne possède donc plus le nœud 2, et a en tête le nœud 3 que nous parcourons lors du prochain tour :



Tour 3

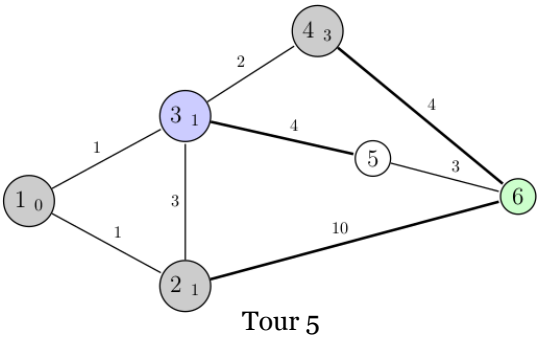
Nous sommes au nœud 3 avec une distance totale depuis le nœud de départ de 1, et maintenant l’algorithme va examiner les voisins du nœud 3 et nous allons chercher à comparer nos anciennes possibilités à celles qui se sont rajoutées (les voisins du nœud 3).

L'algorithme doit choisir entre: 1+2 "nœud 3 → nœud 4", 1+4 "nœud 3 → nœud 5", 1+3 "nœud 2 → nœud 3", et 1+10 "nœud 2 → nœud 6".
Allant du nœud 3 vers le nœud 4 c'est le choix avec la distance minimale.



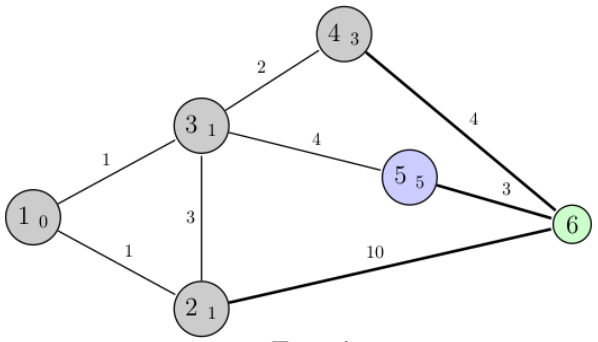
Tour 4

Maintenant, l'algorithme doit choisir entre les différents chemins possibles: 3+4 "nœud 4 → nœud 6", 1+4 "nœud 3 → nœud 5", 1+10 "nœud 2 → nœud 6", 1+3 ""nœud 2 → 3". Le choix avec une distance minimale est donc le dernier : allant du nœud 2 vers le nœud 3.



Tour 5

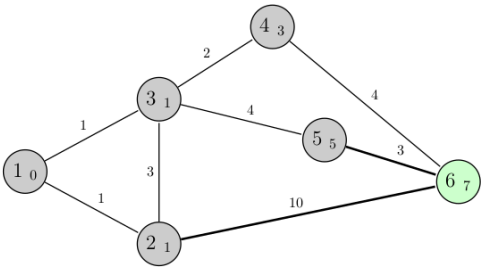
Cependant lorsqu'on arrive sur le nœud 3, on s'aperçoit qu'on a déjà visité ce nœud, donc il est inutile de recommencer cette opération, et l'algorithme va choisir un autre chemin entre : 3+4 "nœud 4 → nœud 6", 1+10 "nœud 2 → nœud 6", et 1+4 "nœud 3 → nœud 5".
Le chemin choisi est donc celui reliant le nœud 3 au 5.



Tour 6

Comme auparavant, nous prenons en compte le nouvel arc possible et nous choisissons donc entre : 3+4 "nœud 4 → nœud 6", 5+3 "nœud 5 → nœud 6", 1+10 "nœud 2 → nœud 6". Le

premier choix étant celui avec le plus petit résultat, c'est que nous décidons de réaliser.



Nous arrivons au nœud d'arrivée (celui en vert), notre algorithme a donc terminé et nous avons trouvé le plus court chemin pour s'y rendre (les autres possibilités restantes étant forcément des chemins avec une plus longue distance).

Dans cet exemple, l'algorithme a visité tous les nœuds du graphe ainsi que la plupart des arcs, mais nous avons choisi ceci pour justement montrer au mieux le fonctionnement de cet algorithme et surtout comment ce dernier réalise son choix pour trouver le chemin optimal en termes de distance.
L'algorithme de Dijkstra est très utile sur d'énormes graphes, car il ne visitera que le nœud dont il a réellement besoin puisque si nous réfléchissons à sa manière de fonctionner, nous ne pouvons pas savoir à l'avance si le chemin qu'il emprunte actuellement restera optimal jusqu'au bout (c'est d'où vient la nécessité de conserver les différents choix rencontrés lors du parcours), et inversement il se peut que les autres chemins soient en réalité pires et qu'il faut donc bien garder celui actuel.

Pseudo-code :

On suppose que $\forall (u, v) \in A,\ p(u, v) \geq 0$

Dijkstra$(G, p, s) =$
  Initialisation(G, s)
  $E \leftarrow \emptyset$
  $Q \leftarrow S$
  tant que $Q \neq \emptyset$ **faire**
    $u \leftarrow$ Extraire-min(Q)
    $E \leftarrow E \cup \{u\}$
    pout tout $v \in Adj(u)$ **faire** Relaxation(u, v, p)

2.2 Preuve de sa correction :

Théorème : Quand Dijkstra termine, nous avons $d[v] = \delta(s, v)$ pour tout $v \in S$.
Nous prouvons l'invariant : $d[v] = \delta(s, v)$ pour tout $v \in S$.
Initialement : $E = \emptyset$, invariant trivialement vrai .
Soit u le premier sommet tel que $d[v] \neq \delta(s, v)$ au moment de son ajout à E
$s \neq u$ car $d[s] = \delta(s, s) = 0$. u est accessible de s, sinon $d[u] = \delta(s, u) = \infty$

Soit c un plus court chemin de s à u dans G.
Avant l'ajout de u à E, le chemin c relie un sommet de E à un sommet de $S\backslash E$. Soit y le premier sommet de c dans $S\backslash E$ et soit x son prédécesseur.
Nous prouvons que $d[y] = \delta(s,x)$ au moment de l'ajout de u à E :

- $x \in E$ et choix de $u \Rightarrow d[u] = \delta(s,x)$ quand x a été ajouté à E
- A ce moment, il y a appel de Relaxation(x,y,p) [propriété convergence]

c est un plus court chemin, les poids sont toujours positifs, c'est-à-dire que $\delta(s,y) \leq \delta(s,u)$, alors $d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u]$.
u et y sont dans $S\backslash E$ au moment du choix de u, donc $d[u] \leq d[y]$
$d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u] \leq d[y] \Rightarrow d[u] = \delta(s,u)$ [Contradiction]

2.3 Étude de sa complexité :

Nous savons que la file à priorité (tas) nous permet deux opérations d'insertion et de suppression en un temps logarithmique de $O(log_2 n)$ avec n le nombre d'éléments de la file.

Comme il est possible d'ajouter un nœud à la file autant de fois qu'il y a d'arcs qui le précédent, dans le pire des cas nous aurons m éléments dans la file. De même, nous devrons supprimer ces m éléments dans le pire des cas (si le dernier est celui qu'on cherche)

La complexité en temps est donc $O(m\ log\ m + m\ log\ m) = O(2m\ log\ m)$ et nous pouvons simplifier en $O(m\ log\ m)$.

Par ailleurs, nous avons la possibilité d'améliorer le nombre d'éléments dans la file à priorité en mettant à jour les valeurs des nœuds déjà enfilés, plutôt que de recréer des éléments en plus dans la file.
Cette opération s'effectue aussi en temps logarithmique, et nous permet de garder au maximum n éléments dans notre file. Enfin, nous avons donc dans le pire des cas n insertions, n suppressions, et m mises à jour d'éléments dans la file.
Ceci nous donne une nouvelle complexité en temps $O(2n\ log\ n + m\ log\ n)$ et que l'on peut simplifier en $O(m\ log\ n)$ (vu que nous aurons $m \geq n$ dans la plupart des cas)

2.4 Vérification de temps de calcul de l'implémentation :

```
void dijkstra tas(GRAPHE* graphe, int sommet src, int* predecesseur,double* poids)
{
    int u, v, n;
    double dist;
    TAS tas;
    ELEMENT TAS tmp;
    LISTE ADJACENCE NOEUD* arete;
    ETAT* etat = initialiser etat(graphe);

    tas.nombre elements = 0;
    tas.elements = NULL;

    for (int i = 0; i < graphe->nombre elements; i++)
    {
        if (i != sommet src)
            poids[i] = -1;
        else
            poids[i] = 0;

        predecesseur[i] = INT MAX;

        //ajouter le à la file de priorité
        tmp.u = i;
        tmp.distance = poids[i];
        ajouter element tas(&tas, tmp);

    }

    while (!tas est vide(&tas))
    {
        tmp = extraire min tas(&tas);
        u = tmp.u;
        marquer noeud(etat,u, DEJA VISITE);

        //verifier tous les arcs sortant de u
        n = graphe->LISTE ADJACENCE[u].nombre elements;
        arete = graphe->LISTE ADJACENCE[u].racine;

        for (int i = 0; i < n; i++)
        {
            v = arete->noeud;
            if (!get etat(etat, v, DEJA VISITE))
            {
                dist = poids[u] + arete->poids;
                if (dist < poids[v])
                {
                    poids[v] = dist;
                    predecesseur[v] = u;
                    tmp.u = v;
                    tmp.poids = dist;
                    mise a jour tas(&tas, tmp);
                }
            }
            arete = arete->suivant;
        }
    }

    desallouer etat(etat);
    desallouer tas(tas);
}
```

Intéressons-nous maintenant au temps de calcul de l'algorithme implémenté. Comme précédemment la fonction *ajouter_element_tas* se fait en temps logarithmique $O(log\ n)$, première boucle *for* parcourt le graphe $|S|$ fois, donc la première boucle se fait en $O(n\ log\ n)$.

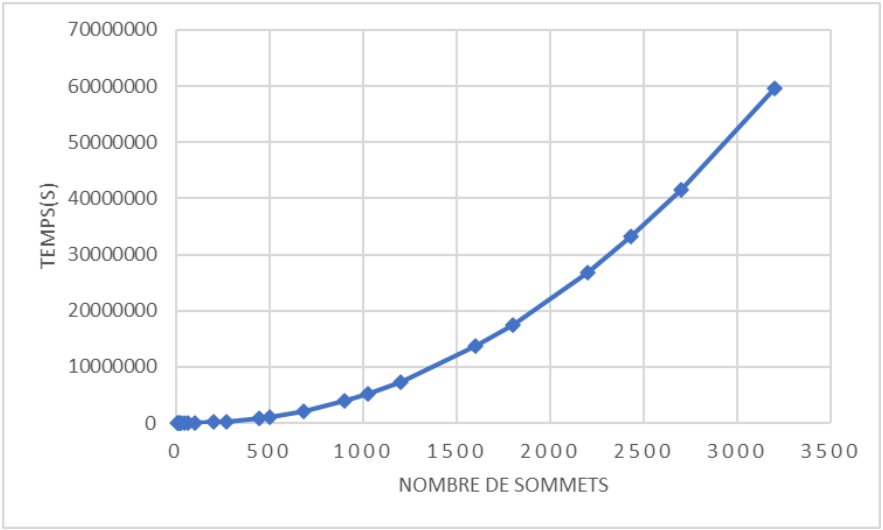
La fonction *extraire_min_tas* se fait en temps $O(|S|\ log|S|) = O(nlogn)$ avec $|S| = n$, et la deuxième boucle for "modification et mise à jour du tas" en $O(|A|\ log|S|) = O(m\ log\ n)$, avec $|A| = m$.

Alors la complexité de l'algorithme implémenté est en $O(nlog\,n\ +\ nlog\,n\ +\ mlog\,n)\ =\ O(2n\,log\,n\ +\ m\,log\,n)$, nous pouvons écrire cette complexité comme ceci : $O(mlog\,n)$ "$m\ \geq\ n$"

Nous allons désormais mettre en pratique le programme en lui donnant des nombres de sommets et des arêtes variables.

Considérons qu'on est toujours dans le pire des cas, donc le nombre d'arêtes est maximal $\frac{n(n-1)}{2}$,

n	temps
5	23,2192809
10	149,486764
15	410,223513
20	821,166338
40	4151,10391
60	10455,1964
100	32887,0881
200	152112,738
270	293309,558
440	848103,721
500	1118481,59
680	2172251,99
900	3970165,18
1024	5237760
1200	7358612,17
1600	13615620,8
1800	17508593,1
2200	26857742,9
2430	33191774,5
2700	41533032,5
3200	59597913,5



Partie 3 : L'algorithme de Dijkstra avec tas de Fibonacci

3.1 Présentation théorique :

Nous pouvons encore améliorer la complexité en temps si l'on utilise une variante de la file à priorité : *le tas de Fibonacci*. Cette structure est similaire au tas binomial, mais avec un meilleur temps d'exécution amorti.
Le nom de tas de Fibonacci vient des nombres de Fibonacci, qui sont utilisés pour calculer son temps d'exécution.

Un tas de Fibonacci est un ensemble d'arbres satisfaisant la propriété de tas-minimum, c'est-à-dire que la clé d'un fils est toujours supérieure ou égale à la clé de son père. Ceci implique que la clé minimum est toujours à la racine d'un des arbres. Cette structure est plus flexible que celle des tas binomiaux, en ce sens qu'elle permet à quelques opérations d'être exécutées de manière paresseuse, en reportant le travail sur des opérations ultérieures. Par exemple, l'union de deux tas est effectuée simplement en concaténant les deux listes d'arbres, et l'opération de diminution de clé coupe parfois un nœud de son père pour former un nouvel arbre.

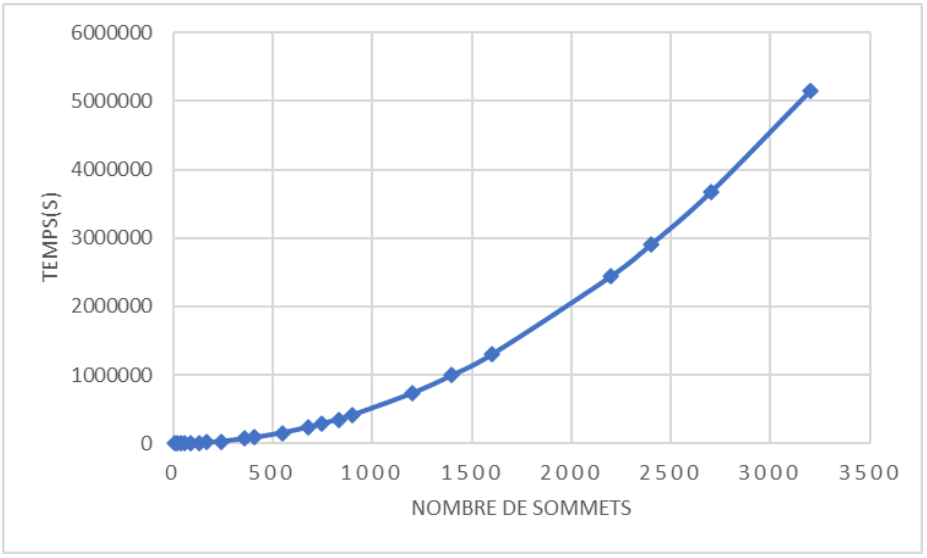
La structure des tas de Fibonacci a un temps constant $O(1)$ pour l'insertion, trouver le minimum, décroître une clé et la mise à jour d'éléments. Les opérations supprimer et supprimer le minimum ont un coût amorti en $O(log\, n)$, ce qui nous donne une complexité finale de $O(n + n\, log\ n + m)$. Supposons que $m \geq n$, nous pouvons écrire cette complexité comme ceci : $O(n\, log\ n + m)$.

3.2 Vérification de temps de calcul de l'implémentation :

Notre algorithme a une complexité en $O(n\ log\ n + m)$. Exécutons maintenant le programme sur un nombre de sommets qui augmente en considérant que le nombre des arêtes est toujours maximal.

```
void dijkstra tas fib(GRAPHE* graphe, int sommet src, int* predec, doouble* poids)
{
    LISTE ADJACENCE NOEUD* arete;
    ELEMENT TAS FIBONACCI* tmp;
    TAS FIBONACCI* tas fib = creer tas fibonacci();
    double dist;
    int u, v, n, x;
    ETAT* etat = initialiser etat(graphe);

    for (int i = 0; i < graphe->nombre elements; i++)
    {
        if (i != sommet src)
            poids[i] = -1;
        else
            poids[i] = 0;

        predec[i] = INT MAX;
    }

    marquer noeud(etat,u, DEJA VISITE);
    //ajouter les noeuds dest de tous les arcs sortant de sommet src au tas de fib
    arete = graphe->LISTE ADJACENCE[sommet src].racine;
    n = graphe->LISTE ADJACENCE[sommet src].nombre elements;

    for (int i = 0; i < n; i++)
    {
        dist = poids[sommet src] + rete->poids;
        v = arete->noeud;
        inserer tas fib(tas fib, nouvel element tas fib(dist, sommet src, v));

        arete = arete->suivant;
    }

    while (! tas fib est vide(tas fib))
    {
        tmp = extraire min tas fib(tas fib);
        u = tmp->from;
        v = tmp->to;
        dist = tmp->cle;

        if (!get etat(etat, v, DEJA VISITE)
        {
            poids[v] = dist;
            predec[v] = u;
            marquer noeud(etat, v, DEJA VISITE);

            arete = graphe->LISTE ADJACENCE[v].racine;
            n = graphe->LISTE ADJACENCE[v].nombre elements;

            for (int i = 0; i < n; i++)
            {
                x = arete->noeud;
                dist = poids[v] + arete->poids;
                inserer tas fib(tas fib, nouvel element tas fib(dist, v, x);

                arete = arete->suivant;
            }
        }
    }
    desallouer etat(etat);
    desallouer tas fib(tas fib);
}
```

n	temps
10	78,2192809
20	276,438562
40	992,877124
60	2124,41344
90	4589,26678
130	9297,90782
170	15624,5965
240	30577,6537
360	67677,0671
410	87403,5868
550	155981,808
680	237258,386
745	284248,117
830	352083,483
900	413382,403
1200	731674,582
1400	993931,696
1600	1296230,17
2200	2443327,23
2400	2905749,16
2700	3674426,61
3200	5155660,34



Partie 4 : Comparaisons des algorithmes

Dans cette partie, nous allons comparer la complexité des principaux algorithmes connus.

U : désigne le maximum de coûts des arcs.
* : Complexité au pire, mais complexité moyenne $O(M + U)$

Graphe	Algorithme	Complexité
$W \geq 0$	Dijkstra, version sans tas	$O(N^2)$
$W \geq 0$	Dijkstra, avec tas	$O(M.logN)$
$W \geq 0$	Dijkstra avec tas de Fibonacci	$O(N.LogN + M)$
$W \geq 0$	Algorithmes à buckets	$O(U.N^2)$ *
Sans circuit	Bellman + tri topologique	$O(M)$
Quelconque	Bellman	$O(N.M)$
Quelconque	Algorithme FIFO	$O(N.M)$

Tableau résume les principaux algorithmes

Conclusion

L'algorithme de Dijkstra est donc un algorithme glouton de recherche de plus court chemin dans un graphe pondéré positivement. L'algorithme en lui-même pose principalement deux problèmes :

1. Il ne peut pas être utilisé sur des graphes pondérés négativement. Dans ce cas, il faut avoir recours à d'autres algorithmes de recherche de plus court chemin, comme celui de ***Bellman-Ford***.

2. Il ne fait aucunes différences entre les multiples chemins qu'il emprunte, c'est-à-dire qu'il ne va jamais en favoriser un par rapport à un autre.
   En effet, ceci peut être un problème, car en fonction du graphe en entrée, si un chemin semble optimal au début il est très probable qu'il soit, ou mène, au plus court chemin. Et c'est ce qu'essaie d'améliorer la variante de l'algorithme appelée : ***A****.

Bibliographie

1. Comparaison d'algorithmes de plus courts chemins sur des graphes routiers de grande taille.
   http://www.numdam.org/item?id=RO_1996__30_4_333_0

2. Algorithme de Dijkstra : terminaison, correction et complexité
   Julie Parreaux 2018-2019

3. Théorie des graphes et optimisation dans les graphes
   Christine Solnon

4. Optimisation de l'utilisation de l'algorithme de Dijkstra pour un simulateur multi-agents spatialisé

5. Rivest R. Stein C. Cormen T., Leiserson C. Algorithmique, 3ème édition. Dunod, 2010

6. https://github.com/mburst/dijkstras-algorithm