Graphes et Application

Projet - Plus court chemin

Année 2021- 2022

réalisé par

ZAROUI Farès et GUYOT DE LA POMMERAYE Benjamin

I. Choix du langage et importation du Graph

1. Langage

Nous avons naturellement choisi de travailler avec le langage de programmation Java. En effet nous avions au préalable réalisé un TP d'initialisation aux graphes sous Java. C'est donc naturellement que nous nous sommes tournés vers ce dernier.
De plus Java possède de nombreuses librairies comme une pour obtenir des tas de fibonacci ou des skip lists. Ce qui nous est très utile dans le cadre de notre projet.
Enfin Java est le seul langage orienté objet que nous connaissons bien et nous ne voulions pas en apprendre un nouveau.

2. L'importation du Graph

Nous avons décidé de structurer notre graphe en trois classes : Edge, Vertex et Graph.

Edge: Représente un arc du graphe. Elle comporte une valeur (le poids de l'arc) et le sommet de l'arc.

Vertex: Représente un sommet du graphe. Elle a comme information le nom de la ville, sa latitude, sa longitude, sa population et son indice au sein du graph.

Graph: Il s'agit de la classe principale de notre projet. Elle comporte le type du graphe (qu'il soit orienté ou non), la liste des sommets et des arcs qui le composent ainsi qu'une liste d'adjacences. Elle comporte aussi les différents algorithmes et méthodes à implémenter ainsi que leur dépendance.

II.Algorithme de Dijkstra

1. Préambule

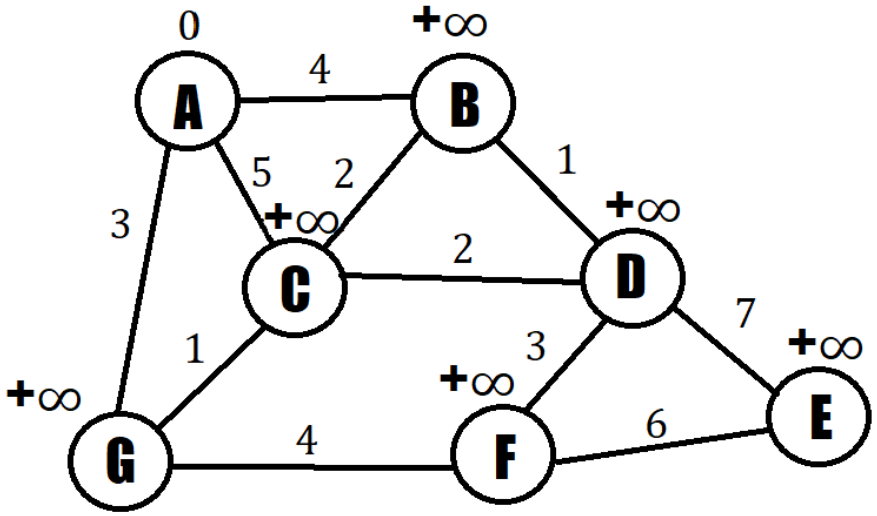
Pour cette première version, nous nous sommes appuyés sur le pseudo-code fourni par notre professeur dans ses diapositives. La valuation des arc correspond à des distances, elles sont donc forcément positives. Nous n'avons donc pas à vérifier si le graphe contient des valeurs négatives et sommes sur que nous pouvons utiliser cet algorithme sans problème.
Nous avons rencontré quelques problèmes avec cet algorithme cependant. En effet, dû à une mauvaise initialisation du sommet de départ, nous n'avions pas le bon chemin à un

sommet près. Nous avions donc des écarts de distances avec d'autres groupes allant de 0.1km à 1.5 km dans le pire des cas.

2. Déroulement

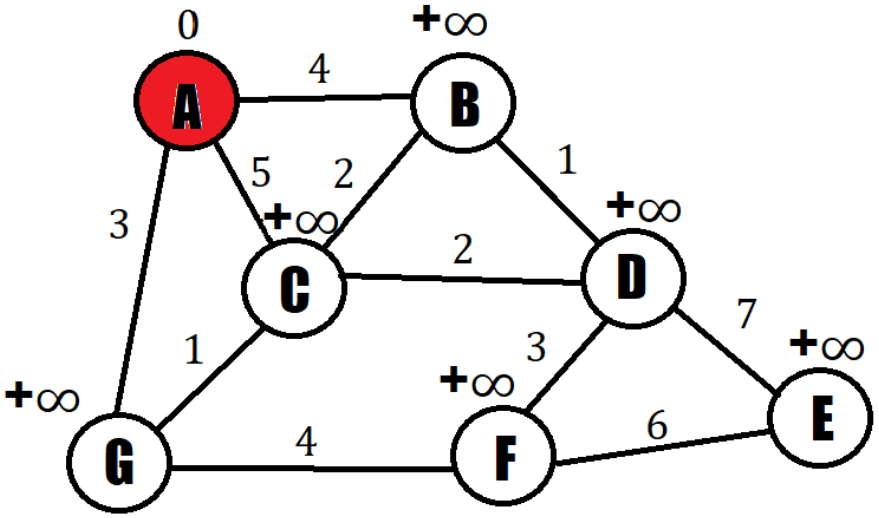
Première étape :

Mettre tous les sommets dans le HashSet (Dictionnaire et les initialisé à l'infini), le sommet de départ lui est initialisé à 0.



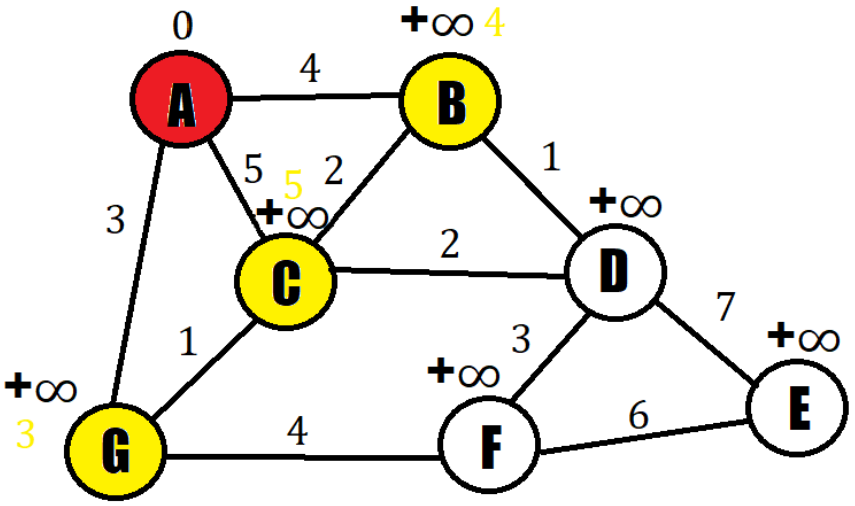
Seconde étape :

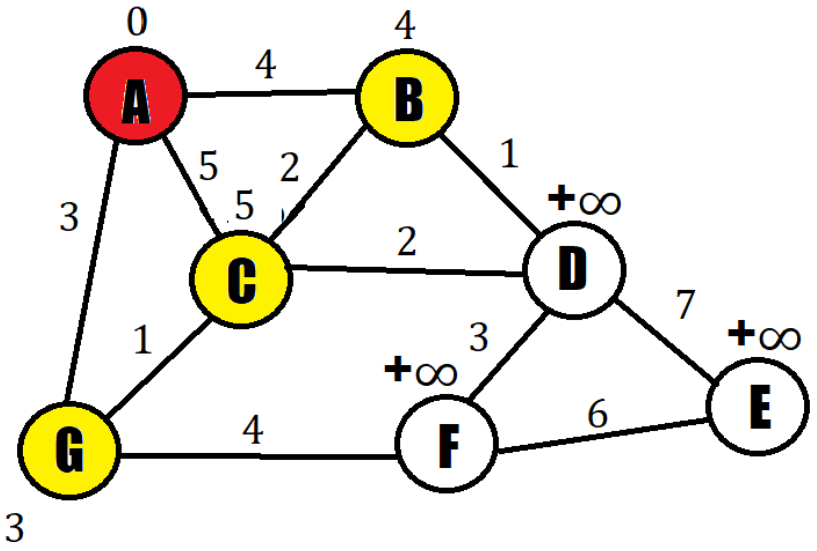
Prendre le sommet X avec le plus faible coefficient. (en rouge)



Troisième étape :

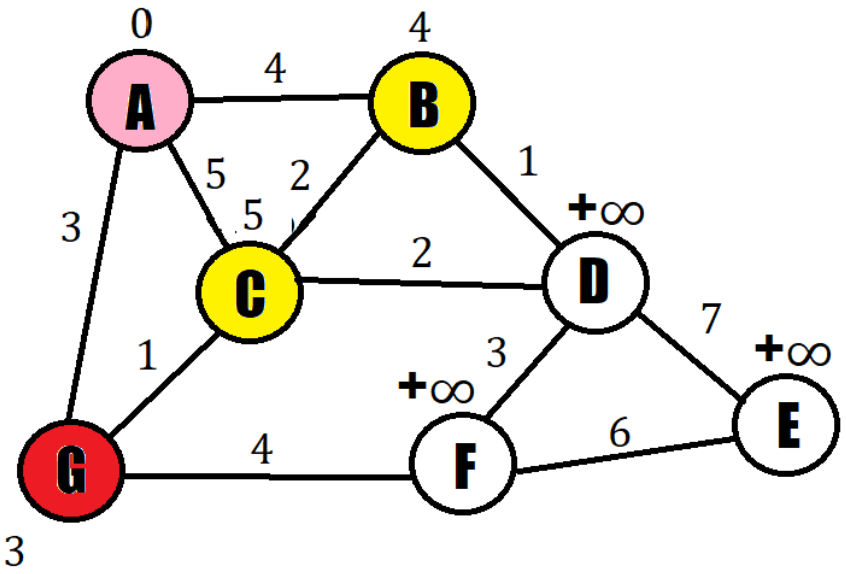
Pour chaque sommet Y (en jaune) adjacent à X calculer la somme du coefficient de X et du poids de l'arête le reliant à Y. Si la somme est inférieure au coefficient de Y, il faut l'inscrire sur le sommet correspondant sinon nous le laissons à son ancienne valeure





Quatrième étape :

Nous marquons le sommet X comme visité (en rose). Nous recommençons à l'étape 2 jusqu'à ne plus pouvoir visité de sommet.



Si nous continuions l'algorithme nous trouverions le chemin le plus court de A à E.

3. Complexité

Première version

Nous réalisons une première boucle de parcours de tous les sommets du graphe. Dans cette même boucle nous re-parcourons ces sommets pour trouver le sommet non traité de valeur minimum. Nous avons donc une complexité en O(n^2)

Deuxième version

Nous remarquons que la complexité de l'algorithme peut être réduite si nous avons à trouver une méthode plus efficace de recherche de minimum. Pour ce faire nous allons utiliser un tas de Fibonacci. Nous pouvons alors réduire la complexité de notre algorithme à O(nlog(n)). Pour l'implémentation, nous remplaçons simplement le tableau contenant les distances par un tas de Fibonacci .

Un tas de Fibonacci est une structure de donnée qui utilise un ensemble d'arbre satisfaisant les propriétés suivantes:

- tas-minimum c'est à dire le parent est toujours plus petit que l'enfant
- un pointeur sur le minimum du tas (important pour Dijkstra)
- il possède un marquage dans ses noeuds
- les arbres ne sont pas ordonnés mais enraciné de manière à ce que chaque sous-arbre soit An-1 et An-2 comme dans une suite de Fibonacci

Le tas de Fibonacci a pour avantage d'avoir toujours le minimum en pointeur ce qui augmente grandement la recherche du minimum ou du plus petit élément dans le Dijkstra. De plus, les opérations de suppression d'une clé ou de valeurs sont en o(log(n)).

Test des deux Dijkstra Paris Marseille

Algo	Taille du Graph	Résultat en seconde
Dijkstra_simple	5000	10.127
Dijkstra_Fibo	5000	0.139
Dijkstra_simple	10000	42.69
Dijkstra_Fibo	10000	0.298

Nous remarquons que le Dijkstra simple met en général bien plus de temps et sa complexité et bien supérieur à celle du Dijkstra_fibo.
En effet pour le Graph de 5000 villes il met déjà 10 secondes complète et si nous doublons le nombre de sommets dans le graph le temps de traitement quadruple ce qui montre une voie d'amélioration dans notre algorithme. Cependant lorsque nous ajoutons un tas de fibonacci nous obtenons une vitesse 11 fois supérieure à notre Dijkstra simple. Cela vient du fait que la recherche de minimum n'est plus " à faire " dans l'étape 2 car le tas de Fibonacci renvoie le minimum en O(1) contrairement à un HashSet qu'il faut parcourir à chaque itération pour trouver son minimum.

III. Algorithme A*

1. Préambule

Un Astar est un Dijkstra ou nous prenons en compte une heuristique. L'objectif est d'éviter de calculer des sommets inutiles dans le graph et diminuer ainsi le temps de traitement pour trouver le plus court chemin. Il est simple et dépend de son heuristique. Il améliore la vitesse de calcul mais perd en exactitude du résultat. Il est donc très utilisé dans les jeux vidéo qui privilégie la vitesse de pathfinding à l'exactitude du résultat.

Fonctions annexes

Import des données

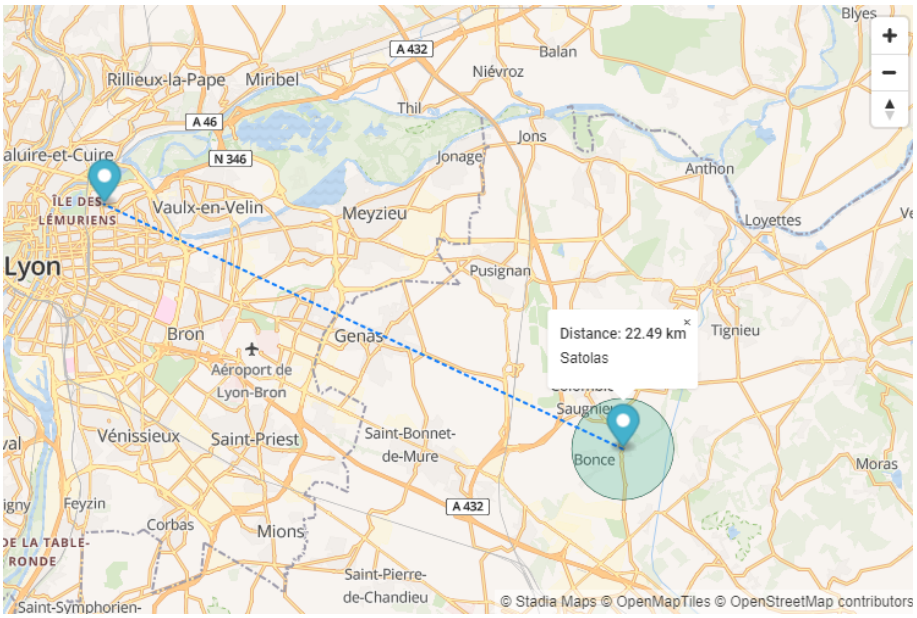
Pour l'import des coordonnées nous avons dû modifier légèrement notre fonction d'import de graphe. A l'inverse des coordonnées GPS, la population de chaque ville se trouve non pas dans le fichier d'import du graphe mais dans un fichier CSV différent. Nous avons donc réalisé une sous-fonction qui lit le fichier CSV et associe à chaque ville sa population

Calcul de la distance " à vol d'oiseaux (Orthodromie)"

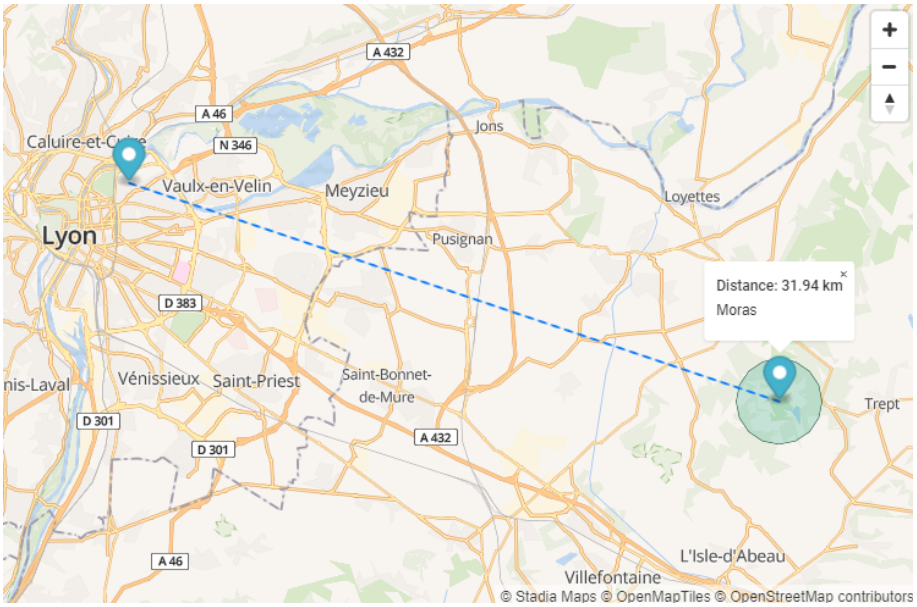
Pour le calcul des Distance à vol d'oiseaux (Orthodromie) nous avons réalisé une sous-fonction qui prend en argument l'indice de deux sommets dans le graph et renvoie la distance à vol d'oiseaux. Nous avons décidé, pour calculer la distance à vol d'oiseaux, de prendre en compte la courbure de la terre.
En effet Calculer cette distance en prenant en compte la courbure de la terre revient à calculer un arc de grand cercle.

2. Déroulement

Nous voulons calculer le chemin optimal entre Lyon et Satolas. Ces deux communes sont distantes à vol d'oiseau à 22.49 km.



Cette distance va nous permettre d'éliminer des sommets qui créeraient des chemins "non pertinents", par exemple Moras qui a une distance par rapport à Polytech Lyon supérieur à 22.49 km ne sera pas traité par notre Dijkstra améliorer par l'heuristique. Nous avons alors réduit le nombre de sommets traités.



3. Les test Paris Marseille

Pour ces tests nous avons pris les villes de Paris et Marseilles et les avons comparé aux précédent résultats des algorithmes de Dijkstra et Dijkstra Fibonacci:

Algorithme	Taille du Graphe	Résultat en seconde
Dijkstra	5000	10.127
Dijkstra_Fibo	5000	0.139
A*	5000	0.132
A*Fibo	5000	0.042
Dijkstra	10000	42.69
Dijkstra_Fibo	10000	0.298
A*	10000	0.372
A*Fibo	10000	0.073

Nous remarquons que le A* est aussi rapide que le Dijkstra avec tas de Fibonacci et n'augmente pas exponentiellement lors d'un graph plus grands. Cependant on a réussi à améliorer notre A* en lui mettant un tas de Fibonacci pour minimiser la recherche de minimum.

4. Complexité

III. VRP : Meilleur positionnement

1. Préambule

Notre objectif est de trouver une ville parmis les 5000 du 3ème fichier tels que sa distance aux villes de plus de 200 000 habitants soit minimal. Pour répondre à ce problème, nous avons découpé le problème en plusieurs étapes.

2. Déroulement de la recherche

Recherche des grandes villes

Dans un premier temps, nous avons parcouru l'ensemble des sommets du graphe et avons stocké les sommets de plus de 200 000 habitants dans un tableau. Pour ce faire, nous utilisons une simple fonction de comparaison. Nous utilisons une sous-fonction dans l'objectif de pouvoir la réutiliser dans la question suivante.

Recherche de la distance moyenne

Nous prenons une ville du graphe. Nous utilisons l'algorithme de Dijkstra pour connaître la plus petite distance entre cette ville et chaque autre ville du graphe. Nous extrayons par la suite la distance entre la ville de départ et chaque grande ville. De cette façon, nous ne réalisons qu'un dijkstra pour chaque ville.
Nous réalisons la moyenne de ces distances et les stockons dans une Hashmap avec pour clé le sommet et en valeur la moyenne des distances aux grandes villes.
Nous répétons cette opération pour chaque ville du graphe. Une fois tous les sommets parcourus, nous avons normalement une Hashmap contenant une moyenne des distances pour chaque ville.

Recherche de la plus petite distance

Une fois notre Hashmap remplie, nous n'avons plus qu'à faire une simple recherche de minimum dans cette dernière. Nous obtenons après exécution du programme :

```
la plus petite ville est : moulins-03avec une moyenne de 414.7763636363636 km
```

3. Complexité

Nous réalisons une première boucle parcourant l'ensemble des sommets du graphe. Dans cette boucle nous réalisons ensuite un algorithme de Dijkstra avec Tas de Fibonacci. Nous réalisons alors n algorithme de Dijkstra. Nous obtenons alors une complexité en O(n*nlog(n)) avec n le nombre de sommet du graphe.

IV. VRP : Meilleur circuit

1. Préambule

Notre objectif est de trouver le trajet le plus court entre les villes de plus de X habitants en passant une et une seule fois par chaque ville. Cela revient à chercher le chemin hamiltonien de poids minimum sur le sous graphe composé des villes de plus de X

habitants. Nous savons que ce problème est NP-complet. Nous allons donc essayer de trouver une solution se rapprochant de la solution optimale.

2. Déroulement de la recherche

Création du sous graphe

Nous réalisons un sous graphe ayant pour sommet les villes de plus de X habitants. Nous réalisons un graphe complet. Pour symboliser les arêtes, nous utilisons l'algorithme de Dijkstra pour trouver la distance entre chaque ville. Nous utilisons toutes ces données pour réaliser une liste d'adjacences.

Recherche d'un circuit hamiltonien de poids faible

Pour parcourir le graphe nous utilisons la méthode suivante :

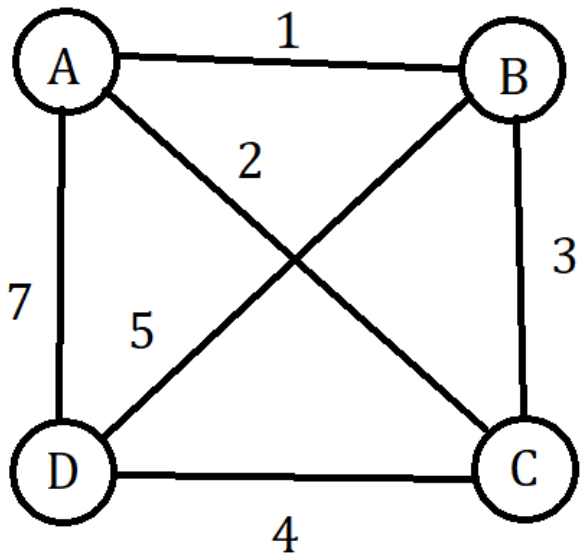
Nous regardons les sommets disponibles à partir de notre sommet. Un sommet n'est pas disponible s'il a déjà été visité.

Nous cherchons le sommet le plus proche via la liste d'adjacence.

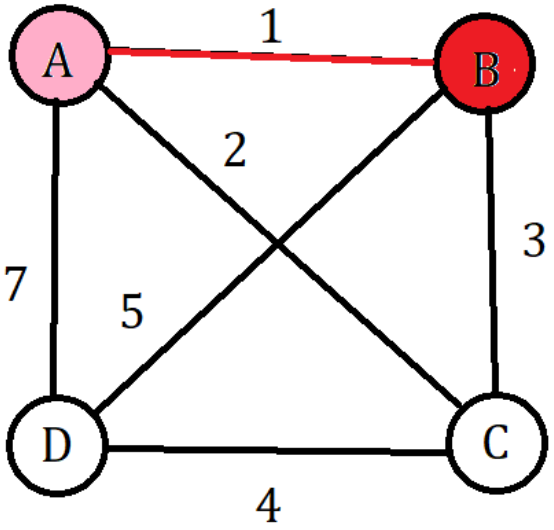
Nous nous rendons à ce sommet et marquons le sommet que nous venons de quitter comme visité.

Une fois tous les sommets visités, nous retournons au sommet de départ.

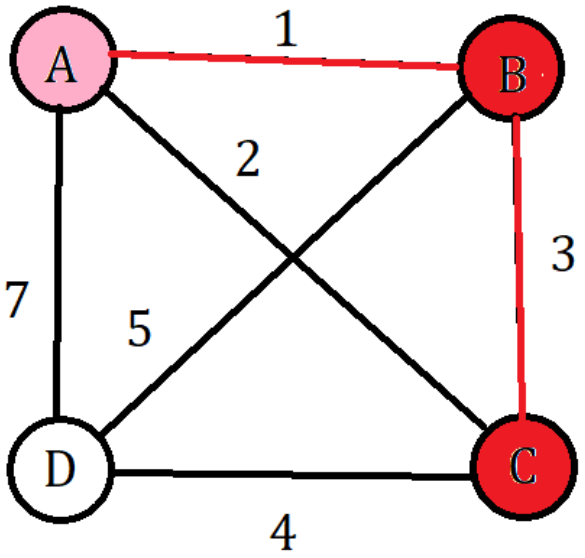
Illustration de cette méthode de parcours sur le graphe k4:



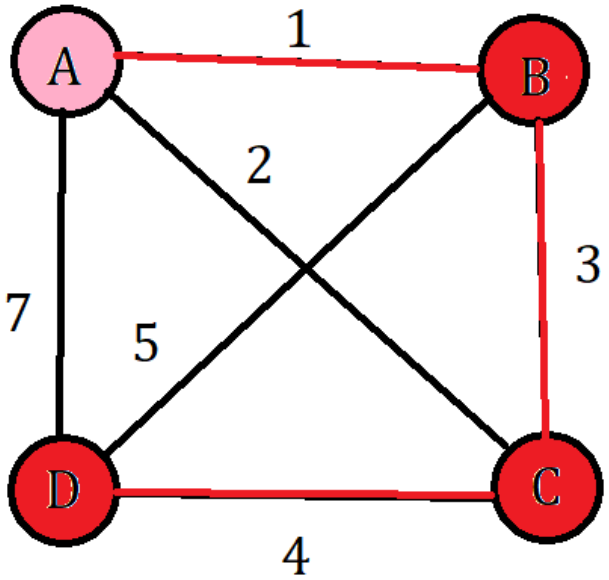
Initialisation du graphe. Pour ce parcours nous partirons du sommet A. Nous recherchons le plus petit arc de A. Nous trouvons que cet arc est celui reliant A et B. Nous marquons donc A comme visité et nous allons au sommet B.



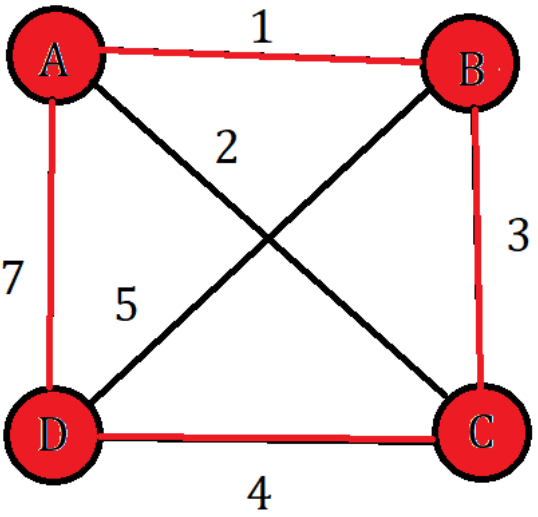
Une fois arrivé en B, nous regardons les chemins les plus courts de B, en excluant l'arc menant à A. Cet arc correspond à la liaison entre B et C. Nous marquons B comme visité et nous allons en C.



Une fois en C, nous remarquons que la seule option est de se rendre en D car les sommets A et B sont déjà marqués. Nous marquons donc C comme visité et allons en D.



Une fois en D, nous retournons au Sommet de départ pour fermer notre circuit hamiltonien



Après notre parcours nous obtenons le chemin ci-dessus avec un poids de 15. Nous remarquons qu'il ne s'agit pas du chemin optimal (A -> B -> D -> C -> A avec un poids de 14). Cependant ce chemin n'est pas le pire, qui serait A -> D ->B -> C -> A avec un poids de 18.

Nous supposerons que sur de plus gros graphes, notre solution sera encore proche de la solution optimal.

Comparaison de plusieurs circuits hamiltonien

Pour obtenir plusieurs circuits possibles, nous réalisons la recherche de circuit hamiltonien en partant de chaque sommet du graphe. De la sorte, nous obtenons un nombre de circuits hamiltonien égale au nombre de sommets. Une fois ces circuits obtenus, nous conservons celui avec le poids minimum.
En se limitant à un circuit hamiltonien par sommet, nous évitons d'avoir une complexité algorithmique trop élevée, tout en ayant plusieurs valeurs à comparer.

3. Application de l’algorithme

Nous allons utiliser notre algorithme sur des villes de plus de 200 000, 150 000, 100 000 habitants

Recherche du court chemin pour les villes de plus de 200 000 habitants

```
Affichage de la route :


toulouse -> montpellier -> marseille -> nice -> lyon ->
strasbourg -> lille -> paris -> rennes ->
nantes -> bordeaux -> toulouse


Le poids de ce circuit est de 2861.24 km.
temps d'execution (en ms) : 18671
```

Pour 11 villes nous obtenons un temps d'exécution d’environ 19 secondes

Recherche du court chemin pour les villes de plus de 150 000 habitants

```
Affichage de la route :


nice -> toulon -> marseille -> montpellier -> toulouse ->
bordeaux -> nantes -> rennes -> havre ->
paris -> reims -> lille -> strasbourg ->
dijon -> lyon -> saint-etienne -> grenoble ->
nice


Le poids de ce circuit est de 3240.38 km.
temps d'execution (en ms) : 53778
```

Pour 17 villes, nous obtenons un temps d'exécution d’environ 54 secondes. Par rapport aux chemin des villes de plus de 200 000 habitants, nous avons un écart de 35 secondes. Cela correspond à 5 secondes en plus de temps de traitement par ville additionnelle.

Recherche du court chemin pour les villes de plus de 100 000 habitants

```
Affichage de la route :


havre -> caen -> rouen -> argenteuil -> saint-denis-93 ->
paris -> montreuil-93 -> boulogne-billancourt -> orleans ->
tours -> le-mans -> angers -> nantes ->
rennes -> brest -> bordeaux -> limoges ->
toulouse -> perpignan -> montpellier -> nimes ->
aix-en-provence -> marseille -> toulon -> nice ->
grenoble -> villeurbanne -> lyon -> saint-etienne ->
clermont-ferrand -> dijon -> besancon -> mulhouse ->
strasbourg -> nancy -> metz -> reims ->
amiens -> lille -> havre


Le poids de ce circuit est de 4873.050000000001 km.
temps d'execution (en ms) : 608584
```

Pour 39 villes nous obtenons un temps d'exécution d'environ 10 mins et 9 secondes. Par rapport au chemin des villes de plus de 150 000 habitants, nous avons un écart de 553 secondes soit 9 mins et 13 secondes. Cela correspond à 25 secondes de traitement additionnel par ville.

Bien que nous n'ayons pas la certitude d'avoir le plus petit résultat, nous rappelons que notre temps de calcul est faible.
En effet, pour 19 villes il faudrait (en supposant qu'un chemin soit traité en une microseconde) 1 * 10e13 micro secondes soit environ 121 jours de traitement.
Pour 39 villes il faudrait 8 * 10e30 années. Ce temps de calcul dépasse très largement l'âge de l'univers (1.3*10e9 années).