SAÉ ALGO 4

Problème du plus court chemin

> What's the shortest way to travel from Rotterdam to Groningen ? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path. As I said, it was a 20-minute invention. In fact, it was published in 1959, three years later. The publication is still quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. Without pencil and paper you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame.
>
> — Philip L Frana et Thomas J Misa : *An interview with Edsger W. Dijkstra.* In : *Communications of the ACM* 53.8 (2010), p. 41-47.

1 Introduction et problématique

Les graphes sont des structures mathématiques utilisées pour modéliser des relations entre des objets. Ils sont largement employés en informatique et plusieurs autres domaines.Ils permettent de représenter des situations variées comme les réseaux de transport, les circuits électriques ou encore les connexions sur les réseaux sociaux.

1.1 Définition d'un Graphe

Un *graphe* est un couple $G = (V, E)$ comprenant deux ensembles :
— V : ensemble des sommets (fini ou infini) ;
— E : ensemble des arêtes, où chaque arête est associée à un couple de paire de sommets $(u, v) \in V \times V$.

Deux sommets u et v sont dits *adjacents* ou *voisins* si $(u, v) \in E$ ou $(v, u) \in E$.

Un graphe est dit *non orienté* lorsque ses arêtes n'ont pas de direction. Cela signifie que si une arête existe entre deux sommets u et v, alors elle est valable dans les deux sens :

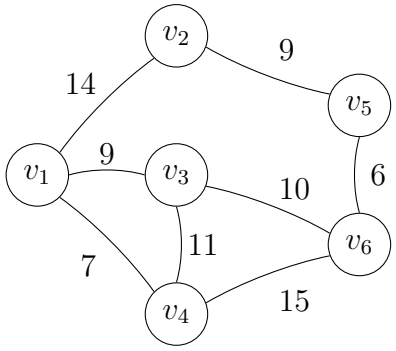
$$\forall u, v \in V : (u, v) \in E \Rightarrow (v, u) \in E.$$

Un *multi-graphe* est un graphe dans lequel il peut exister plusieurs arêtes distinctes reliant deux mêmes sommets. L'ensemble E des arêtes est alors un multi-ensemble.

Un *graphe simple* est un graphe qui ne contient ni boucles ni multi-arêtes. Autrement dit, aucun sommet n'est connecté à lui-même et entre deux sommets donnés, il n'existe qu'une seule arête.

Enfin, un *graphe pondéré* (ou valué) est un graphe dans lequel chaque arête (et parfois chaque sommet) se voit attribuer une valeur numérique, appelée *poids*. Ce poids peut représenter une distance, un coût ou une durée, etc.

Graphiquement, les sommets peuvent être représentés par des points, et chaque arête ou arc par un trait ou une flèche reliant deux sommets. Par exemple :



1.2 Problème du Plus Court Chemin

Un *chemin* dans un graphe est une suite de sommets tels que chaque sommet de la suite est adjacent au suivant. La longueur d'un chemin est définie par le nombre d'arêtes qui le composent (ou la somme des poids de ces arêtes, dans le cas d'un graphe pondéré).

Le problème du *plus court chemin* en théorie des graphes, consiste étant donné un graphe $G = (V, E)$, un sommet source s, un sommet destination d, où $s, d \in V$, et des poids sur les arêtes de E, à trouver le chemin le plus court entre s et d ayant le poids minimal. Les arêtes peuvent être orientées ou non orientées. Elles peuvent avoir des poids explicites ou des poids implicites considérés comme étant égaux à 1.

Nous nous focalisons, dans ce projet, sur la variante classique du problème du plus court chemin à source unique, appelée *Single-Source Shortest-Path*. Formellement, étant donné un graphe $G = (V, E)$ et une source $s \in V$, l'objectif est de déterminer un chemin de longueur minimale à partir du sommet s vers tous les autres sommets du graphe. Ce problème trouve de nombreuses applications pratiques, notamment en optimisation de réseaux, en planification de trajets ou encore en télécommunications.

La recherche en largeur *breadth-first search* peut être employée en commençant la recherche à partir du sommet source et en inspectant tous les sommets voisins. Pour chaque sommet voisin, on explore les sommets non visités jusqu'à ce que le chemin avec le nombre minimal d'arêtes entre le sommet source et le sommet de destination soit identifié.

L'objectif de ce projet est d'explorer et d'implémenter deux algorithmes classiques destinés à résoudre ce problème, en analysant leurs performances à la fois théoriquement et empiriquement, en fonction de la taille du graphe, mesurée par le nombre de sommets et d'arêtes.

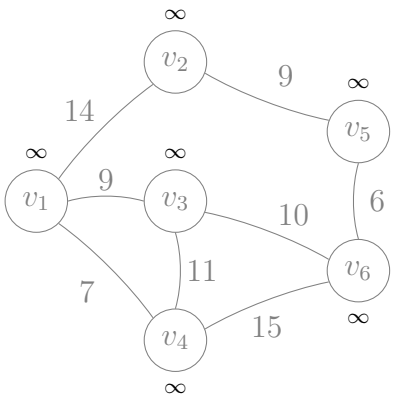
2 Algorithme de Dijkstra

L'algorithme de Dijkstra est une méthode classique pour résoudre le problème du plus court chemin dans un graphe pondéré avec des poids non négatifs. Il permet de déterminer la distance minimale entre un sommet source donné et tous les autres sommets du graphe. Le pseudo-code suivant décrit les étapes principales de l'algorithme :

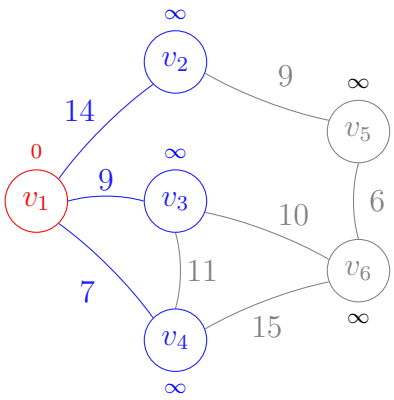
Algorithme 1 : Algorithme de Dijkstra

Entrées : Graphe $G = (V, E)$, sommet source s
Sorties : Distances minimales de s vers chaque sommet
pour *chaque sommet* $v \in V$ **faire**
  $dist[v] \leftarrow \infty$;
  $precedent[v] \leftarrow$ NULL;
fin
$dist[s] \leftarrow 0$;
FileDePriorité $F \leftarrow$ {tous les sommets v avec priorité $dist[v]$}
tant que F *n'est pas vide* **faire**
  $u \leftarrow$ sommet de F avec la plus petite distance;
  Retirer u de F;
  pour *chaque voisin* v *de* u **faire**
    si $dist[u] + poids(u, v) < dist[v]$ **alors**
      $dist[v] \leftarrow dist[u] + poids(u, v)$;
      $precedent[v] \leftarrow u$;
      Mettre à jour la priorité de v dans F;
    fin
  fin
fin
retourner *precedent*

Voici un exemple d'application de l'algorithme ci-dessus avec comme sommet source v_1 :



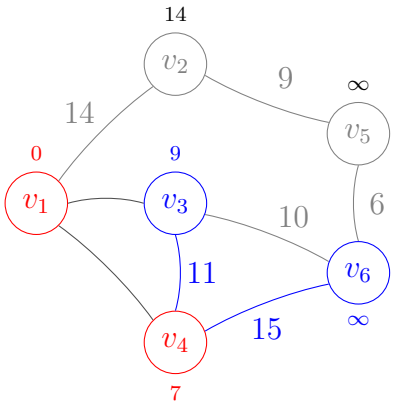
À l'initialisation, la file de priorité contient tous les sommets, chacun ayant une priorité (distance à v_0) initiale égale à $dist[v_i] = \infty$.



On met ensuite $dist[v_1] = 0$. Le sommet v_1 est retiré de la file et on examine chacun de ses voisins et met à jour leurs distances si une distance plus courte est trouvée. On a alors :

$$dist[v_2] = 14 \qquad precedent[v_2] = v_1$$
$$dist[v_3] = 9 \qquad precedent[v_3] = v_1$$
$$dist[v_4] = 7 \qquad precedent[v_4] = v_1$$

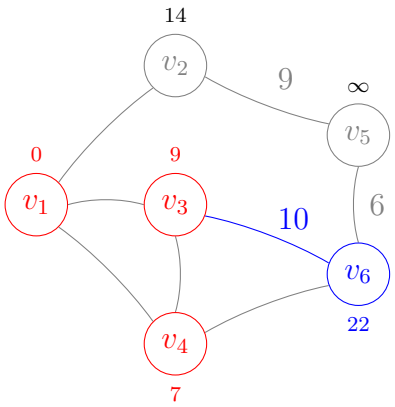
On identifie ensuite le sommet ayant la distance minimale. On trouve v_4 qui retiré de la file (marqué en rouge).



On examine tous les voisins de v_4 et applique le même procédé. On constate que $dist[v_3] = 9$ est déjà inférieure à $7+11$, donc elle ne change pas. En revanche, la distance jusqu'à v_6 est mise à jour.

$$dist[v_6] = 22 \qquad precedent[v_6] = v_4$$

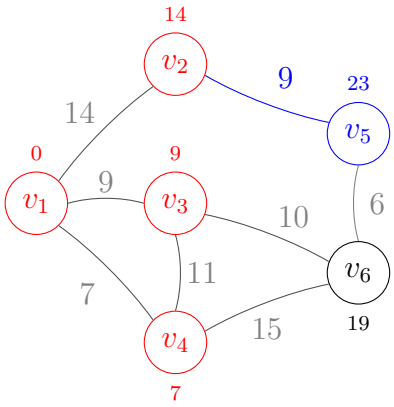
On identifie ensuite le sommet ayant la distance minimale. v_3 est donc retiré de la file.



Puis, comme précédemment, on examine tous les voisins de v_3 qui n'ont pas encore été retirés de la file. On constate que $dist[v_6] = 22$ est supérieure à $9 + 10$.

$$dist[v_6] = 19 \qquad precedent[v_6] = v_3$$

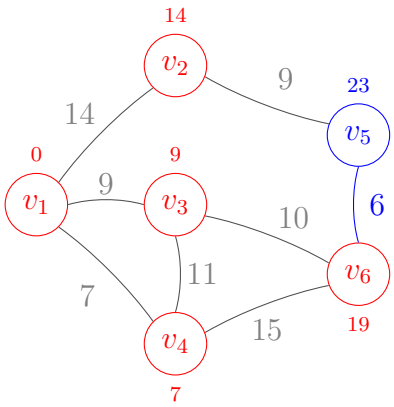
On identifie ensuite le sommet ayant la distance minimale dans la file. On trouve v_2 qui est retiré de la file.



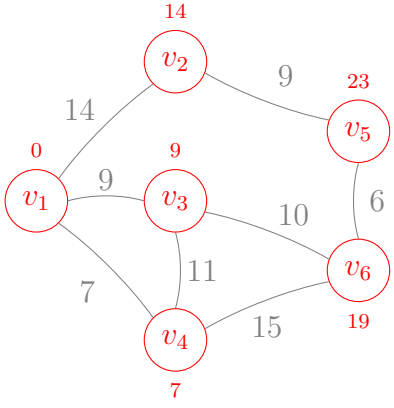
Pareillement, on examine tous les voisins de v_2, met à jour leurs distances depuis v_1.

$$dist[v_5] = 19 \qquad precedent[v_6] = v_2$$

On identifie le sommet ayant la distance minimale. v_6 est sélectionné et retiré de la file.



On examine finalement tous les voisins de v_6. On a $dist[v_5] = 23 < 19 + 6$. On sélectionne v_5 étant le dernier sommet de la file, celui-ci est retiré de la file et l'algorithme s'arrête.



1. Implémenter l'algorithme de Dijkstra en Python : Représentez le graphe sous forme d'un dictionnaire d'adjacence, où chaque clé correspond à un sommet et chaque valeur associée est une liste de tuples, chacun indiquant un voisin et le poids de l'arête correspondante. Pour extraire efficacement le sommet présentant la plus petite distance, utilisez une file de priorité qui devra être implémentée en deux versions : d'abord à l'aide d'un tas binaire, puis en exploitant un tas de Fibonacci.
2. Analyse théorique de la complexité : Déterminez, pour chacune des implémentations, une borne asymptotique supérieure $\mathcal{O}$ du temps d'exécution en fonction du nombre de sommets et d'arêtes. Justifiez rigoureusement chaque étape de votre raisonnement.
3. Analyse empirique : Testez les deux implémentations sur un jeu de données pertinent et accessible en ligne. Décrivez en détail le jeu de données utilisé et citez vos sources. Présentez ensuite vos résultats sous forme de graphiques, en comparant les performances observées aux bornes théoriques établies.

3 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford est une autre méthode classique pour calculer les plus courts chemins depuis un sommet source dans un graphe pondéré. Bien qu'il soit capable de trouver ces chemins en présence de poids négatifs contrairement à l'algorithme de Dijkstra, nous nous limiterons dans ce projet au cas des graphes n'ayant pas de poids négatifs. Le pseudo-code suivant décrit les étapes principales de l'algorithme :

Algorithme 2 : Algorithme de Bellman-Ford

Entrées : Graphe $G = (V, E)$, sommet source s
Sorties : Distances minimales de s vers chaque sommet
pour *chaque sommet* $v \in V$ **faire**
  $dist[v] \leftarrow \infty$;
  $precedent[v] \leftarrow$ NULL;
fin
$dist[s] \leftarrow 0$;
pour *i de 1 à* $|V| - 1$ **faire**
  pour *chaque arête* $(u, v) \in E$ **faire**
    si $dist[u] + poids(u, v) < dist[v]$ **alors**
      $dist[v] \leftarrow dist[u] + poids(u, v)$;
      $precedent[v] \leftarrow u$;
    fin
  fin
fin
retourner *precedent*

1. Implémenter l'algorithme de Bellman-Ford en Python.
2. Analyser de la complexité : Déterminez la borne asymptotique supérieure $\mathcal{O}$ du temps d'exécution en fonction du nombre de sommets et d'arêtes. Justifiez rigoureusement chaque étape de votre raisonnement.
3. Analyser empiriquement : Utilisez le même jeu de données que précédemment. Présentez vos résultats sous forme de graphiques clairs, en comparant les performances observées avec la borne asymptotique obtenue théoriquement.
4. Comparer les performances des deux algorithmes (Dijkstra avec un tas de Fibonacci et Bellman-Ford) : Présentez cette comparaison sous forme d'un tableau synthétique.

Le code et les résultats doivent être présentés sous la forme d'un rapport (max 10 pages) à rendre au format PDF.