Projet
MOGPL

Université Paris 6: M1 Semestre 1

Etienne PENAULT & Ruizheng XU

Sommaire

1. Question 1

1.1. Assertion 1 : Chemin d'arrivée au plus tôt

Un sous-chemin préfixe d'un chemin d'arrivée au plus tôt peut ne pas être un chemin d'arrivée au plus tôt.

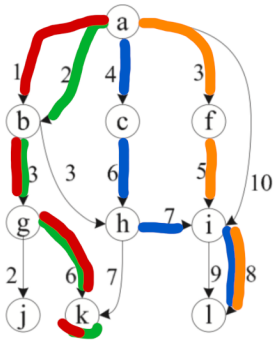

Figure 1: Arrivée au plus tôt

Pour le chemin de **A** à **K**:
Ici on voit que pour le trajet de **A** à **K** plusieurs chemins sont possibles:

- Le trajet P a pour chemin **{(A, B, 2, 1), (B, G, 3, 1), (G, K, 6, 1)}** et arrive au bout du septième jour car **fin(P)** = 6 + 1 = 7.
- Le trajet Q a pour chemin **{(A, B, 1, 1), (B, G, 3, 1), (G, K, 6, 1)}** et arrive au bout du septième jour car **fin(Q)** = 6 + 1 = 7.

fin(P) = fin(Q) donc les trajets **P** et **Q** sont les chemins d'arrivées au plus tôt. **P** et **Q** partent tous deux de **A** pour aller à **B** lors de la première étape du trajet mais ne passent pas par le même arc. **P** passe par l'arc **(A, B, 2, 1)** tandis que **Q** passe par l'arc **(A, B, 1, 1)**.
Pour ces deux sous chemins **P' = {(A, B, 2, 1)}** et **Q' = {(A, B, 1, 1)}**, on se rend compte que **(fin(P') = 3)** < **(fin(Q') = 2)**.

Donc le **P'** n'est pas un sous chemin d'arrivée au plus tôt car **fin(P')** < **fin(Q)**. Par conséquent, il se peut qu'un sous-chemin préfixe d'un chemin d'arrivée au plus tôt peut ne pas être un chemin d'arrivée au plus tôt.

Pour le chemin de **A** à **L**:
On veut regarder si il existe un sous chemin de **A** à **L** non optimal:

- **P = [A, C, H, I, L] → {(A, C, 4, 1), (C, H, 6, 1), (H, I, 7, 1), (I, L, 8, 1)}** est un chemin d'arrivée au plus tôt pour aller du nœud **A** au nœud **I** avec **fin(P)** = 8 + 1 = 9.
- **P' = [A, C, H, I] → {(A, C, 4, 1), (C, H, 6, 1), (H, I, 7, 1)}** est un sous-chemin préfixe de **P** avec **fin(P')** = 7 + 1 = 8.

Le chemin **P'** n'est pas le chemin d'arrivée au plus tôt pour aller de nœud **A** au nœud **I**, puisqu'il existe:

- Q = **[A, F, I] → {(A, F, 3, 1), (F, I, 5, 1), (I, L, 8, 1)}** qui est aussi un chemin d'arrivée au plus tôt.
- Un sous-chemin **Q'** de **Q** allant aussi à **I** tel que **Q' = [A, F, I] → {(A, F, 3, 1), (F, I, 5, 1)}** tel que **fin(Q)** = 5 + 1 = 6, et **fin(Q)** < **fin(P')**.

Ainsi, le sous-chemin **P'** du chemin d'arrivée au plus tôt **P** n'est pas un chemin d'arrivée au plus tôt, car **Q** arrive plus tôt que **P'** sur le nœud **I**. Donc avec cet exemple, on montre qu'un sous-chemin préfixe d'un chemin d'arrivée au plus tôt peut ne pas être un chemin d'arrivée au plus tôt.

1.2. Assertion 2 : Chemin de départ au plus tard

Un sous-chemin postfixe d'un chemin de départ au plus tard peut ne pas être un chemin de départ au plus tard.

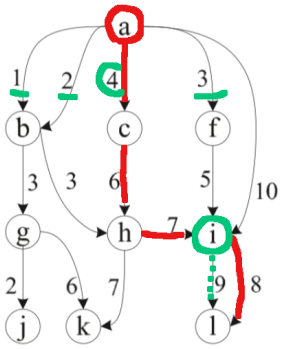

Figure 2: Départ au plus tard

Pour le chemin de **A** à **L**:

- Le chemin **P** = **[A, C, H, I, L]** → **{(A, C, 4, 1), (C, H, 6, 1), (H, I, 7, 1), (I, L, 8, 1)}** est un chemin de départ au plus tard pour aller du nœud **A** au nœud **L**, avec **début(P) = max((A, 1, B), (A, 2, B), (A, 4, C), (A, 3, F))** = 4.
- Un sous-chemin postfixe du chemin **P** est **P' = [I, L] → {(I, L, 8, 1)}**, avec **début(P') = 8**. **P** avec **fin(P')** = 7 + 1 = 8.
- Par ailleurs, pour aller du nœud **I** à **L**, il existe un autre chemin **Q = [I, L] → {(I, L, 9, 1)}**, avec **début(Q)** = 9.

On a alors **début(P')** < **début(Q)**, donc le sous-chemin postfixe d'un chemin de départ au plus tard peut ne pas être un chemin au plus tard.

1.3. Assertion 3 : Chemin le plus rapide

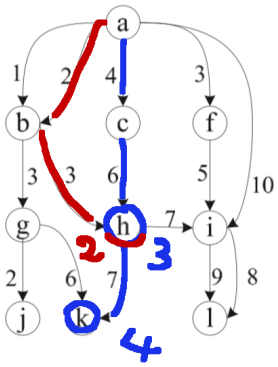

Figure 3: Chemin le plus rapide

Un sous-chemin d'un chemin le plus rapide peut ne pas être un chemin le plus rapide.

Pour le chemin de **A** à **K**:

- Le chemin le plus rapide est **P = [a, c, h, k]** → **{(a, c, 4, 1), (c, h, 6, 1), (h, k, 7, 1)}**. On a donc **durée(P) = fin(P) - début(P)** = 8 - 4 = 4.
- Prenons **P'** un sous chemin de **P** tel que **P' = [a, c, h]** → **{(a, c, 4, 1), (c, h, 6, 1)}**. On a donc **durée(P') = fin(P') - début(P')** = 7 - 4 = 3.

On veut montrer que le sous-chemin **P'** peut ne pas être le chemin le plus rapide.

- Or il existe un autre chemin Q tel que **durée(Q)** < **durée(P')** avec Q = **[a, b, h]** → **{(a, b, 2, 1), (b, h, 3, 1)}**. Calculons la durée de Q. **durée(Q)** = **fin(Q)** - **début(Q)** = 4 - 2 = 2.

On a bien **durée(Q)** < **durée(P')** alors que **P'** est un sous chemin du chemin **P** le plus rapide. On peut conclure qu'un sous-chemin d'un chemin le plus rapide peut ne pas être le chemin le plus rapide.

1.4. Assertion 4 : Plus court chemin

Un sous-chemin d'un plus court chemin peut ne pas être un plus court chemin.

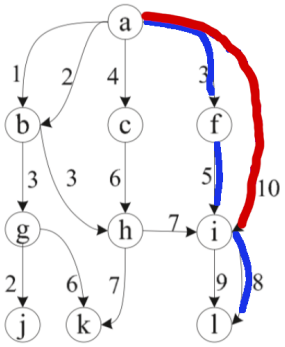

Figure 4: Plus court chemin

Supposons que la contrainte des jours est respectée.

Pour le chemin de **A** à **L**:

- Le chemin P = **[A, F, I, L] = {(A, F, 3, 1), (F, I, 5, 1), (I, L, 9, 0)}** avec **dist(P)** = 3 est le chemin le plus court pour aller de nœud **A** au nœud **L**.
- Un sous-chemin **P'** du chemin **P** est: **P' = [a, f, i] = {(a, f, 3, 1), (f, i, 5, 1)}** avec **dist(P')** = 2, du nœud **A** au nœud **I**.
- Or, il existe un autre chemin Q = **[A, I] = {(A, I, 10, 1)}** avec **dist(Q)** = 1.

Donc on a **dist(Q)** < **dist(P')**, avec **P'** le sous-chemin du chemin le plus court **P**, donc on a montré qu'un sous-chemin d'un plus court chemin peut ne pas être un plus court chemin.

Remarque: Si on allait du nœud **A** au nœud **I** directement, on ne pourrait plus aller au nœud **L** plus tard. Donc si la contrainte des jours n'existait pas, alors l'assertion ne serait pas toujours vraie et les sous-chemins seraient toujours les plus courts chemins (on trouverait au pire un autre chemin de la même distance mais pas meilleur).

2. Question 2

2.1. Transformation du graphe

Pour la transformation du graphe G en $\tilde{G}$, nous avons remarqué que tous les nœuds de G n'ont pas forcément d'interêt dans $\tilde{G}$ selon l'intervalle de temps sur laquelle nous travaillons. Imaginons que nous travaillons sur le graphe G de l'*Exemple 1* (Figure 5) avec l'interval [3, 11]. Nous remarquons que les arcs **(A, B, 1, 1)**, **(A, B, 2, 1)**, **(G, J, 2, 1)** n'ont plus besoin d'exister (car leurs poids en terme de temps < 3). Cela veut dire que nous n'avons pas besoin de rentrer ces valeurs dans les $\tilde{V}_{in}$ des sommets cibles des arcs, ni de rentrer les valeurs des nœuds sources dans $\tilde{V}_{out}$.

En reprenant notre exemple sur l'interval [3, 11], cela voudrait dire que:

- $\tilde{V}_{in}(B)$ serait privé de {(B, 2), (B, 3)}
- $\tilde{V}_{in}(J)$ serait privé de {(J, 3)}
- $\tilde{V}_{out}(A)$ serait privé de {(A, 1), (A, 2)}
- $\tilde{V}_{out}(G)$ serait privé de {(G, 2)}

Nous utiliserons cette modification.

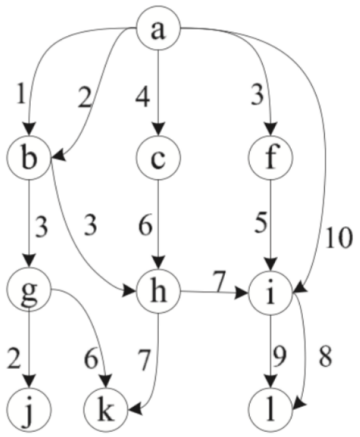

Figure 5: Graphe G de l'*Exemple 1*

Remarque 1: En appliquant cette méthode pour passer de G à $\tilde{G}$, cela implique que nous ne pourront plus appliquer nos algorithmes de recherche avec un interval de temps différent après cette modification vu que $\tilde{G}$ ne possède plus tous les arcs/nœuds. Il faudrait alors reconstruire un nouveau graphe selon le nouvel interval pour enfin travailler dessus...

Remarque 2: Il est tout de même possible de ne pas enlever de nœud au graphe $\tilde{G}$ et de vérifier à la volée, lors des algorithmes sur les chemins, si les nœuds sont visitables selon l'interval spécifié ou non (nous avions commencé par implémenter cela). Cependant, avec une telle version, les algorithmes sur les chemins peuvent être plus long que la méthode énoncée ci dessus car il y a plus de nœuds et d'arc à vérifier.

Un avantage de la modification est que $\tilde{G}$ permet de travailler sur un graphe sans se soucier si on peut atteindre un nœud ou non, selon les contraintes de temps initiales. Par exemple, seul le nœud **(G, 2)** permet d'acceder à **(J,3)**, donc seul **(G, 2)** à un lien vers le nœud **J**. Il est donc impossible dans $\tilde{G}$ d'avoir un lien d'un chemin qui violerait la contrainte de temps comme le lien de **(G, 4)** à **(J, 3)**.

2.2. Méthode de calcul des différents types de chemin

Toutes ces méthodes partent du principe que nous travaillons sur $\tilde{G}$, modifié de la manière expliquée dans le point précédent. Pour toute cette section nous allons appeler le nœud de départ $\boldsymbol{D}$ et le nœud d'arrivée $\boldsymbol{A}$.

Chaque nœud de G est decliné sous forme d'un ensemble $\boldsymbol{\zeta}$ de plusieurs nœuds dans $\tilde{G}$ tel que:

$$\forall \text{ nœud } \boldsymbol{E} \in G \;\Rightarrow\; \underline{\boldsymbol{\zeta(E) = \{(E,temps1),\ (E,temps2),\ \ldots,\ (E,tempsN)\}}} \in \tilde{G}$$

Remarque: $\boldsymbol{\zeta(E)}$ provient de $\tilde{\mathrm{V}}_{in}(\mathrm{E}) \cup \tilde{V}_{out}(\mathrm{E})$. Les éléments d'un ensemble $\boldsymbol{\zeta}$ sont reliés chronologiquement entre eux et pondérés par la valeur 0.

2.2.a. Type I : Chemin d'arrivée au plus tôt

Le chemin d'arrivée au plus tôt consiste à trouver le chemin de $\boldsymbol{D}$ à $\boldsymbol{A}$ tel que l'arrivée à $\boldsymbol{A}$ soit le plus petit possible en terme de temps dans G, tout en étant atteignable depuis $\boldsymbol{D}$.

Dans $\tilde{G}$, cela se résume donc à trouver un chemin de $\boldsymbol{D'}$ à $\boldsymbol{A'}$ tel que:

- $\boldsymbol{D'}$ un nœud tel que $\boldsymbol{D'} \in \boldsymbol{\zeta(D)}$.
- $\boldsymbol{A'}$ tel que $\boldsymbol{A'} \in \boldsymbol{\zeta(A)}$ car on veut tous les nœuds étiquetés "$\boldsymbol{A}$" afin de selectionner le plus petit temps (on cherche à minimiser le temps de $\boldsymbol{A'}$).

Etant donné que nous travaillons sur $\tilde{G}$, tous les nœuds sont reliés suivant l'ordre chronologique et ne peuvent donc pas violer la contrainte de temps. Sachant cela, nous pouvons chercher le chemin d'arrivée au plus tôt via un parcours en largeur de notre graphe, en conservant les prédecesseurs de chaque nœud et en les marquant. Il nous restera plus qu'a regarder tous les nœuds $\boldsymbol{A'}$ marqués, tel que $\boldsymbol{A'} \in \boldsymbol{\zeta(A)}$ et de conserver le plus petit en terme de temps pour avoir notre nœud d'arrivée final et de retracer son chemin.

Remarque: Le parcours en largeur n'est pas un algorithme glouton en temps normal, or le fait de l'appliquer sur $\tilde{G}$ qui est déjà ordonné chronologiquement et qu'il n'ait pas de cycle (positif comme négatifs) le rend glouton, dans ce cas là. Nous avons la certitude que chaque nœud sera parcouru une seule fois. Cela nous permet d'arrêter l'algorithme avec certitude d'avoir le chemin optimal lorsque nous avons traité tous les nœuds d'arrivée atteignables de $\boldsymbol{\zeta(A)}$.

2.2.b. Type II : Chemin de départ au plus tard

Le chemin de départ au plus tard consiste à trouver le chemin de $\boldsymbol{D}$ à $\boldsymbol{A}$ tel que le départ à $\boldsymbol{D}$ soit le plus grand possible en terme de temps dans G, tout en s'assurant que $\boldsymbol{A}$ est atteignable.

Plaçons nous dans $\tilde{G}$.
De la même manière que pour le type de chemin prédédent, nous allons lancer un parcours en largeur sur le graphe. Une fois cela fait, nous cherchons à trouver le plus grand nœud de départ $\boldsymbol{D'} \in \boldsymbol{\zeta(D)}$ tel que $\boldsymbol{D'}$ puisse nous mener à un nœud de $\boldsymbol{\zeta(A)}$.

Pour cela, nous allons récuperer les chemins de tous les nœuds $\boldsymbol{A'}$ marqués, tel que $\boldsymbol{A'} \in \boldsymbol{\zeta(A)}$. Les chemins commençant tous par un nœud $\boldsymbol{D'}$ tel que $\boldsymbol{D'} \in \boldsymbol{\zeta(D)}$, il nous suffit de selectionner le chemin ayant le plus grand $\boldsymbol{D'}$ en terme de temps pour avoir notre chemin de départ au plus tard.

Remarque 1: Cet algorithme peut nous trouver plusieurs chemins pour une même date de départ au plus tard. Nous choisirons alors le chemin d'arrivée au plus tôt parmi ces chemins de départ au plus tard.

Remarque 2: Le parcours en largeur n'est pas un algorithme glouton en temps normal, or le fait de l'appliquer sur $\tilde{G}$ qui est déjà ordonné chronologiquement et qu'il n'ait pas de cycle (positif comme négatif) le rend glouton dans ce cas là. Nous avons la certitude que chaque nœud sera parcouru une seule fois. Cela nous permet d'arrêter l'algorithme avec certitude d'avoir le chemin optimal lorsque nous avons traité tous les nœuds d'arrivée atteignables de $\boldsymbol{\zeta(A)}$.

2.2.c. Type III : Chemin le plus rapide

Le chemin le plus rapide consiste à trouver le chemin de $\boldsymbol{D}$ à $\boldsymbol{A}$ tel que dans G, le départ à $\boldsymbol{D}$ soit le plus grand possible en terme de temps, et que l'arrivée à $\boldsymbol{A}$ soit la plus petite possible en terme de temps.

Cela revient dans $\tilde{G}$ à tester tous les chemins en partant de tous les nœuds de $\boldsymbol{\zeta(D)}$ car nous ne savons pas à l'avance quel départ sera optimal, ni quelle arrivée sera optimale.

Pour se faire, nous allons encore utiliser un parcours en largeur. Cependant, contrairement aux autres parcours en largeur, nous ne nous contenterons pas de marquer les nœuds, mais de conserver leurs "distances en terme de temps". A la manière de l'algorithme de *Dijkstra* ou *Bellman-Ford*, nous allons mettre à jours les "distances de temps" par relaxation.

Une fois le parcours en largeur effectué, il ne nous reste plus qu'à récuperer le chemin ayant la "distance de temps" minimum parmis tous les nœuds de $\boldsymbol{\zeta(A)}$.

Remarque: Le parcours en largeur n'est pas un algorithme glouton en temps normal, or le fait de l'appliquer sur $\tilde{G}$ qui est déjà ordonné chronologiquement et qu'il n'ait pas de cycle (positif comme négatifs) le rend glouton dans ce cas là. Nous avons la certitude que chaque nœud sera parcouru une seule fois. Cela nous permet d'arrêter l'algorithme avec certitude d'avoir le chemin optimal lorsque nous avons traité tous les nœuds d'arrivée atteignables de

$\boldsymbol{\zeta(A)}$.

2.2.d. Type IV : Plus court chemin

Le plus court chemin consiste à trouver le chemin de $\boldsymbol{D}$ à $\boldsymbol{A}$ tel que dans G, le chemin ait le moins d'étapes intermédiaires possibles.

Dans $\tilde{G}$, il suffit d'executer l'algorithme de *Dijkstra* en partant de $\boldsymbol{D' = \min(\zeta(D))}$ car on veut tester tous les nœuds de départ étiquetés "$\boldsymbol{D}$" (on rappel que les nœuds d'un même ensemble $\boldsymbol{\zeta}$ sont reliés chronologiquement entre eux. Donc en prenant le plus petit en terme de temps comme point de départ, nous testerons également les autres nœuds de l'ensemble).

Une fois l'algorithme fini, il suffit de comparer toutes les distances des nœuds $\boldsymbol{A' \in \zeta(A)}$ et de selectionner la plus petite distance pour determiner quel chemin conserver.

Remarque 1: Le graphe $\tilde{G}$ nous offre l'avantage de ne pas se soucier des contraintes de temporalité car il n'y a aucun arc/nœud violant cette contrainte dans ce dernier, contrairement à G.

Remarque 2: *Dijkstra* est un algorithme glouton, cela nous permet d'arreter l'algorithme avec certitude d'avoir le chemin optimal si nous avons traité tous les nœuds de $\boldsymbol{\zeta(A)}$.

2.2.e. Remarque pour tous les algorithmes

A la fin de chaque algorithme de chemin, nous effectuons le "Back tracking" du chemin en remontant ses prédécesseurs. Si plusieurs nœuds à la suite appartiennent tous au même ensemble $\boldsymbol{\zeta}$ (ils sont étiquetés avec le même nom de nœud) , alors nous garderons dans le chemin uniquement celui de plus haut temps (par exemple imaginons qu'on ait la séquence de nœuds [(A,1), (A,2), (A,4)] dans notre chemin, nous pouvons uniquement garder [(A,4)] vu que la distance entre tous les nœuds du même nom est de 0 dans $\tilde{G}$).

3. Question 3

Dans cette section, nous appellerons $|A|$ le nombre d'arcs et $|N|$ le nombre de nœuds.

3.1. Complexités du chemin d'arrivée au plus tôt

Le chemin d'arrivée au plus tôt est basé sur un parcours en largeur dont la compléxité est en $\mathcal{O}(|A| + |N|)$.

A la fin du parcours en largeur, nous devons vérifier les nœuds d'arrivées marqués $\boldsymbol{A'}$ tel que $\boldsymbol{A'} \in \boldsymbol{\zeta(Arrivée)}$. Les nœuds étant ordonnés dans $\boldsymbol{\zeta(Arrivée)}$, nous nous arrêtons dès que l'on en trouve un marqué. Dans le pire cas, cette recherche se fait en $\mathcal{O}(|\boldsymbol{\zeta(Arrivée)}|)$

La complexité finale est:

$$\mathcal{O}(|A| + |N|) + \mathcal{O}(|\boldsymbol{\zeta(Arrivée)}|) = \mathcal{O}(|A| + |N| + |\boldsymbol{\zeta(Arrivée)}|)$$

3.2. Complexité du chemin de départ au plus tard

Le chemin de départ au plus tard est également basé sur un parcours en largeur dont la compléxité est en $\mathcal{O}(|A| + |N|)$.

Cependant, afin de récuperer le bon chemin à la fin du parcours en largeur, nous devons vérifier tous les chemins possibles en partant des nœuds d'arrivées marqués $\boldsymbol{A'}$ tel que $\boldsymbol{A'} \in \boldsymbol{\zeta(Arrivée)}$. Donc dans le pire cas, cette recherche se fait en $\mathcal{O}(|\boldsymbol{\zeta(Arrivée)}|)$

La complexité finale est:

$$\mathcal{O}(|A| + |N|) + \mathcal{O}(|\boldsymbol{\zeta(Arrivée)}|) = \mathcal{O}(|A| + |N| + |\boldsymbol{\zeta(Arrivée)}|)$$

3.3. Complexité du chemin le plus rapide

Comme les deux méthodes précédentes, cette recherche de chemin est également basée sur le parcours en largeur. Nous retrouvons sensiblement la même complexité au pire cas car nous devons comparer les distances de chaque nœuds de $\boldsymbol{\zeta(Arrivée)}$ à la place de comparer les marques. On peut en conclure que la complexité finale est:

$$\mathcal{O}(|A| + |N|) + \mathcal{O}(|\boldsymbol{\zeta(Arrivée)}|) = \mathcal{O}(|A| + |N| + |\boldsymbol{\zeta(Arrivée)}|)$$

3.4. Complexité du plus court chemin

Le plus court chemin est basé sur l'algorithme de *Dijkstra*. La compléxité de Dijkstra est en $\mathcal{O}(|N| + |A| \log |N|)$ en utilisant une "priority queue".

- Nous visitons tous les nœuds dans le pire cas lors de l'algorithme. Chaque nœud est visité qu'une seule est unique fois. Donc tous les nœuds sont visités en $\mathcal{O}(|N|)$.
- Itérer sur les voisins de tous les nœuds et mettre à jour leurs distances se fait en passant une seule et unique fois sur tous les arcs. Cela se fait en $\mathcal{O}(|A|)$.

Donc, tous les nœuds du graphe peuvent être traversés en $\mathcal{O}(|N|) + \mathcal{O}(|A|) = \mathcal{O}(|N| + |A|)$.

- Pour chaque itération de la boucle principal de *Dijkstra*, un sommet se voit supprimé de la priority queue, ce qui se fait en $\mathcal{O}(\log |N|)$.

Nous retrouvons donc $\mathcal{O}(|N| + |A|) * \mathcal{O}(\log |N|) = \mathcal{O}(|N| + |A| \log |N|)$.

Cependant, une fois l'algorithme de *Dijkstra* effectué, nous devons sélectionner notre nœud final d'arrivée. C'est nul autre que le plus petit nœud en terme de distance de l'ensemble $\boldsymbol{\zeta}$ du nœud d'arrivée cible (autrement dit, $\boldsymbol{min(\zeta(Arrivée))}$). Nous pouvons le trouver dans le pire cas en $\mathcal{O}(|\boldsymbol{\zeta(Arrivée)}|)$.

La complexité finale pour trouver le chemin d'arrivée au plus tôt est:

$$\mathcal{O}(|A| \log |N|) + \mathcal{O}(|\boldsymbol{\zeta(Arrivée)}|) = \mathcal{O}(|\boldsymbol{\zeta(Arrivée)}| + |N| + |A| \log |N|)$$

4. Question 4

Tout d'abord, il est important de bien définir la structure que nous allons utiliser pour notre multigraphe. Ce dernier sera implémenté comme une classe Python de nom Graph, avec des attributs suivants.

graph.py:

```
class Graph:

def __init__(self):
    self.__adj = {}
    self.__nb_vertices = 0
    self.__nb_edges = 0
```

- __adj : un dictionnaire dont les clés sont les noms des sommets du multigraphe, et la valeur correspondante de chaque clé est une liste contenant les successeurs de ce sommet, et pour chacun des successeurs, nous avons stocké le nom du successeur, la date de départ et le poids de l'arc (la distance).

  Dans le graphe G de l'Exemple 1 du sujet : G.__adj['g'] = [('j', 2, 1), ('k', 6, 1)]

- __nb_vertices : le nombre total des sommets du multigraphe.

  Dans le graphe G de l'Exemple 1 du sujet : G.__nb_vertices = 10

- __nb_edges : le nombre total des arcs du multigraphe.

  Dans le graphe G de l'Exemple 1 du sujet : G.__nb_edges = 14

Ensuite, nous avons implémenté les fonctions de classe nécessaire pour:

- La création du graphe:

```
    def create_from_file(self, file_name):...
    def create_manually(self):...
```

- La simplification du graphe:

```
    def create_simplified(self, interval):...
```

- La recherche des différents types de chemins:

```
    def earliest_arrival(self, start, end):...
    def latest_departure(self, start, end):...
```

```
3         def fastest_path(self, start, end):...
4         def shortest_path(self, start, end):...
```

Enfin, dans un fichier 'main.py', nous définissons le sommet de départ, le sommet d'arrivée, et aussi l'interval de temps que notre chemin solution doit respecter. Nous appelons les différentes fonctions définies et le programme nous retournera les quatres types de chemins.

Il y a deux manières de lancer le programme:

- Dans le cas ou on veut rentrer le graphe à la main depuis le terminal.

```
1    python main.py
```

- Dans le cas ou on veut charger un graphe depuis un fichier (ici *in.txt*).

```
1    python main.py in.txt
```

5. Question 5

Dans cette partie, nous allons modéliser notre problème du plus court chemin du graphe par la programmation linéaire, et d'implémenter un modèle capable de faire la recherche de chemin de type IV (le plus court chemin) à travers le solver GUROBI.

5.1. Modélisation du problème

5.1.a. Les variables de décisions

Premièrement, nous devons trouver les variables de décisions de notre programme linéaire. Dans notre cas, on cherche le(s) chemin(s) le(s) plus court(s) du sommet de départ jusqu'au sommet d'arrivée, ainsi, les variables de décisions sont tous les arcs de notre multigraphe, et le nombre des variables de décision correspond au nombre d'arcs du graphe. Ces variables de décisions seront des variables binaires, c'est-à-dire qu'elles auront comme valeur 0 ou 1.

Les variables de décisions sont représentées dans une liste x, ainsi : x_i représente $i^{ème}$ variable de décision de la liste. De là, x_i nous donne (u, v, t, λ) avec u, v, t, λ qui correspondent respectivement au nom du sommet de départ, nom du sommet d'arrivée, la date de départ, et le temps à dépenser pour traverser de l'arc i, et $x_i = \begin{cases} 1 & \text{si l'arc appartient au chemin} \\ 0 & \text{sinon} \end{cases}$
Ainsi, sur la figure de l'Exemple 1 du sujet, on peut avoir $x_0 = (a, b, 1, 1) = 0$ ou 1.

```
1  # edges contient l'informations de chaque arcs (futures variables décisions)
2  edges.append((from_node, to_node, int(depart_date), int(travel_time)))
```

```
# cost est le coût de prendre un arc (1 dans notre cas)
cost[(from_node, to_node, int(depart_date), int(travel_time))] = 1
```

Remarque 1 : Le chemin que le programme va retourner sera toutes les variables de décisions dont la valeur est égale à 1.

Remarque 2 : Les arcs qui partent du point de départ, et qui ont une date de départ t inféreure à la date t_α ne seront pas ajoutés dans la liste des variables de décisions, puisque leur date ne sont pas dans l'interval de temps défini. Idem pour les arcs qui arrivent sur le point d'arrivée et qui ont une date de départ $t + \lambda > t_\beta$.

5.1.b. L'objectif

On cherche le chemin de type IV (plus court chemin), donc notre objectif sera de minimiser la somme des coûts de toutes les variables de décisions :

$$Obj : min \sum_{i=1}^{nbEdges} x_i c_i$$

avec c_i qui correspond au coût de l'arc i (dans notre cas 1 pour tous les arcs).

```
    x = m.addVars(edges, vtype=gp.GRB.BINARY, obj=cost)
```

5.1.c. Les contraintes

Une fois l'objectif déterminé, on doit donner des contraintes à notre programme linéaire. Nos contraintes pour trouver un chemin valide seront :

- Contrainte sur les sommets de départ. Le programme doit choisir un arc parmis tous les arcs qui partent du point de départ, donc la somme de tous les arcs dont le sommet de départ est notre point de départ doit être supérieure ou égales à 1 :

$$\sum_i sub\_x_i \geq 1$$

tel que $\forall e \in sub\_x, e[0] =$ sommet de départ de l'arc e = origine, avec origine le point de départ de notre chemin.

```
    m.addConstr(sum(x[i, j, k, l] for i, j, k, l in\
        edges.select(origin, '*', '*', '*')) >= 1)
```

- Contrainte sur les sommets d'arrivé. Le principe est pareil que la contrainte sur les sommets de départ, la somme de tous les arcs dont le sommet d'arrivée est notre point d'arrivé doit être supérieure ou égale à 1 :

$$\sum_i sub\_x_i \geq 1$$

tel que $\forall e \in sub\_x, e[1] =$ sommet d'arrivé de l'arc e = destination, avec destination le point d'arrivé du chemin.

```
    m.addConstr(sum(x[i, j, k, l] for i, j, k, l in\
        edges.select('*', destination, '*', '*')) >= 1)
```

- Contrainte sur le passage d'un sommet à un autre. Un arc qui va du sommet A au sommet B peut être pris si et seulement si un des arcs arrivant sur ce sommet A est pris. C'est-à-dire que pour tous les arcs e dont le sommet de départ u n'est pas le point de départ, e doit être inférieure ou égale à la somme de tous les arcs e' qui ont comme sommet d'arrivée v égal à u. Plus concrétement avec un la figure de l'Exemple 1 du sujet, si l'arc (B, G, 3, 1) et pris, alors (B, G, 3, 1) est inférieure ou égale à la somme des arcs (A, B, 1, 1) et (A, B, 2, 1). Cette contrainte peut se modéliser comme suit:

$$\forall e \in x, e[0] \neq \text{origine}, e \leq \sum_i sub\_x_i$$

tel que $e = (u, v, t, \lambda), sub\_x = \forall e' \in x, e' = (u', v', t', \lambda')$ et $v' = u$. $(v' = u \Leftrightarrow (e'[1] = e[0])$

```
    for i, j, k, l in edges:
    if i != origin:
        m.addConstr(x[i, j, k, l] <=\
            sum(x[i_, j_, k_, l_] for i_, j_, k_, l_ in\
            edges.select('*', i, '*', '*') if k >= k_ + l_ ))
```

5.2. Exécution du programme linéaire

On lit d'abord le fichier contennant toutes les informations du multigraphe, et on construit notre modèle *GUROBI*. Ce modèle aura les variables de décisions, l'objectif et les contraintes qu'on a défini dans les parties précédentes, et il pourra trouver la solution automatiquement en respectant les contraintes.

Les résultats seront donc les variables de décisions qui ont la valeur égale à 1.

6. Question 6

Dans cette partie, nous allons testé notre algorithme sur les différents fichiers de mutligraphe que nous avons généré :

- Test set 1 : 12 graphes qui ont chacun en moyenne 20-40 arcs par sommets.
- Test set 2 : 12 graphes qui ont chacun en moyenne 100-120 arcs par sommets.

Et nous allons noté les temps d'exécution des quatres types de chemin et d'observer leur évolution.

Comme la génération des multigraphes prennent beaucoup de temps lorsque le nombre d'arcs/sommets est important, alors nous nous sommes limités à 12 graphes dans chacun des tests set (24 graphes en total).

6.1. Test Set 1

Pour tester l'impact du nombre de sommets, nous utilisons le test set 1 qui a en moyenne 20-30 arcs par sommets (si cette moyenne est faible, alors on risque de pas trouver de solution pour le graphe). Et nous lançons nos algorithmes sur ces graphes du test set 1 :

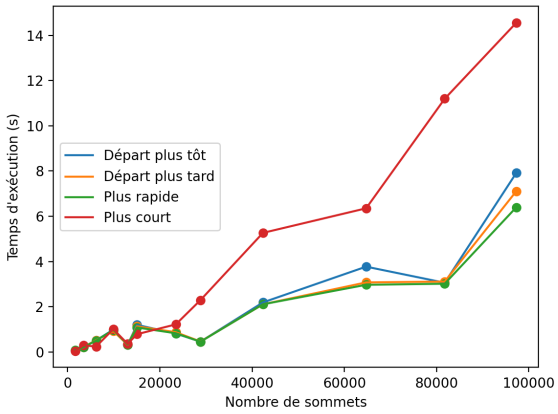

Figure 6: Évolution du temps d'exécution par rapport au nombre de sommets sur test set 1

On peut observer que l'évolution du temps d'exécution est "strictement" croissante lorsqu'on augmente le nombre de sommets de nos graphes. De plus, ces temps correspondent à nos attentes, en effet, nous avons utilisé l'algorithme *Parcours en largeur* pour les chemins de type I, II et III, et l'algorithme *Dijkstra* pour le type IV. Sur la figure 6, le temps d'exécution pour trouver le chemin le plus court est le plus long, *Dijkstra* a une compléxité "généralement" supérieure au *Parcours en largeur*.

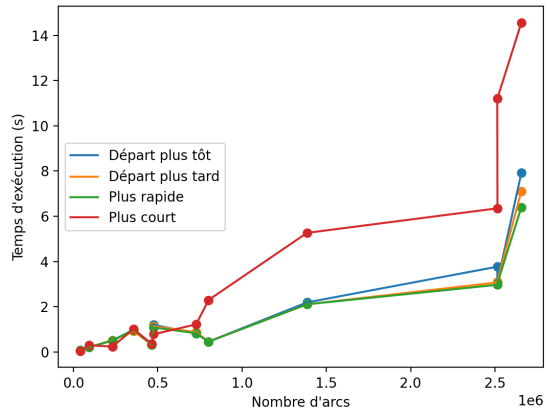

Figure 7: Évolution du temps d'exécution par rapport au nombre d'arcs sur test set 1

Sur cette figure 7, il y a un point intéssant sur le graphique. On remarque que le nombre d'arcs n'a beaucoup pas évolué sur les trois derniers graphes (autour de $2.5 * 10^6$), tandis que le nombre de sommets de ces trois graphes évolue (environ 65000, 80000, 100000 respectivement), et le temps d'exécution augmente quand même (de 2 secondes environ entre chaque graphe). Ainsi, plus il y a de sommets, plus le temps du prgramme augmente. Le nombre de sommets a un impact sur le temps d'exécution, et nous allons maintenant voir les impacts du nombre d'arcs sur le temps.

6.2. Test Set 2

Pour tester l'impact du nombre d'arcs, nous utilisons le test set 2 qui a en moyenne 100-120 arcs par sommets. Et nous lançons nos algorithmes sur ces graphes :

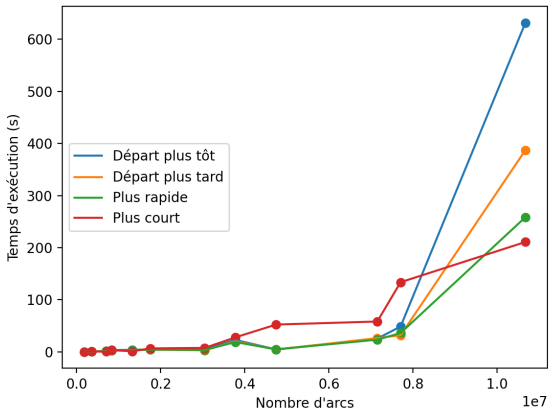

Figure 8: Évolution du temps d'exécution par rapport au nombre d'arcs sur test set 2

Tout de suite, on voit sur la figure 8 que le temps d'exécution est devenu très grand par rapport aux tests sur les graphes du test set 1 (nous sommes sur 600 secondes de test sur le graphe qui a le plus d'arcs).

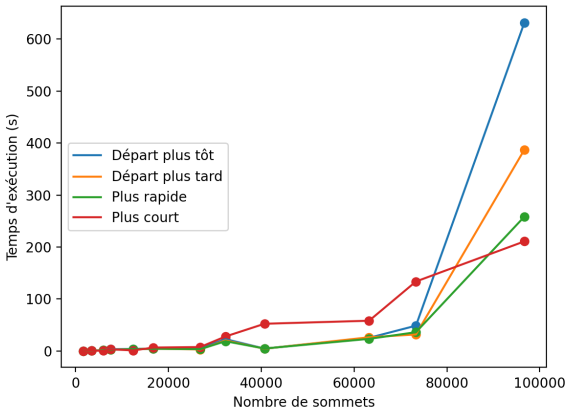

Figure 9: Évolution du temps d'exécution par rapport au nombre de sommets sur test set 2

Et à travers la figure 9, on remarque que le nombre de sommets est similaire aux graphes du test set 1, seul le nombre d'arcs a fortement augmenté. Donc on peut en déduire que le fait d'augmenté le nombre d'arcs a aussi un impact important sur le temps d'exécution, et cela augmente de plus en plus vite lorsque le nombre d'arcs dépasse $8 * 10^5$. Ce seuil peut être résultat du procésseur de la machine ou la limite de notre algorithme.

De même, la recherche du chemin le plus court n'est plus celle qui met le plus de temps, il est possible que cela arrive car les autres algorithmes sont gloutons et il peut arriver de couper des chemins lorsqu'on a déjà une solution, et on voit que ce n'est pas toujours le cas à travers notre test sur le test set 2.

Remarque 1 : Même si le test le plus long est d'ordre 600 secondes, nous n'avons pas compté le temps que l'algorithme met pour la simplification du graphe (cf Exemple 2 du sujet). Et le temps de la simplification augmente encore plus vite que le temps de la recherche des chemins (environ **une heure** pour la simplification du dernier graphe du test set 2).

Toutes les données de tests seront jointent avec ce rapport dans des fichiers CSV.

7. Question 7

Dans cette partie, nous allons comparer la performance de notre modèle *GUROBI* et notre programme de recherche de chemin en *Python* sur le chemin de type IV, sur les deux tests set comme la question précédente.

7.1. Test Set 1

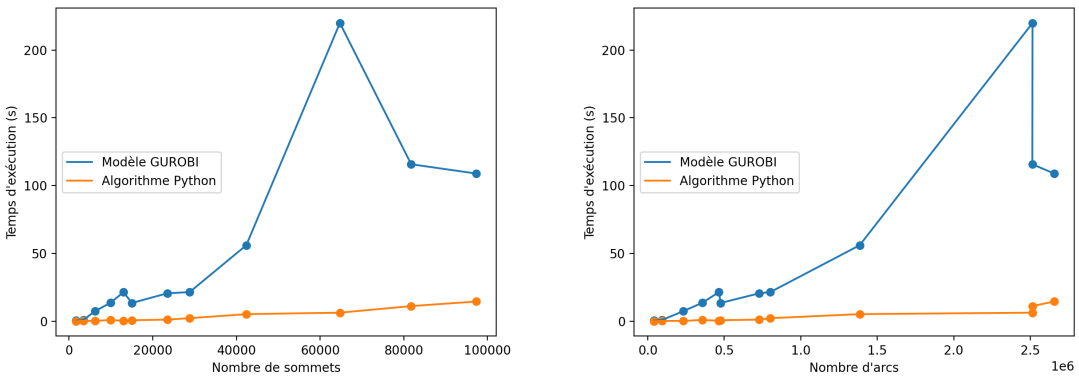

Figure 10: Comparaison du temps d'exécution pour la recherche du chemin de type IV sur test set 1

Sur cette figure 10, nous avons deux graphiques montrant le temps d'exécution du modèle *GUROBI* et les algorithmes Python, en fonction du nombre de sommets et d'arcs. On remarque immédiatement qu'il y a une augmentaton du temps d'exécution. Le modèle *GUROBI* met environ 200 fois plus de temps (sans compter les temps de simplification des graphes etc...) pour la recherche du chemin le plus court.

Par ailleurs, on remarque aussi que sur les deux derniers graphes, le modèle GUROBI obtient un meilleur temps par rapport au $10^{ème}$ graphe, sur lequel le modèle a mis le plus de temps (au-dessus de 200 secondes). L'insatisfication de nombreuses contraintes peut être l'origine de cette diminution de temps, mais il y a aussi d'autres facteurs qui entrent en jeu comme la façon qu'on a généré le graphe, les stratégies adoptées par le solver *GUROBI*, etc...

7.2. Test Set 2

Lorsqu'on a testé notre modèle *GUROBI* sur des graphes de test set 2, le modèle prend vraiment beaucoup trop de temps à se construire et à trouver la solution (surtout à cause du nombre de contraintes trop important). Ainsi, nous avons seulement pu tester les six premiers fichiers de tests avec le nombre d'arcs allant jusqu'à $1.7 * 10^6$.

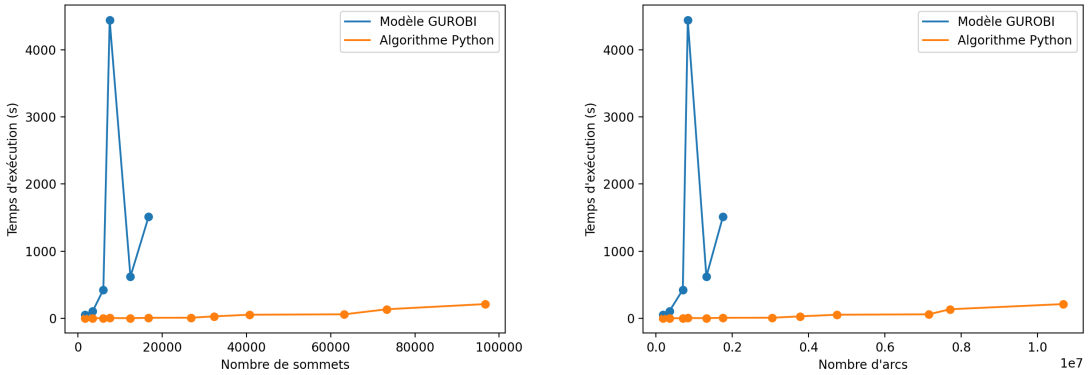

Figure 11: Comparaison du temps d'exécution pour la recherche du chemin de type IV sur test set 2

On remarque les mêmes phénomènes que les tests réalisés avec le test set 1, seulement le temps d'exécution a augmenté exponentiellement. On voit que pour le $4^{ème}$ graphe, la recherche du chemin a mis plus de 4000 secondes... Ce qui est attendu de notre part, car plus il y a d'arcs, plus on a de contraintes et le modèle est surchargé, donc il mettra plus de temps.

7.3. Récapitulatif

Après avoir effectuer les tests proposés dans la question 6 et 7, on voit l'impact de taille de l'entrée sur le temps d'exécution. En effet, le nombre de sommets et d'arcs sont souvent dépendants l'un de l'autre, c'est-à-dire plus on a de sommets, plus on a d'arcs. Mais le plus déterminant restera le **nombre d'arcs par sommets**.

Plus la moyenne d'arcs par sommets est élevée, plus on a de contraintes (pour le modèles *GUROBI*), et de sommets dans le graphe simplifié (pour notre implémentation *Python*), ainsi, on mettra plus de temps à rechercher les chemins. Donc, parallèlement, plus cette moyenne est faible, plus la résolution devient rapide.