Sorbonne Universite

Master 1 - Informatique

Année 2021/2022

Rapport de projet
-
Modélisation, Optimitation, Graphes, Programmation linéaire

Hugo THOMAS

Hakim AMRAOUI

Contents

1 Assertions sur les graphes

Assertion 1 : Un sous-chemin préfixe d'un chemin d'arrivée au plus tôt peut ne pas être un chemin d'arrivée au plus tôt.
Soient x, y deux sommets de G et P un chemin d'arrivée au plus tôt entre x et y. Montrons qu'il existe un sommet y' tel qu'un sous-chemin préfixe de P n'est pas un chemin d'arrivée au plus tôt entre x et y'. Considérons le chemin d'arrivé le plus tôt entre a et l, le chemin d'arrivée au plus tôt est $a \to c \to h \to i \to l$, d'arrivé au temps 8. Avec $y' = i$, on a un nouveau chemin d'arrivé qui est $a \to f \to i$ au temps 5. L'assertion 1 est donc vérifiée.

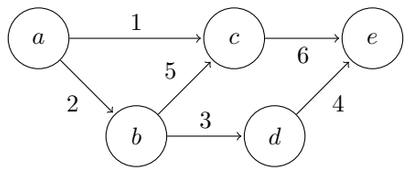

Figure 1: Graphe G utilisé pour notre exemple de l'assertion 2

Assertion 2 : Un sous-chemin postfixe d'un chemin de départ au plus tard peut ne pas être un chemin de départ au plus tard.
Soient x, y deux sommets de G et P un chemin de départ au plus tard entre x et y. Montrons qu'il existe sommet x' tel qu'un sous-chemin postfixe de P entre x' et y n'est pas un chemin de départ au plus tard. En utilisant le graphe G de la figure 1, un chemin de départ au plus tard en allant de a à e serait $a \to b \to d \to e$, de départ au temps 2. Or, en prenant le sous-chemin postfixe de départ en b, le chemin obtenu $b \to d \to e$ de départ au temps 3 n'est plus le chemin de départ au plus tard. En effet, en partant de b, le chemin de départ au plus tard serait $b \to c \to e$ de départ au temps 5. L'assertion 2 est donc vérifiée.

Assertion 3 : Un sous-chemin d'un chemin le plus rapide peut ne pas être un chemin le plus rapide.
Supposons qu'un sous-chemin d'un chemin le plus rapide est toujours un chemin le plus rapide, et trouvons un contre-exemple dans le graphe G du sujet. Entre les sommets a et l, un des chemins le plus rapide est le chemin $a \to c \to h \to i \to l$ de durée 5. Considérons le sous-chemin $a \to f \to i$, ce chemin de durée 3 n'est pas le chemin le plus rapide entre les sommets a et i, en effet le chemin $a \to i$ de durée 1 est plus rapide.

Assertion 4 : Un sous-chemin d'un plus court chemin peut ne pas être un plus court chemin.
Supposons qu'un sous-chemin d'un chemin le plus court est toujours un chemin le plus court, et trouvons un contre-exemple dans le graphe G du sujet. Considérons de nouveau sommets a et l, un des plus courts chemins est le chemin $a \to f \to i \to l$. Considérons le sous-chemin $a \to f \to i$, ce chemin de longueur 2 n'est pas le chemin le plus court entre les sommets a et i, en effet le chemin $a \to i$ de poids 1 est plus court.

2 Algorithmes pour les calculs des 4 types de chemins minimaux

On suppose que l'on a accès à une implémentation de l'algorithme de Dijkstra, sans en donner le pseudo code (il est dans le cours), par la procédure `Dijkstra`(d, f, G), telle que

$$\texttt{Dijkstra}(d,f,\texttt{G}) = \begin{cases} \texttt{nil} & \text{s'il n'y a pas de chemin entre } d \text{ et } f \text{ dans } G \\ (\mathcal{W}, \Pi) & \text{tels que } \mathcal{W}[i] \text{ est le poids du PCC jusqu'à } i, \text{ et} \\ & \Pi[i] \text{ est le prédécesseur de } i \text{ dans ce PCC} \end{cases} \quad (1)$$

La complexité temporelle de cette procédure est en $O\big((|V| + |E|) \log |V|\big)$ où $|E|$ est le nombre d'arcs et $|V|$ est le nombre de sommets, via l'utilisation d'une file de priorité.
Et d'autre part accès aux procédures suivantes :

- `bfs`(G, s) : *Graphe* $\times$ *Sommet* $\to$ *Tableau[Sommet]* : parcourt le graphe G en largeur à partir de s et retourne le tableau des prédécesseurs. La complexité temporelle de `bfs` est $O\big(|E| + |V|\big)$.
- `transpose`(G): *Graphe* $\to$ *Graphe* : renvoie le graphe transposé de G.
- `renverse`(l): *Liste* $\to$ *Liste* : renvoie la liste l dont les éléments dans l'orde inverse
- `duree`(P): *Chemin* $\to$ *Nombre* : renvoie la durée du chemin P

- `delimiter`$(\tilde{G}, d, f, t_\alpha, t_\omega)$: *Graphe* $\rightarrow$ *Graphe* : renvoie $\tilde{G}$ tel qu'un chemin $s \rightarrow f$ P dans $\tilde{G}$ respecte les contraintes $debut(P) > t_\alpha$ et $fin(P) < t_\omega$. Son pseudo code ne concernant pas directement les algorithmes de plus courts chemins, celui-ci est situé en annexe A.

Nous ne prenons pas en compte la compléxité de la transformation du graphe G au graphe $\tilde{G}$, qui sera examinée juste en suivant. $|V|$ et $|E|$ sont donc respectivement le nombre de sommets et le nombre d'arcs de $\tilde{G}$.

I. Arrivée au plus tôt

Algorithm 1 Arrivée au plus tôt

```
1:  procedure APT(d, f, G̃, t_α, t_ω)      ▷ d : Sommet de départ, f: Sommet de fin, G̃: le graphe
2:      PCC ← nil                                                  ▷ PCC avec la contrainte APT
3:      T ← ∞                                                      ▷ Jour d'arrivée au plus tôt
4:      G̃ ← delimiter(G̃, d, f, t_α, t_ω)
5:      G_T ← transpose(G̃)
6:      for all u ∈ Ṽ_in(f) do
7:          pcc ← bfs(G_T, s)
8:          if tmp ≠ nil and t < T then
9:              T ← t
10:             PCC ← pcc
11:     return renverse(PCC)
```

Idée de l'algorithme : Il est très similaire à l'algorithme II. On considère le graphe G_T transposé de G, ainsi nous pouvons, à partir de f, trouver un plus court chemin de f à d avec l'algorithme de Dijkstra. En itérant sur tous les sommets de $\tilde{V}_{in}(f)$ et en conservant le chemin dont le sommet de départ à la fois admet un chemin jusqu'à s et dont le temps de départ est minimal (dans G_T) l'algorithme renvoie le chemin d'arrivée au plus tôt. Si dans G_T il y a un chemin de f à d, alors il y a un chemin de d à f dans G. Et un chemin de départ au plus tôt dans G_T est un chemin d'arrivée au plus tôt dans G.

Complexité : Si le graphe est représenté via une liste d'adjacence, calculer son graphe transposé prend un temps en $O(|E| + |V|)$. L'appel à la procédure `delimiter` prend aussi un temps en $O(|E| + |V|)$ (simple parcours de tous les arcs de tous les noeuds). Il y a $d^+(f)$ éléments dans $\tilde{V}_{in}(f)$ donc $d^+(f)$ appels à `bfs`, et $d^+(f) << |E|$. D'où une compléxité totale en $O\big(|E| + |V|\big)$ si on suppose que $d^+(f)$ est constant devant $|E|$.

II. Départ au plus tard

Algorithm 2 Départ au plus tard

```
1:  procedure DPT(d, f, G̃, t_α, t_ω)      ▷ d : Sommet de départ, f: Sommet de fin, G̃: le graphe
2:      PCC ← nil                                                  ▷ PCC avec la contrainte DPT
3:      T ← −∞                                                     ▷ Jour de départ au plus tard
4:      G̃ ← delimiter(G̃, d, f, t_α, t_ω)
5:      for all u ∈ Ṽ_out(d) do
6:          pcc ← bfs(G̃, u)
7:          if tmp ≠ nil and t > T then
8:              T ← t
9:              PCC ← pcc
10:     return PCC
```

Idée de l'algorithme : Dans $\tilde{G}$ obtenu suite à la transformation, on parcourt tous les états représentant le sommet de départ i.e. $\tilde{V}_{out}(d)$, dans l'exemple 1, pour un départ depuis a, les sommets (a,1), (a,2), (a,4). Pour chacun de ces sommets on applique l'algorithme de Dijkstra afin de trouver le plus court chemin jusqu'à l'objectif. Un plus court chemin obtenu de cette manière et conservé si et seulement s'il n'est pas nul et son temps de départ t est supérieur au plus grand temps de départ déjà enregistré permettant d'arriver à l'objectif T.

Complexité : Il y a $d^-(d)$ éléments dans $\tilde{V}_{out}(d)$, il y a donc $d^-(d)$ appels à la procédure `bfs`, et $d^-(d) << |E|$. L'appel à la procédure `delimiter` prend un temps en $O(|E| + |V|)$, et `bfs` est également en $O(|E| + |V|)$. Donc une compléxité en $O\big(|V| + |E|\big)$ si l'on suppose $d^-(d)$ toujours très petit.

III. Chemin le plus rapide

Algorithm 3 Chemin le plus rapide

```
1: procedure CPR(d, f, G̃, t_α, t_ω)      ▷ d : Sommet de départ, f: Sommet de fin, G̃: le graphe
2:     pcc ← nil                          ▷ PCC avec la contrainte CPR
3:     D ← ∞                              ▷ Durée du tel PCC
4:     G̃ ← delimiter(G̃, d, f, t_α, t_ω)
5:     for all u ∈ Ṽ_out(d) do
6:         tmp, p = Dijkstra(u, f, G̃)
7:         d ← duree(tmp)
8:         if pcc ≠ nil and d < D then
9:             D ← d
10:            pcc ← tmp
11:    return (pcc, D)
```

Idée de l'algorithme : Encore une fois, en itérant sur tous les sommets de $\tilde{V}_{out}(d)$ et en calculant la $duree(pcc)$ à partir de ce sommet, on obtient la solution optimale en conservant le chemin qui minimise la durée. La boucle sur tout les sommets sert à s'assurer que l'on essaye bien tous les arcs sortants du sommet de départ. En effet, il n'est pas dit que la solution optimale soit donnée dès la première itération, si celle-ci ne renvoie pas un chemin passant par le sommet de départ du chemin optimal, aucun traîtement sur le chemin obtenu ne permet d'obtenir la solution optimale.

Complexité : Il y a $d^-(d)$ éléments dans $\tilde{V}_{out}(d)$, il y a donc $d^-(d)$ appels à la procédure `Dijkstra`. Le calcul de la durée peut se faire en $O(1)$ Donc une compléxité en $O\big(d^-(d)(|V| + |E|)\log|V|\big)$.

IV. Plus court chemin

Algorithm 4 Plus court chemin

```
1: procedure PCC(d, f, G̃, t_α, t_ω)
2:     G̃ ← delimiter(G̃, d, f, t_α, t_ω)
3:     return Dijkstra(d, f, G̃)
```

Idée de l'algorithme : Les graphes sont garantis sans circuit et dans $\tilde{G}$ les arcs sont étiquetés par des poids λ représentant la distance entre deux sommets, les poids sont donc positifs ou nuls, l'utilisation de l'algorithme de Dijkstra sur $\tilde{G}$ retourne donc le plus court chemin entre d et f en terme de distance de manière efficace.

Compléxité : Un seul appel à `Dijkstra` : compléxité en $O((|V| + |E|)\log|V|)$.

3 Modélisation du problème PCC en programmation linéaire

La résolution du problème de plus court chemin avec l'algorithme du simplexe étant un problème usuel de programmation linéaire, nous n'avons rien inventé, et nous sommes inspirés du cours de M. Artigues[1]. Nous en détaillerons tout de même le fonctionnement.

Considérons le graphe suivant, dans lequel nous cherchons le plus court chemin entre les sommets d et f.

Un plus court chemin de type IV minimise la distance λ entre deux sommets, nous ne nous intéressons donc pas aux temps t de départ.

On introduit les variables x_{ij} avec $i, j \in V$ qui vaut 1 si l'arc $i \to j$ est dans le plus court chemin, 0 sinon. Comme on recherche le plus court chemin, la fonction objectif à minimiser est

$$z = \sum_{i,j} \lambda_{ij} x_{ij} \tag{2}$$

où λ_{ij} est le poids de l'arc $i \to j$.

Il y a ensuite trois groupes de contraintes. Pour cela, on considère un *module* qui se déplacerait de sommets en sommets, et on rend compte de sa position en se mettant à la place des sommets.

Premier groupe contraintes : le nombre d'arcs dans le chemin entrant en d est plus petit (de 1) que le nombre d'arcs sortant de d.

$$\sum_{k \in S^+(d)} x_{dk} - \sum_{k \in S^-(d)} x_{kd} = 1 \tag{3}$$

Ainsi on est sûr que le *module* s'échappe du sommet d.

Deuxième groupe de contraintes : dans le chemin optimal, le nombre d'arcs entrants et sortants d'un sommet qui n'est ni le départ ni la fin doivent être égaux

$$\forall i \in V \backslash \{d, f\} : \sum_{k \in S^+(i)} x_{ik} - \sum_{k \in S^-(i)} x_{ki} = 0 \tag{4}$$

Ainsi le *module* ne reste pas dans un sommet, sans s'occuper des cas d et f.

Troisième groupe de contraintes : dans le chemin optimal le nombre d'arcs entrants dans le sommet f de fin est plus grand (de 1) que le nombre d'arcs sortant de ce sommet.

$$\sum_{k \in S^-(f)} x_{kf} - \sum_{k \in S^+(f)} x_{fk} = 1 \tag{5}$$

Ainsi, à la fin, le *module* est dans le sommet f. Finalement, en rassemblant la fonction objectif (2) et les contraintes décrites en (3), (4), et (5), nous obtenons le PL suivant

$$(P) \begin{cases} \texttt{min } z = \sum_{i,j} \lambda_{ij} x_{ij} \\ \\ \sum_{k \in S^+(d)} x_{sk} - \sum_{k \in S^-(d)} x_{ks} = 1 \\ \sum_{k \in S^+(i)} x_{ik} - \sum_{k \in S^-(i)} x_{ki} = 0, \forall i \in V \backslash \{d, f\} \\ \sum_{k \in S^-(f)} x_{kf} - \sum_{k \in S^+(f)} x_{fk} = 1 \\ \\ x_{ij} \in \{0, 1\}, \forall i, j \in V \end{cases} \tag{6}$$

3.1 Implémentation

Le fichier `lp.py` contient l'implementation du programme linéaire permettant de résoudre le problème de plus court chemin. Pour l'illustration, nous cherchons dans le deuxième graphe du sujet le plus court chemin entre les sommets a et f, nous utilisons donc le graphe transformé associé $\tilde{G}$

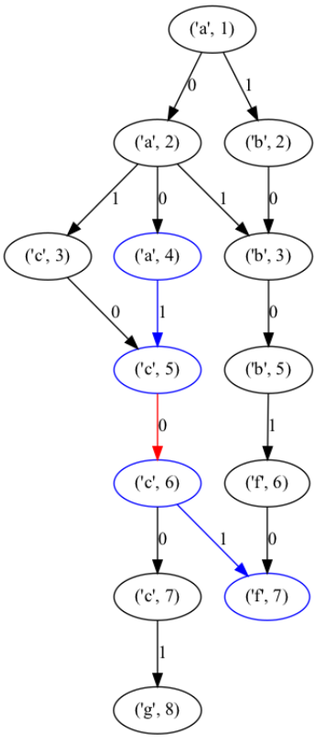

Figure 2: Graphe $\tilde{G}$ utilisé pour notre exemple de résolution

Ce graphe est créé en utilisant la fonction `graph_to_dot`, le plus court chemin est coloré en bleu, l'arc rouge indique un arc qui ne change pas de sommet dans G. A noter : ce plus court

chemin est celui obtenu avec l'exécution de l'algorithme de Dijkstra. Le résultat retourné par le PL peut-être différent, ce qui importe est que le poids des deux chemins soit le même.

En executant le fichier `lp.py`, nous obtenons la sortie suivante :

```
Academic license - for non-commercial use only - expires 2021-12-19
...
Solution count 1:  2

Optimal solution found (tolerance 1.00e-04)
Best objective 2.000000000000e+00, best bound 2.000000000000e+00, gap 0.0000
```

Le chemin optimal obtenu et sa longueur associée sont les suivants :

```
(4, " | x('a', 2),('b', 3) | x('b', 5),('f', 6) | x('b', 3),('b', 5) | x('f', 6),('f', 7) | ")
```

On remarque que les plus courts chemin ne sont pas les mêmes, le programme linéaire retourne le chemin suivant :

$$\texttt{('a', 2)} \rightarrow \texttt{('b', 3)} \rightarrow \texttt{('b', 5)} \rightarrow \texttt{ ('f', 6)} \rightarrow \texttt{('f', 7)} \tag{7}$$

Il est de longueur 4, contrairement au plus court chemin retourné par l'algorithme de Dijkstra qui est de longueur 3, mais les deux ont le même poids : 2.

4 Analyse des performances à plus grande échelle

Les données exactes recueillies pour générer les figures suivantes sont dans l'annexe B.

4.1 Génération des graphes utilisés pour les tests

Pour la génération nous étions donc en mesure de générer des graphes avec un nombre arbitraire de sommets. Dans le cas des réseaux de transport, les graphes ne sont en général pas des graphes complets, et les sommets reliés par une arête sont en général *proches*. Ainsi, deux sommets i et j peuvent être reliés si et seulement si $i \leq j \leq i + 10$ et un tel arc est ajouté avec une probabilité P. Expérimentalement, si P est assez petit, de l'ordre de $0,5$ les graphes semblent *réalistes* (au sens d'un réseau de transport : les sommets sont des arcs sont des routes et les sommets des escales). Comme attendu, le nombre d'arcs des graphes ainsi générés est proportionnel au nombre de sommets de ce graphe, c'est ce que montre la figure 3.

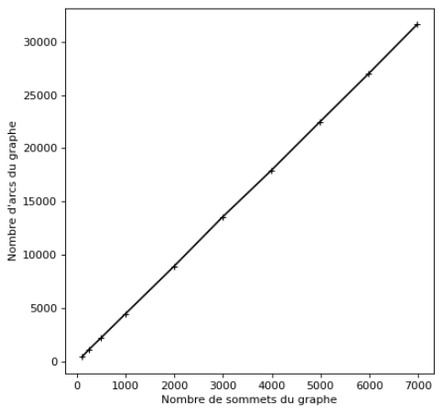

Figure 3: Nombre de d'arcs du graphe obtenu en fonction du nombre de sommets

4.2 Analyse du temps d'exécution du programme linéaire

La figure représente, à gauche, le temps d'écriture des contraines du programme linéaire, et à droite le temps de résolution du programme linéaire une fois les contraines écrites (le temps d'exécution de la fonction `optimize()` de Gurobi).

Le temps d'écriture du PL croit *rapidement* quand le nombre de sommets du graphe traîté augmente, cependant, comme le montre la figure 4, le temps de résolution semble proportionnel au temps d'écriture du PL. Le temps d'écriture du PL étant lié au nombre de contraintes. A noter que Gurobi utilise des heuristiques afin d'approcher la solution, et également des méthodes de *pré-résolution* afin de réduire nombre de contraintes. Il est su qu'une fois les contraintes écrites Gurobi résoud *efficacement* le PL : en temps polynomial par rapport au nombre de contraintes. Gurobi

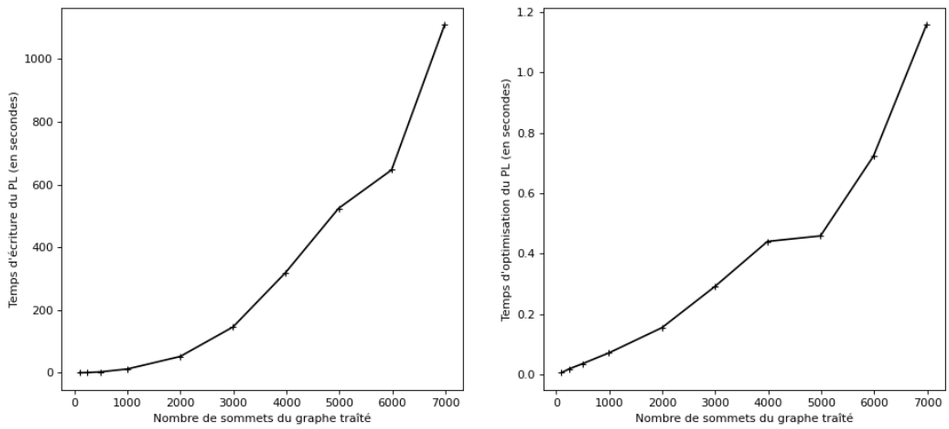

Figure 4: Temps d'écriture et de résolution du PL

est donc efficace, le problème d'efficacité de la résolution de PCC via PL provient du nombre de contraintes nécessaires à la résolution de ce PL.
Essayons d'approximer le temps d'écriture des contraintes pour un graphe comportant nombre arbitraire de sommets. La courbe rouge de la figure 5 provient de l'approxmation suivante : On cherche une approximation polynomiale **simple** (sûrement peu précise) qui représentera l'évolution du temps d'écriture du PL en fonction du nombre de sommets de graphes. On utilise deux points $(x_1, y_1), (x_2, y_2)$, la courbe de la fonction d'approximation passera par ces points. La fonction que l'on cherche est de degré p et est de la forme $f(x) = b(x + a)^p$. En utilisant les points $(99, 0.128)$ et $(5000, 524)$ (obtenus expérimentalement), on a que

$$\begin{cases} 0.128 &= b(99 + a)^2 \\ 524 &= b(5000 + a)^2 \end{cases} \tag{8}$$

Le nombre de contraintes du PL est en $O(|V|^2)$, on cherche donc un polynome du seconde degré; on pose $p = 2$. Pour trouver a et b, on pose $A = \exp\frac{\log(\frac{y_1}{y_2})}{p}$, et

$$\begin{cases} a = \frac{x_1 - A \cdot x_2}{A - 1} \\ b = \frac{y_1}{(x_1 + a)^p} \end{cases} \tag{9}$$

En résolvant numériquement l'équation, on obtient

$$\begin{cases} a \approx -21.2 \\ b \approx 2.12 \cdot 10^{-5} \end{cases} \tag{10}$$

Soit

$$f(x) \approx 2.12 \cdot 10^{-5} * (x - 21.2)^2 \tag{11}$$

C'est la courbe rouge en pointillés sur la figure 5, et l'estimation semble assez proche des résultats obtenus expérimentalement. On peut utiliser f pour calculer un ordre de grandeur du temps d'écriture du PL pour des plus gros graphes. On a, par exemple, que $f(100000) \approx 246509$, c'est à dire qu'il faut environ 246509 secondes pour écrire le PL associé à un graphe comportant 100000 sommets (avant la transformation), soit environ 2 jours, 20 heures et 30 minutes. L'utilisation de la programmation linéaire pour résoudre le problème de plus court chemin semble donc une mauvaise idée pour les gros graphes surtout en utilisant la transformation en $\tilde{G}$.

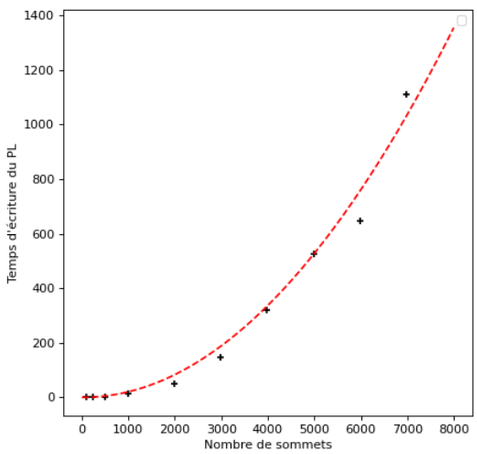

Figure 5: Estimation polynomiale de la durée de l'écriture du PL en fonction du nombre de sommets dans le graphe G de départ

4.3 Analyse du temps d'exécution de l'algorithme de Dijkstra

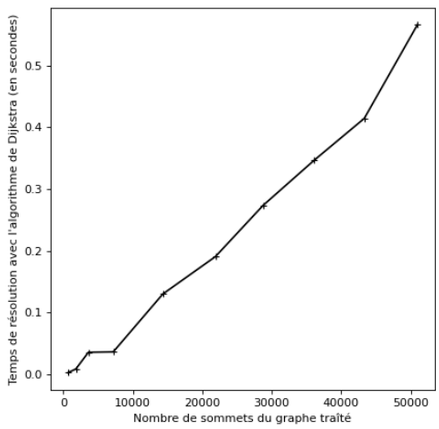

Figure 6: Temps de résolution de l'algorithme de Dijkstra

La figure ci-dessus représente le temps d'éxecution de Dijkstra en fonction du nombre de sommets dans le graphe G. Pour l'implémentation de l'algorithme de Dijkstra, le graphe traîté possède $|V|$ sommets et $|E|$ arcs, le graphe est aussi représenté par des listes d'adjacences et l'implémentation d'une file de priorités donne une compléxité en temps de l'algorithme $O\big((|V| + |E|) \log |V|\big)$. Ce qui correspond à une complexité quasi-linéaire. C'est bien ce que l'on peut observer sur la courbe de la figure 9.

La figure ci-dessus représente le temps d'éxecution de Dijkstra en fonction du nombre de sommets dans le graph G. Pour l'implémentation de l'algorithme de Dijkstra, le graph traité possède $|V|$ sommets et $|E|$ arcs, le graph est aussi représenté par des listes d'adjacences et l'implémentation d'une file de priorités donne une compléxité en temps de l'algorithme $O\big((|V| + |E|) \log |V|\big)$. Ce qui correspond à une complexité quasi-linéaire. C'est bien ce que l'on peut observer sur la courbe de la figure(9).

4.4 Analyse de l'évolution des temps de calcul quand P augmente

Les figures 7 et 8 montrent respectivement l'évolution du temps d'écriture et de résolution du PL. Nous avons utilisé la même méthode que précédemment pour trouver une approximation de l'évolution, en se servant à chaque fois des points en $x_1 = 100$ et $x_2 = 5000$. Nous avons remarqué que les courbes approchaient plus les points en $x = 2500$ lorsque $p = 2.3$, c'est à dire des équation de la forme $f(x) = b(x + a)^{2.3}$. Nous ne pouvons pas conclure de la provenance de ce facteur, cela pourrait venir de la gestion des processus de la machine utilisée, ou bien de la mise en mémoire des graphes : P (pour rappel : P est, lors de la génération des graphes, la probabilité d'avoir un

arc entre deux sommets) étant de plus en plus grand, les graphes de départ sont plus gros et par conséquent il en est de même pour les graphes transformés.

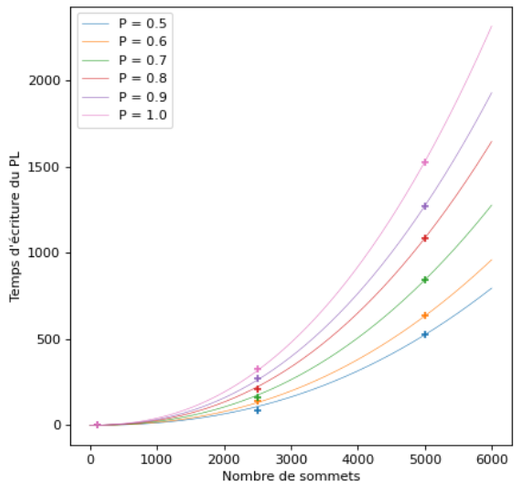

Figure 7: Evolution du temps d'écriture du PL quand P varie

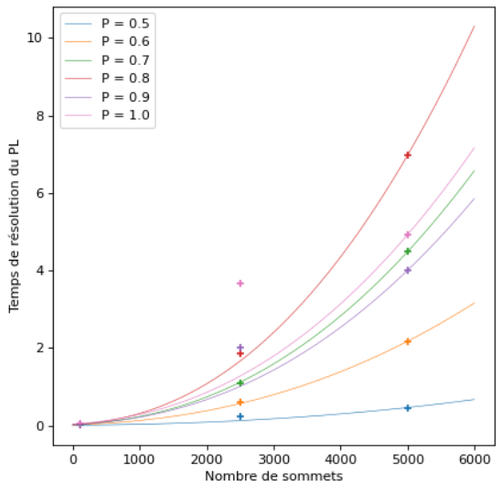

Figure 8: Evolution du temps d'optimisation du PL quand P varie

Pour $P = 0.9$ et $P = 1$, les temps de résolution du PL ne suivent plus la tendance observée pour $P < 0.9$. En effet, Gurobi utilise des nouvelles méthodes de résolution, diminuant drastiquement le temps de résolution. Ce qui indique que Gurobi utilise d'autres méthodes de résolution est le message suivant :

```
Variable types:  0 continuous, 70117 integer (70117 binary)
Found heuristic solution:  objective 936.0000000
Deterministic concurrent LP optimizer:  primal and dual simplex
Showing first log only...
Concurrent spin time:  0.00s
Solved with primal simplex
Root relaxation:  objective 7.890000e+02, 26144 iterations, 4.18 seconds (8.47
work units)
```

Il est donc difficile de conclure sur la compléxité expérimentale du programme linéaire à cause de Gurobi. Cependant, on observe une tendance quadratique quand $P < 0.9$.

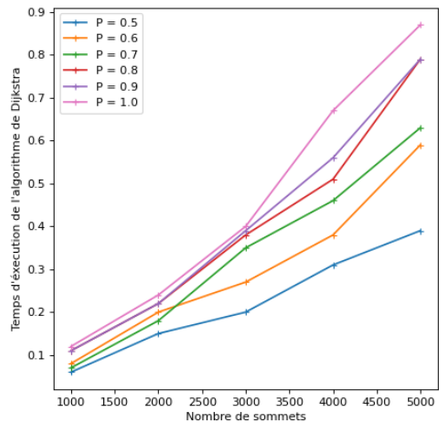

Figure 9: Temps de résolution de l'algorithme de Dijkstra en fonction de P et N

4.5 Analyse du temps de transformation de graphe G en $\tilde{G}$

Pour tous sommet v appartenant à l'ensemble des sommets V du multigraphe initial, soit $\tilde{V} = \tilde{V}_{in} \bigcup \tilde{V}_{out}$ l'ensemble des sommets de $\tilde{G}$, nous avons $v \in \tilde{V}_{in}$ si un arc de G arrive en v, et $v \in \tilde{V}_{out}$

si un arc de G sort de v, par conséquent $|\tilde{V}_{in}| \leq |E|$ et $|\tilde{V}_{out}| \leq |E|$. Ainsi, nous avons

$$|E| \quad \leq \quad |\tilde{V}| \quad \leq \quad 2 \cdot |E| \tag{12}$$

donc l'ordre de grandeur de la borne supérieure de $|\tilde{V}|$ est en $O\big(2 \cdot |E|\big)$.
L'ensemble des arcs $\tilde{E}$ est défini ainsi : pour chaque sommet $v \in V$ avec des arcs sortant ou arrivant en différent temps, soit k le temps d'attente maximum à partir de v, on ajoute les arcs $(v, t_i) \to (v, t_{i+1})$ pour $0 \leq i \leq k$. $|\tilde{E}| \leq |V| \times k$. On ajoute également les arcs $(u, t) \to (v, t + \lambda)$ il y en a $|E|$. Nous avons donc que

$$|E| \quad \leq \quad |\tilde{E}| \quad \leq \quad |V| \cdot k_{max} + |E| \tag{13}$$

d'où un ordre de grandeur pour la borne supérieure de la taille de $\tilde{E}$ en $O\big(|V| \cdot k_{max} + |E|\big)$, k_{max} étant le temps d'attente maximum pour tous les sommets du graphe, défini comme

$$k_{max} = \max_{u \in V} |\tilde{V}_{out}(u)| \tag{14}$$

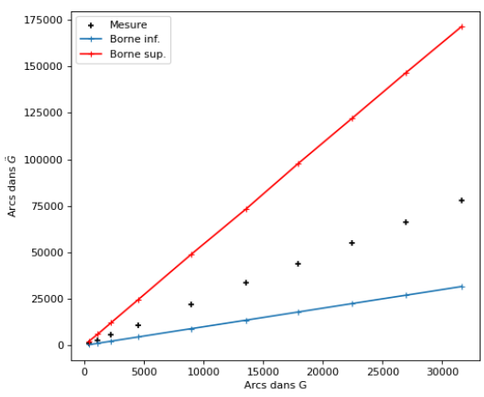

Figure 10: Bornes inférieure et supérieure du nombre d'arcs de $\tilde{G}$ ($k_{max} = 20$)

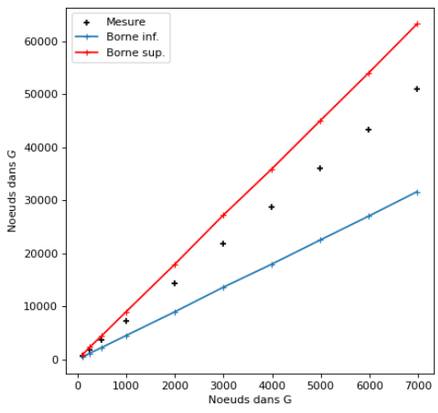

Figure 11: Borne inférieure et supérieure du nombre de noeuds de $\tilde{G}$

4.6 Variation de l'intervalle $[t_\alpha, t_\omega]$

Le temps d'exécution des algorithmes est lié à intervalle $[t_\alpha, t_\omega]$. En effet, après un appel à `delimiter`, les sommets dont les temps associés sont hors de l'intervalle sont supprimés du graphe. Plus l'intervalle est restrictif, plus la résolution est rapide (car la taille du graphe diminue). Cependant, quand l'intervalle se restreint le nombre de chemins dans le graphe diminue. Un intervalle trop restrictif peut donc supprimer tous les chemins entre le sommet de wdépart et un ensemble de sommets d'arrivée.

5 Algorithmes alternatifs

Appendices

A Pseudo-code de la procédure `delimiter`

Pour tout sommet x, $\tilde{V}_{out}(x)$ et $\tilde{V}_{in}(x)$ sont des listes, alors les procédures `supprimer` et `inserer` sont des primitives usuelles.

1: **procedure** DELIMITER($\tilde{G}, d, f, t_\alpha, t_\omega$)
2: **for all** $u \in \tilde{V}_{out}(d)$ **do**
3: $(u, t) \leftarrow u$
4: **if** $t < t_\alpha$ **then**
5: `supprimer`($\tilde{V}_{out}(d), u$)
6: `inserer`($\tilde{V}_{out}(d), (u, \infty)$)
7: **for all** $u \in \tilde{V}_{in}(f)$ **do**
8: $(u, t) \leftarrow u$
9: **if** $t > t_\omega$ **then**
10: `supprimer`($\tilde{V}_{in}(s), u$)
11: `inserer`($\tilde{V}_{in}(s), (u, \infty)$)

Les temps t de *sortie* de d inférieurs à t_α dans $\tilde{G}$, sont remplacés par ∞ : aucun plus court chemin ne pourra contenir les arcs associés à ces temps t, il en est de même pour les temps d'*arrivée* en f. Donc après un appel à la procédure `delimiter`, chaque chemin P retourné respecte les contraintes $debut(P) \geq t_\alpha$ et $fin(P) \leq t_\omega$.

B Temps d'exécution mesurés

Note : Les tests ont tous été réalisés sur la même machine : un MacBook Pro M1 de 2020 doté de 8 Go de mémoire vive.

B.1 Mesures de temps d'exécutions et tailles des graphes pour $P = 0.5$

Noeuds G	Arcs G	Noeuds $\tilde{G}$	Arcs $\tilde{G}$	t. Dijkstra	t. écriture PL	t. opt. PL
99	425	686	1041	0.0027	0.128	0.005
247	1144	1821	2797	0.008	0.794	0.018
497	2244	3621	5533	0.035	3.06	0.035
996	4490	7194	11014	0.036	12.005	0.07
1995	8950	14331	21967	0.130	51.8	0.15
2993	13581	21889	33430	0.190	146.012	0.29
3985	17910	28753	44007	0.273	319.442	0.44
4985	22472	36102	55256	0.346	524.163	0.46
5984	26984	43317	66264	0.414	646.442	0.72
6983	31611	50951	77913	0.565	1108.572	1.157

B.2 Mesures du temps d'écriture et d'optimisation du PL quand P varie

N	P	t. écriture PL (s)	t. opt. PL (s)
100	0.5	0.13	0.005
2500	0.5	87	0.22
5000	0.5	524	0.46
100	0.6	0.18	0.02
2500	0.6	138	0.60
5000	0.6	633	2.17
100	0.7	0.23	0.028
2500	0.7	160	1.08
5000	0.7	841.78	4.48
100	0.8	0.23	0.028
2500	0.9	212	1.86
5000	0.8	1084	6.97
100	0.9	0.31	0.028
2500	0.9	270	2
5000	0.9	1271	4
100	1	0.45	0.044
2500	1	329	3.68
5000	1	1527	4.92

B.3 Mesures du temps d'exécution de l'algorithme de Dijkstra quand P varie

N	P	t. Dijkstra (s)
1000	0.5	0.06
2000	0.5	0.15
3000	0.5	0.20
4000	0.5	0.31
5000	0.5	0.39
1000	0.6	0.08
2000	0.6	0.20
3000	0.6	0.27
4000	0.6	0.38
5000	0.6	0.59
1000	0.7	0.07
2000	0.7	0.18
3000	0.7	0.35
4000	0.7	0.46
5000	0.7	0.63
1000	0.8	0.11
2000	0.8	0.22
3000	0.8	0.38
4000	0.8	0.51
5000	0.8	0.79
1000	0.9	0.11
2000	0.9	0.22
3000	0.9	0.39
4000	0.9	0.56
5000	0.9	0.79
1000	1	0.12
2000	1	0.24
3000	1	0.40
4000	1	0.67
5000	1	0.87

References

[1] Christian Artigues, 2014, https://homepages.laas.fr/artigues/decomp-cours.pdf, p. 35