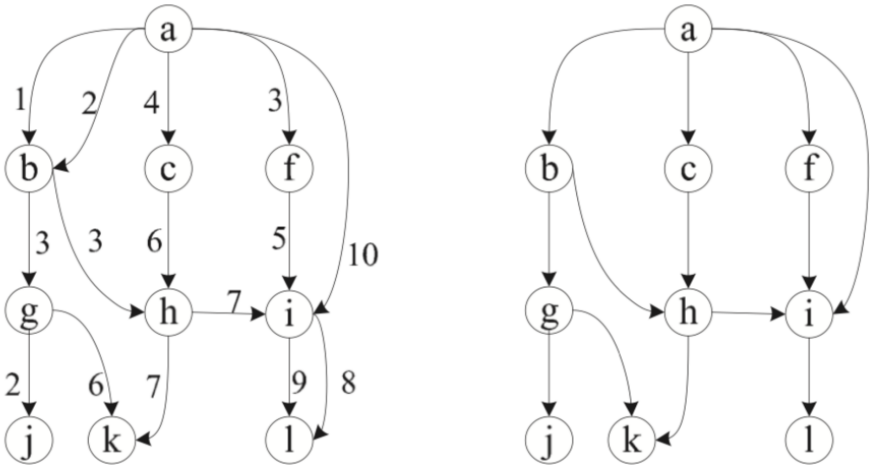
MOGPL

Oriana Rafaela Hernandez Adan , Hugo Riffaud de Turckheim

December 6, 2021

Contents

1 Assertions



1.1

Soit $P1 = ((a, b, 2, 1), (b, g, 3, 1))$, un chemin un chemin d'arrivée au plus-tôt de a à g.
On considère le sous chemin préfixe de $P1$, $P2 = ((a, b, 2, 1))$. P2 n'est pas un chemin d'arrivée au plus tôt car fin(P2) = 3.

Or, $min(fin(P') : P' \in P(a, b, [0, 3])) = 2$.
Un chemin d'arrivée au plus tôt est dans ce cas $P3 = ((a, b, 1, 1)), fin(P3) = 2$.
Donc 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

Soit $P1 = ((f, i, 5, 1), (i, l, 8, 1))$, un chemin de départ au plus tard de f à l.
On considère le sous-chemin postfixe de $P1$, $P2 = ((i, l, 8, 1))$.
$P2$ n'est pas un chemin de départ au plus tard car $debut(P2) = 8$.
Or, $max(debut(P') : P' \in P(i, l, [0, 10])) = 9$.
Un chemin de départ au plus tard est dans ce cas P3 = ((i, l, 19, 1)), début(P3) = 9.
Donc un sous-chemin postfixe d'un chemin de départ au plus tard peut ne pas être un chemin de départ au plus tard.

1.3

Soit P1 = ((a, c, 4, 1), (c, h, 6, 1), (h, k, 7, 1)), chemin le plus rapide de a à k.
On considère le sous-chemin de P1, P2 = ((a, c, 4, 1), (c, h, 6, 1)).
P2 n'est pas un chemin le plus rapide car durée(P2) = 7-4 = 3.
Or min(durée(P') : P' ∈ P(a, h, [0,7])) = 2.
Un chemin le plus rapide est dans ce cas P3 = ((a, b, 2, 1), (b, h, 3, 1)) avec durée(P3) = 4-2 = 2.
Donc un sous-chemin d'un chemin le plus rapide peut ne pas être un chemin le plus rapide.

1.4

Soit P1 = ((a, f, 3, 1), (f, i, 5, 1), (i, l, 8, 1)), un chemin le plus court de a à l.
On considère le sous-chemin de P1, P2 = ((a, f, 3, 1), (f, i, 5, 1)).
P2 n'est pas un chemin le plus court car dist(P2) = 1+1 = 2.
Or min(dist(P') : P' ∈ P(a, i, [0,11])) = 1.
Un chemin le plus court est dans ce cas P3 = ((a, i, 10, 1)), dist(P3) = 1.
Donc un sous-chemin d'un chemin le plus court peut ne pas être un chemin le plus court.

2 Algorithmes proposés

La compléxité des différents algos à notre disposition, pour un graphe (V,E) est :

- tri topologique puis relachement des arcs : $O(|V|+|E|)$
- Bellman-Ford: $O(|V|*|E|)$
- Djikstra: $O(\ |V| \cdot \log\ (|V|) + |E|\ )$

Nous avons fait le choix d'implémenter le premier algorithme, (tri topologique puis relachement des arcs) pour plusieurs raisons:

- Il n'est pas nécéssaire d'implémenter l'algorithme de Bellman-Ford ou Djikstra car nous n'avons pas de cycles dans le graphe.
- Une autre raison de ne pas implémenter Bellman-Ford est qu'il n'y a aucun arc de poids négatif sur les graphs étudiés ici (et pas de cycle de poids négatif).
- Enfin la compléxité de l'algorithme que nous avons choisi est linéaire contrairement à la compléxité des deux autres algorithmes envisagés.

Pour chaque problème on considérera le Graphe G' , étant le graphe G transformé comme proposé dans l'énoncé.

2.1 - Chemin d'arrivée au plus tôt

Pour traiter le problème du chemin d'arrivée au plus tôt nous nous sommes inspirés de la méthode des pénalités précédemment vue en cours , dans le cadre de l'algorithme du SIMPLEX.

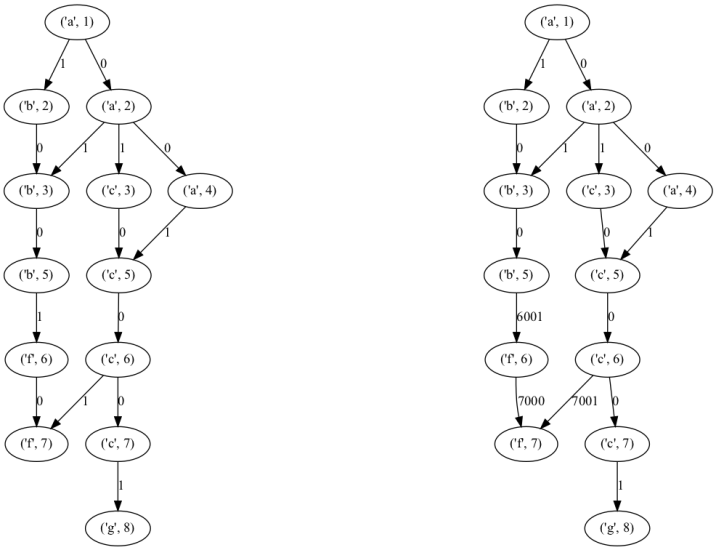
Soit $e \in G$ le sommet d'arrivée choisi et $e_i \in G'$ l'ensemble des sommets d'arrivée à e aux dates i. Pour tous les arcs entrant de e_i resp. (v, e_i, t, λ) on modifie le poids de l'arc par:

$$\lambda = \lambda + i * M$$

On a bien que plus t est grand , plus l'arc est pénalisé. (il n'y a aucune date négative et aucun poids négatif dans aucun cas la pénalité deviendrait un "bonus"). Dans notre cas on a fixé M à 1000 par soucis de simplicité mais une valeur correcte de M serait pour $Pmax_{G'}$ le poids maximal λ d'un arc de G' et $E_{G'}$ le nombre d'arcs de G' :

$$M_{G'} = Pmax_{G'} * E_{G'} * 10$$

On a bien que la $M_{G'}$ a un score de distance bien superieur au plus court chemin même dans le cas ou il y aurait un grand nombre d'arcs.



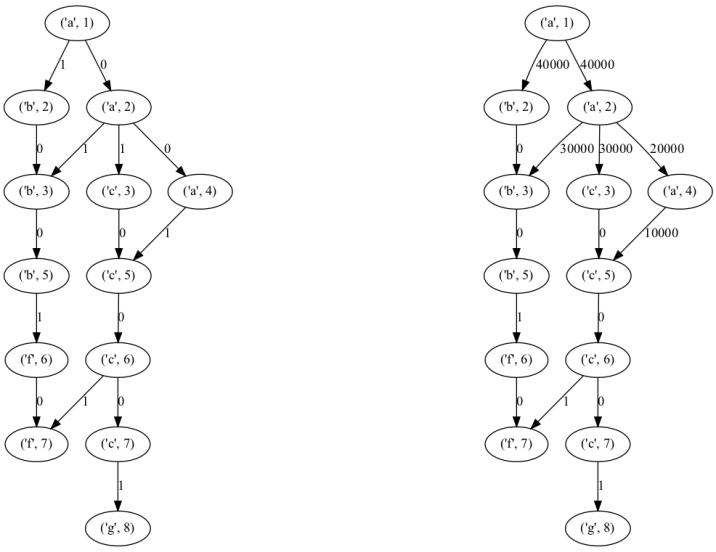
transformation de G' après application des pénalités pour le chemin d'arrivée au plus tôt *a-f*

2.2 - Chemin de départ au plus tard

Pour ce problème nous avons appliqué la même méthode que pour le problème précédent cependant on a pénalisé tous les arcs sortants des sommets s_i avec t_{max} la dernière date de départ partant de s par:

$$\lambda = \lambda + (t_{max} - i) * M$$

on a bien que plus i est petit, plus $t_{max} - i$ est grand et l'arc pénalisé.



transformation de G' après application des pénalités pour le chemin d'arrivée au plus tard a-f

2.3 - Chemin le plus rapide

La procédure reste inchangée par rapport à ce que nous avons fait avant. Cependant nous avons changé la fonction de cout entre les différents arcs. Avant le coût d'un arc était déterminé par sa distance , et dans ce cas il est déterminé par son temps de trajet.

2.4 - Plus court chemin

Le plus court chemin est le même algorithme que celui utilisé précédemment , sans aucune pénalité ni changement de cout. (tri topologique puis relâchement des arcs)

3 Compléxité des algorithmes proposés

Dans cette partie, nous allons présenter les compléxités des algorithmes que nous allons implémenter pour les différents problèmes proposés.

Algorithm 1 Plus-Courts-Chemins-GSS (2.4)

Input : $G' = (V, E), sommet_{start}, sommet_{end}$
Output : plus court chemin de $sommet_{start}$ **à** $sommet_{end}$

distances ← dict()
chemins ← dict()
trier topologiquement les sommets de G'
for chaque sommet $u \in G'$ pris dans l'ordre topologique **do**
  for chaque sommet $v \in Adj[u]$ **do**
    Relacher(u,v,$cout_{u,v}$,chemins,distances) G'
  end for
end for
cheminRes ← Retrace-Chemin($sommet_{start}, sommet_{end}$,chemins,distances)

return cheminRes

Pour calculer le plus court chemin d'un graphe G entre les sommets s_{start} et s_{end} nous appelons Plus-Courts-Chemins-GSS avec les paramètres suivants : G'=(V,E) , s_{start} , s_{end}.

Le tri topologique des sommets de G' s'effectue en $O(|\mathrm{V}|+|\mathrm{E}|)$ opérations.
La procédure Relacher se fait en $O(1)$. Elle est appelée pour chaque arc de G' soit $|\mathrm{E}|$ fois au total.
A la fin des deux boucles for, *distances* contient la distance minimale de chaque sommet $s_i \in G'$ au sommet de départ s_{start}.
Le dictionaire *chemins* contient pour chaque $s_i \in G'$ le prédécésseur de s_i dans le chemin le plus court s_{start}, s_i. Si il n'existe pas un tel chemin alors chemins[s_i]=*vide*.
La procédure Retrace-Chemin conciste simplement à choisir le sommet s_{end} avec la distance à s_{start} minimale. Ensuite il suffit simplement de remonter chemins à partir de chemins[s_{end}] jusqu'a rencontrer un sommet d'étiquette s_{start} ou *vide* puis retourner le chemin obtenu. Dans le pire des cas l'opération $min(distances, s_{end})$ s'effectue en $O(|V|)$ et remonter *chemins* en $O(|V|)$. (le pire des cas étant un chemin qui parcourt tous les sommets de G').

La compléxité de $Plus-Courts-Chemins-GSS$ **(2.4) est** $O(|V|+|E|)$**.**

La compléxité des algorithmes 2.1 et 2.2 est très proche de la compléxité de l'algorithme qu'on vient juste de voir puisque le seul changement qu'on apporte est de pénaliser certains arcs du graphe G' avant de trier ses sommets. La procédure de pénalité des arcs de G' s'effectue en $O(|E|)$ opérations. (Le pire cas étant qu'il faut pénaliser tous les arcs de G').

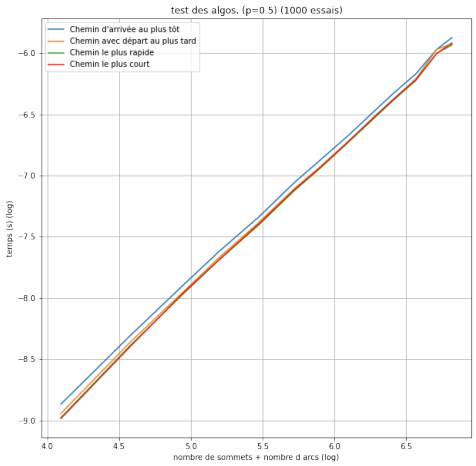
La compléxité des algorithmes vus dans les sections 2.1 **et** 2.2 **est** $O(|V|+|E|)$**.**

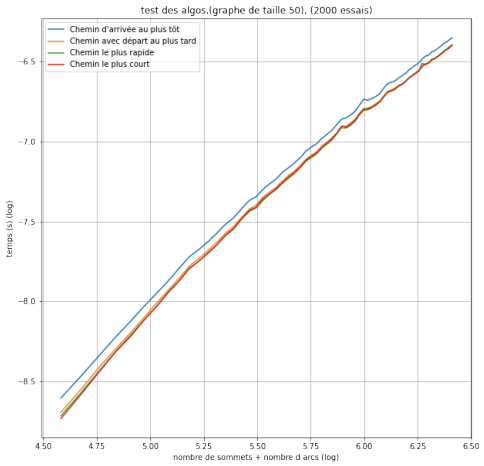
La compléxité de l'algorithme 2.3 est égale à la compléxité de l'algorithme $Plus-Courts-Chemins-GSS$ car on ne fait que changer le côut de chaque arc dans la procédure *Relacher*. Le cout précédent était le poids de l'arc (u,v). On le change par la différence de temps $v_t - u_t$ (information présente dans chaque arc du graphe, comme proposé dans l'énnoncé).

La compléxité de l'algorithme 2.3 **est** $O(|V|+|E|)$**.**

4 Implémentation des algorithmes proposés

Tous les algorithmes vus dans la partie 2 ont été implémentés en python dans le Jupyter Notebook associé au fichier de rendu. Afin de tester l'éfficacité de notre implémentation et vérifier la cohérence des compléxités trouvées dans la partie 3, nous avons fait des tests sur nos algos.





à gauche , on a fait fait varier le nombre de sommets de G en gardant la probabilité d'appartion P des arcs fixe pendant la création des graphes.
à droite , on a fait fait varier la probabilité d'appartion P des arcs en gardant le nombre de sommets fixe pendant la création des graphes.

Comme on peut le voir les deux courbes obtenues sont bien linéaires. la courbe de gauche a un coefficient directeur $\alpha_V \approx 1.04$. la courbe de droite a un coefficient directeur $\alpha_E \approx 1.1$. On peut estimer que notre algorithme a une complexité $O(|V|^{1.04} + |E|^{1.1})$. Ce qui semble cohérent avec les estimations théoriques faites précédemment.

Il n'est pas nécéssaire de faire des tests en gardant le nombre de sommets et la probabilité d'apparition des arcs fixes tout en faisant varier les poids / étiquettes sur les arcs.En effet le seul impact que les étiquettes et les poids des arcs ont sur le code est le changement de la fonction de coût appelée dans *Relacher*. Cette opération s'effectue dans tous les cas en $O(1)$.

5 Modélisation et implémentation du problème avec GUROBI

On considère la transformation $G' = (V, E)$ d'un multigraphe orienté sans circuit, proposée dans l'exemple 2. Les sommets $S_{start}, S_{end} \in G$ respéctivement le sommet de départ et d'arrivée pour lesquels on aimerait connaître le plus court chemin , si il existe dans G.

On cherche à minimiser $dist(P') : P' \in P(x, y, [t_\alpha, t_\omega])$.
On pose les variables X_{ij}, avec $i, j \in V$ et X_{ij} une variable binaire représentant la présence de l'arc (i, j) dans le chemin optimal. Pour minimiser la distance, il suffit de minimiser la somme des variables X_{ij} si elles représentent des arcs de changement de sommet du graphe G initial, c'est à dire si elles ont un poids de 1 dans G'.

On obtient alors l'objectif :

$$\min z = \sum_{i,j}^{E} C_{ij} \cdot X_{ij} \text{ avec Cij le poids de l'arc de i a j}$$

On veut que les arcs choisis constituent bien un chemin dans le graphe du sommet s au sommet t, on représente cela en ajoutant les contraintes suivantes :

- Pour les sommets autres que s et t, la somme des arcs entrants doit être égale à la somme des arcs sortants.
- Pour s il doit y avoir un arc de plus sortant (le premier arc du chemin).
- Pour t il doit y avoir un arc de plus entrant (le dernier arc du chemin).

On obtient pour tout i la contrainte suivante:

$$\sum_j^E X_{ij} - \sum_j^E X_{ji} = \begin{cases} 1 \text{ si } i = s_{start} \\ -1 \text{ si } i = s_{end} \\ 0 \text{ sinon} \end{cases}$$
$$\forall i, j \in E,\ X_{ij} \in \{0, 1\}$$

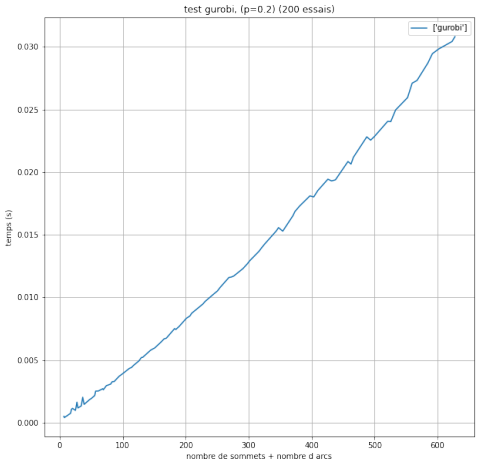
Finalement on définit les sommets $s_{start}, s_{end} \in G'$ à partir des sommets $S_{start}, S_{end} \in G$.

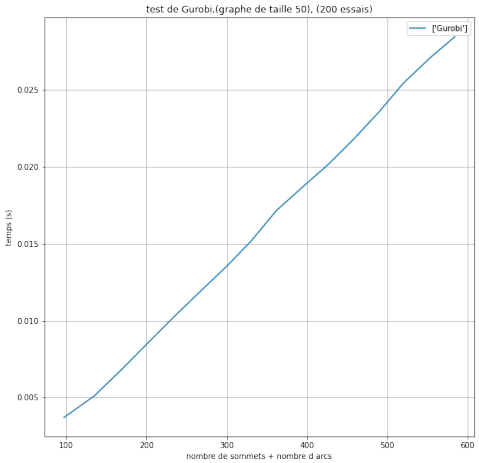
- s_{start} étant le sommet d'identification (nom, date) avec le même nom que S_{start} et la date la plus petite (car cela force à considérer toutes les possibilités de sorties de S).
- s_{end} étant le sommet d'identification (nom, date) avec le même nom que S_{end} et la date la plus grande (car cela force à considérer toutes les possibilités d'entrée en T).

On obtient ainsi une modélisation du problème par programmation linéaire, solvable avec GUROBI.

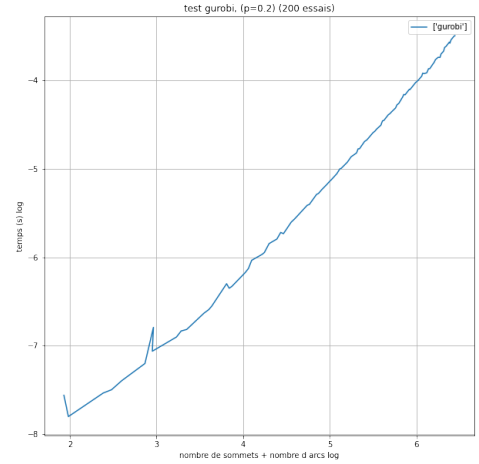
6 Tests de compléxité de l'agorithme avec GUROBI

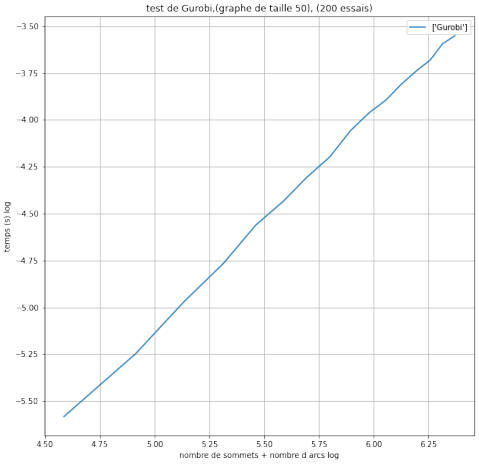
Apres l'implementation de la modelisation du probleme par programmation lineaire en python dans le Jupyter Notebook grace a GUROBI et la realisation de tests de complexite sur l'algorithme, on obtient les courbes suivantes.





de même, on fait varier le nombre de sommets à gauche et la probabilité d'apparition des arcs à droite.

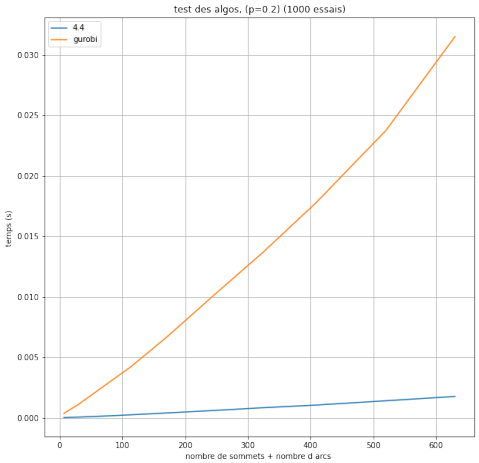


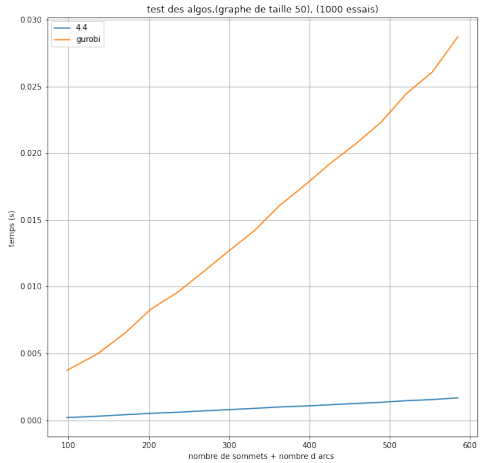


ci-dessus les courbes en échelle logarithmique.

On peut constater que la courbe de variation du nombre de sommets semble exponentielle tandis que celle de variation de la probabilite d'apparition des arcs semble lineaire.

7 Comparaison entre GUROBI et l'algorithme IV. proposé





de même, on fait varier le nombre de sommets à gauche et la probabilité d'apparition des arcs à droite.
en jaune, temps d'exécution en fonction de la taille du graphe de l'algorithme utilisant GUROBI.
en bleu , temps d'exécution en fonction de la taille du graphe des algorithmes implémentés dans la partie 4.

Comme on peut le voir l'algorithme basé sur GUROBI est beaucoup moins performant que l'algorithme Plus-Courts-Chemins présenté en 2.4. Cela semble logique étant donné que le premier algorithme semble avoir une compléxité exponentielle alors que le second est linéaire.

8 Validité des Algorithmes

Afin de tester la validité de l'algorithme présenté en 2.4 et l'algorithme basé sur GUROBI nous avons comparé la taille des plus-courts-chemins renvoyés par les deux algorithmes en faisant varier la taille des graphes et la probabilité d'apparition des arcs lors de la génération des graphes.

	p=0.1	p=0.2	p=0.3	p=0.4	p=0.5	p=0.6	p=0.7	p=0.8	p=0.9
t=10	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=20	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=30	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=40	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=50	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=60	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=70	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=80	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=90	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=100	ok	ok	ok	ok	ok	ok	ok	ok	ok

tests réalisés sur 10.000 essais par catégorie, t = la taille des graphes générés et p la probabilité d'apparition des arcs.

On peut supposer que les algorithmes 2.4 **et** *Gurobi* **sont valides.**

Il n'est pas forcément nécéssaire de faire varier à la fois les paramètres t et p lors du test de validité. On aurait simplement pu faire 10.000 tests sur des graphes avec t et p fixés. Cependant faire varier t et p permet de maximiser les chances de tomber des des graphes "particuliers" et donc de bien pouvoir assurer que nos algorithmes fonctionnent pour tout type de graphes.

9 Question subsidiaire

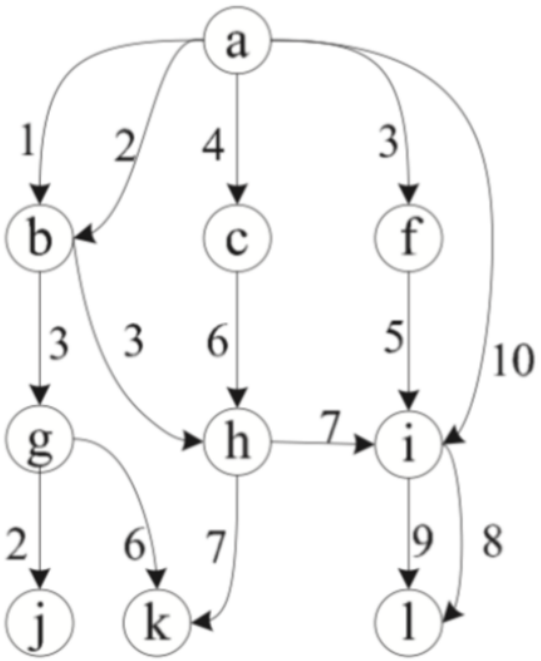
L'algorithme que nous avons implémenté pour répondre à la question subsidiaire (appelé S.IV dans la suite) est assez proche de celui présenté dans la partie 2.4.

9.1 - présentation de l'algorithme proposé

Etant donné que G est un graphe acyclique , on peut toujours utiliser l'algorithme que nous avons présenté précédemment. C'est à dire :

- trier topologiquement G
- relacher les arcs des sommets du graphe dans l'ordre topologique et mettre à jour les distances et pour chaque sommet associé qui un sommet "précédent" avec la distance minimale vers le sommet de départ.
- remonter le chemin ainsi créé du sommet d'arrivée vers le sommet de départ.

L'intérêt de transformer G en G' est de ne plus avoir besoin de faire "attention" aux dates lors du parcours du graphe, puisque chaque noeud était relié aux noeuds qui lui sont toujours accessibles. Les sommets associés dans la liste de chemins optimisent la distance au point de départ cependant dans certains cas ils ne sont pas accessibles. (l'algorithme optimise la distance sans faire attention aux dates)




```
a  :  []
b  :  [('a', 'b', 2, 1), ('a', 'b', 1, 1)]
c  :  [('a', 'c', 4, 1)]
f  :  [('a', 'f', 3, 1)]
g  :  [('b', 'g', 3, 1)]
h  :  [('b', 'h', 3, 1), ('c', 'h', 6, 1)]
i  :  [('h', 'i', 7, 1), ('f', 'i', 5, 1), ('a', 'i', 10, 1)]
j  :  [('g', 'j', 2, 1)]
k  :  [('g', 'k', 6, 1), ('h', 'k', 7, 1)]
l  :  [('i', 'l', 8, 1), ('i', 'l', 9, 1)]
['a', 'f', 'i', 'l']
```

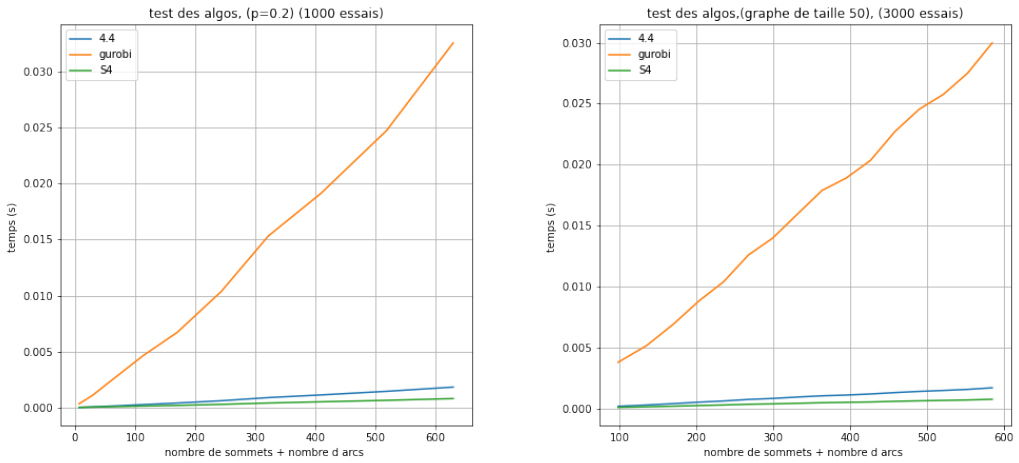
contenu de chemins pour le graphe présenté au dessus

dans la figure ci-dessus on peut voir le contenu de "chemins" pour chaque sommet du graphe pour le parcours: $s_{start} = a$ et $s_{end} = l$. L'idée est de vérifier les dates en remontant le chemin constitué par les arcs optimaux qu'on vas emprunter, et de ne garder que les chemins avec des arcs accessibles.

- On commence par le dernier élément de chemins de s_{end} soit l'arc $(i, l, 9, 1)$ il est bien accessible car on a pas de date maximale d'arrivée.
- On va donc partir du principe que l'arc $(i, l, 9, 1)$ fait parti du chemin optimal. le sommet de départ de cet arc est i. On va donc aller vérifier les arcs associés à chemin$[i]$ tout en gardant que la date limite d'arrivée au sommet suivant est 9.
- l'arc optimal associé à chemin$[i]$ est $(a, i, 10, 1)$. On ne peut pas emprunter cet arc car en l'utilisant on arriverait en i à $t = 11$. Ce qui rend l'emprunt de l'arc suivant $(i, l, 9, 1)$ impossible. On va re-tester avec les deuxième arc le plus optimal contenu dans chemin$[i]$ soit $(f, i, 5, 1)$. on arrive bien en $t = 6$ en i. le chemin est encore valide.
- on procède ainsi jusqu'a revenir à chemin$[s_{start}]$ puis on renvoie le chemin ainsi construit en empruntant les arcs séléctionnés.
- Dans le cas ou chemin$[i]$ ne contenait que l'arc $(a, i, 10, 1)$ on serait remontés au sommet suivant donc l et aurions re-testé avec les autres arcs contenus dans chemin$[l]$. Si aucun arc avait fonctionné on aurait conclu qu'il n'y a pas de chemin entre s_{start} et s_{end} dans G.

9.2 - Comparaisons entre 2.4,GUROBI et S.IV

Nous avons réalisé les mêmes tests que ceux présentés dans la partie 6 en intégrant l'algorthme S.IV.



à gauche on fait varier le nombre n de sommets en gardant la probabilité d'appation des arcs p fixe. à droite l'inverse. la courbe verte représente S.IV , la jaune GUROBI et la bleue 2.4.

Comme on peut le voir l'algorithme S.IV est plus performant que les algorithmes proposés précédemment. Cela peut s'éxpliquer par le fait que le transformation de G en G' amène à créer beaucoup de sommets dans G' qui n'existaient pas dans G.

9.3 - tests de validité

Afin de tester la validité de l'algorithme S.IV nous avons comparé la taille des plus-courts-chemins renvoyés par S.IV et 2.4 en faisant varier la taille des graphes et la probabilité d'apparition des arcs lors de la génération des graphes.

	p=0.1	p=0.2	p=0.3	p=0.4	p=0.5	p=0.6	p=0.7	p=0.8	p=0.9
t=10	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=20	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=30	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=40	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=50	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=60	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=70	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=80	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=90	ok	ok	ok	ok	ok	ok	ok	ok	ok
t=100	ok	ok	ok	ok	ok	ok	ok	ok	ok

tests réalisés sur 10.000 essais par catégorie, t = la taille des graphes générés et p la probabilité d'apparition des arcs.

On peut supposer que l'algorithme $S.IV$ est valide.

Il n'est pas forcément nécéssaire de faire varier à la fois les paramètres t et p lors du test de validité. On aurait simplement pu faire 10.000 tests sur des graphes avec t et p fixés. Cependant faire varier t et p permet de maximiser les chances de tomber des des graphes "particuliers" et donc de bien pouvoir assurer que nos algorithmes fonctionnent pour tout type de graphes.