Sorbonne Université 2021 - Master d'Informatique
MOGPL - Modélisation, Optimisation, Graphes et Programmation Linéaire

Multi-graphes orientés pondérés par le temps

Kiara GIGACZ, Alessia LOI
Groupe 3

Question 1. En utilisant l'instance de la figure gauche de l'exemple 1 ou une autre instance, montrer que les assertions suivantes sont vraies.

Dans un multigraphe orienté sans circuit avec des contraintes de temps :

- 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.

  Considérons les chemins de x = a à y = l dans G de l'exemple 1.
  Pour tout parcours réalisable (qui respecte les contraintes de vol) de a à l, la date d'arrivée au plus tôt est le jour 9.
  Considérons un parcours particulier P de date d'arrivée = 9 :
  P = ((a, c, 4, 1), (c, h, 6, 1), (h, i, 7, 1), (i, l, 8, 1)) fin(P) = 8+1 = 9

  Considérons le chemin P', sous-chemin de P, réalisable et préfixe, allant de x = a à y = i, obtenu en retirant le dernier sommet l. Or, P' n'est pas le chemin d'arrivée au plus tôt de a à i, car il existe le chemin P2 de a à i ayant une date d'arrivée antérieure :
  P' = ((a, c, 4, 1), (c, h, 6, 1), (h, i, 7, 1)) fin(P') = 7+1 = 8
  P2 = ((a, f, 3, 1), (f, i, 5, 1)) fin(P2) = 5+1 = 6

  Conclusion : P est un chemin d'arrivée au plus tôt de a à l, P' est un sous-chemin de P et va de a à i, P n'est pas le chemin d'arrivée au plus tôt de a à i car il existe le chemin P2 de a à i et de fin(P2) < fin(P').
  Donc, un sous-chemin préfixe d'un chemin d'arrivée au plus tôt peut ne pas être un chemin d'arrivée au plus tôt.

- 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.

  Considérons les chemins de x = b à y = k dans G de l'exemple 1.
  Parcours réalisables de b à k en respectant les contraintes de vol :

P1 = ((b, g, 3, 1), (g, k, 6, 1)) début(P1) = 3
P2 = ((b, h, 3, 1), (h, k, 7, 1)) début(P2) = 3

Considérons le chemin P1', sous-chemin de P1, réalisable et postfixe, allant de x = g à y = k, obtenu en retirant le premier sommet b. Or, P1' n'est pas le chemin de départ au plus tard par rapport à la destination k, car il existe le chemin P3 de h à k ayant une date de départ postérieure :

P1' = ((g, k, 6, 1)) debut(P1') = 6
P3 = ((h, k, 7, 1)) debut(P3) = 7

Conclusion : P1 est un chemin de départ au plus tard de b à k, P1' est un sous-chemin de P1 et va de g à k, P1' n'est pas le chemin de départ au plus tard pour arriver à k, car il existe le chemin P3 de h à k et de début(P3) > debut(P1').
Donc, un sous-chemin postfixe d'un chemin de départ au plus tard peut ne pas être un chemin de départ au plus tard.

- Assertion 3 : Un sous-chemin d'un chemin le plus rapide peut ne pas être un chemin le plus rapide.

Considérons les chemins de x = a à y = k dans G de l'exemple 1.
Parcours réalisables de a à k en respectant les contraintes de vol :

P1 = ((a, c, 4, 1), (c, h, 6, 1), (h, k, 7, 1)) durée(P1) = (7+1) - 4 = 4
P2 = ((a, b, 1, 1), (b, h, 3, 1), (h, k, 7, 1)) durée(P2) = (7+1) - 1 = 7
P3 = ((a, b, 2, 1), (b, h, 3, 1), (h, k, 7, 1)) durée(P3) = (7+1) - 2 = 6
P4 = ((a, b, 1, 1), (b, g, 3, 1), (g, k, 6, 1)) durée(P4) = (6+1) - 1 = 6
P5 = ((a, b, 2, 1), (b, g, 3, 1), (g, k, 6, 1)) durée(P5) = (6+1) - 2 = 5

Chemin le plus rapide : P1, car durée(P1) = min {durée(Pi)} avec i ϵ {1, 2, 3, 4, 5}.

Considérons le chemin P1', sous-chemin de P1, allant de x = a à y = h. Or, P1' n'est pas le chemin le plus rapide de a à h, car il existe le chemin P3' de a à h et de durée inférieure :

P1' = ((a, c, 4, 1), (c, h, 6, 1)) durée(P1') = (6+1) - 4 = 3
P3' = ((a, b, 2, 1), (b, h, 3, 1)) durée(P3') = (3+1) - 2 = 2

Conclusion : P1 est le chemin le plus rapide de a à k, P1' est un sous-chemin de P1 et va de a à h, P1 n'est pas le plus rapide chemin de a à h car il existe le chemin P3 de a à h et de durée(P3') < durée(P1').
Donc, un sous-chemin d'un chemin le plus rapide peut ne pas être un chemin le plus rapide.

- Assertion 4 : Un sous-chemin d'un plus court chemin peut ne pas être un plus court chemin.

  Considérons les chemins de x = a à y = l dans G de l'exemple 1.
  Parcours réalisables de a à l en respectant les contraintes de vol :

  P1 = ((a, b, 1, 1), (b, h, 3, 1), (h, i, 7, 1), (i, l, 8, 1)) dist(P1) = 4
  P2 = ((a, b, 1, 1), (b, h, 3, 1), (h, i, 7, 1), (i, l, 9, 1)) dist(P2) = 4
  P3 = ((a, b, 2, 1), (b, h, 3, 1), (h, i, 7, 1), (i, l, 8, 1)) dist(P3) = 4
  P4 = ((a, b, 2, 1), (b, h, 3, 1), (h, i, 7, 1), (i, l, 9, 1)) dist(P4) = 4
  P5 = ((a, c, 4, 1), (c, h, 6, 1), (h, i, 7, 1), (i, l, 8, 1)) dist(P5) = 4
  P6 = ((a, c, 4, 1), (c, h, 6, 1), (h, i, 7, 1), (i, l, 9, 1)) dist(P6) = 4
  P7 = ((a, f, 3, 1), (f, i, 5, 1), (i, l, 8, 1)) dist(P7) = 3
  P8 = ((a, f, 3, 1), (f, i, 5, 1), (i, l, 9, 1)) dist(P8) = 3

  Plus court chemin : P7 ou P8, car dist(P7) = dist(P8) = min {dist(Pi)} avec i ϵ {1,2,3,4,5,6,7, 8}.

  Considérons le chemin P7', sous-chemin de P7, allant de x = a à y = i. Or, P7' n'est pas le chemin le plus court de a à i, car il existe le chemin P9 de a à i et de distance inférieure :

  P7' = ((a, f, 3, 1), (f, i, 5, 1)) dist(P7) = 2
  P9 = ((a, i, 10, 1)) dist(P9) = 1

  Conclusion : P7 est un plus court chemin de a à l, P7' est un sous-chemin de P7 et va de a à i, P7' n'est pas le plus court chemin de a à i car il existe le chemin P9 de a à i et de dist(P9) < dist(P7').
  Donc, un sous-chemin d'un plus court chemin peut ne pas être un plus court chemin.

Question 2. En utilisant comme base cette transformation de G à G', montrer comment on peut calculer de manière efficace les 4 types de chemins minimaux définis précédemment. Il sera peut-être nécessaire d'adapter G' de manière appropriée.

Algorithme I : chemin d'arrivée au plus tôt

- Entrée : un graphe G'=(V,E) traduisant un multigraphe pondéré par le temps, une source x, une destination y et un intervalle de temps [t1, t2]
- Déroulement :
  1. instancier une liste L avec les noeuds contenant y dans leur étiquette classés par ordre de t croissant
  2. pour chaque sommet dans la liste L, vérifier s' il existe un chemin de x à y en remontant le sens des arcs de G'
  3. l'algorithme s'arrête au premier chemin de x à y trouvé
- Sortie : si il existe un chemin de x à y, alors ceci est un chemin d'arrivée au plus tôt de x à y ; sinon, il n'existe pas de chemin de x à y.

Modélisation : à la création du graphe simple issu de la transformation du multigraphe original, nous créons aussi une liste d'adjacence. Puis, nous créons un arbre couvrant avec l'algorithme Breadth-First Search à partir de la racine que nous déterminons en cherchant le sommet

- contenant x dans son étiquette
- avec date minimale
- et date incluse dans l'intervalle précisé en paramètre.

Partir du sommet avec la date minimale nous assure que toutes les copies du noeud x d'origine qui sont dans l'intervalle seront parcourues. Ainsi, nous n'avons besoin de parcourir l'arbre qu'une seule fois, malgré les sommets de départ différents possibles. Pour optimiser, nous arrêtons la recherche sur un chemin particulier quand le nœud rencontré est hors de l'intervalle (car tous ses successeurs seront hors de l'intervalle aussi).
L'arbre couvrant ainsi instancié est interrogé grâce à une méthode traceback qui permet de parcourir l'arbre d'un certain nœud vers la racine et obtenir un chemin s'il en existe un. Nous recherchons un sommet d'arrivée avec date minimale >= t1, donc nous parcourons les différentes copies de x dans l'ordre croissant puis nous retournons le premier chemin trouvé.

Algorithme II : chemin de départ au plus tard

- Entrée : un graphe G'=(V,E) traduisant un multigraphe pondéré par le temps, une source x, une destination y et un intervalle de temps [t1, t2]
- Déroulement :
    1. instancier une liste L avec les noeuds contenant x dans leur étiquette classés par ordre de t décroissant
    2. pour chaque sommet dans la liste L, vérifier s' il existe un chemin de x à y en suivant le sens des arcs de G'
    3. l'algorithme s'arrête au premier chemin de x à y trouvé
- Sortie : si il existe un chemin de x à y, alors ceci est un chemin de départ au plus tard de x à y ; sinon, il n'existe pas de chemin de x à y.

Modélisation : de maniere symetrique à l'implémentation de l'algorithme I, nous avons prévu la création d'un arbre couvrant "inversé", construit avec BFS à partir de la racine contenant y dans son étiquette et date maximale incluse dans l'intervalle précisé en paramètre, et une liste d'adjacence inversée (construite au même moment que la liste d'adjacence normale). Puis nous utilisons la méthode traceback pour obtenir un chemin, s'il en existe un, en parcourant les différentes copies de y dans l'ordre décroissant. Nous retournons le premier chemin trouvé.

Algorithme III : chemin le plus rapide

- Entrée : un graphe G'=(V,E) traduisant un multigraphe pondéré par le temps, une source x, une destination y et un intervalle de temps [t1, t2]
- Déroulement :
    1. instancier une liste L avec des doublets (noeudX, noeudY), dont noeudX est le noeud contenant x dans son étiquette et noeudY est le noeud contenant y dans son étiquette, classés par ordre de [t(y) - t(x)] croissant

2. pour chaque doublet dans la liste L, vérifier s' il existe un chemin de noeudX à noeudY en suivant le sens des arcs de G'.
3. l'algorithme s'arrête au premier chemin de x à y trouvé

- Sortie : si il existe un chemin de noeudX à noeudY, alors ceci est le chemin le plus rapide de x à y ; sinon, il n'existe pas de chemin de x à y.

Modélisation : pour cet algorithme nous avons décidé d'utiliser BFS, même si nous n'avons pas réussi à minimiser les instances nécessaires comme dans les algorithmes I et II. Nous avons préféré l'utilisation de cet algorithme car il présente une complexité inférieure par rapport à celle de l'algorithme de Dijkstra et nous n'avions pas de contraintes sur le poids des arcs, mais sur les dates d'arrivée et de départ des sommets x et y extrémités.

Proposition : Nous avons réfléchi à une optimisation que nous n'avons pas implémenté : il serait peut-être possible de sélectionner un sous-ensemble de la liste de permutations de doublets initiale et, en suivant l'ordre de [t(y) - t(x)] croissant, lorsqu'on considère le premier sommet étiqueté par x et on calcule le BFS sur cette racine, calculer en même temps toutes les autres paires avec même x mais y différent. De cette façon, nous pourrions au moins éviter de recalculer l'arbre couvrant quand cela n'est pas nécessaire, en stockant l'information dans un tableau de booléens. Une autre possibilité à laquelle nous avons réfléchi serait d'autoriser la visite de certains nœuds plusieurs fois si nous les rencontrons à nouveau avec une durée de temps plus courte, en mettant à jour le temps minimal nécessaire pour l'atteindre dans un tableau, mais nous n'avons pas eu le temps d'explorer cette option plus loin pour s'assurer qu'elle retournerait un chemin correcte. Nous avions essayé avec Dijkstra mais notre modélisation n'était pas la bonne pour éviter le calcul de l'arbre plusieurs fois.

Algorithme IV : plus court chemin

- Entrée : un graphe G'=(V,E) traduisant un multigraphe pondéré par le temps, une source x, une destination y et un intervalle de temps [t1, t2]
- Déroulement :
  1. déterminer noeudX = le noeud contenant x dans son étiquette avec t minimal >= t1.
  2. calculer un chemin de coût minimal dans G' qui part du noeudX et termine dans un nœud contenant y dans son étiquette. Pour cela, mémoriser la somme des poids des arcs pour chaque chemin trouvé, et retourner le chemin ayant la plus petite valeur de somme.
- Sortie : si il existe un chemin de x à y, alors ceci est un plus court chemin de x à y ; sinon, il n'existe pas de chemin de x à y.

Modélisation : pour cet algorithme - le seul parmi les 4 proposés qui tient compte du poids des arcs pour calculer le plus court chemin - nous avons utilisé l'algorithme de Dijkstra. L'algorithme nous trouve le chemin de coût minimal entre un des sommets étiquetés par x et l'un des sommets étiquetés par y. Pour cela, l'algorithme utilise une file de priorité. Dans notre implémentation nous nous sommes servis du module heapq qui existe en Python.

Question 3. Calculer la complexité de différents algorithmes proposés.

Nous avons choisi de modéliser le graphe G'=(V,E) traduisant un multigraphe pondéré par le temps sous forme de liste d'adjacence.

- Création de liste d'adjacence : O(nb sommets multiG * nb sommets G simple)
- Tri de liste d'adjacence : O (n log m)

Complexité algorithme I : chemin d'arrivée au plus tôt

- Instanciation liste L : O(1) , référence à la liste d'adjacence
- Recherche la racine du BFS avec t minimal (noeudX) : O(n)
- Construction arbre couvrant avec BFS : O(|V| + |E|)
- Recherche du premier chemin de x à y traceback : O(hauteur de l'arbre couvrant)
- **Complexité totale :** O(|V| + |E|)

Complexité algorithme II : chemin de départ au plus tard

- Recherche la racine du Dijkstra avec t minimal (noeudX) : O(n)
- Instanciation liste L : O(1) , référence à la liste d'adjacence
- Inversion liste L : O(nb de copies de y)
- Recherche la racine du BFS avec t maximal (noeudX) : O(n)
- Construction arbre couvrant avec BFS : O(|V| + |E|)
- Recherche du premier chemin de x à y traceback : O(hauteur de l'arbre couvrant)
- **Complexité totale :** O(|V| + |E|) + hauteur de l'arbre couvrant

Complexité algorithme III : chemin le plus rapide

- Instanciation liste L : O(1) , référence à la liste d'adjacence
- Combinaison des éléments de l'ensemble des noeudX avec les éléments de l'ensemble des noeudY : card(nb copies de x) * card(nb copies de y)
- Tri de L par ordre de [t(y) - t(x)] croissant : O (n log m)
- Recherche du premier chemin de x à y avec BFS : O(|V| + |E|) * len(combinaisons(x,y))
- **Complexité totale : O(|V| + |E|) * len(combinaisons(x,y)) + card(nb copies de x) * card(nb copies de y)**

Complexité algorithme IV : plus court chemin

- Recherche la racine du Dijkstra avec t minimal (noeudX) : O(n)
- Recherche du chemin de coût minimal de noeudX à y avec Dijkstra : O(|V|+|E| log |V|)
- **Complexité totale : O((|V|+|E| log |V|) * n)**

Question 4.

Voir les fichiers Python.

Question 5. Proposer une modélisation du problème de plus court chemin par programmation linéaire et implanter une méthode de recherche de chemin de Type IV en faisant appel à GUROBI.

La méthode python exécutant le programme linéaire du plus court chemin se trouve dans la classe MinimalDistanceProblem dans le fichier *MinimalDistanceProblem.py*.

Modélisation du PL du plus court chemin :

- Variables de décision : << si_sj >> représente un arc du graphe orienté reliant deux sommets s_i et s_j. si_sj vaut 1 si l'arc fait partie du plus court chemin trouvé ; 0 sinon. (PL binaire)

- Fonction objectif z : soit un sommet de départ x et un sommet d'arrivée y. On souhaite calculer le plus court chemin entre x et y, qui correspond au meilleur chemin de coût minimum. Pour cela on veut minimiser la distance de x à y, étant la distance la somme des coûts des arcs empruntés.

$$z = \min \sum_{i,j \epsilon \{S\}} \lambda_{(sisj)} \cdot s_i s_j$$

  S : l'ensemble des sommets de G.

  si_sj : arc orienté de si à sj (variable du PL)

  s_i : sommet ϵ S, si est un sommet qui a au moins un successeur

  s_j : sommet ϵ S, qui est successeur de si

  $\lambda(i, j)$: coût du chemin entre deux sommets s_i et s_j, autrement dit le poids de l'arc entre s_i et s_j (coefficient de la variable du PL). Dans le contexte de la transformation du multigraphe en graphe, le poids d'un arc peut prendre soit la valeur 0, soit la valeur $\lambda(i, j) > 0$.

  Si on recherche le plus court chemin entre x et y, nous avons intérêt à choisir de préférence les arcs qui ont un coût de 0, car ceux-ci réduisent la distance de P, c'est-à-dire le coût global du chemin. En multipliant les coefficients $\lambda(i, j)$ par les variables respectives dans le PL, nous annulons les arcs préférablement empruntables de coût 0, ce qui a pour effet de minimiser la fonction objectif et ainsi donner la solution optimale.

- Contraintes : les contraintes traduisent les règles de parcours des graphes orientés :
  1) Modélisation de la contrainte d'un seul arc empruntable parmi plusieurs choix possibles : pour chaque nœud du graphe, le nombre d'arcs entrants est égal au nombre d'arcs sortants.

Exemple :

c : si_sj = sj_sk + sj_st traduit que pour le nœud j, s_k et s_t sont les successeurs de s_j, et s_j est successeur de s_i.

Cette modélisation signifie : "si l'arc si_sj n'a pas été emprunté - donc si_sj = 0 - alors les arcs sj_sk et sj_st ne sont pas empruntables non plus. Dans l'autre sens, si l'arc si_sj a été emprunté - donc si_sj = 1 - alors l'un entre les arcs sj_sk ou sj_st vaudra 1, mais pas les deux au même temps". Cela implique que nous pouvons enchaîner les sommets visités en choisissant à chaque fois un seul successeur parmi les plusieurs successeurs disponibles.

2) Modélisation des chemins obligés, notamment:

   a) Modélisation du départ à partir du sommet x : un et un seul parmi les arcs sortant de x doit être emprunté pour avoir un chemin valide.
      c : sxa + … + sxb = 1
   b) Modélisation de l'arrivée au sommet y : un et un seul parmi les arcs entrant en y doit être emprunté pour avoir un chemin valide.
      c : suy + … + swy = -1

```
Optimize a model with 13 rows, 15 columns and 30 nonzeros
Model fingerprint: 0x1a854c0f
Variable types: 0 continuous, 15 integer (15 binary)
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 1e+00]
Found heuristic solution: objective 2.0000000
Presolve removed 13 rows and 15 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.00 seconds (0.00 work units)
Thread count was 1 (of 8 available processors)

Solution count 1: 2

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

Solution optimale:
(('a', 1), ('b', 2), 1) = 0.0
(('a', 2), ('b', 3), 1) = 0.0
(('a', 2), ('c', 3), 1) = 0.0
(('a', 4), ('c', 5), 1) = 1.0
(('b', 5), ('f', 6), 1) = 0.0
(('c', 6), ('f', 7), 1) = 0.0
(('c', 7), ('g', 8), 1) = 1.0
(('a', 1), ('a', 2), 0) = 1.0
(('a', 2), ('a', 4), 0) = 1.0
(('b', 2), ('b', 3), 0) = 0.0
(('b', 3), ('b', 5), 0) = 0.0
(('c', 3), ('c', 5), 0) = 0.0
(('c', 5), ('c', 6), 0) = 1.0
(('c', 6), ('c', 7), 0) = 1.0
(('f', 6), ('f', 7), 0) = 0.0

Valeur de la fonction objectif : 2.0
Chemin de type IV (Chemin le plus court) calculé avec PL : [(('a', 1), ('a', 2), 0), (('a', 2), ('a',
 4), 0), (('a', 4), ('c', 5), 1), (('c', 5), ('c', 6), 0), (('c', 6), ('c', 7), 0), (('c', 7), ('g',
8), 1)]
```

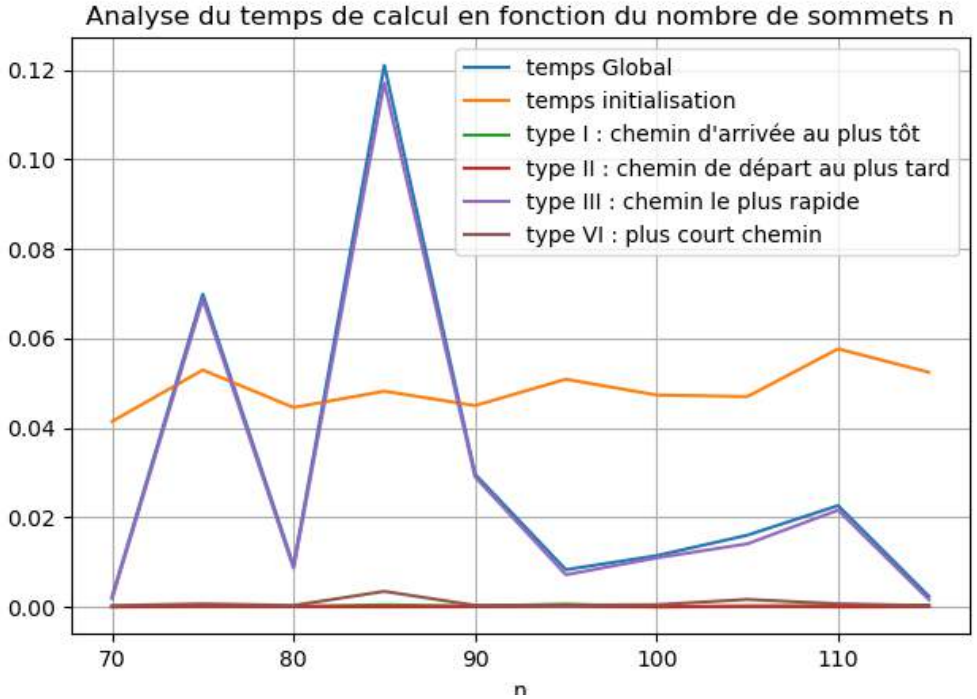
Question 6. Effectuer des tests pour mesurer le temps d'exécution de votre algorithme par rapport à la taille de l'entrée (nombre de sommets, nombre d'arcs, étiquettes sur les sommets).

Pour effectuer les tests, nous avons créé des fonctions de génération aléatoire de graphes. Nous avons deux fonctions différentes, et nous pouvons voir ci-dessous que le temps pris pour les graphes générés d'une manière sont plus lentes. La complexité reste néanmoins la même. Nous pouvons aussi noter que la phase d'initialisation de nos problèmes est très coûteuse.

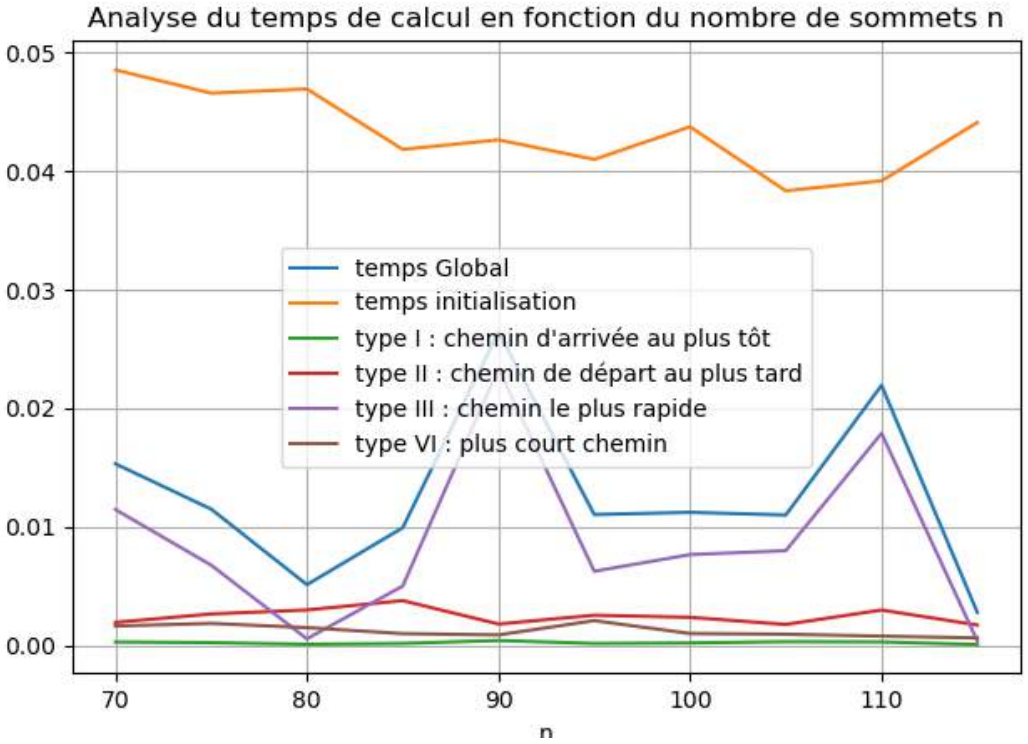
Paramètres utilisés pour l'affichage des courbes :

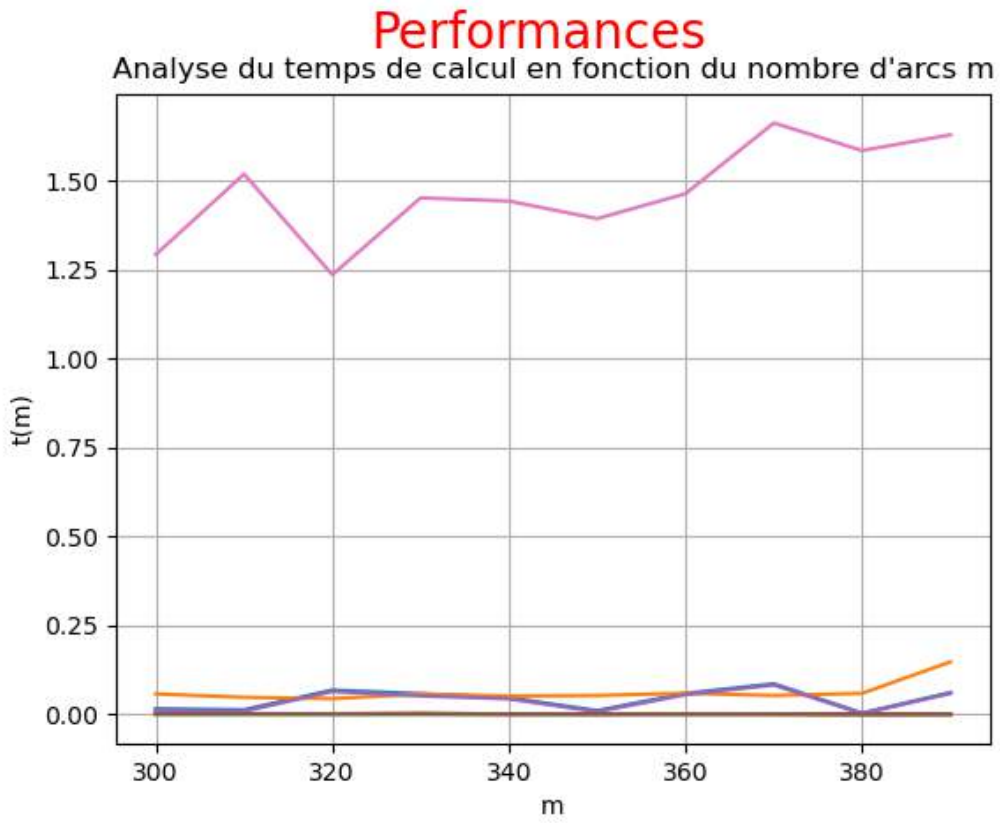
```
nbTests = 10
nbIterations = 5
# Parametres pour plotPerformances_n
minN = 50
maxN = 70

# Parametres pour plotPerformances_m
minM = 100
maxM = 300

# Parametres pour plotPerformances_d
maxInterval_dates = [1, 50]

n_fixe = 50 # Valeur de n  fixé pour m ou interval qui évoluent
m_fixe = 210 # Valeur de m fixé pour n ou interval qui évoluent
interval_dates_fixe = [1, 100] # Valeur de interval fixé pour n ou m qui évoluent
```

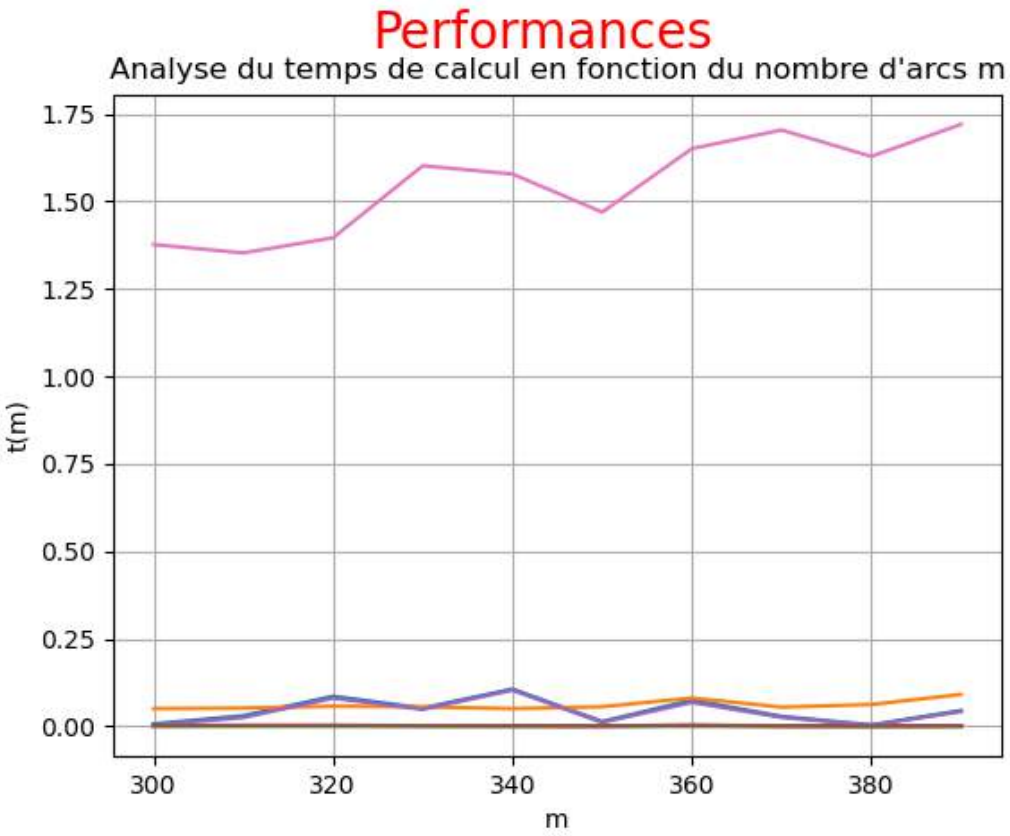
Performances



Performances

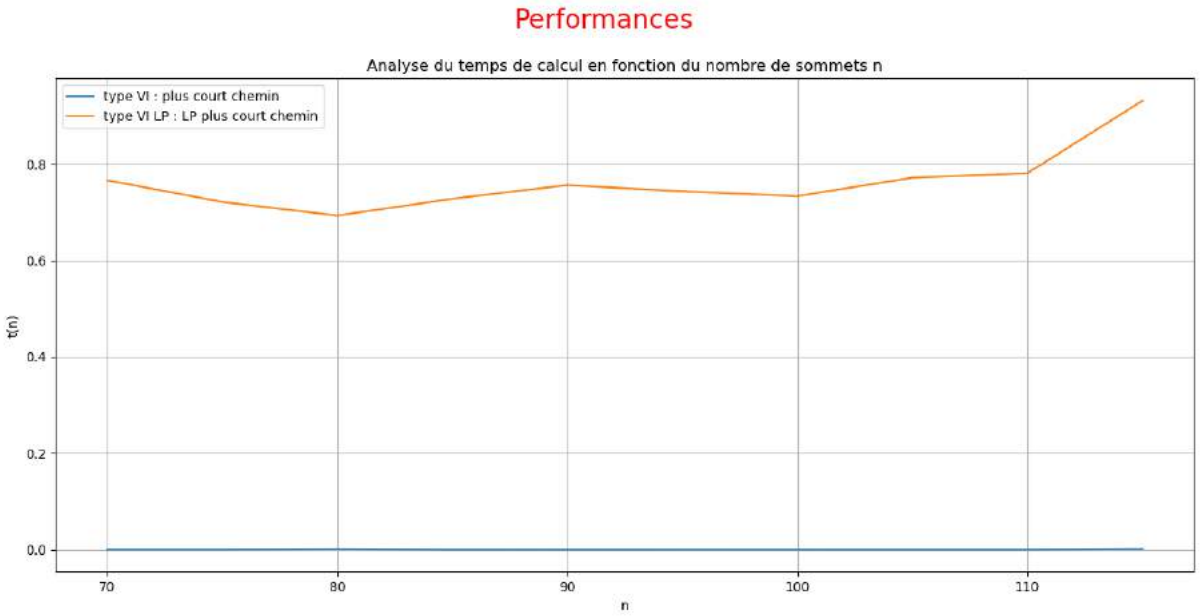


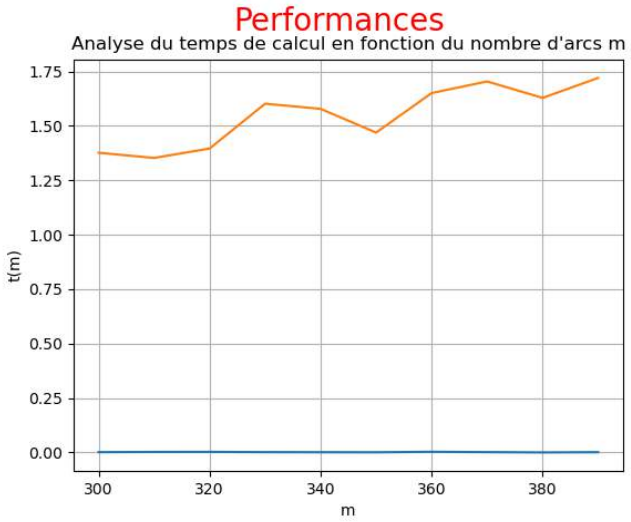


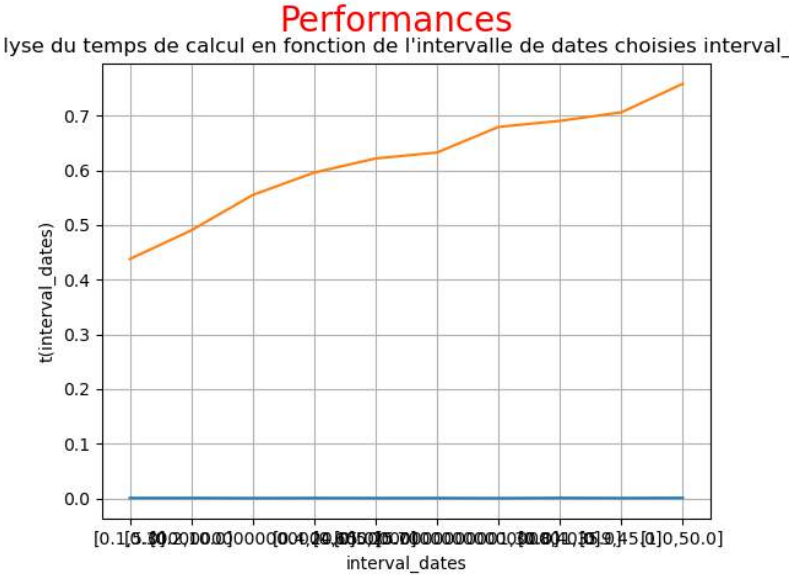


Question 7. Comparer les algorithmes implantés pour le calcul de chemins de type IV dans les questions 4 et 5.

Ces courbes nous permettent de voir que la modélisation par programmation linéaire est beaucoup plus lente que l'algorithme de Dijkstra, et que sa complexité est polynomiale.







Annexe

Description des modèles :

Nous avons créé une classe Multigraph qui contient les attributs n, m, vertices (liste des nœuds), edges (liste des arcs sous forme de tuples). Cette classe contient une fonction transform_to_graph() qui transforme un multigraphe en graphe simple.

Ces graphes simples sont des objets Graph qui contiennent les attributs n, m, vertices (dictionnaire qui map les labels des nœuds originaux à la liste de leurs copies), edges (liste des arcs sous formes de tuples (source, dest, poids). Ils contiennent aussi une liste d'adjacence à l'endroit, ainsi qu'une liste d'adjacence à l'envers (avec le sens des arcs inversés). Ces listes sont créées lors de l'instanciation d'un Graph par une fonction obtain_adjacency_list(). La classe contient également une fonction BFS() qui calcule un chemin avec l'algorithme du parcours en largeur et une fonction Dijkstra() qui calcule un chemin avec l'algorithme du même nom.

Ensuite nous avons créé une classe MinimalDistanceProblem qui représente un problème donné : un sommet de départ x, un sommet d'arrivée y, et un intervalle dans lequel le chemin doit être effectué.