Sorbonne Université
M1 Androide / IQ

MOGPL

Projet - Automne 2021

Hugo Abreu
Krisni Almehdi

soutenu le 17 Décembre 2021

Contents

1 Introduction **1**

2 Préliminaires **1**

3 Algorithmes de plus court chemin dans des multigraphes orientés pondérés par le temps **3**

3.1 Structure de données pour la représentation de multigraphes 3
3.2 Transformation multigraphe → graphe 3
3.3 Problème du chemin d'arrivée au plus tôt 4
3.4 Problème du chemin de départ au plus tard 6
3.5 Problème du chemin le plus rapide 6
3.6 Problème du plus court chemin 6
3.7 Problème du plus court chemin en Programmation Linéaire 6
3.8 Algorithmes sans transformation de graphe 8
3.9 Tests de performance entre les différents algorithmes 10

1 Introduction

Dans ce projet, on considère des multigraphes orientés pondérés par le temps. Ce type de graphe est utile notamment pour la plannification temporelle - nous considèrerons le cas où un réseau de transport aérien est représenté par un tel graphe.

2 Préliminaires

Question 1. En utilisant l'instance de la figure de gauche de l'Exemple 1 (dans l'énoncé) ou une autre instance, montrer que les assertions suivantes sont vraies.

Par soucis de simplicité et pour être plus concis, une instance alternative est proposée. Considérons $G_1 = (V_1, E_1)$, le multigraphe orienté pondéré par le temps donné par le diagramme de la Figure 1.

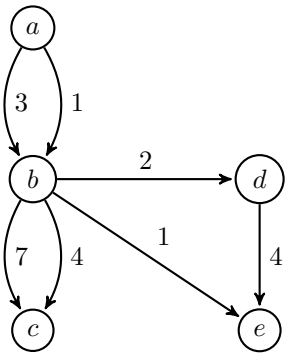

Figure 1: Diagramme représentant le multigraphe orienté pondéré par le temps G_1

Comme dans l'énoncé, G_1 représente un réseau de transport aérien où chaque arrête est un vol. On suppose que la durée de chaque vol est de 1 jour (i.e. $\forall e_i \in E_1, e_i = (u_i, v_i, t_i, 1)$).

Assertion 1.1. *Un sous-chemin préfixe d'un chemin d'arrivée au plus tôt peut ne pas être un chemin d'arrivée au plus tôt.*

Réponse. Considérons l'ensemble de chemins $\mathcal{P}(a, c, [0, \infty])$ dans le multigraphe G_1, qui correspond à l'ensemble de tous les chemins réalisables de a à c dans G_1:

$$\begin{aligned}\mathcal{P}(a, c, [0, \infty]) = \{&P_1 = ((a, b, 1, 1), (b, c, 4, 1)),\\ &P_2 = ((a, b, 1, 1), (b, c, 7, 1)),\\ &P_3 = ((a, b, 3, 1), (b, c, 4, 1)),\\ &P_4 = ((a, b, 3, 1), (b, c, 7, 1))\}.\end{aligned} \tag{1}$$

Pour déterminer le(s) chemin(s) d'arrivée au plus tôt de a à c dans le graphe G_1, soit un chemin P tel que $\text{fin}(P) = \min(\{\text{fin}(P') : P' \in \mathcal{P}(a, c, [0, \infty]\})$, calculons les dates de fin pour tout P appartenant à $\mathcal{P}(a, c, [0, \infty])$:

$$\begin{aligned}\text{fin}(P_1) &= 4 + 1 = 5,\\ \text{fin}(P_2) &= 7 + 1 = 8,\\ \text{fin}(P_3) &= 4 + 1 = 5,\\ \text{fin}(P_4) &= 7 + 1 = 8.\end{aligned} \tag{2}$$

Ainsi, $\min(\{5, 8, 5, 8\}) = 5$ et les chemins d'arrivée au plus tôt sont P_1 et P_3.

$P'_3 = ((a, b, 3, 1))$, un chemin de a vers b, est un sous-chemin préfixe (un sous-chemin partant du sommet de départ) de P_3. Cependant, il existe un chemin $P_{a\to b} = ((a, b, 1, 1))$ tel que

$$\text{fin}(P'_3) = 3 + 1 = 4 > \text{fin}(P_{a\to b}) = 1 + 1 = 2, \tag{3}$$

donc P'_3 n'est pas un chemin d'arrivée au plus tôt de a à b.

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

Assertion 1.2. *Un sous-chemin postfixe d'un chemin de départ au plus tard peut ne pas être un chemin de départ au plus tard.*

Réponse. Considérons de nouveau l'ensemble de chemins $\mathcal{P}(a, c, [0, \infty])$, donné en Équation 1.

Un chemin de départ au plus tard de a à c, dans le multigraphe G_1, correspond à un chemin P tel que $\text{début}(P) = \max(\{\text{début}(P') : P' \in \mathcal{P}(a, c, [0, \infty]\})$. On observe facilement que

$$\text{début}(P_1) = \text{début}(P_2) = 1 < \text{début}(P_3) = \text{début}(P_4) = 3, \tag{4}$$

donc les chemins de départ au plus tard entre a et c sont P_3 et P_4.

$P''_3 = ((b, c, 4, 1))$, un chemin de b vers c, est un sous-chemin postfixe (un sous-chemin partant du sommet final) de P_3. Cependant, il existe un chemin $P_{b\to c} = ((b, c, 7, 1))$ tel que

$$\text{début}(P''_3) = 3 < \text{début}(P_{b\to c}) = 7 \tag{5}$$

donc P''_3 n'est pas un chemin de départ au plus tard de b à c.

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

Assertion 1.3. *Un sous-chemin d'un chemin le plus rapide peut ne pas être un chemin le plus rapide.*

Réponse. Considérons les chemins de a à d dans G_1:

$$\mathcal{P}(a, d, [0, \infty]) = \{P_5 = ((a, b, 1, 1), (b, d, 2, 1), (d, e, 4, 1))\}. \tag{6}$$

Comme il n'existe qu'un seul chemin réalisable, c'est forcément un chemin le plus rapide de a à d, un chemin P tel que $\text{durée}(P) = \min(\{\text{durée}(P') : P' \in \mathcal{P}(a, d, [0, \infty])\})$.

$P'_5 = ((b, d, 2, 1), (d, e, 4, 1))$, un chemin de b à e, est un sous-chemin de P_5. Cependant, il existe un chemin $P_{b\to e} = ((b, e, 1, 1))$ de b vers e tel que

$$\text{durée}(P'_5) = (4 + 1) - 1 = 4 > \text{durée}(P_{b\to e}) = (1 + 1) - 1 = 1, \tag{7}$$

donc P'_5 n'est pas un chemin le plus rapide de b à e.

Ainsi, un sous-chemin d'un chemin le plus rapide peut ne pas être un chemin le plus rapide. □

Assertion 1.4. *Un sous-chemin d'un plus court chemin peut ne pas être un plus court chemin.*

Réponse. Considérons de nouveau les chemins $\mathcal{P}(a, d, [0, \infty])$ de a vers d donnés en Équation 6.

De même, comme il n'existe qu'un seul chemin réalisable, P_5 est forcément le chemin le plus court: un chemin P tel que $\text{dist}(P) = \min(\{\text{dist}(P') : P' \in \mathcal{P}(a, d, [0, \infty])\})$.

Considérons de nouveau les chemins P''_5 et $P_{b\to e}$ donnés en Assertion 1.3.

$$\text{dist}(P'_5) = 1 + 1 = 2 > \text{dist}(P_{b\to e}) = 1, \tag{8}$$

donc P'_5 n'est pas un chemin le plus court de b à e.

Ainsi, un sous-chemin d'un chemin le plus court peut ne pas être un chemin le plus court.

□

3 Algorithmes de plus court chemin dans des multigraphes orientés pondérés par le temps

Dans cette partie, les réponses aux questions 2 et 3 et du sujet sont données au fûr et à mesure de la présentation des algorithmes.

Les algorithmes développés pour la question 4 sont mis en annexe à ce rapport.

Question 2. En utilisant comme base la transformation de G à $\tilde{G}$, montrer comment calculer de manière efficace les 4 types de chemins minimaux.

Question 3. Calculer la complexité de différents algorithmes proposés.

Question 4. Ecrire un programme qui prend en entrée un multigraphe G sans circuit et retourne les 4 types de chemins minimaux correspondants.

Pour obtenir les 4 types de chemins minimaux d'un multigraphe G dans un fichier txt, on peut faire tourner le programme main. La commande suivante calcule ces 4 chemins, entre a et c, pour le graphe dans exempleGraphe.txt:

```
$ python3 main.py "Repertoire_Graphes/exempleGraphe.txt" a c
```

3.1 Structure de données pour la représentation de multigraphes

Pour représenter des multigraphes (et des graphes classiques, également), on choisit de travailler avec des dictionnaires représentant des listes d'adjacence (dans ce cas, des listes de succésseurs). Ce choix est efficace vu que l'accès aux dictionnaires en python est d'une complexité de $\mathcal{O}(1)$, via l'utilisation fonctions de hashage gérées automatiquement par python.

Pour un multigraphe G, les clés du dictionnaire correspondent aux sommets de G, et ses valeurs à la liste des arcs sortants pour chaque sommet. Comme un arc (s_1, s_2, dDD, cDT) - où s_1 correspond au sommet de départ, s_2 au sommet de destination, dDD la date de départ du vol, et cDT la durée du vol - est forcément associé à un sommet de départ dans le dictionnaire (la clé), on ne garde que (s_2, dDD, cDT).

3.2 Transformation multigraphe $\rightarrow$ graphe

Pour adapter des algorithmes standards à des multigraphes orientés pondérés par le temps (MOPLT), une technique est proposée dans le sujet pour tranformer un MOPLT en un graphe classique qui prend en compte le temps de manière explicite.

On propose une approche légèrement différente à celle proposée dans le sujet. On calcule $\tilde{V}_{in}$ et $\tilde{V}_{out}$, et on crée une liste de d'adjacence. Cependant, on ne donne pas de poids aux arcs: avec une comparaison en $\mathcal{O}(1)$ on peut déterminer si deux noeuds sont les mêmes (et donc un poids de 0) où différents (un poids de 1).

Pour le faire, on a implémenté une fonction `transformeGraphe` qui prend en argument un multigraphe G (représenté sous-forme d'un dictionnaire) et retourne le graphe $\tilde{G}$ (lui aussi sous forme de dictionnaire). Cette fois, les clés du dictionnaire représentant $\tilde{G}$ correspondent au tuple (s_1, t_1) où $s_1 \in G$ et t_1 correspond à la valeur de dDD dans un des arcs sortant de s_1. Les valeurs du dictionnaire de $\tilde{G}$ correspondent à deux listes de sommets (s_2, t_2) et (s_3, t_3) où (s_2, t_2) correspond à un vol de s_1 à s_2 le jour t_2, et (s_3, t_3) correspond à un vol de s_3 à s_1 le jour t_3.

Pour que les algorithmes proposés dans les sections suivantes soient plus efficaces, on choisit également d'ordonner les dates des sommets sortants par ordre croissante.

Le pseudo-code pour cette fonction est donné en Algorithme 1.

Pour calculer la complexité, on assume que $|V| < |E|$. Ainsi, la complexité temporelle de l'algorithme 1 est la suivante:

$$\begin{aligned}\mathcal{O}(\text{Algorithme 1}) &= \mathcal{O}((|V|+|E|)+(|V|+(|V|+|E|)\log(|V|+|E|))) \\ &= \mathcal{O}(2(|V|+|E|)+(|V|+|E|)\log(|V|+|E|)) \\ &= \mathcal{O}((|V|+|E|)\log(|V|+|E|)),\end{aligned} \tag{9}$$

et la complexité spatiale (la taille de $\tilde{G}$):

$$\mathcal{O}(\text{Algorithme 1}) = \mathcal{O}(|V|+|E|) \tag{10}$$

Algorithm 1: tranformeGraphe

Data: G multigraphe orienté pondéré par le temps, sous forme de dictionnaire
Result: $\tilde{G}$ graphe orienté, sous forme de dictionnaire
$\tilde{G} \leftarrow$ dictionnaire vide ;
foreach s_i *in sommets(G)* **do**
  foreach (s_j, dDD, cDT) *in arcs(*$G[s_i]$*)* **do**
    sommet_courant $\leftarrow (s_i, dDD)$;
    sommet_suivant $\leftarrow (s_j, dDD + cDT)$;
    $\tilde{G}$[sommet_courant][1] $\leftarrow \tilde{G}$[sommet_courant][1] + (sommet_suivant);
    $\tilde{G}$[sommet_suivant][0] $\leftarrow \tilde{G}$[sommet_suivant][0] + (sommet_courant);
  end
end
foreach s_i *in sommets(G)* **do**
  foreach (s_j, dDD_j) *in sommets(*$\tilde{G}$*)* **do**
    sommets_a_ajouter $\leftarrow$ liste vide ;
    if $s_i = s_j$ **then**
      sommets_a_ajouter $\leftarrow$ sommets_a_ajouter + (s_j, dDD_j);
    end
    sommets_a_ajouter $\leftarrow$ triage_croissant(sommets_a_ajouter);
    idx_sommet $\leftarrow 1$;
    while $i <$ *len(sommets_a_ajouter)* **do**
      sommet_courant $\leftarrow$ sommets_a_ajouter$[i-1]$;
      sommet_suivant $\leftarrow$ sommets_a_ajouter$[i]$;
      $\tilde{G}$[sommet_courant][1] $\leftarrow \tilde{G}$[sommet_courant][1] + sommet_suivant;
      $\tilde{G}$[sommet_suivant][0] $\leftarrow \tilde{G}$[sommet_suivant][0] + sommet_courant;
      idx_sommet $\leftarrow$ idx_sommet + 1;
    end
  end
end

3.3 Problème du chemin d'arrivée au plus tôt

Pour le problème du chemin d'arrivée au plus tôt, comme pour les deux problèmes suivants, on utilise une variante du BFS (Breadth-First Search) - c'est un algorithme efficace lorsqu'il s'agit de trouver un seul chemin (optimal) d'un point à un autre dans un graphe. Ce sera la méthode utilisée pour les 3 premiers problèmes.

Vu que ce n'étais pas précisé, on a décidé de garder uniquement les chemins d'un point du graphe à un autre. Mais notre algorithme parcourt forcément tous les points, donc la complexité pour récupérer les chemins les plus court d'un point à tous les autres serait la même.

L'idée de cet algorithme est de partir du sommet de départ dans $\tilde{G}$ et de faire tourner BFS.

Pour calculer le chemin d'arrivée au plus tôt, on crée une pile contenant tous les successeurs du sommet de départ - et on ajoute progressivement les fils de ces sommets en vérifiant si on est au sommet de destination et en gardant le meilleur temps (indiqué directement dans le sommet de $\tilde{G}$). En plus de garder les successeurs, on considère aussi ce qu'on appelle un "state": c'est un tuple contenant le sommet courant, le jour actuel, et le père (le noeud précédent) de l'état. Pour récupérer le chemin optimal, on n'a qu'à dépiler.

Le pseudo-code pour cet algorithme est donné en Algorithme 2.

Algorithm 2: cheminArriveeAuPlusTot

Data: $\tilde{G}$ graphe orienté, sous forme de dictionnaire. sommetDepart, tuple. sommetArrivee, tuple.
Result: *res*, liste de sommets de $\tilde{G}$ formant un chemin d'arrivée au plus tôt de sommetDepart à sommetArrivee.

```
pile ← liste vide;
res ← liste vide;
state ← (sommetDepart, 1, None);
if (sommetDepart, 1) in sommets(G̃) then
    foreach (s_i, dDD_i) in arcs(G̃[(sommetDepart, 1)]) do
        pile ← pile + (s_i, dDD_i, state);
    end
else
    foreach (s_i, dDD_i) in sommets(G̃) if s_i = sommetDepart do
        pile ← pile + (s_i, dDD_i, state);
    end
end
bestChemin ← emtpy tuple;
bestTime ← None;
while pile ≠ ∅ do
    currentState ← pile[0];
    pile ← pile[1 :];
    if currentState = sommetArrivee and (bestTime > currentState[1] or bestTime = None)
     then
        bestTime ← currentState[1];
        bestChemin ← currentState;
    else
        foreach (s_i, dDD_i) in sommets(G̃[(currentState[0], currentState[1])]) do
            pile ← (s_i, dDD_i, currentState);
        end
    end
end
while bestChemin ≠ None do
    res ← (bestChemin[0], bestChemin[1]);
    bestChemin ← bestChemin[2];
end
inverse(res);
```

Vu que l'on utilise la représentation $G = (\tilde{V}_1, E_1)$ du multigraphe $G = (V_2, E_2)$, juste un seul BFS est nécessaire pour trouver le chemin optimal. Supposons que $|V_2| < |E_2|$: ainsi, tant $|V_1|$ comme $|E_1|$ sont bornés par $\mathcal{O}(|E_2|)$. La complexité cet algorithme est donc:

$$\begin{aligned}\mathcal{O}(\text{Algorithme 2}) &= \mathcal{O}(|V_1| + |E_1|) \\ &= \mathcal{O}(2|E_2|) \\ &= \mathcal{O}(|E_2|).\end{aligned} \tag{11}$$

L'algorithme proposé pour le problème de chemin d'arrivée au plus tôt est donc en temps linéaire après la transformation du graphe.

3.4 Problème du chemin de départ au plus tard

Pour le problème du chemin de départ au plus tard, on procède d'une manière similaire à celui de l'arrivée au plus tôt. L'idée est de partir du sommet de départ dans $\tilde{G}$ et de faire tourner BFS à l'envers. Le temps maximal trouvé correspond au chemin de départ au plus tard.

Le pseudo-code pour cet algorithme est donné en Algorithme 3.

Cet algorithme, comme le premier, est également en $\mathcal{O}(|E_2|)$.

3.5 Problème du chemin le plus rapide

Pour le problème du chemin le plus rapide, on procède encore une fois de manière similaire aux deux derniers algorithmes. Cette fois, on considère la différence entre le temps du sommet d'arrivée et le temps du sommet de départ. Une différence est que l'on utilise une fonction `fatherState()` qui récupère le state précédent dans le chemin actuel.

Le pseudo-code pour cet algorithme est donné en Algorithme 4.

Cet algorithme a une complexité (au pire des cas) de $\mathcal{O}(|E_2| \log(|E_2|))$, ce qui pourrait être amélioré sans la fonction `fatherState`. Comme cette fonction n'est que appellée lorsqu'un chemin est réalisable est trouvé, dans la plupart des cas la complexité se rapproche plus de $\mathcal{O}(|E_2|)$.

3.6 Problème du plus court chemin

Pour le problème du plus court chemin, c'est plus compliqué. On aurait voulu implémanter une méthode de type Dijkstra, pour laquelle on aurait obtenu une complexité de $\mathcal{O}(|E_2| \log(|E_2|))$, mais on a pas eu le temps pour adapter notre structure de données pour fonctionner de cette facon.

Ainsi, notre algorithme possède une compléxité de $\mathcal{O}(|E_2|^2)$, ce qui provient de l'utilisation de deux boucles while imbriquées.

Le pseudo-code pour cet algorithme est donné en Algorithme 5

3.7 Problème du plus court chemin en Programmation Linéaire

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

Pour résoudre le problème de plus court chemin par programmation linéaire, on doit d'abord choisir une représentation pour le graphe qui soit naturelle à utiliser dans un programme linéaire.

On commence avec le graphe $\tilde{G}$. On représente le graphe par une matrice d'adjacence C avec les poids des arcs entre les différents sommets.

Les variables de décision X correspondent à une autre matrice, binaire, qui indique si on considère où non l'arc à la position (i,j) dans le chemin allant du sommet de départ au sommet de destination.

On introduit les contraintes suivantes:

1. Il n'y a qu'une seule et unique arrête sortante depuis le sommet de départ.

2. Il n'y a qu'une seule et unique arrête entrante sur le sommet d'arrivée.

Algorithm 3: cheminDepartAuPlusTard

Data: $\tilde{G}$ graphe orienté, sous forme de dictionnaire. sommetDepart, tuple. sommetArrivee, tuple.
Result: *res*, liste de sommets de $\tilde{G}$ formant un chemin de départ au plus tard de sommetDepart à sommetArrivee.

```
pile ← liste vide;
res ← liste vide;
state ← (sommetDepart, 1, None);
if (sommetDepart, 1) in sommets(G̃) then
    foreach (s_i, dDD_i) in arcs(G̃[(sommetDepart, 1)]) do
        pile ← pile + (s_i, dDD_i, state);
    end
else
    foreach (s_i, dDD_i) in sommets(G̃) if s_i = sommetDepart do
        pile ← pile + (s_i, dDD_i, state);
    end
end
bestChemin ← emtpy tuple;
lastDeparture ← None;
while pile ≠ ∅ do
    currentState ← pile[0];
    pile ← pile[1 :];
    if currentState = sommetArrivee then
        while currentState[0] ≠ sommetDepart do
            currentState ← currentState[2];
        end
        startTime ← currentState[1];
        if lastDeparture < startTime[1] or lastDeparture = None then
            lastDeparture ← startTime;
            bestChemin ← currentState;
        end
    else
        foreach (s_i, dDD_i) in sommets(G̃[(currentState[0], currentState[1])]) do
            pile ← (s_i, dDD_i, currentState);
        end
    end
end
while bestChemin ≠ None do
    res ← (bestChemin[0], bestChemin[1]);
    bestChemin ← bestChemin[2];
end
inverse(res);
```

3. Pour chaque sommet (sauf celui de départ et d'arrivée), s'il existe un arc entrant vers le sommet il faut forcément un arc sortant.

La fonction objectif correspond à minimiser: $\sum\sum c_{i,j} \times x_{i,j}$, pour $c_{i,j} \in C$ et $x_{i,j} \in X$.

La méthode `optPlusCourtChemin()`, qui prend en argument un graphe modifié par la méthode `transformeGrapheOptimisation()`, calcule un plus court chemin en résolvant un programme linéaire.

Algorithm 4: CheminPlusRapide

Data: $\tilde{G}$ graphe orienté, sous forme de dictionnaire. sommetDepart, tuple. sommetArrivee, tuple.
Result: *res*, liste de sommets de $\tilde{G}$ formant un chemin le plus rapide de sommetDepart à sommetArrivee.

```
pile ← liste vide;
res ← liste vide;
state ← (sommetDepart, 1, None);
if (sommetDepart, 1) in sommets(G̃) then
    foreach (s_i, dDD_i) in arcs(G̃[(sommetDepart, 1)]) do
        pile ← pile + (s_i, dDD_i, state);
    end
else
    foreach (s_i, dDD_i) in sommets(G̃) if s_i = sommetDepart do
        pile ← pile + (s_i, dDD_i, state);
    end
end
bestChemin ← emtpy tuple;
shortestTime ← None;
while pile ≠ ∅ do
    currentState ← pile[0];
    pile ← pile[1 :];
    if currentState = sommetArrivee then
        timeSpent ← currentState[1] − fatherState(currentState)[1];
        if shortestTime > timeSpent or shortestTime = None then
            shortestTime ← timeSpent;
            bestChemin ← currentState;
        end
    else
        foreach (s_i, dDD_i) in sommets(G̃[(currentState[0], currentState[1])]) do
            pile ← (s_i, dDD_i, currentState);
        end
    end
end
while bestChemin ≠ None do
    res ← (bestChemin[0], bestChemin[1]);
    bestChemin ← bestChemin[2];
end
inverse(res);
```

3.8 Algorithmes sans transformation de graphe

Nous avons aussi développé des algorithmes qui n'utilisent pas la transformation de graphe. Ils correspondent aux versions V2 dans le fichier `algorithmesChemin.py`. Ils sont bien commentés donc le rapport ne les abordera pas en détail.

Algorithm 5: cheminPlusCcourt

Data: $\tilde{G}$ graphe orienté, sous forme de dictionnaire. sommetDepart, tuple. sommetArrivee, tuple.
Result: res, liste de sommets de $\tilde{G}$ formant un chemin le plus court de sommetDepart à sommetArrivee.

pile ← liste vide;
res ← liste vide;
state ← (sommetDepart, 1, $None$);
if ($sommetDepart, 1$) *in sommets($\tilde{G}$)* **then**
  foreach (s_i, dDD_i) *in arcs($\tilde{G}$[($sommetDepart, 1$)])* **do**
    pile ← pile + ($s_i, dDD_i, state$);
  end
else
  foreach (s_i, dDD_i) *in sommets($\tilde{G}$) if* $s_i = sommetDepart$ **do**
    pile ← pile + ($s_i, dDD_i, state$);
  end
end
bestChemin ← emtpy tuple;
fewestMoves ← None;
while $pile \neq \emptyset$ **do**
  currentState ← pile[0];
  pile ← pile[1 :];
  if $currentState = sommetArrivee$ **then**
    differentPlaces ← liste vide;
    tempState ← $currentState$;
    while $tempState \neq \emptyset$ **do**
      differentPlaces ← differentPlaces + tempState[0];
      tempState = tempState[2];
    end
    numberMoves = $len(differentPlaces)$;
    if $fewestMoves < numberMoves$ *or* $fewestMoves = None$ **then**
      bestTime ← currentState[1];
      bestChemin ← currentState;
    end
  else
    foreach (s_i, dDD_i) *in sommets($\tilde{G}$[($currentState$[0], $currentState$[1])])* **do**
      pile ← (s_i, dDD_i, currentState);
    end
  end
end
while $bestChemin \neq None$ **do**
  res ← (bestChemin[0], bestChemin[1]);
  bestChemin ← bestChemin[2];
end
inverse(res);

3.9 Tests de performance entre les différents algorithmes

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).

Réponse. Les tests sont dans le jupyter notebook `tests.ipynb`. □

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

Réponse. Pour le graphe `/Repertoire_Graphes/exempleGraphe.txt`, un graphe de 10 sommets et 15 arcs, les temps d'exécution pour trouver trois le chemin le plus court entre 3 paires de sommets choisis aléatoirement (pour lesquels il existe un chemin réalisable) sont donnés

```
Résultats de l'algorithme :
Chemin recherché                Temps d'exécution              Solution
Chemin de b à h                  0.000185               [('b', 1), ('b', 3), ('h', 4)]
Chemin de i à l                  0.000262               [('i', 1), ('i', 8), ('l', 9)]
Chemin de f à l                  0.000220               [('f', 1), ('f', 5), ('i', 6),
                                                         ('i', 8), ('l', 9)]


Résultats de l'optimisation :
Chemin recherché                Temps d'exécution              Solution
Chemin de b à h                  0.024072               [('Arc_b,h', 1.0)]
Chemin de i à l                  0.002960               [('Arc_i,l', 1.0)]
Chemin de f à l                  0.002942               [('Arc_f,i', 1.0),
                                                         ('Arc_i,l', 1.0)]
```

On remarque que l'algorithme de programmation linéaire est plus d'une ordre de magnitude plus lent. Cela n'est pas surprenant, vu que les programmes linéaires ne sont pas une bonne méthode pour résoudre des algorithmes dans les graphes (le nombre de contraintes croit exponentiellement). □