Algorithmique et Combinatoire des Structures Discrètes

2-approximation du routard

1 Description du problème

Nous considérons un graphe connexe non orienté $G = (V, E)$ composé de n sommets. Ce graphe est pondéré : à chaque arête $\{u, v\} \in E$ correspond un poids $w(u, v) \in \mathbb{R}$.

Étant donné une *séquence* $\sigma = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$ de sommets, où les sommets consécutifs sont voisins, on définit son *poids* par

$$w(\sigma) = \sum_{j=1}^{k-1} w(v_{i_j}, v_{i_{j+1}}).$$

Le problème du *routard* consiste à déterminer une séquence σ de sommets de G, contenant au moins une fois chacun des sommets de G, dont le poids $w(\sigma)$ est le plus petit possible et tel que $v_{i_1} = v_{i_k}$. Autrement dit, on cherche un *cycle* σ dans G de *poids minimum*. Ce cycle peut passer plusieurs fois par un même sommet ; la seule contrainte est de passer au moins une fois sur chaque sommet et de retourner au sommet de départ. Il est parfaitement autorisé de passer plusieurs fois par une même arête.

Vous pouvez deviner la motivation du problème du routard : cela peut-être un représentant de commerce qui veut passer dans chaque grande ville de France durant les 4 prochaines semaines, puis rentrer chez lui. Ou alors, les gardiens de l'université qui doivent faire leur ronde en passant par des points stratégiques puis retourner au poste de sécurité. Ou peut-être même l'intelligence artificielle d'un nouveau robot-aspirateur. Vous pouvez imaginer beaucoup d'applications à ce problème de graphe.

La recherche d'un cycle dans G est facile : il suffit de fixer un ordre arbitraire des n sommets de G, puis d'aller d'un sommet à l'autre en suivant un chemin quelconque (qui existe nécessairement puisque G est connexe). On obtient ainsi un cycle σ qui contient bien au moins une fois chaque sommet du graphe. Par contre, la recherche d'un cycle σ *de poids minimum* est bien plus compliquée. Ce problème est qualifié de « *NP-complet* » et à ce jour, on ne connaît pas d'algorithme qui résout ce problème avec un temps d'exécution polynomial en n.

2 Résolution du problème

Nous étudions dans la suite un algorithme polynomial qui résout ce problème de façon 2-*approchée*. Formellement, s'il existe une séquence σ^* de poids minimum, alors l'algorithme construit une séquence σ vérifiant

$$w(\sigma^*) \leq w(\sigma) \leq 2 \cdot w(\sigma^*).$$

Cela signifie que la séquence construite par l'algorithme a un poids qui n'est pas nécessairement le plus petit possible, mais dont on a la garantie que son poids est au plus 2 fois celui du poids minimum d'une séquence.

2.1 Algorithme pour la résolution

Nous décrivons l'algorithme qui atteint cet exploit dans la suite. L'objet de ce projet n'est pas de démontrer cette jolie propriété d'approximation ; nous aurons l'occasion d'y revenir dans vos prochaines années d'études. [1] L'objectif est d'implémenter en Python l'algorithme donné ci-après.

Algorithme RoutardApprox(G)

Calculer un arbre couvrant $\mathcal{T}$ de poids minimum de G à partir d'un sommet racine choisit arbitrairement.

Effectuer un *parcours préfixe* de l'arbre $\mathcal{T}$. Soit $\rho = (v_{i_1}, v_{i_2}, \ldots, v_{i_n})$ la séquence des n sommets de G dans l'ordre du parcours préfixe.

$\sigma \leftarrow (v_{i_1})$

pour j *de* 1 *à* $n-1$ **faire**

- Calculer un plus court chemin $\mu = (v_{i_j}, \ldots, v_{i_{j+1}})$ de v_{i_j} à $v_{i_{j+1}}$ dans G
- Retirer de μ le premier sommet (c'est-à-dire v_{i_j})
- Ajouter μ à la fin de σ

`/* On referme le cycle` σ `en revenant au point de départ: */`

1 Calculer un plus court chemin $\mu = (v_{i_n}, \ldots, v_{i_1})$ de v_{i_n} à v_{i_1} dans G

2 Retirer de μ le premier sommet (c'est-à-dire v_{i_n})

3 Ajouter μ à la fin de σ

4 **retourner** la séquence σ.

Avant de poursuivre, il est temps de faire quelques rappels de notations grecques : σ se lit *sigma*, ρ se lit *rho* et μ se lit *mu*.

L'algorithme est en réalité assez simple : après avoir calculé un arbre couvant de poids minimum, un parcours préfixe de l'arbre est effectué. Nous rappelons dans une prochaine section comment effectuer ce parcours. On obtient ainsi un ordre sur les sommets, noté ρ. Les sommets sont ensuite parcourus dans l'ordre de ρ et, pour chaque paire u, v de sommets consécutifs dans ρ, on calcule un plus court chemin dans G entre u et v. Dans la séquence σ, cela signifie que l'on va du sommet u au sommet v, en suivant ce plus court chemin. Puis, on continue avec la prochaine paire de sommets consécutifs dans ρ.

2.2 Représentation du graphe pondéré

Pour représenter un graphe avec une pondération des arêtes, nous pouvons utiliser en Python un dictionnaire de dictionnaires. Ceci facilitera le parcours du graphe et l'obtention des poids, comme le montre ce code Python :

```
G = {
    'a': {'b':1.54, 'c':2.97},
    'b': {'a':1.54, 'f':5, 'c':10.5},
    'c': {'a':2.97, 'b':10.5, 'f':2.33, 'e':4.25},
    'd': {'f':8.01, 'e':3.73},
    'e': {'c':4.25, 'd':3.73},
    'f': {'b':5, 'c':2.33, 'd':8.01}
    }
```

1. Il suffit de prouver que (1) l'arbre couvrant de poids minimum est de poids inférieur à la solution optimale pour le routard, et (2) l'algorithme produit une solution d'un poids au pire double à celui de l'arbre.

```
for u in G:
    for v in G[u]:
        print("L'arête (%s,%s) a pour poids %f"%(u,v,G[u][v]))
```

Étant donné un graphe G, vous pouvez obtenir la liste de ses sommets ainsi :

```
list(G.keys())
```

2.3 Séquences de sommets

Les séquences de sommets manipulées tout au long de l'algorithme peuvent être représentées en Python par des listes.

2.4 Arbre couvrant de poids minimum

L'algorithme demande de calculer un *arbre couvrant de poids minimum*. Il est demandé d'utiliser l'algorithme de Prim, que nous rappelons ci-après. Comme sommet racine r, vous prendrez un sommet quelconque du graphe.

Algorithme ACM-Prim($G = (V, E)$, w, r)

$F \leftarrow V$
pour chaque $u \in F$ **faire**
  $cle[u] \leftarrow \infty$
$cle[r] \leftarrow 0$
$\pi[r] \leftarrow NIL$
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow ExtraireMin(F)$
  pour chaque $v \in Adj[u]$ **faire**
    si $v \in F$ *et* $w(\{u, v\}) < cle[v]$ **alors**
      $\pi[v] \leftarrow u$
      $cle[v] \leftarrow w(\{u, v\})$

Pour la gestion de la file de priorité, vous utiliserez la bibliothèque `priority_dict.py` dont l'implémentation est basée sur des *tas*. Nous disposons des plusieurs fonctions et méthodes pour gérer une file de priorité à travers cette bibliothèque :

- `F = priority_dict()` crée une file de priorité `F`, initialement vide ;
- `F[s] = val` ajoute l'élément `s` de valeur `val` à la file `F`, ou modifie la valeur associée à `s` si `s` est déjà présent dans `F` ;
- `F.smallest()` retourne l'élément de plus petite valeur dans la file `F` ;
- `F.pop_smallest()` retourne et supprime l'élément de plus petite valeur dans la file `F`.

Voici un exemple d'utilisation en Python :

```
from priority_dict import priority_dict

F = priority_dict() # F est définie comme une file de priorité

F['x'] = 30 # l'élément x a pour clé 30
F['y'] = 10
F['z'] = 20
```

```
while F:
    u = F.pop_smallest()
    print(u)
    if u=='y':
        F['x'] = 5

# Résultat de l'exécution :
>>> y
>>> x
>>> z
```

2.5 Parcours préfixe

Etant donné un arbre $\mathcal{T}$, un ordre préfixe des sommets consiste à numéroter les sommets de 1 à n de la façon suivante : pour chaque sommet v ayant comme fils $v_1, v_2, \ldots, v_k$ (on fixe arbitrairement l'ordre des fils), on numérote d'abord v, puis tous les sommets du sous-arbre enraciné en v_1, puis en v_2, ..., puis en v_k. La Figure 1 propose un exemple.

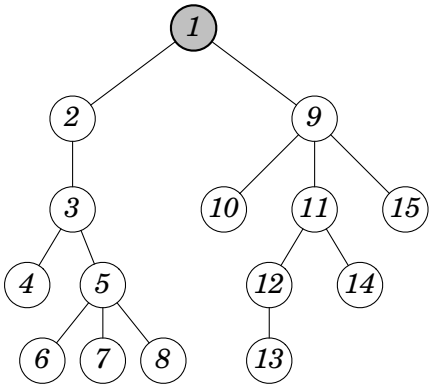


FIGURE 1 – Un exemple de parcours préfixe d'un arbre. Le parcours est effectué à partir du sommet gris (la racine de l'arbre). La numérotation obtenue par l'ordre préfixe est indiquée dans les sommets de l'arbre.

L'algorithme de PRIM construit un arbre $\mathcal{T}$ en affectant à chaque sommet son père dans l'arbre. Pour le parcours préfixe, il est certainement plus facile d'avoir, pour chaque sommet, la liste de ses fils. Cette liste peut être vide pour les feuilles de l'arbre. Vous pouvez calculer les fils de chaque sommet à partir de l'arbre $\mathcal{T}$. De plus, il est certainement plus facile d'implémenter le parcours préfixe comme une fonction récursive.

2.6 Plus courts chemins

Vous utiliserez l'algorithme de DIJKSTRA, rappelé ci-après, pour le calcul des plus cours chemins. Là encore, cet algorithme utilise une file de priorité que vous pouvez gérer en utilisant la bibliothèque `priority_dict`.

Algorithme Source-Unique-Initialisation($G = (V, E)$, s)

pour chaque *sommet* $v \in V$ **faire**
- $d[v] \leftarrow \infty$
- $\pi[v] \leftarrow NIL$

$d[s] \leftarrow 0$

Algorithme Relacher(u, v, w)

si $d[v] > d[u] + w(u, v)$ **alors**
  $d[v] \leftarrow d[u] + w(u, v)$
  $\pi[v] \leftarrow u$

Algorithme Dijkstra($G = (V, A)$, w, s)

Source-Unique-Initialisation($G = (V, A)$, s)
$X \leftarrow \emptyset$
$F \leftarrow V$ `/* file de priorité */`
tant que $F \neq \emptyset$ **faire**
  $u \leftarrow ExtraireMin(F)$ `/*` u `est aussi supprimé de` F `*/`
  $X \leftarrow X \cup \{u\}$
  pour chaque *sommet* $v \in Adj[u]$ **faire**
    Relacher(u, v, w)

2.7 Un exemple complet

La figure 2 donne un exemple à partir d'un graphe dont les sommets représentent des villes et les arêtes des routes entre ces villes. Le poids des arêtes du graphe sont égaux aux distances entre les villes.

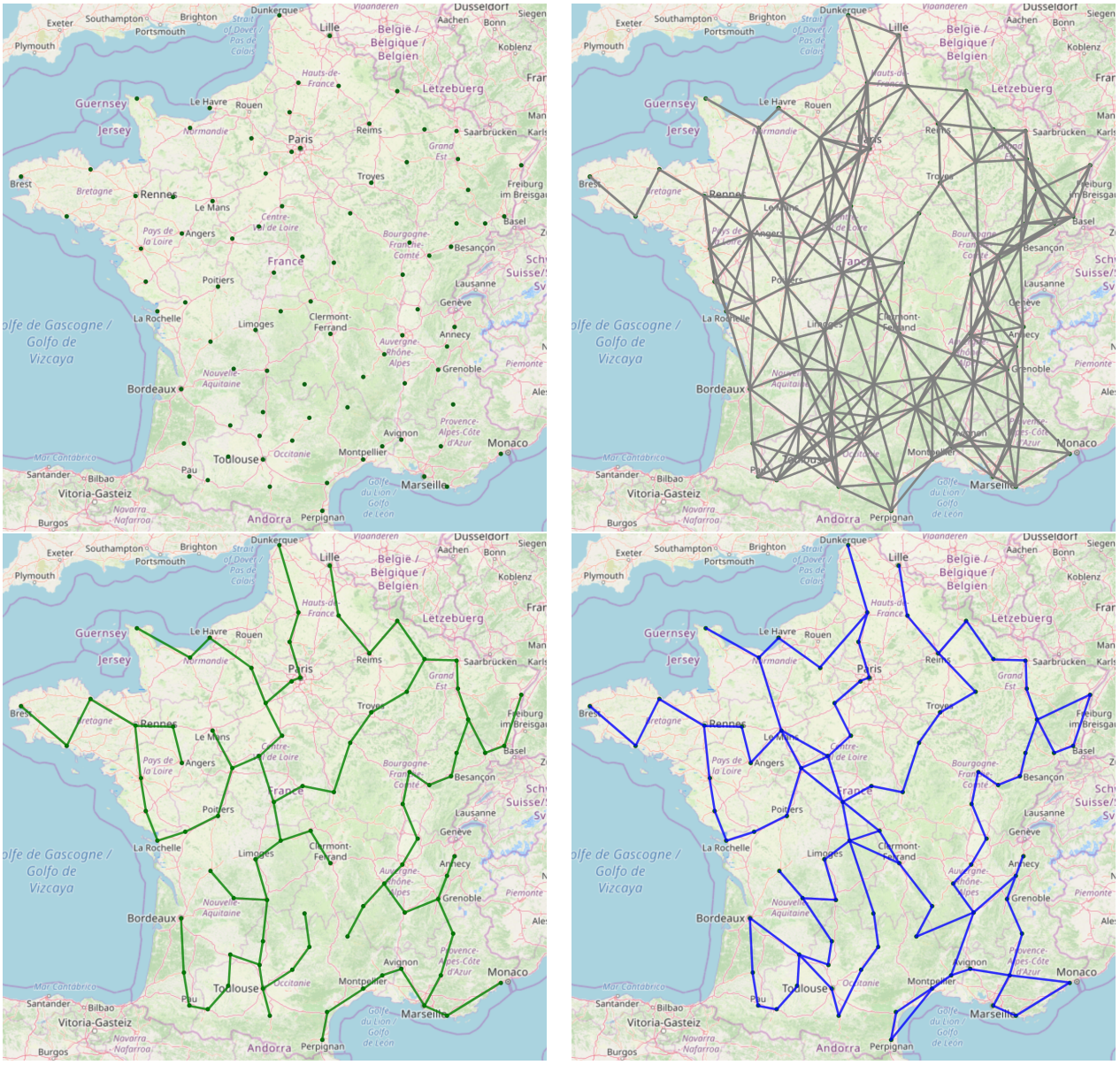
Un arbre couvrant de poids minimum est d'abord calculé. Puis un parcours préfixe du graphe donne, dans l'ordre, les sommets [2] de la liste suivante :

$\rho =$(**Chambéry**, **Grenoble**, **Gap**, **Manosque**, **Marseille**, **Toulon**, **Nice Avignon**, **Nîmes**, **Montpellier**, **Narbonne**, **Perpignan**, **Valence**, **Saint-Étienne**, **Lyon**, **Bourg-en-Bresse**, **Chalon-sur-Saône**, **Dijon**, **Dole**, **Besançon**, **Vesoul**, **Épinal**, **Belfort**, **Mulhouse**, **Strasbourg**, **Nancy**, **Metz**, **Verdun**, **Charleville-Mézières**, **Reims**, **Saint-Quentin**, **Lille**, **Saint-Dizier**, **Troyes**, **Auxerre**, **Nevers**, **Bourges**, **Châteauroux**, **Guéret**, **Montluçon**, **Clermont-Ferrand**, **Limoges**, **Brive-la-Gaillarde**, **Périgueux**, **Angoulême Cahors**, **Montauban**, **Agen**, **Auch**, **Tarbes**, **Pau**, **Mont-de-Marsan**, **Bordeaux**, **Toulouse**, **Pamiers**, **Albi**, **Rodez**, **Aurillac Blois**, **Tours**, **Poitiers**, **Niort**, **La Rochelle**, **La Roche-sur-Yon**, **Nantes**, **Rennes**, **Saint-Brieuc**, **Lorient**, **Brest Laval**, **Angers**, **Le Mans**, **Orléans**, **Chartres**, **Versailles**, **Paris**, **Beauvais**, **Amiens**, **Calais**, **Évreux**, **Le Havre**, **Caen**, **Cherbourg-Octeville**, **Le Puy-en-Velay**, **Mende**, **Annecy**).

Un plus court chemin est calculé entre chaque sommets consécutifs de la séquence ρ et permet d'obtenir la séquence σ. Nous avons laissé en gras les villes dans l'ordre de la séquence ρ ; les autres villes en italique sont celles obtenues de plus courts chemins.

$\sigma =$(**Chambéry**, **Grenoble**, **Gap**, **Manosque**, **Marseille**, **Toulon**, **Nice**, *Manosque*, **Avignon**, **Nîmes**, **Montpellier**, **Narbonne**, **Perpignan**, *Montpellier*, *Nîmes*, **Valence**, **Saint-Étienne**, **Lyon**, **Bourg-en-Bresse**, **Chalon-sur-Saône**, **Dijon**, **Dole**, **Besançon**, **Vesoul**, **Épinal**, **Belfort**, **Mulhouse**, **Strasbourg**, *Épinal*, **Nancy**, **Metz**, **Verdun**, **Charleville-Mézières**, **Reims**, **Saint-Quentin**, **Lille**, *Saint-Quentin*, *Reims*, **Saint-Dizier**, **Troyes**, **Auxerre**, **Nevers**, **Bourges**, **Châteauroux**, **Guéret**, **Montluçon**, **Clermont-Ferrand**, *Guéret*, **Limoges**, **Brive-la-Gaillarde**, **Périgueux**, **Angoulême**, *Périgueux*, **Cahors**, **Montauban**, **Agen**, **Auch**, **Tarbes**, **Pau**, **Mont-de-Marsan**, **Bordeaux**, *Agen*, **Toulouse**, **Pamiers**, **Albi**, **Rodez**, **Aurillac**, *Guéret*, *Châteauroux*, **Blois**, **Tours**, **Poitiers**, **Niort**, **La Rochelle**, **La Roche-sur-Yon**, **Nantes**, **Rennes**, **Saint-Brieuc**, **Lorient**, **Brest**, *Lorient*, *Saint-Brieuc*, *Rennes*, **Laval**, **Angers**, **Le Mans**, *Blois*, **Orléans**, **Chartres**, **Versailles**, **Paris**, **Beauvais**, **Amiens**, **Calais**, *Amiens*, **Évreux**, **Le Havre**, **Caen**, **Cherbourg-Octeville**, *Caen*, *Le Mans*, *Tours*, *Châteauroux*, *Montluçon*, *Clermont-Ferrand*, **Le Puy-en-Velay**, **Mende**, *Valence*, *Chambéry*, **Annecy**, *Chambéry*).

2. Si besoin, wikipedia vous aidera à replacer ces villes sur la carte.



σ =(**Chambéry**, **Grenoble**, **Gap**, **Manosque**, **Marseille**, **Toulon**, **Nice**, Manosque, **Avignon**, **Nîmes**, **Montpellier**, **Narbonne**, **Perpignan**, Montpellier, Nîmes, **Valence**, **Saint-Étienne**, **Lyon**, **Bourg-en-Bresse**, **Chalon-sur-Saône**, **Dijon**, **Dole**, **Besançon**, **Vesoul**, **Épinal**, **Belfort**, **Mulhouse**, **Strasbourg**, Épinal, **Nancy**, **Metz**, **Verdun**, **Charleville-Mézières**, **Reims**, **Saint-Quentin**, **Lille**, Saint-Quentin, Reims, **Saint-Dizier**, **Troyes**, **Auxerre**, **Nevers**, **Bourges**, **Châteauroux**, **Guéret**, **Montluçon**, **Clermont-Ferrand**, Guéret, **Limoges**, **Brive-la-Gaillarde**, **Périgueux**, **Angoulême**, Périgueux, **Cahors**, **Montauban**, **Agen**, **Auch**, **Tarbes**, **Pau**, **Mont-de-Marsan**, **Bordeaux**, Agen, **Toulouse**, **Pamiers**, **Albi**, **Rodez**, **Aurillac**, Guéret, Châteauroux, **Blois**, **Tours**, **Poitiers**, **Niort**, **La Rochelle**, **La Roche-sur-Yon**, **Nantes**, **Rennes**, **Saint-Brieuc**, **Lorient**, **Brest**, Lorient, Saint-Brieuc, Rennes, **Laval**, **Angers**, **Le Mans**, Blois, **Orléans**, **Chartres**, **Versailles**, **Paris**, **Beauvais**, **Amiens**, **Calais**, Amiens, **Évreux**, **Le Havre**, **Caen**, **Cherbourg-Octeville**, Caen, Le Mans, Tours, Châteauroux, Montluçon, Clermont-Ferrand, **Le Puy-en-Velay**, **Mende**, Valence, Chambéry, **Annecy**, Chambéry)

FIGURE 2 – Un exemple d'exécution de `RoutardApprox`. En haut à gauche : un graphe où les points représentent ses sommets. En haut à droite : le graphe avec ses arêtes ; le poids des arêtes est égal à la distance entre les points. En bas à gauche : l'arbre couvrant de poids minimum de ce même graphe. En bas à droite : le cycle σ obtenu du parcours préfixe de l'arbre couvrant puis des plus courts chemins. Le tour σ avoisine les 10000 km.

3 Construction des graphes difficiles

Puisque le problème du routard est difficile à résoudre, on pense que même pour l'algorithme `RoutardApprox`, il doit exister des graphes pour lesquels la séquence σ trouvée est loin d'être la meilleure possible. Néanmoins, nous avons la garantie que l'algorithme `RoutardApprox` ne retournera jamais une séquence dont le poids est plus grand que deux fois le plus petit poids possible.

On se propose d'écrire une fonction `ConstruireGrapheDifficile(n)` qui attend en paramètre un entier n. Votre fonction doit construire un graphe connexe pondéré contenant n sommets. Si on note σ^* une séquence de poids minimum au problème du routard, votre fonction doit construire un graphe pour lequel l'algorithme `RoutardApprox` *pourrait construire* une séquence σ telle que $w(\sigma)$ soit très proche de $2 \cdot w(\sigma^*)$.

Bien sûr, la construction d'une séquence σ par `RoutardApprox` dépend du choix de l'arbre couvrant minimum et de sa racine, de l'ordre d'exploration des fils de chaque noeud lors du parcours préfixe, et des choix des plus courts chemins calculés. L'algorithme ne spécifie pas de choix particuliers pour ces éléments et étant donné un graphe construit par `ConstruireGrapheDifficile` il se peut que l'exécution de votre fonction `RoutardApprox` ne retourne pas nécessairement une séquence σ dont le poids serait très grand et proche de $2 \cdot w(\sigma^*)$. Vous devez donc expliquer pourquoi, il se pourrait tout de même, que `RoutardApprox` calcule un cycle σ dont le poids serait très grand et proche de $2 \cdot w(\sigma^*)$ sur le graphe construit par votre fonction `ConstruireGrapheDifficile(n)`.

La fonction `ConstruireGrapheDifficile(n)` retournera un couple (G, s) où G sera un graphe et s une séquence correspondant à une séquence optimale σ^* au problème du routard. Là encore, la séquence s est représentée en Python par une liste de sommets. Cette séquence σ^* de poids minimum, vous devez la construire vous-même (sans application de l'algorithme `RoutardApprox`) car vous connaissez la construction particulière de votre graphe G et donc une solution optimale.

4 Consignes, travail à rendre et évaluation

4.1 Le programme

Vous devez implémenter l'algorithme `RoutardApprox` en Python.

Afin de pouvoir tester votre code, le fichier `RoutardApprox.py` devra contenir une fonction `RoutardApprox(G)` qui attend en entrée un graphe G et qui retourne en sortie une liste. Cette liste correspond à une séquence σ de sommets obtenue de l'application de l'algorithme. Elle constituera donc une solution 2-approchée au problème décrit. N'oubliez pas que le premier sommet de σ doit être le même que le dernier (puisqu'on referme le cycle en revenant au sommet de départ). Vous pouvez parfaitement programmer d'autres fonctions supplémentaires dans le fichier `RoutardApprox.py` afin de rendre votre programme plus lisible.

Pour tester votre code, nous utiliserons un programme similaire à celui contenu dans `RoutardApproxWrapper.py` et mis à votre disposition. Ce programme se charge d'importer la fonction `RoutardApprox` de votre fichier `RoutardApprox.py` puis de l'exécuter sur différents graphes. Si vous souhaitez tester que nous arriverons bien à importer votre programme, nous mettons à votre disposition le programme `RoutardApproxWrapper.py`.

Nous mettons également à votre disposition des graphes stockés respectivement dans les fichiers `graphe1.json`, `graphe2.json`, `graphe3.json` et `graphe4.json`.

L'évaluation de votre programme portera sur les critères suivants :
— la clarté du code et ses commentaires ;
— l'implémentation de l'algorithme de Prim ;
— l'implémentation du parcours préfixe ;
— l'implémentation de l'algorithme de Dijkstra ;
— le passage d'une série de tests.

4.2 Le rapport

Vous fournirez (maximum 2 pages, au format PDF) des explications sur la construction du graphe par votre fonction `ConstruireGrapheDifficile`. En particulier vous donnerez la séquence σ^* optimale (c'est-à-dire, de poids minimum) et vous expliquerez les mauvais choix que pourrait faire l'algorithme `RoutardApprox` sur ce graphe pour construire une séquence σ dont le poids $w(\sigma)$ serait proche de $2 \cdot w(\sigma^*)$.

L'évaluation de votre fonction `ConstruireGrapheDifficile` tiendra compte :
— de la proximité de $w(\sigma)$ par rapport à $2 \cdot w(\sigma^*)$;
— de la clarté de vos explications dans le PDF.

4.3 Le rendu

Ce projet est individuel et doit être réalisé seul. Vous devez rendre un unique fichier Python (`RoutardApprox.py`) et un unique fichier PDF (`rapport.pdf`) contenant vos explications. Dans le fichier PDF, vous mentionnerez votre nom et prénom.

Ces deux fichiers sont à déposer sur la page Celene du cours pour le 18 mai 2020, 8h, au plus tard.

`http://celene.univ-orleans.fr`