INF 8215 - Intelligence artif.: méthodes et algorithmes

Automne 2018 - TP1 - Méthodes de recherche

Membres de l'équipe

- Jean-Raphael Cornel
- Nathan Heraief
- Paul-Arthur Thiery

LE VÉLO À MONTRÉAL

Chaque année, Montréal accueille à peu près 10 millions de touristes. Soucieuse de la qualité de leur séjour, Tourisme Montréal a entamé un projet de développement d'une nouvelle application mobile afin d'assister les touristes lors de leurs déplacements dans la ville. Cette application a pour but d'aider l'utilisateur à planifier sa visite des importantes attractions de la ville, de la façon la plus efficace possible (ie, sur la durée la plus courte). Étant donné qu'il a été observé que le moyen de transport privilégié des touristes pour explorer Montréal est le vélo, cette application a pour but de générer des circuits cyclables de durée minimale. Plus précisément, étant donné une liste d'attractions munie de points de départ et d'arrivée, la tâche est de proposer, à chaque fois, un chemin qui passe par toutes les attractions indiquées une seule fois, qui débute au point de départ et qui s'achève au point d'arrivée et dont la durée de trajet est minimale.

Le travail demandé dans ce TP est de développer l'algorithme interne de l'application. Nous explorerons trois mécanismes de résolution différents : 1. Définition et exploration naïve d'un arbre de recherche 2. Exploration plus efficace en utilisant l'algorithme A* 3. Optimisation locale en utilisant une métaheuristique de recherche à voisinage variable (Variable Neighborhood Search, VNS)

PRÉSENTATION DU PROBLÈME

Une façon naturelle de représenter notre problème est d'utiliser un graphe $G = (V, A)$ dirigé et complet. Chaque sommet dans V est une attraction donnée et chaque arc dans A représente une piste cyclable entre deux attractions distinctes. Chaque paire de sommets i et j est reliée par une paire d'arcs a_{ij} et a_{ji} dont les poids respectifs $w(a_{ij})$ et $w(a_{ji})$ ne sont pas nécessairement égaux. Concrètement, ces poids représentent la durée du trajet d'un sommet à l'autre (ainsi, w est telle que $w : A \rightarrow \mathrm{R}^+$).

La liste des attractions à visiter est indiquée comme la suite $P = (p_1, \ldots, p_m)$ où p_1 et p_m sont les sommets de départ et d'arrivée, respectivement.

[2]
```
import numpy as np

def read_graph():
    return np.loadtxt("montreal", dtype='i', delimiter=',')
```

```
graph = read_graph()
```

Notre première tâche est de définir la classe qui représente une solution partielle. Son constructeur est donné et reçoit comme argument la liste des sommets (attractions P) à visiter et le graphe (G). Celui-ci crée la solution S_{root} avec les attributs suivants :

- g : le coût de la solution partielle
- visited : représente $V(S)$, discuté plus haut. Par définition, vistited$[-1]$ représente le dernier sommet ajouté, c.
- not_visited : représente $P \setminus V(S)$
- graph: représente le graphe G

Ensuite, il est demandé d'implanter la méthode add qui mets à jour la solution partielle en ajoutant une nouvelle attraction à visiter parmi la liste not_visited. Cette méthode reçoit comme arguments l'index du sommet à visiter parmi not_visited ainsi que le graphe courant.

Implantez add :

[3]
```
mport copy

lass Solution:
    def __init__(self, places, graph):
        """
        places: a list containing the indices of attractions to visit
        p1 = places[0]
        pm = places[-1]
        """
        self.g = 0 # current cost
        self.graph = graph
        self.visited = [places[0]] # list of already visited attractions
        self.not_visited = copy.deepcopy(places[1:]) # list of attractions n

    def __lt__(self, other):
        if(self.g<other.g):
            return true
        return false

    def add(self, idx):
        """
        Adds the point in position idx of not_visited list to the solution
        """
        self.visited.append(self.not_visited[idx])
        self.g += self.graph[self.visited[-2]][self.visited[-1]]
        self.not_visited.pop(idx)
```

La prochaine étape est d'implanter une stratégie de parcours de l'arbre de recherche. Une première méthode simple est naïve est de mettre en œuvre une recherche en largeur (Breadth-first search, BFS).

Implantez bfs qui mets en œuvre cette recherche. Elle prend en arguments le graphe courant ainsi que la liste des attractions à visiter P et elle retourne la meilleure solution trouvée.

[4]
```python
from queue import Queue

def bfs(graph, places):
    """
    Returns the best solution which spans over all attractions indicated
    """
    solution = Solution(places, graph)
    queue = Queue()
    queue.put(solution)
    finalSolutions = []
    while(queue.empty()==False):
        currentSolution = queue.get()
        if(currentSolution.not_visited == [places[-1]]):
            currentSolution.add(0)
            finalSolutions.append(currentSolution)
        else:
            for i in range(len(currentSolution.not_visited)) :
                tempSolution = copy.deepcopy(currentSolution)
                tempSolution.add(i)
                queue.put(tempSolution)
    min = 1000
    for i in range(len(finalSolutions)):
        if(finalSolutions[i].g<min):
            min = finalSolutions[i].g
            solutionToReturn = finalSolutions[i]
    return solutionToReturn
```

1.2 Expérimentations

On propose trois exemples d'illustration pour tester notre recherche en largeur. Le premier exemple prend en compte 7 attractions, le second 10 et le dernier 11. Vu que cette recherche énumère toutes les solutions possibles, le troisième exemple risque de prendre un temps considérable à s'achever.

Mettez en œuvre ces expériences et notez le nombre de nœuds explorés ainsi que le temps de calcul requis.

[7]
```python
import time

#test 1  --------------  OPT. SOL. = 27
start_time = time.time()
places=[0, 5, 13, 16, 6, 9, 4]
sol = bfs(graph=graph, places=places)
print(sol.g)
print(sol.graph)
```

```
print("--- %s seconds ---" % (time.time() - start_time))
```

```
27
[[ 0  3  8 12 15 17 20 13 18 20 23 22 20 21 21 23 26 22 25 28 29]
 [ 2  0  4 10 12 14 17  8 16 16 20 20 15 17 18 18 24 17 23 25 25]
 [ 8  4  0  2  5 10 10  3 10 11 15 12 11 14 11 14 19 13 15 20 21]
 [10 10  2  0  3  5  7  1  5  6  9  8  7  7 10 12 12 11 12 15 16]
 [14 12  4  4  0  2  6  3  1  3  6  6  5  4  5  7 10  5  9 14 13]
 [18 12  9  4  1  0  4  2  2  2  7  5  4  3  2  4  8  3  6  9 12]
 [20 16  8  7  5  2  0  7  1  1  4  2  1  1  1  1  4  2  3  9 10]
 [11  6  3  2  4  3  6  0  3  6  9  8  7  9  9 11 13  7 12 13 16]
 [19 16  9  5  1  3  1  2  0  3  6  3  1  4  4  5  6  5  7 10 12]
 [19 15  9  6  1  1  1  7  1  0  1  3  1  1  2  3  6  1  3  9 10]
 [23 20 16  7  4  8  5  7  5  2  0  2  3  3  2  1  3  1  3  4  5]
 [21 19 12  7  5  4  2  7  1  2  1  0  1  1  2  2  2  1  4  5  7]
 [19 14 11  6  6  2  2  6  1  2  1  2  0  1  1  2  4  3  6  6  7]
 [20 15 15  5  2  3  1  8  3  2  1  2  2  0  1  1  3  1  4  8  9]
 [22 18 10  8  4  2  1  9  3  1  3  1  1  2  0  3  6  1  3  6  6]
 [22 16 15 12  8  2  1 11  5  3  2  3  1  1  1  0  4  1  1  6  4]
 [26 25 20 13 11  8  2 12  5  6  3  1  4  4  5  4  0  4  1  2  3]
 [22 16 13 10  6  2  1  5  5  2  2  2  1  2  2  2  4  0  3  4  8]
 [26 22 13 13  8  6  3 13  6  4  4  2  6  4  3  2  2  3  0  3  3]
 [27 23 19 13 15  8  8 13 11 10  5  4  6  7  5  4  3  2  3  0  2]
 [28 26 22 17 13 13 10 14 12 10  3  7  7  9  4  5  2  6  2  3  0]]
--- 0.10705995559692383 seconds ---
```

[6]
```
#test 2 -------------- OPT. SOL. = 30
start_time = time.time()
places=[0, 1, 4, 9, 20, 18, 16, 5, 13, 19]
sol = bfs(graph=graph, places=places)
print(sol.g)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
30
--- 43.761394739151 seconds ---
```

[45]
```
#test 3 -------------- OPT. SOL. = 26
start_time = time.time()
places=[0, 2, 7, 13, 11, 16, 15, 7, 9, 8, 4]
sol = bfs(graph=graph, places=places)
print(sol.g)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
26
--- 319.4595263004303 seconds ---
```

2. RECHERCHE GUIDÉE À L'AIDE DE L'ALGORITHME A* (7.5 points)

Pour notre deuxième méthode de recherche, au lieu d'énumérer toutes les solutions possibles, nous effectuons une recherche guidée à l'aide de l'algorithme A*. Comme vu en classe, A* est une recherche où les nœuds à explorer sont priorisés en fonction du coût courant d'une

solution $g(S)$ ainsi que d'une estimation du coût restant vers la solution finale donné par une heuristique $h(S)$.

Dans le cas d'une minimisation, $h(S)$ est une borne inférieure du coût réel restant et on priorise l'exploration des nœuds dont $f(S) = g(S) + h(S)$ est le plus petit. Avec cette méthode, la première solution complète trouvée est assurément la solution optimale.

Pour une solution donnée S avec un dernier sommet visité c, une possible fonction h est telle que :

$h(S) =$ Le poids du chemin le plus court entre c et p_m dans le sous graphe G_S contenant les sommets $P \setminus V(S) \cup \{c\}$

Remarque que ce chemin le plus court utilisé dans le calcul de l'estimation h entre l'attraction courante et l'arrivée ne passera pas nécessairement pas tous les sommets restants.

Notre algorithme A* se présente comme ceci : 1. Définir l'arbre de recherche T exactement comme auparavant. Le calcul de h pour la solution initiale est inutile : c'est la seule solution qu'on a. 2. Sélectionner le meilleur nœud candidat pour expansion. La solution partielle S_b de ce nœud candidat est telle que :

$$f(S_b) \leq f(S) \quad \forall S \in \mathrm{T} \qquad S_B, S \text{ pas encore sélectionnés}$$

Si S_b est une solution complète, l'algorithme s'arrête et S_b est assurément la solution optimale,

[13]
```python
def fastest_path_estimation(sol):
    """
    Returns the time spent on the fastest path between
    the current vertex c and the ending vertex pm
    """
    c = sol.visited[-1]
    tempDistances = [10000]*len(sol.not_visited)
    pm = c
    traveledDistance = 0
    while(pm != sol.not_visited[-1]) :
        for i in range(len(tempDistances)) :
            if(tempDistances[i] != -1 and sol.graph[pm][sol.not_visited[i
                tempDistances[i] = sol.graph[pm][sol.not_visited[i]]+trav
        pm = sol.not_visited.index(min(tempDistances))
        traveledDistance = tempDistances[tempDistances.index(min(tempDist
        tempDistances[tempDistances.index(min(tempDistances))] = -1

    return traveledDistance
```

Finalement, il est temps d'implanter A*. On aura besoin d'une file de priorité qui retournera toujours le meilleur nœud candidat de T pour l'étendre (l'opérateur surchargé de comparaison assure cela).

Prescriptions d'implantation (cf. détail des étapes de l'algorithme plus haut) :

- Tant que les solutions extraites de la file de priorité ne sont pas complètes :
  - Sélectionner et étendre le nœud extrait de la file comme détaillé plus haut
  - Calculer g et h pour chaque nouvelle solution partielle obtenue
  - Remettre ces solutions dans la file

- Retourner la première solution complète extraite de la file (c'est la solution optimale)

```
[ ]  import heapq

     def A_star(graph, places):
         """
         Performs the A* algorithm
         """

         # blank solution
         root = Solution(graph=graph, places=places)

         # search tree T
         T = []
         heapq.heapify(T)
         heapq.heappush(T, root)
```

2.2 Expérimentations

On ajoute un Quatrième exemple d'exécution avec 15 attractions. Là encore, mettez en œuvre ces expériences avec le nouvel algorithme A* conçu et notez le nombre de nœuds explorés ainsi que le temps de calcul requis.

```
[ ]  #test 1  --------------  OPT. SOL. = 27
     start_time = time.time()
     places=[0, 5, 13, 16, 6, 9, 4]
     astar_sol = A_star(graph=graph, places=places)
     print(astar_sol.g)
     print(astar_sol.visited)
     print("--- %s seconds ---" % (time.time() - start_time))
```

```
[ ]  #test 2  --------------  OPT. SOL. = 30
     start_time = time.time()
     places=[0, 1, 4, 9, 20, 18, 16, 5, 13, 19]
     astar_sol = A_star(graph=graph, places=places)
     print(astar_sol.g)
     print(astar_sol.visited)
     print("--- %s seconds ---" % (time.time() - start_time))
```

```
[ ]  #test 3  --------------  OPT. SOL. = 26
     start_time = time.time()
     places=[0, 2, 7, 13, 11, 16, 15, 7, 9, 8, 4]
     astar_sol = A_star(graph=graph, places=places)
     print(astar_sol.g)
     print(astar_sol.visited)
     print("--- %s seconds ---" % (time.time() - start_time))
```

```
[ ]  #test 4  --------------  OPT. SOL. = 40
```

```
start_time = time.time()
places=[0, 2, 20, 3, 18, 12, 13, 5, 11, 16, 15, 4, 9, 14, 1]
astar_sol = A_star(graph=graph, places=places)
print(astar_sol.g)
print(astar_sol.visited)
print("--- %s seconds ---" % (time.time() - start_time))
```

2.3 Une meilleure borne inférieure

Notre algorithme A* est déjà beaucoup plus efficace qu'une recherche naïve. Cependant, la qualité de l'heuristique h a un très grand impact sur la vitesse de A*. Une heuristique plus serrée devrait accélérer A* de façon significative. Notre estimation h basée sur Dijkstra est très large à cause du fait qu'elle ne considère pas toutes les attractions restantes.

Une meilleure heuristique pourrait être basée sur la **Spanning Arborescence of Minimum Weight** qui s'apparente à une Minimum Spanning Tree pour graphes orientés. On propose de construire une telle Spanning Arborescence sur le reste des attractions $P \setminus V(S) \cup \{c\}$. Ici la racine est la dernière attraction visitée c. Une façon classique de résoudre ce problème est d'utiliser l'algorithme de Edmonds.

Implantez cet algorithme et refaites les expériences avec A* en utilisant cette nouvelle heuristique :

```
[ ]  def minimum_spanning_arborescence(sol):
         """
         Returns the cost to reach the vertices in the unvisited list
         """
```

3. RECHERCHE LOCALE À VOISINAGE VARIABLE (7.5 points)

Cette fois-ci, au lieu de construire une solution optimale depuis une solution vide, on commence d'une solution complète, non-optimale, qu'on améliore à l'aide d'une recherche locale en utilisant une recherche locale à voisinage variable (Variable Neighborhood Search, VNS).

3.1 Code

On commence par créer une solution initiale. Celle-ci est une suite ordonnée des attractions de p_1 à p_m dans P. Pour cela, on fait appel à une recherche en profondeur (Depth-First Search, DFS) qu'on arrête aussitôt qu'une solution complète est trouvée. Pour aider à diversifier la recherche, la méthode permettant de générer une solution initiale peut être randomisée de telle sorte que l'algorithme VNS puisse lancer la recherche dans différentes régions de l'espace solution. Ainsi, dans la fonction DFS, la sélection de l'enfant pour continuer la recherche doit être aléatoire.

Prescriptions d'implantation :

- Mettre en œuvre une recherche en profondeur
- Créer un objet $\mathrm{Solution}$ relatif à cette solution
- Ajuster les attributs de cet objet avec les bonnes valeurs de coûts et d'attractions visitées
- Retourner la solution trouvée.

```
[ ]  from random import shuffle, randint

     def initial_sol(graph, places):
         """
         Return a completed initial solution
         """

     def dfs(...):
         """
         Performs a Depth-First Search
         """
```

Pour définir une VNS, il faut définir les $k_{\max}$ voisinages de recherche locale possibles. Pour notre problème, une bonne et simple répartition des voisinages est telle qu'un voisinage k correspond à la permutation de k-paires de sommets dans $V(S)$.

On appelle **shaking** l'étape de génération d'une solution dans le voisinage k. Le travail qui suit correspond à l'implantation de cette étape. shaking admet 3 arguments que sont la solution de départ, l'indice du voisinage k ainsi que le graph courant.

Attention, avant d'implanter shaking, il est nécessaire de créer une méthode swap dans la classe Solution. Cette méthode permet de mettre en œuvre la permutation dans une solution donnée (en mettant à jour tous les attributs nécessaires pour que la solution soit cohérente).

Prescriptions d'implantation de shaking :

- Sélectionner au hasard deux indices i et j différents et tels que $i, j \in \{2, \ldots, m-1\}$
- Faire une copie de la solution courante et faire la permutation
- Retourner la solution créée

```
[ ]  def shaking(sol, k):
         """
         Returns a solution on the k-th neighrboohood of sol
         """
```

Une dernière étape essentielle dans une VNS est l'application d'un algorithme de recherche locale à la solution issue du shaking. Pour cela, on propose la recherche locale 2-opt. Celle-ci intervertit deux arcs dans la solution, à la recherche d'une qui est meilleure.

Pour un sommet i, soit $i^{'}$ le successeur immédiat de i dans la séquence $V(S)$. L'algorithme 2-opt fonctionne comme suit: pour chaque paire de sommets non consécutifs i, j, vérifiez si en échangeant la position des sommets $i^{'}$ et j entraîne une amélioration du coût de la solution. Si oui, effectuez cet échange. Ce processus se répète jusqu'à ce qu'il n'y ait plus d'échanges rentables. On réalise cette opération pour toutes les paires d'arcs éligibles à la recherche du plus petit coût.

Implantez local_search_2opt.

Prescriptions d'implantation :

- Considérer chaque paire d'indices $i = \{2, . . , m - 3\}$ and $j = \{i + 2, m - 1\}$
- Si l'échange donne un plus bas coût, on le réalise
- Répéter jusqu'à optimum local.

```
[ ]  def local_search_2opt(sol):
         """
         Apply 2-opt local search over sol
         """
```

Finalement, il est temps d'implanter notre VNS. La méthode vns reçoit une solution complète, le graphe courant, le nombre maximal de voisinages et un temps de calcul limite. Celle-ci retourne la solution optimale trouvée

Prescriptions d'implantation :

- À chaque itération, la VNS génère une solution dans le k-ème voisinage (shaking) à partir de la meilleure solution courante et applique une recherche locale 2-opt dessus
- Si la nouvelle solution trouvée a un meilleur coût, mettre à jour la meilleure solution courante
- Répéter le processus jusqu'à t_max

```
[ ]  def vns(sol, k_max, t_max):
         """
         Performs the VNS algorithm
         """
```

3.2 Experiments

Mettez en oeuvre la VNS sur les exemples d'illustration suivants et raportez les solutions obtenue:

```
[ ]  # test 1  --------------  OPT. SOL. = 27
     places=[0, 5, 13, 16, 6, 9, 4]
     sol = initial_sol(graph=graph, places=places)
     start_time = time.time()
     vns_sol = vns(sol=sol, k_max=10, t_max=1)
     print(vns_sol.g)
     print(vns_sol.visited)
     print("--- %s seconds ---" % (time.time() - start_time))
```

```
[ ]  #test 2  --------------  OPT. SOL. = 30
     places=[0, 1, 4, 9, 20, 18, 16, 5, 13, 19]
     sol = initial_sol(graph=graph, places=places)

     start_time = time.time()
     vns_sol = vns(sol=sol, k_max=10, t_max=1)
     print(vns_sol.g)
```

```
print(vns_sol.visited)

print("--- %s seconds ---" % (time.time() - start_time))
```

```
[ ]
# test 3  --------------  OPT. SOL. = 26
places=[0, 2, 7, 13, 11, 16, 15, 7, 9, 8, 4]
sol = initial_sol(graph=graph, places=places)

start_time = time.time()
vns_sol = vns(sol=sol, k_max=10, t_max=1)
print(vns_sol.g)
print(vns_sol.visited)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
[ ]
# test 4  --------------  OPT. SOL. = 40
places=[0, 2, 20, 3, 18, 12, 13, 5, 11, 16, 15, 4, 9, 14, 1]
sol = initial_sol(graph=graph, places=places)

start_time = time.time()
vns_sol = vns(sol=sol, k_max=10, t_max=1)
print(vns_sol.g)
print(vns_sol.visited)
print("--- %s seconds ---" % (time.time() - start_time))
```

4. BONUS (1 point)

Expliquez dans quelle situation chacun des algorithmes développés est plus approprié (prenez en compte l'évolutivité du problème)

```
[ ]
```