ALGORITHMIQUE AVANCÉE

Voyageur de commerce : 3. heuristiques et approximations

QUALITÉS DES RÉPONSES INEXACTES

S'il est constaté qu'obtenir des solutions optimales est trop difficile, peut-on se contenter de choisir de bonnes solutions sous-optimales. Quelles garanties pouvons-nous avoir sur ces solutions sous-optimales ?

Cette nouvelle feuille de tp sert à faire la distinction entre une heuristique et un solver. Quand on écrit un algorithme qui détermine la solution optimale on veut avoir la garantie qu'il s'agit bien de la solution optimale.Vous vous êtes rendus compte dans la Feuille 2 qu'obtenir une solution optimale n'est pas si évident que cela. On se dit alors qu'il serait plus simple d'obtenir des solutions quasi-optimales, d'autant plus que pour le voyageur de commerce si on trouve une solution à 5% de l'optimal cela nous convient.

Question 1. *Première heuristique par tirage aléatoire de solutions. Notions d'heuristique et de calibrage du temps d'exploration.*

Cette première heuristique est très naïve, elle sert essentiellement à faire la distinction entre une heuristique et un algorithme mais aussi à s'intéresser aux garanties que nous offre cette heuristique.

Dans cette première question on va tirer au sort des 10000 tournées aléatoires et parmi ces 10000 tournées aléatoires on va sélectionner la meilleure. On remarquera que le nombre d'essais doit être inférieur au nombre de permutations car sinon cela revient à faire de l'exhaustif. Par exemple, pour n villes vous avez $n!$ permutations donc on peut faire n^2 ou n^3 ou même 2^n essais qui restent négligeables par rapport à $n!$ mais surtout pas $n!/2$, dans ce cas, autant faire toutes les permutations.

1. Utilisez la fonction

   ```
   def random_permutation(n:int)->List[int]:
   ```

   pour ne retenir que la meilleure solution dans

   ```
   def solve_by_best_among_random_solutions(tsp:Tsp,nb_attempts:int)->List[int]:
   ```

   SOLUTION.

   ```
   #<TODO 3.1>
   def solve_by_best_among_random_solutions(tsp:Tsp,number_of_attempts:int)->(float,Permutation):
       smallest_length = None
       best_permutation = None
       for t in range(number_of_attempts):
           p = random_permutation(tsp.n)
           its_length = total_length(tsp.n,tsp.cities,p)
           if smallest_length == None or its_length < smallest_length:
               smallest_length = its_length
               best_permutation = copy.deepcopy(p)
               if not tsp.skip_print:
                   # draw each new record
                   print('{0}/{1}'.format(t,number_of_attempts))
               tsp.permutation = best_permutation
               tsp.draw()
       return (smallest_length,tsp.permutation)

   #</TODO 3.1>
   ```

2. Observez dans les tests la qualité de cette solution sur les tailles accessibles par la méthode exhaustive via le test *TEST 3.1.a*.

   SOLUTION. À vous de jouer.

3. Empiriquement via les statistiques qu'elle semble être la qualité de la solution (en fonction du nombre de tirages aléatoires de solutions).

   SOLUTION. À vous de jouer.

4. En théorie, quelle garantie avez-vous sur la qualité de la solution par cette méthode ?
   Solution.
   On a aucune garantie en utilisant cette méthode. On peut imaginer qu'à chaque fois qu'on utilise `random_permutation` cette fonction nous renvoie systématiquement la pire permutation. Au final, notre heuristique nous renverra la plus mauvaise tournée.
5. Que pensez-vous des solutions proposées par cette heuristique avec $65536 = 2^{16}$ essais sur des problèmes de plus en plus grande taille visualisées par le test *TEST 3.1.b* ?
   Solution. À vous de jouer.

Question 2. *Utilisez l'heuristique pour fixer un seuil plus efficace dans l'exploration garantie.*

Une exploration sans garantie de l'espace des solutions vous permet d'obtenir une première solution qui donne une garantie sans lien avec la longueur de la tournée optimale. En effet la longueur ℓ de cette première solution assure l'existence d'une solution sous (au sens large) le seuil ℓ.

1. Utilisez ce seuil dans une nouvelle version de la fonction

   ```
   def solve_by_truncated_heuristic_threshold\
   (tsp:Tsp, heuristic:Callable[[Tsp],Tuple[float,Permutation]] ) -> (float,Permutation):
   ```

   qui prend en paramètre une heuristique comme

   ```
   heuristic = lambda tsp: solve_by_best_among_random_solutions(tsp,2**4)
   ```

   où vous avez l'occasion d'observer l'intérêt de cacher un paramètre dans une fonction anonyme.
   Solution.

   ```
   #<TODO 3.2>

   def solve_by_truncated_heuristic_threshold\
   (tsp:Tsp,heuristic:Callable[[Tsp],Tuple[float,Permutation]])->(float,Permutation):
       smallest_length = heuristic(tsp)[0]
       best_permutation = None
       t = 0
       p = first_permutation(tsp.n)
       while True:
           t += 1
           (its_length,partial_length) = total_length_stopped(tsp.n,tsp.cities,p,smallest_length)
           if partial_length == tsp.n:
               if its_length < smallest_length:
                   smallest_length = its_length
                   best_permutation = copy.deepcopy(p)
                   if not tsp.skip_print:
                       # draw each new record
                       print('{0}/{1}'.format(t,math.factorial(tsp.n)))
                   tsp.permutation = best_permutation
                   tsp.draw()
           else:
               last_permutation_with_fixed_prefix(p,partial_length-2)
           if not next_permutation(p):
               break
       return (smallest_length,tsp.permutation)

   #</TODO 3.2>
   ```

2. Validez avec le test *TEST 3.2.a* puis observez les performances avec le test *TEST 3.2.b*. Elles ne sont pas terribles mais cela pourrait s'améliorer avec des heuristiques plus efficaces.
   Solution. À vous de jouer.

Vous êtes arrivés à la fin des tests écrits de `runtest.py`, nous espérons que vous en êtes suffisamment familier pour pouvoir avoir envie d'en écrire vous-même et savoir le faire (plutôt que de confesser qu'effectivement nous n'avons pas eu le temps de les écrire). L'avantage est que cela vous laisse peut-être plus libre dans la déclaration de vos fonctions.

Question 3. *Heuristique gloutonne.*

1. Implémentez la solution tirant un sommet au hasard puis visitant le sommet non visité le plus proche du dernier sommet visité.
   Solution. À vous de jouer.

2. Discutez empiriquement de la qualité de la solution sur les petites tailles puis des tailles maximales atteignables par cette heuristique.
   Solution. À vous de jouer.
3. Discutez, comme pour le bloc 2 l'an dernier et le cours de cette année, de la construction de pires cas pour cette heuristique.
   Solution. À vous de jouer.
4. Étudiez les qualités statistiques de cette solution.
   Solution. À vous de jouer.
5. Utilisez cette heuristique pour trouver un seuil dans `solve_by_truncated_heuristic_threshold`. Commentez les performances.
   Solution. À vous de jouer.

Question 4. *Heuristique par flips.*

Pour deux trajets entre quatre villes distinctes, il est parfois possible de faire un changement des routes parcourues qui n'implique que ces 4 villes et qui permet d'améliorer la tournée courante (cf. le cours de cette année).

1. Formalisez cette intuition de ce qu'on nommera un flip.
   Solution. À vous de jouer.
2. Implémentez la recherche d'un flip améliorant la solution courante.
   Solution. À vous de jouer.
3. Implémentez la fonction appliquant un flip à la solution courante (discuter du choix du meilleur ou du premier que vous avez sous la main).
   Solution. À vous de jouer.
4. Implémentez l'heuristique améliorant par flips, tant que c'est possible, une solution initiale tirée au hasard.
   Solution. À vous de jouer.
5. Est-ce que la suite de flips peut ne pas se terminer selon votre convention du cas d'égalité ?
   Solution. À vous de jouer.
6. Empiriquement, quelle est la qualité de votre solution ? Quel est le nombre de flips ?
   Solution. À vous de jouer.

```
#<TODO 3.4>
def distance_edge_vertex(p:'Point',e: ('Point','Point')) -> float :
    """ allongement du trajet u-v de l'arete e au trajet u-p-v."""
    (u,v) = e
    return distance(u,p)+distance(p,v)-distance(v,u)
#</TODO 3.4>


#<TODO 3.4>
def pair_of_closest_points(points:List[Point]) -> (int,int): # naive version
    """return the pair (i,j) of indices i < j at minimal distance in points"""
    assert(len(points) > 1)
    minimal_distance = None
    closest_pair = None
    for i in range(len(points)):
        for j in range(i+1,len(points)):
            d = distance(points[i],points[j])
            if minimal_distance == None or d < minimal_distance:
                minimal_distance = d
                closest_pair = (i,j)
    return closest_pair
#</TODO 3.4>

#<TODO 3.4>
def closest_edge_and_vertex(circuit:List[Point],points:List[Point]) -> (int, int):
    """return (i,j) where (circuit[i-1],circuit[i]) is an edge and points[j] an inserted vertex"""
    assert(len(circuit) > 1 and len(points) > 0)\
    # XXX look at similitude with pair_of_closest_points (factorize ?)
    minimal_distance = None
    closest_pair = None
    for i in range(len(circuit)):
        e = (circuit[i-1],circuit[i])
        for j in range(len(points)):
            d = distance_edge_vertex(points[j],e)
```

```
            if minimal_distance == None or d < minimal_distance:
                minimal_distance = d
                closest_pair = (i,j)
    return closest_pair
#</TODO 3.4>
```

Question 5. *Vers une 2-approximation.*

On se propose de travailler sur une heuristique vue en cours qui devrait s'avérer être une 2-approximation.

Commencer par la tournée partielle n'impliquant que les deux points les plus proches. Ensuite, insérez entre deux villes de la tournée la ville qui augmente le moins la longueur de la tournée.

Question 6. Partie optionnelle. *Metropolis-Hasting. Naviguer avec aussi des flips allongeant la tournée pour sortir des minimums locaux.*

C'est un algorithme apparu en 1953 dans une revue de Chimie pour y traiter un problème de cette spécialité.

Il a sa propre page wikipédia

`https://fr.wikipedia.org/wiki/Algorithme_de_Metropolis-Hastings`

et la version anglaise est plus détaillée.

Ici, on va adapter cet algorithme sans soigneusement en vérifier toutes les hypothèses qui donnent des garanties probabilistes sur son comportement (essentiellement la symétrie des mouvements). On va dire que l'énergie d'une solution est proportionnelle à la longueur de la tournée.

On choisit un flip au hasard parmi ceux qui sont possibles et :

- si l'énergie/longueur décroît avec le flip, appliquer le flip,
- si l'énergie/longueur croit de dE avec le flip, appliquer le flip avec probabilité $exp(-kTdE)$ où k est la constante de Boltzmann, T la température.

Dans les cas favorables, cet algorithme occupe chaque solution avec une probabilité proportionnelle à $exp(-kTE)$ ce qui correspond à la distribution boltzmanienne classique en physique statistique. En particulier, l'intérêt est que cela favorise les solutions les plus courtes à faible température (exploitation), et donne la même chance à chaque solution aux températures très élevées (exploration).

1. Notez qu'à la température nulle vous obtenez le premier algorithme sur les flips dans sa version aléatoire.
2. Notez qu'en principe à une température non nulle vous ne pouvez pas être bloqué dans un minimum local (en supposant que l'espace des tournées est connexe par flips).
3. Implémentez le pas de navigation par l'algorithme de Metropolis-Hasting.
   Solution. À vous de jouer.
4. Implémentez la navigation sur plusieurs pas et calibrez sa longueur en fonction du nombre de villes.
   Solution. À vous de jouer.
5. Discussion sur les liens avec l'IA (notamment recuit simulé des années 70, et plus récemment le fait qu'en grande dimension les minimums locaux ont finalement peu de chances d'exister en pratique).
   Solution. À vous de jouer.