ALGORITHMIQUE AVANCÉE

Voyageur de commerce : 1. méthodes exhaustives

On rappelle la définition du problème :

VOYAGEUR DE COMMERCE

Instance: Un ensemble $V = \{v_0, \ldots, v_{n-1}\}$ de points et une distance d sur V.
Question: Trouver une tournée de longueur minimum passant par tous les points de V, c'est-à-dire une permutation σ des indices des éléments de V telle que $\sum_{i=0}^{n-1} d(v_{\sigma(i)}, v_{\sigma(i+1 \bmod n)})$ est minimum.

Dans la suite on supposera que $V \subset \mathbb{R}^2$ est un ensemble de n points du plan et que d est la distance euclidienne entre deux points. L'objectif des TP va être de programmer et de tester les performances de plusieurs algorithmes sur certaines instances du problème.

1 En TP

Avant de commencer, veuillez visionner la vidéo des consignes qui se trouve ici

http://www.labri.fr/perso/hocquard/PresentationTSP.mp4

Les sources du projet sont disponibles ici

http://www.labri.fr/perso/borgne/DIU

2 Mise en bouche

Le point P de coordonnées (x, y) est représenté par l'objet p de type `Point` tel que `p.x` donne l'abscisse et `p.y` donne l'ordonnée.

Question 1. *Détermination de la distance euclidienne entre deux points. Savoir utiliser : import math et la classe* `Point` *pour accéder aux coordonnées.*

1. La fonction distance retourne 0 par défaut.

```
def distance(a:Point,b:Point)->float:
    return 0
```

   Discutez le typage optionnel en Python depuis la version 3.5.
2. Implémentez la distance euclidienne.
3. Constatez avec doctest que les tests unitaires passent (cf. le test *TEST 1.1*).

On représentera une permutation σ de $\{0, \ldots, n-1\}$ par un tableau `P` de taille n tel que `P[i]` $= \sigma(i)$. Par exemple, `P=[3,1,2,0]` représentera une permutation pour $n = 4$ qui inverse le premier élément avec le dernier, ce qui définit la tournée v_3, v_1, v_2, v_0, v_3. Dit autrement, `P` représente l'ordre de visite des points de V dans la tournée.

Question 2. *Détermination de la distance d'une tournée circulaire. Savoir écrire une boucle "circulaire" sur un tableau.*

1. Écrivez la fonction `total_length(cities: List[Point],permutation:Permutation,n:int)->float` qui renvoie la longueur de la tournée des `n` points de `V` selon la permutation `P`. Consultez sa spécification dans la documentation.
2. Vous pouvez sommairement tester votre solution avec le test *TEST 1.2*.

3 Approche « Brute-Force »

Les permutations peuvent être rangées par ordre lexicographique de la plus petite (dans notre exemple `P=[0,1,2,3]`) à la plus grande (`P=[3,2,1,0]`). On suppose donnée la fonction `next_permutation(p:Permutation)->bool` qui calcule, en mettant à jour `P`, la permutation suivant immédiatement `P` dans l'ordre lexicographique. De plus, la fonction renvoie `False` si et seulement si, à l'appel de la fonction, la permutation `P` correspond à la plus grande permutation. Vous pourrez utiliser, mais ce n'est pas nécessaire, la constante `math.inf` qui définit la plus grande valeur pouvant être représentée par une variable de type `float`.

Question 3. *Solution parcourant exhaustivement l'espace des solutions. Savoir écrire la recherche du min sur son domaine.*

1. Utilisez la documentation pour comprendre la spécification de la fonction.

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

2. Utilisez cette fonction pour écrire une recherche de solution optimale pour le voyageur de commerce.

   ```
   def solve_by_exhaustive_search(tsp: Tsp) -> (float,List[int]):
   ```

Observez que nous préférons transmettre la structure de données `tsp.Tsp` (similaire à un `__struct__` en C) qui contient :

- le nombre de villes dans `tsp.n:int`,
- les emplacements des villes dans `tsp.cities: List[Point]`,
- la tournée sélectionnée dans `tsp.permutation: Permutation`.

Cela limite le nombre d'arguments mais

```
def solve_by_exhaustive_search(n:int, cities:List[Point], permutation: List[int])-> float:
```

serait tout aussi acceptable. En effet, dans ce cas la tournée est stockée dans permutation et la valeur de retour est la longueur de la tournée optimale.

Nous avons préparé les tests de performance *TEST 1.3.a* et *TEST 1.3.b* pour la suite des questions de cet exercice.

3. Visualisez chaque record dans la recherche d'une solution avec 6 villes (`tsp.draw`). Vous pouvez contrôler l'affichage à l'aide de `tsp.skip_draw` mis à `False` pour afficher et à `True` pour ne pas afficher.
4. Jusqu'à combien de villes pouvez-vous traiter le problème ?
5. Quelle est la nature des limites de cette solution ? (temps ? espace ? lire les résultats des tests)
6. En particulier, où en est votre programme lorsque vous l'interrompez ? Étudiez notamment *TEST 1.3.c*.

Question 4. *Parcours en mémoire $O(n)$ de l'espace des solutions. Comprendre la navigation dans les permutations et adapter la recherche du min.*

1. Implémentez la fonction

   ```
   def visit_permutations_from_first_to_last(n:int)->None:
   ```

   en utilisant

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

   et

   ```
   def next_permutation(List[int])-> bool:
   ```

2. Vérifiez avec le test *TEST 1.4.b* que la mémoire utilisée reste faible comparée à `list_permutations_via_inversion_tables`.
3. Implémentez la fonction

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

   qui va permettre de vérifier par le test *TEST 1.4.a* que votre parcours est bien celui des mêmes permutations que celle de `list_permutations_via_inversion_tables`.
4. Implémentez dans

   ```
   def solve_via_exhaustive_search_from_first_to_last(tsp:Tsp)->(float,Permutation):
   ```

   une seconde résolution du tsp en copiant le code de la fonction `solve_via_exhaustive_search` et en remplaçant la probable boucle `_for_` par votre nouvelle technique de parcours des permutations.

5. Vérifiez par le test *TEST 1.4.c* que les deux résolutions coïncident pour la longueur de la tournée optimale.
6. Expliquez pourquoi les solutions peuvent ne pas toujours être les mêmes.
7. Comprenez-vous en quoi en général elles diffèrent ?
8. Pourquoi certains cas empêchent d'enlever le "en général" de la phrase précédente ?
9. Utilisez le test *TEST 1.4.d* pour vérifier les performances de cette nouvelle méthode de résolution.
10. En particulier, observez jusqu'à combien de villes vous pouvez résoudre le problème en au plus quelques secondes.

Nous déclarons cette solution suffisamment efficace pour générer des tests pertinents pour vérifier nos prochaines solutions.

11. Observez les fonctions les plus coûteuses via le test *TEST 1.4.e*. Vous pourrez éventuellement comparer avec le test *TEST 1.3.b*.

Question 5. *Détection d'un préfixe trop mauvais de solutions. Préparation de raccourcis ...*

Une optimisation consiste à arrêter l'évaluation pendant le calcul de `total_length` dès que la longueur courante dépasse celle de la meilleure tournée déjà obtenue, disons w. Dans cette optimisation, n'oubliez pas le retour au point de départ, c'est-à-dire qu'une tournée partielle utilisant les $i+1$ premiers points $v_{\sigma(0)}, \ldots, v_{\sigma(i)}$ possédera $i+1$ arêtes et aura pour longueur $w_i = \left(\sum_{j=0}^{i-1} d(v_{\sigma(j)}, v_{\sigma(j+1)})\right) + d(v_{\sigma(i)}, v_{\sigma(0)})$. Dans la question suivante on observera que si la longueur de la tournée partielle $w_i \geqslant w$, alors on peut directement passer à la plus grande permutation de préfixe $\sigma(0), \ldots, \sigma(i)$.

1. Implémentez la fonction

   ```
   def total_length_stopped(n:int,cities:List[Point],permutation:List[int],threshold:float)-> (float,int):
   ```

   qui renvoie la longueur de la tournée (partielle si elle est suffisamment mauvaise pour dépasser le seuil (=threshold) donné).
2. Testez sommairement sur les exemples de la docstring (test *TEST 1.5.a*).
3. Remplacez par cette nouvelle fonction la fonction `total_length` dans une nouvelle version

   ```
   def solve_by_exhaustive_search_from_first_to_last_stopped(tsp:Tsp)->(float,Permutation):
   ```

   de `solve_by_exhaustive_search_from_first_to_last`.
4. Testez la validité avec le test *TEST 1.5.b* puis observez la dégradation des performances sur le test *TEST 1.5.c*.

Question 6. *Utilisation de raccourcis dans le parcours des permutations.*

1. Comprendre la spécification et l'utilité algorithmique de la fonction

   ```
   def last_permutation_with_fixed_prefix(p:List[int],k:int)->None:
   ```

   qui permettra de prendre des raccourcis dans le parcours des permutations.
2. Si nécessaire, implémentez la fonction

   ```
   def sort_subpermutation(p:Permutation,i:int,j:int)->None:
   ```

   qui trie en ordre décroissant la sous-partie `p[i:j]` de la permutation `p` comprise entre les indices `i` et `j-1`.
3. Implémentez la fonction

   ```
   def swap_interval(p:Permutation,i:int,j:int)->None:
   ```

   qui se contente d'inverser l'ordre des élements dans la sous-partie `p[i:j]` de la permutation `p` comprise entre les indices `i` et `j-1`.
4. Utilisez une des fonctions précédentes pour implémenter `last_permutation_with_fixed_prefix`.
5. Utilisez la fonction précédente pour parcourir seulement une partie des permutations tout en garantissant de visiter une solution optimale dans votre implémentation de la fonction

   ```
   def solve_by_exhaustive_search_with_short_cuts_1(tsp:Tsp)->(float,Permutation):
   ```

6. Validez avec le test *TEST 1.6.a* puis observez comment les performances s'améliorent sur le test *TEST 1.6.b*.

Question 7. Partie optionnelle. *Encore plus de raccourcis...*

1. Pourquoi dans une tournée partielle peut-on ajouter au circuit une ville de plus pour détecter le dépassement du seuil ?

2. Implémentez une seconde version que l'on peut interrompre du calcul de la longueur de la tournée en intégrant au circuit cette ville supplémentaire.

   ```
   def total_length_stopped2(n:int,cities:List[Point],permutation:List[int],threshold:float) -> (float,int):
   ```

   Cela devrait permettre la détection de plus de raccourcis dans

   ```
   def solve_by_exhaustive_search_with_short_cuts_2(tsp:Tsp)->(float,Permutation):
   ```

3. Validez par *TEST 1.7.a* et observez les performances *TEST 1.7.b*.
4. (Optionnel) Est-il efficace de chercher puis ajouter la ville supplémentaire qui allonge le plus la tournée partielle ? Si vous implémentez une troisième version `total_length_stopped3` adaptez les tests *TEST 1.7.a* et *TEST 1.7.b* pour étudier le compromis que vous avez anticipé.

Question 8. Partie Optionnelle. *Toujours plus de raccourcis avec une équivalence de solutions.*

On va maintenant optimiser la procédure précédente. Une optimisation consiste à fixer l'un des points de la permutation, le premier ou le dernier (à voir ce qui est le plus pratique). Cela permet de gagner un facteur n sur le nombre de permutations à tester. On peut aussi fixer le sens de parcours de la tournée ce qui fait gagner un facteur 2.

1. Pourquoi est-il possible de supposer que, dans la permutation p solution, $p[1] = 1$ et $p[0] < p[2]$?
2. Implémentez

   ```
   def next_constrained_permutation(p:Permutation)->bool:
   ```

   et utilisez-la dans une nouvelle version de l'exploration exhaustive.

   ```
   def solve_by_exhaustive_search_with_short_cuts_3(tsp:Tsp)->(float,Permutation):
   ```

3. Validez par *TEST 1.8.a* et observez les performances *TEST 1.8.b*.
4. Comparez les résultats de *TEST 1.4.d* et *TEST 1.8.c* pour estimer le fruit de nos efforts.