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*).

SOLUTION.

```
def distance(p:Point,q:Point)->float: # XXX Point instead of 'Point' created a problem in first versions
    """distance euclidienne entre self et l'autre point p.

    >>> from point import *
    >>> q = Point(0,0); p = Point(-3,4); distance(q,p)
    5.0

    >>> assert(distance(q,p) == distance(p,q))
    >>> r = Point(3,4); distance(r,q)
    5.0

    >>> s = Point(1,1); distance(s,q)
    1.4142135623730951
```

```
    >>> assert(distance(s,q) == math.sqrt(2))

    """
    #<TODO 1.1>
    dx = p.x-q.x
    dy = p.y-q.y
    return math.sqrt(dx**2+dy**2)
    #</TODO 1.1>
```

On représentera une permutation σ de $\{0,\dots,n-1\}$ 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.
   SOLUTION.
   La difficulté ici est de comprendre que les villes sont dans un tableau List[Point] que vous avez en paramètre de la fonction. On ne change pas l'ordre des villes, on utilise permutation comme un ordre sur les indices dans le tableau.

   ```
   #<TODO 1.2>
   def total_length(n:int, cities : List[Point], permutation: Permutation) -> float:
       """ sum distances along the circular travel permutation between the n cities.

       >>> from point import *; from tsp import *;
       >>> cities = [Point(0,0),Point(1,0),Point(1,1),Point(0,1)];
       >>> total_length(len(cities),cities,list(range(len(cities))))
       4.0

       """
       found_length = 0
       for i in range(n):
           found_length += distance(cities[permutation[i-1]],cities[permutation[i]])
       return found_length
   #</TODO 1.2>
   ```

   À noter que permutation[-1] renvoie la dernière permutation de la tournée.
2. Vous pouvez sommairement tester votre solution avec le test *TEST 1.2*.
   SOLUTION. À vous de jouer.

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]:
   ```

   Vous pouvez remarquer que `list_permutations_via_inversion_tables(3)` renvoie $[[0,1,2],[1,0,2],[0,2,1],[1,2,0],[2,0,1],[2,1,0]]$.

   Cette fonction n'est pas facile à coder, c'est pour cette raison que nous l'avons codé pour vous, ce qui nous intéresse ici c'est ce qu'elle renvoie à savoir toutes les permutations de n éléments (dans notre exemple $n = 3$).
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.

SOLUTION.

```
#<TODO 1.3>
def solve_by_exhaustive_search(tsp:Tsp)->(float,Permutation):
    """ solve tsp by first listing all solutions then finding one of the minimal length. """
    smallest_length = None
    best_permutation = None
    t = 0
    for p in list_permutations_via_inversion_tables(tsp.n):
        t += 1
        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,math.factorial(tsp.n)))
            tsp.permutation = best_permutation
            tsp.draw()
    return (smallest_length,tsp.permutation)
#</TODO 1.3>
```

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 ?
   SOLUTION. À vous de jouer.
5. Quelle est la nature des limites de cette solution ? (temps ? espace ? lire les résultats des tests)
   SOLUTION. À vous de jouer.
6. En particulier, où en est votre programme lorsque vous l'interrompez ? Étudiez notamment *TEST 1.3.c*.
   SOLUTION. À vous de jouer.

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:
   ```

   La fonction `first_permutation(n)` renvoie une permutation de taille n. Par exemple, `first_permutation(3)` renvoie $[0, 1, 2]$. Maintenant, si on écrit

   ```
   p=first_permutation(3);
   While next_permutation(p):
           print(p)
   ```

on obtiendra la séquence

```
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

On peut remarquer qu'on est partis des valeurs triées par ordre croissant sur la première permutation ([0,1,2]) pour finir par la dernière permutation avec des valeurs triées par ordre décroissant ([2,1,0]). On a visité toutes les permutations possibles grâce à `next_permutation` qui utilise un ordre lexicographique. L'intérêt de cette fonction est qu'on utilise un seul tableau de taille n et qu'on a bien visité les $n!$ tournées. Ceci permet de limiter la saturation de la mémoire.

SOLUTION.

```
#<TODO 1.4>
def visit_permutations_from_first_to_last(n:int)->None:
    """ visit all permutations from first to last (unused except for testing).

    >>> from permutation import *; visit_permutations_from_first_to_last(3)
    [0, 1, 2]
    [0, 2, 1]
    [1, 0, 2]
    [1, 2, 0]
    [2, 0, 1]
    [2, 1, 0]

    """
    p = first_permutation(n)
    while True:
        #print(p)
        if not next_permutation(p):
            break
#</TODO 1.4>
```

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

   SOLUTION. À vous de jouer.

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`.

   SOLUTION.

   ```
   #<TODO 1.4>
   def list_permutations_from_first_to_last(n:int)->List[Permutation]:
       """ list all permutations from first to last (unused except for testing).

       >>> from permutation import *; list_permutations_from_first_to_last(3)

       """
       p = first_permutation(n)
       permutations = []
       while True:
           permutations += [copy.deepcopy(p)]
           if not next_permutation(p):
               break
       return permutations
   #</TODO 1.4>
   ```

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.

   SOLUTION.

   ```
   #<TODO 1.4>
   def solve_by_exhaustive_search_from_first_to_last(tsp:Tsp)->(float,Permutation):
   ```

```
        smallest_length = None
        best_permutation = None
        t = 0
        p = first_permutation(tsp.n)
        while True:
            t += 1
            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,math.factorial(tsp.n)))
                tsp.permutation = best_permutation
                tsp.draw()
            if not next_permutation(p):
                break
        return (smallest_length,tsp.permutation)
    #</TODO 1.4>
```

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

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*.
    SOLUTION. À vous de jouer.

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 $\sigma(0), \ldots, \sigma(i)$.
Concrètement, dès que nous avons repéré un mauvais préfixe (i.e. le début de la séquence d'une tournée qui ne va pas être optimale) on utilise `next_permutation()` qui va directement modifier le préfixe que l'on vient d'identifier comme étant « mauvais ». On économise ainsi de nombreux appels à `total_length()` inutiles.

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).
   SOLUTION.

   ```
   #<TODO 1.5>
   def total_length_stopped(n:int, cities : List[Point], permutation:Permutation, threshold:float) -> (float,int):
       """ sum length between cities along permutation while sum remains below threshold:
       return the sum of length and the number of visited cities.

       >>> from tsp import *; cities = [Point(0,0),Point(1,0),Point(1,1),Point(0,1)];
   ```

```
    >>> permutation = first_permutation(4);
    >>> total_length_stopped(4,cities,permutation,5)
    (4.0, 4)

    >>> total_length_stopped(4,cities,permutation,2.5)
    (3.414213562373095, 3)

    """
    found_length = 0
    for i in range(1,n):
        found_length += distance(cities[permutation[i-1]],cities[permutation[i]])
        incremented_length = found_length+ distance(cities[permutation[i]],cities[permutation[0]])
        if incremented_length > threshold:
            return (incremented_length,i+1)
    found_length += distance(cities[permutation[n-1]],cities[permutation[0]])
    return (found_length,n)
#</TODO 1.5>
```

2. Testez sommairement sur les exemples de la docstring (test *TEST 1.5.a*).
   SOLUTION. À vous de jouer.
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`.
   SOLUTION.

   ```
   #<TODO 1.5>
   def solve_by_exhaustive_search_from_first_to_last_stopped(tsp:Tsp)->(float,Permutation):
       smallest_length = math.inf
       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 and 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()
           if not next_permutation(p):
               break
       return (smallest_length,tsp.permutation)
   #</TODO 1.5>
   ```

4. Testez la validité avec le test *TEST 1.5.b* puis observez la dégradation des performances sur le test *TEST 1.5.c*.
   SOLUTION. À vous de jouer.

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.

   ```
   def last_permutation_with_fixed_prefix(p:Permutation,k:int)->None:
       """ find maximal permutation for fixed first k values."""
       for j in range(k+1,len(p)):
           imax = p[j:].index(max(p[j:]))
           swap(p,j,j+imax)
   ```

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`.
   SOLUTION. À vous de jouer.

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`.

Solution.

```
def swap_interval(p:Permutation,i:int,j:int)->None:
    """ reverse the entries of array p in the interval p[i:j]."""
    if i not in range(len(p)) or j not in range(len(p)+1) or j < i:
        return
    for k in range((j-i+1)//2):
        swap(p,i+k,j-1-k)
```

4. Utilisez une des fonctions précédentes pour implémenter `last_permutation_with_fixed_prefix`.

Solution.

```
def last_permutation_with_fixed_prefix(p:Permutation,k:int)->None:
    """ find maximal permutation for fixed first k values."""
    for j in range(k+1,len(p)):
        imax = p[j:].index(max(p[j:]))
        swap(p,j,j+imax)
```

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):
```

Solution.

```
#<TODO 1.6>
def solve_by_exhaustive_search_with_short_cuts_1(tsp:Tsp)->(float,Permutation):
    smallest_length = math.inf
    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) # partial_length may be decremented by 2
        if not next_permutation(p):
            break
    return (smallest_length,tsp.permutation)
#</TODO 1.6>
```

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

Solution. À vous de jouer.

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 ?

Solution.

Lorsqu'on rajoute une ville au circuit pour détecter le dépassement de seuil cela n'a pas d'incidence à cause de l'inégalité triangulaire.

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):
```

Solution.

```
#<TODO 1.7>
def total_length_stopped2(n:int, cities : List[Point], permutation:Permutation, record:float) -> (float,int):
    found_length = 0
    for i in range(1,n):
        found_length += distance(cities[permutation[i-1]],cities[permutation[i]])
        incremented_length = found_length + distance(cities[permutation[i]],cities[permutation[n-1]])\
          + distance(cities[permutation[n-1]],cities[permutation[0]])
        if  incremented_length > record:
            return (found_length,i+1)
    found_length += distance(cities[permutation[n-1]],cities[permutation[0]])
    return (found_length,n)

def solve_by_exhaustive_search_with_short_cuts_2(tsp:Tsp)->(float,Permutation):
    smallest_length = math.inf
    best_permutation = None
    t = 0
    p = first_permutation(tsp.n)
    while True:
        t += 1
        (its_length,partial_length) = total_length_stopped2(tsp.n,tsp.cities,p,smallest_length) # single change
        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 1.7>
```

3. Validez par *TEST 1.7.a* et observez les performances *TEST 1.7.b*.
   Solution. À vous de jouer.
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.
   Solution. À vous de jouer.

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 $p[0] < p[2]$?
   Solution.
   On cherche ici à limiter le nombre d'explorations. Pour cela on va s'intéresser au préfixe $p_0p_1p_2$. Ainsi, en choisissant $p[1] = 1$, on fixe la deuxième ville qui va être visitée dans la tournée, la deuxième visitée est donc la ville 1. En rajoutant la deuxième condition $p[0] < p[2]$, on pourra déclarer un préfixe « mauvais » dès lors qu'il ne satisfait pas ces deux conditions.Ceci va permettre de gagner en vitesse d'exécution de l'algorithme pour visiter les $n - 3$ villes restantes. On gagne ainsi un facteur $2n$.
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):
   ```

   Solution.

```
#<TODO 1.8>
def next_constrained_permutation(p:Permutation)->bool:
    while True:
        if not next_permutation(p):
            return False
        if p[1] != 1:
            last_permutation_with_fixed_prefix(p,1)
        elif p[0] > p[2]:
            last_permutation_with_fixed_prefix(p,2)
        else:
            return True

def solve_by_exhaustive_search_with_short_cuts_3(tsp:Tsp)->(float,Permutation):
    smallest_length = math.inf
    best_permutation = None
    t = 0
    p = first_permutation(tsp.n)
    while True:
        assert( p[1] == 1 and p[0] < p[2]) # ensured by next_constrained_permutation
        t += 1
        (its_length,partial_length) = total_length_stopped2(tsp.n,tsp.cities,p,smallest_length) # single change
        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-1) # le -2 ne passe plus, pourquoi ?
        if not next_constrained_permutation(p): # single change
            break
    return (smallest_length,tsp.permutation)
#</TODO 1.8>
```

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