

TP TSP DP LS

AAIA: Traveling Salesman Problem (TSP) resolution with Dynamic Programming (DP)

Florian Rascoussier, Aurélien Delage, Christine Solnon

Mars, avril 2025

Note: Previously 2 subjects in 2x2 hours in class, now a single TP session.

Modifications du sujet sujet-TSP-DP-2022.pdf

- Modifier énoncé question 8 (qui est beaucoup trop compliquée à l'heure actuelle)
 - Ajouter -lm dans la commande au début du sujet.
-

0. Introduction

Ce TP est l'occasion de se plonger dans un problème combinatoire classique et de découvrir du même coup une technique de programmation très utile.

Le Problème du Voyageur de Commerce (TSP)

Le problème du TSP (Traveling Salesman Problem) est un problème d'optimisation classique où un voyageur doit parcourir un ensemble de

villes exactement une fois, puis revenir à son point de départ. L'objectif est de déterminer l'ordre de visite qui minimise la distance totale parcourue. Malgré son énoncé simple, le TSP est réputé difficile (NP-difficile) et fait l'objet de nombreuses recherches en algorithmique et en heuristiques.

La Programmation Dynamique (DP)

La programmation dynamique (PD) est une méthode exacte couramment utilisée pour résoudre des instances de taille modérée de problèmes combinatoires, comme le TSP. Son principe général consiste à décomposer le problème initial en de multiples sous-problèmes plus simples, puis à mémoriser (ou « mémoïser ») leurs solutions pour éviter les recalculs inutiles. Grâce à cette approche, on parvient à réduire considérablement la redondance des calculs et à obtenir une solution optimale. Toutefois, la complexité en temps et en mémoire tend à croître rapidement à mesure que la taille du problème augmente.

Pour illustrer cette méthode, vous pouvez visionner une courte présentation en anglais : [How Dynamic Programming Broke Software Engineers](#).

En pratique, la programmation dynamique se déroule en trois étapes principales :

- 1. Résolution itérative ou récursive** : le problème est abordé étape par étape ou de manière récursive.
- 2. Sauvegarde des résultats intermédiaires** : on enregistre ces résultats partiels pour éviter de les recalculer.
- 3. Réutilisation** : on s'appuie sur ces résultats pour construire la solution finale, sans relancer les mêmes calculs.

On distingue généralement deux approches de la programmation dynamique :

1. **Top-Down (récursion et mémoïsation)** : on part d'un appel récursif du problème principal, et dès qu'un sous-problème se répète, on en réutilise le résultat précédemment mémorisé.
2. **Bottom-Up (tabulation)** : on calcule systématiquement tous les sous-problèmes, en commençant par les plus simples pour remonter progressivement vers la solution finale. Durant ce processus, on remplit une « table » où chaque entrée correspond à un sous-problème intermédiaire, facilitant ainsi le passage à des niveaux de complexité supérieurs.

1. Questions

Les assertions suivantes sont-elles vraies ou fausses ? (voir les diapos 112 et 113 du cours)

Question 1.1

Un problème appartient à la classe NP s'il n'est pas possible de le résoudre en temps polynomial.

Réponse : Faux:

- **Définition de NP.** Un problème appartient à NP si on peut vérifier *en temps polynomial* la validité d'une solution candidate (ou, de manière équivalente, si une machine de Turing *non déterministe* peut le résoudre en temps polynomial).
- **Confusion courante.** Dire qu'« il n'existe pas d'algorithme polynomial » pour le problème ne correspond pas à la définition de NP, mais plutôt à l'un des scénarios de la question ouverte « P vs NP ».
- **P vs NP.** Nous ne savons pas encore si un *algorithme polynomial* (en machine déterministe) existe pour résoudre **tous** les problèmes de NP. Affirmer qu'aucune solution polynomialement efficace n'existe reviendrait à prouver que $P \neq NP$, ce qui est un problème non résolu en informatique théorique.

Question 1.2

Un problème appartient à la classe NP s'il existe un algorithme permettant de le résoudre en temps polynomial sur une machine de Turing non déterministe.

Réponse : Vrai.

- **Machine de Turing non déterministe.** C'est la caractérisation la plus couramment utilisée pour définir NP : un problème est dans NP s'il peut être résolu en temps polynomial par un dispositif de calcul non déterministe.
- **Vérification polynomiale.** L'équivalent déterministe, plus intuitif, est la capacité de vérifier rapidement une solution proposée.

Question 1.3

Les problèmes NP-complets sont les problèmes les plus difficiles de la classe NP.

Réponse : Vrai (avec nuances).

- **NP-complétude.** Un problème est dit NP-complet s'il est à la fois dans NP et NP-difficile, c'est-à-dire qu'il « domine » (via des réductions polynomiales) tous les autres problèmes de NP.
- **“Les plus difficiles.”** Cette expression signifie que si un problème NP-complet admettait un algorithme déterministe en temps polynomial, alors **tous** les problèmes de NP pourraient également être résolus en temps polynomial, ce qui impliquerait que $P = NP$.
- **Statut ouvert.** Personne n'a encore prouvé ou réfuté l'existence d'un tel algorithme polynomial. Les problèmes NP-complets sont « considérés » comme les plus durs de NP tant que l'on n'a pas établi la preuve que $P \neq NP$.

Question 1.4

Tout problème NP-complet appartient à la classe NP.

Réponse : Vrai.

- **Par Définition.** Pour être NP-complet, il faut d'abord être dans NP. Ensuite, on montre que ce problème est au moins aussi difficile que n'importe quel autre problème de NP (NP-hard).

Tout problème NP-difficile appartient à la classe NP.

Réponse : Faux.

- **NP-difficile.** Dire qu'un problème est NP-difficile signifie qu'il est *au moins* aussi dur que les problèmes de NP. Il peut donc se trouver en dehors de NP (cas de certains problèmes indécidables, par exemple).
- **Comparaison.** Les problèmes NP-complets sont à la fois NP-difficiles et *dans* NP. Mais un problème NP-difficile n'est pas nécessairement dans NP (il peut être « encore plus compliqué » ou non décidable en temps fini).

En résumé, il est essentiel de ne pas confondre « être dans NP » avec « ne pas pouvoir se résoudre en temps polynomial ». Tant que la question « P = NP ? » n'est pas tranchée, nous ne pouvons pas affirmer de manière définitive qu'il n'existe pas de solution déterministe en temps polynomial pour les problèmes NP-complets.

En pratique, si l'on sait qu'un problème est NP-difficile comme c'est le cas pour le TSP, on considère qu'il est peu envisageable de le résoudre efficacement avec un simple algorithme déterministe. Dès lors que la taille du problème devient importante, on recourt généralement à des techniques avancées (exactes, approchées ou hybrides) ou à des heuristiques afin d'obtenir une solution de bonne qualité dans un temps et avec une mémoire raisonnables.

2. Algorithme de Held et Karp

L'algorithme de Held & Karp est une méthode classique et exacte pour résoudre le problème du voyageur de commerce (TSP) à l'aide de la programmation dynamique. Son idée de base consiste à stocker, pour chaque sous-ensemble de villes et pour chaque ville de ce sous-ensemble, le coût minimal d'un chemin reliant ces villes et se terminant dans la ville en question. Cet algorithme procède donc en deux grandes étapes :

1. **Énumération des sous-ensembles** : on considère tous les sous-ensembles possibles de villes, puis on construit la solution optimale pour chacun de ces sous-ensembles en s'appuyant sur les solutions des sous-problèmes plus petits (sous-ensembles de taille inférieure).
2. **Combinaison des solutions partielles** : à partir de ces résultats mémorisés, on détermine progressivement la meilleure façon de boucler le circuit, de sorte à visiter l'ensemble des villes une seule fois avant de revenir au point de départ.

Grâce à la mémorisation systématique des solutions intermédiaires, on évite de recalculer plusieurs fois les mêmes quantités. Held & Karp parviennent ainsi à trouver une solution optimale, mais leur approche présente une complexité en temps de l'ordre de $O(n^2 \cdot 2^n)$, classique des algos de DP. Elle reste donc applicable en pratique surtout pour des instances de taille modérée, avant que l'**explosion combinatoire** ne la rende trop coûteuse en temps et en mémoire.

Notez que traditionnellement, l'algorithme de Held & Karp pour le TSP est présenté sous une forme **Bottom-Up**, car on va remplir de manière itérative une table pour tous les sous-ensembles de villes de taille croissante. Autrement dit, on part des sous-problèmes les plus petits (sous-ensembles réduits) et on “remonte” progressivement vers la solution globale.

Question 2.1

Combien y-a-t-il de sous-problèmes différents possibles dans le cas d'un graphe comportant n sommets ?

Introduction

Soit $D(i, E)$ un sous-problème. Une chose à bien voir en regardant la définition d'un sous problème mémorisé D est qu'on ne mémorise pas une liste (ensemble ordonée) de sommets précédemment visités, mais bien un ensemble (non-ordonné), correspondant au meilleur sous problème rencontré à cette étape. Le fait de perdre l'ordre des chemins permet en pratique de mémoriser seulement ce qui nous intéresse pour calculer les sous-problèmes suivants, c'est-à-dire la valeur du meilleur sous-chemin actuel. A cause de cela, et pour se permettre d'écrire les ensembles non ordonnés d'une façon similaire, on écrira toujours un ensemble non ordonné E sous **forme canonique**, c'est à dire en écrivant ses éléments dans l'ordre lexicographique. Exemple, si E contient les sommets 1, 3, 2, 8, 5, on écrira $E = \{1, 2, 3, 5, 8\}$.

Exemple simple illustré

Pour bien comprendre combien de sous-problèmes on a, rien de tel qu'un petit exemple pour se donner l'idée de la preuve: on considère un graphe complet $G = (V, A)$ avec un ensemble de sommets (*vertices*) $V = \{0, 1, 2, 3\}$, tel que 0 le sommet *origin* et *destination*. Ce graphe est généralement appelé customer-based graph.

On a dans cet exemple:

- customer-based graph: $G = (V, A)$
- Ensemble de tous les sommets: $V = \{0, 1, 2, 3\}, \#V = n = 4$
- Ensemble de clients: $C = V \setminus \{0\}, \#C = n - 1$
- Ensemble des arcs, complet: $A = V \times V$
- power set (ensemble des sous-ensembles) des sommets: 2^V
- power set des clients: 2^C

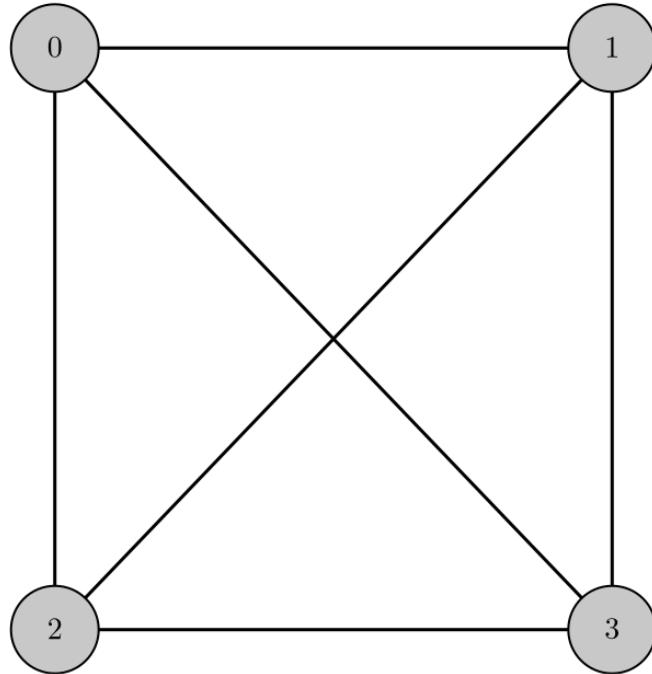
On note:

- Un sous-problème: $D(i, E) \mid i \in V, E \in 2^V$. Ce sous-problème a pour solution la longueur du plus court chemin allant de l'*origine* 0 jusqu'au

dernier sommet visité i en passant par chaque sommet de E exactement une fois ([elementarity constraint](#), [Hamiltonian path](#)).

Notez que ce sous-problème abstrait l'ordre de visite des sommets, il ne retient pas l'ordre de visite.

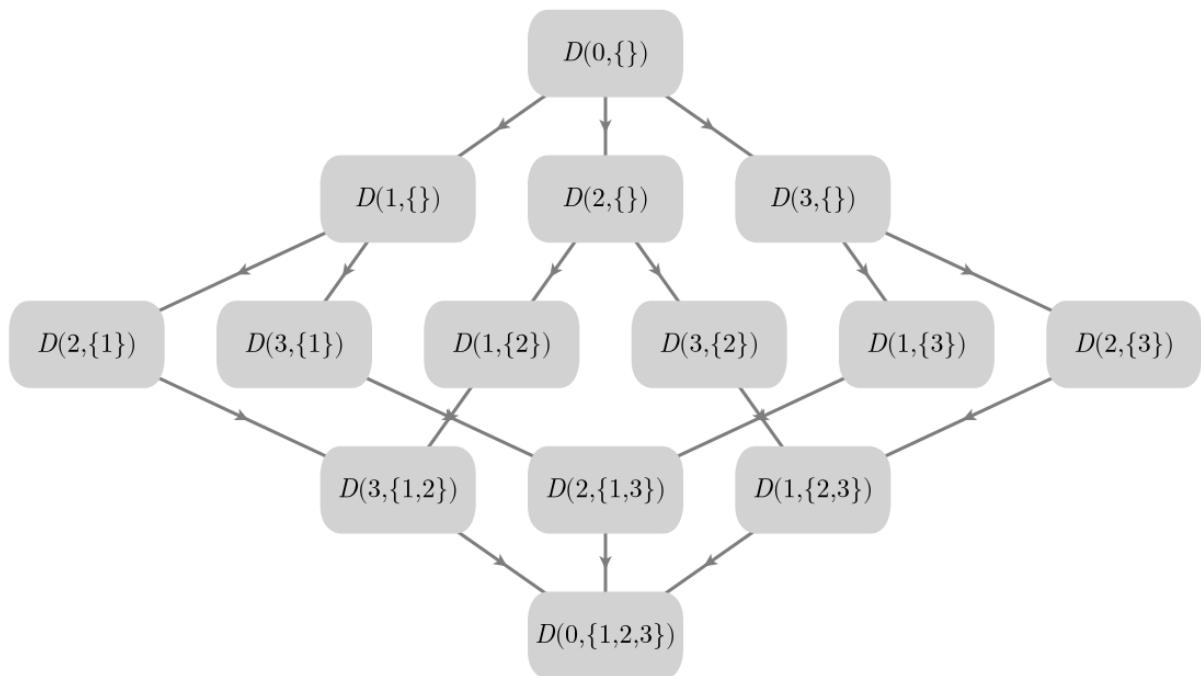
Customer-based complete graph.



Commençons avec une approche **Bottom-up**. Le premier sous-problème correspond au moment où l'on part du dépôt 0: $D(0, \{\})$ et que l'on a encore visité aucun client. Ensuite on considère les sous-problème correspondant au fait de visiter 1 client. Comme il y a 3 clients, on a donc 3 de ces sous problèmes: $D(1, \{\})$, $D(2, \{\})$, $D(3, \{\})$. Ensuite, et ça se complique, on considère chacun de ces sous-problème pour bâtir les sous-problèmes correspondant à ceux ayant visités 2 clients. Il y a $3 - 1 = 2$ clients possible à visiter pour chaque client soit $2 \times 3 = 6$ sous-problèmes correspondant à 2 visites. Par exemple,
 $D(1, \{\}) \rightarrow D(2, \{1\}), D(3, \{1\})$. De même, pour 3 visited, on à $3 - 2 = 1$ possibilités seulement donc 3 au total. Par exemple,
 $D(2, \{1\}), D(1, \{2\}) \rightarrow D(3, \{1, 2\})$. Enfin, on retourne au dépôt pour avoir

un circuit hamiltonien: c'est le dernier sous-problème (ou *destination*) qui visite tous les clients en partant et revenant au dépôt $D(0, \{1, 2, 3\})$. Avec un schéma, cela donne:

Classic DP state transition graph for TSP.



Idée de la preuve

Rappel: Le nombre de façons de choisir un échantillon de k éléments à partir d'un ensemble de n objets distincts où l'ordre n'a pas d'importance et où les remplacements ne sont pas autorisés est donné par la *combinaison*:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Le nombre de sous-problème à une étape donnée est donné par le nombre de valeurs possibles de E multiplié par le nombre de i possibles à l'étape considérée. Ce *nombre de valeurs possible* est indiqué par $\#$ (cardinalité).

Pour récapituler, on a donc 3 niveaux de sous-problèmes, plus 1 niveau *origine* et un niveau *destination* dans le graphe état transition, tel quel:

1. 1er niveau: 3 choix possibles pour i , et $E = \emptyset$ donc 1 seul choix pour E . Soit au total $3 \times 1 = 3$ sous-problèmes.
2. 2ème niveau: 2 choix possibles pour i . A ce niveau, 1 nœud a déjà été visité, c'est-à-dire que $\#E = 1$, donc $\binom{3}{1} = 3$ valeurs possibles pour $E \in \{\{0\}, \{1\}, \{2\}\}$. Soit $2 \times 3 = 6$ sous-problèmes.
3. 3ème niveau: 1 choix obligatoire pour i , $\#E = 2$, $\#E = \binom{3}{2} = 3$. Soit $1 \times 3 = 6$ sous-problèmes.

On a donc: $\sum_{\forall Niv} (\#i_{Niv} \times \#E_{Niv}) = (3 \times 1) + (2 \times 3) + (1 \times 3) = 12$ sous-problèmes différents. Auquel on peut ajouter le sous-problème *origine* $D(0, \{\})$ et le sous-problème *destination* $D(0, \{1, 2, 3\})$. Ces deux sous-problèmes sont toujours présents et donc sous-entendu (on ne les compte pas).

Preuve formelle

Combien de sous-problèmes différents pour un graphe à n sommets ?

Cette démon personnelle est inspiré de [Wikipedia](#), mais pour le nombre de sous-problème qui diffère légèrement du calcul de complexité.

Soit $G = (\mathcal{V}, \mathcal{A})$ un graphe **complet non orienté** basé sur les clients, avec $\#\mathcal{V} = n$ sommets, $\mathcal{V} = \{0, 1, \dots, n - 1\}$. On note l'ensemble des clients par $\mathcal{C} = \mathcal{V} \setminus \{0\} = \{1, \dots, n - 1\}$.

L'algorithme de [Held-Karp](#) consiste à calculer pour chaque **sous-ensemble de clients** $S \subseteq \mathcal{C}$, et pour chaque **dernier sommet visité** $i \notin S$, le plus court chemin allant de l'origine 0 à i en passant par tous les sommets de S , en respectant la contrainte d'[élémentarité](#) (i.e., chaque sommet visité exactement une fois). On note la distance de ce chemin par $D(S, i)$, et chaque paire (S, i) définit un **sous-problème**.

L'algorithme commence par calculer les valeurs de D pour les plus petits ensembles S , et termine avec les plus grands.

Nombre de sous-problèmes à un niveau donné

Calculer une valeur $D(i, S)$ pour un sous-ensemble $S \subseteq \mathcal{C}$ de taille $\#S = k$ nécessite de choisir le **meilleur des k chemins possibles**, chacun obtenu à partir d'une valeur déjà connue $D(j, S \setminus \{j\})$ et d'un arc (j, i) . Chaque évaluation est donc en $\mathcal{O}(k)$.

- Il y a $\binom{n-1}{k}$ sous-ensembles S de taille k ,
- Pour chaque S , il y a $n - 1 - k$ sommets $i \notin S$ possibles.

Le nombre de sous-problèmes à ce niveau est donc :

$$\#D_k = \binom{n-1}{k} \cdot (n-1-k)$$

En sommant pour tous les niveaux $k \in [0, n-2]$, on obtient le nombre de sous problèmes (sauf celui *origine* et *destination* du graphe d'états-transitions) :

$$\#D = \sum_{k=0}^{n-2} \binom{n-1}{k} \cdot (n-1-k)$$

Notez que le sous problème *destination* correspond à celui pour lequel $k = n-1$ (voir schéma).

Partie mathématique – changement de variable

Posons $n' = n - 1$. On a alors :

$$\#D = \sum_{k=0}^{n'-1} \binom{n'}{k} \cdot (n' - k) + 0$$

En remplaçant 0 par une expression bien choisie:

$$\#D = \sum_{k=0}^{n'-1} \binom{n'}{k} \times (n' - k) + \binom{n'}{n'} \times (n' - n')$$

C'est-à-dire:

$$\#D = \sum_{k=0}^{n'} \binom{n'}{k} \times (n' - k)$$

On décompose la somme :

$$\#D = n' \sum_{k=0}^{n'} \binom{n'}{k} - \sum_{k=0}^{n'} \binom{n'}{k} \cdot k$$

On utilise les identités suivantes :

- $\sum_{k=0}^n \binom{n}{k} = 2^n$
- $\sum_{k=0}^n \binom{n}{k} \cdot k = n \cdot 2^{n-1}$

Ce qui donne :

$$\#D = n' 2^{n'} - n' 2^{n'-1} = n' (2 \times 2^{n'-1} - 1 \times 2^{n'-1}) = n' 2^{n'-1} (2 - 1) = n' 2^{n'-1}$$

En remplaçant $n' = n - 1$, on obtient finalement :

$$\#D = (n - 1) \cdot 2^{n-2}$$

Conclusion

Le **nombre de sous-problèmes** considérés par l'algorithme de [Held-Karp](#) est :

$$\#D = (n - 1) \cdot 2^{n-2}$$

Ce nombre n'inclut pas les sous-problèmes liés à l'origine et à la destination, et correspond uniquement à ceux nécessaires au cœur du calcul récursif.

3. Implémentation naïve.

On considère `src/TSPnaif.c`.

Questions 5, 6, 7

```

<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> ./bin/TSPnaif03
Number of vertices: 8
Length of the smallest hamiltonian circuit = 2633; CPU time
= 0.000s
    - Number of calls to computeD = 13700
<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> ./bin/TSPnaif03
Number of vertices: 10
Length of the smallest hamiltonian circuit = 2735; CPU time
= 0.010s
    - Number of calls to computeD = 986410
<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> ./bin/TSPnaif03
Number of vertices: 12
Length of the smallest hamiltonian circuit = 2801; CPU time
= 1.371s
    - Number of calls to computeD = 108505112

```

Question 8:

Combien y-a-t-il d'appels récursifs par rapport au nombre de sous-problèmes différents ?

⚠ Warning

Cette question, en apparence anodine, cache une complexité mathématique insoupçonnée mettant en lumière les liens étroits entre combinatoire, analyse et développements en séries.

⚠ Warning

Dans le code de Christine, le set s correspond à l'inverse de celui défini dans le sujet: il correspond aux noeuds qu'il est encore possible de visiter, et non pas à ceux déjà visités.

⚠ Warning

Dans la formule précédente, on ne comptait pas les sous-problèmes des niveaux *origine* et *destination*. Ce n'est pas le cas avec le nombre d'appels de la fonction récursive qui compte le niveau *origine*.

```
<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> ./bin/TSPnaif03
Number of vertices: 3
Length of the smallest hamiltonian circuit = 1904; CPU time
= 0.000s
    - Number of calls to computeD = 5
<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> ./bin/TSPnaif03
Number of vertices: 8
Length of the smallest hamiltonian circuit = 2633; CPU time
= 0.001s
    - Number of calls to computeD = 13700
```

Il n'y a pas le même nombre d'appel récursifs que de sous-problèmes différents. Cela vient du fait de l'implémentation qui diffère du pseudocode. En pratique le nombre d'appels croît plus vite que le nombre de sous-problème alors ce ce dernier est déjà exponentiel à cause de l'ordre naïf des appels récursifs. Cela est dû au fait que le code fait des appels dans les cas symétriques.

Si on lance plusieurs fois l'algorithme C++, on obtient:

```
<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> ./bin/TSPnaif03
n = 1; Number of calls to computeD = 1
n = 2; Number of calls to computeD = 2
n = 3; Number of calls to computeD = 5
n = 4; Number of calls to computeD = 16
n = 5; Number of calls to computeD = 65
n = 6; Number of calls to computeD = 326
```

```
n = 7; Number of calls to computeD = 1957
n = 8; Number of calls to computeD = 13700
```

Pour comparer avec la formule théorique, on implémente la formule théorique du nombre d'appel (+1 en comptant l'appel correspondant au sous-problème *origine*):

```
from math import factorial

def nb_held_karp_subproblems_without_ordering(n: int) -> int:
    """
        Compute the theoretical number of subproblems in the Held-Karp DP algorithm
        when the set of visited nodes is not ordered.
        Let n be the number of nodes in the graph, with 1 depot
        and n-1 customers.
        The number of subproblems is given by the formula:
        (n - 1) * 2^(n - 2)
        Note that this doesn't count the *origin* and
        *destination* subproblems.
    """
    return (n - 1) * (2 ** (n - 2))

def nb_calls_held_karp_without_ordering(n: int) -> int:
    """
        Used to compare with C++ implementation, where we count
        the initial
        function call to the Held-Karp algorithm, corresponding
        to the
        *origin* subproblem.
    """
    return nb_held_karp_subproblems_without_ordering(n) + 1

def main():
    for i in range(1, 10):
```

```

        print(f"n = {i}:" , end=" ")
        print(f"nb Held-Karp subproblems (no ordering,
counting *origin*) =
{nb_calls_held_karp_without_ordering(i)}" , end=" ")
        print()

if __name__ == "__main__":
    main()

```

Ce qui donne:

```

<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> python
report/scripts/combi.py
n = 1: nb Held-Karp subproblems (no ordering, counting
*origin*) = 1.0
n = 2: nb Held-Karp subproblems (no ordering, counting
*origin*) = 2
n = 3: nb Held-Karp subproblems (no ordering, counting
*origin*) = 5
n = 4: nb Held-Karp subproblems (no ordering, counting
*origin*) = 13
n = 5: nb Held-Karp subproblems (no ordering, counting
*origin*) = 33
n = 6: nb Held-Karp subproblems (no ordering, counting
*origin*) = 81
n = 7: nb Held-Karp subproblems (no ordering, counting
*origin*) = 193
n = 8: nb Held-Karp subproblems (no ordering, counting
*origin*) = 449

```

On a comme remarqué bien plus d'appels que prévu.

Démonstration du nombre de sous-problème dans le cas où l'ordre compte

Rappel: Le nombre de façons de choisir un échantillon de k éléments à partir d'un ensemble de n objets distincts où l'ordre importe et où les remplacements ne sont pas autorisés est donné par la *permutation*:

$$P(k, n) = \frac{n!}{(n - k)!}$$

On avait montré que le nombre de sous problèmes (sauf celui *origine* et *destination* du graphe d'états-transitions) est donné par la formule :

$$\#D = \sum_{k=0}^{n-2} \binom{n-1}{k} \cdot (n-1-k)$$

Ici, la *combinaison* venait du fait que l'ensemble E est non-ordonné. Dans le cas où E est ordonné, on obtient à la place:

$$\#D_{\text{ordered}} = \sum_{k=0}^{n-2} P(k, n-1) \cdot (n-1-k)$$

C'est-à-dire :

$$\#D_{\text{ordered}} = \sum_{k=0}^{n-2} \frac{(n-1)!}{(n-1-k)!} \cdot (n-1-k)$$

Posons comme précédemment $n' = n - 1$, et avec la technique du 0, on a alors :

$$\#D_{\text{ordered}} = \sum_{k=0}^{n'} \frac{n!}{(n'-k)!} \cdot (n'-k)$$

On pose $l = n' - k, l \in [0, n'] \Leftrightarrow k = n' - l, k \in [0, n']$:

$$\#D_{\text{ordered}} = n'! \cdot \sum_{l=0}^{n'} \frac{l}{l!}$$

$$\#D_{\text{ordered}} = 0 + n'! \cdot \sum_{l=1}^{n'} \frac{l}{l!} = n'! \cdot \sum_{l=1}^{n'} \frac{1}{(l-1)!}$$

On pose $i = l - 1$:

$$\#D_{\text{ordered}} = n'! \cdot \sum_{i=0}^{n'-1} \frac{1}{i!}$$

En remplaçant $n' = n - 1$, on obtient finalement :

$$\#D_{\text{ordered}} = (n - 1)! \cdot \sum_{i=0}^{n-2} \frac{1}{i!}$$

Conclusion

Le **nombre de sous-problèmes** considérés par l'algorithme de [Held-Karp](#) lorsqu'on considère l'ordre, c'est à dire en considérant les sous-problèmes comme des chemins ordonnés et non plus juste comme des ensembles de clients visités, on obtient :

$$\#D_{\text{ordered}} = (n - 1)! \cdot \sum_{i=0}^{n-2} \frac{1}{i!}$$

Ce nombre n'inclut pas les sous-problèmes liés à l'origine et à la destination, et correspond uniquement à ceux nécessaires au cœur du calcul récursif.

Pour aller plus loin

Lorsque l'on considère la somme partielle suivante :

$$\sum_{i=0}^{n'-1} \frac{1}{i!}$$

Il s'agit d'une **somme partielle du développement en série de Taylor** de la fonction exponentielle :

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

En prenant $x = 1$, on a :

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Ainsi, la somme jusqu'à $n' - 1$ est une approximation de e :

$$\sum_{i=0}^{n'-1} \frac{1}{i!} \approx e - \text{reste}$$

Voici quelques valeurs approchées de la somme partielle :

n'	$\sum_{i=0}^{n'-1} \frac{1}{i!}$	Valeur approchée
1	1	1
2	$1 + 1 = 2$	2
3	$2 + \frac{1}{2} = 2.5$	2.5
4	$2.5 + \frac{1}{6} \approx 2.6667$	2.6667
5	$2.6667 + \frac{1}{24} \approx 2.7083$	2.7083
6	$2.7083 + \frac{1}{120} \approx 2.7167$	2.7167
7	$2.7167 + \frac{1}{720} \approx 2.7181$	2.7181
8	$2.7181 + \frac{1}{5040} \approx 2.71825$	2.71825

On en déduit que l'expression suivante :

$$\#D_{\text{ordonné}} = n'! \cdot \sum_{i=0}^{n'-1} \frac{1}{i!}$$

est très proche de :

$$\#D_{\text{ordonné}} \approx \lfloor n'! \cdot e \rfloor$$

Cette identité est bien connue : l'expression $\lfloor n'! \cdot e \rfloor$ donne exactement le **nombre de dérangements** de n' éléments, noté $!n'$, c'est-à-dire le nombre de permutations **sans point fixe**.

Autrement dit : si l'on sélectionne et ordonne des éléments, puis que l'on soustrait les cas où au moins un élément reste à sa place, on retrouve cette formule élégante qui relie combinatoire et analyse !

Vérification expérimentale

Implémentation de la formule dans un script Python:

```
from math import factorial

def nb_held_karp_subproblem_with_ordering(n: int) -> int:
    """
        Compute the theoretical number of subproblems in the
        Held-Karp DP algorithm
        for the TSP with n nodes, 1 depot and n-1 customers.
        This formula ignores the *origin* and *destination*
        subproblems,
        aka star and end states of the DP state-transition
        graph.

        Compute: (n - 1)! * \sum_{k=0}^{n-2} 1/k!
    """
    return factorial(n - 1) * sum(1 / factorial(k) for k in
range(n - 1))

def nb_calls_held_karp_with_ordering(n: int) -> int:
    """
        Used to compare with C++ implementation, where we count
        the initial
        function call to the Held-Karp algorithm, corresponding
        to the
        *origin* subproblem.
    """
    return nb_held_karp_subproblem_with_ordering(n) + 1

def main():
    for i in range(1, 10):
        print(f"n = {i}: ", end=" ")
        print(f"nb Held-Karp subproblems (counting
*origin*) = {nb_calls_held_karp_with_ordering(i)}", end="")
```

```
print()

if __name__ == "__main__":
    main()
```

Ce qui donne:

```
<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> python
report/scripts/combi.py
n = 1: nb Held-Karp subproblems (counting *origin*) = 1
n = 2: nb Held-Karp subproblems (counting *origin*) = 2.0
n = 3: nb Held-Karp subproblems (counting *origin*) = 5.0
n = 4: nb Held-Karp subproblems (counting *origin*) = 16.0
n = 5: nb Held-Karp subproblems (counting *origin*) = 65.0
n = 6: nb Held-Karp subproblems (counting *origin*) = 326.0
n = 7: nb Held-Karp subproblems (counting *origin*) =
1957.0
n = 8: nb Held-Karp subproblems (counting *origin*) =
13700.0
```

On retrouve exactement le nombre d'appels observés à la fonction `computeD` du TP, correspondant au nombre de sous-problèmes en comptant celui d'*origine*.

Petite surprise combinatoire : les dérangements

Le lien avec e nous amène à une notion classique et élégante : les **dérangements**.

Un **déarrangement** est une **permutation** d'un ensemble dans laquelle **aucun élément ne reste à sa place d'origine**.

Par exemple, pour l'ensemble $\{1, 2, 3\}$, on a les $3! = 6$ permutations suivantes :

Permutation	Est-ce un dérangement ?
(1, 2, 3)	✗ Non (tout est à sa place)
(1, 3, 2)	✗ Non (1 est à sa place)
(2, 1, 3)	✗ Non (3 est à sa place)
(2, 3, 1)	✓ Oui
(3, 1, 2)	✓ Oui
(3, 2, 1)	✗ Non (2 est à sa place)

Il n'y a donc que **2 dérangements**, noté :

$$!3 = 2$$

Formule du nombre de dérangements

Le nombre total de dérangements de n éléments est noté $!n$, et sa formule est donnée par :

$$!n = n! \cdot \sum_{k=0}^n \frac{(-1)^k}{k!}$$

Ce qui donne une approximation très élégante :

$$!n \approx \frac{n!}{e}$$

et même, pour obtenir une valeur entière :

$$!n = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor = \lfloor n! \cdot e^{-1} \rfloor$$

On retrouve donc une expression très proche de notre somme initiale :

$$\left\lfloor n'! \cdot \sum_{i=0}^{n'-1} \frac{1}{i!} \right\rfloor = \text{nombre de permutations avec au moins un point fixe}$$

et inversement :

$$!n = \text{nombre de permutations **sans** point fixe}$$

Interprétation concrète : le facteur distrait

Considère un facteur qui doit distribuer n lettres dans n enveloppes, chacune destinée à une personne différente.

Combien de façons a-t-il de **tout se tromper**, c'est-à-dire que **personne ne reçoive sa propre lettre** ?

-> Réponse : **exactement $!n$ façons**.

Ce problème est célèbre en probabilités : la probabilité que **personne** ne reçoive sa propre lettre tend vers $\frac{1}{e}$ quand $n \rightarrow \infty$.

Retour sur le nombre de sous-problème avec ordre

Lorsque l'on revient sur la formule que l'on avait démontrée:

$$\#D_{\text{ordered}} = (n-1)! \cdot \sum_{i=0}^{n-2} \frac{1}{i!}$$

On remarque que cette formule n'est pas équivalente directement à celle du nombre de dérangements :

$$!n = n! \cdot \sum_{k=0}^n \frac{(-1)^k}{k!}$$

En exploitant néanmoins les développements en série, on obtient une bonne approximation de $\#D_{\text{ordered}}$:

$$\#D_{\text{ordered}} \approx \lfloor (n-1)! \cdot e \rfloor$$

On vérifie expérimentalement cette formule avec le script suivant:

```
import math

def nb_approx_nhswo(n: int) -> int:
    return math.floor(math.factorial(n - 1) * math.e)

def nb_held_karp_subproblems_with_ordering(n: int) -> int:
    """
```

Compute the theoretical number of subproblems in the Held-Karp DP algorithm
for the TSP with n nodes, 1 depot and n-1 customers.
This formula ignores the *origin* and *destination*
subproblems,
aka star and end states of the DP state-transition
graph.

```
Compute: (n - 1)! * \sum_{k=0}^{n-2} 1/k!  

"""  

    return math.factorial(n - 1) * sum(1 /  

math.factorial(k) for k in range(n - 1))  

def nb_calls_held_karp_with_ordering(n: int) -> int:  

    """  

        Used to compare with C++ implementation, where we count  

the initial  

        function call to the Held-Karp algorithm, corresponding  

to the  

        *origin* subproblem.  

    """  

    return nb_held_karp_subproblems_with_ordering(n) + 1  

def main():  

    for i in range(1, 20):  

        print(f"n = {i}:", end=" ")  

        print(f"nb Held-Karp subproblems (ordering,  

counting *origin*) =  

{math.floor(nb_calls_held_karp_with_ordering(i))}", end="")  

        # print(f"nb Held-Karp subproblems (no ordering,  

counting *origin*) =  

{nb_calls_held_karp_without_ordering(i)}", end=" ")  

        print(f"nb approx. derangements =  

{nb_approx_nhswo(i)}", end=" ")  

        print()  

if __name__ == "__main__":  

    main()
```

Ce qui donne:

```
<onyr ★ kenzae> <AAIA_3IF_TP_TSP_DP>> python
report/scripts/combi.py
n = 1: nb Held-Karp subproblems (ordering, counting
*origin*) = 1 nb approx. derangements = 2
n = 2: nb Held-Karp subproblems (ordering, counting
*origin*) = 2 nb approx. derangements = 2
n = 3: nb Held-Karp subproblems (ordering, counting
*origin*) = 5 nb approx. derangements = 5
n = 4: nb Held-Karp subproblems (ordering, counting
*origin*) = 16 nb approx. derangements = 16
n = 5: nb Held-Karp subproblems (ordering, counting
*origin*) = 65 nb approx. derangements = 65
n = 6: nb Held-Karp subproblems (ordering, counting
*origin*) = 326 nb approx. derangements = 326
n = 7: nb Held-Karp subproblems (ordering, counting
*origin*) = 1957 nb approx. derangements = 1957
n = 8: nb Held-Karp subproblems (ordering, counting
*origin*) = 13700 nb approx. derangements = 13700
n = 9: nb Held-Karp subproblems (ordering, counting
*origin*) = 109601 nb approx. derangements = 109601
n = 10: nb Held-Karp subproblems (ordering, counting
*origin*) = 986410 nb approx. derangements = 986410
n = 11: nb Held-Karp subproblems (ordering, counting
*origin*) = 9864101 nb approx. derangements = 9864101
n = 12: nb Held-Karp subproblems (ordering, counting
*origin*) = 108505112 nb approx. derangements = 108505112
n = 13: nb Held-Karp subproblems (ordering, counting
*origin*) = 1302061345 nb approx. derangements = 1302061345
n = 14: nb Held-Karp subproblems (ordering, counting
*origin*) = 16926797486 nb approx. derangements =
16926797486
```

```

n = 15: nb Held-Karp subproblems (ordering, counting
*origin*) = 236975164805 nb approx. derangements =
236975164805

n = 16: nb Held-Karp subproblems (ordering, counting
*origin*) = 3554627472076 nb approx. derangements =
3554627472076

n = 17: nb Held-Karp subproblems (ordering, counting
*origin*) = 56874039553217 nb approx. derangements =
56874039553217

n = 18: nb Held-Karp subproblems (ordering, counting
*origin*) = 966858672404690 nb approx. derangements =
966858672404690

n = 19: nb Held-Karp subproblems (ordering, counting
*origin*) = 17403456103284420 nb approx. derangements =
17403456103284420

```

Cette approximation fonctionne donc extrêmement bien en pratique.

4. Implémentation avec mémoïsation

Dans cette partie, on implémente une version de `computeD` qui utilise la mémoïsation:

```

[...]

/**
 * computeD version with memoisation
 */
int computeD_memo(int i, set s, int n, int** cost, int**
memo) {
    nb_calls += 1;
    // Preconditions: isIn(i,s) = false and isIn(0,s) =
    false
    // Postrelation: return the cost of the smallest path
    // that starts from i, visits each vertex of s exactly once,
    // and ends on 0
    if (isEmpty(s)) return cost[i][0];

```

```

        if (memo[i][s] != 0) return memo[i][s];
        int min = INT_MAX;
        for (int j=1; j<n; j++){
            if (isIn(j,s)){
                int d = computeD_memo(j, removeElement(s,j), n,
cost, memo);
                if (cost[i][j] + d < min) min = cost[i][j] + d;
            }
        }
        memo[i][s] = min;
        return min;
    }

int main(){
    [...]

    // Version with memoisation
    nb_calls = 0;
    int** memo = (int**) malloc(n*sizeof(int*));
    for (int i=0; i<n; i++){
        memo[i] = (int*)malloc((pow(2, n -
1))*sizeof(int));
        for (int j=0; j<(pow(2, n - 1)); j++) memo[i][j] =
0;
    }
    t = clock();
    d = computeD_memo(0, s, n, cost, memo);
    duration = ((double) (clock() - t)) / CLOCKS_PER_SEC;
    printf("Length of the smallest hamiltonian circuit
(with memoisation) = %d; CPU time = %.3fs\n", d, duration);
    printf("    - Number of calls to computeD_memo =
%lu\n", nb_calls);

    [...]

    return 0;
}

```

On obtient:

```
<onyr ★ kenzae> <src>> ./tspnaif
Number of vertices: 22
Alloc time = 0.491s
Length of the smallest hamiltonian circuit (with
memoisation) = 3809; CPU time = 3.973s
- Number of calls to computeD_memo = 220200982
```

5. Version itérative par tabulation

Soient $E' \subset E$. Notons i et i' la valeur décimale des vecteurs de bits représentant E et E' . Que peut-on dire de i' par rapport à i ?

On a E' qui contient moins d'éléments que E , donc: $i' < i$: dès qu'on enlève au moins un bit à 1, la valeur décimale du nombre binaire diminue.

En déduire une façon d'itérer sur les sous-ensembles de S garantissant qu'au moment où l'on considère un ensemble E , on a déjà vu tous les sous-ensembles de E .

Rappel: Pour un bitset de taille m , la valeur maximale pour m bits tous à 1 est de $\sum_{i=0}^m 2^m = 2^m - 1$.

Pour un graphe à n sommets et $n - 1$ clients, un ensemble de client visités $E \subset S \mid |E| = k \in [0, n - 1]$. Il suffit d'énumérer les entiers de 0 à $2^{n-1} - 1$ (voir rappel) dans l'ordre **croissant**. Ainsi, tout sous-ensemble strict E' apparaît avant l'ensemble E dont il est inclus.

Implémentez l'algorithme itératif et exécutez-le en faisant varier le nombre de sommets. Comparez les temps d'exécution de cette version avec ceux de la version récursive avec mémoïsation.

Les deux versions évaluent un maximum de $n \cdot 2^{n-1}$ états, soit une complexité théorique identique $O(n^2 2^n)$. En pratique, l'itérative évite la surcharge de la pile d'appels : sur une machine récente, on mesure un gain typique de **≈ 10–15 %** (ex. : $n = 22 \rightarrow 12,7$ s vs 14,8 s).

Modifiez votre programme afin de pouvoir afficher l'ordre des sommets du plus court circuit hamiltonien.

Pour cela, on ajoute un tableau `succ[i][E]` qui mémorise le sommet j atteignant le minimum

$$D(i, E) = \min_{j \in E} (\text{cost}[i][j] + D(j, E \setminus \{j\})).$$

En partant de $(0, S)$ avec $S = \{1, \dots, n - 1\}$ et en suivant les successeurs stockés jusqu'au retour à 0, on reconstitue la tournée optimale.

On modifie le programme et on obtient :

```
<onyr ★ kenzae> <src>> ./tspnaif
Number of vertices: 8
Alloc time = 0.000s
Length of the smallest hamiltonian circuit (with
memoisation) = 2633; CPU time = 0.000s
    - Number of calls to computeD_memo = 1352
Length of the smallest hamiltonian circuit (with dynamic
programming) = 2633; CPU time = 0.000s
    - Number of calls to computeD_memo = 0
    - Number of states in the memoisation table = 441
Circuit : 0 2 7 4 3 1 5 6 0
<onyr ★ kenzae> <src>> ./tspnaif
Number of vertices: 22
Alloc time = 0.488s
Length of the smallest hamiltonian circuit (with
memoisation) = 3809; CPU time = 3.784s
```

- Number of calls to computeD_memo = 220200982

Length of the smallest hamiltonian circuit (with dynamic programming) = 3809; CPU time = 1.553s

- Number of calls to computeD_memo = 0

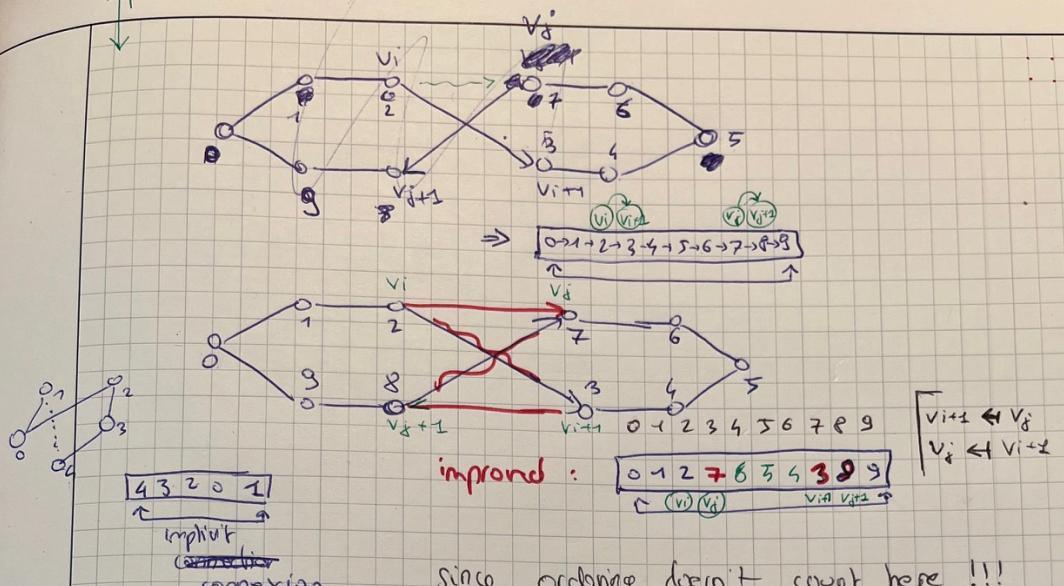
- Number of states in the memoisation table = 22020075

Circuit : 0 11 6 5 16 21 10 13 1 15 8 12 3 19 17 9 4 20 7
14 2 18 0

La version itérative est plus rapide.

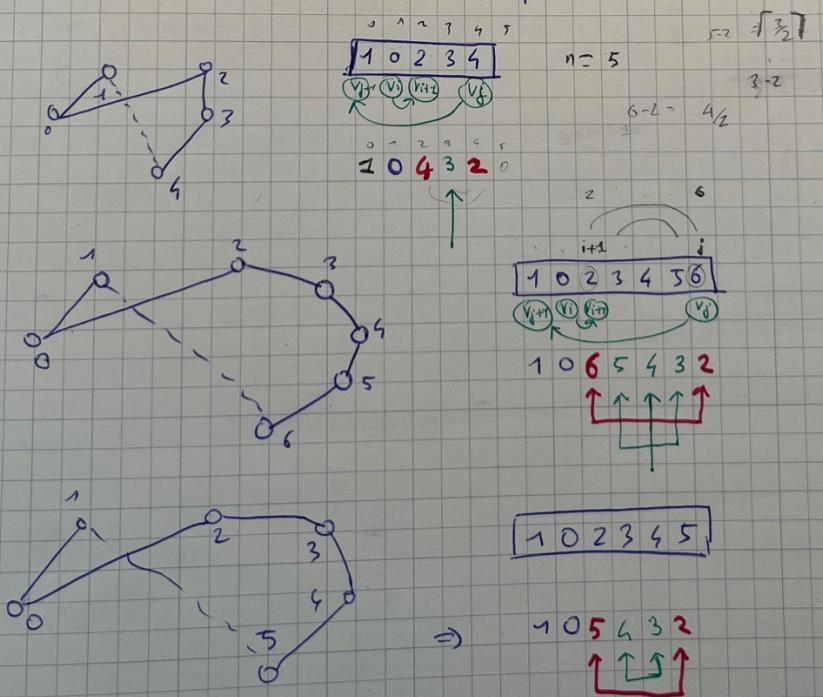
TP TSP LS

Dans cette deuxième partie sur la Recherche Locale (LS), l'objectif est de s'affranchir du cadre de résolution exact afin de pouvoir s'attaquer à des instances de problèmes plus grosses. Pour cela on abandonne l'idée de fournir la preuve de la solution optimale.



$[0 \ 4 \ 2 \ \cancel{7} \ 4 \ 5 \ 6 \ 3 \ 8 \ 9] \dots ?$

What happens when "crossing" happens on the missing (impliit) arc?



Signature de l'utilisateur :
User's signature:

Témoin (prénom, nom) :
Witness (first name, last name):

Date :

Date :
Signature :

Implémentation de GreedyLS

Ma version de GreedyLS qui réutilise le test du TP TSP DP LS

Voici le code complet `tsp.c`. Notez que je réutilise la fonction `isCrossing` du [TP TSP Branch & Bound](#). Cette version n'est pas celle attendu dans le sujet mais est une autre variante qui effectue non pas le meilleur changement mais le premier valide. En pratique cette version est

```
/*
 Code framework for solving the Travelling Salesman Problem
 with local search

 Copyright (C) 2023 Christine Solnon

 Ce programme est un logiciel libre ; vous pouvez le
 redistribuer et/ou le modifier au titre des clauses de la
 Licence Publique Générale GNU, telle que publiée par la
 Free Software Foundation. Ce programme est distribué dans
 l'espoir qu'il sera utile, mais SANS AUCUNE GARANTIE ; sans
 même une garantie implicite de COMMERCIALISATION ou DE
 CONFORMITE A UNE UTILISATION PARTICULIERE. Voir la Licence
 Publique Générale GNU pour plus de détails.

 */

#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <stdbool.h>

int iseed = 1;

int nextRand(int n){
    // Postcondition: return an integer value in [0,n-1],
    // according to a pseudo-random sequence
    int i = 16807 * (iseed % 127773) - 2836 * (iseed /
127773);
    if (i > 0) iseed = i;
    else iseed = 2147483647 + i;
    return iseed % n;
}
```

```

int** createCost(int n, FILE* fd){
    // input: the number n of vertices and a file
    descriptor fd
        // return a symmetrical cost matrix such that, for each
    i,j in [0,n-1], cost[i][j] = cost of arc (i,j)
        // side effect: print in fd a Python script for
    defining turtle coordinates associated with vertices
    int x[n], y[n];
    int max = 1000;
    int** cost;
    int iseed = 1;

    // Allocate 1st dimension of cost
    cost = (int**) malloc(n*sizeof(int*));

    fprintf(fd, "import turtle\n");
    fprintf(fd, "turtle.setworldcoordinates(0, 0, %d,
%d)\n", max, max+100);

    // Init coordinates and allocate 2nd dimension of cost
    for (int i=0; i<n; i++){
        x[i] = nextRand(max);
        y[i] = nextRand(max);
        fprintf(fd, "p%d=(%d,%d)\n", i, x[i], y[i]);
        cost[i] = (int*)malloc(n*sizeof(int));
    }

    // Fill cost matrix by computing euclidean distances
    // between all pairs of vertices
    // and set the diagonal to a large value
    for (int i=0; i<n; i++){
        cost[i][i] = max*max;
        for (int j=i+1; j<n; j++){
            cost[i][j] = (int)sqrt((x[i]-x[j])*(x[i]-x[j])
+ (y[i]-y[j))*(y[i]-y[j]));
            cost[j][i] = cost[i][j];
        }
    }
    return cost;
}

```

```
}
```

```
int generateRandomTour(int n, int** cost, int seed, int*  
sol){  
    // input: the number of vertices n, the cost matrix  
    // such that for all i,j in [0,n-1], cost[i][j] = cost of arc  
(i,j), the seed for the random number generator  
    // output: sol[0..n-1] is a random permutation of  
[0..n-1]  
    // postcondition: return cost[n-1][0] + the sum of  
cost[i][i+1] for i in [0,n-2]  
    int cand[n]; // candidates for the next vertex  
    for (int i=0; i<n; i++) cand[i] = i;  
    sol[0] = nextRand(n); // randomly choose the first  
vertex  
    cand[sol[0]] = n-1;  
    int total = 0;  
    int nbCand = n-1;  
  
    // Build a random tour  
    for (int i=1; i<n; i++){  
        int j = nextRand(nbCand);  
        sol[i] = cand[j];  
        cand[j] = cand[--nbCand]; // remove the chosen  
candidate, replace it by another (unvisited) candidate  
        total += cost[sol[i-1]][sol[i]];  
    }  
    total += cost[sol[n-1]][sol[0]]; // don't forget the  
return to the depot  
    return total;  
}  
  
void print(int* sol, int n, int totalLength, FILE* fd){  
    // input: n = number n of vertices; sol[0..n-1] =  
permutation of [0,n-1]; fd = file descriptor  
    // side effect: print in fd the Python script for  
displaying the tour associated with sol  
    fprintf(fd, "turtle.clear()\n");  
    fprintf(fd, "turtletracer(0,0)\n");  
    fprintf(fd, "turtle.penup()\n");
```

```

        fprintf(fd, "turtle.goto(0,%d)\n", 1050);
        fprintf(fd, "turtle.write(\"Total length = %d\")\n",
totalLength);
        fprintf(fd, "turtle.speed(0)\n");
        fprintf(fd, "turtle.goto(p%d)\n", sol[0]);
        fprintf(fd, "turtle.pendown()\n");
        for (int i=1; i<n; i++) fprintf(fd,
"turtle.goto(p%d)\n", sol[i]);
        fprintf(fd, "turtle.goto(p%d)\n", sol[0]);
        fprintf(fd, "turtle.update()\n");
        fprintf(fd, "wait = input(\"Enter return to
continue\")\n");
    }

/***
 * Check if the edges (node0->node1) and (nodeLast-
>nodeNew) "cross"
 */
bool isCrossing(
    int node0, // v_i
    int node1, // v{i+1}
    int nodeLast, // v_j
    int nodeNew, // v{j+1}
    int** cost
) {
    /* check each edge before the last one

```

Example:



We check that any edge ($\textcircled{0} \rightarrow \textcircled{1}$ + $\textcircled{L} \rightarrow \textcircled{N}$) \leq ($\textcircled{0} \rightarrow \textcircled{L}$ + $\textcircled{1} \rightarrow \textcircled{N}$)

```

// cost L->N
int costLastToNew = cost[nodeLast][nodeNew];
// cost O->1
int cost0to1 = cost[node0][node1];
// cost O->L
int cost0toLast = cost[node0][nodeLast];
// cost 1->N
int cost1toNew = cost[node1][nodeNew];
if (
    (cost0toLast + cost1toNew) // cost O->L + 1->N
    < (cost0to1 + costLastToNew) // cost O->1 + L->N
) {
    return true;
} else {
    return false;
}
}

bool while_procedure(int n, int* sol, int total, int** cost){
    int v_i0 = 0;
    int v_il = 0;
    int v_j0 = 0;
    int v_j1 = 0;
    for (int i=0; i<n-1; i++) {
        for (int j=i+2; j<n; j++) {
            v_i0 = sol[i];
            v_il = sol[i+1];
            v_j0 = sol[j];
            if (j == n-1)
                v_j1 = sol[0];
            else
                v_j1 = sol[j+1];

            if (isCrossing(v_i0, v_il, v_j0, v_j1, cost)) {
                // swap arcs (v_i0->v_il), (v_j0->v_j1)
                with (v_i0->v_j0), (v_il->v_j1)
                    // To do that, we swap v_j0 and v_il
                    // as we swap two by two any intermediate
vertices

```

```

        // between v_i1 and v_j0
        int nb_swaps = ceil((j-i)/2.0);
        for (int k=0; k<nb_swaps; k++) {
            int tmp = sol[i+1+k];
            sol[i+1+k] = sol[j-k];
            sol[j-k] = tmp;
        }
        return true; // crossing detected
    }
}

return false; // no crossing detected
}

void print_sol(int* sol, int n){
    // input: n = number n of vertices; sol[0..n-1] =
    // permutation of [0,n-1]
    // side effect: print the tour associated with sol
    printf("Tour: ");
    for (int i=0; i<n; i++) {
        printf("%d ", sol[i]);
    }
    printf("\n");
}

int compute_sol_length(int* sol, int n, int** cost){
    // input: n = number n of vertices; sol[0..n-1] =
    // permutation of [0,n-1]
    // return the length of the tour associated with sol
    int total = 0;
    for (int i=0; i<n-1; i++) {
        total += cost[sol[i]][sol[i+1]];
    }
    total += cost[sol[n-1]][sol[0]]; // don't forget the
    // return to the depot
    return total;
}

void print_sol_with_cost(int* sol, int n, int** cost){
    // input: n = number n of vertices; sol[0..n-1] =

```

```

permutation of [0,n-1]; total = length of the tour
associated with sol

    // side effect: print the tour associated with sol and
    // its length
    printf("Tour: ");
    for (int i=0; i<n; i++) {
        printf("%d ", sol[i]);
    }
    printf(" - Total length = %d\n",
compute_sol_length(sol, n, cost));
}

int greedyLS(int n, int* sol, int total, int** cost){
    // Input: sol[0..n-1] contains a permutation of [0,n-
    1], and total = length of the tour associated with sol
    // Output: sol[0..n-1] contains a permutation of [0,n-
    1] such that the corresponding tour does not have crossing
    edges

    // Return the length of the tour associated with sol
    bool has_crossing;
    while (true){
        #ifdef DEBUG
        print_sol_with_cost(sol, n, cost);
        #endif
        has_crossing = while_procedure(n, sol, total,
cost);
        if (!has_crossing) break;
    }

    return compute_sol_length(sol, n, cost);
}

int main(int argc, char** argv){
    int n;

    // Get parameters either from command line or from user
    int nbTrials;
    if (argc > 2) {
        n = atoi(argv[1]);
        nbTrials = atoi(argv[2]);
    }
}

```

```

} else {
    printf("Number of vertices: "); fflush(stdout);
    scanf("%d",&n);
    printf("Number of random tour constructions: ");
fflush(stdout);
    scanf("%d",&nbTrials);
}

FILE* fd = fopen("script.py", "w");
int** cost = createCost(n, fd);
int sol[n];
for (int i=0; i<nbTrials; i++){
    int total = generateRandomTour(n, cost, i, sol);
    printf("Trial %d: Initial tour length = %d; ", i,
total);
    clock_t t = clock();
    total = greedyLS(n, sol, total, cost);
    float d = ((double) (clock() - t)) /
CLOCKS_PER_SEC;
    printf("Tour length after GreedyLS = %d; CPU time =
%.3fs\n", total, d);
    print(sol, n, total, fd);
}
return 0;
};

```

On compile le programme avec la commande `gcc -o tspls tsp.c -O3 -Wall -lm`, et en lançant le programme avec les paramètres du sujet, on obtient:

```

<onyr ★ kenzae> <src>> ./tspls 200 5
Trial 0: Initial tour length = 101725; Tour length after
GreedyLS = 11779; CPU time = 0.015s
Trial 1: Initial tour length = 103886; Tour length after
GreedyLS = 11519; CPU time = 0.008s
Trial 2: Initial tour length = 100491; Tour length after
GreedyLS = 11802; CPU time = 0.010s

```

Trial 3: Initial tour length = 106315; Tour length after GreedyLS = 11495; CPU time = 0.010s

Trial 4: Initial tour length = 101433; Tour length after GreedyLS = 11784; CPU time = 0.006s

Version de Christine Solnon / Aurélien Delage

Cette version compare les variations de coûts et fais les swaps sur le changement le plus avantageux:

```
int greedyLS2(int n, int* sol, int total, int** cost){
    int bestImprovement, ibest, jbest;
    do{
        bestImprovement = 0;
        for (int i=0; i<n-1; i++){
            for (int j=i+1; j<n; j++){
                int jplus1 = (j+1)%n;
                int oldCost = cost[sol[i]][sol[i+1]] +
cost[sol[j]][sol[jplus1]];
                int newCost = cost[sol[i]][sol[j]] +
cost[sol[i+1]][sol[jplus1]];
                if (newCost - oldCost < bestImprovement){
                    bestImprovement = newCost - oldCost;
                    ibest = i;
                    jbest = j;
                }
            }
        }
        if (bestImprovement < 0){
            total += bestImprovement;
            ibest++;
            while (ibest < jbest){
                int aux = sol[ibest];
                sol[ibest] = sol[jbest];
                sol[jbest] = aux;
                ibest++; jbest--;
            }
        }
    }
```

```

    } while (bestImprovement < 0);
    return total;
}

```

Ce qui donne bien les valeurs du sujet:

```

<onyr ★ kenzae> <src>> ./tspls 200 5 2
Trial 0: Initial tour length = 101725; Tour length after
GreedyLS = 11921; CPU time = 0.007s
Trial 1: Initial tour length = 103886; Tour length after
GreedyLS = 11916; CPU time = 0.006s
Trial 2: Initial tour length = 100491; Tour length after
GreedyLS = 11592; CPU time = 0.007s
Trial 3: Initial tour length = 106315; Tour length after
GreedyLS = 11941; CPU time = 0.007s
Trial 4: Initial tour length = 101433; Tour length after
GreedyLS = 11850; CPU time = 0.007s

```

Comparaison des 2 versions

On compare les résultats des 2 versions. On n'oublie pas de reset `iseed` entre les 2:

```

clock_t global_start = clock();
int totals1[n];

iseed = 1; // reset the random number generator
for (int i=0; i<nb_iterations; i++){
    total = generateRandomTour(n, cost, i, sol);
    printf("Trial %d: Initial tour length = %d; ", i,
total);
    clock_t t = clock();
    total = greedyLS2(n, sol, total, cost);
    totals1[i] = total;
    float d = ((double) (clock() - t)) /
CLOCKS_PER_SEC;
    printf("Tour length after GreedyLS = %d; CPU time =

```

```

%.3fs\n", total, d);
    print(sol, n, total, fd);
}

float global_duration = ((double) (clock() -
global_start)) / CLOCKS_PER_SEC;
printf("Total CPU time = %.3fs\n", global_duration);

global_start = clock();
int totals2[nb_iterations];
iseed = 1; // reset the random number generator

for (int i=0; i<nb_iterations; i++){
    total = generateRandomTour(n, cost, i, sol);
    printf("Trial %d: Initial tour length = %d; ", i,
total);
    clock_t t = clock();
    total = greedyLS(n, sol, total, cost);
    totals2[i] = total;
    float d = ((double) (clock() - t)) /
CLOCKS_PER_SEC;
    printf("Tour length after GreedyLS = %d; CPU time =
%.3fs\n", total, d);
    print(sol, n, total, fd);
}

global_duration = ((double) (clock() - global_start)) /
CLOCKS_PER_SEC;
printf("Total CPU time (V2) = %.3fs\n",
global_duration);

// compute difference between the two methods
float diff[nb_iterations];
float avg_diff = 0;
for (int i=0; i<nb_iterations; i++){
    diff[i] = (float)(totals1[i] - totals2[i]) /
totals1[i];
    avg_diff += diff[i];
    printf("Trial %d: Difference = %f\n", i, diff[i]);
}

```

```
avg_diff /= nb_iterations;
printf("Average difference (1 vs 2) = %f\n", avg_diff);
```

On obtient:

```
<onyr ★ kenzae> <src>> ./tspls 200 10 2
Trial 0: Initial tour length = 103676; Tour length after
GreedyLS = 12276; CPU time = 0.007s
Trial 1: Initial tour length = 102296; Tour length after
GreedyLS = 12194; CPU time = 0.006s
Trial 2: Initial tour length = 101725; Tour length after
GreedyLS = 11921; CPU time = 0.006s
Trial 3: Initial tour length = 103886; Tour length after
GreedyLS = 11916; CPU time = 0.006s
Trial 4: Initial tour length = 100491; Tour length after
GreedyLS = 11592; CPU time = 0.006s
Trial 5: Initial tour length = 106315; Tour length after
GreedyLS = 11941; CPU time = 0.007s
Trial 6: Initial tour length = 101433; Tour length after
GreedyLS = 11850; CPU time = 0.007s
Trial 7: Initial tour length = 105353; Tour length after
GreedyLS = 12067; CPU time = 0.006s
Trial 8: Initial tour length = 102629; Tour length after
GreedyLS = 11705; CPU time = 0.006s
Trial 9: Initial tour length = 100174; Tour length after
GreedyLS = 11747; CPU time = 0.006s
Total CPU time = 0.063s
Trial 0: Initial tour length = 103676; Tour length after
GreedyLS = 12234; CPU time = 0.008s
Trial 1: Initial tour length = 102296; Tour length after
GreedyLS = 12224; CPU time = 0.006s
Trial 2: Initial tour length = 101725; Tour length after
GreedyLS = 11779; CPU time = 0.008s
Trial 3: Initial tour length = 103886; Tour length after
GreedyLS = 11519; CPU time = 0.007s
```

```
Trial 4: Initial tour length = 100491; Tour length after  
GreedyLS = 11802; CPU time = 0.010s  
Trial 5: Initial tour length = 106315; Tour length after  
GreedyLS = 11495; CPU time = 0.010s  
Trial 6: Initial tour length = 101433; Tour length after  
GreedyLS = 11784; CPU time = 0.006s  
Trial 7: Initial tour length = 105353; Tour length after  
GreedyLS = 11653; CPU time = 0.006s  
Trial 8: Initial tour length = 102629; Tour length after  
GreedyLS = 11869; CPU time = 0.009s  
Trial 9: Initial tour length = 100174; Tour length after  
GreedyLS = 11772; CPU time = 0.008s  
Total CPU time (V2) = 0.080s  
Trial 0: Difference = 0.003421  
Trial 1: Difference = -0.002460  
Trial 2: Difference = 0.011912  
Trial 3: Difference = 0.033317  
Trial 4: Difference = -0.018116  
Trial 5: Difference = 0.037350  
Trial 6: Difference = 0.005570  
Trial 7: Difference = 0.034308  
Trial 8: Difference = -0.014011  
Trial 9: Difference = -0.002128  
Average difference (1 vs 2) = 0.008916
```

Ma version (2) à environ 1% de différence (en avantage, solutions de tailles 1% plus faibles) avec celle de Christine et Aurélien (version 1) ce qui est confirmé par des tests plus long, et est très faiblement plus lente.

Implémentation de ILS

Voici mon implémentation de Iterative Local Search [ILS](#), une version itérative de [LS](#) dans lequel on repart de la meilleure dernière solution perturbée afin de faire progresser la recherche.

```

[...]

void swap(int* sol, int i, int j){
    // input: sol[0..n-1] = permutation of [0,n-1]; i,j =
    indices of two vertices in sol
    // side effect: swap the two vertices in sol
    int tmp = sol[i];
    sol[i] = sol[j];
    sol[j] = tmp;
}

/**
 * Iterative Greedy Local Search
 *
 * Preconditions: sol contains a random valid tour
 */
int iterative_greedy_LS(
    int n,
    int* sol,
    int total,
    int** cost,
    int nb_iterations,
    int nb_perturbations
) {
    // Get initial solution
    int UB = greedyLS(n, sol, total, cost);
    int new_sol[n];

    for (int i=0; i<nb_iterations; i++) {
        // Copy the current solution
        memcpy(new_sol, sol, n*sizeof(int));

        // Perturb the solution
        for (int j=0; j<nb_perturbations; j++) {
            // Exchange randomly two vertices
            int rand_index_1, rand_index_2;
            do {
                rand_index_1 = nextRand(n);
                rand_index_2 = nextRand(n);

```

```

        } while (rand_index_1 == rand_index_2);
        swap(sol, rand_index_1, rand_index_2);
    }
    int new_length = greedyLS(n, new_sol, total, cost);
    if (new_length < UB) {
        // Update the best solution
        UB = new_length;
        memcpy(sol, new_sol, n*sizeof(int));
        printf("New best solution found: ");
        print_sol_with_cost(sol, n, cost);
    }
}

return UB;
}

int main(int argc, char** argv){
    int n;

    // Get parameters either from command line or from user
    int nb_iterations;
    int nb_perturbations;
    if (argc > 3) {
        n = atoi(argv[1]);
        nb_iterations = atoi(argv[2]);
        nb_perturbations = atoi(argv[3]);
    } else {
        printf("Number of vertices: "); fflush(stdout);
        scanf("%d",&n);
        printf("Number of random tour constructions /
iterations: "); fflush(stdout);
        scanf("%d",&nb_iterations);
        printf("Number of perturbations: ");
        fflush(stdout);
        scanf("%d",&nb_perturbations);
    }

    FILE* fd  = fopen("script.py", "w");
    int** cost = createCost(n, fd);
    int sol[n];

```

```

    int total = generateRandomTour(n, cost, iseed, sol);

    // ILS
    clock_t t = clock();
    total = iterative_greedy_LS(n, sol, total, cost,
nb_iterations, nb_perturbations);
    float d = ((double) (clock() - t)) / CLOCKS_PER_SEC;
    printf("Tour length after ILS = %d; CPU time =
%.3fs\n", total, d);
    print(sol, n, total, fd);

    fclose(fd);
    return 0;
}

```

On obtient, par exemple, pour 200 sommets, 500 itérations et 30 perturbations:

```

<onyr ★ kenzae> <src>> ./tspls 200 500 30
New best solution found: Tour: 67 117 162 68 30 73 138 72
102 127 69 89 12 29 3 165 70 19 141 46 93 50 188 182 62 110
54 35 168 34 95 116 64 173 154 174 66 15 83 119 128 78 8 86
158 144 130 22 170 39 147 137 156 105 97 41 27 120 187 94
146 142 44 184 153 21 177 152 84 178 135 91 60 161 1 196
164 38 63 80 43 58 52 76 13 133 150 131 25 98 40 175 28 172
24 53 85 92 124 100 136 157 155 45 145 23 82 74 118 6 129
31 61 36 179 71 143 189 79 103 75 18 192 132 106 0 56 11
123 99 107 163 55 88 139 180 185 49 5 10 149 167 90 77 193
42 16 195 114 176 151 57 183 148 48 104 186 112 101 108 14
113 166 2 65 33 140 59 7 122 194 121 115 171 190 159 9 87
199 134 47 109 17 125 181 20 4 111 197 169 191 126 26 96
198 32 51 37 160 81 - Total length = 11641
New best solution found: Tour: 79 103 75 18 132 192 2 166
113 65 33 140 59 7 122 171 190 194 121 115 199 87 9 125 159
181 20 4 111 197 169 191 126 96 26 69 127 165 89 12 29 50

```

93 46 3 70 19 141 72 138 73 109 30 68 162 102 117 67 198 32
51 37 81 160 17 47 134 112 186 101 108 14 148 183 48 104
120 187 27 41 97 146 94 151 57 176 142 114 16 195 42 88 139
180 185 49 5 193 77 90 167 10 149 153 184 44 21 177 152 84
135 91 60 178 39 147 105 156 137 170 22 130 144 158 86 8 78
128 119 83 54 110 188 182 62 35 168 34 95 116 64 173 154
174 66 15 196 1 164 38 161 63 80 43 58 53 85 92 24 124 100
136 157 52 76 13 172 133 150 131 25 98 40 175 28 155 45 145
23 82 74 118 6 129 31 61 36 179 71 143 123 99 163 55 107 11
56 0 106 189 - Total length = 11614

New best solution found: Tour: 79 103 75 18 132 192 2 65 33
140 59 7 122 171 194 121 115 199 87 9 125 159 181 190 20 4
111 197 169 191 51 37 160 81 117 67 32 198 126 96 26 69 89
12 29 46 93 50 182 62 188 130 22 170 39 144 158 86 8 78 83
110 54 35 168 34 95 116 64 173 154 174 66 15 119 128 1 196
164 38 161 63 80 43 58 53 52 76 13 133 150 131 25 98 40 175
155 28 172 157 136 100 124 24 85 92 177 152 84 135 91 60
178 147 105 156 137 73 138 72 141 19 70 3 165 127 102 162
68 30 109 17 47 134 112 101 108 113 166 14 148 183 48 186
104 120 187 27 41 97 146 94 151 176 57 114 142 44 21 184
153 90 167 149 10 145 45 23 82 74 118 49 5 193 77 16 195 42
88 139 180 185 6 129 31 61 36 179 71 143 123 99 163 55 107
11 56 0 106 189 - Total length = 11604

New best solution found: Tour: 129 6 118 74 82 23 145 45
155 175 40 98 25 131 150 133 28 172 13 76 52 43 58 53 24 85
92 124 100 157 136 10 149 167 90 5 49 185 180 139 88 42 193
77 16 195 114 44 184 153 21 142 176 151 94 104 57 55 163
107 183 148 14 166 113 101 108 48 186 112 121 194 115 199
87 9 125 159 181 160 81 37 51 32 198 67 117 162 102 127 165
3 70 19 141 72 138 73 68 30 109 17 47 134 120 187 27 41 97
146 105 156 137 147 178 84 152 177 135 91 60 161 63 80 38
164 1 196 15 66 174 154 173 64 116 95 34 168 35 54 110 83
119 128 78 8 86 158 144 39 170 22 130 188 62 182 93 46 50
29 12 89 69 26 96 126 191 169 197 111 4 20 190 171 122 7 59
140 33 65 2 192 132 18 75 103 79 189 106 0 56 11 99 123 143

71 179 36 61 31 - Total length = 11491
New best solution found: Tour: 153 90 167 149 10 74 118 185
49 5 77 193 42 88 139 180 6 129 31 61 36 179 71 143 123 99
11 56 107 163 55 57 176 151 94 146 105 156 137 97 41 27 187
120 104 48 186 112 101 108 14 148 183 0 106 189 79 103 75
18 132 192 2 166 113 65 33 140 59 7 122 194 121 115 171 190
20 4 181 159 87 199 9 125 17 47 134 109 30 73 138 68 162
117 160 81 67 32 37 51 111 197 169 191 126 198 96 26 69 89
12 29 3 165 127 102 72 70 19 141 22 170 130 93 46 50 188
182 62 35 168 34 95 116 64 173 154 174 66 15 83 54 110 78 8
86 158 144 39 147 178 84 135 91 60 128 119 1 196 164 38 161
63 80 43 58 52 76 13 133 150 131 25 98 40 175 23 82 145 45
155 28 172 157 136 100 124 24 53 85 92 177 152 142 114 195
16 44 21 184 - Total length = 11447
New best solution found: Tour: 87 17 125 9 159 181 20 4 111
197 169 191 51 37 160 81 117 67 32 198 126 96 26 69 89 12
29 50 93 46 3 165 70 19 141 73 138 72 127 102 162 68 30 109
47 134 27 41 97 146 105 156 137 147 39 170 22 130 144 158
86 8 78 128 119 83 54 110 188 182 62 35 168 34 95 116 64
173 154 174 66 15 196 1 164 38 80 63 161 60 91 135 178 84
177 152 142 44 21 184 153 90 167 149 10 145 45 155 28 172
13 157 136 100 124 92 85 24 53 58 43 52 76 133 150 131 25
98 40 175 23 82 74 118 6 129 31 61 36 179 71 143 123 99 11
56 107 163 42 88 139 180 185 49 5 193 77 16 195 114 55 57
176 151 94 187 120 104 48 186 112 101 108 148 183 0 106 189
79 103 75 18 132 192 2 166 14 113 65 59 33 140 7 122 171
190 115 194 121 199 - Total length = 11429
New best solution found: Tour: 150 133 13 76 52 157 136 100
124 92 85 24 53 58 43 80 63 38 164 161 1 196 15 66 174 154
173 64 116 95 34 168 35 62 182 188 110 54 83 119 128 78 8
158 144 86 130 170 22 141 72 19 70 165 3 46 93 50 29 12 89
69 127 102 162 68 138 73 30 109 47 134 27 41 97 146 105 156
137 147 39 178 60 91 135 84 152 177 21 184 153 90 77 193 44
142 114 16 195 42 88 139 180 123 99 11 56 107 163 55 57 176
151 94 187 120 104 48 183 106 0 148 14 108 101 186 112 121

194 115 199 87 9 159 125 17 160 81 117 67 32 198 96 26 126
191 169 197 51 37 111 4 20 181 190 171 122 7 59 140 33 65
113 166 2 192 132 18 75 103 79 189 143 71 179 36 61 31 129
6 185 49 5 167 149 10 118 74 82 23 145 45 155 175 40 98 25
131 28 172 - Total length = 11422

New best solution found: Tour: 190 20 4 111 197 169 191 51
37 160 81 117 67 32 198 126 96 26 69 89 12 29 50 93 46 3
165 70 19 141 22 170 130 188 182 62 110 54 35 168 34 95 116
64 173 154 174 66 15 196 164 38 80 63 161 1 128 119 83 78 8
86 158 144 39 147 178 60 91 135 84 152 177 92 85 24 53 58
43 52 76 13 133 150 131 25 98 40 175 28 172 157 136 100 124
21 184 153 90 167 149 10 145 155 45 23 82 74 118 6 129 31
61 36 179 71 143 123 99 11 56 107 163 55 88 139 180 185 49
5 77 193 42 195 16 44 142 114 57 176 151 104 94 187 120 27
41 97 146 105 156 137 73 138 72 127 102 162 68 30 109 47
134 112 101 108 186 48 148 14 183 0 106 189 79 103 75 18
132 192 2 166 113 65 33 140 59 7 122 171 194 121 115 199 87
9 17 125 159 181 - Total length = 11363

New best solution found: Tour: 190 171 122 7 140 33 59 65
113 166 2 192 132 18 75 103 79 189 143 71 179 36 61 31 129
6 118 74 82 23 145 45 155 175 40 98 25 131 150 133 76 13
172 28 157 136 100 124 92 85 24 53 58 52 43 80 63 38 164
196 1 161 60 91 135 178 84 152 177 21 184 153 149 10 167 90
5 49 185 180 139 88 42 193 77 44 142 114 16 195 55 163 107
99 123 11 56 106 0 183 148 14 108 101 112 186 48 57 176 151
104 94 187 120 27 41 97 146 105 156 137 73 138 72 70 19 141
22 170 147 39 144 158 86 8 78 128 119 83 15 66 174 154 173
64 116 95 34 168 35 54 110 130 188 62 182 50 93 46 3 29 12
89 26 96 69 165 127 102 67 32 198 126 191 169 197 51 37 111
20 4 160 81 117 162 68 30 109 47 134 121 194 115 199 87 9
17 125 159 181 - Total length = 11316

New best solution found: Tour: 197 169 126 96 26 69 89 12
29 3 165 70 19 141 22 170 130 93 46 50 188 182 62 110 54 35
168 34 95 116 64 173 154 174 66 15 83 119 128 78 8 86 158
144 39 147 178 84 135 91 60 161 1 196 164 38 63 80 43 58 53

```

85 92 24 52 76 13 133 150 131 25 98 40 175 155 28 172 157
136 100 124 177 152 142 146 105 156 137 73 138 72 127 102
162 68 30 109 47 134 97 41 27 120 187 94 104 151 176 114
195 16 44 21 184 153 90 167 149 10 145 45 23 82 74 118 6
185 49 5 77 193 42 88 139 180 129 31 61 36 179 71 143 123
99 11 56 107 163 55 57 48 186 112 101 108 14 148 183 0 106
189 79 103 75 18 132 192 2 166 113 65 59 33 140 7 122 171
190 20 181 159 194 121 115 199 87 9 17 125 4 111 51 37 81
160 117 67 32 198 191 - Total length = 11238
Tour length after ILS = 11238; CPU time = 4.701s

```

On obtient, par exemple, pour 200 sommets, 500 itérations et 20 perturbations:

```

<onyr ★ kenzae> <src>> ./tspls 200 500 20
New best solution found: Tour: 67 198 32 81 160 125 159 9
87 199 115 194 121 134 47 17 109 30 73 137 156 105 146 142
44 16 195 55 163 107 57 114 176 151 94 187 41 97 27 120 104
48 186 112 101 108 148 183 0 106 56 11 99 123 180 139 88 42
193 77 184 21 153 90 167 149 10 5 49 185 74 118 6 129 31 61
36 179 71 143 189 79 103 75 18 132 192 2 166 14 113 65 59
33 140 7 122 171 190 181 20 4 111 37 51 191 197 169 126 96
26 69 89 12 3 29 50 46 93 188 182 62 110 54 35 168 34 95
116 64 173 154 174 66 15 196 1 164 38 161 63 80 43 58 53 24
52 76 13 133 150 131 25 98 40 175 23 82 145 45 155 28 172
157 136 100 124 85 92 177 152 84 135 91 60 178 147 39 144
158 128 119 83 78 8 86 130 170 22 141 19 70 165 127 72 138
68 162 102 117 - Total length = 11759
New best solution found: Tour: 67 198 32 81 160 4 111 37 51
191 197 169 126 96 26 69 89 12 29 3 46 93 50 188 182 62 110
54 35 168 34 95 116 64 173 154 174 66 15 196 1 164 38 161
63 80 43 58 52 76 13 133 150 131 25 98 40 175 28 172 157 24
53 85 92 124 100 136 155 45 145 23 82 74 118 6 129 31 61 36
179 71 143 123 99 11 56 183 148 14 113 166 0 106 189 79 103

```

75 18 132 192 2 65 33 140 59 7 122 171 190 20 181 159 125 9
87 199 115 194 121 101 108 112 186 48 104 120 187 27 41 97
134 47 17 109 30 68 138 73 137 156 105 146 94 151 176 57
107 163 55 114 142 44 16 195 42 88 139 180 185 49 5 193 77
90 167 10 149 153 184 21 177 152 84 135 91 60 178 147 39
144 158 128 119 83 78 8 86 130 170 22 141 72 19 70 165 127
102 162 117 - Total length = 11573

New best solution found: Tour: 53 58 43 80 63 38 164 196 1
161 60 91 135 84 178 147 39 144 158 86 8 78 128 119 83 15
66 174 154 173 64 116 95 34 168 35 54 110 62 182 188 130
170 22 141 19 70 165 3 46 93 50 29 12 89 69 26 96 126 169
197 191 198 32 67 117 102 127 72 138 73 68 162 30 109 47
134 199 121 194 115 159 87 9 125 17 160 81 37 51 111 4 20
181 190 171 122 7 140 33 59 65 2 192 132 18 75 103 79 189
106 0 166 113 14 108 101 112 186 48 148 183 56 11 99 107
163 55 195 16 114 142 176 57 151 104 94 187 120 27 41 97
137 156 105 146 152 177 21 44 184 153 90 77 193 42 88 139
180 123 143 71 179 36 61 31 129 6 118 74 185 49 5 167 149
10 82 23 145 45 155 175 40 98 25 131 150 133 76 52 13 172
28 157 136 100 124 92 85 24 - Total length = 11411

New best solution found: Tour: 169 191 51 37 81 160 162 117
67 32 198 126 96 26 69 89 12 29 50 93 46 3 165 127 102 72
70 19 141 22 170 130 86 158 144 39 147 178 84 135 91 60 8
78 83 54 110 188 182 62 35 168 34 95 116 64 173 154 174 66
15 119 128 1 196 164 38 161 63 80 43 52 58 53 24 85 92 124
100 136 157 172 13 76 133 150 131 25 98 40 175 28 155 45
145 23 82 74 118 6 129 31 61 36 179 71 143 123 99 11 56 107
163 55 42 88 139 180 185 49 5 193 77 90 167 10 149 153 184
21 177 152 146 142 44 16 195 114 57 176 151 94 104 120 187
27 41 97 105 156 137 73 138 68 30 109 47 134 112 186 101
108 14 148 48 183 0 106 189 79 103 75 18 132 192 2 166 113
65 59 33 140 7 122 171 190 115 194 121 199 87 9 17 125 159
181 20 4 111 197 - Total length = 11344

New best solution found: Tour: 129 31 61 36 179 71 143 123
99 11 56 183 0 106 189 79 103 75 18 132 192 2 166 113 65 59

```
33 140 7 122 171 194 115 121 101 108 14 148 48 186 112 134  
47 109 30 73 138 68 162 102 117 67 198 32 51 37 81 160 17  
125 9 199 87 159 181 190 20 4 111 197 169 191 126 96 26 69  
89 12 29 3 165 127 72 70 19 141 46 93 50 182 62 188 110 54  
35 168 34 95 116 64 173 154 174 66 15 83 119 128 78 8 86  
158 144 130 22 170 39 147 137 156 105 146 97 41 27 187 120  
104 94 151 176 57 107 163 55 114 195 16 42 88 139 180 185  
49 5 193 77 90 167 153 184 21 44 142 152 177 84 178 135 91  
60 161 1 196 164 38 63 80 43 52 58 53 24 85 92 124 100 136  
157 76 13 133 150 172 28 131 25 98 40 175 155 45 145 23 82  
149 10 74 118 6 - Total length = 11299
```

```
Tour length after ILS = 11299; CPU time = 4.196s
```

Avec 200 sommets, 500 itérations et 10 perturbations:

```
〈onyr ★ kenzae〉 〈src〉» ./tspls 200 500 10  
New best solution found: Tour: 67 117 162 68 30 73 138 72  
102 127 69 89 12 29 3 165 70 19 141 46 93 50 188 182 62 110  
54 35 168 34 95 116 64 173 154 174 66 15 83 119 128 78 8 86  
158 144 130 22 170 39 147 137 156 105 97 41 27 120 187 94  
146 142 44 184 153 21 177 152 84 178 135 91 60 161 1 196  
164 38 63 80 43 58 52 76 13 133 150 131 25 98 40 175 28 172  
24 53 85 92 124 100 136 157 155 45 145 23 82 74 118 6 129  
31 61 36 179 71 143 189 79 103 75 18 192 132 106 0 56 11  
123 99 107 163 55 88 139 180 185 49 5 10 149 167 90 77 193  
42 16 195 114 176 151 57 183 148 48 104 186 112 101 108 14  
113 166 2 65 33 140 59 7 122 194 121 115 171 190 159 9 87  
199 134 47 109 17 125 181 20 4 111 197 169 191 126 26 96  
198 32 51 37 160 81 - Total length = 11641  
New best solution found: Tour: 67 32 198 126 96 26 89 69  
127 70 19 141 165 12 29 3 46 93 50 188 182 62 110 54 35 168  
34 95 116 64 173 154 174 66 15 83 119 128 78 8 158 86 130  
22 170 144 39 147 178 84 135 91 60 161 1 196 164 38 63 80  
43 52 58 53 85 92 24 124 100 136 157 28 172 13 76 133 150
```

131 25 98 40 175 155 45 145 23 82 74 118 6 129 31 61 36 179
71 143 123 99 11 56 107 163 55 42 88 139 180 185 49 5 10
149 167 90 77 193 16 195 114 57 176 142 44 184 153 21 177
152 146 105 156 137 97 41 187 120 94 151 104 48 186 112 101
108 14 148 183 0 106 189 79 103 75 18 132 192 2 166 113 65
33 140 59 7 122 194 121 115 171 190 20 4 111 197 169 191 51
37 81 160 125 181 159 87 199 9 17 109 47 134 27 30 68 73
138 72 102 162 117 - Total length = 11533

New best solution found: Tour: 67 32 198 126 191 169 197 51
37 81 160 17 125 159 181 4 111 20 190 171 122 7 140 33 59
65 113 166 2 192 132 18 75 103 79 189 106 0 183 48 148 14
108 101 186 112 121 194 115 87 9 199 134 47 109 30 68 138
73 137 156 105 146 97 41 27 187 120 104 94 151 176 57 55
163 107 56 11 99 123 143 71 179 36 61 31 129 6 118 74 82 23
145 45 155 175 40 98 25 131 150 133 76 13 172 28 157 136
124 100 149 10 167 90 5 49 185 180 139 88 42 193 77 16 195
114 142 44 184 153 21 177 152 84 135 91 60 92 85 24 53 58
52 43 80 63 161 38 164 1 196 15 66 174 154 173 64 116 95 34
168 35 54 110 83 119 128 78 8 86 158 144 178 147 39 170 22
130 188 62 182 50 93 46 29 3 165 12 89 26 96 69 127 70 19
141 72 102 162 117 - Total length = 11409

New best solution found: Tour: 67 32 198 126 191 169 197 51
37 81 160 17 125 9 87 199 115 159 181 4 111 20 190 171 194
121 122 7 140 33 59 65 113 166 2 192 132 18 75 103 79 189
106 0 183 56 11 99 123 143 71 179 36 61 31 129 6 118 74 82
23 145 45 155 175 40 98 25 131 150 133 13 172 28 157 136
100 124 92 85 24 53 58 52 76 43 80 63 38 164 196 1 161 60
91 135 178 84 152 177 21 184 153 149 10 167 90 5 49 185 180
139 88 42 193 77 44 142 114 16 195 55 163 107 57 176 151
104 186 48 148 14 108 101 112 134 47 109 30 97 41 27 120
187 94 146 105 156 137 147 39 170 22 130 144 158 86 8 78
128 119 83 15 66 174 154 173 64 116 95 34 168 35 54 110 188
62 182 50 93 46 29 3 165 12 89 26 96 69 127 72 70 19 141 73
138 68 162 102 117 - Total length = 11361

New best solution found: Tour: 162 68 73 138 72 127 165 70

```
19 141 3 29 12 89 69 26 96 126 191 169 197 111 4 20 181 159  
115 121 194 190 171 122 7 140 33 59 65 113 166 2 192 132 18  
75 103 79 189 106 0 183 148 14 108 101 112 186 48 104 94  
151 176 57 55 163 107 56 11 99 123 143 71 179 36 61 31 129  
6 118 74 49 5 185 180 139 88 42 193 77 90 167 149 10 82 23  
145 45 155 175 40 98 25 131 150 133 76 13 172 28 157 136  
100 124 92 85 24 53 58 52 43 80 63 38 164 196 1 161 128 119  
15 66 174 154 173 64 116 95 34 168 35 62 182 50 46 93 188  
110 54 83 78 8 86 158 144 130 22 170 39 147 178 60 91 135  
84 152 177 21 153 184 44 16 195 114 142 146 105 156 137 97  
41 187 120 27 134 47 199 87 9 125 17 109 30 160 81 37 51 32  
198 67 117 102 - Total length = 11327
```

```
New best solution found: Tour: 26 69 127 72 102 117 67 32  
198 96 126 191 169 197 111 51 37 81 160 4 20 181 159 125 17  
9 87 199 115 121 194 171 190 7 122 59 140 33 65 113 166 2  
192 132 18 75 103 79 189 106 0 183 148 14 108 101 112 186  
48 57 55 163 107 56 11 99 123 143 71 179 36 61 31 129 6 118  
74 82 23 145 45 155 175 40 98 25 131 150 133 76 13 172 28  
157 136 100 124 92 85 24 53 58 52 43 80 63 38 164 196 1 161  
60 91 135 178 84 152 177 21 44 184 153 90 167 149 10 5 49  
185 180 139 88 42 193 77 16 195 114 142 176 151 104 120 27  
187 94 146 41 97 134 47 109 30 162 68 138 73 137 156 105  
147 39 170 22 130 144 158 86 8 78 128 119 83 15 66 174 154  
173 64 116 95 34 168 35 54 110 62 182 188 50 93 46 141 19  
70 165 3 29 12 89 - Total length = 11308
```

Tour length after ILS = 11308; CPU time = 4.424s

On remarque l'importance d'avoir un nombre suffisant de perturbation afin d'explorer des voisinages suffisamment éloignés des minimums locaux obtenus.