INSTITUT NATIONAL

DE STATISTIQUE ET D'ECONOMIE APPLIQUEE

--*-*-*

PROBLEME DU VOYAGEUR DE COMMERCE

Realise par

FADLI HICHAM

ABDELHAKIM ZENIBI

Master's degree of Information Systems and Intelligent Systems

Supervised by : RACHID BENMANSOUR
COMPUTER SCIENCE DEPARTMENT
INSEA
MOROCCO
NOVEMBER 15, 2022

Abstract

TSP SOLVING BY DYNAMIC PROGRAMMING & HEURISTIQUE

Abstract

Travelling Salesman Problem (**TSP**) is one of the problems which is being widelyused in transportation industry which its optimization would speed up services andincrease customer satisfaction. The traveling salesman problem involves finding the shortest path that visits n specified locations, starting and ending at the same placeand visiting the other n-1 destinations exactly once. To the layman, this problem might seem a relatively simple matter of connecting dots, but that could not be further from the truth.

The **TSP** is one of the most significant problems in the history of applied mathematics. In 1952, three operations researchers (Danzig, Fulkerson, and Johnson, the first group to really crack the problem) successfully solved a TSP instance with 49 US cities to optimality.

Consequently, it is fair to say that the **TSP** has birthed a lot of significant combinatorial optimization research, as well as help us recognize the difficulty of solving discrete problems accurately and precisely.

The **TSP**'s wide applicability (school bus routes, home service calls) is one contributor to its significance, but the other part is its difficulty. It's an NP-hard combinatorial problem, and therefore there is no known polynomial-time algorithmthat is able to solve all instances of the problem. Some instances of the TSP can bemerely understood, as it might take forever to solve the model optimally.

Consequently, researchers developed heuristic algorithms to provide solutions that are strong, but not necessarily optimal. In this document, we will show the why andthe how of two methods for the **TSP**.

Keywords Travelling Salesman Problem , meheuristics, operations research, optimization

Résumé

Problème du voyageur de commerce (TSP) est l'un des problèmes largement utilisés dans l'industrie du transport dont l'optimisation accélérerait les services et augmenterait la satisfaction des clients. Le problème du voyageur de commerce consiste à trouver le chemin le plus court qui visite n emplacements spécifiés, en commençant et en terminant au même endroit et en visitant les autres n-1 destinations exactement une fois. Pour le profane, ce problème peut sembler une question relativement simple de relier des points, mais cela ne pouvait pas être plus éloigné de la vérité.

Le TSP est l'un des problèmes les plus significatifs de l'histoire des mathématiques appliquées. En 1952, trois chercheurs en opérations (Danzig, Fulkerson et Johnson, le premier groupe à vraiment résoudre le problème) ont résolu avec succès une instance TSP avec 49 villes américaines jusqu'à l'optimalité.

Par conséquent, il est juste de dire que le TSP a donné naissance à de nombreuses recherches importantes en optimisation combinatoire, et nous a aidés à reconnaître la difficulté de résoudre des problèmes discrets avec exactitude et précision.

La large applicabilité du TSP (routes d'autobus scolaires, appels de service à domicile) est l'un des facteurs de son importance, mais l'autre partie est sa difficulté. C'est un problème combinatoire NP-difficile, et donc il n'y a pas d'algorithme connu en temps polynomial qui soit capable de résoudre toutes les instances du problème. Certaines instances du TSP peuvent être simplement comprises, car la résolution optimale du modèle peut prendre une éternité

En conséquent, les chercheurs ont développé des algorithmes heuristiques pour fournir des solutions solides, mais pas nécessairement optimales. Dans ce document, nous allons montrer le pourquoi et le comment de deux méthodes pour le TSP.

Thanks to our beloved teacher **RACHID BENMANSOUR**

Table des matières

Liste des figures

Listes des tableaux

Introduction générale

Le problème du voyageur de commerce a été proposé par les mathématiciens Carl Menger et Hustler Wietni en 1930 . Le problème est qu'un voyageur de commerce veut visiter un grand nombre de villes et son but est de trouver le chemin le plus court, tel qu'il passe par toutes les villes, et chaque ville ne passe qu'une fois et revient finalement au point de départ. Un exemple de ce problème est illustré dans la figure ci-dessous.

Dans la première partie de cette figure, il y a 40 points qui montrent les villes et dans la partie "B", le chemin optimal que le vendeur doit emprunter pour visiter toutes les villes est représenté. TSP est un problème NP-difficile . L'ordre de complexité de ces problèmes est exponentiel ce qui n'a pas un temps d'exécution acceptable. Pour résoudre de tels problèmes, l'obtention de certaines solutions peut nécessiter beaucoup plus de temps que la durée de vie du système.

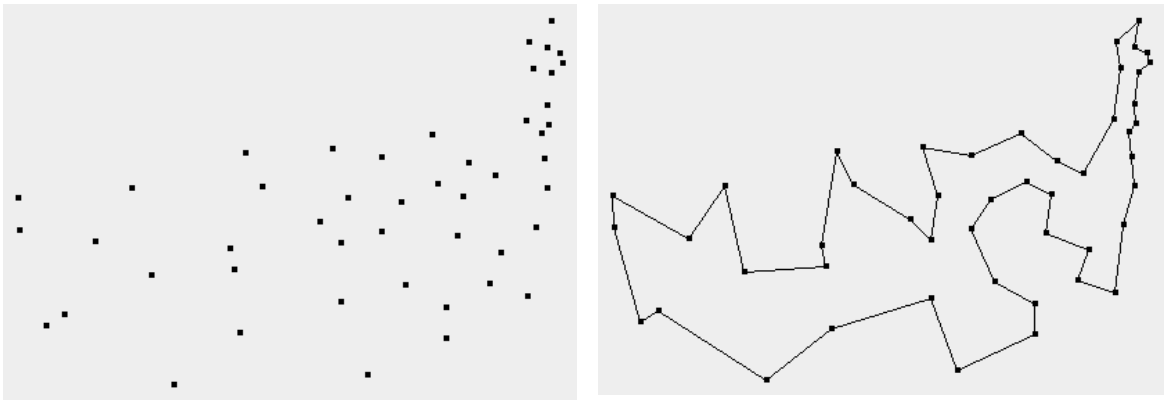

Figure 1 : Exemple de chemin optimal

Les solutions de programmation dynamique sont l'une des meilleures méthodes pour résoudre ces termes en matière de temps d'exécution et convertir l'ordre temporel du problème en forme polynomiale. Le problème des problèmes de programmation dynamique est leur consommation de mémoire où, dans les grands problèmes, le système ne peut pas répondre aux exigences de programmation dynamique.

La taille et la complexité des problèmes d'optimisation comme le voyageur de commerce et les problèmes du monde réel ont attiré l'attention des chercheurs sur les algorithmes métaheuristiques et l'investigation locale pour s'inspirer de l'intelligence sociale des créatures. Les algorithmes métaheuristiques tentent d'obtenir des résultats logiques dans un temps acceptable en consommant un minimum de mémoire.

Plusieurs algorithmes ont été proposés pour résoudre le TSP. la plupart de ces algorithmes sont des algorithmes métaheuristiques et calculent la solution approchée en un temps très court. Le problème du voyageur de commerce est très similaire au routage des véhicules. Plusieurs algorithmes sont proposés pour résoudre le problème du routage de véhicules qui peuvent être utilisés pour résoudre le TSP.

Chapitre 1 :
Programmation dynamique et heuristique

1.1 Programmation dynamique

La programmation dynamique est une approche ou méthode de résolution de problèmes, elle a été introduite par *Richard Bellman* dans les années 1950s et elle a rapidement trouvée des utilisations dans différents domaines. Le principe de cette méthode est de décomposer un problème en une suite de sous-problèmes simples à résoudre et à partir des solutions de chaque sous problème on retrouve la solution du problème initial.

Les problèmes qui peuvent être résolus par la programmation dynamique sont ceux qui suivent le principe d'optimalité i.e la solution optimale du problème peut être construite à partir des solutions optimales de ses sous-problèmes.

Exemple d'application de la programmation dynamique :

Prenons l'exemple de la suite de Fibonacci, et cherchons à trouver un algorithme qui permet de retourner la valeur du nième terme de cette suite.

On peut faire ce travail en utilisant l'algorithme suivant :

```
function fibonacci(n) :

if n <= 1 Then
  return n

return fibonacci(n-1) + fibonacci(n-2)
```

Figure 2 :La suite de fibonacci

Maintenant prenons l'exemple de n=5, voici l'arbre des appels récursifs :

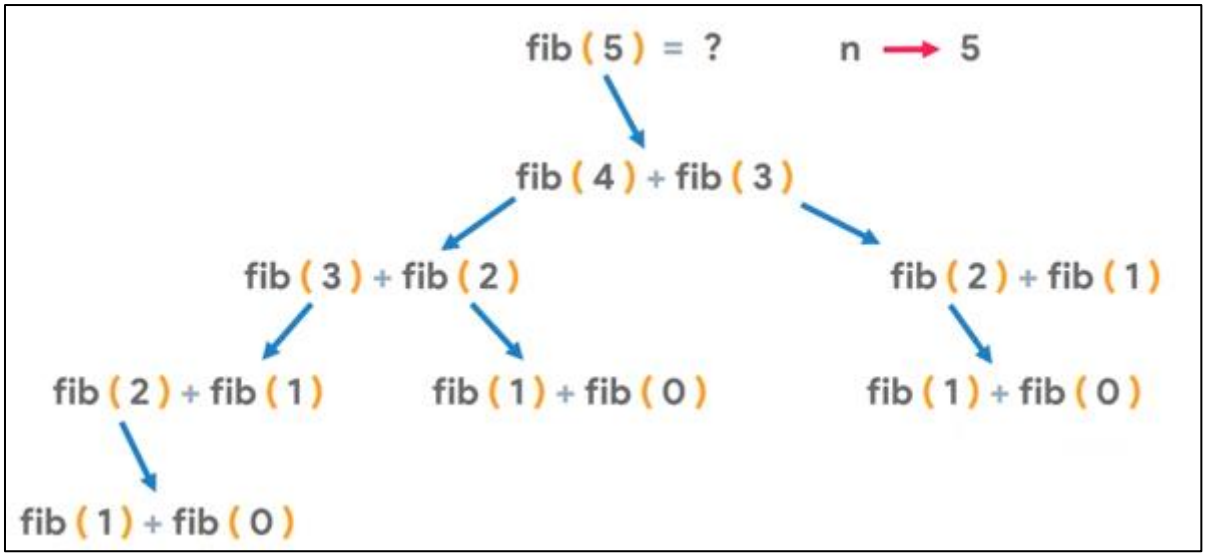

Figure 3:L'arbre des appels de fibonacci

On remarque que même si on a déjà calculé la valeur de fibonacci(2) dans le sous arbre gauche, on a refait l'appel dans le sous arbre droit, et cela n'est pas assez pratique lorsque n est grand. D'où vient la puissance de la programmation dynamique, car il suffit de faire le calcule de fibonacci(2) une seule fois et dans pour chaque n>2 on ne va plus faire le calcule de cette valeur car on va l'utiliser directement.

Voici un algorithme appliquant la programmation dynamique pour faire la même chose mais en réduisant considérablement le nombre d'appels récursifs :

```
tab ← [0,1]

fibonacci(n, tab):

  if n in tab Then:
    return tab[n]

  tab[n] ← fibonacci(n-1, tab) + fibonacci(n-2, tab)

  return tab[n]
```

Figure 4:Algorithme Fibonacci par Programmation Dynamique

Ici le tableau tab va stocker les valeurs calculées pour en servir dans les autres appels, on a initialisé par 0 et 1 qui sont les condition initiales de fibonacci : fibonacci(0) = 0 et fibonacci(1) = 1.

En général, une formulation d'une solution utilisant la programmation dynamique doit contenir les éléments suivants :

1. La fonction objectif
2. Une relation de récurrence
3. Les conditions initiales (ou conditions d'arrêts)

Dans la programmation dynamique, on distingue entre la programmation dynamique en avant (forward dynamic programming) qui commence par le plus petit sous problème et ascend vers le problème initial à résoudre, et la programmation dynamique vers l'arrière (backward dynamic programming) qui commence par le problème initial et descend vers les sous-problèmes.

Algorithme de programmation dynamique utilisé pour résoudre le problème de ce projet :

Dans ce document, cet algorithme est utilisé pour résoudre le TSP. Compte tenu du principe d'optimalité et de la programmation dynamique, il convient de noter quel sous problème convient à ce problème. On suppose que nous sommes partis de la ville 1 et que quelques villes ont été visitées et maintenant nous sommes dans la ville j. La meilleure ville doit être sélectionnée en considérant qu'elle n'a pas été visitée auparavant.

Pour un sous-ensemble de villes $\boldsymbol{S} \subseteq \{\boldsymbol{1}, \boldsymbol{2}, \ldots, \boldsymbol{n}\}$, la longueur du plus court chemin parcouru en S est représentée par $\boldsymbol{C}(\boldsymbol{S}, \boldsymbol{j})$ qui a commencé à partir de la ville 1 et s'est terminé à la ville j. Dans cette méthode,

$\boldsymbol{C}(\boldsymbol{S}, \boldsymbol{1}) = \infty$, car le chemin n'a pas pu commencer à partir de la ville 1 et se terminer à la ville 1.

Maintenant, les sous-problèmes doivent être développés pour résoudre le problème principal. Après avoir visité la ville j, la ville i doit être visitée, ce qui est calculé en fonction de la fonction de longueur représentée dans cette équation.

$$\boldsymbol{C}(\boldsymbol{S}, \boldsymbol{j}) = \boldsymbol{min}(\boldsymbol{C}(\boldsymbol{S}, \boldsymbol{j}) + \boldsymbol{dij})\ \boldsymbol{i} \in \boldsymbol{S}, \boldsymbol{i} \neq \boldsymbol{j}$$

Dans cette équation, $\boldsymbol{C}(\boldsymbol{S}, \boldsymbol{j})$ est la longueur du chemin entre la ville 1 et la ville j et dij est la longueur du chemin entre la ville i et j.

1. $C(\{1\},1) = 0$
2. $for\ s\ =\ 2\ to\ n$
3. $\quad for\ all\ subsets\ S \in \{1,2,\ldots,n\}\ of\ size\ s\ and\ containing\ 1$
4. $\quad\quad C(S,1) = \infty$
5. $\quad\quad for\ all\ j \in S, j \neq 1$
6. $\quad\quad\quad C(S,j) = min\{C(S-\{j\},i) + d_{ij} :\ i \in S, i \neq j\}$
7. $return\ min_{j \neq 1}\ [C(\{1,\ldots,n\},j) +$

Figure 5 :le pseudo-code de l'algorithme de programmation dynamique pour TSP.

Le nombre de sous-problèmes dans cette méthode est 2n*n et chaque sous-problème peut être résolu en un temps dans un ordre linéaire. Par conséquent, le coût de cette méthode est égal à O(n2×2n).

1.2 Heuristique

Une heuristique est une technique conçue pour résoudre plus rapidement des problèmes lorsque les méthodes classiques sont trop lentes, ou pour trouver une solution approximative lorsque les méthodes classiques ne parviennent pas à trouver une solution exacte , Ceci est réalisé en échangeant l'optimalité, l'exhaustivité, l'exactitude ou la précision contre la vitesse.

Généralement une heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre sans offrir aucune garantit quant à la qualité de la solution calculée. Les heuristiques peuvent être classées en deux catégories :

– Méthodes constructives qui génèrent des solutions à partir d'une solution initiale en essayant d'en ajouter petit à petit des éléments jusqu'à ce qu'une solution complète soit obtenue,

– Méthodes de fouilles locales qui démarrent avec une solution initialement complète (probablement moins intéressante), et de manière

répétitive essaie d'améliorer cette solution en explorant son voisinage.

Algorithme de programmation dynamique utilisé pour résoudre le problème de ce projet :

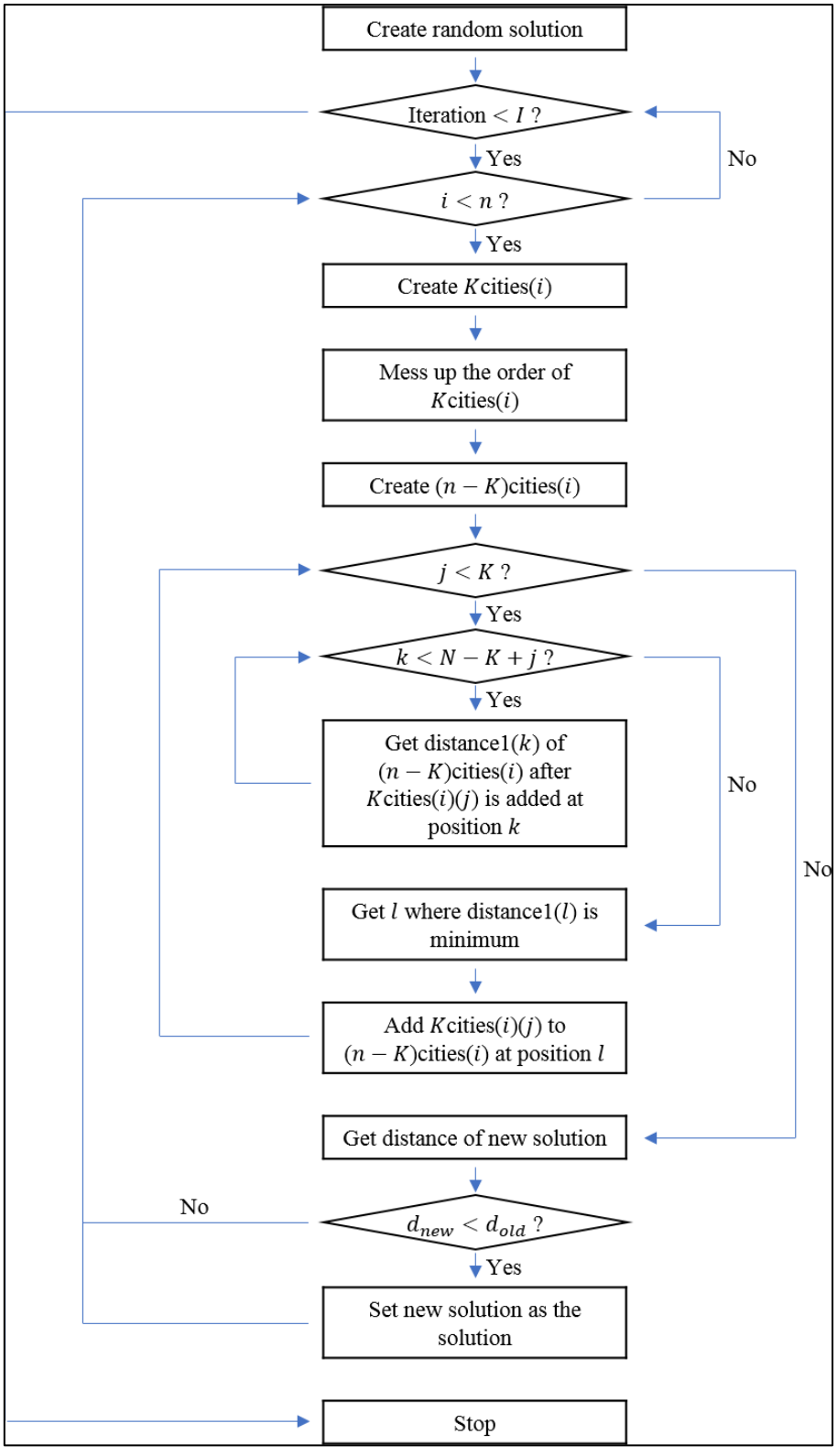

Figure 6:Algorithme de programmation dynamique utilisé pour résoudre le problème

Chapitre 2

Implementation des solutions pour le problème TSP:

Dans le chapitre 1 on a parlé et de présenté les algorithmes de programmation dynamique et de métaheuristique , ainsi que l'algorithme de Heuristique, qu'on va utiliser pour résoudre ce problème, dans ce chapitre on va parler de l'implémentation de ces algorithmes en utilisant python pour résoudre les instances du TSP.

2.1 Programmation dynamique

Pour implementer cette algorithme, on a besoin d'une fonction qui permet d'extraire les données d'après les fichiers, et les ranger dans une liste imbriqué, il s'agit de la fonction read_data_1

```
@staticmethod
def read_data_1(path,instance):
    df = pd.read_excel(path,sheet_name=instance)
    df = df.drop('Unnamed: 0', axis=1)
    matr = df.values.tolist()
    matr = np.triu(matr) + np.triu(matr,1).T
    return matr
```

Figure 7:fonction qui permet d'extraire les donnees

En appliquant cette fonction au fichier qui contient l'instance 1 on obtient le résultat suivante:

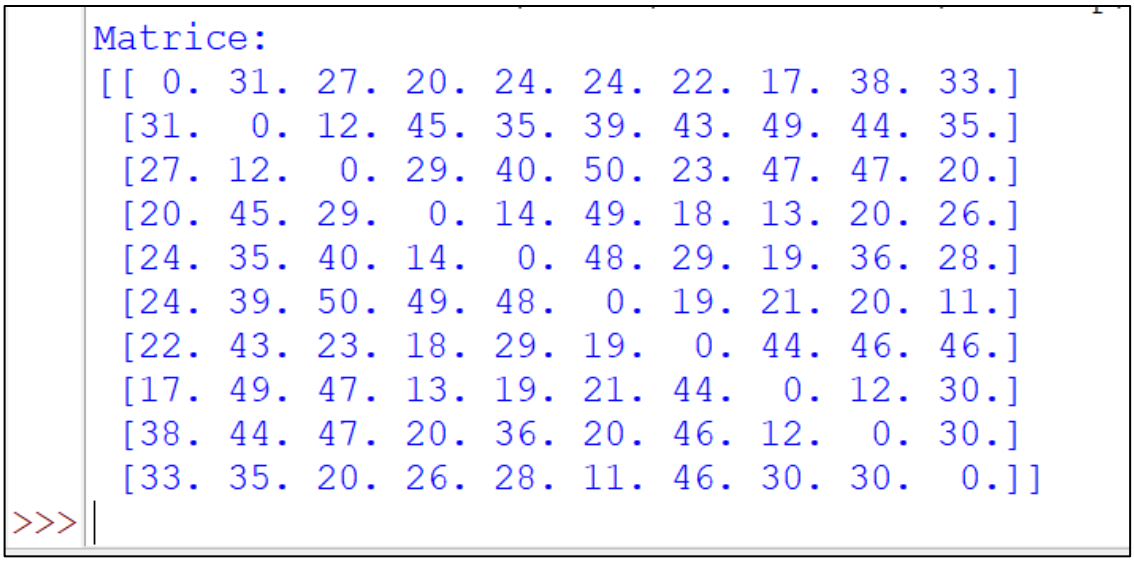

Figure 8:Resultat de extraire l'instance 1

Après on va utiliser la fonction read_data_1 dans les methodes de la classe Travelling Salesman Problem pour initialiser et manipuler les données de fichier.exel afin de trouver le résultat :

```
def solve_tsp_dynamic_programming(first_city,path,instance) -> Tuple[List, int]:
    instance=read_data_1(path,instance)
    N = frozenset(range(0, len(instance))))
    N = N.différence({première_ville-1})
    mémo : Dict[tuple, int] = {}

    # Step 1: get minimum distance
    @lru_cache(maxsize=len(instance)**2)
    def dist(ni, N: frozenset):
        if not N:
            return instance[ni][first_city-1]

        # Store the costs in the form (nj, dist(nj, N))
        costs = [
            (nj, instance[ni][nj] + dist(nj, N.difference({nj})))
            for nj in N
        ]
        nmin, min_cost = min(costs, key=lambda x: x[1])
        memo[(ni, N)] = nmin
        return min_cost

    best_distance = dist(first_city-1, N)

    # Step 2: get path with the minimum distance
    ni = first_city-1
    solution = [first_city]
    while N:
        ni = memo[(ni, N)]
        solution.append(ni+1)
        N = N.difference({ni})
    solution.append(first_city)
    return solution, best_distance
```

Figure 9:La fonction objectif

Cette fonction prend en paramètre l'emplacement de l'instance (path) qui contient

les données des distances et le numero de l'instance (instance) et l'indice de ville de depart (start),et retourner les variables suivantes :

- **best_distance** : Une permutation de nœuds de 1 à n qui produit la plus petite distance totale.
- **solution** : Le moins de distance parcourue entre les villes visites

Cette fonction dirise par deux objectifs principale le calcul de coût du chemin optimal et le extraire leur chemin

1. Algorithme : coût du chemin optimal

Considérez une instance TSP avec 3 nœuds : {0, 1, 2}. Soit dist(0, {1, 2}) le distance de 0, en visitant tous les nœuds de {1, 2} et en revenant à 0. Cela peut être calculé récursivement comme :

dist(ni, N) = min (c_{ni, nj} + dist(nj, N - {nj}))) /nj dans N

et

dist(ni, {}) = c_{ni, 0}

Avec le point de départ comme dist(0, {1, 2, ..., tsp_size}). La notation N - {nj} est l'opérateur de différence, c'est-à-dire l'ensemble N sans nœud nj.

2. Algorithme : calcul du chemin optimal

Le processus précédent renvoie la distance du chemin optimal. Pour trouver le chemin réel, nous devons stocker dans une mémoire les clés/valeurs suivantes :

mémo[(ni, N)] = nj_min

On va test la fonction **solve_tsp_dynamic_programming** et le resultat suivante :

```
OPTIMAL POLICY : [1, 2, 3, 10, 6, 9, 8, 5, 4, 7, 1]
OPTIMAL VALUE : 179.0


time : 1.91477632522583 s
```

Figure 10:Resultat de Execution dynamique

2.1 Heuristique

L'algorithme k-opt est la méthode heuristique la plus populaire pour le TSP, où k signifie le nombre d'échanges/repermutations de bords dans certains voisinages. Cependant, les approches k-opt existantes ne sont pas adaptatives, et le nombre k est soit fixe, soit sondé séquentiellement en suivant un certain ordre de sondage.

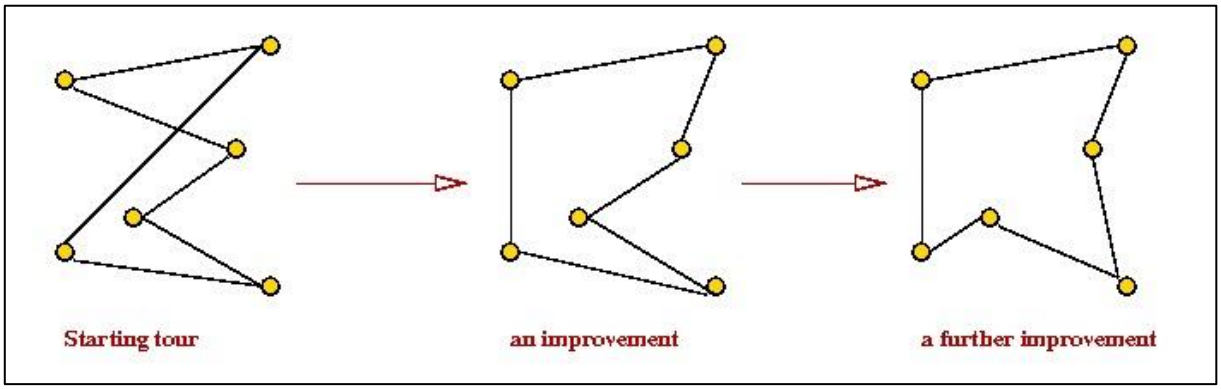

Figure 11:shema presente la methode heuristique

Dans la pratique, les algorithmes d'amélioration de tour les plus utilisés sont probablement le 2-opt et le 3-opt. (k est fixé à 2 et 3, respectivement.) Les méthodes à 2 ou 3 options sont avantageuses en ce que la mise en œuvre est relativement facile et les méthodes peuvent être terminées à tout moment (algorithmes à tout moment).

Code :

```
def optimise(self):
        better = True
        self.solutions = set()
        self.neighbours = {}
        for i in self.heuristic_path:
            self.neighbours[i] = []
            for j, dist in enumerate(TSP.edges[i]):
                if dist > 0 and j in self.heuristic_path:
                    self.neighbours[i].append(j)
        while better:
            if time.time() < timeout:
                better = self.improve()
                self.solutions.add(str(self.heuristic_path))
                print(self.heuristic_cost)
            else:
                break
        self.save(self.heuristic_path, self.heuristic_cost)
```

Figure 12 :La fonction optimise

cette fonction comparait entre la distance de séquence de x et x-1 si la distance séquence x et optimal continuel boucle jusqu'à trouver séquence optimale où le temps presse sinon retourne la séquence de x-1

```
def improve(self):
        tour = Tour(self.heuristic_path)
        for t1 in self.heuristic_path:
            around = tour.around(t1)
            for t2 in around:
                broken = set([makePair(t1, t2)])
                # Initial savings
                gain = TSP.dist(t1, t2)
                close = self.closest(t2, tour, gain, broken, set())
                tries = 5
                for t3, (_, Gi) in close:
                    if t3 in around:
                        continue
```

```
                joined = set([makePair(t2, t3)])
                Log.debug("Start: X: {} - Y: {} = {}".format((t1, t2),(t2, t3), Gi))
                if self.chooseX(tour, t1, t3, Gi, broken, joined):
                    return True
                tries -= 1
                if tries == 0:
                    break
        return False
```

Figure 13:La fonction improve

Chapitre 3

Comparaison entre les méthodes pour les deux problèmes

Dans ce chapitre, on va appliquer l'implémentation de la programmation dynamique, la heuristique présentés dans les chapitres précédents pour résoudre les instances de problème du voyageur de commerce. les spécifications de l'ordinateur utilisé sont:

Processeur	Intel(R) Core(TM) i3-6100U CPU @ 2.30GHz 2.30 GHz
Mémoire RAM installée	8,00 Go (7,88 Go utilisable)

Tableau 1:Resultat par programmation dynamique

instance	Distance optimal	séquence	temps
Instance_1.xlsx	179	[1, 2, 3, 10, 6, 9, 8, 5, 4, 7, 1]	1.91s
Instance_2.xlsx	339	[1, 7, 15, 8, 19, 12, 3, 13, 17, 6, 4, 16, 10, 2, 14, 11, 18, 20, 5, 9, 1]	2.35s
Instance_3.xlsx	-------	--------	-------

Tableau 2:Résultats d'heuristique

Instance	Distance optimal	Séquence	temps
Instance_1.xlsx	179	[1, 7, 4, 5, 8, 9, 6, 10, 3, 2, 1]	0.20 s
Instance_2.xlsx	339	[1, 7, 15, 8, 19, 12, 3, 13, 17, 6, 4, 16, 10, 2, 14, 11, 18, 20, 5, 9, 1]	0.27 s
Instance_3.xlsx	1836	[1, 24, 46, 13, 37, 12, 9, 30, 45, 11, 27, 28, 15, 36, 7, 50, 29, 47, 10, 31, 34, 14, 2, 19, 42, 26, 39, 43, 23, 35, 38, 17, 22, 18, 48, 49, 3, 8, 41, 20, 40, 4, 33, 25, 16, 44, 21, 5, 6, 32, 1]	1.66 s

Comparaison:

Pour l'heuristique, les résultats dans le tableau sont approximatifs et ils ne sont pas les mêmes à chaque fois qu'on exécute le code, on a mis dans le tableau la plus petite valeur atteinte après plusieurs exécutions. Mais pour la programmation dynamique le résultat est exact et on obtient le même résultat chaque fois.

D'autre part, pour notre implémentation, la programmation dynamique ne peut pas résoudre les instances de taille supérieures à 25 (Dans le même ordinateur dont les spécifications sont notées ci-dessus); la RAM se remplit et le programme cesse de fonctionner.

Tandis que l'heuristique peut résoudre des instances de grandes tailles mais toujours on aura juste des résultats approximatifs et le temps d'exécution augmente lorsque la taille de l'instance augmente.

On a essayé d'exécuter le programme utilisant la programmation dynamique dans le cloud pour voir si on peut aller vers des tailles plus grandes que 25 est combien de temps il va prendre. Pour cela on a utilisé google colab qui donne 12Go de RAM et voici le résultat de cette **expérience :**

- On a essayé avec n=50 mais la RAM se remplit et la session se termine
- On a essayé avec n=30 et encore le même problème
- On a essayé avec n=25 et encore le même problème

On remarque que même avec 12Go de RAM on ne peut pas résoudre des instances de taille supérieure à 25.

Conclusion :

Après on a présenté les différents résultats pour chaque méthode à savoir les valeurs optimales, les séquences optimales et les temps d'exécution. Et en utilisant ces résultats on a fait une comparaison entre ces méthodes et on n'a conclu que:

- La programmation dynamique n'est pas efficace pour les instances de grande taille
- La programmation dynamique donne des résultats exacts pour les instances qu'elle peut résoudre
- L'heuristique est intéressante car elle permet de résoudre des instances de grande taille
- Mais, les solutions trouvées par l'heuristique ne sont pas exactes et elles différents d'une exécution à l'autre

Conclusion

Il est vrai que la programmation dynamique donne une solution exacte, mais elle est toujours limitée en matière de capacité de mémoire, donc pour résoudre un problème avec des données massives, les solutions accessibles sont les meilleures, dans notre cas nous avons utilisé l'heuristique qui n'est pas limitée comme la programmation dynamique, mais nous n'avons aucune garantie qu'elle donne la solution exacte (valeur optimale).

Bibliography

[1] **Meta-Heuristic Approaches for Solving Travelling Salesman Problem**
ElhamDamghanijazi, Arash Mazidi

[2]http://www.fsr.ac.ma/DOC/cours/maths/Souad%20Bernoussi/Cours%20C2SI.pdf

[3] https://iopscience.iop.org/article/10.1088/1742-6596/1550/3/032027