Institut National de Statistique et d'Economie Appliquée

Master Systèmes d'informations et systèmes intelligents

Traveling Salesmen Problem Solving by Dynamic Programming & Heuristic

Minimiser une fonction objective en utilisant la programmation dynamique et l'heuristique (Algorithme 2-opt) : Comparaison entre ces deux techniques

Realiser Par:

- **MOUMAD Hamza**
- **MIRI Taha-Amine**

Encadre Par :

- **Pr. Rachid BENMANSOUR**

05/11/2022

Année Académique: 2022-2023

Table des matières

Table des figures

RESUME

Le problème du voyageur de commerce (TSP) est un problème classique d'optimisation combinatoire NP-complet. Des heuristiques ont été proposées permettant de trouver des solutions presque optimales. On distingue deux classes d'heuristiques : de construction et d'amélioration. LinKernighan est la meilleure heuristique d'amélioration. L'heuristique n-opt est une méthode composite de construction et d'amélioration. Nous proposons une hybridation de la méthode Heuristique. Nous proposons aussi la solution du TSP par la programmation dynamique, afin de faire une comparaison entre les deux méthodes à la fin de ce projet

MOTS-CLÉS : TSP, heuristique, programmation dynamique, 2-opt Algorithme

ABSTRACT

The travelling salesman problem (TSP) is NP-hard classical optimisation problem. There are heuristics that give near optimal solutions. These heuristics are classified into two categories: for construction and for improvement. Lin-Kernighan is known as the best construction heuristic while Colony of Ants is composite mixing construction and improvement. We propose to hybridize of Heuristic method. We also propose the solution of TSP by dynamic programming, in order to make a comparison between the two methods at the end of this project.

KEYWORDS : TSP, heuristic, Dynamic Programming, 2-opt Algorithm

INTRODUCTION

En informatique actuel, **Le problème du voyageur de commerce** (*en Anglais Traveling Salesman Problem*), étudié depuis le 19eme siècle, est l'un des plus connus dans le domaine de la recherche opérationnelle. Jouez à trouver le meilleur parcours possible... et découvrez différentes méthodes informatiques proposées pour résoudre ce problème.

Le problème du voyageur de commerce, ou problème du commis voyageur, est un problème d'optimisation qui consiste à déterminer, étant donné une liste de villes et les distances entre toutes les paires de villes, le plus court-circuit qui passe par chaque ville une et une seule fois.

Malgré la simplicité de l'énoncé, on ne connaît pas d'algorithme permettant de trouver une solution exacte rapidement dans tous les cas. Plus précisément, on ne connaît pas d'algorithme en temps polynomial, et la version décisionnelle du problème du voyageur de commerce *(pour une distance D, existe-t-il un chemin plus court que D passant par toutes les villes et qui termine dans la ville de départ ?)* est un problème NP-complet, ce qui est un indice de sa difficulté.

C'est un problème algorithmique très célèbre, qui a donné lieu à de nombreuses recherches et qui est souvent utilisé comme introduction à l'algorithmique ou à la théorie de la complexité. Il présente de nombreuses applications, que ce soit en planification, en logistique ou dans des domaines plus éloignés, comme la génétique (*en remplaçant les villes par des gènes et la distance par la similarité*).

Les domaines d'application du **TSP** sont nombreux : problèmes de logistique, de transport aussi bien de marchandises que de personnes, et plus largement toutes sortes de problèmes d'ordonnancement. Certains problèmes rencontrés dans l'industrie se modélisent sous la forme d'un problème de voyageur de commerce, comme l'optimisation de trajectoires de machines-outils : comment percer plusieurs points sur une carte électronique le plus vite possible ?

Ce document est structuré comme suit : nous allons d'abord couvrir certaines bases théoriques afin de mieux comprendre le problème, définir notre problème et son environnement, comparer les différents algorithmes qui s'offrent à nous, implémenter le plus adéquat, puis observer et comparer les résultats qu'il donne dans l'environnement de déploiement.

I. Chapitre 1 : Présentation Générale du problème

Le problème du voyageur de commerce peut être modélisé à l'aide d'un graphe constitué d'un ensemble de sommets (*qui représente l'ensemble des villes*) et d'un ensemble d'arêtes (*qui représente les distances entre les villes*). Chaque sommet représente une ville, une arête symbolise le passage d'une ville à une autre, et on lui associe un poids pouvant représenter une distance, un temps de parcours ou encore un coût. Ci-contre, un exemple de graphe à 4 sommets.

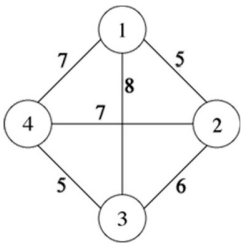

Figure 1 - graphe à 4 sommets (villes).

Résoudre le problème du voyageur de commerce revient à trouver dans ce graphe un cycle passant par tous les sommets une unique fois (un tel cycle est dit **« hamiltonien »**) et qui soit de longueur minimale. Pour le graphe ci-contre, une solution à ce problème serait le cycle 1, 2, 3, 4 et 1, correspondant à une distance totale de 23. Cette solution est optimale, il n'en existe pas de meilleure.

Comme il existe une arête entre chaque paire de sommets, on dit que ce graphe est « complet ». Pour tout graphe, une matrice de poids peut être établie. En lignes figurent les sommets d'origine des arêtes et en colonnes les sommets de destination ; le poids sur chaque arête apparaît à l'intersection de la ligne et de la colonne correspondantes. Pour notre exemple, cette matrice est la suivante :

$$\begin{pmatrix} 0 & 5 & 8 & 7 \\ 5 & 0 & 6 & 7 \\ 8 & 6 & 0 & 5 \\ 7 & 7 & 5 & 0 \end{pmatrix}$$

Dans cet exemple, le graphe est « non orienté », c'est-à-dire qu'une arête peut être parcourue indifféremment dans les deux sens, cela explique que la matrice soit symétrique. Cette symétrie n'est pas forcément respectée dans le cas d'un graphe orienté. Il existe alors deux catégories de problèmes : le cas symétrique (le poids de l'arc du sommet *X* vers *Y* est égal au poids de l'arc du sommet *Y* vers *X*) et le cas asymétrique (le poids de l'arc du sommet *X* vers *Y* peut être différent du poids de l'arc du sommet *Y* vers *X*).

1. Applications du problème du voyageur de commerce

Le TSP trouve toujours des applications dans tous les secteurs verticaux. Des solutions efficaces trouvées grâce au TSP sont utilisées en astronomie, en informatique et même en routage de véhicules.

Le problème du voyageur de commerce peut également être utilisé pour savoir comment un laser doit se déplacer lorsqu'il perce des points dans une carte de circuit imprimé. Les astronomes utilisent les concepts de TSP pour déterminer le mouvement d'un télescope sur la distance la plus courte entre différentes étoiles dans une constellation.

Parmi les autres utilisations du TSP, citons la planification Internet, l'agriculture, la production de micropuces et l'art généré par ordinateur. Vous voulez voir par vous-même comment Route4Me peut augmenter vos profits ? Que vous souhaitiez réduire le temps qu'il vous faut pour planifier les itinéraires de vos chauffeurs, augmenter le nombre d'arrêts qu'ils peuvent effectuer ou satisfaire vos clients en sachant que vos chauffeurs se présentent à l'heure…

2. Méthodes de résolution du problème

Comme pour le **problème du sac à dos** (un autre des problèmes les plus connus dans le domaine de l'optimisation combinatoire et de la recherche opérationnelle), il existe deux grandes catégories de méthodes de résolution : les méthodes exactes et les méthodes approchées.

Les méthodes exactes permettent d'obtenir une solution optimale à chaque fois, mais le temps de calcul peut être long si le problème est compliqué à résoudre.

Les méthodes approchées, encore appelées heuristiques, permettent quant à elles d'obtenir rapidement une solution approchée, mais qui n'est donc pas toujours optimale.

II. Chapitre 2 : Programmation dynamique & Heuristique

3. Programmation Dynamique

a. Introduction à la programmation dynamique

La programmation dynamique est une approche ou méthode de résolution de problèmes, elle a été introduite par *Richard Bellman* dans les années *1950* 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 c-à-d la solution optimale du problème peut être construite à partir des solutions optimales de ses sous-problèmes.

b. Principe de la programmation dynamique

La programmation dynamique travaille sur une formulation récursive du problème, pour laquelle on fait de la tabulation des résultats intermédiaires. Prenons par exemple, le calcul de la suite de Fibonacci.

```
function fibo(n) {
    if (n <= 1) {
        return 1
    } else {
        return (fibo(n-1) + fibo(n-2))
    }
}
```

Nous remarquons rapidement que la **complexité algorithmique** de cette fonction est **exponentielle** — **O(e^n)** c'est **déplorable** algorithmiquement. La raison de cette inefficacité se résume en fait à la **multiplicité de calcul d'un même nombre**. Nous nous appuyons sur le schéma suivant pour illustrer cela :

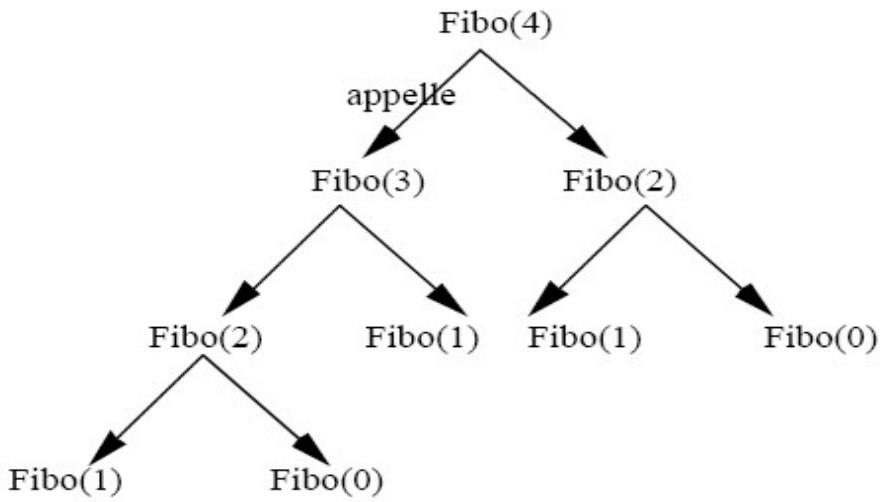

Figure 2 - L'arbre des appels récursifs à la fonction Fibo

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

1. La fonction objective.
2. La relation de récurrence.
3. Les conditions initiales.

c. L'algorithme de la programmation dynamique

```
function algorithm TSP (G, n) is

    for k := 2 to n do
        g({k}, k) := d(1, k)
    end for

    for s := 2 to n-1 do
        for all S ⊆ {2, ..., n}, |S| = s do
            for all k ∈ S do
                g(S, k) := min_{m≠k, m∈S} [g(S\{k}, m) + d(m, k)]
            end for
        end for
    end for
    opt := min_{k≠1} [g({2, 3, ..., n}, k) + d(k, 1)]
    return (opt)
end function
```

d. Implémentation par Python

```python
def tspDynamic(
    distance_matrix: np.ndarray,
    maxsize: Optional[int] = None,
) -> Tuple[List, float]:

    # Get initial set {1, 2, ..., tsp_size} as a frozenset because @lru_cache
    # requires a hashable type
    N = frozenset(range(1, distance_matrix.shape[0]))
    memo: Dict[Tuple, int] = {}

    # Step 1: get minimum distance
    @lru_cache(maxsize=maxsize)
    def dist(ni: int, N: frozenset) -> float:
        if not N:
            return distance_matrix[ni, 0]

        # Store the costs in the form (nj, dist(nj, N))
        costs = [
            (nj, distance_matrix[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(0, N)

    # Step 2: get path with the minimum distance
    ni = 0  # start at the origin
    solution = [0]
    while N:
        ni = memo[(ni, N)]
        solution.append(ni)
        N = N.difference({ni})

    return solution, best_distance
```

e. Explication du programme

❖ La fonction prend en paramètre une matrice, cette matrice et générer automatiquement à partir du fichier Excel {Instance 1, 2 ou 3}.

- Au sein de la fonction principale on déclare une autre fonction qui calcule la distance entre deux villes de notre système de villes.
- En suit, stocker ces distances dans une liste, et calculer le minimum de cette liste.
- Finalement la fonction retourne le cout minimal (la distance minimal) et le chemin le plus court pour parcourir toutes les villes.

Fonction principale

```
def Tsp(instance):
    fileName = os.path.join( os.getcwd(), '../projet', 'tsp-maroc.xlsx' )
    path = filename

    df = pd.read_excel(path, sheet_name=instance)
    df = df.replace(np.nan, 0)
    df.drop(['Unnamed: 0'], axis=1, inplace=True)
    df = df.astype(int)
    matrix = df.to_numpy()

    if instance == 'instance_1':
        distance_matrix = np.maximum(matrix, matrix.transpose())
    else:
        distance_matrix = matrix

    start_time = time.time()
    permutation, distance = tspDynamic(distance_matrix)
    end_time = time.time()
    timing = end_time - start_time

    return {'result': permutation, 'cost': distance, 'Mytime': timing}
```

- La fonction **TSP()** prend en paramètre une chaine de caractère qui stock le numéro de l'instance [exemple : Instance_1]
- La lecture du fichier Excel, puis générer une matrice à partir du nom de l'instance obtenu en paramètre.
- On fait appel à la fonction précédente pour calculer la distance et le chemin les plus courts. À l'entrer et la sortie de la fonction on prend le temps du système, finalement en calcule le temps d'exécution de la fonction dynamique = temps sortie – temps d'entrer

4. Méthode Heuristique

f. Principe de l'algorithme 2-opt

2-opt est un algorithme itératif : à chaque étape, on supprime deux arêtes de la solution courante et on reconnecte les deux tours ainsi formés. Cette méthode permet, entre autres, d'améliorer le coût des solutions en supprimant les arêtes sécantes lorsque l'inégalité triangulaire est respectée (voir figure ci-contre).

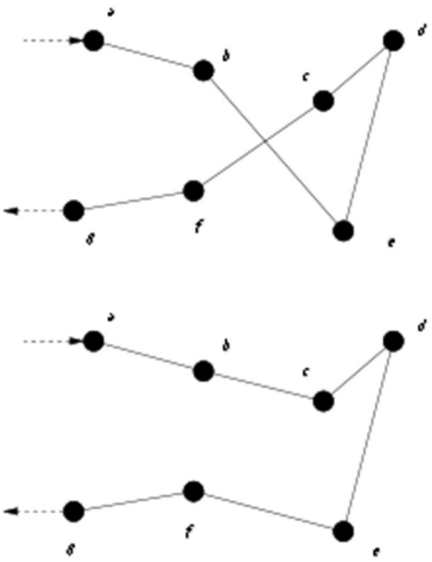

Figure 3 - Exemple de permutation 2-opt

Sur le schéma de droite, la route *<a; b; e; d; c; f; g>* est changée en *<a; b; c; d; e; f; g>* en inversant l'ordre de visite des villes e et c. Plus généralement, lorsqu'on inverse l'ordre de parcours de deux villes, il faut aussi inverser l'ordre de parcours de toutes les villes entre ces deux villes.

g. Formalisation de l'algorithme

Dans le cas du problème du voyageur de commerce géométrique, l'algorithme fonctionne par la procédure suivante :

- Trouver une 2-permutation de H produisant le cycle H' tel que coût(H') < coût(H).
- Remplacer H par H'.
- Recommencer tant qu'une telle 2-permutation est possible.

Pour des raisons de performance, la solution initial H est souvent générée par une heuristique constructiviste ou gloutonne rapide, voire aléatoirement. On peut noter que diverses stratégies peuvent être appliquées lors du choix de la permutation : ce peut être la première trouvée, la meilleure ou la pire au regard de la tournée courante, ou encore un choix aléatoire parmi un ensemble de permutations admissibles.

h. L'algorithme de la méthode Heuristique

```
Function 2-opt (G : Graphe, H : CycleHamiltonien)

  Amélioration : booléen := vrai ;
  Tant que Amélioration = vrai Faire

    Amélioration := faux ;
    Pour tout somme xi de H Faire

       Pour tout somme Xj de H, avec j différent de i-1, de i et de i+1 Faire

          Si distance(Xi,xi+1)+distance(Xj,Xj+1)>distance(Xi, Xj)+distance(Xi+1, Xj+1) Alors
          Remplacer les arrêtes (Xi, Xi+1) et (Xj, Xj+1) par (Xi, Xj) et (Xi+1, Xj+1) dans H

          Amélioration := vrai ;

Return H ;
```

i. L'implémentation par Python

```python
def TspHeuristic(instance):

    fileName = os.path.join( os.getcwd(), '../projet', 'tsp-maroc.xlsx' )
    path = fileName

    df = pd.read_excel(path, sheet_name=instance)
    df = df.replace(np.nan, 0)
    df.drop(['Unnamed: 0'], axis=1, inplace=True)
    df = df.astype(int)
    matrix = df.to_numpy()

    if instance == 'instance_1':
        distance_matrix = np.maximum(matrix, matrix.transpose())
    else:
        distance_matrix = matrix

    start_time = time.time()

    permutation, distance = tspHeuristic(distance_matrix)

    end_time = time.time()

    timing = end_time - start_time

    return {'result': permutation, 'cost': distance, 'Mytime': timing}
```

j. Explication du programme

- Cette fonction prend comme paramètre une chaine de caractère qui représente le nom de l'instance avec laquelle on va travailler.
- En suit la fonction permet de lire un fichier Excel avec l'instance en paramètre, puis elle fait appel a une autre fonction qui permet de calculer le cout (distance) et le chemin les plus courts.
- La fonction calculer aussi le temps d'exécution de la méthode Heuristique, on déclare le temps du système au début et en fin de la fonction, finalement on trouve le temps d'exécution par la relation suivante : **Temps d'exécution = temps final – temps initial**

III. Chapitre 3 : Résultats obtenus par les deux méthodes

Dans ce chapitre, on va appliquer l'implémentation de la programmation dynamique et l'Heuristique (2-opt), présentés dans les chapitres précédents pour résoudre les instances des deux problèmes.

Les résultats si dessous sont obtenue depuis un ordinateur dont les spécifications suivantes :

Nom de l'appareil	DESKTOP-9DAE5F1
Processeur	Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz 1.70 GHz
Mémoire RAM installée	8.00 Go
ID de périphérique	4FDB0EA0-CC36-454F-8C8E-99D2F97E7857
ID de produit	00326-10000-00000-AA285
Type du système	Système d'exploitation 64 bits, processeur x64

Figure 4 - Les configurations de l'ordinateur

5. Résultats de la programmation dynamique

Les résultats obtenus à partir de la programmation dynamique sont regrouper dans le tableau si dessous dont :

- La colonne instance représente le numéro de l'instance du fichier Excel.
- La matrice : un élément a_{ij} de la matrice repensent la distance entre la ville i et j.
- La valeur optimale : représente la distance du chemin le plus court pour parcourir toutes les villes une seule fois et de revenir à la ville de départ.
- Le chemin optimal : représente le chemin c.-à-d. toutes les villes parcourues pour avoir la valeur optimale.
- Le temps d'exécution : est le temps nécessaire pour exécuter le programme.

Instance	Matrice	Valeur optimal	Chemin optimal	Temps d'exécution
Instance 1	[[0, 13, 27, 20, 24, 24, 22, 17, 38, 33], [31, 0, 12, 45, 35, 39, 43, 49, 44, 35], [27, 12, 0, 29, 40, 50, 23, 47, 47, 20], [20, 45, 29, 0, 14, 49, 18, 13, 20, 26], [24, 35, 40, 14, 0, 48, 29, 19, 36, 28], [24, 39, 50, 49, 48, 0, 19, 21, 20, 11], [22, 43, 23, 18, 29, 19, 0, 44, 46, 46], [17, 49, 47, 13, 19, 21, 44, 0, 12, 30], [38, 44, 47, 20, 36, 20, 46, 12, 0, 30], [33, 35, 20, 26, 28, 11, 46, 30, 30, 0]]	179	City 1 ➔ city 2 ➔ city 3 ➔ city 10 ➔ city 6 ➔ city 9 ➔ city 8 ➔ city 5 ➔ city 4 ➔ city 7 ➔ city 1	0.032 ms
Instance 2	[[0, 85, 48, 74, 74, 84, 10, 78, 20, 69, 15, 50, 70, 17, 23, 83, 84, 61, 47, 37] [38, 0, 78, 65, 85, 77, 78, 30, 76, 12, 29, 33, 34, 13, 37, 41, 73, 42, 89, 50] [65, 13,0, 78, 60, 95, 74, 36, 94, 36, 73, 11, 15, 38, 29, 25, 59, 24, 25, 76] [24, 34, 18, 0, 93, 46, 58, 97, 89, 49, 64, 72, 81, 40, 46, 19, 95, 99, 96, 72] [55, 2, 21, 16, 0, 99, 96, 45, 12, 37, 21, 88, 38, 48, 26, 15, 94, 76, 42, 12] [60, 99, 82, 14, 52, 0, 93, 60, 75, 33, 21, 31, 49, 66, 21, 49, 16, 62, 94, 47] [92, 83, 15, 40, 24, 13, 0, 94, 50, 99, 88, 93, 56, 47, 20, 52, 44, 60, 94, 41] [11, 27, 87, 97, 68, 19, 85, 0, 65, 91, 90, 99, 60, 50, 12, 85, 37, 59, 28, 96] [11, 87, 83, 80, 13, 69, 78, 89, 0, 93, 37, 65, 70, 82, 27, 75, 98, 36, 89, 41] [48, 88, 66, 21, 38, 67, 29, 22, 15, 0, 62, 100, 81, 56, 13, 17, 62, 75, 36, 68] [44, 57, 45, 44, 59, 72, 48, 27, 89, 63, 0, 87, 43, 13, 54, 72, 23, 13, 16, 38] [76, 40, 99, 52, 82, 79, 96, 39, 53, 52, 25, 0, 31, 39, 24, 100, 73, 82, 19, 86] [64, 55, 89, 57, 21, 68, 100, 77, 48, 41, 58, 25, 0, 13, 62, 99, 12, 70, 46, 14] [83, 34, 44, 71, 81, 68, 56, 46, 75, 38, 50, 98, 61, 0, 18, 63, 51, 74, 70, 56] [13, 80, 36, 100, 32, 69, 95, 74, 58, 78, 17, 27, 13, 81, 0, 76, 66, 44, 91, 45] [70, 55, 19, 87, 51, 35, 14, 54, 88, 35, 81, 98, 61, 43, 69, 0, 95, 50, 69, 63] [13, 28, 87, 36, 39, 46, 89, 31, 15, 62, 82, 78, 62, 60, 19, 79, 0, 61, 32, 70] [69, 39, 21, 20, 79, 74, 67, 98, 15, 88, 60, 35, 51, 27, 62, 44, 38, 0, 65, 19] [79, 67, 12, 55, 87, 33, 36, 92, 50, 25, 25, 92, 54, 16, 75, 65, 38, 26, 0, 47] [26, 24, 99, 47, 95, 94, 98, 10, 60, 89, 83, 33, 18, 17, 35, 31, 40, 44, 10, 0]]	306	City 1 ➔ city 7 ➔ city 6 ➔ city 4 ➔ city 16 ➔ city 3 ➔ city 12 ➔ city 11 ➔ city 18 ➔ city 9 ➔ city 5 ➔ city 20 ➔ city 19 ➔ city 14 ➔ city 15 ➔ city 13 ➔ city 17 ➔ city 2 ➔ city 10 ➔ city 8 ➔ city 1	202. 728 ms
Instance 3	———	——	———	——
Instance 4	———	——	———	——

6. Résultats de la méthode Heuristique

Instance	Matrice	Valeur optimal	Chemin optimal	Temps d'exécution
Instance 1	[[0, 13, 27, 20, 24, 24, 22, 17, 38, 33], [31, 0, 12, 45, 35, 39, 43, 49, 44, 35], [27, 12, 0, 29, 40, 50, 23, 47, 47, 20], [20, 45, 29, 0, 14, 49, 18, 13, 20, 26], [24, 35, 40, 14, 0, 48, 29, 19, 36, 28], [24, 39, 50, 49, 48, 0, 19, 21, 20, 11], [22, 43, 23, 18, 29, 19, 0, 44, 46, 46], [17, 49, 47, 13, 19, 21, 44, 0, 12, 30], [38, 44, 47, 20, 36, 20, 46, 12, 0, 30], [33, 35, 20, 26, 28, 11, 46, 30, 30, 0]]	179	City 1 ➔ city 2 ➔ city 3 ➔ city 10 ➔ city 6 ➔ city 9 ➔ city 8 ➔ city 5 ➔ city 4 ➔ city 7 ➔ city 1	0.1519 ms
		189	City 1 ➔ city 5 ➔ city 4 ➔ city 8 ➔ city 9 ➔ city 10 ➔ city 6 ➔ city 7 ➔ city 3 ➔ city 2 ➔ city 1	0.1769 ms
		186	City 1 ➔ city 5 ➔ city 4 ➔ city 7 ➔ city 3 ➔ city 2 ➔ city 10 ➔ city 6 ➔ city 9 ➔ city 8 ➔ city 1	0.0559 ms
Instance 2	[[0, 85, 48, 74, 74, 84, 10, 78, 20, 69, 15, 50, 70, 17, 23, 83, 84, 61, 47, 37] [38, 0, 78, 65, 85, 77, 78, 30, 76, 12, 29, 33, 34, 13, 37, 41, 73, 42, 89, 50] [65, 13,0, 78, 60, 95, 74, 36, 94, 36, 73, 11, 15, 38, 29, 25, 59, 24, 25, 76] [24, 34, 18, 0, 93, 46, 58, 97, 89, 49, 64, 72, 81, 40, 46, 19, 95, 99, 96, 72] [55, 2, 21, 16, 0, 99, 96, 45, 12, 37, 21, 88, 38, 48, 26, 15, 94, 76, 42, 12] [60, 99, 82, 14, 52, 0, 93, 60, 75, 33, 21, 31, 49, 66, 21, 49, 16, 62, 94, 47] [92, 83, 15, 40, 24, 13, 0, 94, 50, 99, 88, 93, 56, 47, 20, 52, 44, 60, 94, 41] [11, 27, 87, 97, 68, 19, 85, 0, 65, 91, 90, 99, 60, 50, 12, 85, 37, 59, 28, 96] [11, 87, 83, 80, 13, 69, 78, 89, 0, 93, 37, 65, 70, 82, 27, 75, 98, 36, 89, 41] [48, 88, 66, 21, 38, 67, 29, 22, 15, 0, 62, 100, 81, 56, 13, 17, 62, 75, 36, 68] [44, 57, 45, 44, 59, 72, 48, 27, 89, 63, 0, 87, 43, 13, 54, 72, 23, 13, 16, 38] [76, 40, 99, 52, 82, 79, 96, 39, 53, 52, 25, 0, 31, 39, 24, 100, 73, 82, 19, 86] [64, 55, 89, 57, 21, 68, 100, 77, 48, 41, 58, 25, 0, 13, 62, 99, 12, 70, 46, 14] [83, 34, 44, 71, 81, 68, 56, 46, 75, 38, 50, 98, 61, 0, 18, 63, 51, 74, 70, 56] [13, 80, 36, 100, 32, 69, 95, 74, 58, 78, 17, 27, 13, 81, 0, 76, 66, 44, 91, 45] [70, 55, 19, 87, 51, 35, 14, 54, 88, 35, 81, 98, 61, 43, 69, 0, 95, 50, 69, 63] [13, 28, 87, 36, 39, 46, 89, 31, 15, 62, 82, 78, 62, 60, 19, 79, 0, 61, 32, 70] [69, 39, 21, 20, 79, 74, 67, 98, 15, 88, 60, 35, 51, 27, 62, 44, 38, 0, 65, 19] [79, 67, 12, 55, 87, 33, 36, 92, 50, 25, 25, 92, 54, 16, 75, 65, 38, 26, 0, 47] [26, 24, 99, 47, 95, 94, 98, 10, 60, 89, 83, 33, 18, 17, 35, 31, 40, 44, 10, 0]]	487	City 1 ➔ city 7 ➔ city 15 ➔ city 11 ➔ city 4 ➔ city 16 ➔ city 3 ➔ city 2 ➔ city 8 ➔ city 6 ➔ city 12 ➔ city 13 ➔ city 14 ➔ city 10 ➔ city 19 ➔ city 18 ➔ city 20 ➔ city 17 ➔ city 5 ➔ city 9 ➔ city 1	0.5696 ms
		515	City 1 ➔ city 11 ➔ city 6 ➔ city 10 ➔ city 9 ➔ city 5 ➔ city 14 ➔ city 15 ➔ city 18 ➔ city 20 ➔ city 13 ➔ city 12 ➔ city 19 ➔ city 17 ➔ city 4 ➔ city 16 ➔ city 7 ➔ city 3 ➔ city 2 ➔ city 8 ➔ city 1	0.8065 ms
		577	City 1 ➔ city 12 ➔ city 19 ➔ city 13 ➔ city 17 ➔ city 9 ➔ city 11 ➔ city 18 ➔ city 20 ➔ city 14 ➔ city 7 ➔ city 15 ➔ city 3 ➔ city 16 ➔ city 8 ➔ city 6 ➔ city 5 ➔ city 2 ➔ city 10 ➔ city 4 ➔ city 1	0.2728 ms
	[0, 208, 433,254, 404, 442, 317, 191, 468 … 274, 46, 164, 344, 47, 453, 118, 37] [208, 0, 303, 148, 436, 74, 491, 335, … 167, 368, 450, 268, 226, 287, 471, 242] [433 ,303, 0, 196, 276, 138, 101, 54 … 179, 210, 174, 396, 244, 287, 69, 308] [254, 148, 196, 0, 217, 314, 383, 168 … 372, 89, 331, 36, 429, 236, 301, 341] . . .	923	City 1 ➔ city 32 ➔ city 13 ➔ city 5 ➔ city 48 ➔ city 38 ➔ city 34 ➔ city 17 ➔ city 30 ➔ city 23 ➔ city 16 ➔ city 43 ➔ city 28 ➔ city 9 ➔ city 7 ➔ city 6 ➔ city 49 ➔ city 27 ➔ city 22 ➔ city 10 ➔ city 45 ➔ City 39 ➔ city 15 ➔ city 42 ➔ city 37 ➔ city 19 ➔ city	3.1741 ms

Instance 3	. . . . . [47, 226, 244, 429, 193, 349, 249, 424 … 497, 359, 485, 249, 0, 488, 160, 216] [453, 287, 287, 236, 105, 224, 180, 80 … 183, 229, 378, 59, 488, 0, 36, 313] [118, 417, 69, 301, 286, 121, 287, 405 … 338, 210, 72, 485, 160, 36, 0, 378] [37, 242, 308, 341, 105, 419, 30, 404… 337, 311, 197, 235, 216, 313, 378, 0]		14 ➔ city 35 ➔ city 20 ➔ city 40 ➔ city 29 ➔ city 25 ➔ city 18 ➔ city 4 ➔ city 46 ➔ city 44 ➔ city 3 ➔ city 26 ➔ city 24 ➔ city 50 ➔ city 47 ➔ city 41 ➔ City 36 ➔ city 2 ➔ city 33 ➔ city 31 ➔ city 12 ➔ city 11 ➔ city 8 ➔ city 1	
		1034	City 1 ➔ city 34 ➔ city 23 ➔ city 17 ➔ city 2 ➔ city 33 ➔ city 48 ➔ city 49 ➔ city 39 ➔ city 24 ➔ city 50 ➔ city 47 ➔ city 21 ➔ city 44 ➔ city 35 ➔ city 32 ➔ city 27 ➔ city 18 ➔ city 28 ➔ city 6 ➔ city 4 ➔ City 3 ➔ city 45 ➔ city 14 ➔ city 9 ➔ city 30 ➔ city 16 ➔ city 46 ➔ city 38 ➔ city 26 ➔ city 15 ➔ city 10 ➔ city 8 ➔ city 41 ➔ city 22 ➔ city 42 ➔ city 25 ➔ city 19 ➔ city 43 ➔ city 37 ➔ city 31 ➔ city 20 ➔ City 40 ➔ city 29 ➔ city 7 ➔ city 36 ➔ city 13 ➔ city 12 ➔ city 11 ➔ city 5 ➔ city 1	3.7356 ms
Instance 4	———	——	———	——

IV. Chapitre 4 : Comparaison entre les deux Algorithmes

Pour la méthode 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 les valeurs atteintes après plusieurs exécutions.

Mais pour la programmation dynamique le résultat est exact et on obtient le même résultat chaque fois qu'on exécute une instance.

D'autre part, pour notre implémentation, la programmation dynamique ne peut pas résoudre les instances de taille supérieures à 20 (Dans le même ordinateur dont les spécifications sont notées ci-dessus figure 4), la RAM se remplit et le programme cesse de fonctionner.

Tandis que la méthode 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.

Vous pouvez remarquer que le temps d'exécution de l'heuristique est donne des résultats optimaux même si la taille du problème est très grande (pour 50 villes le résultat est obtenu dans 3.1741 ms).

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 22 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 50 villes mais la RAM se remplit et la session se termine.
- On a essayé avec 30 villes mais encore le même problème…

On conclut que l'approche de programmation dynamique pour résoudre ces deux problèmes n'est pas utile et que l'heuristique est une méthode qu'on peut utiliser pour avoir des résultats suffisants dans un temps rationnel pour les deux problèmes.

Le tableau si dessous regroupe la comparaison entre programmation dynamique et l'Heuristique.

		Heuristique	Dynamique
Temps d'exécution	Instance 1	0.1516 ms	0.032 ms
		0.1769 ms	
		0.0559 ms	
	Instance 2	0.5696 ms	202.7280 ms
		0.8065 ms	
		0.2728 ms	
	Instance 3	3.1741 ms	———
		3.7356 ms	
		3.5743 ms	
Résultats obtenu	Instance 1	179	179
		189	
		186	
	Instance 2	487	306
		515	
		577	
	Instance 3	923	———
		104	
		1053	

V. Chapitre 5 : Les Interface graphique

Pour fournir aux utilisateurs une interface graphique pour utiliser les deux algorithmes pour résoudre le problème de TSP, on a réalisé une application web développer par Flask pour la partie Back-End et Javascript Pour la partie Front-End.

Flask est un micro Framework open-source de développement web en Python. Il est classé comme microFramework car il est très léger. Flask a pour objectif de garder un noyau simple mais extensible. Flask permet de créer des applications web d'une manière rapide et avec des bonnes performances.

L'objectif de cette application web est de recevoir les paramètres (type d'algorithme et le numéro de l'instance) choisis par l'utilisateur en utilisant notre interface, et selon ces paramètres faire appel a la fonction adéquate. Apres l'exécution l'application affiche les résultats aux utilisateurs.

k. Page d'accueil de l'application

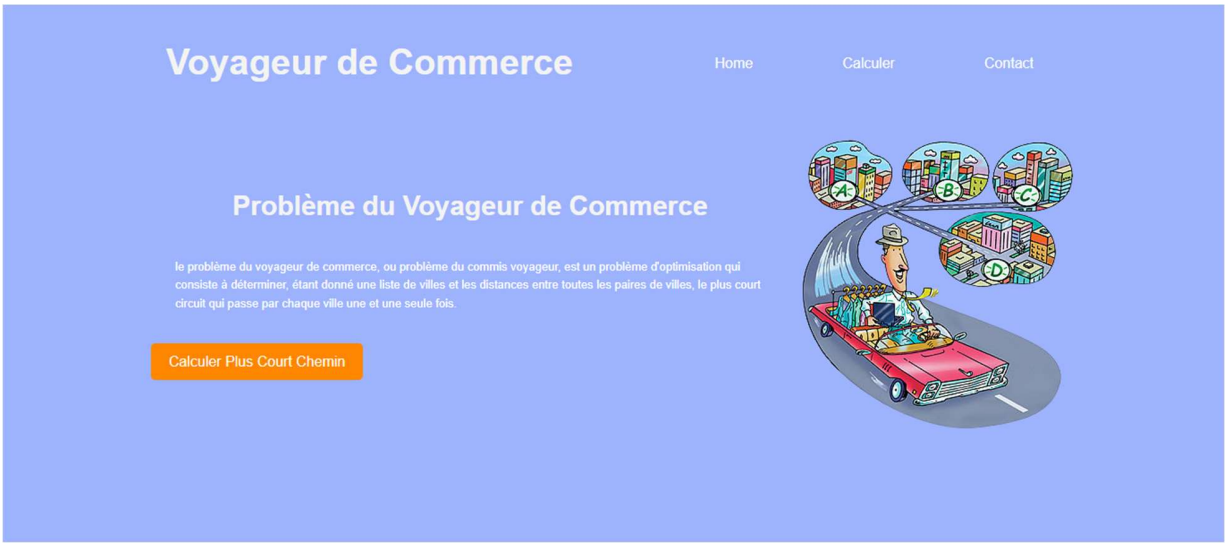

Figure 5 - Page D'accueil

l. Choix de l'algorithme

Cette interface permet aux utilisateurs de choisir la méthode préférer pour exécuter et visualiser les résultats de TSP problème.

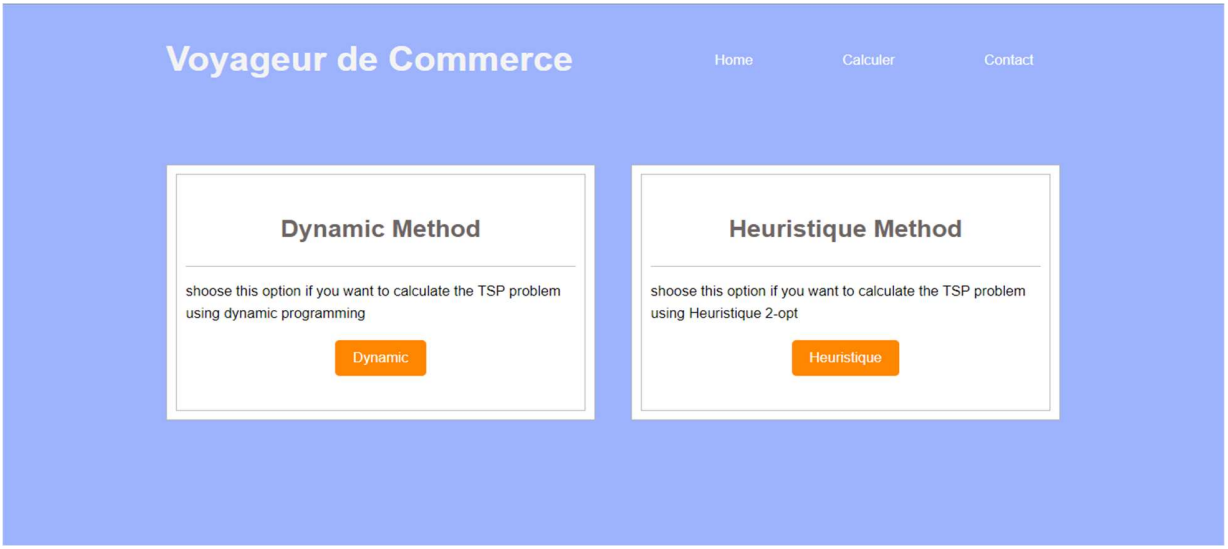

Figure 6 - choix de l'algorithme

m. Choix de l'instance

Apre avoir choisi l'algorithme (soit Dynamique ou Heuristique) notre application afficher un Modal pour lui donner la main de choisir l'instance.

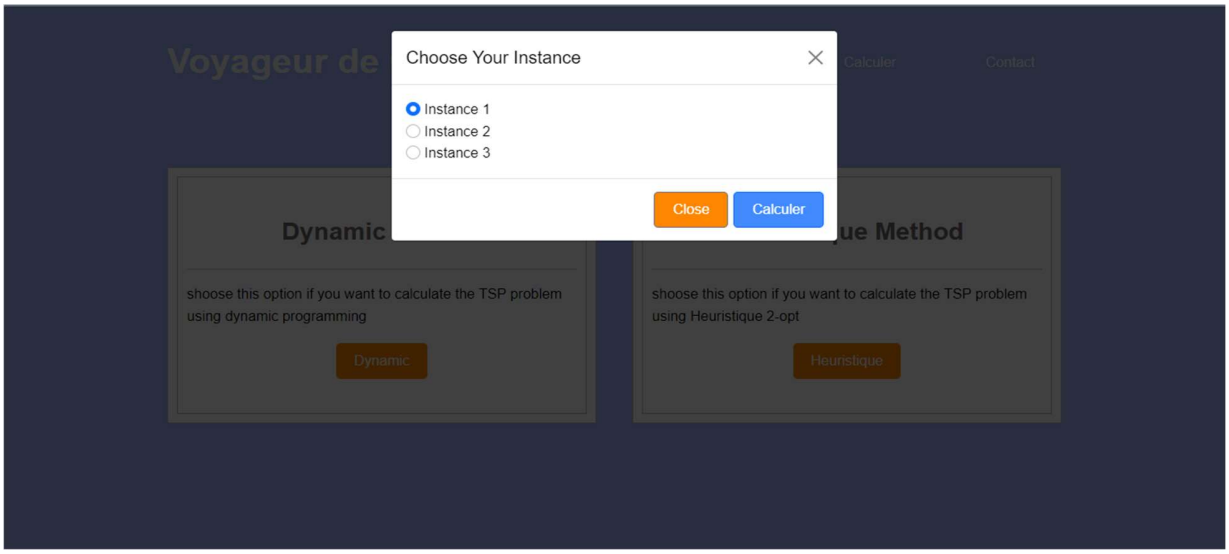

Figure 7 - choix de l'instance

n. Affichage des résultats

Dans notre application les résultats sont affichés comme la montre l'image si dessous, cette dernière affiche le résultat obtenu de l'algorithme Dynamique avec l'instance 2 (20 villes).

Figure 8 - Affichage des résultats dynamique

L'image suivante affiche le résultat obtenu de la méthode Heuristique avec l'instance 3 (50 villes).

Figure 9 - Affichage des résultats Heuristique

Conclusion

Dans ce Projet, on a donné une explication et une vue globale sur la méthode Heuristique et la programmation dynamique pour la résolution du problème TSP. Puis, on a implémenté ces deux méthodes en utilisant python pour résoudre le problème traité dans ce projet.

Après, on a présenté les différents résultats pour chaque méthode à savoir les valeurs optimales, le chemin optimales et les temps d'exécution de chaque méthode. A partir de ces résultats on sorte avec les conclusions suivantes :

- La programmation dynamique n'est pas efficace pour les instances de grandes tailles.
- La programmation dynamique donne des résultats exacts pour les instances qu'elle

  Peut résoudre (instance de taille inferieur a 20 villes pour notre problème).
- L'Heuristique contrairement au dynamique peut résoudre des problèmes de taille très importante.
- Mais les solutions trouver par la méthode Heuristique ne sont pas exactes et elle différents d'une exécution a l'autre.

Puis on a réalisé une application web (développer par Flask) pour permettre aux utilisateurs de tester les implémentations et on a fourni toutes les séquences optimales des 3 instances de taille 10, 20 et 50 pour TSP.

Bibliographie/webographie

[1]. Mohammed LACHGAR, étudiant du master M2SI 2eme années INSEA Rabat

https://lachmed27.pythonanywhere.com/

[2]. Zakaria ALOUANI et Abdelilah SABRI, lauréats du master M2SI INSEA Rabat

https://github.com/zakaria76al/mtc/

[3]. Officiel web site Wikipédia, pour les définition et explication des notions

https://fr.wikipedia.org/wiki/2-opt

[4]. Officiel Web site Flask pour étudier le Flask et produit une application web

https://flask.palletsprojects.com/en/2.2.x/