INSEA

Institut National de Statistique
et d'Economie Appliquée

Rapport de Mini Projet

Sujet :

Comparaison entre la programmation dynamique et l'heuristique pour résoudre le problème de TSP

Elaboré par :

Mohamed ELAZZAOUI

Yassine AFRACHE

Oussama BOUKOUTAYA

Encadré par :

Pr. Rachid BENMANSOUR

Année universitaire 2022/2023

Sommaire

Contents

Introduction Générale

En 1962, Procter et Gamble lancèrent un concours pour résoudre le problème suivant : Trouver un parcours passant par 33 villes figurant sur une carte et le plus court possible.

C'était une des premières versions de ce qui deviendra le célèbre problème dit du "voyageur de commerce", problème étudié par Dantzig dès 1954.L'origine de ce nom est assez obscure et probablement peu de voyageurs de commerce l'utilisent !

Chapitre 1 : Le problème du voyageur de commerce

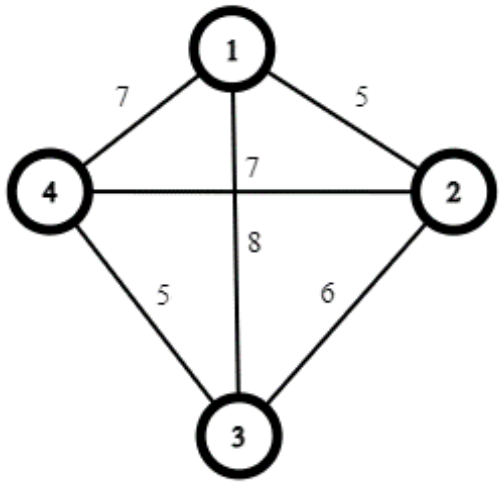
I. Définition :

Le problème du voyageur de commerce (également connu sous le nom de Traveling Salesman Problem ou TSP) pose la question suivante : Un voyageur de commerce doit visiter N villes données, en passant par chaque ville exactement une fois. Il commence dans n'importe quelle ville et se termine en retournant à la ville de départ. La distance entre les villes est connue. Quel chemin choisir pour minimiser la distance parcourue ? La notion de distance peut être remplacée par d'autres notions comme le temps passé ou l'argent dépensé : dans notre cas on parle de la distance ».

Il s'agit d'un problème NP-difficile en optimisation combinatoire, important en informatique théorique et en recherche opérationnelle.

II. Représentation du problème :

Le problème du voyageur de commerce peut être modélisé à l'aide d'un graphe composé d'un ensemble de sommets et d'un ensemble d'arêtes. Chaque sommet représente une ville, une arête symbolise la transition d'une ville à une autre, et elle est associée à un poids qui peut représenter la distance. Résoudre le problème du voyageur de commerce équivaut à trouver un cycle dans ce graphe qui passe par tous les sommets une fois (un tel cycle est « appelé un hamiltonien ») et a une longueur minimale.



Pour le graphique ci-contre, la solution à ce problème est de cycle 1, 2, 3, 4 et 1, correspondant à une distance totale de 23. Cette solution est optimale, il n'y a pas de meilleure solution.

III. Solution triviale :

Pour un nombre de villes égal à N, le nombre de trajets possibles est égal à $\frac{(N-1)!}{2}$, Par exemple, le nombre de parcours pour 5 villes est de 12, mais 10 est déjà 181,440 et 20 est d'environ 60 × 1,015.

Supposons qu'un ordinateur soit assez rapide pour évaluer un itinéraire en une demi-microseconde : le cas des 5 villes sera résolu en moins de 6 microsecondes, le cas des 10 villes en 0,09 secondes, la solution pour 20 villes va durer 964 ans. C'est pourquoi des techniques de résolution efficaces doivent être développées pour trouver rapidement la meilleure solution, ou du moins une solution de qualité.

IV. Conclusion :

Étant donné que le problème du voyageur de commerce a été, et continue à être largement étudié du fait de sa complexité et du nombre important de problèmes dérivés, il existe de nombreux méthodes et algorithmes d’approximation.

Chapitre 2 : Optimisation et approximation

I. Introduction :

En plus de la solution triviale, Il existe deux grandes catégories de méthodes de résolution de ce problème : 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.

Au cours de cette étude, nous aborderons deux méthodes :

- Heuristique : 2-opt
- Programmation Dynamique

II. Résolution par la méthode heuristique 2-opt :

1. Définition :

Les heuristiques sont des méthodes de calcul qui fournissent des solutions rapides, pas nécessairement optimales ou exactes, réalisables à des problèmes d'optimisation difficiles.

2-opt est un algorithme de recherche locale proposé par Georges A. Croes en 19581 pour résoudre le problème du voyageur de commerce en améliorant une solution initiale.

2. Pseudo-Code :

Fonction 2-opt (G : Graphe, H : CycleHamiltonien):

amélioration : booléen := vrai
Tant que amélioration = vrai **faire**

amélioration := faux;
Pour tout sommet x_i de H **faire**

Pour tout sommet x_j de H, avec j différent de i-1, de i et de i+1 **faire**

Si distance(x_i, x_{i+1}) + distance(x_j, x_{j+1}) > distance(x_i, x_j) + distance(x_{i+1}, x_{j+1}) **alors**

Remplacer les arêtes (x_i, x_{i+1}) et (x_j, x_{j+1}) par (x_i, x_j) et (x_{i+1}, x_{j+1}) dans H
amélioration := vrai;
Retourner H

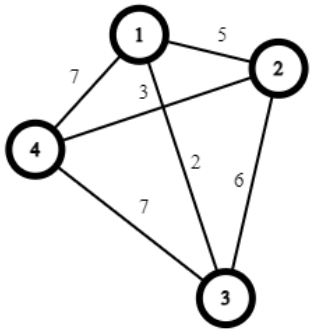
3. Code programmation:

```
import random
import numpy as np

def cost(g,h): #Calculer le cout totale d'une cycle hameltonien
    s=0
    for i in range(len(h)-1):
        s+=g[h[i],h[i+1]]

    return s+g[h[i+1],h[0]]

def randomVectInitZero(n): #Cycle Hamiltonien Aleatoire
    h = list(range(1,n))
    random.shuffle(h)
    return [0]+h

def deux_opt (g): #Fonction 2-opt
    v_dep=0
    nb_v=len(g)
    h = randomVectInitZero(nb_v)
    am = True
    while(am ==True):
        am = False
        for i in range(0,len(h)-1):
            for j in range(i+1,len(h)-1):
                if h[j] not in [h[i-1],h[i],h[i+1]]:
                    if (g[h[i],h[i+1]] + g[h[j], h[j+1]] > g[h[i], h[j]] +
g[h[i+1], h[j+1]]):
                        x=h[i+1]
                        h[i+1]=h[j]
                        h[j]=x
                        am = True
    return h
```

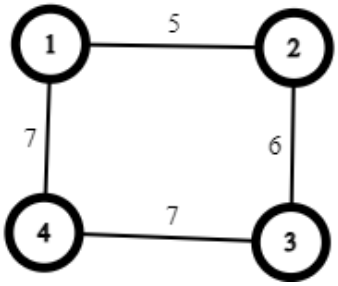
4. Exemple :

Soit 4 villes {1,2,3,4}, la distance entre ces villes est présentée par la matrice suivante :

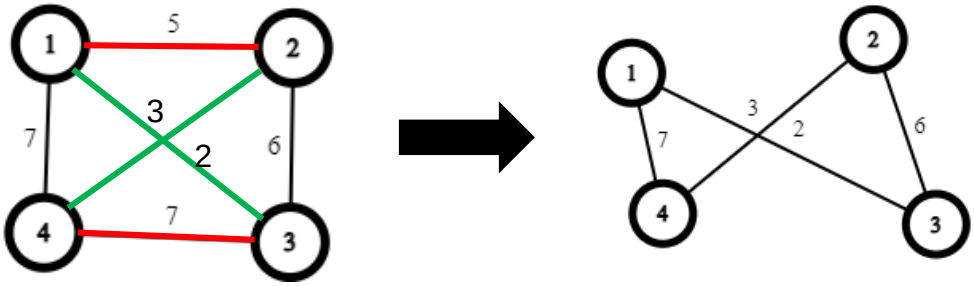
$$\begin{bmatrix} 0 & 5 & 2 & 7 \\ 5 & 0 & 6 & 3 \\ 2 & 6 & 0 & 7 \\ 7 & 3 & 7 & 0 \end{bmatrix}$$



Comme première étape on prend un chemin aléatoirement : [1,2,3,4,1]



Or distance(1,2) + distance(3,4) > distance(1,3) + distance(4,2), on remplace les chemins (1,2) et (3,4) par (1,3) et (4,2)



III. Résolution par la programmation dynamique :

1. Définition :

La programmation dynamique est une technique mathématique visant à prendre des décisions séquentielles indépendantes les unes des autres.

Contrairement à la programmation linéaire, il n'y a pas de formalisme mathématique standard. C'est une approche pour spécifier la formule en fonction du problème à résoudre.

L'approche de programmation dynamique consiste à résoudre le problème étape par étape en tenant compte des connexions entre les étapes. Autrement dit, remplacez la

solution du problème "global" par la solution d'une séquence de sous-problèmes plus petits et plus simples.

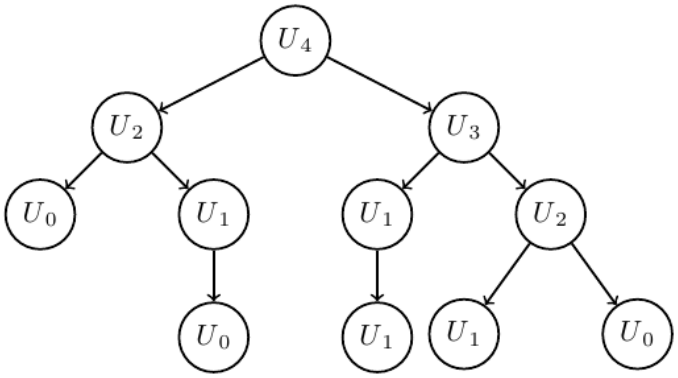
Le principe d'optimalité de Bellman : « Un ensemble de décisions optimales a la propriété que, quel que soit le sous ensemble de décisions correspondant en 1er stage, les décisions restantes doivent être optimales relativement au sous problème résultant des décisions du 1er stage ».

- **Exemple :**

Pour mieux comprendre le concept de programmation dynamique, la suite de Fibonacci est un pertinent choix afin d'atteindre ce but.

La suite de Fibonacci est une suite mathématique récursive qui nous permet de calculer un terme i de cette suite à partir des deux derniers termes calculés.

$$U_i = U_{i-1} + U_{i-2}$$



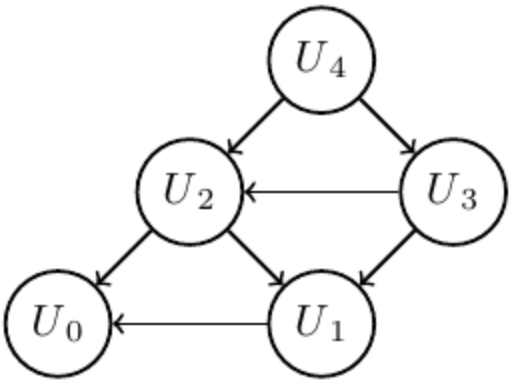
Comme nous voyons dans le dessin au-dessus, le calcul de terme 2 se répète deux fois, cela se traduit par une complexité **exponentielle 2^n** avec n terme à calculer (ici 4).

Le nombre de terme à calculer dans cet exemple est presque 2^4.

La programmation dynamique résout se problème en rendant sa complexité **linéaire**

Cela est réalisé par la technique de **mémoïsation** qui consiste à mémoriser le calcul de chaque terme dans une structure de données et récupérer sa valeur dès que nous l'avons besoin sans le recalculer.

L'illustration visuelle au-dessus montre ce qu'on vient à dire, le calcul de terme 3 se fait une fois sans le recalculer pour un deuxième fois. De plus on remarque bien que pour n terme les calculs se font exactement n fois d'où vient la linéarité de la complexité.



En fait il y a deux approches pour effectuer un programme dynamique soit par l'approche **« Backward »** qui est brièvement basée sur la récursivité ou **« Forward »** qui consiste à itérer.

Backward ou Top-down : Cette approche est basée sur le fait de diviser un problème en sous problème en utilisant la récursivité. Elle commence par le problème de départ jusqu'à arriver au le petit sous problème (état élémentaire) et à chaque fois essaye de mémoriser le résultat des sous problèmes.

Dans la plupart des temps cette approche est facile à implémenter car, on a besoin que d'ajouter une structure qui enregistre le résultat durant la récursivité.

L'approche top-down est très lente par rapport à l'autre approche à cause de ses appels récursives, de plus la récursivité est couteuse au niveau de la mémoire, car tout appel récursif a besoin de placer une adresse mémoire sur la pile au but de pouvoir revenir prochainement à ce point.

Forward ou Buttom-up : Cette méthode basée sur le fait d'itérer, à partir de premier terme, on calcule le deuxième, troisième… jusqu'à arriver à calculer le terme le plus grand (dans l'exemple de Fibonacci au-dessus à partir de fib(1) = 1, on arrive à fib(5) = 5.

En revanche au récursivité, l'itération garantit moins de mémoire à consommer, alors que la récursivité demande beaucoup des appels récursifs à effectuer. Mais elle a besoin de définir un ordre itératif en tenant en compte à toutes les conditions.

2. Algorithme :

Le problème de TSP comme la suite de Fibonacci, on peut aussi le résoudre en adoptant le même concept, mais il faut dans un premier temps définir notre solution mathématiquement puis procédera au pseudo code.

Les étapes à suivre pour résoudre mathématiquement le problème de TSP est les mêmes pour tout autre problème traité par la programmation dynamique.

Afin de présenter l'algorithme lequel on va l'implémenter pour résolvant le problème de TSP, un exemple élémentaire constitué de 4 villes {1,2,3,4} sera associé à notre algorithme.

Soit la matrice d'adjacence suivante :

$$G = \begin{bmatrix} 0 & 5 & 2 & 7 \\ 5 & 0 & 6 & 3 \\ 2 & 6 & 0 & 7 \\ 7 & 3 & 7 & 0 \end{bmatrix}$$

Avec chaque ligne présente une ville, et considérant la ville 1 comme une ville de départ.

a) Fonction récursive :

$$f(S,i) = \min_{k \epsilon S - \{i\}} \{C_{i,k} + f(S - \{k\}, k)\}$$

Avec :

- i la ville actuelle
- S l'ensemble des villes à parcourir
- k la ville prochaine éventuelle

b) Condition Initiale :

$$CI : \begin{cases} f(\emptyset, 2) = C_{2,1} = 5 \\ f(\emptyset, 3) = C_{3,1} = 2 \\ f(\emptyset, 4) = C_{4,1} = 7 \end{cases}$$

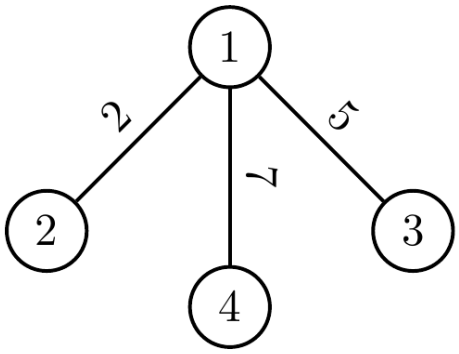
c) Fonction Objective :

On se procède de manière récursive jusqu'à arriver au :

$$f(\{2,3,4\},1)$$

Une illustration visuelle de ce que on vient de dire est judicieuse, pour mieux comprendre ce qu'il se passe à chaque itération.

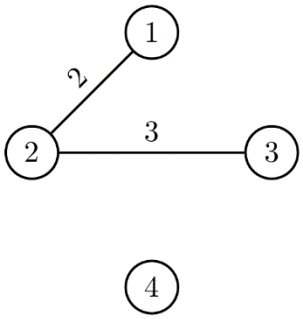
- **Dans l'itération 1 :**

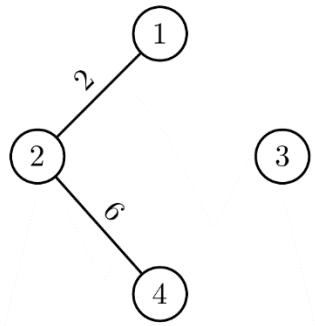


Commençant par la ville numéro 1, le voyageur peut accéder aux villes 2,3 ou 4.

- **Dans l'itération 2 :**

Si la prochaine ville est 2, le successeur peut être 3 ou 4 :

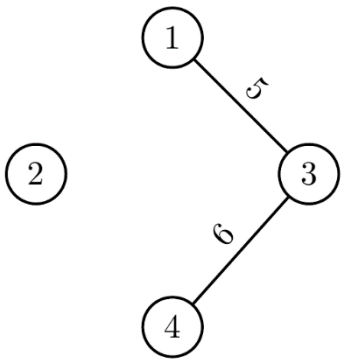


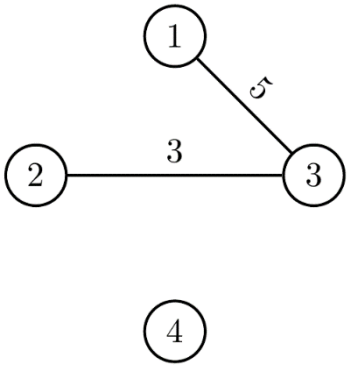


Le chemin optimum dans ce cas est (1,2,3) puisque son cout est 8.

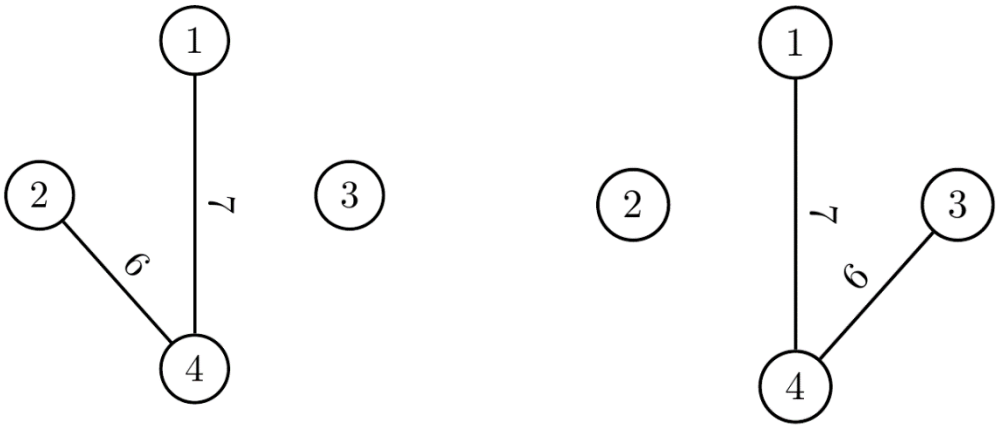
Si la prochaine ville est 3, les successeurs possibles sont 2 et 4 :

Le chemin optimum dans ce cas est (1,3,2), son cout est 8.





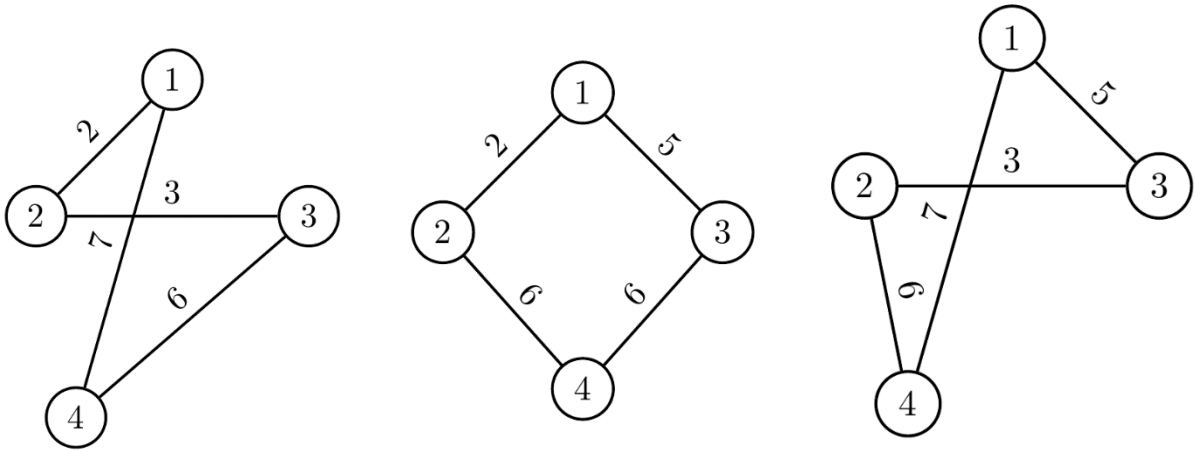
Aussi, si la prochaine ville est 4, le peut être 2 ou 3 :



Le chemin optimum dans ce cas est (1,4,3), son cout est 10.

- **Dans l'itération 3 :**

Pendant Cette étape, on a complété notre cycle hamiltonien est le programme choisira le chemin le plus optimum parmi ceux-ci.



Le chemin optimum dans cet exemple est **(1,3,2,4),** et son cout est **11.**

Après savoir comment l'algorithme de programmation dynamique traite le problème de TSP, maintenant on verra son pseudo-code.

d) Pseudo-Code :

Les données :

- V l'ensemble des villes V.
- v une ville de départ .
- Cout est la matrice d'adjacence qui présente les distances entre les villes.

Le résultat : Le plus court tour qui lie toutes les villes de V.

1. $D_{tsp}(S,w)$ chargé à grader le cout de sous-chemin parcouru dont sa ville d'arriver est w. cette valeur sera initialisée par 0:
2. P(S,w) une liste qui est chargée à garder les prédécesseurs de w .
3. v est initialisée par la ville de départ
4. Pour chaque w € V :
    a) $D_{tsp}(\{w\},w)$ <- c(v,w)
    b) P({w},w) <- v;
5. Pour i = 2...|V| :
    Pour S ⊂ V où |S| = i : //S prend les combinaisons des villes
        Pour chaque w € S : //dont leur cardinal est i
            Pour chaque u € S :
                a. z <- $D_{tsp}(S\backslash\{w\},u)$ + c(u,w)
                // ce bloc conçu à récupérer le min des couts et son chemin
                b. Si z < $D_{tsp}(S,w)$:
                    i. $D_{tsp}(S,w)$ <- z:
                    ii. P(S,w) <- u:
6. Retourner le chemin obtenu après parcourir toutes les villes de la table P.

e) Code Python :

```
import itertools
import numpy as np

def dynamique(g):
    n = len(g)
    A = {(frozenset([0, i+1]), i+1): (cout, [0, i+1]) for i, cout in
enumerate(g[0][1:]) }

    for m in range(2, n):
        B = {}
        for S in [frozenset(C) | {0} for C in itertools.combinations(range(1, n),
m)]:
            for j in S - {0}:
                B[(S, j)] =min((A[(S-{j},k)][0] + g[k][j], A[(S-{j},k)][1] + [j])
for k in S if k != 0 and k!=j)
        A = B
    res = min([(A[d][0] + g[0][d[1]], A[d][1]) for d in iter(A)])
```

```
    Resultat = res[0], [i for i in  res[1]]

    return Resultat
```

IV. Conclusion :

Les méthodes adoptées dans ce travail, chacune à ses propres caractéristiques.

L'heuristique 2-opt nous garantit une vitesse d'exécution mais nous ne donne pas des résultats fiables, puisqu'elle est une méthode approximative.

La programmation dynamique cependant, nous donne une solution exacte de problème puisqu'il est une méthode exacte c'est-à-dire parcours toutes les solutions possibles, mais cela coute très chère au niveau de ressources de machine.

Le chapitre suivant est consacré à aborder davantage ses différences avec une comparaison basé sur notre étude empirique.

Chapitre 3 : Implémentation, Résultats et Discussion :

I. Implémentation

1. Les données utilisées :

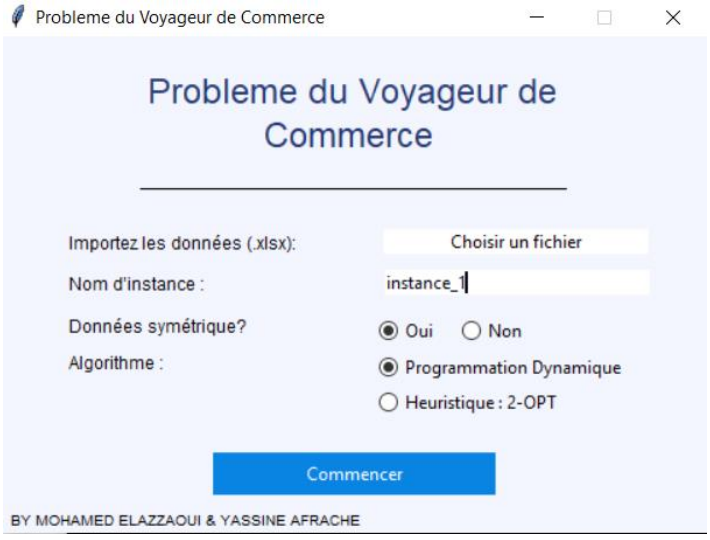
Un fichier Excel est mis à nous disposition, ce fichier-là contient 3 instances, une contient 10 villes, une 20 villes et la dernière 30 villes, en considérant que la ville de 0 est la ville de départ et d'arrivée.

I17 | 54

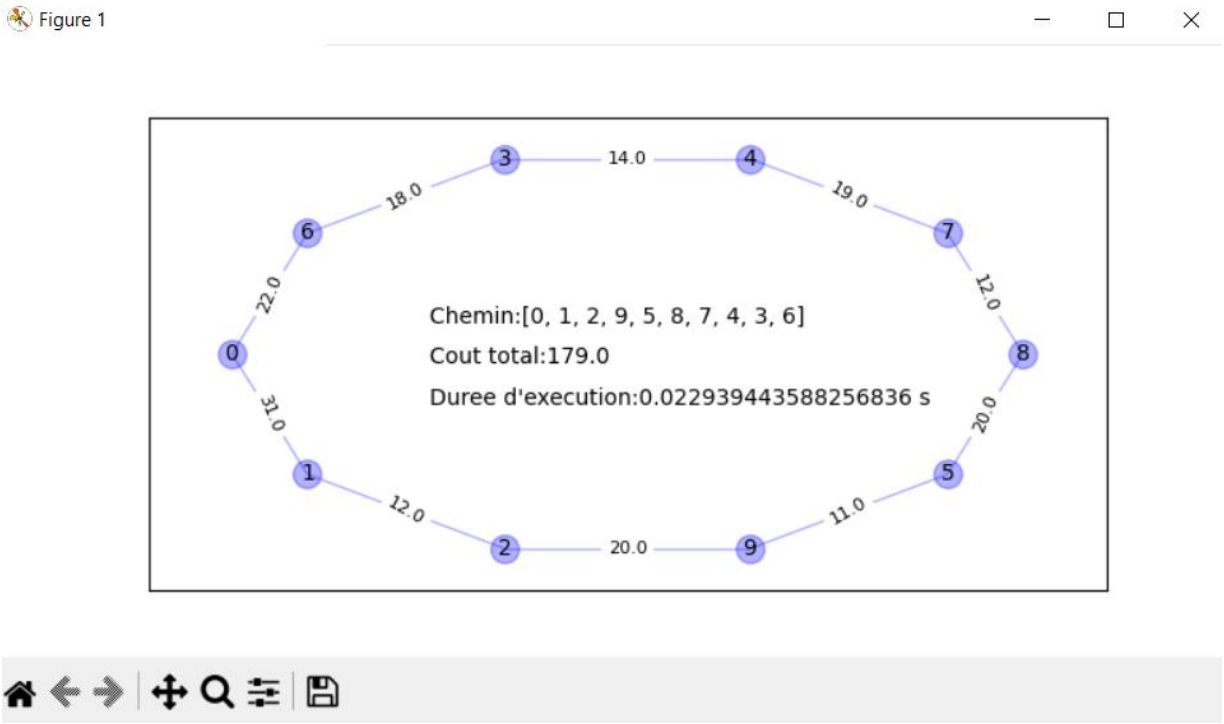
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	1	0	85	48	74	74	84	10	78	20	69	15	50	70	17	23	83	84	61	47	37
3	2	38	0	78	65	85	77	78	30	76	12	29	33	34	13	37	41	73	42	89	50
4	3	65	13	0	78	60	95	74	36	94	36	73	11	15	38	29	25	59	24	25	76
5	4	24	34	18	0	93	46	58	97	89	49	64	72	81	40	46	19	95	99	96	72
6	5	55	22	21	16	0	99	96	45	12	37	21	88	38	48	26	15	94	76	42	12
7	6	60	99	82	14	52	0	93	60	75	33	21	31	49	66	21	49	16	62	94	47
8	7	92	83	15	40	24	13	0	94	50	99	88	93	56	47	20	52	44	60	94	41
9	8	11	27	87	97	68	19	85	0	65	91	90	99	60	50	12	85	37	59	28	96
10	9	11	87	83	80	13	69	78	89	0	93	37	65	70	82	27	75	98	36	89	41
11	10	48	88	66	21	38	67	29	22	15	0	62	100	81	56	13	17	62	75	36	68
12	11	44	57	45	44	59	72	48	27	89	63	0	87	43	13	54	72	23	13	16	38
13	12	76	40	99	52	82	79	96	39	53	52	25	0	31	39	24	100	73	82	19	86
14	13	64	55	89	57	21	68	100	77	48	41	58	25	0	13	62	99	12	70	46	14
15	14	83	34	44	71	81	68	56	46	75	38	50	98	61	0	18	63	51	74	70	56
16	15	13	80	36	100	32	69	95	74	58	78	17	27	13	81	0	76	66	44	91	45
17	16	70	55	19	87	51	35	14	54	88	35	81	98	61	43	69	0	95	50	69	63
18	17	13	28	87	36	39	46	89	31	15	62	82	78	62	60	19	79	0	61	32	70
19	18	69	39	21	20	79	74	67	98	15	88	60	35	51	27	62	44	38	0	65	19
20	19	79	67	12	55	87	33	36	92	50	25	25	92	54	16	75	65	38	26	0	47
21	20	26	24	99	47	95	94	98	100	60	89	83	33	18	17	35	31	40	44	10	0
22																					
23																					
24																					
25																					

instance_1 | instance_2 | instance_3 | instance 4

2. Interface graphique :



Nous avons conçu une interface graphique qui contient un champ pour importer le fichier Excel qui contient les données, un champ pour insérer le nom de l'instance, un champ pour indiquer si les informations symétriques ou non. Enfin, le choix de la façon de résoudre le problème. Après avoir appuyé sur Commencer, le programme s'exécute et à la fin, une fenêtre apparaît contenant les résultats (Graphe qui présente la cycle hamiltonien la plus courte et son cout avec la durée d'exécution de l'algorithme utilisé).



II. Résultats et discussion :

Afin d'arriver à savoir laquelle des deux méthodes (Programmation dynamique et heuristique 2opt) est la plus favorable pour résoudre le problème de TSP, une comparaison des méthodes est exigeante. Cette comparaison consiste à évaluer ces deux méthodes avec des instances différentes en incrémentant le nombre de villes.

	2-opt	**Programmation dynamique**
Instance_1	Temps d'exécution 0 ms Cout : 196	Temps d'exécution 20ms Cout : 179
Instance_2	Temps d'exécution 0.9ms Cout : 510	Temps d'exécution 1770ms Cout : 339
Instance_3	Temps d'exécution 1278ms Cout : 3195	Problème de contrainte machine

N.B : Les temps d'exécution sont approchés.

La table au-dessus présente les résultats obtenus par les deux méthodes après les avoir essayés sur les instances 1, 2 et 3.

Pour toutes les instances la méthode 2-opt est toujours rapide que la méthode de programmation dynamique, pourtant cette dernière arrive au résultat plus optimum que la première. Le cas de troisième instance révèle la limitation de la méthode de programmation dynamique, alors que la méthode de 2-opt reste encore valable.

La programmation dynamique au niveau de résultat est plus fiable que 2-opt, car elle est exacte alors que 2-opt est approximative.

Au niveau de rapidité, 2-opt est plus rapide que la programmation dynamique car elle n’a pas besoin de parcourir toutes les solutions possibles, une fois elle trouve un chemin optimum par rapport à ses voisins, il est considéré comme optimum.

De plus, la programmation dynamique après un certain nombre des villes, sa solution sera plus couteuse au niveau mémoires et calculs, et exigera plus des ressources et temps.

Conclusion Générale

Après les expériences, les recherches et les comparaisons que nous avons faites, il s'avère que le problème de la TSP est toujours un problème de recherche, malgré le développement que la technologie connaît d'un point de vue matériel, mais la solution matérielle ne sera jamais convaincante solution, Il nécessite une solution mathématique ou un algorithme de moins complexité.

Ni la méthode heuristique n'a donné de solutions parfaites, ni la méthode de programmation dynamique n'a donné de solution plus rapide.

Comme perspective nous souhaitons re-optimiser notre algorithme de programmation dynamique afin d'atteindre une durée d'exécution minimale que la dernière.

Bibliographie et Références

Bouman, P., Agatz, N., & Schmidt, M. (2017). *Dynamic Programming Approaches for the Traveling Salesman Problem with Drone.* Récupéré sur ResearchGate: https://www.researchgate.net/publication/320075655_Dynamic_Programming_Approaches_for_the_Traveling_Salesman_Problem_with_Drone

Dynamic programming. (s.d.). Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/Dynamic_programming

Heuristic : 2-opt. (s.d.). Récupéré sur Wikipedia: https://en.wikipedia.org/wiki/2-opt

Le Probeme de Voyageur de Commerce Et La Coloration Des Sommets D'un Graphe. (2013, May). Récupéré sur Scribd: https://fr.scribd.com/document/140666634/Le-Probeme-de-Voyageur-de-Commerce-Et-La-Coloration-Des-Sommets-d-Un-Graphe

Premkumar, K. (2020). *Traveling Salesman Problem - Dynamic programming*. Récupéré sur Medium: https://medium.com/ivymobility-developers/traveling-salesman-problem-9ab623c88fab

Probléme du voyageur de commerce. (2022, May). Récupéré sur Scribd: https://fr.scribd.com/document/573317075/ift1575-doc-tsp-1

Voyageur de Commerce. (2015, November). Récupéré sur Scribd: https://fr.scribd.com/doc/290579566/Voyageur-de-Commerce-recuit-Simule