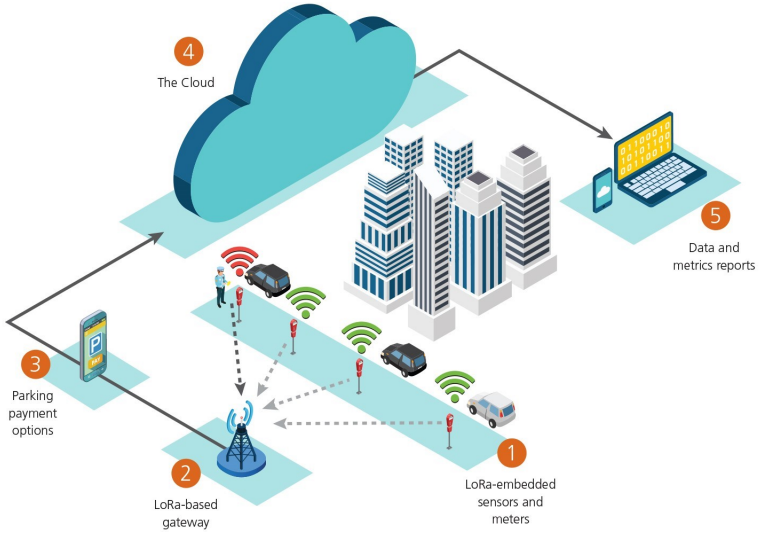
المعهد الوطني للبريد والمواصلات
ⵓⵔⵣⵣⵓ ⵓⵏⵎⵓⵔⵉ ⵏ ⵜⵓⵙⵙⵓⵔⵜ ⴰ ⵙⵎⵙⵓⵍⵓⵔ
Institut National des Postes et Télécommunications

STRUCTURES DE DONNÉES ET ALGORITHMIQUE

RAPPORT DE PROJET

Problème du voyageur de commerce avec tour euclidien bitonique



Réalisé par :
TCHAH OUSSAMA
LAHSINI MOHAMED

Encadré par :
MR. BENSAID HICHAM

Filière SUD - Cloud & IOT

Table des matières

Première partie

Rapport

0.1 Introduction

Le problème du voyageur de commerce, étudié depuis le 19e siècle, est l'un des plus connus dans le domaine de la recherche opérationnelle. C'est déjà sous forme de jeu que William Rowan Hamilton a posé pour la première fois ce problème, dès 1859. Sous sa forme la plus classique, son énoncé est le suivant : « Un voyageur de commerce doit visiter une et une seule fois un nombre fini de villes et revenir à son point d'origine. Trouvez l'ordre de visite des villes qui minimise la distance totale parcourue par le voyageur ». Les domaines d'application sont nombreux : problèmes de logistique, de génétique, de transport aussi bien de marchandises que de personnes, et plus largement toutes sortes de problèmes d'ordonnancement. Malgré la simplicité de son énoncé, il s'agit d'un problème d'optimisation pour lequel on ne connait pas d'algorithme permettant de trouver une solution exacte rapidement dans tous les cas. Plus précisément, on ne connait pas d'algorithme en temps polynomial, et sa version décisionnelle (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.

0.2 Modélisation du problème et méthodes de résolution

0.2.1 Modélisation

Le problème du voyageur de commerce peut être modélisé à l'aide d'un graphe constitué d'un ensemble de sommets et d'un ensemble d'arêtes. 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. Formellement, une instance est graphe complet G=(V,A,w) avec V un ensemble de sommets, A un ensemble d'arêtes et w une fonction de coût sur les arcs. 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.

0.2.2 Méthodes de résolution

Face à ce type de problèmes, 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.

1) Méthodes exactes

Pour le problème du voyageur de commerce, l'une des méthodes exactes les plus classiques et les plus performantes reste la Procédure par Séparation et Evaluation (PSE). Cette méthode repose sur le parcours d'un arbre de recherche. Dans un chemin de cet arbre, le premier nœud représente la ville de départ, son successeur la deuxième ville visitée, puis la troisième ville visitée, etc. À chaque étape de l'algorithme, on crée autant de nœuds qu'il reste de villes à visiter. À chaque nœud, le choix consiste à sélectionner la prochaine ville à visiter parmi les villes restantes.

2) Méthodes approchées

Dans le cas d'un nombre de villes si grand que même les meilleures méthodes exactes nécessitent un temps beaucoup trop long de résolution, des méthodes approchées, ou algorithmes d'approximation, sont utilisées. Elles permettent d'obtenir en un temps très rapide de bonnes solutions, pas nécessairement optimales, mais d'une qualité suffisante. Nous citons ci-dessous quelques exemples de ces algorithmes :

a) Algorithme du plus proche voisin :

Il s'agit de l'un des nombreux algorithmes gloutons traitant le problème du voyageur de commerce est l'algorithme **du plus proche voisin**. La première étape repose sur le choix aléatoire d'une première ville, et les étapes suivantes consistent à se déplacer de ville en ville en appliquant la règle du plus proche voisin, c'est-à-dire en sélectionnant la prochaine ville telle que le poids entre la ville courante et la prochaine ville soit minimal, et ce, jusqu'à avoir visité toutes les villes. Il faut enfin revenir à la première ville choisie, pour obtenir un cycle.

b) Méthode par insertion :

Cet algorithme fait partie également de la famille des algorithmes gloutons et se base, comme son nom l'indique, sur la **méthode d'insertion**. En effet, l'idée est la suivante : le parcours du voyageur de commerce est construit pas à pas en y insérant de nouvelles villes. À un instant donné de l'algorithme, un certain cycle de villes a été construit. L'étape suivante consiste à insérer une ville supplémentaire dans le cycle de manière optimale, c'est-à-dire qu'elle augmente au minimum la longueur totale du cycle. À l'étape initiale de l'algorithme, le parcours de voyageur est composé de deux villes, la ville de départ et celle qui en est la plus proche. L'algorithme se termine lorsque toutes les villes à visiter ont été insérées.

Il existe d'autres méthodes approchées, fondées sur des principes totalement différents. Nous évoquerons deux d'entre elles, assez innovantes et intéressantes puisqu'elles s'inspirent de phénomènes naturels : les algorithmes génétiques et les algorithmes de colonies de fourmis.

0.3 Objectif

L'objectif de ce projet est donc d'étudier le problème du voyageur de commerce dans le cas particulier du tour euclidien bitonique, et ce en étudiant la solution naïve et en proposant une solution efficace, puis tracer la complexité de chacune dans le but de les comparer et en tirer une conclusion.

0.4 Description du problème

Le problème du voyageur de commerce euclidien consiste à déterminer la plus petite tournée permettant de relier un ensemble donné de n points du plan. La figure 1(a) montre la solution pour un problème à 7 points. Comme nous l'avons mentionné précédemment, le problème général est NP-complet, et il y a donc de fortes chances pour que sa solution demande un temps suprapolynomial. J. L. Bentley a suggéré de simplifier le problème en se restreignant aux tournées bitoniques, autrement dit, celles qui partent du point le plus à gauche, continuent strictement de gauche à droite vers le point le plus à droite, puis retournent vers le point de départ en se déplaçant strictement de droite à gauche. La figure 1(b) montre la plus courte tournée bitonique pour ces mêmes 7 points. Dans ce cas, on peut trouver un algorithme à temps polynomial. Dans notre cas, et pour simplifier les calculs ainsi que la sous structure optimale et la relation d'ordre utilisée, que nous allons voir dans la partie suivante, nous avons choisi de numéroter les sommets dans un ordre croissant de gauche à droite.

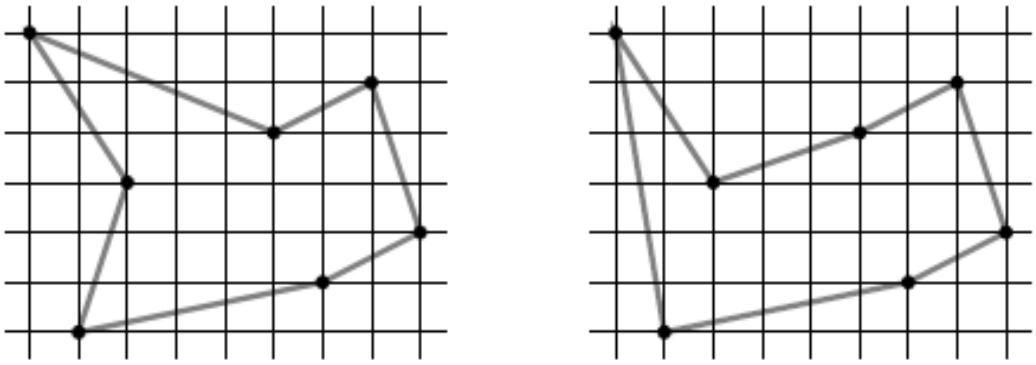


FIGURE 1 – **Sept points du plan, représentés sur une grille orthonormée. (a) La tournée la plus courte, d'une longueur de 24, 88 . . . Cette tournée n'est pas bitonique. (b) La tournée bitonique la plus courte pour le même ensemble de points. Sa longueur fait environ 25,58 . . .**

0.4.1 Solution naïve

La solution naïve consiste à énumérer tous les chemins possibles entre le point de départ et le dernier point à l'extrême droite du graphe. Le fait de passer par tous les cas possibles est assuré par un compteur allant de i=0 jusqu'à p=len(liste)-1, tel que, par exemple, pour i=4, notre algorithme calcule tous les chemins possibles entre 1 et ce dernier point à droite passant par 4 autres points. Prenons le graphe suivant : 1,2,3,4,5, pour i=2, l'algorithme retourne les chemins suivants : [1,2,3,5], [1,2,4,5] et [1,3,4,5]. Dans notre code, c'est la fonction **combinations** qui permet ceci. Une fois ces chemins calculés, le programme calcule le chemin retour correspondant à chacun des chemins calculés précédemment puis détermine le chemin optimal*.
***Voir annexe (figure 2 et 3)**

0.4.2 Solution efficace

Pour remédier à ce problème de temps d'exécution fastidieux, nous avons opté pour l'outil de programmation dynamique tout en adaptant une stratégie **bottom-up**, en effet, dans la première partie du trajet, c'est à dire entre le point de départ et le point le plus à droite, celui-ci calcule tous les sous-chemins et stocke leurs valeurs. Ainsi pour calculer des chemins plus longs il suffit de prendre les valeurs déjà calculées et y rajouter la valeur du bout de chemin manquant et stocke aussi sa valeur, donc nous évitons de recalculer la valeur d'un chemin plus qu'une et une seule fois. En outre, cela nous assure que toutes les valeurs que nous allons utiliser pour calculer le chemin retour, du point le plus à droite au point de départ, sont déjà calculées et stockées*. Dans ce cadre de la programmation dynamique, nous avons utilisé la structure optimale suivante : **chemin(i)=min(chemin(i+1), chemin(i+2), ... , chemin(n)+value_retour)** où value_retour est la valeur du chemin retour du point le plus à droite à notre point de départ, et on note aussi que nous avons **i<i+1<i+2<...<n**.

Ceci est résumé dans le tableau suivant :

Sommet / Itération	1	2	3	4	5	6	7
K=1		C(1-2)	C(1-3)	C(1-4)	C(1-5)	C(1-6)	C(1-7)
K=2			C(1-2-3)	C(1-2-4) C(1-3-4)	C(1-2-5) C(1-3-5) C(1-4-5)	C(1-2-6) C(1-3-6) C(1-4-6) C(1-5-6)	C(1-2-7) ...
K=3 ...				C(1-2-3-4)	C(1-2-3-5) C(1-2-4-5) C(1-3-4-5) ...	C(1-2-3-6) C(1-2-4-6) C(1-2-5-6) C(1-3-4-6) ...	C(1-2-3-7) ...

La colonne numéro 7 contient tous les chemins "aller" dont l'un sera choisi avec son retour comme étant le chemin optimal recherché.
***Voir annexe (figure 4 et 5)**

0.5 Calcul et tracé des complexités

Notre but étant de comparer les deux démarches, nous devons alors calculer la complexité de chaque algorithme, tracer sa courbe et interpréter le résultat. Le calcul de la complexité à été fait de la manière suivante : on considère un graphe pour lequel nous changeons le nombre de sommets et calculons le temps d'exécution à chaque reprise pour chacune des deux méthodes*et traçons sa courbe en utilisant **matplotlib***.
***Voir annexe (figure 6, 7 et 8)**

La complexité de l'algorithme de programmation dynamique étant en O(n^2), et celle de la méthode itérative, qui est notre solution naïve, étant exponentielle , nous remarquons alors qu'à chaque fois la méthode itérative prend plus de temps pour s'exécuter que la méthode de programmation dynamique, et qu'à chaque fois que nous augmentons le nombre de sommets, l'écart entre les deux courbes devient de plus en plus important.

0.6 Conclusion

Le problème du voyageur de commerce demeure jusqu'à nos jours l'un des problèmes les plus délicats en informatique et optimisation. D'après ce que nous avons vu précédemment, il n'existe pas d'algorithme avec un temps d'exécution raisonnable capable de trouver une solution efficace et optimale à ce problème. Et c'est pour cela, que nous optons toujours pour des cas particuliers du problème comme celui du **tour euclidien bitonique** afin de diminuer la complexité de l'algorithme et donc son temps d'exécution. Ce projet nous a donc permis de consolider nos connaissances en algorithmique et en programmation via les deux outils : **la programmation dynamique** et le **le langage Python** ; et ce en mettant en épreuve nos acquis en cours de structures de données et algorithmique afin d'essayer de trouver une solution à ce problème. En outre, ce projet nous a donné l'occasion de renforcer notre esprit d'analyse, de modélisation et de conception ; sans oublier notre capacité de gestion et de travail en groupe via le logiciel **gitHub** ainsi que d'autres logiciels de chat, surtout dans ces conditions par lesquelles nous passons dernièrement.

0.7 Remerciements

Avant de finir, nous tenons à remercier infiniment notre cher professeur et encadrant monsieur BENSAID Hicham pour tous les efforts fournis, son implication et tout le temps qu'il nous a accordé, non seulement durant la période du projet mais aussi tout au long de cette année universitaire.

Deuxième partie

Annexe

0.8 Annexe

Cette partie contient les différentes captures d'écran du code indexées précédemment.

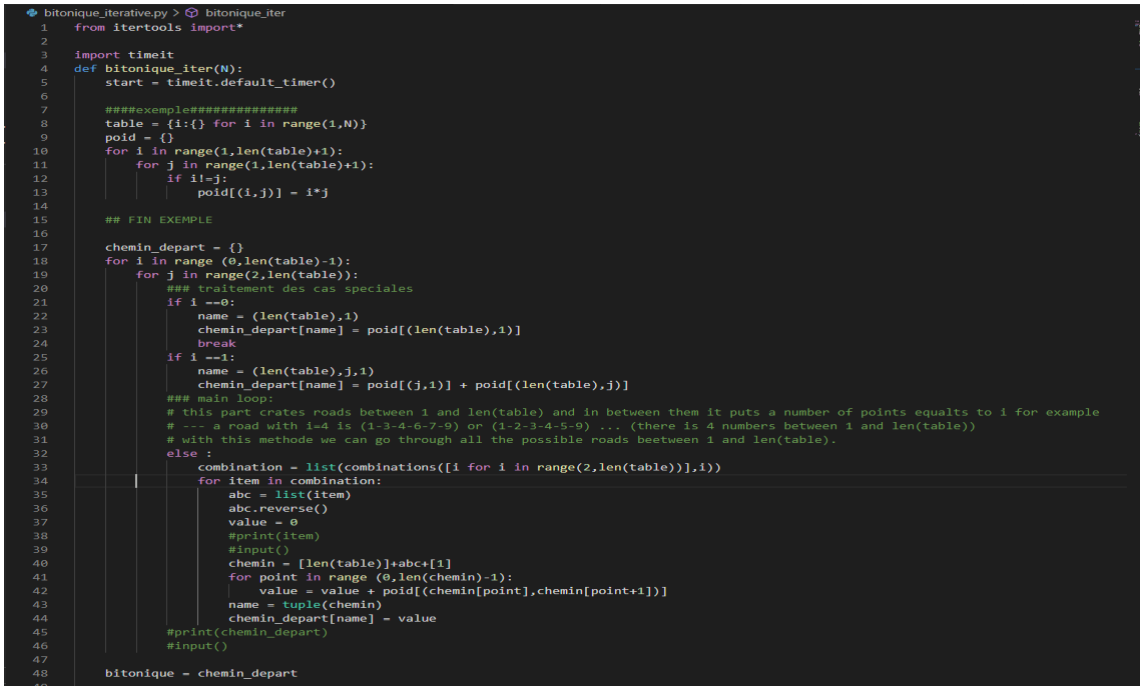
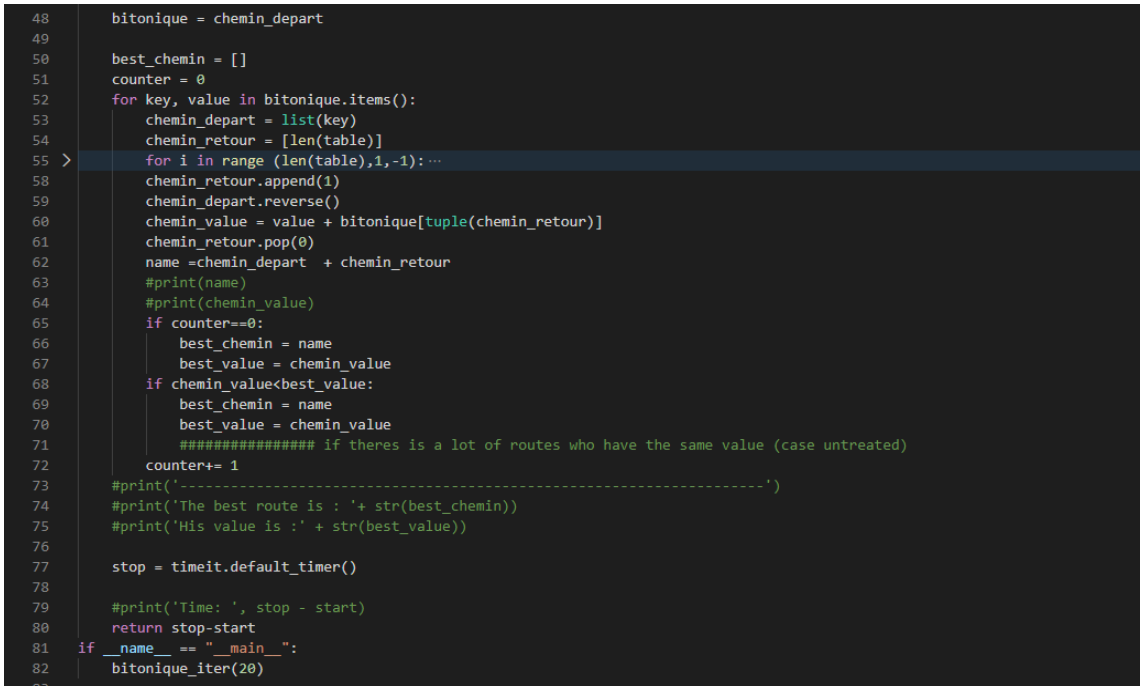


FIGURE 2 – **Implémentation de l'algorithme itératif en langage PYTHON**



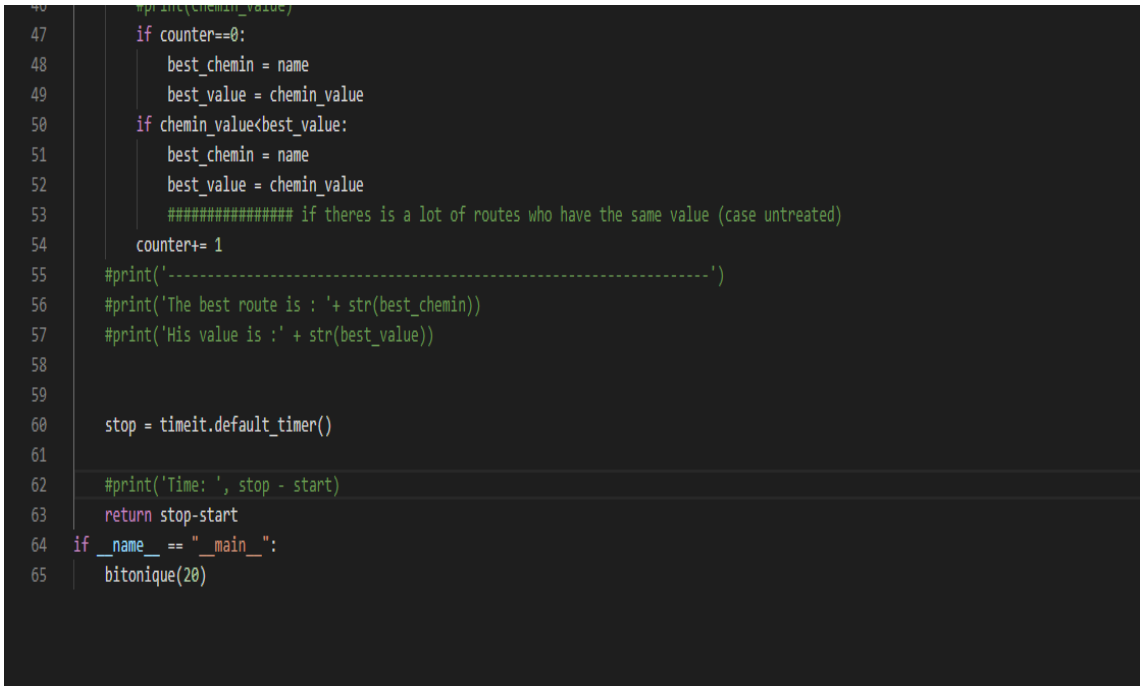
```
    bitonique = chemin_depart

    best_chemin = []
    counter = 0
    for key, value in bitonique.items():
        chemin_depart = list(key)
        chemin_retour = [len(table)]
        for i in range (len(table),1,-1):···
        chemin_retour.append(1)
        chemin_depart.reverse()
        chemin_value = value + bitonique[tuple(chemin_retour)]
        chemin_retour.pop(0)
        name =chemin_depart  + chemin_retour
        #print(name)
        #print(chemin_value)
        if counter==0:
            best_chemin = name
            best_value = chemin_value
        if chemin_value<best_value:
            best_chemin = name
            best_value = chemin_value
            ################ if theres is a lot of routes who have the same value (case untreated)
        counter+= 1
    #print('-----------------------------------------------------------------------')
    #print('The best route is : '+ str(best_chemin))
    #print('His value is :' + str(best_value))

    stop = timeit.default_timer()

    #print('Time: ', stop - start)
    return stop-start
if __name__ == "__main__":
    bitonique_iter(20)
```

FIGURE 3 – **Implémentation de l'algorithme itératif en langage PYTHON (suite)**

bitonique.py

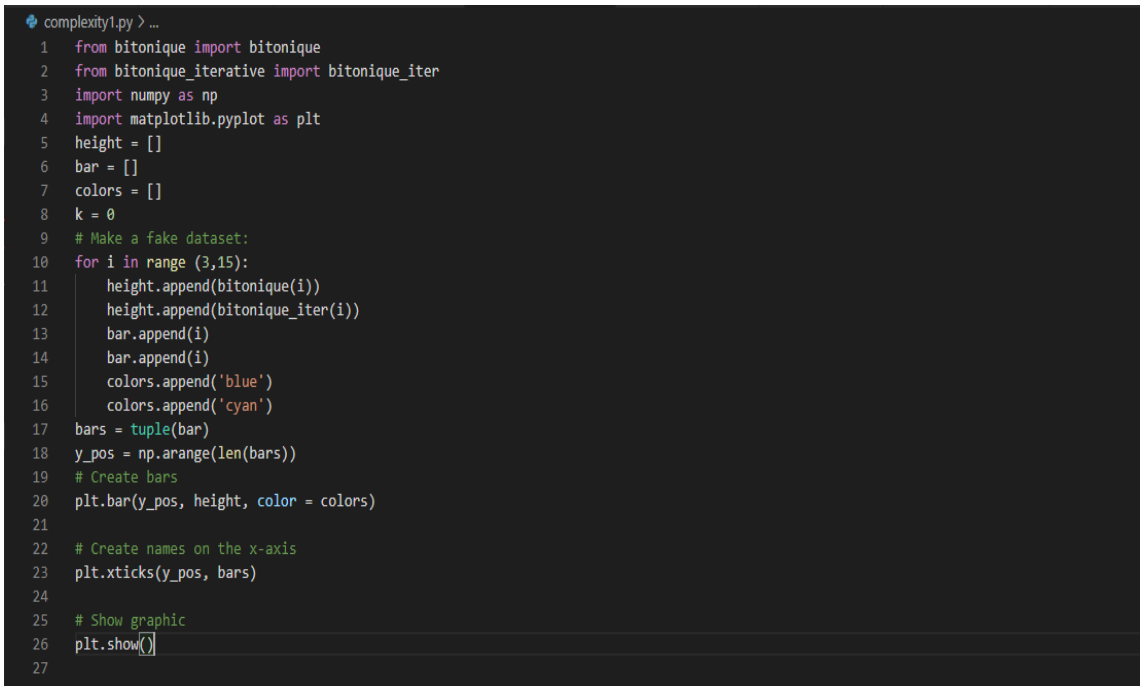
```
from itertools import*

import timeit
def bitonique(N):
    start = timeit.default_timer()
    ####exemple##############
    table = {i:{} for i in range(1,N)}
    poid = {}
    for i in range(1,len(table)+1):
        for j in range(1,len(table)+1):
            if i!=j:
                poid[(i,j)] = i*j

    ## FIN EXEMPLE

    ## THE DYNAMIC PART
    for i in range (2,len(table)+1):
        table[i][(i,1)] = poid[i,1]
        #print('i = ' + str(i))
        #input()
        for j in range (i-1,1,-1):
            #print('j = '+ str(j))
            #input()
            for key, value in table[j].items():
                name = [i] + list(key)
                index = tuple(name)
                table[i][index] = poid[i,j]+value

    bitonique = table[len(table)]
    best_chemin = []
    counter = 0
    for key, value in bitonique.items():
        chemin_depart = list(key)
        chemin_retour = [len(table)]
        #print(chemin_depart)
        #input()
        for i in range (len(table),1,-1):
            if i not in chemin_depart:
                chemin_retour.append(i)
        chemin_retour.append(1)
        chemin_depart.reverse()
        chemin_value = value + bitonique[tuple(chemin_retour)]
        chemin_retour.pop(0)
        name =chemin_depart  + chemin_retour
        #print(name)
        #print(chemin_value)
```

FIGURE 4 – **Implémentation de l'algorithme de programmation dynamique en langage PYTHON**



```
        if counter==0:
            best_chemin = name
            best_value = chemin_value
        if chemin_value<best_value:
            best_chemin = name
            best_value = chemin_value
            ############### if theres is a lot of routes who have the same value (case untreated)
        counter+= 1
    #print('-----------------------------------------------------------------')
    #print('The best route is : '+ str(best_chemin))
    #print('His value is :' + str(best_value))


    stop = timeit.default_timer()

    #print('Time: ', stop - start)
    return stop-start
if __name__ == "__main__":
    bitonique(20)
```

FIGURE 5 – **Implémentation de l'algorithme de programmation dynamique en langage PYTHON (suite)**



```
complexity1.py > ...
from bitonique import bitonique
from bitonique_iterative import bitonique_iter
import numpy as np
import matplotlib.pyplot as plt
height = []
bar = []
colors = []
k = 0
# Make a fake dataset:
for i in range (3,15):
    height.append(bitonique(i))
    height.append(bitonique_iter(i))
    bar.append(i)
    bar.append(i)
    colors.append('blue')
    colors.append('cyan')
bars = tuple(bar)
y_pos = np.arange(len(bars))
# Create bars
plt.bar(y_pos, height, color = colors)

# Create names on the x-axis
plt.xticks(y_pos, bars)

# Show graphic
plt.show()

```

FIGURE 6 – **Calcul de la complexité des deux algorithmes en utilisant le langage PYTHON**

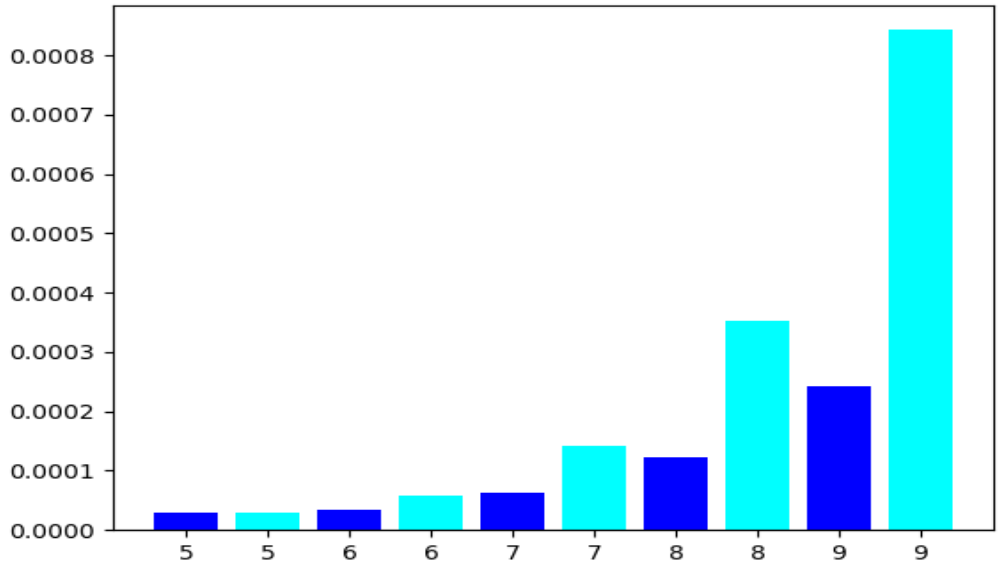


FIGURE 7 – **Le bleu représente le temps d'exécution de l'algorithme méthode de programmation dynamique tandis que le cyan représente la méthode itérative (solution naïve) en fonction de la taille du graphe**

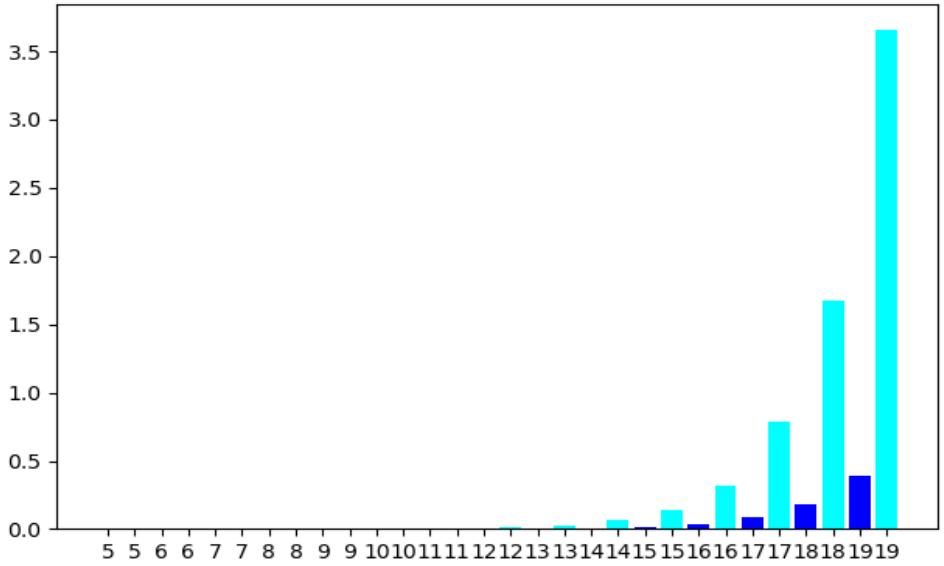


FIGURE 8 – **Le bleu représente le temps d'exécution de l'algorithme méthode de programmation dynamique tandis que le cyan représente la méthode itérative en fonction de la taille du graphe**