Institut national de statistique et d'économie appliquée

Filière Master
en Systemes d'information et Systemes Intelligents(M2SI)

Rapport de projet de programmation dynamique

Résolution de problèmes de voyageur de commerce par programmation dynamique et GVNS

Présenté par :
BOUYARMANE Chaimae
EL AYADI Fatima Ezzahra
ELYAZIJI Boutaina
Supervisé par :
M.BENMANSOUR Rachid

Année Universitaire 2022-2023

Résumé

L'objectif de notre projet est de résoudre le problème du voyageur de commerce. Il s'agit d'un problème mathématique qui consiste, étant donné un ensemble des villes séparées par des distances données, à trouver le plus court chemin qui relie toutes les villes.

C'est ainsi un problème d'optimisation pour lequel nous ne connaissons pas l'algorithme permettant de trouver une solution exacte en un temps polynomial. De plus, la version décisionnelle de l'énoncé (pour une distance D, existe-t-il un chemin plus court que D passant par toutes les villes ?) est connue comme étant un problème NP-complet.

Nous avons mis en place plusieurs méthodes pour essayer de résoudre ce problème : une méthode exacte représenter par la programmation dynamique ainsi que une autre méthode méta-heuristique (GVNS).

Mots clés : TSP, NP-complet, programmation dynamique, optimisation, méta-heuristique, GVNS

Abstract

The objective of our project is to solve the problem of the traveling salesman. It is a mathematical problem which consists, given a set of cities separated by given distances, in finding the shortest path which connects all the cities.
It is thus an optimization problem for which we do not know the algorithm allowing to find an exact solution in a polynomial time. Moreover, the decision version of the statement (for a distance D, is there a path shorter than D passing through all the cities ?) is known to be an NP-complete problem.
We have implemented several methods to try to solve this problem : an exact method represented by dynamic programming as well as another metaheuristic method (GVNS).

Keywords : TSP, NP-complete, dynamic programming, optimization, metaheuristic, GVNS

Table des Abréviations

Abréviation	Signification
TSP	Traveling Salesman Problem
GVNS	General Variable Neighborhood Search
VNS	Variable Neighborhood Search
RVNS	Reduced Variable Neighborhood Search
VND	Variable Neighborhood Descent
VScode	Visual Studio Code

Table des matières

Table des figures

Introduction

Un voyageur de commerce désire visiter un certain nombre de villes, débutant et finissant son parcours dans la même ville en visitant chacune des autres villes une et une seule fois. Il désire sélectionner la tournée qui minimise la distance totale parcourue. Ce problème est connu sous le nom du problème du voyageur de commerce (en anglais travelling salesman problem : TSP) et est NP-Complet. La complexité en temps des algorithmes exacts proposés croît exponentiellement avec n (la taille du problème ou le nombre de villes).

Dans le cadre de notre formation à l'Institut National de la Statistique et de l'Economie Appliquée, nous avons été amenés à réaliser un projet de résolution de problèmes de TSP en éléments de programmation dynamique.

Notre projet consiste à réaliser une application web qui permette à l'utilisateur de trouver le chemin le plus court et la distance minimale à travers une ville donnée, afin qu'il puisse choisir entre deux méthodes soit la méthode de programmation dynamique ou bien la méta-heuristique. En se basant sur le langage orienté objet Python.

Ce rapport s'oriente autour de quatre axes principaux :

— Le premier chapitre décrit une présentation et la problématique ainsi qu'une analyse des différents outils et langages utilisés.

— Le deuxième chapitre est consacré à la résolution du problème par la programmation dynamique.

— Le troisième chapitre désigne la résolution du problème par la métaheuristiques.

— Le quatrième chapitre représente une comparaison entre les deux méthodes.

En conclusion nous donnerons un bilan de notre projet, de son avancement et des difficultés rencontrées.

Chapitre 1

Contexte Général Du Projet

Un voyageur de commerce désire visiter un certain nombre de villes, débutant et finissant son parcours dans la même ville en visitant chacune des autres villes une et une seule fois. Il désire sélectionner la tournée qui minimise la distance totale parcourue. Ce problème est connu sous le nom du problème du voyageur de commerce (en anglais travelling salesman problem : TSP) est NP-Complet. La complexité en temps des algorithmes exacts proposés croît exponentiellement avec n (la taille du problème ou le nombre de villes). Plusieurs méthodes d'approximation (heuristiques 1) ont été proposées qui approchent en temps raisonnable la solution optimale.

1.1 Problème du voyageur de commerce

1.1.1 Le contexte du problème du voyageur de commerce

Le problème du voyageur de commerce (TSP) a été formulé en 1930, mais c'est encore aujourd'hui l'un des problèmes d'optimisation combinatoire les plus étudiés.

En 1972, Richard Karp a prouvé que le problème du cycle hamiltonien était NP-complet qui est une classe de problèmes d'optimisation combinatoire. Cela signifie que le TSP était NP-difficile et que la complexité du calcul du meilleur itinéraire augmentera de façon exponentielle lorsque davantage de destinations s'ajouteront au problème.

Par exemple, il y a trois itinéraires possibles s'il y a quatre villes, mais il y a 360 itinéraires possibles s'il y a six villes. En effet, les scientifiques ne calculent pas seulement le chemin le plus efficace, mais aussi celui qui fonctionne. Ce calcul utilise l'approche de la force brute pour résoudre le problème en déterminant chaque possibilité, puis en choisissant la meilleure. En d'autres termes, recherchez chaque chemin unique que le vendeur pourrait emprunter.

Même si les difficultés de calcul augmentent avec chaque ville ajoutée à l'itinéraire, les informaticiens ont pu calculer la solution optimale à ce problème pour des milliers de villes depuis le début des années 90. Par exemple, de nombreuses villes d'un État

FIGURE 1.1.1 – TSP

américain pourraient faire partie de la zone de livraison du grand fournisseur de services logistiques. Déterminer l'itinéraire le plus court entre tous les arrêts que le véhicule doit effectuer au quotidien permettrait d'économiser beaucoup de temps et d'argent.

1.1.2 Pourquoi le problème du voyageur de commerce est-il si difficile à résoudre ?

En théorie, le TSP est facile à énoncer et peut être résolu en vérifiant chaque itinéraire aller-retour pour trouver le plus court. Cependant, à mesure que le nombre de villes augmente, le nombre correspondant d'allers-retours dépasse les capacités des ordinateurs les plus rapides.

Avec 10 villes, il peut y avoir plus de 300 000 permutations et combinaisons aller-retour. Avec 15 villes, le nombre d'itinéraires possibles pourrait dépasser 87 milliards. Dans les premiers jours de l'informatique, les informaticiens espéraient que quelqu'un trouverait un algorithme ou une approche améliorée pour résoudre le problème du voyageur de commerce dans un laps de temps raisonnable. Cependant, alors que les scientifiques ont fait des progrès avec des scénarios spécifiques, il n'y avait pas quelqu'un pour résoudre le problème efficacement pour les vendeurs itinérants. Un algorithme unique pour tous n'est peut-être pas possible.

1.2 Les outils et langages utilisés

Anaconda :

Anaconda est une distribution des langages de programmation Python et R pour le calcul scientifique (science des données, applications d'apprentissage automatique,

FIGURE 1.2.1 – Anaconda

traitement de données à grande échelle, analyse prédictive, etc.), qui vise à simplifier la gestion et le déploiement des packages. La distribution comprend des packages de science des données adaptés à Windows, Linux et macOS. Il est développé et maintenu par Anaconda, qui a été fondée par Peter Wang et Travis Oliphant en tant que produit Anaconda, il est également connu sous le nom d'Anaconda Distribution ou Anaconda Individual Edition, tandis que les autres produits de la société sont Anaconda Team Edition et Anaconda Enterprise Edition, qui ne sont pas gratuits.

Jupyter :

FIGURE 1.2.2 – Jupyter

L'application Jupyter Notebook vous permet de créer et de modifier des documents qui affichent l'entrée et la sortie d'un script en langage Python ou R.

Python :

FIGURE 1.2.3 – python

Python est un langage de script de haut niveau, interprété, interactif et orienté objet. Python est conçu pour être très lisible. Il utilise fréquemment des mots-clés anglais alors que d'autres langues utilisent la ponctuation, et il a moins de constructions syntaxiques que d'autres langues.

Chapitre 2

Problème du voyageur de commerce avec Programmation Dynamique

2.1 Introduction

La solution de programmation dynamique est l'un des meilleurs moyens de résoudre ces termes en termes d'exécution et de convertir l'ordre temporel du problème sous forme polynomiale. Le problème avec les problèmes de programmation dynamique est leur consommation de mémoire, et dans les gros problèmes, le système ne peut pas satisfaire les exigences de programmation dynamique.

2.2 Programmation Dynamique

La programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. Le concept a été introduit au début des années 1950 par Richard Bellman. À l'époque, le terme « programmation » signifie planification et ordonnancement. La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires. Elle a d'emblée connu un grand succès, car de nombreuses fonctions économiques de l'industrie étaient de ce type, comme la conduite et l'optimisation de procédés chimiques, ou la gestion de stocks.

2.2.1 Le principe d'optimalité

La programmation dynamique est largement utilisée pour les problèmes d'optimisation. La condition de base pour l'utilisation de la méthode de calcul du cas optimal est appelée le principe d'optimalité. Le principe d'optimalité consiste à résoudre un problème de manière optimale, en incluant des solutions optimales à tous les sous-problèmes. En d'autres termes, le problème doit être tel qu'en trouvant sa solution optimale, la solution optimale à tous les sous-problèmes puisse également être obtenue. Par exemple, pour trouver le chemin le plus court entre deux villes, le point de départ sur le chemin optimal

et le chemin entre chaque nœud est le chemin optimal entre ces villes.

la programmation dynamique prend le contre-pied de cette approche et cherche à factoriser autant que possible les calculs pour faire le minimum de travail possible à chaque étape et éviter tout raisonnement redondant. La programmation dynamique n'énumère ainsi pas toutes les solutions séparément les unes des autres, mais filtre l'ensemble des solutions pour énumérer des groupes de solutions. Il s'agit ainsi d'une énumération factorisée et complète.

2.3 L'algorithme

La solution la plus directe dans la programmation dynamique serait d'essayer toutes les permutations (combinaisons ordonnées) et de voir laquelle est la moins chère (en utilisant la recherche par force brute). Le temps d'exécution a un compléxité de l'ordre O(n !), la factorielle du nombre de villes, de sorte que cette solution devient impraticable, même pour seulement 20 villes.
L'une des premières applications de la programmation dynamique est l'algorithme de Held – Karp qui résout le problème dans le temps $O(n^2 2^n)$.

2.3.1 Held–Karp algorithme

L' algorithme de Held-Karp , également appelé algorithme de Bellman-Held-Karp , est un algorithme de programmation dynamique proposé en 1962 indépendamment par Bellman et par Held et Karp pour résoudre le problème du voyageur de commerce (TSP), dans lequel l'entrée est une matrice de distance entre un ensemble de villes, et l'objectif est de trouver une visite de longueur minimale qui visite chaque ville exactement une fois avant de revenir au point de départ.

2.3.2 Description et motivation de l'algorithme

Numéroter les villes $\mathbf{1, 2, ..., n}$, avec $\mathbf{1}$ désignée arbitrairement comme ville de "départ". L'algorithme Held-Karp commence par calculer, pour chaque ensemble de villes $\boldsymbol{S \subseteq (2, ..., n)}$ et chaque ville $\neq \mathbf{1}$ non contenu dans $\boldsymbol{S}$, le plus court chemin à sens unique depuis $\mathbf{1}$ à $\boldsymbol{e}$ qui traverse toutes les villes de $\boldsymbol{S}$ dans un certain ordre (mais pas à travers d'autres villes). Dénoter cette distance $\boldsymbol{g}$(S,e), et écris $\boldsymbol{d}$(u,v) pour la longueur du bord direct de $\boldsymbol{u}$ à $\boldsymbol{v}$. Nous allons calculer les valeurs de $\boldsymbol{g}$(S,e) en commençant par les plus petits ensembles $\boldsymbol{S}$ et en terminant par le plus grand.

Plus généralement, supposons $\boldsymbol{S = \{s_1, \ldots, s_k\}}$ est un ensemble de $\boldsymbol{k}$ villes. Pour tout entier $\boldsymbol{1 \leqslant i \leqslant k}$, écrivez $\boldsymbol{S_i = S \setminus \{s_i\} = \{s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_k\}}$ pour l'ensemble créé en supprimant $\boldsymbol{s_i}$ de $\boldsymbol{S}$. Alors si le chemin le plus court depuis $\mathbf{1}$ à travers $\boldsymbol{S}$ à $\boldsymbol{e}$ a $\boldsymbol{s_i}$ en tant qu'avant-dernière ville, la suppression du dernier bord de ce chemin doit donner le chemin le plus court depuis $\mathbf{1}$ à $\boldsymbol{s_i}$ à travers $\boldsymbol{S_i}$. Cela

signifie qu'il n'y a que k chemins les plus courts possibles depuis 1 à e à travers S, un pour chaque avant-dernière ville possible s_i avec longueur $g(S_i, s_i) + d(s_i, e)$, et $g(S, e) = \min_{1 \leq i \leq k} g(S_i, s_i) + d(s_i, e)$.

Cette étape de l'algorithme se termine lorsque $g(\{2, \ldots, i-1, i+1, \ldots, n\}, i)$ est connu pour tout entier $2 \leq j \leq n$, donnant la distance la plus courte de la ville 1 à la ville i qui traverse toutes les autres villes. La deuxième étape beaucoup plus courte ajoute ces distances aux longueurs de bord $d(i, 1)$ pour donner $n - 1$ cycles les plus courts possibles, puis trouve le plus court.

Le chemin le plus court lui-même (et pas seulement sa longueur), enfin, peut être reconstruit en stockant à côté $g(S, e)$ l'étiquette de l'avant-dernière ville sur le chemin de 1 à e à travers S, n'augmentant les besoins en espace que d'un facteur constant.

```
fonction TSP (G, n)
    pour k := 2 à n faire
        g({k}, k) := d(1, k)
    fin pour

    pour s := 2 à n-1 faire
        pour tout S ⊆ {2, ..., n}, |S| = s faire
            pour tout k ∈ S faire
                g(S, k) := min m≠k,m∈S [g(S\{k}, m) + d(m, k)]
             fin pour
        fin pour
    fin pour

    opt := min k≠1 [g({2, 3, ..., n}, k) + d(k, 1)]
     return (opt)
 end function
```

FIGURE 2.3.1 – Pseudocode de TSP

```python
def solve_tsp_dynamic(
    distance_matrix: np.ndarray,
    maxsize: Optional[int] = None,
) -> Tuple[List, float]:
    N = frozenset(range(1, len(distance_matrix)))
    memo: Dict[Tuple, int] = {}
    @lru_cache(maxsize=maxsize)
    def dist(ni: int, N: frozenset) -> float:
        if not N:
            return distance_matrix[ni, 0]

        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)

    ni = 0
    solution = [1]
    while N:
        ni = memo[(ni, N)]
        solution.append(ni+1)
        N = N.difference({ni})
    solution.append(1)

    return solution, best_distance
```

FIGURE 2.3.2 – Pseudocode de TSP

Chapitre 3

Problème de voyageur de commerce avec la méthode Méta-Heuristique (GVNS)

3.1 Méta-Heuristiques

Une **méta-heuristique** est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile. C'est un concept utilisé entre autres en optimisation combinatoire, en théorie des graphes, en théorie de la complexité des algorithmes, en intelligence artificielle, dans la programméta-mation des jeux (comme les échecs ou go), dans la primalité des nombres entiers et dans la démonstration de théorème.

Une **méta-heuristique** s'impose quand les algorithmes de résolution exacte sont impraticables, à savoir de complexité polynomiale de haut degré, exponentielle ou plus. Généralement, une méta-heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre, mais il existe des approches fondées sur des principes généraux.

3.2 La recherche à voisinage variable

La recherche de voisinage variable (VNS), proposée par Mladenović et Hansen en 1997, est une méthode métaheuristique pour résoudre un ensemble de problèmes d'optimisation combinatoire et d'optimisation globale. Il explore des quartiers éloignés de la solution actuelle en place, et passe de là à un nouveau si et seulement si une amélioration a été apportée. La méthode de recherche locale est appliquée à plusieurs reprises pour passer des solutions du voisinage aux optima locaux. VNS a été conçu pour approximer des solutions de problèmes d'optimisation discrets et continus et selon ceux-ci, il est destiné à résoudre des problèmes de programmes linéaires, des problèmes de programmes entiers, des problèmes de programmes mixtes entiers, des problèmes de programmes non linéaires

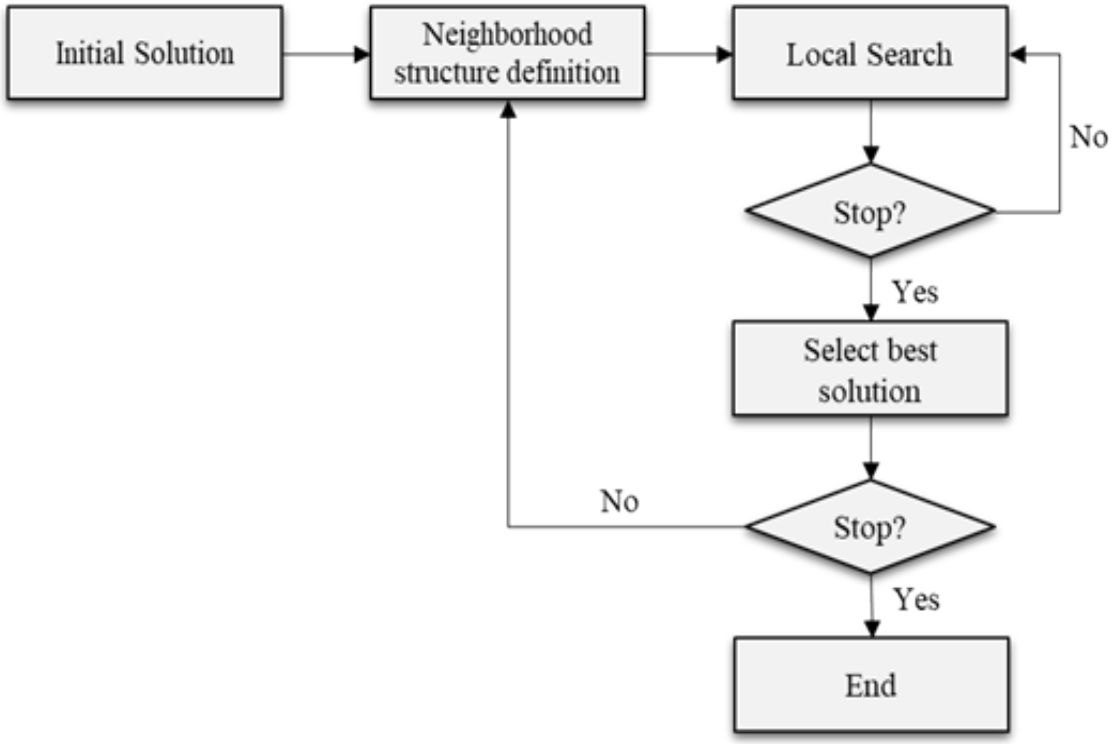
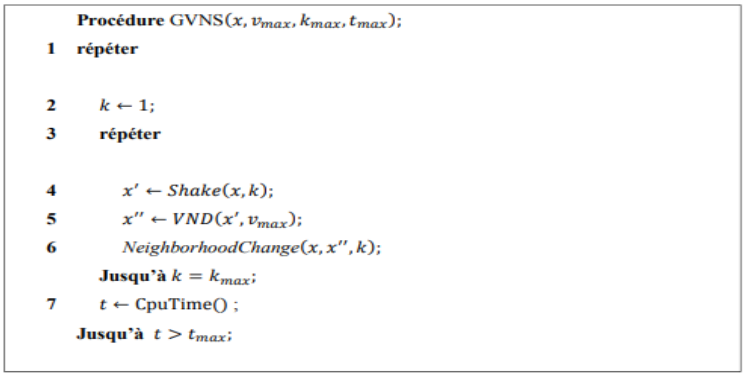


FIGURE 3.2.1 – VNS

3.3 Générale de la recherche à voisinage variable(GVNS)

Cette méthode est composée de deux phases. La première phase, dite constructive, se charge de trouver une solution réalisable à l'aide d'une heuristique de recherche à voisinage variable (VNS pour Variable Neighborhood Search). Cette nouvelle méthode s'avère plus efficace pour trouver une solution réalisable. La deuxième phase, dite d'amélioration, se charge d'améliorer la solution trouvée dans la première partie. L'algorithme utilise également un prétraitement de base pour éliminer des arcs incompatibles (nous rappelons qu'un arc incompatible est un arc qui ne peut pas apparaitre dans une solution réalisable). Les résultats obtenus améliorent les meilleures solutions connues pour quelques instances.
Dans les pages suivantes vous trouverez l'algorithme et l'implémentation du GVNS.

L'algorithme de GVNS :



Procédure GVNS($x, v_{max}, k_{max}, t_{max}$);
1 **répéter**
2 $k \leftarrow 1$;
3 **répéter**
4 $x' \leftarrow Shake(x, k)$;
5 $x'' \leftarrow VND(x', v_{max})$;
6 $NeighborhoodChange(x, x'', k)$;
Jusqu'à $k = k_{max}$;
7 $t \leftarrow$ CpuTime() ;
Jusqu'à $t > t_{max}$;

FIGURE 3.3.1 – Générale de VNS

```
def GVNS(x, t=5, k_max=3, l_max=2):
    start_time = time.time()
    x=RVNS(x,k_max,0.2)
    while time.time() - start_time < t*60:
        k=1
        while k <= k_max:
            x1 = shake(x,k)
            x2 = VND(x1, l_max)
            x, k = change_neighborhood(x, x2, k)
    return x,f(x)
```

FIGURE 3.3.2 – Code de GVNS

L'algorithme de VNS réduite :

La méthode VNS réduite (RVNS) est obtenue si des points aléatoires sont sélectionnés de Nk(x) et aucune descente n'est effectuée. Au contraire, les valeurs de ces nouveaux points sont par rapport à celui de l'opérateur historique et une mise à jour a lieu en cas d'amélioration. On suppose aussi qu'une condition d'arrêt a été choisie comme la temps CPU maximal autorisé tmax ou le nombre maximal d'itérations entre deux améliorations. Pour simplifier la description des algorithmes, nous utilisons toujours tmax ci-dessous. Par conséquent, RVNS utilise deux paramètres : tmax et kmax.

```
Procédure RVNS(x,kmax,tmax)
     répété
          k <- 1
          répété
               x' <- Shake(x,k)
               x,k <- NeighborhoodChange(x,x',k)
          jusqu'à k=kmax
          t <-  CpuTime()
     jusqu'à t > tmax
return x
```

FIGURE 3.3.3 – VNS réduite

```python
k_max = 3
def RVNS(x, k_max, t = 5):
    start_time = time.time()
    while time.time() - start_time < t*60:
        k=1
        while k <= k_max:
            xp = shake(x,k)
            x, k = change_neighborhood(x, xp, k)
    return x
```

FIGURE 3.3.4 – Code de RVNS

L'algorithme de descente de voisinage variable (VND) :
La méthode de descente de voisinage variable (VND) est obtenue si un changement de neighborhoods est réalisée de manière déterministe. Il est présenté dans l'algorithme, où les voisinages sont notés Nk,k = 1,...,kmax.

Procédure VND(x, v_{max});

1 **répéter**

2 $v \leftarrow 1$;

3 **répéter**

4 $x' \leftarrow \arg min_{y \in V_v(x)} f(x)$ /* Trouver le meilleur voisin dans $V_v(x)$ */;

5 *NeighborhoodChange*(x, x', v) ;

jusqu'à $v = v_{max}$;

jusqu'à *pas d'amélioration possible*;

FIGURE 3.3.5 – VND

```
l_max=2
def VND(x, l_max):
    l = 1
    while l <= l_max:
        xp = shake(x, l)
        xp = first_improvement(x, l)
        x,l = change_neighborhood(x, xp, l)
    return x
```

FIGURE 3.3.6 – Code de VND

L'algorithme de la Recherche Locale :

La VNS est fondée selon trois idées principales : un minimum local par rapport à un certain voisinage n'est pas forcément un minimum local par rapport à un autre voisinage ; un minimum global est un minimum local par rapport à tous les voisinages possibles ; pour plusieurs problèmes, les minima locaux par rapport à un ou plusieurs voisinages sont relativement proches l'un de l'autre.

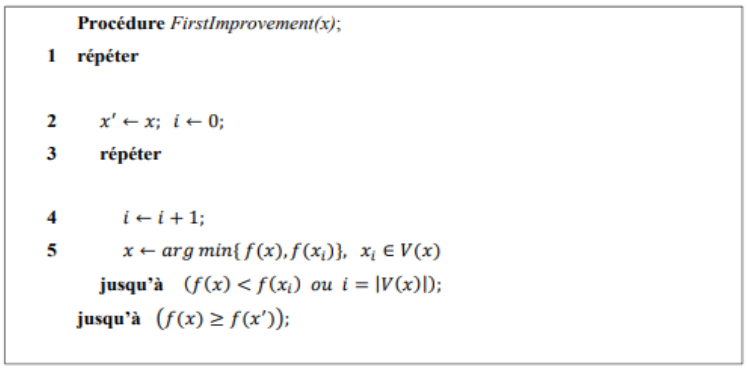


FIGURE 3.3.7 – Algorithme de la Recherche Locale

```
def first_improvement(x, l):
    N=neighbrehood(x,l)
    for i in range(0,len(N)):
        if f(N[i])< f(x):
            x=N[i]
            break
    return x
```

FIGURE 3.3.8 – Code de first improvement

L'algorithme de changement de voisinage :

Cette fonction qu'on appelle NeighborhoodChange permet un changement systématique de voisinage. En effet, étant donné une solution 'actuelle' x ou un point actuel (incumbent point), dont la valeur de la fonction objectif est f(x), et un nombre entier k, la fonction NeighborhoodChange compare la valeur f(x) à la valeur f(x') d'une autre solution x'. Si f(x')<f(x) alors on réinitialise k (k=1) et x' devient la nouvelle solution 'actuelle', sinon on incrémente la valeur de k (k=k+1) et la solution 'actuelle' reste inchangée (i.e., x continue à être la solution 'actuelle').

Procédure *NeighborhoodChange* (x, x', k);

1 **si** $f(x') < f(x)$ **alors**

2 $x \leftarrow x'$; $k \leftarrow 1$ /* Modifier la solution actuelle */;

sinon

3 $k \leftarrow k + 1$;

fin

FIGURE 3.3.9 – Algorithme de changement de voisinage

```
def change_neighborhood(x, xp, k):
    if f(xp) < f(x):
        x = xp
    else:
        k += 1
    return x,k
```

FIGURE 3.3.10 – Code de NeighborhoodChange

L'algorithme de la fonction shake :

Cette méthode sélectionne aléatoirement l'un des voisinages prédéfinis et l'applique k fois (1 < k < kmax, où kmax est le nombre maximum de itérations shaking) dans la solution actuelle.

Procédure **Shake**(x,k)
w <- [1+Rand(0,1)*|Nk(x)|]
x' <- x^w
return x'

FIGURE 3.3.11 – Algorithme de la fonction shake

```
def shake(x, k):
    N=neighbrehood(x,k)
    xp=rd.choice(N)
    return xp
```

FIGURE 3.3.12 – Code de la fonction shake

L'algorithme de la fonction neighborhood :

Le voisinage Nk(x) désigne l'ensemble des solutions dans le kième voisinage de x.

```
def neighbrehood(x, k):
    bound = len(x)
    N=[]
    if(k==3):
        for i in range(1,bound-2):
            for j in range(i+1,bound-1):
                N.append(NS_swapping(x,i,j))
    elif(k==2):
        for i in range(1,bound-2):
            for j in range(i+1,bound-1):
                N.append(NS_insertion_before(x,i,j))
    elif(k==1):
        for i in range(1,bound-2):
            for j in range(i+1,bound-1):
                N.append(NS_two_opt(x,i,j))
    return N

global voisins
```

FIGURE 3.3.13 – Code de la fonction neighbrehoo

3.4 La méthode two-opt

L'algorithme 2-opt est l'une des heuristiques les plus basiques et les plus largement utilisées pour obtenir une solution approximative du problème TSP. 2-opt commence par une tournée initiale aléatoire et améliore progressivement la tournée en échangeant 2 arêtes de la tournée avec deux autres arêtes.

FIGURE 3.4.1 – Two-Opt

A chaque étape, l'algorithme 2-opt supprime deux arêtes **(u1,u2)** et **(v1,v2)**, où **u1,v1,u2,v2** sont distincts, créant ainsi 2 sous-tours et reconnecte le tour avec les arêtes **(u1,v1)** et **(u2,v2)** dans le cas où le remplacement réduit la longueur du tour (2 arêtes ne peuvent pas être incidentes au même nœud). Le coût final de la solution du mouvement unique à 2 options peut être exprimé par $\Delta ij = c(u1, u2)+c(v1, v2)(c(u1, v1) + c(u2, v2))$

L'algorithme se termine par l'optimum local, dans lequel aucune autre étape d'amélioration n'est trouvée. Un mouvement à deux options a une complexité temporelle de $O(n^2)$.

- **Structures de Neighborhood :**

Trois opérateurs de recherche locale sont considérés pour explorer différentes solutions :
1. NS two opt : Le mouvement 2 Opt casse deux arcs dans la solution actuelle et les reconnecte d'une manière différente.
2. NS swapping : Ce mouvement permute deux nœuds dans la route actuelle.
3. NS insertion before : Ce déplacement supprime le nœud i de son emplacement actuel position dans la route et le réinsère avant un nœud sélectionné b.

```python
def NS_swapping(x, lb, ub): #lb>0
    bound = len(x)
    xc = None
    if (lb < bound and ub < bound):
        xc = x.copy()
        xc[lb], xc[ub] = xc[ub], xc[lb]
    return xc

def NS_insertion_before(x, lb, ub):  #lb>0
    bound = len(x)
    xc = None
    if (lb < bound and ub < bound):
        xc = x.copy()
        xc.insert(lb, x[ub])
        xc.pop(ub+1)
    return xc


def NS_two_opt(x, lb, ub):
    bound = len(x)
    x1=[]
    if lb < ub and (0 < lb < bound-1 and 0 < ub < bound-1) :
        x1=x[:lb]
        x1.extend(reversed(x[lb:ub+1]))
        x1.extend(x[ub+1:])
    return x1
```

FIGURE 3.4.2 – Structures de Neighborhood

Chapitre 4

Comparaison

4.1 Introduction

Dans cette partie nous allons faire la comparaison entre la méthode de programmation dynamique et entre la méthode métaheuristique à trois instances.

4.2 Resultats :

		Instance 1	Instance 2	Instance 3
Programmation Dynamique		Time : 2 secondes Cout : 179	Time : 6 minutes Cout : 308	———
Meta-heuristique	1 minute	Cout : 179	Cout : 308	Cout :1961
	4 minutes	Cout : 179	Cout : 314	Cout :1879
	8 minutes	Cout : 179	Cout :336	Cout :1894

FIGURE 4.2.1 – Comparaison

4.3 Conclusion :

Sur la base des comparaisons que nous avons faites, nous avons constaté que la méthode de programmation dynamique donne une solution exacte, mais qu'elle nécessite beaucoup de temps et d'espace mémoire. D'un autre côté, la méthode méta-heuristique ne donne qu'une solution approximative, et si vous voulez améliorer la solution, vous devez passer plus de temps.

Chapitre 5

Mise en oeuvre d'une application web

Dans cette partie nous allons présenter l'interface de notre projet qui est une "sidebar" qui contient quatre champs "about", "DP Approach", "GVNS Approach" et "Contact".

5.1 Zone de la résolution par la programmation dynamique :

Dans cette partie l'utilisateur mettra le fichier qui va travailler dessus. ensuite, il sélectionnera la ville de départ et lancera le calcul.

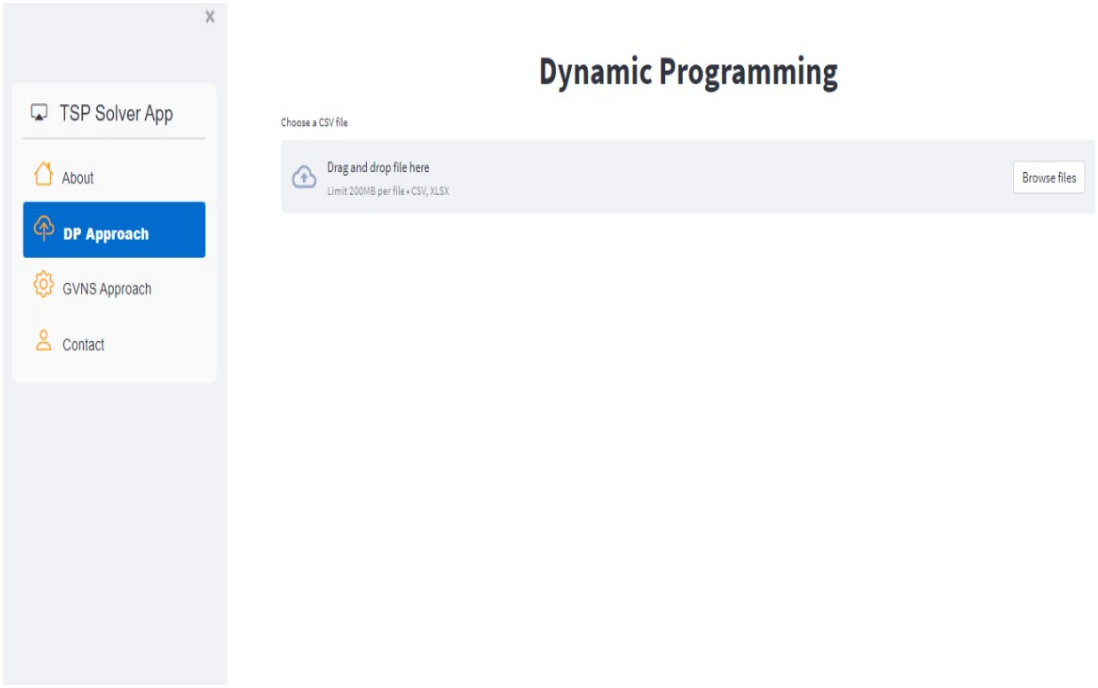


FIGURE 5.1.1 – Importer fichier

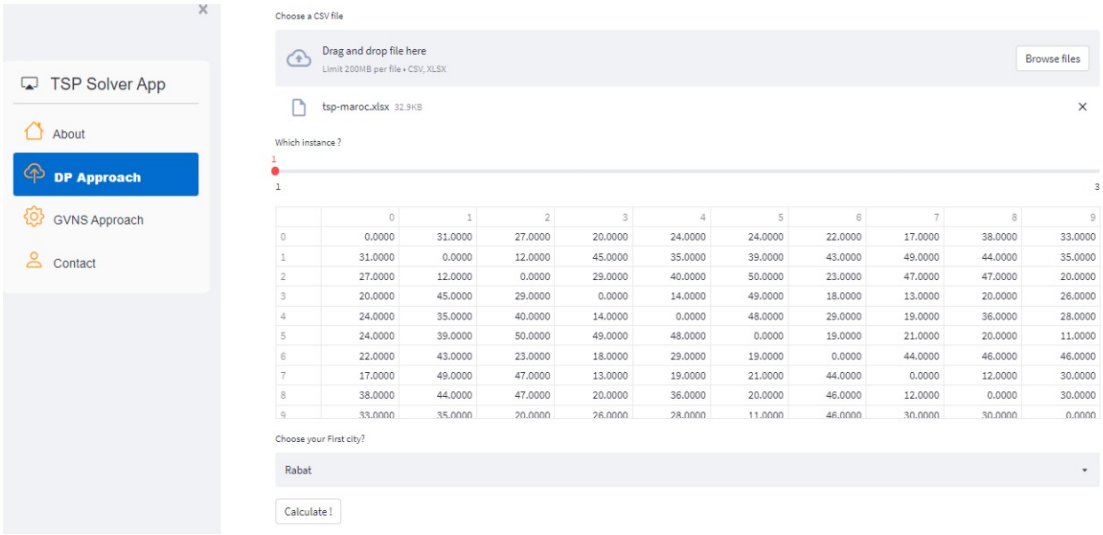


FIGURE 5.1.2 – choix de la ville

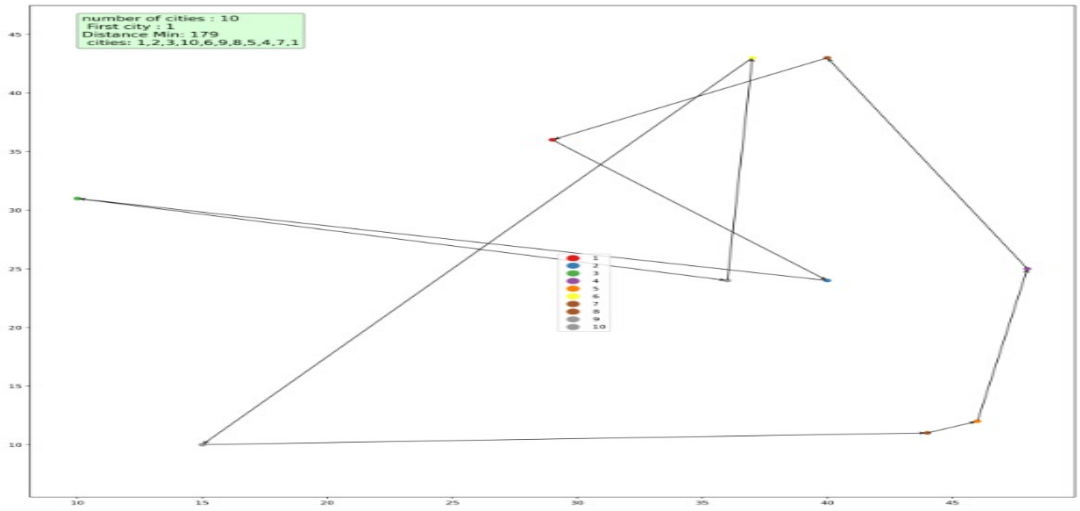


FIGURE 5.1.3 – Résultat

5.2 Zone de la résolution par gvns :

Dans cette partie l'utilisateur suivra les mêmes étapes de programmation dynamique sauf qu'il est nécessaire d'entrer le temps d'exécution.

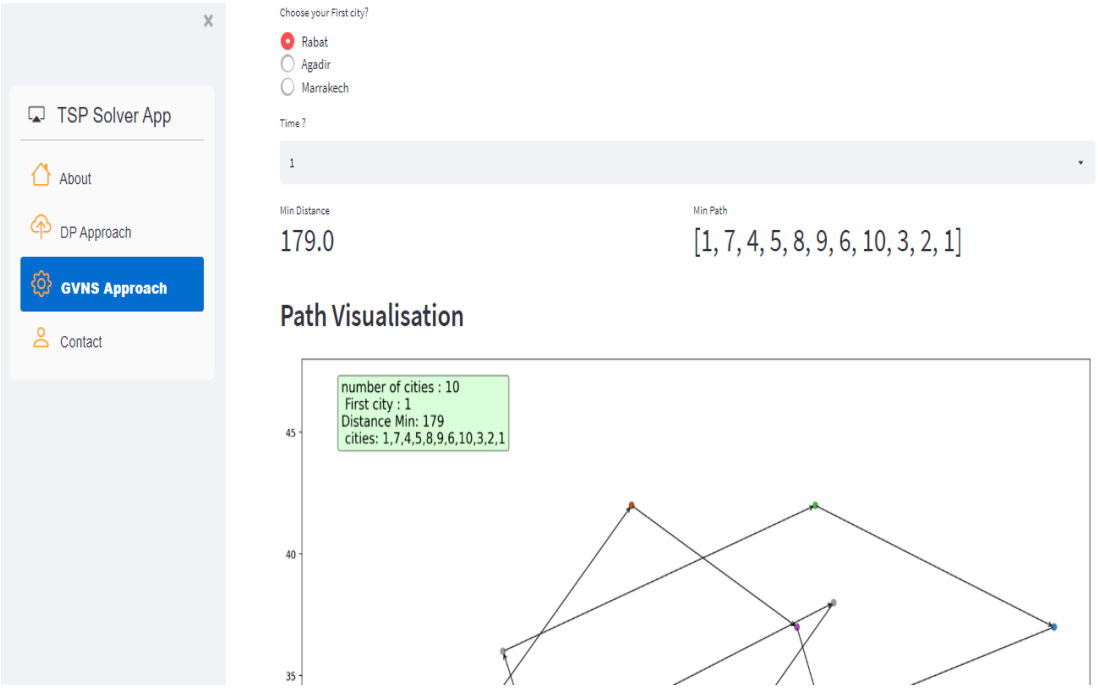


FIGURE 5.2.1 – le choix de fichier , ville et le temps

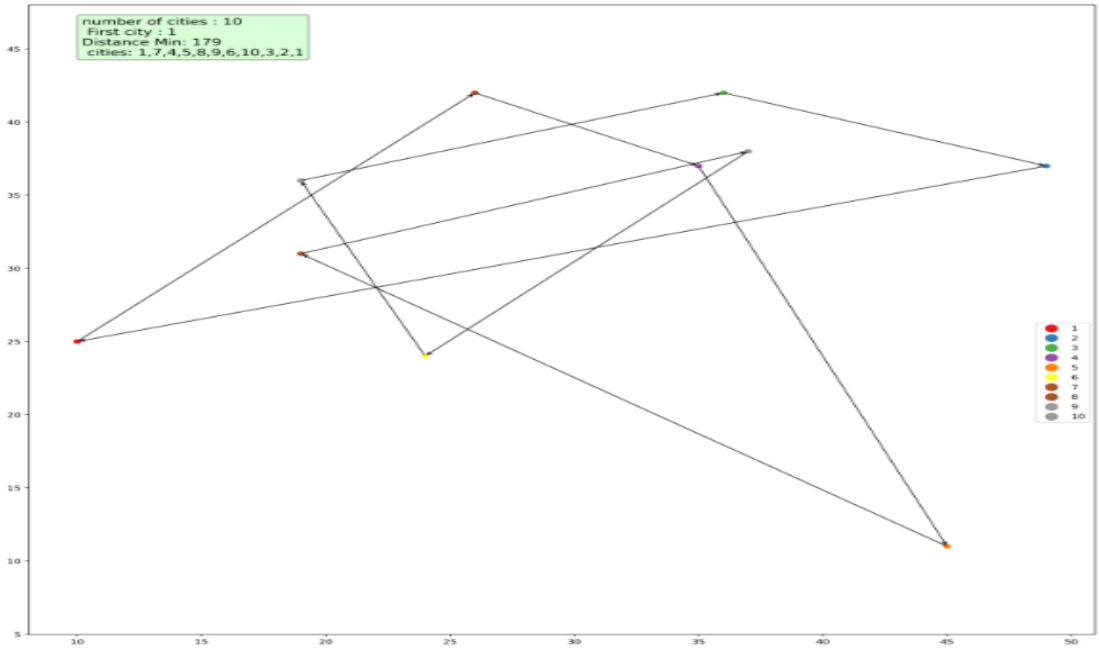


FIGURE 5.2.2 – Solution graphique

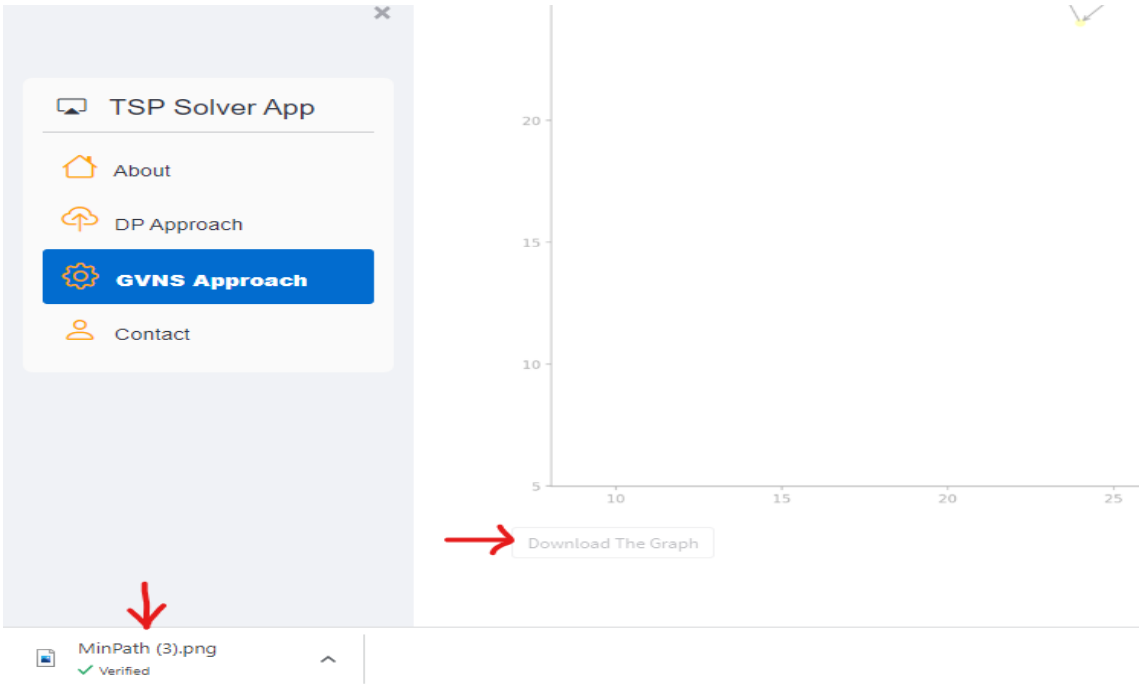


FIGURE 5.2.3 – Télécharger dans un fichier externe

Conclusion

L'objectif de ce rapport est de résumer le travail qui a été réalisé dans le cadre de notre formation à l'Institut National de la Statistique et de l'Economie Appliquée. Il s'est concentré sur la création d'une application web qui permette à l'utilisateur de trouver le chemin le plus court et la distance minimale à travers une ville donnée afin qu'il puisse choisir entre une approche de programmation dynamique ou une approche métaheuristique.
Parmi toutes les méthodes précédentes, il n'y en a pas une qui est meilleure que l'autre au sens large du mot. Tout dépend de l'instance du problème. Si on n'est pas pressées et il s'agit d'un ordonnancement de grands projets, alors il est conseillé de choisir un algorithme exact (dynamique), mais elle est encore limitée en termes 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 la métaheuristique GVNS qui n'est pas limitée que la programmation dynamique, mais nous n'avons aucune garantie qu'elle donne la solution exacte (valeur optimale).
Il est toujours possible de faire des hybrides efficaces en mélangeant les méthodes intelligemment, comme l'amélioration de l'algorithme Backtrack qui a diminué la complexité énormément en pratique, même si ce n'est pas le cas en théorie.

Bibliographie

[1] http ://www.iro.umontreal.ca/ dift6751/VNS.pdf

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

[3] https ://www.cari-info.org/actes2006/144.pdf

[4] http ://www.iro.umontreal.ca/ dift6751/VNS.pdf

[5] https ://www.etudier.com/dissertations/Rapport-Tsp/72601642.html

[6] https ://en.wikipedia.org/wiki/Held

[7] https ://www.researchgate.net/publication/336923519$_{P}robleme_{d}u_{v}oyageur_{d}e_{c}ommerce_{-T}SP$