ROYAUME DU MAROC
--*-*-*
HAUT COMMISSARIAT AU PLAN
--*-*-*-*-*-*

INSTITUT NATIONAL
DE STATISTIQUE ET D'ECONOMIE APPLIQUEE

Le Mini-Projet de module : Techniques d'Optimisation

*****

La résolution du problème du voyageur de commerce (TSP) avec la programmation dynamique et une méta-heuristique (GVNS)

Préparé par : *M. Abdessalam BAHAFID*
M. Amine BAHADDOU

Sous la direction de : *Mr. Rachid BENMANSOUR*

Master de recherche en Systèmes d'Information et Systèmes Intelligents
(M2SI – 1A – S1 – P1)

Le 17 janvier 2021

Table de matière

Liste des figures

Remerciement

Avant d'entamer la rédaction de ce rapport, nous profitons de l'occasion pour remercier infiniment notre chère professeur Mr. Rachid BENMANSOUR pour cette opportunité précieuse de passer à pratiquer ce qu'on avait déjà appris dans son module « Techniques d'Optimisation 1 », ainsi pour sa générosité en matière de formation et d'encadrement.

En effet, sa pédagogie et sa patience ont rendu le cours passionnant. Il a toujours su rester à notre écoute et son soutien permanent nous a été réellement précieux.

Finalement, on remercie tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

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 TSP pour *Traveling Salesman Problem*), c'est un problème 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 ont été proposées qui approchent en temps raisonnable la solution optimale.

Ce problème est sans doute un des plus vieux et problèmes d'optimisation combinatoire énoncé la première fois par W.R Hamilton en 1859. Un des problèmes qui ont été largement étudié par la communauté de la recherche opérationnelle. Plusieurs variantes de ce problème sont traitées dans la littérature (Ilavarasi et Joseph, 2014). Une des variantes les plus répandues de ce problème est le voyageur de commerce avec fenêtres de temps (TSPTW pour *Traveling Salesman Problem with Time Windows*).

I. Présentation du problème

1. Enoncé de problème du voyageur de commerce

Etant donné *n* points (des « villes ») et les distances séparant chaque point, trouver un chemin de longueur totale minimale qui passe exactement une et une seule fois par chaque point et revienne au point de départ. Formellement, une instance est un graph complet $G = (V, A, \omega)$ avec V un ensemble de sommets, A un ensemble d'arrêtes et ω une fonction de coût sur les arcs. Le problème est de trouver le plus court cycle hamiltonien dans le graphe G, où le cycle hamiltionien est un cycle dans un graphe orienté ou non orienté qui passe par tous les sommets une fois et une seule.

2. Représentation formel

On peut décrire formellement le problème TSP comme suit :

TRAVELING SALESMAN PROBLEM
Entrée : Un ensemble de villes $V = \{v_1, \cdots, v_n\}$, une distance $d(i,j)$ pour toute paire $i, j \in \{1, \cdots, n\}$, $i \neq j$
Sortie : Une permutation ϕ de $\{1, \cdots, n\}$
But : Minimiser $\left(\sum_{i=1}^{n-1} d(\phi(i), \phi(i+1))\right) + d(\phi(n), \phi(1))$

Figure 1: La représentation formel de problème TSP

3. TSP symétrique vs. TSP asymétrique

Il existe deux types de TSP, le TSP symétrique et le TSP asymétrique. Dans le premier, le coût pour relier une ville A vers une ville B est le même pour relier B vers A, alors que dans le second, il est permis d'avoir des coûts différents, la distance AB peut différer de la distance BA, ce qui peut par exemple se produire sur une carte routière contenant des voies à sens unique ou des voies plus encombrées dans un sens que dans l'autre.

En théorie des graphes, cela se traduit par :

$$\textbf{TSP symétrique : } \forall s, s' \in S,\ p(s, s') = p(s', s)$$

$$\textbf{TSP asymétrique : } \exists (s, s'), \text{ tel que : } p(s, s') \neq p(s', s)$$

Un TSP asymétrique peut être réduit vers un TSP symétrique en doublant à peine la taille du problème sans que cela ne change le TSP à résoudre. Dans ce qui suit nous ne considérerons donc que le cas du TSP symétrique.

4. Domaines d'application

Les applications du problème du voyageur de commerce sont nombreuses : D'une part, certains problèmes d'optimisation de parcours en robotique ou en conception de circuits électroniques ainsi que certains problèmes de séquencement (passage de trains sur une voie, atterrissage d'avions, processus de fabrication en industrie chimique, etc.) s'expriment directement sous forme de TSP. D'autre part, et c'est sans doute la famille d'application la plus importante sont généralement plus complexes que le TSP, mais comportent des sous-problèmes de type TSP. L'étude du problème du voyageur de commerce est donc un préalable à la résolution de ces problèmes de transport.

II. Les méthodes proposés pour la résolution des instances

Les algorithmes de résolution du TSP peuvent être répartis en deux classes :

1. La méthode de programmation dynamique

La programmation dynamique est une méthode algorithmique qui permet de 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, mais cette approche algorithmique est borné par une complexité de $\mathbf{n^2 2^n}$. Ce fut une avancée majeure dans nombreuses domaines mais qui reste loin de satisfaire les contraintes et les limites pratiques.

La programmation dynamique est l'implémentation améliorée de la version récursive d'un algorithme. Au lieu de faire des appels récursifs, on utilise la mémorisation qui économise ainsi des calculs (et du temps) au détriment de l'espace mémoire. On utilise une table où les valeurs sont ainsi calculées « dynamiquement » en fonction des précédentes.

Il existe deux types :

- De haut en bas (Top-Down) : Commencez à résoudre le problème en le décomposant. Si vous voyez que le problème a déjà été résolu, renvoyez la réponse enregistrée. Si cela n'a pas été résolu, résolvez-le et enregistrez la réponse. Ceci est généralement facile à penser et très intuitif. Ceci est appelé Mémorisation.
- De bas en haut (Bottom-Up): Analysez le problème et voyez l'ordre dans lequel les sous-problèmes sont résolus et commencez à résoudre du sous-problème trivial jusqu'au problème donné. Dans ce processus, il est garanti que les sous-problèmes sont résolus avant de résoudre le problème. Ceci est appelé programmation dynamique.

2. La méthode de méta-heuristique

Une méta-heuristique est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficile (souvent issus des domaines de la recherche opérationnelle, de l'ingénierie ou de l'intelligence artificielle) pour lesquels on ne connaît pas de méthode classique plus efficace.

Les méta-heuristiques sont généralement des algorithmes stochastiques itératifs, qui progressent vers un optimum global, c'est-à-dire l'extremum global d'une fonction, par échantillonnage d'une fonction objectif. Elles se comportent comme des algorithmes de recherche, tentant d'apprendre les caractéristiques d'un problème afin d'en trouver une approximation de la meilleure solution (d'une manière proche des algorithmes d'approximation).

Il existe un grand nombre de méta-heuristiques différentes, allant de la simple recherche locale à des algorithmes complexes de recherche globale. Ces méthodes utilisent cependant un haut niveau d'abstraction, leur permettant d'être adaptées à une large gamme de problèmes différents.

III. Implémentation des algorithmes à utiliser

1. L'algorithme de programmation dynamique

Ci-après l'algorithme de résolution de problème de voyageur de commerce utilisant l'approche exact en programmation dynamique.

- La fonction « **tsp_dynamic_programming** » prend deux arguments :
  - **Costs** : est un tableau de deux dimension, contient les couts de transport entres les différentes villes.
  - **First_city**: est un entier qui représente la ville d'arrivée (*ie* la ville de départ, car on a un cycle hamiltonien).

  Et elle retourne deux paramètres :
  - **Best_tour** : est un tableau d'un dimension, qui représente le cycle de voyage optimale.
  - **Optimal_cost** : est un entier qui représente le cout optimal de cycle hamiltonien.

- La fonction « **memoize** » prend un argument :
  - **F** : est une fonction laquelle on veut enregistrer les résultats de ses appellations pour optimiser la redondance d'appellation de la fonction pour les mêmes paramètres.

  Et elle retourne un dictionnaire qui contient les binômes (arguments, résultats).

- La fonction « **rec_tsp_solving** » est un fonction récursive qui prend deux arguments :
  - **Taget_city** : est un entier qui représente la ville cible d'itération en cours.
  - **Inter_cities** : est un tableau d'un dimension, qui représente les villes à visités.

  Et elle retourne deux paramètres :
  - Le minimum des couts entre l'ensemble des villes intermédiaire et la ville en cours.
  - La prochaine ville è visitée.

```python
def tsp_dynamic_programming(first_city: int):
  def memoize(f):
      memo_dict = {}
      def memo_func(*args):
          if args not in memo_dict:
              memo_dict[args] = f(*args)
          return memo_dict[args]
      memo_func.clear = lambda: memo_dict.clear()
      return memo_func
  ff=first_city
  @memoize
  def rec_tsp_solving(target_city, inter_cities):
      assert target_city not in inter_cities,"ERROR: target city can not be in intermediate cities"
      if inter_cities:
          return min((costs[lc][target_city] + rec_tsp_solving(lc, inter_cities - set([lc]))[0], lc)
                     for lc in inter_cities)
      else:
        return (costs[ff][target_city], target_city)

  best_tour, costs_length = [], len(costs)
  init_city = first_city
  inter_cities = frozenset([i for i in range(costs_length) if i!=first_city])
  optimal_cost=-1 #unknown
  while True:
      l_cost, before_last_city = rec_tsp_solving(first_city, inter_cities)
      if optimal_cost==-1: optimal_cost=l_cost
      if before_last_city == first_city: break
      best_tour.insert(0, before_last_city)
      first_city = before_last_city
      inter_cities -= frozenset([before_last_city])

  return [init_city]+best_tour+[init_city],optimal_cost
```

Figure 2: algorithme de programmation dynamique (utilisant Python)

2. L'algorithme de méta-heuristique GVNS

2.1. Algorithme de base

- *Descente à voisinage variable (VND)*

La descente à voisinage variable (Variable Neighborhood Descent (VND) en anglais) consiste donc à explorer l'espace de recherche en utilisant plusieurs structures de voisinages (au minimum deux). Soit V = (V_1, …, V_{Kmax}) l'ensemble des structures de voisinages à utiliser. Le changement de voisinage se fait de façon systématique. Pour clarifier la présentation, nous introduisons la procédure de changement de voisinage . Les différentes étapes sont d'écrites dans l'algorithme Celui-ci prend en entrée la solution courante x, une solution x' voisine de x, et une valeur entière notée k, qui correspond à l'indice du voisinage courant (1 <= k <= k_{max}). Les sorties de d'algorithmes sont la solution x (modifiée si x' est meilleure), et la nouvelle valeur de k.

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

Figure 3: Algorithme de changement de voisinage

```
def neighborhood(x, e = 1):
  N = []
  bound = int(len(x) - e)
  for i in range(bound):
    nx = x.copy()
    nx[i],nx[i + e] = nx[i + e],nx[i]
    N.append(nx)
  return N
```

Figure 4: Algorithme de sélection des voisins d'un sommet donnée

```
def shake(x, k):
  N = neighborhood(x, e = k)
  init_x = random.choice(N)
  return init_x
```

Figure 5: Algorithme responsable de choix aléatoire de point initial

```
def FirstImprovement(bx, k):
  while True:
    N = neighborhood(bx, k)
    x = bx
    for i in N:
      if f(i)<f(x):
        x = i
        break
    if f(x)>=f(bx): break
    else: bx = x
  return bx
```

Figure 6: Algorithme d'amélioration de la structure de voisinage

```
#Variable Neighbourhood Descent Algorithm
def VND(bx, N):
  k = 0
  while k<N:
      x = FirstImprovement(bx, k)
      bx, k = change_neighborhood(x, bx, k)
  return bx
```

Figure 7: Algorithme VND

- *Algorithme GVNS*

La recherche de voisinage variable générale (General Variable Neighborhood Search (GVNS)) propose une généralisation de l'algorithme VNS, elle est une méta-heuristique très connue et largement utilisée pour résoudre efficacement de nombreux problèmes d'optimisation combinatoire.

```
def GVNS(bx, execution_time, max_k=3, max_l=2, first_city=0):
  start_time = time.time()
  while time.time()-start_time < execution_time*60:
      k = 1
      while k <= max_k:
          x_ = VND(shake(bx, k), max_l)
          bx, k = change_neighborhood(bx, x_, k)
          bx=[first_city, *[i for i in bx if i!=first_city], first_city]
  return bx,f(bx)
```

Figure 8: Algorithme GVNS

2.2. L'algorithme 2-opt

L'algorithme 2-opt appartient à la classe des algorithmes k-opt. Les algorithmes k-opt effectuent k quantité d'échanges d'escalade à chaque virage. À chaque étape, l'heuristique choisit les options optimales parmi les valeurs adjacentes. Ainsi, chaque valeur a O (n^k) voisins. Dans le contexte du problème TSP, l'algorithme 2-Opt sélectionne de manière aléatoire deux arêtes et leurs quatre extrémités. Les deux arêtes se croisent alors, de sorte que le cycle hamiltonien est maintenu. Si cela conduit à une amélioration, cet échange est maintenu. Sinon, l'algorithme conserve la solution d'origine. Puisque la méta-heuristique de recherche locale stochastique commence par une solution valide et apporte des améliorations incrémentielles, nous pouvons arrêter l'algorithme et il renverra une solution valide. Les algorithmes avec cette propriété sont appelés algorithmes «à tout moment».

L'algorithme 2-opt est une méthode de recherche locale simple avec un mécanisme de permutation spécial qui fonctionne comme son heuristique. L'idée principale de la méthode à 2 options est de supprimer les croisements de chemins dans chaque quartier des villes. Le 2-opt peut être implémenté facilement et exécuté rapidement. Par exemple, TSP avec 120 villes peut être résolu en moins de 5 secondes sur Intel Core i7 en utilisant cette méthode.

Cette méthode similaire à d'autres algorithmes de recherche heuristique ne garantit pas de trouver l'optimum global. Elle est sensible à son point initial dans l'espace de recherche. Cela signifie que les résultats finaux sont modifiés par différents points initiaux. La méthode à 2-opt peut être facilement piégée dans les optimums locaux car elle n'a pas de mécanisme pour en sortir. Connaissant toutes ces failles, cette méthode fonctionne toujours très bien dans TSP puisque son heuristique est très pertinente, donc efficace, dans ce problème**.**

cette algorithme est utilisé implicitement dans la fonction « ***neighborhood*** ».

2.3. La fonction objectif

Le terme « fonction objectif » ou fonction économique, est utilisé en optimisation mathématique et en recherche opérationnelle pour désigner une fonction qui sert de critère pour déterminer la meilleure solution à un problème d'optimisation. Concrètement, elle associe une valeur à une instance d'un problème d'optimisation. Le but du problème d'optimisation est alors de minimiser ou de maximiser cette fonction jusqu'à l'optimum, par différents procédés comme l'algorithme du simplexe.

Dans le cas de problème de voyageur de commerce l'objectif est de ***minimiser*** le chemin traversé.

```python
def f(x: list):
    cost = 0
    for i in range(1,len(x)):
        cost += costs[x[i - 1]][x[i]]
    return cost
```

Figure 9: L'algorithme de la fonction objectif

IV. La résolution des instances

Dans chaque une des trois instances ci-après, on essaie de trouver la solution optimal utilisant l'approche de programmation dynamique (si le temps d'exécution est raisonnable), et approche méta-heuristique (GVNS), et par le suite on donnant une modélisation utilisant des coordonnées aléatoire, le chemin optimal, et le cout optimal.

On notant que :

- ✓ les noms des villes sont de 0 an n-1, ou bien il sont donnée par l'utilisateur.
- ✓ Le site utilisé pour l'exécution des instances est https://colab.research.google.com

1. Le résolution de l'instance 1

La première instance est de taille 17.

```
0,633,257,91,412,150,80,134,259,505,353,324,70,211,268,246,121
633,0,390,661,227,488,572,530,555,289,282,638,567,466,420,745,518
257,390,0,228,169,112,196,154,372,262,110,437,191,74,53,472,142
91,661,228,0,383,120,77,105,175,476,324,240,27,182,239,237,84
412,227,169,383,0,267,351,309,338,196,61,421,346,243,199,528,297
150,488,112,120,267,0,63,34,264,360,208,329,83,105,123,364,35
80,572,196,77,351,63,0,29,232,444,292,297,47,150,207,332,29
134,530,154,105,309,34,29,0,249,402,250,314,68,108,165,349,36
259,555,372,175,338,264,232,249,0,495,352,95,189,326,383,202,236
505,289,262,476,196,360,444,402,495,0,154,578,439,336,240,685,390
353,282,110,324,61,208,292,250,352,154,0,435,287,184,140,542,238
324,638,437,240,421,329,297,314,95,578,435,0,254,391,448,157,301
70,567,191,27,346,83,47,68,189,439,287,254,0,145,202,289,55
211,466,74,182,243,105,150,108,326,336,184,391,145,0,57,426,96
268,420,53,239,199,123,207,165,383,240,140,448,202,57,0,483,153
246,745,472,237,528,364,332,349,202,685,542,157,289,426,483,0,336
121,518,142,84,297,35,29,36,236,390,238,301,55,96,153,336,0
```

Figure 10: la première instance

1.1. Utilisant la programmation dynamique

```
Make your choise: 1
First city to visit: 0

  Optimal path is: [0, 15, 11, 8, 4, 1, 9, 10, 2, 14, 13, 16, 5, 7, 6, 12, 3, 0]
  Optimal distance is: 2085
  Execution time is: 8.876453161239624
```

Figure 11:la solution de 1ère instance utilisant DP

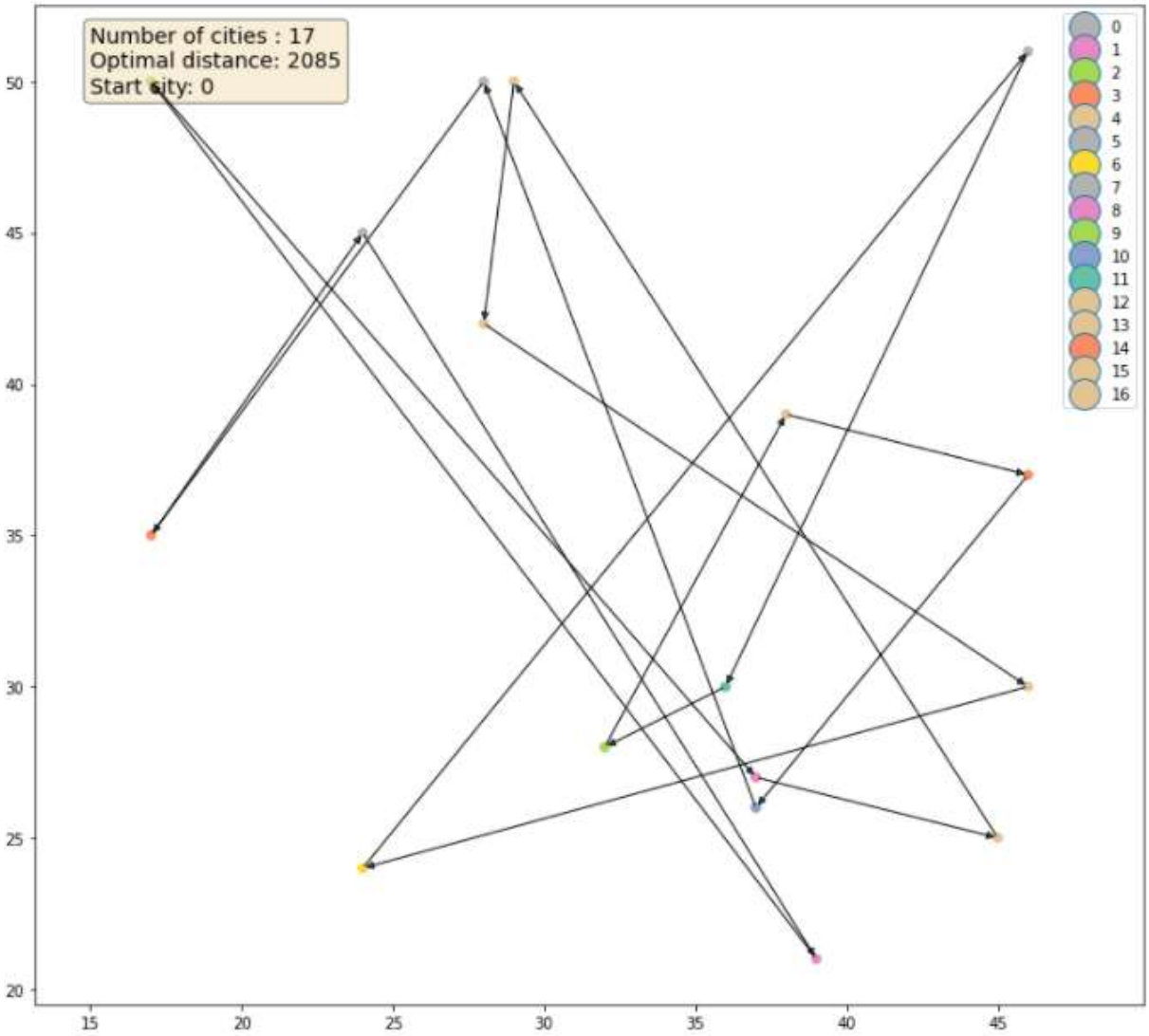

Figure 12:le plot (instance1 , DP) données utilisant des coordonnées aléatoire

1.2. Utilisant GVNS

Il donne un solution approché en fonction de temps d'exécution.

```
Make your choise: 2
First city to visit: 0
Execution time (in minutes): 1

  Optimal path is: [0, 13, 14, 9, 1, 4, 10, 2, 5, 7, 16, 6, 12, 3, 8, 11, 15, 0]
  Optimal distance is: 2153
  Execution time is: 60.010435819625854
```

Figure 13:la solution de 1ère instance utilisant GVNS

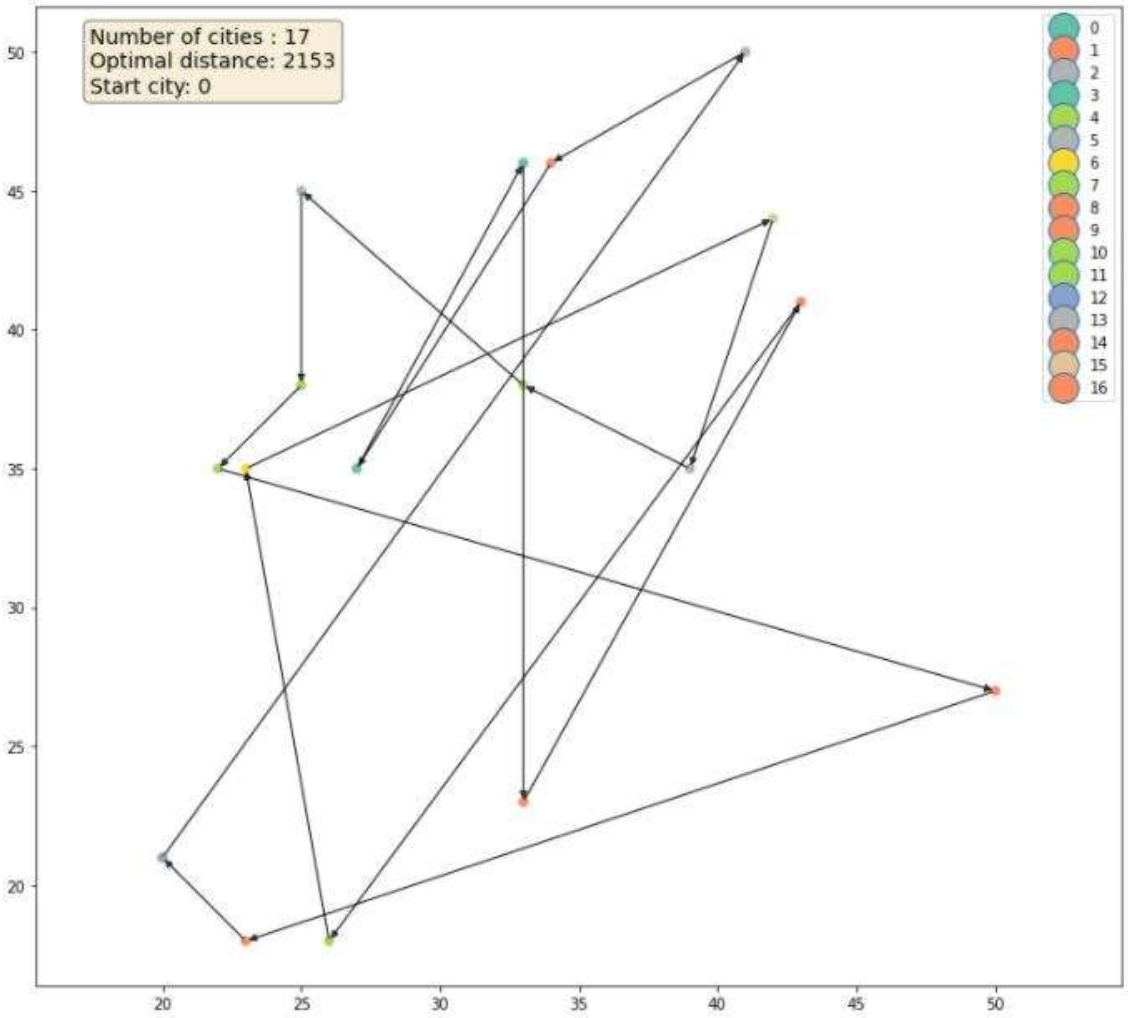

Figure 14: : le plot (instance1 , GVNS) données utilisant des coordonnées aléatoire

2. Le résolution de l'instance 2

La deuxième instance est de taille 48.

2.1. Utilisant la programmation dynamique

Le temps d'exécution très énorme, il demande beaucoup de RAM.

2.2. Utilisant GVNS

Il donne un solution approché en fonction de temps d'exécution.

```
Make your choise: 2
First city to visit: 1
Execution time (in minutes): 1

  Optimal path is: [1, 40, 22, 37, 7, 24, 12, 17, 6, 5, 18, 16, 42, 45, 10, 11, 32, 46, 38, 41, 25, 9, 23, 44, 34, 4, 31, 20, 13, 47, 28, 33, 2, 39, 14, 19, 27, 43, 30, 8, 0, 15, 29, 26, 36, 35, 21, 3, 1]
  Optimal distance is: 68613
  Execution time is: 60.0066282749176
```

Figure 15:la solution de 2ème instance utilisant GVNS

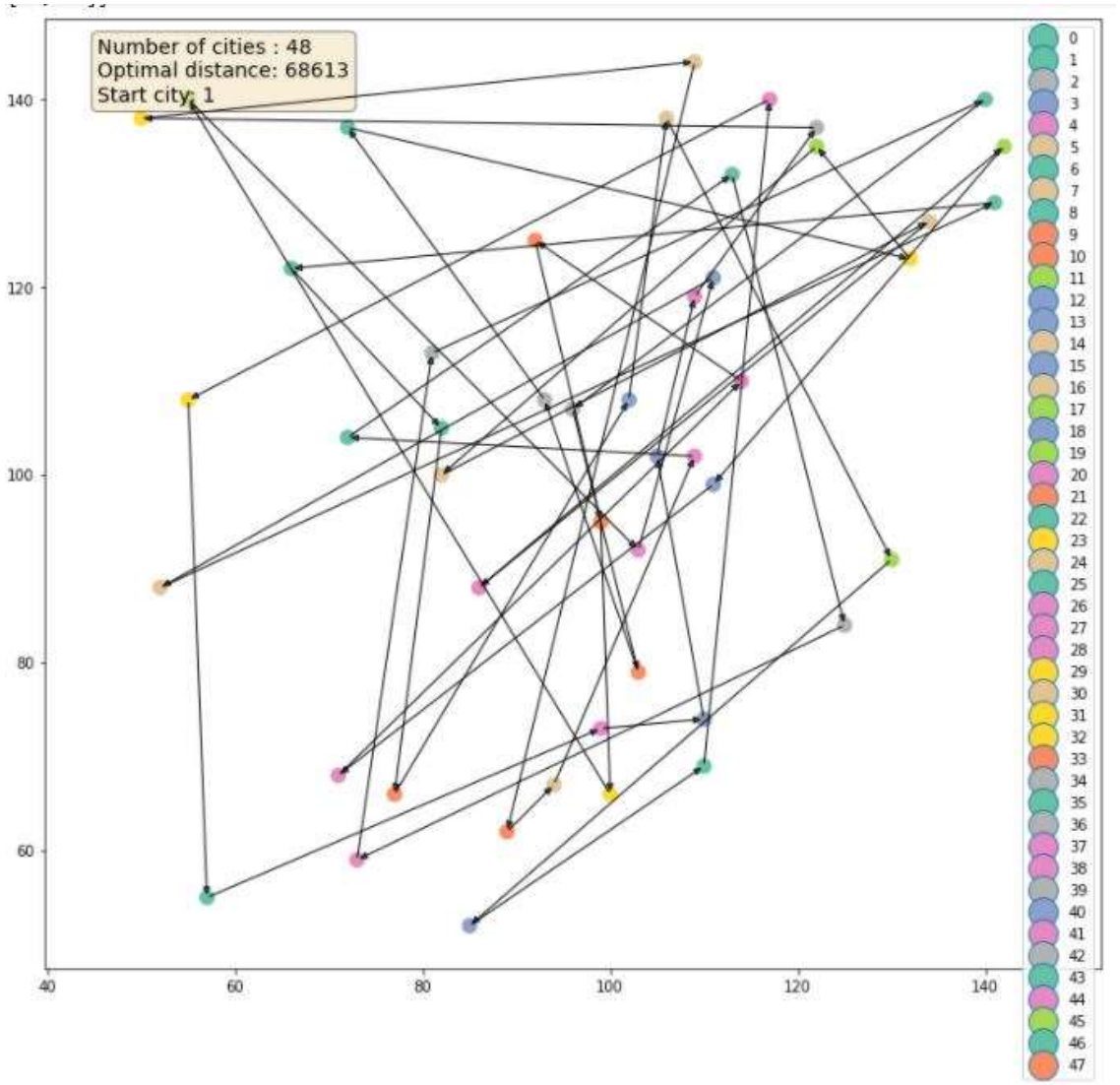

Figure 16:le plot (instance2 , GVNS) données utilisant des coordonnées aléatoire

3. Le résolution de l'instance 3

La troisième instance est de taille 27.

3.1. Utilisant la programmation dynamique

Le temps d'exécution est in peut près 4 heures, ms les ressources de RAM demander c'est plus que 12 Gb.

3.2. Utilisant GVNS

Il donne un solution approché en fonction de temps d'exécution.

```
Make your choise: 2
First city to visit: 3
Execution time (in minutes): 1

  Optimal path is: [3, 5, 11, 17, 19, 9, 6, 2, 12, 10, 20, 26, 18, 24, 13, 4, 14, 25, 7, 0, 21, 15, 22, 23, 16, 8, 1, 3]
  Optimal distance is: 8002
  Execution time is: 60.07615065574646
```

Figure 17:la solution de 3ème instance utilisant GVNS

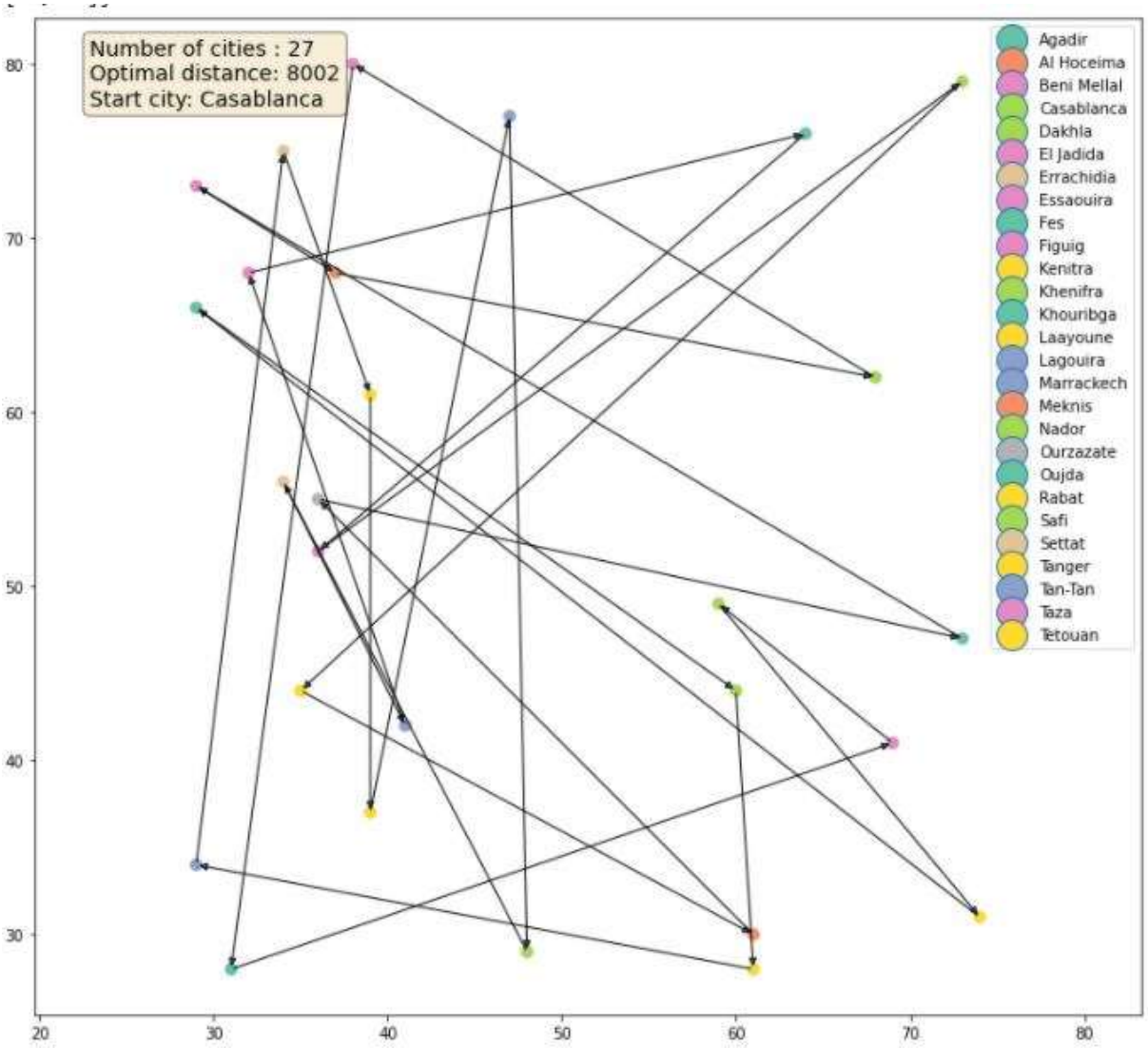

Figure 18:le plot (instance3 , GVNS) données utilisant des coordonnées aléatoire

Conclusion

D'après les résultats obtenu au cours de l'exécution des instances, on constate que la programmation dynamique donne des résultats exactes, mais ne reste pas opérationnelle pour des instances d'un taille grand, car il consomme beaucoup d'espace mémoire pour faire la mémorisation des résultats des étapes précédentes, ainsi qu'il a besoin d'un temps énorme pour s'exécuter. Tante que la méthode GVNS peuvent être adaptées à tout type de problème d'optimisation avec des tailles différentes, mais il donne des résultats approché.

En général nous ne pouvons pas trancher laquelle des deux est la plus performante. Nous pensons que cela dépend des champs d'application et de la structure initial des instances.

Références

https://interstices.info/le-probleme-du-voyageur-de-commerce/
https://www.cari-info.org/actes2006/144.pdf
http://polymorphe.free.fr/cours/ia/tsp/these_chap_4(TSP).pdf
https://fr.wikipedia.org/wiki/M%C3%A9taheuristique#:~:text=Une%20m%C3%A9taheuristique%20est%20un%20algorithme,de%20m%C3%A9thode%20classique%20plus%20efficace.
https://dept-info.labri.fr/~gavoille/UE-TAP/cours.pdf