



## Problème de voyageur de commerce avec contraintes de fenêtres de temps souples type 1

**ALVES COSTA OLIVEIRA** Nícolas  
**VILELA MOURA** Alvaro Augusto

Automne 2024

## Table des Matières

<b>1. Brève description du problème</b>	<b>3</b>
<b>2. Détails de chaque étape</b>	<b>5</b>
2.1 Modèle linéaire	5
2.1.1 Paramètres	5
2.1.2 Variables de décision	5
2.1.3 Fonction objectif	5
2.1.4 Contraintes	5
2.2 Modélisation avec python	6
2.3 Construction du distancier	7
2.4 Solutions initiales	8
2.4.1 Plus Proche Voisin	8
2.4.2 Clarke & Wright savings	10
2.4.3 Meilleure Insertion	12
2.5 Recherche locale par 2-Opt	14
2.6 Algorithme génétique	17
<b>3. Analyse de complexité</b>	<b>20</b>
3.1 Heuristiques constructives	20
3.2 Recherche locale	21
<b>4. Conclusion</b>	<b>22</b>
<b>5. Bibliographie</b>	<b>22</b>

## Liste des Algorithmes

Algorithme 01: Pseudo-code pour l'heuristique du Plus Proche Voisin	9
Algorithme 02: Pseudo-code pour l'heuristique de Clarke & Wright	11
Algorithme 03: Pseudo-code pour l'heuristique de Meilleure Insertion	13
Algorithme 04: Pseudo-code pour l'heuristique de recherche local par 2-Opt	15
Algorithme 05: Pseudo-code pour l'heuristique de l'Algorithme Génétique	17

## Liste des Figures

Figure 01: Représentation graphique du problème proposé	3
Figure 02: La fenêtre temporelle	4
Figure 03: Les temps d'attente et temps de retard	4
Figure 04: Routes pour les instances basés sur l'algorithme du Plus Proche Voisin	10
Figure 05: Routes pour les instances basés sur l'algorithme de Clarke & Wright	12
Figure 06: Routes pour les instances basés sur l'algorithme de Meilleure Insertion	14
Figure 07: Routes pour les instances basés sur l'algorithme de 2-Opt	16
Figure 08: Image représentative de l'algorithme génétique utilisé dans le projet	18
Figure 09: Variation du coût par rapport au nombre d'itérations (générations)	20

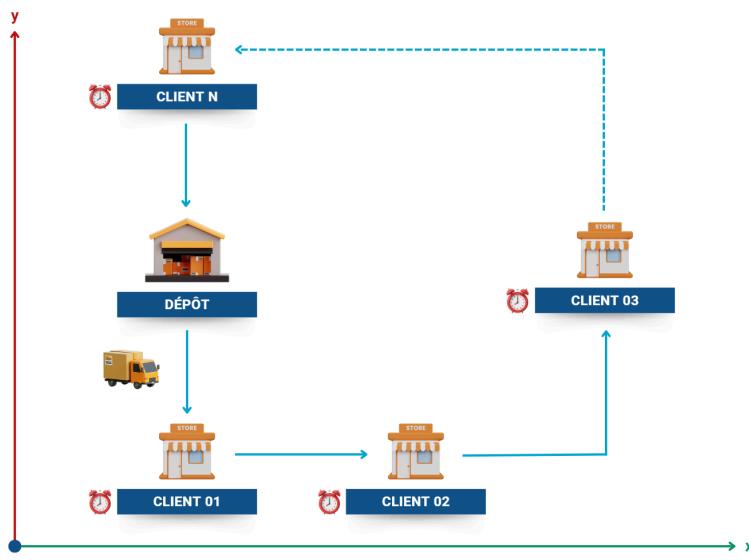
## Liste des Tableaux

Tableau 01: Résultats des instances pour le modèle d'optimisation linéaire	7
Tableau 02: Distancier construit avec une instance de 40 sommets	8
Tableau 03: Résultats des instances pour l'algorithme du Plus Proche Voisin	9
Tableau 04: Résultats des instances pour l'algorithme de Clarke & Wright	12
Tableau 05: Résultats des instances pour l'algorithme de Meilleure Insertion	14
Tableau 06: Résultats des instances pour l'algorithme de 2-Opt	16
Tableau 07: Pourcentage d'utilisation des heuristiques et de l'aleatoire pour générer la population initiale	19
Tableau 08: Paramétrage de l'Algorithme Génétique	19
Tableau 09: Résultats des instances pour l'Algorithme Génétique	19

# 1. Brève description du problème

Le **Traveling Salesman Problem** (TSP), étudié depuis le 19<sup>e</sup> siècle, est l'un des problèmes les plus emblématiques de la **recherche opérationnelle**. Formulé initialement par William Rowan Hamilton en 1859 comme un jeu, il consiste à trouver le parcours le plus court permettant à un voyageur de visiter un ensemble fini de villes exactement une fois, avant de retourner à son point de départ (Pop et al., 2024).

Figure 01: Représentation graphique du problème proposé



Ce problème d'optimisation combinatoire est classé parmi les **NP-complets**, ce qui signifie qu'il ne peut être résolu de manière exacte dans un temps polynomial (Karp, 1972). En pratique, cela implique que le **temps nécessaire** pour trouver une solution optimale augmente de façon **exponentielle** en fonction de la taille des données d'entrée et sont donc inexploitables en pratique même pour des instances de taille modérée.

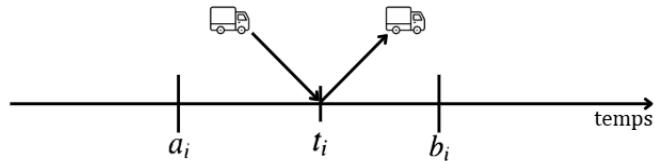
Le TSP a de nombreuses **applications** dans la **vie réelle** (Matai, Singh & Mitall, 2010) dans **divers domaines** tels que la planification, la logistique, les problèmes de planification de machines, la fabrication de microprocesseurs et le séquençage de l'ADN. Par exemple, dans la logistique, le TSP est utilisé pour optimiser les trajets de livraison afin de réduire les coûts de transport. En planification de machines, il permet d'organiser les tâches de manière efficace pour minimiser le temps de production. En raison de ses nombreuses applications pratiques, le TSP demeure un **problème central** dans la recherche opérationnelle et l'optimisation (Puerto & Valverde, 2024).

Bien que le concept soit simple, le TSP présente une grande diversité d'applications et la résolution de cas de grande envergure reste un **défi informatique** majeur. Les **méthodes** utilisées pour résoudre le TSP se répartissent en deux grandes catégories: les algorithmes exacts et les stratégies heuristiques. Les **approches exactes**, comme les plans de coupe, la méthode de séparation et évaluation (branch-and-bound) ou encore la programmation dynamique, visent à trouver des **solutions optimales**, mais elles exigent souvent des ressources de calcul importantes. En revanche, les **méthodes heuristiques** permettent

d'obtenir des solutions proches de l'optimal en un **temps réduit**, bien qu'elles ne garantissent pas toujours l'optimalité (Doganbas & Shin, 2024).

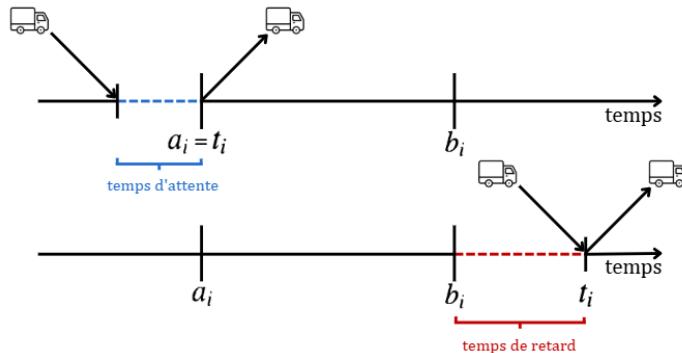
Dans la littérature, il existe différentes variantes de problèmes qui utilisent le TSP comme base pour formuler des problèmes encore plus complexes. Le **TSP avec fenêtres temporelles (TSP-TW)** est une variante du problème classique du voyageur de commerce. Dans cette version, chaque noeud doit être visité à l'intérieur d'une fenêtre temporelle  $(a_i, b_i)$  prédéfinie, qui spécifie un intervalle de temps durant lequel la visite est possible.

Figure 02: La fenêtre temporelle



Dans le présent travail, nous étudions une version **relaxée** du TSP avec fenêtres temporelles. Cela signifie qu'il est permis d'arriver en dehors des limites de l'intervalle. En cas d'arrivée anticipée, le véhicule doit attendre jusqu'à l'ouverture de la fenêtre. En revanche, si l'arrivée est tardive, une **pénalité** est appliquée pour le retard. L'objectif est de déterminer l'itinéraire optimal minimisant les coûts liés au trajet et aux pénalités de retard.

Figure 03: Les temps d'attente et temps de retard



Les fichiers fournis contiennent des **données variables** dans les feuilles de calcul, les trois premières ayant **40 nœuds** et les trois dernières ayant **100 nœuds**. Les documents contiennent des colonnes représentant les informations suivantes: le numéro du sommet, les coordonnées X et Y, le début et la fin de la fenêtre de temps pour chaque client, le coût de pénalité lié à un retard d'une unité de temps (identique pour tous les clients, sauf pour le dépôt, qui n'a pas de pénalité), ainsi que la demande de chaque client et la capacité du véhicule. Ce format de données a été utilisé pour modéliser les **contraintes temporelles et logistiques** du problème, avec des variations dans les coûts de pénalité selon les différentes versions.

## 2. Détails de chaque étape

### 2.1 Modèle linéaire

À partir du problème proposé, nous avons fait une **modélisation linéaire**. Dans cette section, nous essayons d'expliquer le modèle et de présenter dans les sous-sections suivantes les paramètres, les variables de décision, la fonction objective et les contraintes.

#### 2.1.1 Paramètres

$n$	Nombre de sommets.
$\alpha_i$	Coefficient de pénalité lié au retard.
$a_i$	Limite inférieure de la fenêtre de temps.
$b_i$	Limite supérieure de la fenêtre de temps.
$dist_{ij}$	Distance totale parcourue entre les points $i$ et $j$ .

#### 2.1.2 Variables de décision

$x_{ij}$	1, si le véhicule passe des sommets $i$ à $j$ dans le processus de livraison; 0 sinon.
$t_i$	Temps de départ au client $i$ .
$r_i$	Retard associé au client $i$ .

#### 2.1.3 Fonction objectif

La fonction objective du projet vise à **minimiser le coût total**, qui se compose de deux éléments: **a)** le coût lié à la distance parcourue par le véhicule; **b)** le coût associé aux pénalités, si celles-ci sont appliquées.

$$\text{Min} \sum_{i=1}^n (x_{ij} \cdot dist_{ij}) + \sum_{i=1}^n (\alpha_i \cdot r_i) \quad (0)$$

#### 2.1.4 Contraintes

Dans le processus de visite, la **contrainte 1** indique qu'il faut visiter tous les clients une et une seule fois.

$$\sum_{i=1}^n x_{ij} = 1 \quad \forall j \in N \quad (1)$$

La **contrainte 2** montre que si un véhicule visite un client, il doit le quitter.

$$\sum_{i=1}^n x_{ij} - \sum_{i=1}^n x_{ji} = 0 \quad \forall j \in N \quad (2)$$

La **contrainte 3** impose que, si un véhicule dessert le client  $j$  immédiatement après le client  $i$ , le temps de départ du client  $j$  doit être supérieur au temps de départ du client  $i$  plus la durée nécessaire pour parcourir l'arc  $(i, j)$ . Cette contrainte est essentielle pour éliminer les sous-tours et garantir une tournée cohérente. De plus,  $j$  doit être strictement supérieur à 0.

$$t_j \geq t_i + d_{ij} - M \cdot (1 - x_{ij}) \quad \forall i \in N \quad \forall j \in N' \quad \forall i \neq j \quad (3)$$

Pour calculer le temps de retard pour chaque client  $i$ , nous pouvons utiliser la formule  $\max(0, t_i - b_i)$ . Afin de maintenir le modèle linéaire, nous utilisons les **contraintes 4** et **5** pour effectuer ce calcul :

$$r_i \geq t_i - b_i \quad \forall i \in N \quad (4)$$

$$r_i \geq 0 \quad \forall i \in N \quad (5)$$

Dans notre problème, il est possible d'arriver chez un client avant le début de sa fenêtre de temps. Dans ce cas, il est nécessaire d'attendre jusqu'à l'heure de début avant de partir pour le client suivant. La **contrainte 6** empêche le départ d'un client avant le moment où il est disponible pour être servi.

$$t_i \geq a_i \quad \forall i \in N \quad (6)$$

Enfin, nous avons présenté les dernières **contraintes 7** et **8** qui indiquent, respectivement, que  $x$  est une variable booléenne et que  $t$  est une variable non négative.

$$x_{ij} = \{0, 1\} \quad \forall i \in N, \forall j \in N \quad (7)$$

$$t_i \geq 0 \quad \forall i \in N \quad (8)$$

## 2.2 Modélisation avec python

Le code implémente le **modèle d'optimisation linéaire** pour résoudre la variante du problème de tournée de véhicules où des contraintes de temps et des pénalités sont ajoutées. Le modèle, formulé sur **Google Collabs**, vise à minimiser la fonction coût qui combine les distances parcourues entre les points et les pénalités associées aux retards.

Enfin, le modèle est optimisé avec une **limite de temps** spécifiée. Une fois la solution trouvée, le code extrait la tournée optimale en lisant les variables  $x[i, j]$  et construit une **liste des successeurs** pour chaque point. Il affiche également le **coût total** de la tournée optimale, les relations entre les points (sous forme d'arcs suivis), ainsi que le temps de résolution.

Afin de tester la fonctionnalité du code construit, qui a été fourni en annexe au présent rapport, nous avons effectué un **test** avec le **fichier d'exemple**. Dans ce sens, nous avons inséré une limite de temps de **1800 secondes** et obtenu un coût de **1005,20 unités** comme solution optimale pour le problème.

Ce même processus a été reproduit pour les **autres instances**, comme le montre le tableau ci-dessous. De manière générale, le coût optimal trouvé pour l'instance 40 (1) s'est révélé supérieur à celui des instances 40 (2) et 40 (3) pour un temps de traitement de 1800 secondes.

Tableau 01: Résultats des instances pour le modèle d'optimisation linéaire

Instance	Méthode Exacte		
	Distance	Coût	CPU (s)
Inst. Test	447,92	1005,20	1800
Inst. 40 (1)	529,53	8396,70	1800
Inst. 40 (2)	397,90	1788,92	1800
Inst. 40 (3)	381,32	1109,20	1800

On souligne également que nous avons essayé de trouver des résultats pour les instances 100 (1), 100 (2) et 100 (3), mais, même avec un **temps de traitement élevé** (nous avons testé jusqu'à 12.600 secondes), notre code n'a pas réussi à trouver un coût optimal. Cela s'explique par le fait que ces instances sont **plus complexes** et nécessitent un temps de traitement supérieur à celui des instances de 40 sommets.

## 2.3 Construction du distancier

Le script que nous avons utilisé construit un **tableau de distance** entre les sommets à partir de leurs coordonnées géographiques. Les coordonnées `cox[i]` et `coy[i]` pour chaque point *i* sont extraites d'un fichier Excel, puis la **distance euclidienne** entre chaque paire de points *i* et *j* est calculée avec la formule:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad (10)$$

Ces distances sont ensuite stockées dans une **matrice carrée** appelée `dist`, où l'élément `[i][j]` représente la distance entre les points *i* et *j*. Le script initialise également une variable `dmax`, qui **accumule la somme totale** de toutes les distances dans la matrice. Cela est fait en additionnant chaque distance calculée lors de l'étape précédente. Ainsi, `dmax` peut être utilisé pour représenter une **mesure globale** des distances dans l'ensemble des points.

La matrice ci-dessous montre les valeurs obtenues pour l'**instance** d'exemple, avec **40 sommets** et **distances** comprises dans une intervalle de **[1, 50]** pour les **axes x et y**. La diagonale principale est définie sur zéro car il n'est pas possible d'obtenir une distance entre le même sommet.

Tableau 02: Distancier construit avec une instance de 40 sommets

	0	1	2	3	4	5	6	7	8	9	...	30	31	32	33
0	0.000000	5.099020	12.000000	6.403124	25.632011	22.847319	30.016662	34.058773	14.866069	13.038405	...	21.931712	20.000000	15.811388	21.260292
1	5.099020	0.000000	13.928388	10.816654	23.345235	27.202941	35.057096	37.443290	19.416488	12.165525	...	26.925824	19.646883	19.697716	25.806976
2	12.000000	13.928388	0.000000	8.062258	37.107951	29.698485	31.953091	42.801869	22.022716	7.071068	...	20.223748	32.000000	24.698178	16.124515
3	6.403124	10.816654	8.062258	0.000000	31.780497	22.022716	26.305893	34.828150	14.142136	12.529964	...	16.492423	25.317978	16.643317	15.000000
4	25.632011	23.345235	37.107951	31.780497	0.000000	33.541020	46.324939	35.057096	29.832868	35.057096	...	43.931765	9.848858	26.627054	45.486262
5	22.847319	27.202941	29.698485	22.022716	33.541020	0.000000	13.453624	13.928388	8.062258	34.409301	...	18.027756	23.706539	8.000000	23.537205
6	30.016662	35.057096	31.953091	26.305893	46.324939	13.453624	0.000000	23.345235	17.088007	38.327536	...	12.806248	36.619667	19.723083	19.104973
7	34.058773	37.443290	42.801869	34.828150	35.057096	13.928388	23.345235	0.000000	20.808652	46.669048	...	31.575307	26.076810	18.384776	37.363083
8	14.866069	19.416488	22.022716	14.142136	29.832868	8.062258	17.088007	20.808652	0.000000	26.400758	...	15.231546	20.518285	4.123106	19.104973
9	13.038405	12.165525	7.071068	12.529964	35.057096	34.409301	38.327536	46.669048	26.400758	0.000000	...	27.073973	31.780497	28.284271	23.194827
10	30.886890	34.928498	38.183766	30.479501	37.107951	8.485281	16.278821	7.071068	16.401219	42.801869	...	25.000000	27.459060	15.231546	31.016125

Il est important de noter que le code a également été appliqué aux **autres instances** disponibles dans le fichier relatif au projet, donc plusieurs distanciers ont été créés.

## 2.4 Solutions initiales

Au départ, nous avons utilisé plusieurs **approches heuristiques** pour résoudre le problème. Une méthode heuristique est une technique d'optimisation qui permet d'obtenir une **solution réalisable en un temps relativement court**, souvent basée sur une intuition ou une stratégie pragmatique. Cependant, bien que ces méthodes soient efficaces pour trouver rapidement des solutions, elles **ne garantissent pas** nécessairement d'obtenir une **solution optimale**.

### 2.4.1 Plus Proche Voisin

Dans cette première **méthode constructive** que nous avons faite, nous cherchons, à partir d'un sommet donné, choisir le sommet le plus proche, non visité. Comme cette heuristique considère seulement la valeur de la distance entre les sommets et ne tient pas compte des fenêtres temporelles de chaque point, on se rend compte que la solution faisable trouvée est beaucoup plus grande que celle trouvée par des méthodes exactes.

Pour expliquer de **manière claire** et **structurée** les étapes du programme que nous avons développé, tout en utilisant un langage accessible et sans se restreindre aux contraintes syntaxiques propres à un langage de programmation, nous avons conçu un **pseudo-code** pour l'heuristique du Plus Proche Voisin.

Algorithme 01: Pseudo-code pour l'heuristique du Plus Proche Voisin

---

**Algorithme 1:** Plus Proche Voisin

---

**Input:** *racine*: sommet de départ.

**Output:** la tournée construite, débutant et se terminant à la racine, avec un coût total minimal selon l'heuristique du PPV.

---

01. **1. Initialisation**
  02. *distance*  $\leftarrow 0$ ,
  03. *tour*  $\leftarrow$  vecteur de taille  $n + 1$
  04. *visited*  $\leftarrow$  vecteur booléen de taille  $n$ , initialisé à false
  05. *tour[0]*  $\leftarrow$  *racine*, *visited[racine]*  $\leftarrow$  true
  06. **2. Procedure**
  07. **for**  $i = 0$  to  $n$ :
  08.   *tour[i]*  $\leftarrow$  *plus\_proche(racine, visited)*
  09.   *distance*  $+= dist[racine][tour[i]]$
  10.   *racine*  $\leftarrow$  *tour[i]*
  11.   *visited[racine]*  $\leftarrow$  true
  12. **endfor**
  13. *tour[n+1]*  $\leftarrow$  *tour[0]*
  14. *distance*  $= dist[racine][tour[0]]$
  15. **3. Retourner** *distance*, *tour*
- 

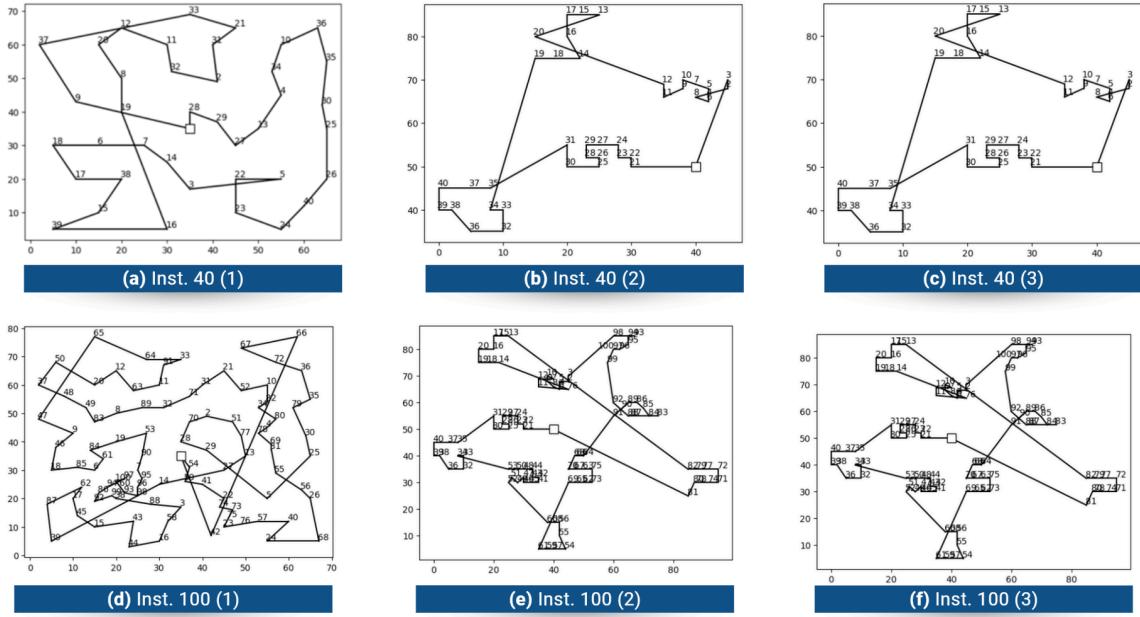
Le **pseudo-code** ci-dessus commence par **initialiser** la distance totale à zéro et crée un vecteur pour stocker la tournée et un autre pour suivre les sommets visités. Ensuite, dans une **boucle** qui **itère** sur **chaque sommet**, l'algorithme appelle la fonction **plus\_proche** pour trouver le sommet non visité le plus proche du sommet courant (racine), met à jour la distance totale en ajoutant la distance entre le sommet courant et le voisin choisi, puis marque ce sommet comme visité et le définit comme le nouveau sommet courant pour la prochaine itération. À la **fin de la boucle**, l'algorithme retourne à la racine pour compléter la tournée et met à jour la distance finale avant de renvoyer la distance totale ainsi que le vecteur de la tournée construite.

Tableau 03: Résultats des instances pour l'algorithme du Plus Proche Voisin

Instance	Plus Proche Voisin		
	Distance	Coût	CPU (s)
Inst. Test	306,43	5681,14	0,0008
Inst. 40 (1)	485,24	8807,43	0,0011
Inst. 40 (2)	225,85	4413,26	0,0008
Inst. 40 (3)	225,85	4620,87	0,0013
Inst. 100 (1)	806,40	41331,78	0,0040
Inst. 100 (2)	593,95	14521,28	0,0071
Inst. 100 (3)	593,95	19329,14	0,0041

Le tableau 03 montre les **coûts** pour chacune des instances disponibles. Nous observons que les coûts des instances avec 40 sommets sont inférieurs à ceux des instances avec 100 sommets. De plus, le **temps de traitement** pour ces instances est également plus court, bien que, dans tous les cas, les six temps soient négligeables.

Figure 04: Routes pour les instances basés sur l'algorithme du Plus Proche Voisin



On constate que les **graphiques (b)** et **(c)** présentent le même itinéraire, tout comme les **graphiques (e)** et **(f)**. Cela s'explique par le fait que les seules modifications apportées à ces instances concernent les fenêtres temporelles et les pénalités, ce qui n'a aucun impact sur la localisation des sommets. Dans ce cas, comme l'algorithme du Plus Proche Voisin prend uniquement en compte les distances, on observe que les itinéraires restent inchangés.

#### 2.4.2 Clarke & Wright savings

La méthode heuristique de Clarke & Wright peut être décrite en deux étapes principales: l'initialisation et l'itération. Lors de l'**étape d'initialisation**, une tournée est créée pour chaque client, partant du dépôt et y revenant. Ensuite, les économies  $s_{ij}$  sont calculées à l'aide de la formule  $s_{ij} = c_{0i} + c_{0j} - c_{ij}$ , où  $c_{ij}$  représente les coûts entre les points i et j. Ces économies sont ensuite triées par **ordre décroissant** pour prioriser les combinaisons les plus avantageuses.

Lors de l'**étape d'itération**, pour chaque paire de tournées, deux économies possibles sont calculées en fonction des **connexions aux extrémités** des tournées. La décision de **fusionner** les tournées est prise en tenant compte des économies  $s_{ij}$ . Il est important de noter que les  $s_{ij}$  peuvent être calculés une seule fois lors de l'initialisation et mis à jour au fur et à mesure que les extrémités des tournées changent. Il est également essentiel d'**enregistrer les tournées** résultantes pour éviter les conflits ou les doublons dans le processus.

### Algorithme 02: Pseudo-code pour l'heuristique de Clarke & Wright

---

**Algorithme 2: Clarke & Wright**

---

**Input:** *racine*: sommet de départ.

**Output:** la tournée construite, débutant et se terminant à la racine, avec un coût total minimal selon l'heuristique de Clarke & Wright.

---

01. **1. Initialisation**
02. *tours*  $\leftarrow \{i: [i]\}$  pour chaque sommet *i* si *i*  $\neq$  *racine*
03. *total\_cost*  $\leftarrow$  somme(*dist[racine][i]*  $\cdot$  2) pour chaque *i* si *i*  $\neq$  *racine*
04. **2. Procedure**
05. *économie* = []
06. **for** *i* to *n*:
07.     **for** *j* to (*i* + 1, *n*):
08.         *if* *i* et *j* sont différents de *racine*:
09.             *économie* = *dist[racine][i]* + *dist[racine][j]* - *dist[i][j]*
10.             Ajouter (*économie[i][j]*) à *économie*
11.         **endif**
12.     **endfor**
13. **endfor**
14. Trier *économie* en ordre décroissant
15. **for** *économie[i][j]* to *économie*:
16.     *if* *i* est visité ou *j* est visité: continuer
17.     Identifier *tour i* et *tour j* contenant *i* et *j*
18.     *if* *tour i* est différent de *tour j*:
19.         Combiner *tours[tour i]* et *tours[tour j]*
20.         Mettre à jour *visited* et réduire *total\_cost* par *économie*
21.     **endif**
22. **endfor**
23. Construire la tournée finale:
24.     *final\_tour*  $\leftarrow$  [*racine*]
25.     Ajouter toutes les routes restantes dans *tours* à *final\_tour*
26.     Ajouter *racine* pour fermer la tournée
27. Calculer le coût final
28. **4. Retourner** *total\_cost, final\_tour*

---

Le **pseudo-code initialise** après créer une **ensemble de tournées** individuelles pour chaque sommet, sauf la racine, et en **calculant le coût total** basé sur les distances entre la racine et chaque sommet, multiplié par deux pour tenir compte du retour. Ensuite, l'algorithme calcule les **économies potentielles** d'associer chaque paire de sommets non-racine en utilisant la formule qui prend en compte la distance directe entre ces sommets par rapport aux distances individuelles à partir de la racine.

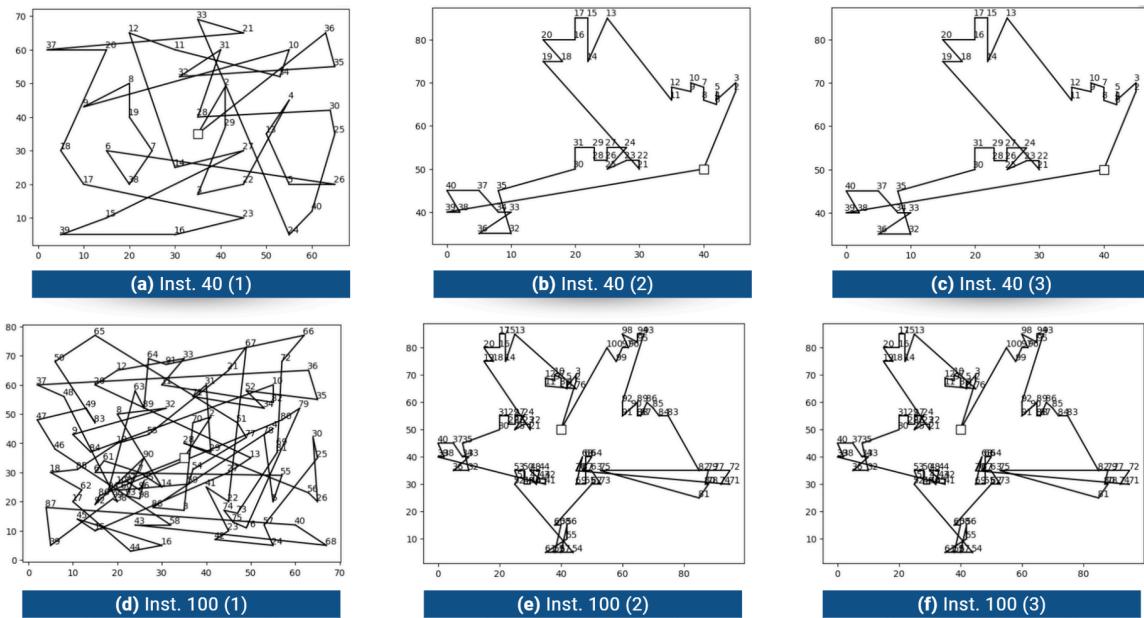
Une fois les économies calculées, elles sont triées par **ordre décroissant**, et l'algorithme **combine les tours** des sommets qui offrent le plus grand gain économique, tout en maintenant un suivi des sommets visités et en mettant à jour le coût total en conséquence. Enfin, l'algorithme construit la **tournée finale** en ajoutant tous les sommets restants, en fermant la tournée avec la racine et en calculant le coût final avant de retourner le coût total et la tournée construite.

Tableau 04: Résultats des instances pour l'algorithme de Clarke & Wright

Instance	Clarke & Wright		
	Distance	Coût	CPU (s)
Inst. Test	1043,31	9191,91	0,0029
Inst. 40 (1)	1252,56	18109,62	0,0026
Inst. 40 (2)	1113,39	4669,88	0,0014
Inst. 40 (3)	1113,39	3959,41	0,0014
Inst. 100 (1)	2808,74	90560,32	0,0105
Inst. 100 (2)	3106,76	16917,97	0,0092
Inst. 100 (3)	3106,76	17636,54	0,0125

Le tableau ci-dessus présente les distances, les coûts et les temps de traitement pour chacune des instances. On constate que les résultats obtenus sont **supérieurs** à ceux obtenus avec l'algorithme du Plus Proche Voisin, sauf pour **Inst. 40 (3)** et **Inst. 100 (3)**, qui ont présenté des coûts inférieurs.

Figure 05: Routes pour les instances basés sur l'algorithme de Clarke & Wright



La figure ci-dessus montre les itinéraires générés à partir de l'application de l'algorithme de Clarke & Wright. Tout comme dans l'algorithme présenté précédemment, nous pouvons observer une similarité entre les itinéraires générés pour les **graphiques (b)** et **(c)** ainsi que pour les **graphiques (e)** et **(f)**.

#### 2.4.3 Meilleure Insertion

L'algorithme de la meilleure insertion commence par initialiser une tournée en sélectionnant un **sommet racine i** (choisi comme le sommet le plus proche ou le plus éloigné du dépôt, par exemple) et en formant une **tournée initiale** du type **0 – i – 0**.

Ensuite, à chaque étape, un sommet j, non encore visité, est sélectionné comme **le plus proche** de n'importe quel sommet déjà dans la tournée. Ce sommet j est ensuite inséré dans la tournée existante à la position qui **minimise le coût total** de la tournée, en utilisant la

formule  $c_{i,j} + c_{j,k} - c_{i,k}$ , où i et k représentent les sommets adjacents dans la tournée. Ce processus est répété jusqu'à ce que tous les sommets soient insérés dans la tournée. Enfin, étant donné une route partielle et un ensemble des sommets non visités, cette méthode trouve le **couple sommet** – meilleur point d'insertion.

#### Algorithme 03: Pseudo-code pour l'heuristique de Meilleure Insertion

---

##### **Algorithme 3:** Meilleure Insertion

---

**Input:** nous avons utilisé seulement les paramètres globales.

**Output:** la tournée construite, débutant et se terminant à la racine, avec un coût total minimal selon l'heuristique de Meilleure Insertion.

---

01. **1. Initialisation**
  02.  $\text{tour} \leftarrow [0,0]$  (commence et termine au sommet 0)
  03.  $\text{visited} \leftarrow$  vecteur de taille  $n + 1$ , initialisé à false
  04.  $\text{visited}[0]$  et  $\text{visited}[n] \leftarrow$  true
  05. Trouver *racine*, le sommet le plus proche du dépôt (0) et l'ajouter à *tour*[1]
  06. Marquer *racine* comme visité
  07. **2. Procédure**
  08. Insertion des sommets restants
  09. **for** chaque sommet non visité:
  10. Trouver un sommet i non visité de façon aléatoire
  11. Utiliser  $\delta \leftarrow d_{\max} + n \cdot b[0]$
  12. **for** chaque position j possible dans la tournée actuelle:
  13. Insérer i à la position j dans une tournée temporaire *pseudo\_tour*
  14. Calculer le coût de *pseudo-tour*
  15. Si le coût est inférieur à  $\delta$ , mettre à jour  $\delta$  et *jref*
  16. **endfor**
  17. Insérer i dans *tour* à la position *jref*
  18. Marquer i comme visité
  19. **endfor**
  20. **3. Retourner tour**
- 

L'algorithme commence par **initialiser** la tournée avec le **sommet de départ** (racine), en définissant le vecteur *visited* pour suivre les sommets visités. Il identifie ensuite le sommet **le plus proche** du dépôt (sommel 0) et l'ajoute à la tournée tout en le marquant comme visité. La **procédure principale** consiste à insérer les sommets restants un par un: pour chaque sommet non visité, l'algorithme choisit un sommet aléatoire, puis évalue diverses positions d'insertion dans la tournée actuelle pour **minimiser** le coût total.

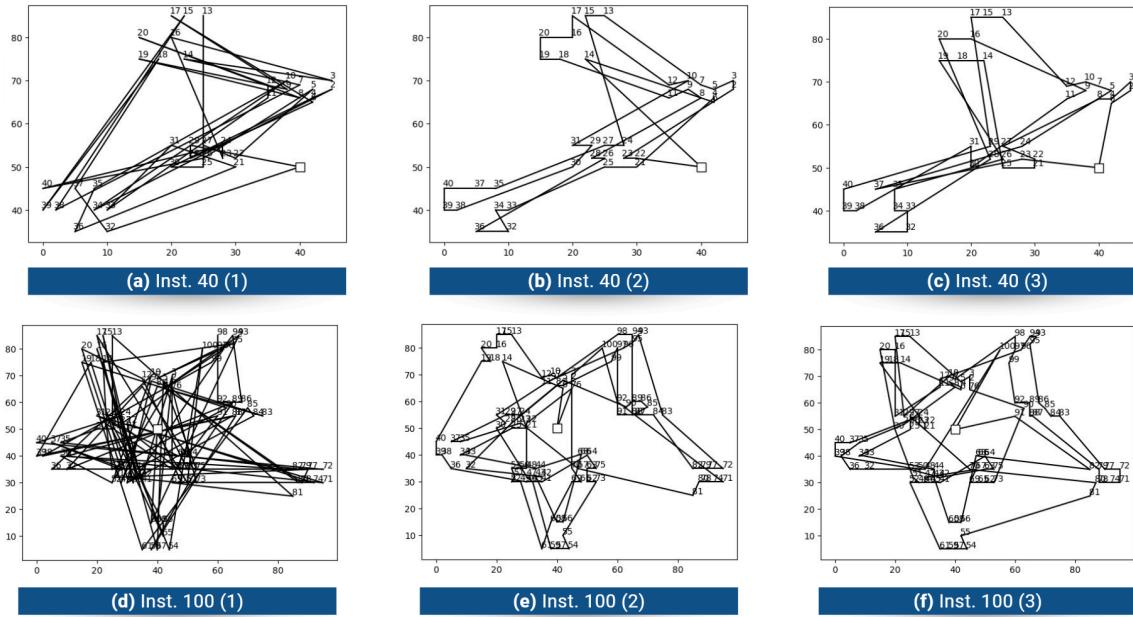
Pour chaque position possible, il insère **temporairement** le sommet dans une version fictive de la tournée (*pseudo\_tour*), calcule son coût, et si ce coût est inférieur à la valeur précédemment maximale ( $\delta$ ), il met à jour la **position optimale** pour l'insertion. Une fois la meilleure position trouvée, le sommet est inséré dans la tournée finale. À la fin de l'algorithme, la **tournée** construite est retournée, comprenant tous les sommets visités dans l'ordre approprié.

Tableau 05: Résultats des instances pour l'algorithme de Meilleure Insertion

Instance	Meilleure Insertion		
	Distance	Coût	CPU (s)
Inst. Test	486,91	558,65	0,0317
Inst. 40 (1)	501,06	6728,88	0,0371
Inst. 40 (2)	418,10	823,15	0,0330
Inst. 40 (3)	461,06	730,36	0,0356
Inst. 100 (1)	873,15	27142,27	0,6041
Inst. 100 (2)	1198,64	4240,67	0,5775
Inst. 100 (3)	1202,41	4822,55	0,5350

Dans notre cas, l'algorithme est initialisé en établissant une connexion avec le nœud le plus proche du dépôt. Il est important de souligner que cet algorithme produira toujours des **résultats différents** à chaque exécution du code. Cela s'explique par les particularités du **processus de choix aléatoire** des nœuds de cette méthode entre 1 et  $n - 1$ . Le tableau 05 présente donc les résultats d'une des exécutions effectuées sur Google Colab, qui indiquent, de manière générale, une **amélioration des coûts** obtenus pour toutes les instances par rapport aux deux algorithmes précédents, c'est-à-dire que tous les **résultats** se sont avérés **supérieurs**.

Figure 06: Routes pour les instances basés sur l'algorithme de Meilleure Insertion



Contrairement aux figures 04 et 05, on observe que tous les graphiques tracés sont différents les uns des autres. Ce fait peut, une fois de plus, être attribué au caractère **partiellement aléatoire** de cette méthode de résolution constructive.

## 2.5 Recherche locale par 2-Opt

L'opération 2-opt, initialement proposée par Croes et al. (1958), est une **méthode de recherche locale** simple et largement répandue. Elle est couramment utilisée en combinaison avec des algorithmes évolutifs pour améliorer la qualité des solutions. À la

différence des méthodes constructives, les approches d'**amélioration itérative** nécessitent une **solution initiale complète et réalisable**. Elles fonctionnent en explorant le voisinage de cette solution, c'est-à-dire des solutions proches obtenues par des modifications simples, appelées mouvements (Engels & Manthey, 2009). À chaque itération, une solution voisine qui améliore le résultat est sélectionnée, ce qui permet de progresser vers une **solution de meilleure qualité**.

La méthode de **recherche locale par 2-Opt** est une technique couramment utilisée pour améliorer une solution dans des problèmes d'optimisation. Elle consiste à modifier une tournée en **supprimant deux arêtes** non consécutives dans la solution actuelle, ce qui casse la structure de la tournée. Ensuite, ces deux segments sont **reconnectés différemment** pour former une nouvelle tournée. Ce mouvement peut potentiellement réduire la longueur totale de la tournée, car il **élimine des croisements** ou des **segments inefficaces**. Ce processus est répété dans le voisinage des solutions jusqu'à ce qu'il ne soit plus possible de réduire la distance totale en appliquant cette méthode (Wang et al., 2022).

**Algorithme 04: Pseudo-code pour l'heuristique de recherche local par 2-Opt**

---

**Algorithme 4:** Recherche Local par 2-Opt

---

**Input:** *tour*: une route trouvée précédemment.

**Output:** la nouvelle tournée construite avec un coût total minimal selon la recherche local par 2-Opt.

---

01. **1. Initialisation**
  02. *meilleur*  $\leftarrow$  true (indique si une meilleure solution a été trouvée)
  03. **2. Procédure**
  04. **while** *meilleur* = true:
  05.     *meilleur*  $\leftarrow$  false
  06.     **for** chaque paire (*i*, *j*) de sommets dans la route:
  07.         vérifier si *j* est valide (*j*  $\neq$  *i*, *j*  $\neq$  *i* + 1, *j*  $\neq$  *i* - 1)
  08.         créer une *pseudo\_tour* en inversant les sommets entre *i* + 1 et *j*
  09.         **if** le coût de *pseudo\_tour* est inférieur au coût de *tour*:
  10.             Mettre à jour  $\leftarrow$  *pseudo\_route*
  11.             *meilleur*  $\leftarrow$  true
  12.         **endif**
  13.     **endfor**
  14. **endwhile**
  15. **3. Retourner** *tour*
- 

Ce pseudo-code **initialise** une **variable meilleure** à true, indiquant qu'une meilleure solution pourrait être trouvée. La procédure principale se déroule dans une boucle qui continue tant qu'une **amélioration** est détectée. Pour chaque paire de sommets (*i*, *j*) dans la tournée, l'algorithme vérifie la validité de *j* (c'est-à-dire qu'il ne doit pas être le même sommet que *i*, ni être directement adjacent à *i*).

Si *j* est valide, une **nouvelle version** de la tournée, appelée ***pseudo\_tour***, est créée en inversant l'ordre des sommets entre *i* + 1 et *j*. Si le coût de cette *pseudo\_tour* est inférieur au coût de la tournée originale, l'algorithme met à jour la tournée actuelle avec cette nouvelle version et marque meilleur comme true, indiquant qu'une amélioration a été réalisée. La boucle continue jusqu'à ce qu'aucune **amélioration supplémentaire** ne soit trouvée. À la fin de la procédure, l'algorithme retourne la tournée optimisée, qui devrait avoir un **coût total minimal** par rapport à la tournée initiale.

Tableau 06: Résultats des instances pour l'algorithme de 2-Opt

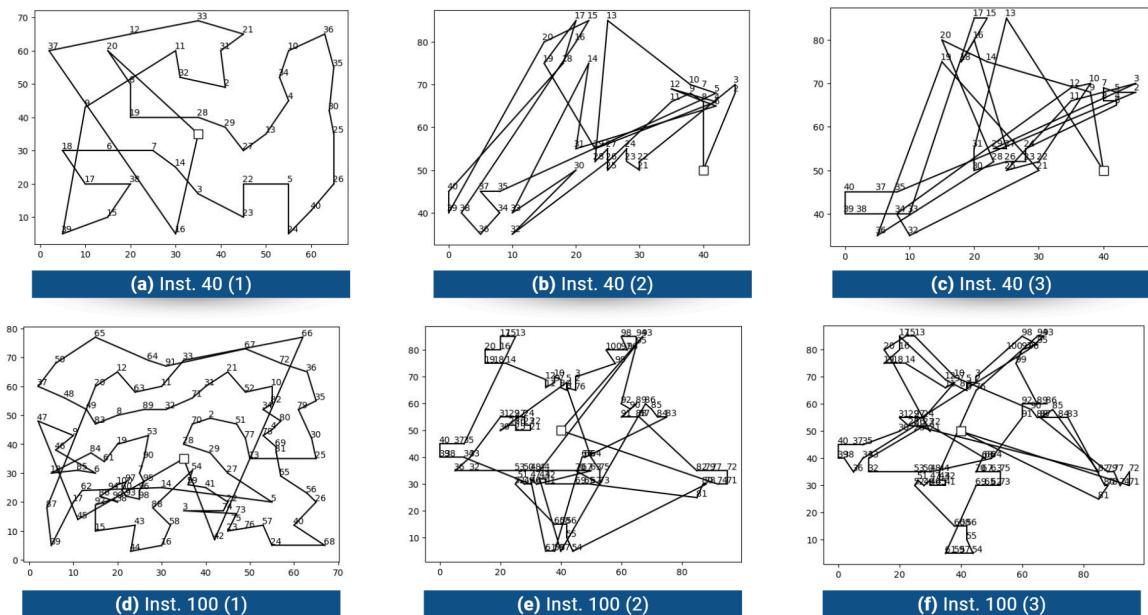
Instance	Recherche locale par 2-Opt			
	Distance	Coût	CPU (s)	% Amélioré
Inst. Test	575,09	3089,15	0,4937	45,62%
Inst. 40 (1)	555,08	7696,87	0,3448	12,61%
Inst. 40 (2)	582,16	2198,21	0,3425	50,19%
Inst. 40 (3)	594,59	2211,07	0,3281	52,15%
Inst. 100 (1)	925,52	33411,93	5,0649	19,16%
Inst. 100 (2)	1006,65	11355,57	6,7421	21,80%
Inst. 100 (3)	987,11	13703,94	5,0108	29,10%

Afin d'évaluer le **potentiel de l'heuristique** 2-Opt, nous avons mené des tests comparatifs en utilisant les résultats obtenus par l'algorithme du **Plus Proche Voisin** comme solution initiale. L'application de la **recherche locale** par 2-Opt a démontré son efficacité en **réduisant** significativement les **coûts** des solutions initiales fournies par le PPV, générant ainsi de nouvelles tournées avec des distances et des coûts optimisés.

Dans le tableau 06, on observe des **améliorations substantielles** des coûts pour toutes les instances testées. Nous observons que tous les coûts ont été réduits d'**environ 33%** (en moyenne), ce qui témoigne d'une amélioration globale significative. Parmi les résultats, les **instances 40 (2) et 40 (3)** se distinguent par les améliorations les plus notables, avec une réduction des coûts **supérieure à 50%** par rapport aux valeurs obtenues avec l'algorithme **PPV**.

Un point important à souligner est que les **temps de traitement** sont considérablement plus élevés par rapport à ceux des algorithmes précédents. L'heuristique 2-Opt s'impose comme un **outil robuste** et **performant**, particulièrement utile dans des contextes où la qualité des solutions est essentielle et où des améliorations significatives sont nécessaires à partir de solutions initiales rapides, comme celles générées par le PPV.

Figure 07: Routes pour les instances basés sur l'algorithme de 2-Opt



Un aspect important qui ressort clairement de la figure ci-dessus est que les graphes générés présentent un **nombre de croisements** significativement **inférieur** par rapport aux graphes illustrés dans la figure 04, lesquels sont, quant à eux, issus de l'algorithme du Plus Proche Voisin.

## 2.6 Algorithme génétique

Les algorithmes génétiques (AG) sont des **méthodes de résolution approchées**, inspirées de l'évolution naturelle, mais sans garantie d'optimalité. Les AG fonctionnent en simulant des processus comme la **variation**, l'**adaptation** et l'**héritérité**. Chaque individu au sein de la population représente une solution pour le problème, et ceux ayant de **meilleures caractéristiques** ont plus de **chances de survivre** et de se **reproduire**, transmettant ainsi leurs traits aux générations suivantes (Rosa Villamayor-Paredes et al., 2023).

Selon Gridin (2021), le fonctionnement des AG repose sur **trois étapes** principales: la **sélection**, le **croisement** et la **mutation**. La sélection imite la sélection naturelle en favorisant les individus les mieux adaptés. Le croisement partage la composition génétique de deux individus, créant de nouvelles solutions. Enfin, la mutation introduit des changements aléatoires pour augmenter la diversité de la population. Ainsi, bien que ces algorithmes soient inspirés des mécanismes évolutifs, ils restent une analogie qui permet de résoudre des **problèmes complexes** (Zhao et al., 2023).

**Algorithme 05: Pseudo-code pour l'heuristique de l'Algorithme Génétique**

---

**Algorithme 5:** Algorithme Génétique

**Input:** #TAILLE\_POP, #GENERATIONS, #TAILLE\_TOURNOI, #PROB\_MUTATION.

**Output:** l'expression avec la meilleure valeur de *fitness*.

01. **1. Initialisation**
  02. Générer la première population avec #POP\_SIZE tournées
  03. Évaluer chaque tournée de la population initiale avec *fitness*
  04. **2. Procédure**
  05. **for** *gen* allant de 2 à #GENERATIONS, faire:
  06.   **for** *p* allant de 1 à #TAILLE\_POP, faire:
  07.     **Sélection:** sélectionne #TAILLE\_TOURNOI tournées aléatoires de la population précédente pour un tournoi. Parmi ceux-ci, choisissez les deux individus avec les meilleures valeurs de *fitness* comme paire de parents
  08.     **Croisement:** combine des parties des deux parents pour créer deux nouvelles tournées enfantées
  09.     **Mutation:** apporte des modifications aléatoires dans les tournées enfantées selon #PROB\_MUTATION.
  10.     Évaluer la *fitness* des nouvelles tournées.
  11.     Ajouter la tournée avec la meilleure *fitness* parmi les parents et les enfants à la population p
  12.   **endfor**
  13. **endfor**
  14. **3. Retourner** l'expression avec la meilleure valeur de *fitness*
- 

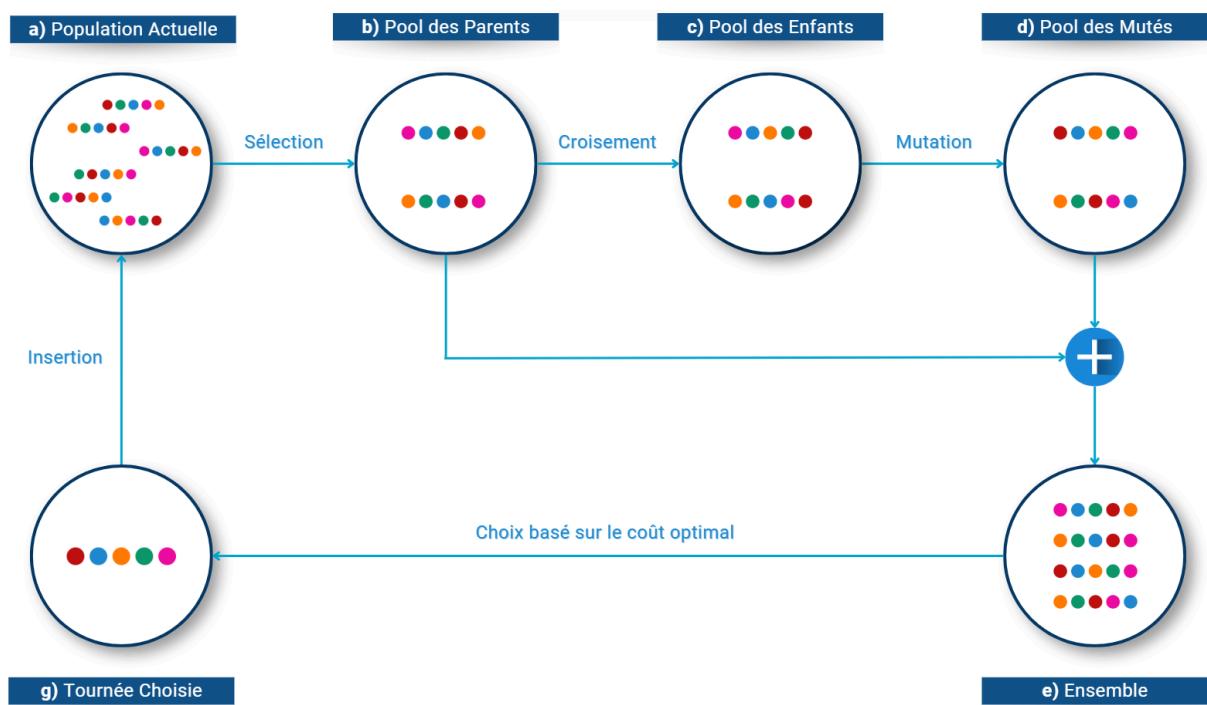
Le dernier algorithme commence par générer une **population initiale** de tournées (individus) de taille spécifiée par **#TAILLE\_POP**, puis évalue chaque tournée en fonction de sa valeur de **fitness**, qui mesure la qualité de la solution. La procédure principale se déroule sur un nombre défini de générations (**#GENERATIONS**), où pour chaque génération, un processus de **sélection** est effectué: un sous-ensemble de tournées est choisi **aléatoirement** pour

former un tournoi, et les **deux** tournées ayant les **meilleures valeurs** de fitness sont sélectionnées comme **parents**.

Ensuite, un **croisement** est effectué pour combiner ces deux parents et générer **deux** nouvelles tournées (**enfants**). Une **mutation** est ensuite appliquée à ces enfants, avec une probabilité déterminée par **#PROB\_MUTATION**, pour introduire des **variations aléatoires**. Les nouvelles tournées sont évaluées en termes de **fitness**, et la **meilleure tournée** parmi les parents et les enfants est ajoutée à la population pour la génération suivante. Ce processus de sélection, croisement et mutation se répète jusqu'à ce que le **nombre défini** de générations soit **atteint**.

À la fin, l'algorithme **retourne la tournée** avec la meilleure valeur de fitness trouvée au cours de toutes les générations, représentant une **solution optimale** ou **proche de l'optimal** au problème traité. L'**image** ci-dessous a été conçue dans le but de faciliter la **compréhension** de l'algorithme génétique que nous avons développé.

Figure 08: Image représentative de l'algorithme génétique utilisé dans le projet



Pour **générer la population initiale** dans notre application de l'Algorithme Génétique, nous avons utilisé une **combinaison de solutions** issues d'une **version aléatoire** de l'heuristique de **Meilleure Insertion**, de solutions générées par l'heuristique du **Plus Proche Voisin** et de solutions entièrement aléatoires. La **proportion** de chaque méthode est présentée dans le tableau 07. Cette stratégie vise à **accélérer la convergence** de l'Algorithme Génétique vers une **solution satisfaisante**.

Ces heuristiques ont été choisies pour construire la population initiale car elles se sont révélées les **plus efficaces** en termes de **qualité de solution** et de **temps d'exécution** lors des **tests précédents**. Les solutions aléatoires constituent la plus grande part de la population initiale afin d'éviter une **convergence prématuée** vers un **optimum local**.

Tableau 07: Pourcentage d'utilisation des heuristiques et de l'aléatoire pour générer la population initiale

Création de la Population Initial		
M. Insertion	PPV	Aléatoire
15,00%	50,00%	80,00%

Une fois la population initiale constituée, nous pouvons appliquer l'Algorithm Génétique. Pour ce faire, il est nécessaire de **définir les paramètres** présentés précédemment dans l'algorithme 05. Les paramètres choisis sont indiqués dans le tableau 08.

Tableau 08: Paramétrage de l'Algorithm Génétique

Paramètres de l'Algorithm Génétique			
Qt. Gen.	Taille Pop.	Taille Tournoi	% Mut.
50	150	5	20,00%

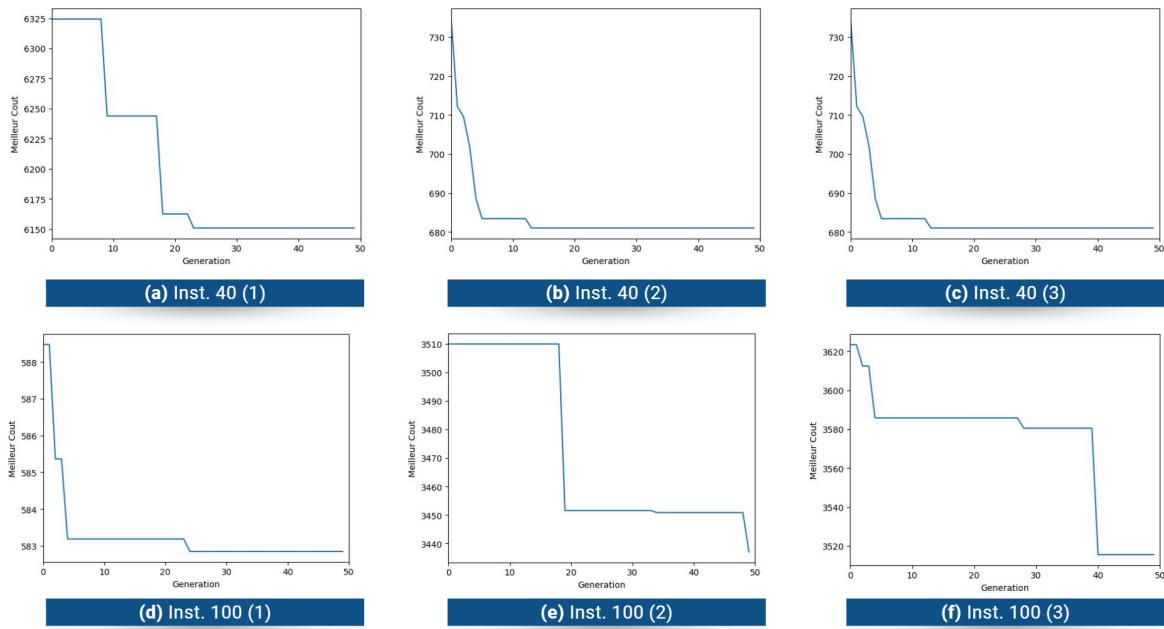
Le tableau 09 présente les **résultats obtenus** pour chacune des sept instances, avec les valeurs du **meilleur résultat** pour le coût de la **génération actuelle** (best of run) et du meilleur résultat **historique** (best so far). De manière générale, on constate que les coûts obtenus après 50 générations se sont révélés **compétitifs** par rapport aux autres **heuristiques** et au **modèle de programmation linéaire** développés précédemment. De plus, les **temps de traitement** pour cette heuristique se sont avérés nettement supérieurs pour toutes les instances, notamment pour celles comportant **100 nœuds**. Ces temps d'exécution, cependant, n'ont même pas atteint une minute, démontrant l'**efficacité** de l'algorithme génétique.

Tableau 09: Résultats des instances pour l'Algorithm Génétique

Instance		Algorithm Génétique						CPU (s)
		Gen: 0	Gen: 10	Gen: 20	Gen: 30	Gen: 40	Gen: 49	
Inst. Test	best of run	544,08	499,01	498,69	497,95	496,47	496,47	4,482
	best so far	544,08	499,01	498,69	497,95	496,47	<b>496,47</b>	
Inst. 40 (1)	best of run	6324,27	6243,92	6162,67	6151,04	6151,04	6151,04	5,063
	best so far	6324,27	6243,92	6162,67	6151,04	6151,04	<b>6151,04</b>	
Inst. 40 (2)	best of run	734,61	683,44	681,04	681,04	681,04	681,04	4,786
	best so far	734,61	683,44	681,04	681,04	681,04	<b>681,04</b>	
Inst. 40 (3)	best of run	588,47	583,19	583,19	582,85	582,85	582,85	6,650
	best so far	588,47	583,19	583,19	582,85	582,85	<b>582,85</b>	
Inst. 100 (1)	best of run	25809,97	25807,35	25704,18	25699,43	25699,43	25699,43	34,235
	best so far	25809,97	25807,35	25704,18	25699,43	25699,43	<b>25699,43</b>	
Inst. 100 (2)	best of run	3510,00	3510,00	3451,56	3451,56	3450,83	3437,02	30,397
	best so far	3510,00	3510,00	3451,56	3451,56	3450,83	<b>3437,02</b>	
Inst. 100 (3)	best of run	3623,33	3585,72	3585,72	3580,45	3515,48	3515,48	31,248
	best so far	3623,33	3585,72	3585,72	3580,45	3515,48	<b>3515,48</b>	

La figure 09 illustre l'**évolution de la solution** de l'algorithme en fonction des générations. Nous constatons que, pour toutes les instances, il y a une **diminution du coût total** au fil des **générations**, ce qui démontre l'**efficacité** de cet algorithme pour résoudre le problème du TSP avec fenêtres temporelles et potentiellement d'autres variations du TSP.

Figure 09: Variation du coût par rapport au nombre d'itérations (générations)



Enfin, il convient de noter que les populations initiales et finales de chaque application, ainsi que leurs coûts, ont été sauvegardées dans des fichiers Excel qui ont été envoyés avec ce rapport.

**Un point d'attention :** dans les fichiers Excel, les nœuds sont indexés à partir de 0 et non de 1

### 3. Analyse de complexité

Une analyse de complexité est une méthode permettant d'**évaluer les performances** d'un algorithme en termes des **ressources** qu'il consomme, principalement:

- **Complexité temporelle** (temps d'exécution): la quantité de temps nécessaire à l'exécution de l'algorithme en fonction de la taille de l'entrée.
- **Complexité spatiale** (mémoire utilisée): la quantité de mémoire requise par l'algorithme pour traiter les données.

Ces complexités sont exprimées à l'aide de la **notation Big-O** ( $O$ ), qui fournit une estimation du comportement asymptotique de l'algorithme lorsque la taille des données d'entrée augmente considérablement ( $n \rightarrow \infty$ ). Cette notation permet de comparer l'efficacité des algorithmes et d'identifier ceux qui seront les **plus adaptés** aux grandes quantités de données ou aux contraintes spécifiques d'un système.

#### 3.1 Heuristiques constructives

De manière générale, sur la base des **codes** que nous avons développés, les heuristiques constructives du **Plus Proche Voisin** et **Clarke & Wright** ont une complexité de  $O(n^2)$ . Cela signifie que le temps d'exécution est **quadratique** et qu'il croît proportionnellement au **carré de n**.

Par **exemple**, dans l'algorithme du PPV (Asani et al. 2024), nous avons  $n$  points dans l'espace, et pour chaque point donné, l'algorithme calcule la distance avec les autres points jusqu'à  $n - 1$  afin de trouver celui qui est le plus proche. Dans ce contexte, pour trouver le noeud le plus proche d'un point spécifique, il est nécessaire de le comparer avec tous les autres points de l'ensemble, ce qui nécessite  $n - 1$  comparaisons. Si ce processus est répété pour chaque point de l'ensemble (composé de  $n$  points), cela nécessitera:

$$n \times (n - 1) = n^2 - n \approx n^2 \quad (11)$$

Dans une analyse de complexité, nous sommes généralement intéressés par le comportement de l'algorithme pour des problèmes où  $n$  est une **valeur grande**. Dans ce cas, le terme soustrait ( $n$ ) dans l'**équation 11** devient insignifiant par rapport au terme  $n^2$ . C'est cela le motif que nous utilisons  $O(n^2)$  comme une **approximation** pour représenter la **croissance asymptotique** de l'algorithme.

Contrairement aux autres algorithmes constructifs, l'algorithme de **Meilleure Insertion** a une complexité de  $O(n^3)$  dans le **pire des cas**. Cela s'explique par une **boucle externe** qui s'exécute  $n - 2$  fois pour insérer les sommets non visités dans la route, et une **boucle interne** qui parcourt toutes les positions possibles dans la route actuelle. À chaque itération, les opérations principales, comme la copie de la route, l'insertion, et le calcul du coût, ont un coût combiné de  $O(m^2)$ , où  $m$  est la taille de la route. En cumulant ces coûts pour toutes les itérations, on atteint  $O(n^3)$ , ce qui est efficace pour des problèmes de **taille modérée** mais peut devenir coûteux pour de **grandes instances**.

## 3.2 Recherche locale

L'**algorithme 2-Opt** présente une **complexité** qui **dépend de la taille de la route**, représentée par  $n$ , et du **nombre d'itérations** nécessaires pour converger. La boucle externe, qui s'exécute tant qu'une amélioration est trouvée, effectue un nombre indéterminé d'itérations, noté  $k$ . À chaque itération, deux boucles imbriquées parcouruent toutes les paires possibles de sommets, soit  $O(n^2)$  itérations, avec des opérations associées telles que la copie de la route, l'inversion d'un segment, et le calcul du coût, chacune ayant une complexité  $O(n)$ . Ainsi, chaque itération de la boucle externe a une complexité totale de  $O(n^3)$ , et pour  $k$  itérations, cela donne une complexité globale de  $O(k \cdot n^3)$ .

Dans le pire des cas,  $k$  peut atteindre  $O(n^2)$  si la convergence est **particulièrement lente**, entraînant une **complexité maximale** de  $O(n^5)$ . Cependant, en pratique,  $k$  est souvent bien inférieur, ce qui rend l'algorithme plus efficace qu'il n'y paraît dans une analyse théorique pessimiste. Cet algorithme est donc bien adapté pour des problèmes de **taille modérée**, mais peut devenir coûteux en termes de calcul pour de très grands ensembles de sommets.

## 4. Conclusion

En conclusion, l'application de l'**Algorithme Génétique** combiné à d'**autres heuristiques** pour le problème du **Voyageur de Commerce avec Fenêtres de Temps** montre un potentiel

intéressant pour résoudre ce **problème complexe**. L'AG permet d'explorer efficacement l'espace de recherche, tandis que l'intégration d'autres techniques heuristiques, comme la **recherche locale**, améliore la **qualité** des solutions et **réduit le temps de calcul**.

Bien que cette **approche hybride** n'offre pas de garantie de **solution optimale**, elle fournit des **solutions compétitives adaptées** aux **contraintes temporelles** du problème. Les résultats dépendent fortement des **paramètres** utilisés, ce qui souligne l'importance d'une bonne calibration de l'algorithme et des heuristiques associées pour **maximiser les performances**.

Ce travail représente une possible **contribution académique** pour ce type de problème, en offrant des **perspectives** et des **outils** pour aborder des problèmes complexes **d'optimisation**.

## 5. Bibliographie

Asani, E. O., Okeyinka, A. E., Ajagbe, S. A., Adebiyi, A. A., Ogundokun, R. O., Adekunle, T. S., Mudali, P., & Adigun, M. O. (2024). **A novel insertion solution for the travelling salesman problem**. *Computers, Materials and Continua*, 79(1), 1581-1597.  
<https://doi.org/10.32604/cmc.2024.047898>

Balakrishnan, N. (1993). **Simple heuristics for the vehicle routing problem with soft time windows**. *The Journal of the Operational Research Society*, 44(3), 279–287.  
<https://doi.org/10.2307/2584198>

Doganbas, M., & Shin, H. (2024). **Traveling salesman problem with backend information processing**. *Operations Research Letters*, 57, 107193.  
<https://doi.org/10.1016/j.orl.2024.107193>

Clarke, G., & Wright, J. W. (1964). **Scheduling of vehicles from a central depot to a number of delivery points**. *Operations Research*, 12(4), 568-581.  
<http://dx.doi.org/10.1287/opre.12.4.568>

Croes, G. A. (1958). **A method for solving traveling-salesman problems**. *Operations Research*, 6(6), 791-812. <https://doi.org/10.1287/opre.6.6.791>

Engels, C., & Manthey, B. (2009). **Average-case approximation ratio of the 2-opt algorithm for the TSP**. *Operations Research Letters*, 37(2), 83-84.  
<https://doi.org/10.1016/j.orl.2008.12.002>

Fu, Z., Eglese, R., & Li, L. Y. O. (2008). **A unified tabu search algorithm for vehicle routing problems with soft time windows**. *The Journal of the Operational Research Society*, 59(5), 663–673. <https://doi.org/10.1057/palgrave.jors.2602466>

Gridin, I. (2021). **Learning genetic algorithms with Python: Empower your AI applications with powerful genetic algorithm capabilities**. BPB Publications.

Karp, R. M. (1972). **Reducibility among combinatorial problems**. *The IBM Research Symposia Series*. Springer. [https://doi.org/10.1007/978-1-4684-2001-2\\_9](https://doi.org/10.1007/978-1-4684-2001-2_9)

Matai, R., Singh, S. P., & Mittal, M. L. (2010). **Traveling salesman problem: an overview of applications, formulations, and solution approaches**. *Traveling salesman problem, theory and applications*, 1(1), 1-25.

Pop, P. C., Cosma, O., Sabo, C., & Pop Sitar, C. (2024). **A comprehensive survey on the generalized traveling salesman problem**. *European Journal of Operational Research*, 314(3), 819–835. <https://doi.org/10.1016/j.ejor.2023.07.022>

Puerto, J., & Valverde, C. (2024). **The hampered travelling salesman problem with neighbourhoods**. *Computers & Industrial Engineering*, 188, 109889. <https://doi.org/10.1016/j.cie.2024.109889>

Rosa Villamayor-Paredes, M. M., Maidana-Benítez, L. V., Colbes, J., & Pinto-Roa, D. P. (2023). **Routing, modulation level, and spectrum assignment in elastic optical networks**. A route-permutation based genetic algorithms. *Optical Switching and Networking*, 47, 100710. <https://doi.org/10.1016/j.osn.2022.100710>

Taillard, É., Badeau, P., Gendreau, M., Guertin, F., & Potvin, J.-Y. (1997). **A tabu search heuristic for the vehicle routing problem with soft time windows**. *Transportation Science*, 31(2), 170–186. <https://doi.org/10.1287/trsc.31.2.170>

Wang, C., Zhu, R., Jiang, Y., Liu, W., Jeon, S.-W., Sun, L., & Wang, H. (2022). **A scheme library-based ant colony optimization with 2-opt local search for dynamic traveling salesman problem**. CMES - Computer Modeling in Engineering and Sciences, 135(2), 1209-1228. <https://doi.org/10.32604/cmes.2022.022807>

Zhao, J., Hu, H., Han, Y., & Cai, Y. (2023). **A review of unmanned vehicle distribution optimization models and algorithms**. *Journal of Traffic and Transportation Engineering (English Edition)*, 10(4), 548-559. <https://doi.org/10.1016/j.jtte.2023.07.002>