CL10 – MOBILITE ET LOGISTIQUE URBAINE

[C&W, 2-OPT, Meilleure insertion, TSPTW, VRPTW]

SIMON BAYLE PAUL CHESNEAU

Automne

utt
UNIVERSITÉ DE TECHNOLOGIE
TROYES

Table des matières

I. Introduction à la problématique du voyageur de commerce

Le problème du voyageur de commerce (TSP pour Travelling Salesman Problem) est un problème classique de la recherche opérationnelle et de l'informatique.

Il consiste à trouver le parcours le plus court qui permet à un voyageur de visiter un ensemble de villes et de retourner à sa ville de départ.

Le TSP est un problème NP-difficile, ce qui signifie qu'il est très difficile de trouver une solution optimale pour des instances de grande taille en temps raisonnable.

Cependant, il existe de nombreuses heuristiques qui permettent de trouver des solutions de bonne qualité en un temps raisonnable pour de nombreuses, ceci est l'objet de notre étude dans ce projet. Nous nous intéresserons à la résolution par méthode exacte et par des heuristiques, algorithme génétique, Clark and Wright, meilleure insertion, Closest Neighbors ainsi qu'à de la recherche locale avec 2-OPT.

Le TSP a de nombreuses applications pratiques, notamment dans les domaines de la logistique et de la gestion de flottes de véhicules. Il est également utilisé comme un problème de test pour évaluer les performances de différents algorithmes de résolution de problèmes de recherche opérationnelle.

II. Modèle mathématique et méthode exacte

A. Modèle mathématique

Le problème étudié dans ce projet est celui du voyageur de commerce avec des contraintes de temps souples. C'est-à-dire que l'on s'autorise un certain retard lors du parcours de la tournée (ce dernier étant pénalisé par un coefficient), il est cependant impossible d'arriver en avance, avant l'ouverture de la fenêtre de temps.

1. Variables de modélisation du problème

Les variables du problème sont les suivantes :

$$x_{i,j} : nombre\ binaire\ tq, \begin{cases} x_{i,j} = 1\ si\ le\ client\ j\ est\ visité\ après\ le\ client\ i. \\ x_{i,j} = 0\ sinon. \end{cases}$$

$R : variable\ servant\ à\ linéariser\ la\ fonction\ max.$

$\alpha : coefficient\ de\ pénalité\ de\ retard.$

$t(i) : temps\ de\ visite\ du\ client\ i.$

$a(i)$: $début\ de\ la\ fenêtre\ de\ temps\ du\ client\ i$.

$b(i) : fin\ de\ la\ fenêtre\ de\ temps\ du\ client\ i$.

$dist(i, i+1) : renvoie\ la\ distance\ entre\ le\ client\ i\ et\ i+1\ de\ la\ route.$

n : $nombre\ total\ de\ client$ à $visiter.$

C : $ensemble\ de\ client$ à $visiter.$

C' : $ensemble\ de\ client$ à $visiter\ sans\ prendre\ en\ compte\ le\ dépôt.$

$dmax : somme\ de\ toutes\ les\ distances\ qu'il\ est\ théoriquement\ possible\ de\ parcourir.$

$t(i) : modélisation\ \ du\ temps\ d'arrivée\ au\ client\ i.$

2. Modélisation

La fonction objectif donnée par le sujet est la suivante :

$$\sum_{i=0}^{n-1} dist(i, i+1) + dist(n-1, 0) + \alpha \sum_{i=1}^{n-1} t(i) - b(i) \qquad (1)$$

Le premier terme prend en compte l'optimisation suivant la distance entre les points et le deuxième terme la pénalité induite par les fenêtres de temps.

Pour résoudre la problématique de TSP, on va plutôt utiliser la formulation de la fonction objectif suivante :

$$\sum_{i=0}^{n-1} dist(i, i+1) + dist(n-1,0) + \sum_{i=1}^{n-1} \alpha(i) \times R(i) \qquad (1)$$

En effet, on ne doit pas récompenser le livreur s'il arrive en avance de la fenêtre de temps, le deuxième terme de la fonction objectif ne peut pas être négatif. On va donc remplacer le deuxième terme par $\max(0, t(i) - b(i))$.

Cependant, la fonction $\max()$ n'est pas implémentable sur un solver tel que pyscipopt car elle n'est pas linéaire. Pour pallier à ce problème, on va utiliser la variable R qui permet de linéariser la fonction $\max()$, cela est rendu possible par les contraintes (5) et (6).

Les contraintes sont les suivantes :

$$\sum_{j=0}^{n} x_{0,j} = 1 \qquad (2)$$

Cette contrainte permet de faire en sorte qu'il n'y ait qu'un seul arc sortant du dépôt initialement.

$$\forall j \in C, \quad \sum_{i=0}^{n} x_{i,j} = 1 \qquad (3)$$

Cette équation permet faire en sorte que tous les clients sont visités.

$$\forall j \in C, \quad \sum_{i=0}^{n} x_{i,j} - \sum_{i=O}^{n} x_{j,i} = 0 \qquad (4)$$

Cette équation permet de respecter la contrainte de flux.

$$\forall i \in C, \quad R(i) \geq 0 \qquad (5)$$

Première équation de linéarisation de la fonction $\max()$.

$$\forall i \in C, \quad R(i) \geq t(i) - b(i) \qquad (6)$$

Deuxième équation de linéarisation de la fonction $\max()$.

$$\forall i \in C, \forall j \in C', \quad t(j) \geq t(i) + dist(i,j) - 4dmax \times (1 - x_{i,j}) \tag{7}$$

Contrainte MTZ qui permet l'élimination des sous-tours, ici on prend $4dmax$ comme une grande valeur supposée ne pas pouvoir être atteinte.

$$\forall j \in C, \quad t(j) \geq a(i) \tag{8}$$

Cette contrainte permet de respecter la borne inférieure des fenêtres de temps. On ne peut pas arriver avant le début de la fenêtre de temps.

$$t(0) = 0 \tag{9}$$

On initialise le temps à 0.

B. Analyse des résultats sur différentes instances

Après avoir implémenté ces différentes contraintes sur python, on a testé notre algorithme sur plusieurs instances différentes dont voici les résultats :

	Solving Time	Gap	Fonction obj
Instance projet	3600	150,07%	880.273269527334
Instance projet	7200	93,77%	701.6410105805238
Instance 40_1	7200	890,84%	6960.495095888837
Instance 40_2	7200	543,96%	1627.9747611543569
Instance 40_3	7200	81,37%	1138.931326546927
Instance 100_1	3600	21246,25%	269134.1196802761
Instance 100_2	3600	117871,79%	75538.80074909255
Instance 100_3	3600	153550.56 %	98384.36528056275

Tableau 1 Bilan des solutions obtenues par le solver pour un temps d'exécution donné en seconde

En les comparants avec les algorithmes génétiques on se rend compte que la méthode exacte met beaucoup de temps pour converger et que les résultats sont loins de ce qu'on obtient.

III. Etude des heuristiques, Clark and Wright, meilleure insertion

A. Principe et fonctionnement des heuristiques

1. Principe

Les différentes heuristiques étudiées dans ce projet sont utilisées afin d'obtenir une première solution du TSP correcte et pouvant ensuite servir de brique de base pour l'initialisation de la population d'un algorithme génétique.

2. Fonctionnement

- Clarke and Wright :

Notre implantation de Clark and Wright se divise en deux parties. Dans une première partie notre algorithme initialise une population de route qui sont constituées d'un client et du dépôt les solutions sont de cette forme $[0, i, 0]$, cette première population visite les clients une et une seule fois. Chaque élément de cette route est noté $route[i]$.

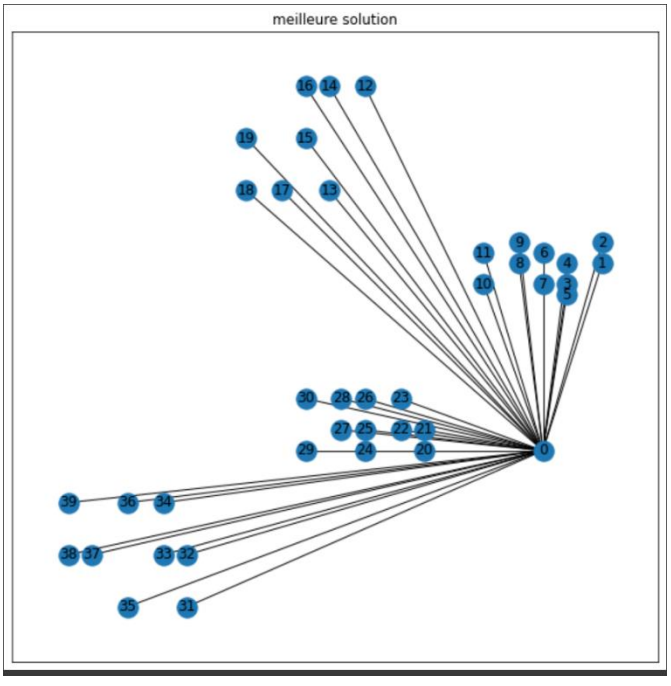


Figure 1 solution initiale du Clarke&Wright pour une instance de 40 clients (Inst40_2)

Ensuite, en fin d'initialisation on calcule les savings avec la formule suivante :

$$\forall i,j \ tq \ i \neq j, \qquad saving = dist\big[0, route[i][1]\big] + dist[route[j][1], 0] - dist[route[i][1], route[j][1]]$$

A la suite de ces calculs on peut sélectionner les couples de clients qui ont les saving les plus élevés et les intégrer dans la route. Ainsi on obtient en fin d'initialisation :

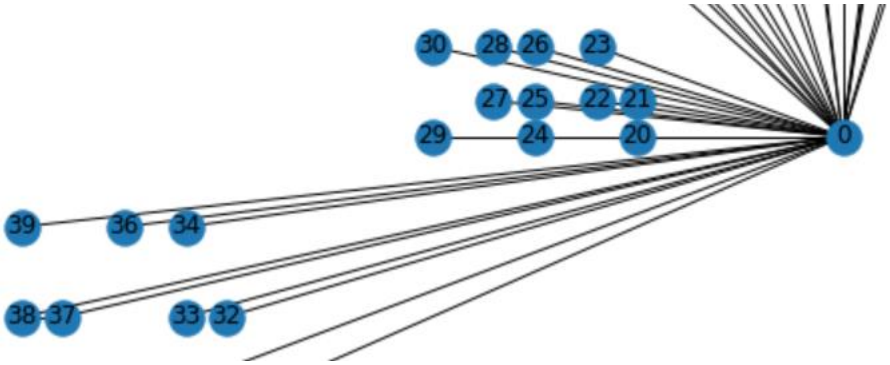


Figure 2 fin de l'initialisation Clarke&Wright instance de 40 clients (Inst40_2)

En fin d'initialisation on voit que les clients 38 et 37 sont intégrés dans la même route. On itère ensuite de la même façon jusqu'à ce que tous les clients soient dans une seule et même tournée. Pour cela on poursuit l'insertion de clients dans la route qui contient le plus de clients en fin d'initialisation, ici la route $[0,38,37,0]$ Jusqu'à obtenir une tournée qui visite tous les clients.

On a le pseudo-code suivant :

```
Visited = Liste vierge pour stocker les sommets visités
Route = Liste vierge
C_route =
Tant que la route ne passe pas par tous les sommets :
      a = somment choisis aléatoirement
      Si a n'est pas dans visited :
            Route = Route + pétale : [dépôt, A, dépôt]
            Ajoute A à visited
Pour tous les sommets i dans Route :
      Pour tous les sommets j dans Route :
            Si i != j et (dist[0,route[i][1]] + dist[route[j][1],0] -
            dist[route[i][1], route[j][1]]) n'est pas déjà dans c_route[][0] alors :
                  On ajoute à C_route :
                  [dist[0,route[i][1]]+dist[route[j][1],0] - dist[route[i]
                  [1],route[j][1]], [0,route[i][1],route[j][1],0]]
I = 0
Tant que i != de la longueur de la route :
      Si i est inférieur (n-1) -3 :
            Pour tous les sommets j compris dans la route :
```

```
                    Si Route[i][j] in C_route[0][1] :
                          On supprime la Route[i]
I += 1
On ajoute à la route C_route[0][1]
Route_init = C_route[0][1]
Amelioration == True
Tant que Amelioration == True :
      Amelioration = False
      C_route = []
      Pour chaque sommet j dans la route
            Si route[j][-2] n'est pas dans Route_init :
                  Ajoute à C_route : [dist[0,route_init[1]] + dist[route[j][-2],0] -
            dist[route[j][-2],route_init[1]], [route[j][:len(route[j])-
            1]+route_init[1:]]]
                  Ajoute à C_route : [dist[route_init[-2],0] + dist[0,route[j][1]] -
            dist[route[j][1],route_init[-2]], [route_init[:len(route_init)-
            1]+route[j][1:]]]
      On inverse C_route
      Pour tous les sommets i dans Route[i]
            Si i est supérieur au nombre de sommets dans la route :
                  Break
            Pour tous les sommets j dans Route[j]
                  Si route[i][j] est dans C_route[0][1][0] :
                        On supprime route[i]
                        Si i égale au nombre de sommets dans la route :
                              Break
      S'il y a plus que 3 sommets dans route[-1] :
            On supprime route[-1]
      Si C_route n'est pas vide :
            On ajoute à route : C_route[0][1][0]
            Route_init = C_route[0][1][0]
            Amélioration = True
      Si le nombre de sommet dans C_route[0][1][0] égale nombre de sommet total + 1:
            On vide route
            On ajoute à route : C_route[0][1][0]
            Amélioration = False
Route = route[0]
Retourne route
```

Analyse de la complexité :

En ce qui concerne la première boucle de l'algorithme, qui génère une solution initiale, la complexité est de l'ordre de $O(n)$, Car elle parcourt tous les points une seule fois. Ensuite pour le calcul des savings on a 2 boucles imbriqués, la complexité évolue donc en $O(n^2)$.

L'initialisation est donc en $O(n^2)$

Enfin on a une boucle while qui itère tant qu'on ne trouver pas d'amélioration mais cela revient à itérer sur une boucle de taille la longueur de la route. On a ensuite des boucles for qui itère sur n. Cette boucle est donc aussi en $O(n^2)$.

Le Clarke&Wright a donc une complexité en $O(n^2)$.

Nous avons 2 versions de cette heuristique car elle peut aussi être randomisée si, en fin d'initialisation on ne sélectionne plus le client qui a le meilleur saving mais un client random. On obtient ainsi des résultats différents pour chaque utilisation de Clarke&Wright ce qui nous permet d'avoir de la diversité dans nos solutions. C'est cette méthode qui est ensuite utilisée pour l'initialisation de la population de notre algorithme génétique.

Cependant la façon dont l'algorithme Clarke&Wright présenté calcule les coûts n'est pas optimale car il ne prend pas en compte les fenêtres de temps lors du calcul du coût. C'est une heuristique qui permet d'avoir une solution optimisée pour les distances.

- Meilleure insertion

De même que pour Clarke&Wright, notre méthode de meilleure insertion se divise en deux étapes. La première partie qui constitue l'initialisation consiste à créer une route avec un client pris de façon aléatoire, on alors une solution du type $[0, client, 0]$
A partir de cette route et tant que la route ne parcours pas l'ensemble des clients on vient rajouter les points qui minimisent les coûts de la route. Les points sont rajoutés aléatoirement devant ou derrière un des points de la route lorsqu'ils minimisent la fonction couts.

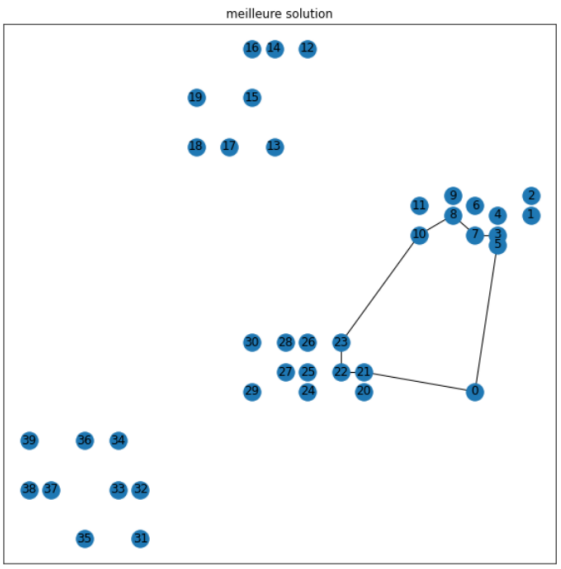


Figure 3 état de l'insertion des clients au cours d'une meilleure insertion lorsque la population de la route vaut 10 pour une instance de 40 clients (Inst40_2)

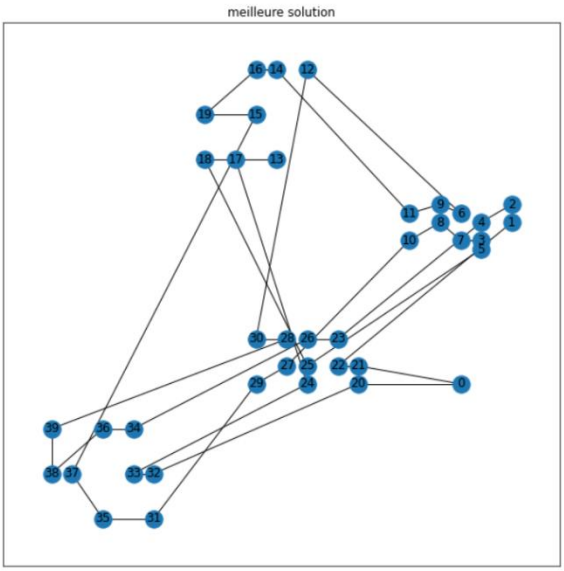


Figure 4 Meilleure insertion terminé pour une instance de 40 clients (Inst40_2)

On a le pseudo-code suivant :

```
Insertion1 = choix d'un somment au hasard
R = [dépôt, insert, dépôt]
Tant que la route R ne passe pas par tous les sommets :
      Création de Min_liste = Array de dimensions : len(R)-1 par 2
      Création de : Liste
      Pour chaque sommet i dans la route R:
            Pour chaque sommet :
                  Si le somment n'est pas dans la route :
                        Création : variable aléatoire binaire : var_rand
                        Si var_rand = 0 :
                              On insert p au rang i + 1 dans la route R
                              Liste[cout(R)] = [p, i +1]
                        Si var_rand = 1 :
                              On insert p au rang i - 1 dans la route R
                              Liste[cout(R)] = [p, i -1]
                        On retire p de la route
      Tris de la Liste pour enlever les éléments vides
      Si Liste n'est pas vide :
            On insère dans la route au rang Liste[0][1][1] le sommet Liste[0][1][0]
Retourne route
```

Analyse de la complexité :

La complexité de l'algorithme de meilleure insertion est $O(n^2)$ car pour chaque sommet d'une route donnée on regarde l'ensemble des sommets qui ne sont pas dans la route avant d'insérer le bon sommet au bon endroit.

Etant calculé à partir de la fonction couts, méthode de meilleure insertion permet d'inclure des clients dans une route en fonction de la pénalité de temps et des distances ce qui en fait l'heuristique la plus précise de ce projet. De plus le premier client inséré dans la route est aléatoire et l'insertion de nouveaux clients dans la route est également aléatoire ce qui en fait une heuristique randomisée.

- Plus proche voisin

Le plus proche voisin est une heuristique qui permet de construire une route en y ajoutant les clients par proximité géographique. Cette heuristique optimise localement la solution et dans notre algorithme, nous avons utilisé une version aléatoire qui ajoute des clients de manière aléatoire a la route en essayant de sélectionner les candidats parmi les plus proche.

Voici le pseudo code de la solution :

```
Visited = Liste d'une taille égale au nombre de sommet remplis de False
Route = Liste d'une taille égale au nombre de sommet
Route[0] = 0
Visited[0] = True
Pour le sommet i compris dans l'ensemble des sommets – le dépôt :
       R = rand_two_closest(route[i], visited)[1]
       Route[i+1] = R
       Visited[R] = True
On ajoute le dépôt à la route
Retourne Route


[1] Rand_two_closest (r, visited):
V1 = 0
Dref = 10000
Pour chaque sommet i dans l'ensemble des sommets :
       Si dist[r,i] est inférieur à dref et visited [i] est Faux alors :
              Dref = dist[r,j]
              V1 = i
V2 = -1
Dref = 10000
Pour chaque sommet i dans l'ensemble des sommets :
       Si dist[r,i] est inférieur à dref et visited [i] est Faux et i ! = v1 alors :
              Dref = dist[r,j]
              V2 = i
A = random.randint(0,1)
Si v2 égale -1 ou A égale 1 alors :
Retourne v1
Sinon retourne v2
```

Analyse de la complexité :

La complexité de l'algorithme du plus proche voisin est $O(2n^2)$ car on parcourt deux fois l'ensemble des sommets pour trouver les deux plus proches voisins et en sélectionner un aléatoirement dans l'algorithme rand_two_opt et cet algo est appelé dans une boucle pour avec un nombre d'itération égale au nombre de sommets.

- 2-OPT

Notre implémentation du 2-OPT est un algorithme de recherche locale qui part d'une solution initiale et essaie de l'améliorer en effectuant des modifications locales à l'itinéraire. Le principe de base de l'algorithme 2-OPT est de choisir deux arêtes dans l'itinéraire et de les "couper" avant de les "recoller" de manière à ce qu'elles forment un nouvel itinéraire.

Si le nouvel itinéraire est meilleur que l'ancien, on le garde et on recommence le processus jusqu'à ce qu'on ne puisse plus améliorer l'itinéraire de cette manière.

Pour un cas de TSP classique nous avons les résultats suivants.

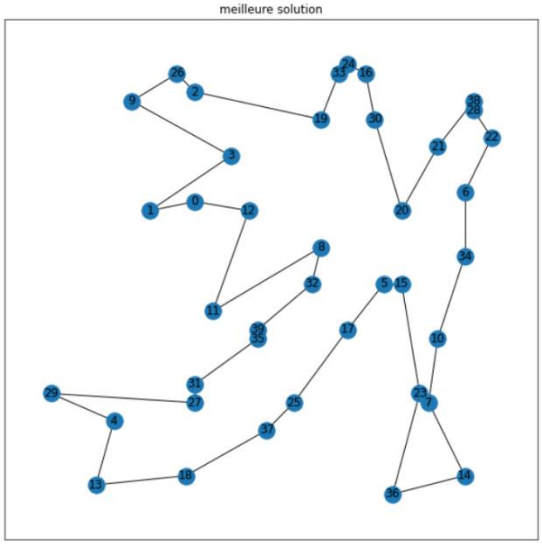


Figure 5 tournée de véhicule sans recherche locale avec 2-OPT pour une tournée R sans fenêtre de temps

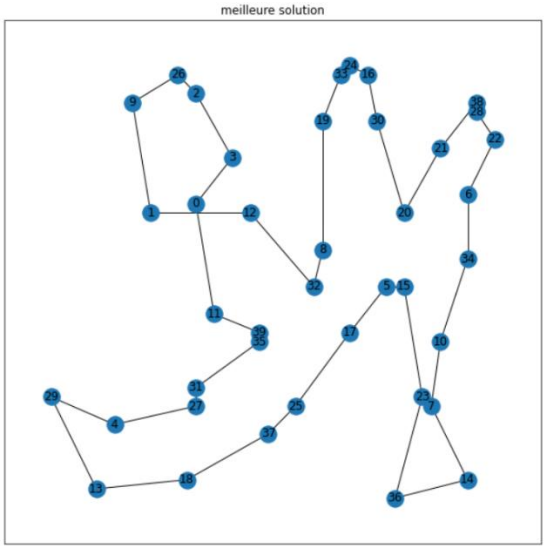


Figure 6 tournée de véhicule avec recherche locale 2-OPT pour une tournée R sans fenêtre de temps

Nous analyserons ensuite notre version du 2-OPT dans un cas avec fenêtre de temps car c'est ce qui est utilisé par la suite dans notre algorithme génétique. Dans ce cas le calcul qui sert à savoir si on interchange 2 arrêtes de solution est fait en fonction de la fonction coût.

Pseudo-code :

```
Initialisation d'amélioration à true
Tant qu'amélioration vaut true:
	Amélioration == false
	Pour chaque sommet i entre 1 et longueur de la route – 2 :
		Pour chaque sommet i entre 1 et longueur de la route – 2 :
			Si j != i-1 et j != i et j != i+1 :
				Initialise route_TW
				Pour chaque somment k de la route :
					Route_TW[k] = route[k]
				Route_TW[i+1] = route[j]
				Route_TW[j] = route[i+1]
				Route_TW[i+2:j] = inverse de route[i+1:j]
				Si le cout de route > au cout de la route TW
					Route = route_TW
					Amélioration = true
Retourne route
```

Analyse de complexité :

La complexité de l'algorithme 2-Opt est $O(n^2)$ car tant qu'il n'y a pas d'amélioration on parcourt la route avec deux index pour trouver les meilleurs sommets à échanger.

IV. Principe et analyse de l'algorithme génétique

A. Principe

1. Introduction

Un algorithme génétique est un type d'algorithme qui utilise des techniques inspirées de la biologie pour trouver des solutions à des problèmes complexes. Le principe de base d'un algorithme génétique est de simuler le processus de sélection naturelle, qui est le mécanisme par lequel la nature favorise les individus les plus adaptés à leur environnement. Pour cela on fait évoluer une population dans un environnement en n'en sélectionnant qu'une partie.

Notre algorithme est constitué des différentes heuristique étudiée précédemment, elles servent à initialiser la population. Cette population est ensuite mutée pour essayer d'améliorer la solution optimale. Pour assurer un lien entre les différentes générations et pour pouvoir permettre d'optimiser la solution, une fonction cross_over permet de générer de nouvelle solution à partir de certaines préexistantes. Cependant dans notre algorithme nous ne supprimons pas les solutions parentes de nouvelle solution. A la fin d'une génération on procède à la sélection des meilleurs individus, dans notre algorithme nous sélectionnons les m Meilleurs.

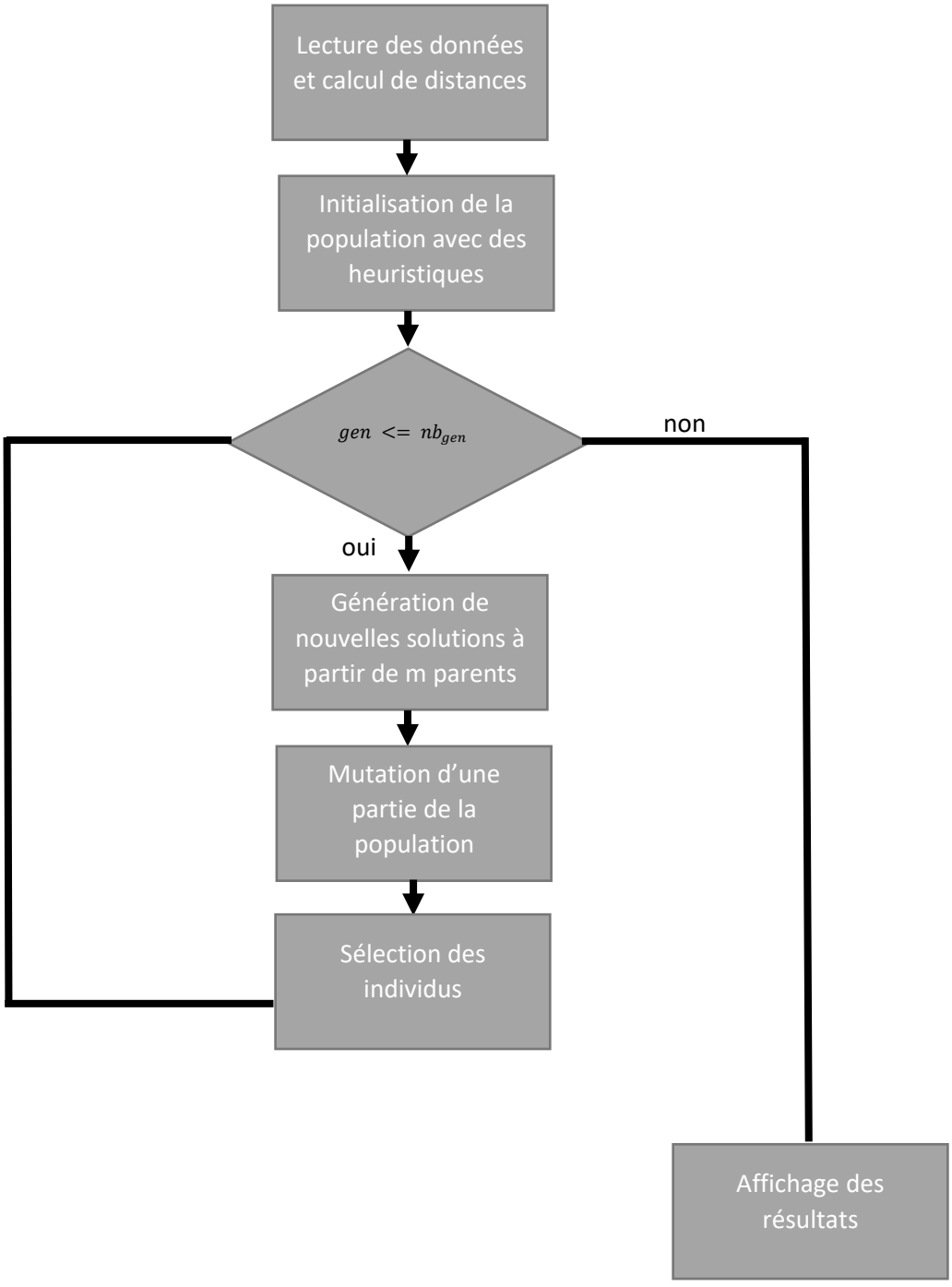
On a les notations suivantes :

m : $taille\ de\ la\ population\ de\ solution$

nb_{gen} : $nombre\ de\ génération\ de\ l'algorithme\ génétique$

gen : $l'indice\ d'itération\ de\ la\ boucle\ principale\ (\leq nb_{gen})$

Voici le logigramme de notre algorithme génétique :



2. Analyse des différentes fonctions de l'algorithme

- Cross_over

Cette fonction sert à générer de nouvelles solutions à partir de la population existante. Ces solutions sont ensuite ajoutées à la population comme si c'était des « enfants ». Dans notre implémentation de l'algorithme génétique cette fonction consiste à générer des enfants qui sont à moitié constitués du parent 1 et à l'autre moitié du parent 2. Ensuite, dans un deuxième temps si l'enfant est constitué de clients qui se répètent on vient les remplacer par un autre point de manière aléatoire.

- Mutation

On a 2 types de mutation, une première qui va réaliser un nombre aléatoire de swap et une deuxième qui vient inverser totalement la solution existante.

- Sélection

Nous avons testé plusieurs types de sélection, tournois, sélection aléatoire, et sélection semi aléatoire. Finalement nous avons gardé en solution finale une sélection semi aléatoire qui va sélectionner une partie des individus de façon aléatoire et l'autre partie est constitué des meilleurs individus.

B. Analyse des résultats

Pour l'analyse des résultats, nous séparerons les résultats en deux parties. En effet, on a 2 types d'algorithmes, dans le premier algorithme on n'utilise pas la meilleure insertion et dans un second algorithme on utilise l'heuristique de meilleure insertion. On obtient donc des courbes de convergence bien différentes.

L'algorithme utilisant la meilleure insertion donne de meilleurs résultats et est plus précis avec un écart type plus faible mais il converge beaucoup plus vite et n'améliore que très peu la solution. L'algorithme génétique sans la meilleure insertion converge mon vite mais il a une moins bonne précision et donne des résultats moins bons.

1. Algorithme génétique sans meilleure insertion

a) Test sur une instance

On a testé les algorithmes sur diverses instances de test, dont l'instance : Instance

Pour le paramétrage suivant, voici le type de courbe de convergence que l'on obtient :

Paramétrage :	m	110
	Probabilité de mutation :	40%
	Pourcentage de la population mutée :	20%
	Nombre de génération :	1100
	Pourcentage meilleure insertion :	0
	Pourcentage random	m/3
	Pourcentage clark and wright	m/4
	Pourcentage closest neighbour	m - m/4-m/3
	Pile face	1
	Elite	1

Tableau 1 paramétrage du test

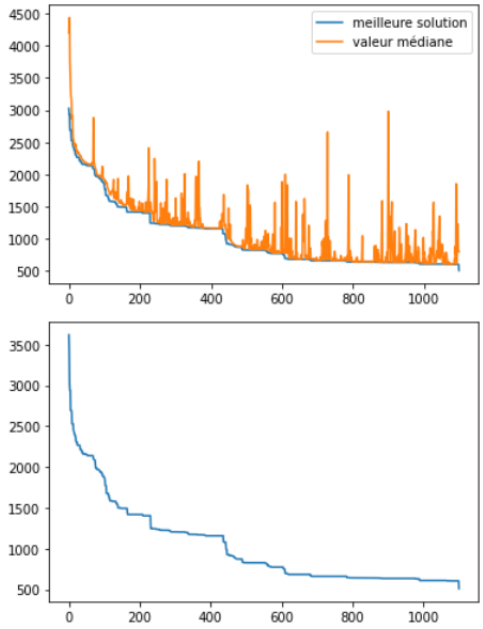


Figure 7 évolution de la meilleure valeur de coût et de la médiane de la population en fonction du nombre d'itérations

Sur ce test la valeur du coût à l'optimal est de 509,47

La solution trouvée est la suivante : [0, 31, 27, 33, 34, 32, 29, 13, 18, 11, 2, 30, 38, 28, 22, 6, 20, 15, 5, 17, 10, 23, 14, 36, 37, 25, 35, 39, 12, 3, 9, 26, 19, 24, 16, 21, 8, 7, 4, 1, 0]

Le temps d'exécution est de 29,1 s

On observe une bonne convergence sur cette instance et on remarque l'utilité de la recherche locale à la fin qui permet d'améliorer grandement la solution finale. On observe également que la valeur médiane de la population fluctue beaucoup ce qui est peut-être une conséquence de notre méthode de sélection semi-aléatoire.

b) Analyse statistique globale de la solution

Afin de pouvoir avoir une idée des écarts entre les différentes valeurs obtenues, on a calculé les écarts-type sur plusieurs instances. Pour cela on a utilisé les valeurs des différentes instances, pour un même paramétrage en lançant chaque algorithme 10 fois.

On obtient les résultats suivants :

Meilleure	Moyenne	écart-type	Instance
486,74	536,641	44,72	Instance
6505,81	6878,914	318,37	Instance_40_1
709,21	805,437	58,99	Instance_40_2
625,69	694,999	83,05	Instance_40_3

Tableau 2 Bilan des résultats d'analyse sur un panel de 10 tests

On observe que les écart-type sont proportionnels à la valeur de la fonction objectif. On observe surtout que les valeurs obtenues ont une forte dispersion car on sait que 95% des valeurs sont comprises dans un intervalle de 2 fois l'écart type autour de la valeur moyenne. Cela laisse présager d'une forte dispersion des données.

2. Algorithme avec meilleure insertion

a) Test sur une instance

On a également testé cet algorithme sur différentes instances de test.

La présentation des résultats suivant a été faite sur l'instance : Instance

Pour le paramétrage suivant, voici le type de courbe de convergence que l'on obtient :

Paramétrage :	m	110
	Probabilité de mutation :	40%
	Pourcentage de la population mutée :	45%
	Nombre de génération :	700
	Pourcentage meilleure insertion :	m/10
	Pourcentage random	m/3
	Pourcentage clark and wright	m/4
	Pourcentage closest neighbour	m - m/4-m/3-m/10
	Pile face	1
	Elite	1

Tableau 3 paramétrage du test sur Instance

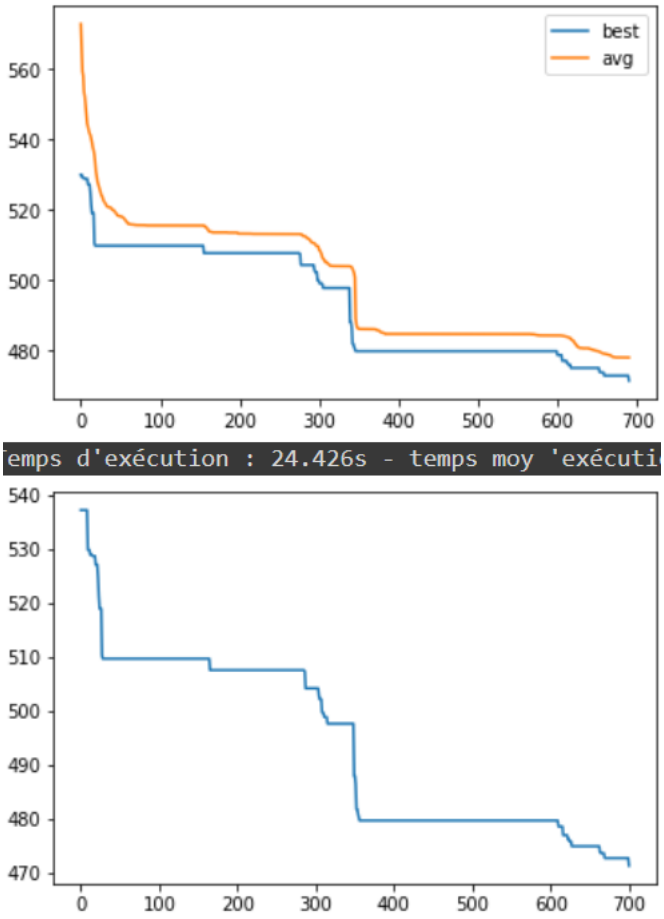


Figure 8 évolution de la meilleure valeur de coût et de la médiane de la population en fonction du nombre d'itérations

Sur ce test la valeur optimale est de 471,23

La solution trouvée est la suivante : [0, 33, 34, 31, 27, 29, 13, 18, 11, 32, 2, 30, 38, 28, 22, 21, 6, 20, 5, 15, 10, 23, 14, 36, 37, 25, 35, 39, 12, 3, 9, 26, 24, 16, 19, 8, 17, 7, 4, 1, 0]

Le temps d'exécution est de 24,4 s

De même que sur le premier algorithme, on observe une convergence vers la solution optimale. Cependant avec cette méthode, la convergence peut être parfois très rapide et l'heuristique de meilleure insertion peut parfois trouver une solution assez proche de l'optimal dès le début.

Ainsi, on peut également tomber sur ce type de convergence :

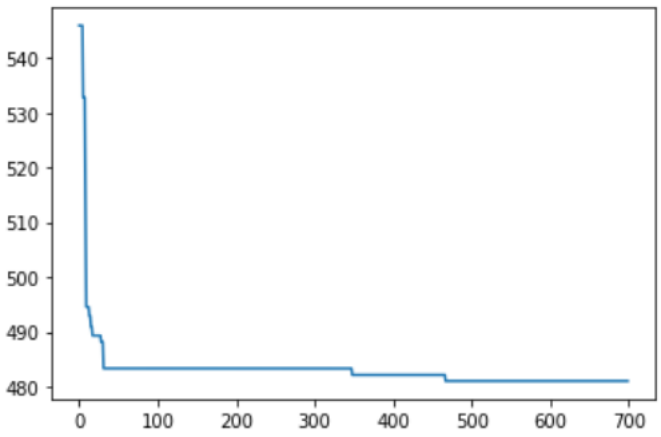


Figure 9 Exemple de convergence très rapide avec l'algorithme utilisant la meilleure insertion

b) Analyse statistique globale de la solution

Dans les mêmes conditions que précédemment, on a testé notre algorithme génétique 10 fois et voici les résultats :

Meilleure	Moyenne	Écart-type	Instance
471,23	486,406	13,33	Instance
6482,66	6616,37667	196,55	Instance_40_1
719,23	732,87	13,17	Instance_40_2
626,65	645,91	14,06	Instance_40_3

Tableau 4 Bilan des résultats d'analyse sur un panel de 10 tests

On remarque que comparativement aux données précédentes, les écart-types sont bien plus fiables et les moyennes bien plus basses sur toutes les instances. Cependant les meilleurs résultats ne sont pas forcément meilleurs avec cette solution. Pour les instances 2 et 3 la méthode précédente donne de meilleurs résultats. Cependant une optimisation des hyperparamètres sur ce modèle sera beaucoup plus efficace que sur l'autre car l'écart type est plus faible.

3. Instance de 100 Clients

Les 2 algorithmes ont également été lancé sur des instances de 100 clients et voici les résultats.

Type de méthode	Moyenne	meilleure	écart-type	Instance
Algo génétique avec meilleure insertion	27336,8775	25690,54	1408	Instance_100_1
Algo génétique sans meilleure insertion	27985,47	27116,99	733,81	Instance_100_1

Tableau 5 bilan des résultats d'analyse sur un panel de 10 tests

Avec cette instance de 100 clients on observe que l'algorithme génétique avec la meilleure insertion a l'écart type le plus élevé. Ce résultat vient contre dire les analyses précédentes, peut être que la précision de cette méthode baisse pour de plus grandes instances, ou alors c'est le nombre de test qui n'est pas suffisant pour pouvoir se prononcer sur ce type d'analyse.

V. Algorithme génétique pour résoudre un VRPTW avec SPLIT

A. Utilisation de l'algorithme SPLIT

Dans cette dernière partie du projet, nous avons implémenté l'algorithme SPLIT dans notre précédent algorithme afin de transformer le TSPTW en VRPTW.

Cependant l'utilisation de SPLIT nous sert ici d'heuristique elle ne nous donne pas la solution optimale. En effet on utilise le même fonctionnement que précédemment, on optimise une tournée de véhicule unique comme un TSP classique et au moment de calculer les coûts et d'intégrer la tournée dans la population on utilise SPLIT pour découper la tournée en respectant la capacité du véhicule.

On obtient les résultats suivants :

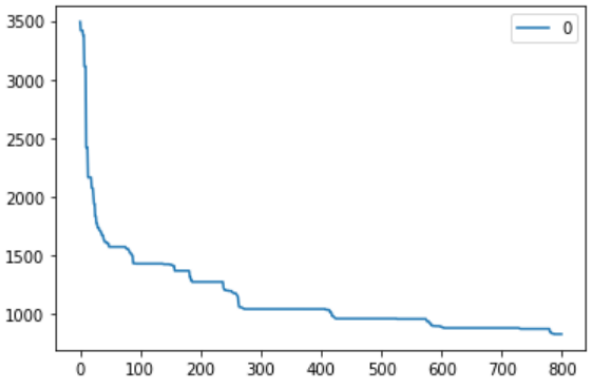


Figure 10 Evolution du cout de la meilleure solution pour un VRPTW sur l'instance Instance

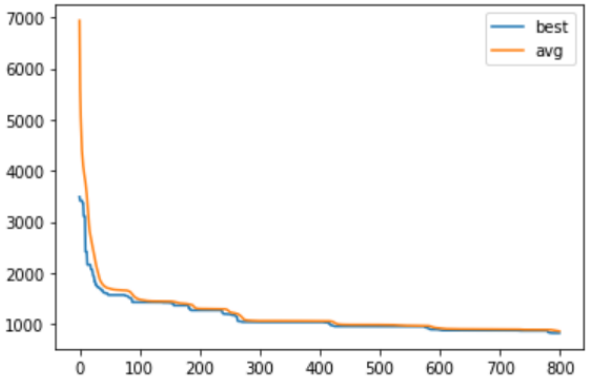


Figure 11 Evolution du cout de la meilleure solution et de la moyenne pour un VRPTW sur l'instance Instance

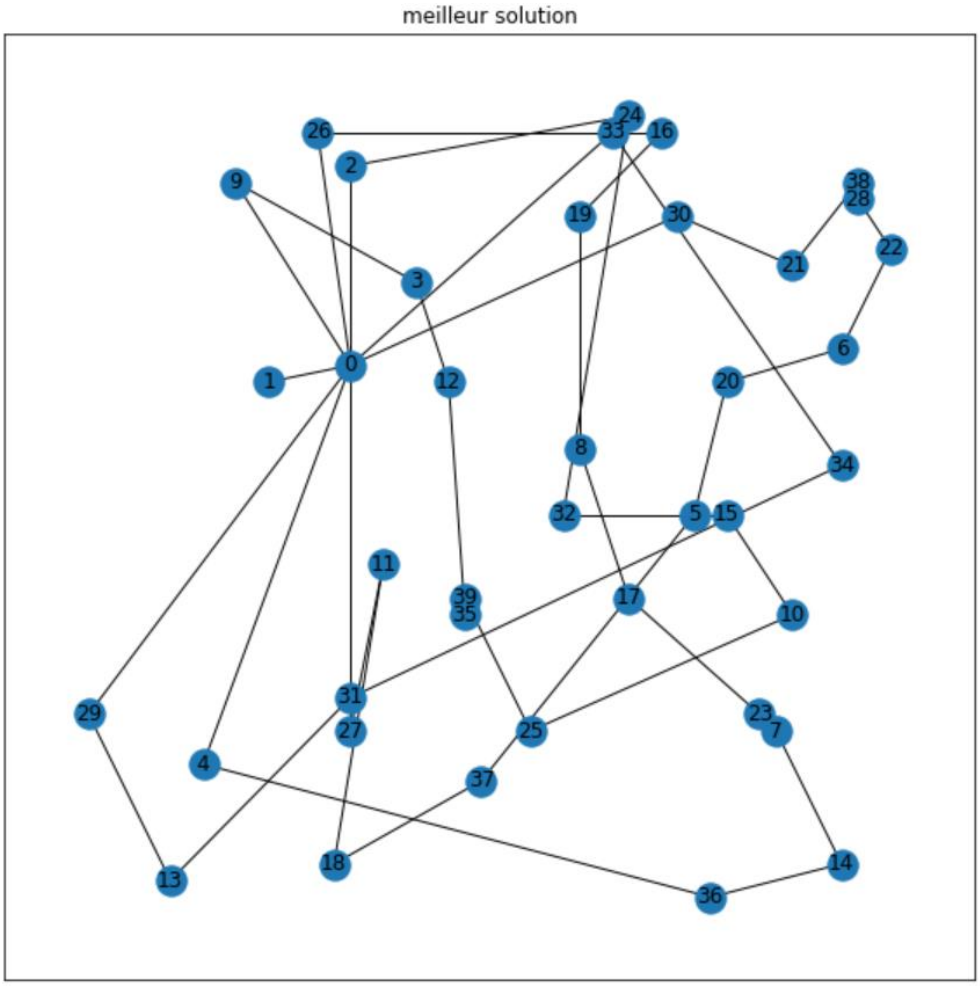


Figure 12 Affichage de la tournée VRPTW pour l'instance Instance

La solution optimale est 826.6829902266036

La tournée trouvée est : [[0, 33, 34, 31, 13, 29, 0], [0, 27, 11, 18, 37, 5, 20, 6, 22, 28, 38, 21, 30, 0], [0, 2, 24, 32, 15, 10, 25, 39, 35, 12, 3, 9, 0], [0, 26, 16, 19, 8, 17, 23, 7, 14, 36, 4, 0], [0, 1, 0]]

L'algorithme donne un résultat en 651,9s