Université de Technologie de Troyes - Automne 2020

Projet CL10

Sujet :

Mobilité et logistique urbaine

Murat Afsar

Fait par :

Matthieu LEPICIER

Table des matières

1 Introduction **2**
1.1 Mise en contexte 2

2 Modélisation **3**
2.1 Objectif et Hypothèses 3
2.2 Variables de décision et paramètres 3
2.3 Formulation du modèle mathématique 4

3 Méthode de résolution **6**
3.1 Formulation du modèle mathématique de l'heuristique 6
3.2 Population initiale 6
3.3 Anti-convergence prématurée 8
3.4 Recherche locale 9

4 Résultats **12**
4.1 Présentation 12
4.1.1 Modèle optimal 12
4.1.2 Modèle heuristique : Algorithme génétique 13
4.2 Analyse numérique 14

5 Conclusion **16**

1 Introduction

1.1 Mise en contexte

Dans le cadre de l'UE CL10 (Mobilité et logistique urbaine), nous sommes amenés à réaliser un projet par groupe de 4 étudiants. L'objectif principal de ce projet est de modéliser un TSPTW (Travelling Salesman Problem with Time Windows), autrement dit, le problème de voyageur de commerce avec contraintes de fenêtres de temps. Afin de simplifier la résolution de ce problème NP-difficile, les fenêtres de temps, dans ce projet, seront considérées comme souples. C'est-à-dire qu'elles peuvent être violées et entrainer une pénalité que nous décrirons par la suite.

Du fait de la prise de conscience de l'impact environnemental des transports individuels, on observe une augmentation de l'utilisation des transports en commun en France. En effet d'après une étude, en 2019, il s'agit de 73% de Francais qui utilisent quotidiennement les bus, métros ou tramways et trains, contre 63% en 2014. La mobilité urbaine devient alors un enjeu crucial, et le sera d'autant plus dans les années à venir. Il devient alors primordial de maîtriser et optimiser sa logistique urbaine. Le TSPTW peut être appliqué pour prendre des décisions concernant ce sujet, tel que les parcours des bus scolaires. Il s'applique également beaucoup à ce qu'on appelle la logistique de dernier kilomètre et le transport de messagerie, comme par exemple la livraison de petits commerces.

Ce rapport présentera la modélisation mathématique du problème de TSPTW à travers différentes méthodes et outils de résolutions, ainsi qu'une analyse et comparaison de ces résultats.

2 Modélisation

2.1 Objectif et Hypothèses

Lorsque l'on travaille sur un problème de tournée de véhicule, le critère principal à prendre en compte est la distance totale parcourue par le véhicule pour effectuer la tournée complète. Le problème du voyageur de commerce est le suivant. À partir d'une liste de villes, et des distances entre toutes les paires de villes, il faut déterminer un plus court chemin qui visite chaque ville et qui termine dans la ville de départ. Notre objectif est de trouver une chaîne élémentaire qui minimise la distance totale.

Afin de modéliser le problème, nous effectuons différentes hypothèses. Chaque client est visité une seule et unique fois, idem pour le dépôt. Il ne peut donc pas y avoir plusieurs tournées. Lorsqu'on visite un sommet, on le quitte forcément. On impose également des fenêtres de temps. Chaque client doit être visité dans une plage horaire, mis à part le dépôt. Si le sommet est visité après ça, une pénalité s'applique. On considère que le temps d'arrivée à un sommet i dépend de la distance totale effectuée avant d'arriver à ce sommet. Pour finir, aucune contrainte de capacité n'est pris en compte.
À partir de ces hypothèses, nous avons formulé un modèle mathématique programmé sur GUSEK détaillé dans la prochaine partie, les variables et paramètres, la fonction objectif et les contraintes.

2.2 Variables de décision et paramètres

Pour modéliser mathématiquement le problème, il est nécessaire d'adopter certaines notations pour les paramètres et les variables de décisions. Pour simuler ce problème nous utiliserons les paramètres suivants :

n : le nombre de clients que notre tournée doit inclure ainsi que le dépôt
M : big M que l'on utilisera pour nos contraintes. Nous l'avons fixé à 100 000
cox_i et coy_i : les coordonnées des clients et du dépôt avec $i = 1, 2, ..., n$
$dist_{ij}$: Distance entre un sommet i et un sommet j, le dépôt est représenté par le sommet 1 et les clients le reste avec $i = 1, 2, ..., n$ et $j = 1, 2, ..., n$. On le calcule à partir des coordonnées cox_i et coy_i : si $i = j$ $dist_{ij} = 100000$, sinon $dist_{ij} = sqrt((cox[i] - cox[j])^2 + (coy[i] - coy[j])^2)$
a_i et b_i : Les fenêtres de temps pendant lesquels un client i peut être visité, avec $i = 1, 2, ..., n$
$penalite_i$: Si les fenêtres de temps ne sont pas respectées, une pénalité est appliquée. Pour chaque sommet i, une pénalité a été fixée, $i = 1, 2, ..., n$.

L'objectif du modèle est d'attribuer un ordre de visite des clients en partant du dépôt qui minimise la distance parcourue en respectant les fenêtres de temps ou appliquer des pénalités. Les variables de décisions sont donc les suivantes :

x_{ij} : variable binaire égale à 1 si le chemin du sommet i vers le sommet j est effectué, sinon égale à 0, avec $i = 1, 2, ..., n$.
y_i : variable binaire égale à 1 si l'instant de visite du sommet i dépasse la borne supérieure de la fenêtre de temps, sinon égale à 0, avec $i = 1, 2, ..., n$.
t_i : variable qui indique l'instant auquel un sommet i est visité, avec $i = 1, 2, ..., n$.
k_i : variable égale à t_i si $t_i > b_i$, sinon égale à b_i, avec $i = 1, 2, ..., n$.

2.3 Formulation du modèle mathématique

Maintenant que nous avons déclaré toutes les notations, nous pouvons modéliser le problème.

$$\text{Minimize} \sum_{i=1}^{n} \sum_{j=1}^{n} dist_{ij} * x_{ij} + \sum_{i=2}^{n} penalite_i * (k_i - b_i) \tag{1}$$

$$\text{subject to} : \sum_{i=1}^{n} x_{ij} = 1, \forall j = 1..n \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} - \sum_{i=1}^{n} x_{ji} = 0, \forall j = 1..n \tag{3}$$

$$t_j \geq a_j, \forall j = 1..n \tag{4}$$

$$t_j \geq t_i + dist_{ij} - M * (1 - x_{ij}), \forall i = 1..n, j = 2..n \tag{5}$$

$$t_i - b_i \leq M * y_i, \forall i = 2..n \tag{6}$$

$$b_i - t_i \leq M * (1 - y_i), \forall i = 2..n \tag{7}$$

$$k_i \geq t_i, \forall i = 2..n \tag{8}$$

$$k_i \geq (1 - y_i) * b_i, \forall i = 2..n \tag{9}$$

1. `Fonction objectif` : Lorsque la variable x_{ij} vaut 1, la distance du sommet i vers le sommet j est effectuée. On somme donc sur i et j le paramètre $dist_{ij}$ multiplié par la variable x_{ij} pour connaitre la distance totale parcourue. De plus, nous sommes contraints à des fenêtres de temps. S'il n'y a pas de dépassement, k_i vaudra b_i et la différence des deux donnera 0. Dès lors que celles-ci sont dépassées, une pénalité est à prendre en compte. La variable k_i sera alors égale à t_i, et la différence avec b_i nous donnera le temps de dépassement de la fenêtre de temps. On multiplie cela par le taux de pénalité $penalite_i$. Pour comptabiliser l'ensemble des pénalités, on somme ces valeurs sur i allant de 2 à n, puisqu'on ne prend pas en compte le dépôt.

2. `Contrainte 1` : Contrôle que chaque sommet j soit visité une seule et unique fois à partir d'un sommet i.

3. `Contrainte 2` : Contrôle que lorsqu'on visite un sommet j à partir d'un sommet i, qu'il soit quitté pour visiter un autre sommet i.

4. `Contrainte 3` : Contrôle qu'un sommet j ne soit pas visité avant la borne inférieure de la fenêtre de temps a_j.

5. `Contrainte 4` : Si le chemin du sommet i vers j est effectué, on force t_j à prendre l'instant d'arrivée au sommet i plus la distance entre i et j. À ce moment, si t_j est inférieur à a_j, la contrainte 3 l'obligera à prendre la valeur a_j.

6. `Contrainte 5` : Afin de prendre en compte les pénalités de violation des fenêtres de temps, si t_i est supérieure à b_i, on force y_i à prendre la valeur 1 à l'aide du big M.

7. `Contrainte 6` : Au contraire, si t_i est inferieur ou égale à b_i, on force y_i à prendre 0.

8. `Contrainte 7` : La variable k_i prend la valeur de t_i.

9. `Contrainte 8` : Si la fenêtre de temps est respectée, y_i vaut 0 et b_i étant supérieur à t_i, la variable k_i prendra la valeur de b_i. En revanche, si elle n'est pas respectée, y_i vaut 1 et t_i étant supérieur à à 0, la variable k_i prendra la valeur de t_i sous la contrainte 7.

3 Méthode de résolution

3.1 Formulation du modèle mathématique de l'heuristique

Au regard de la NP-Complétude du TSPTW décrit ci-précédemment l'approche heuristique de notre problème présente beaucoup d'avantages. Nous pouvons ainsi espérer obtenir des solutions de très bonne qualité en un temps très faible et sur des instances très larges. Au cours de l'UE de CL10 nous avons eu l'occasion de découvrir l'algorithme génétique, un algorithme évolutionnaire qui lorsque bien codé converge vers de bonnes solutions. La structure initiale de ce dernier est un des acquis de l'UE CL10 et se trouve extrêmement bien documentée sur Moodle et partout sur le Web.

Toutefois, afin d'obtenir un algorithme de qualité il convient d'affiner la structure de ce dernier et notamment les paramètres utilisés. En qualité d'étudiant en génie industriel nous avons ainsi décidé d'opter pour certaines stratégies. Nous avons fait des choix quant à la structure de notre algorithme, nous souhaitons au travers de ce rapport décrire les choix qui ont été fait et décrire plus en détails les raisons qui les ont motivés.

3.2 Population initiale

La population initiale est une étape très importante dans la réalisation d'un algorithme génétique. Si cette dernière est trop uniforme, ou trop influencée par une typologie d'individus donnés il est possible qu'elle biaise par avance l'algorithme en l'orientant vers des solutions biaisées à leur tours. L'unicité d'un pattern est donc à fuir. De plus, l'absence totale de toute typologie (le hasard pur) n'est en soit pas un problème mais peut éventuellement manquer de donner un coup de pouce à l'algorithme ce qui pour les grandes instances, n'est pas un luxe. Ainsi la règle d'or est la diversité, toutefois un brin d'intelligence dans certains individus, si tant est qu'on évite la redondance, est la bienvenue.

Il est ainsi courant de proposer des versions aléatoires des heuristiques connues pour un problème et d'en faire des individus. Nos quelques lignes de codes ci-dessous présentent notre étape d'initialisation dans l'algorithme :

```
For j = 1 To m  'Population of m initial solution

    RndChoice = Int(100 * Rnd) + 1
    If RndChoice < 40 Then
        Call InitRandom(InitSolution)
    Else
        If RndChoice > 70 Then
            Call Init_Best_Insertion_Randomized(InitSolution)
        Else
            Call Init(InitSolution)
        End If
    End If
```

Le code reflète notre choix de conserver 40% de la population comme formée de manière aléatoire, ce si large choix se justifie par un désir de conserver une grande diversité pour explorer une grande partie du champ des possibles, nous ne craignons pas l'absence de bons résultats puisque nous savons pour nos instances être en mesure de faire tourner l'algorithme un grand nombre d'itérations et ceux

sur des grandes populations. Parmi les 60% restant, la moitié soit 30% seront formés selon la fonction : $Init\_Best\_Instertion\_Randomized()$ cette fonction constructive donne de très bons résultats. Comme son nom le suggère, cette dernière choisira aléatoirement un noeud (chromosome) et l'insérera dans la meilleure position qui soit dans l'état d'avancement actuelle de la création de l'individu.

Mais que deviennent les 30% restant, quel mystère se cache derrière la fonction $Init$? À l'initiale l'heuristique randomizée par notre professeur était celle du $PlusProcheVoisin$. Nous résolvions à ce moment-là le TSP ou VRP uniquement axé sur la distance. Dans le cadre de ce projet, comme les parties précédentes le suggèrent, le retard est aussi compris dans la fonction objectif, ainsi selon la valeur que prend alpha ce dernier peut jouer un rôle très important dans la qualité de nos solutions. Dans une perspective de généralisation nous avons souhaité donner à notre algorithme la capacité d'être performant sur tout type de dataset, certains seront complexes de part leurs distances, d'autres par leurs fenêtres de temps où les deux. D'un dataset à l'autre la structure varie et la typologie de solutions aussi. C'est alors que nous est venu l'idée suivante : nous allons conserver l'heuristique du *Nearest Neighbor Randomized* mais nous allons là dupliquer, nous avons proposé une version qui base son critère de proximité non plus sur le critère de la distance mais sur celui de la date butoir, une forme de distance temporelle. La dernière version se basera elle sur la distance plus alpha fois la date butoir, reflétant ainsi plus la fitness function cette dernière se présente comme une conciliation raisonnable des deux critères ci précédents. Ainsi comme vous pouvez le voir la fonction $Init()$ va construire des solutions sur ces trois versions selon les probabilités respectives suivantes : 0.3, 0.3, 0.4.

```
InitSolution(1) = 1
RndChoice = Int(100 * Rnd) + 1
If RndChoice < 40 Then
    For i = 1 To n - 1
        InitSolution(i + 1) = randomized_two_closest_distalphaDD(i, visited)
        visited(InitSolution(i + 1)) = True
    Next i
Else
    If RndChoice > 70 Then
        For i = 1 To n - 1
            InitSolution(i + 1) = randomized_two_closest_dist(i, visited)
            visited(InitSolution(i + 1)) = True
        Next i
    Else
        For i = 1 To n - 1
            InitSolution(i + 1) = randomized_two_closest_DD(i, visited)
            visited(InitSolution(i + 1)) = True
        Next i
    End If
End If
```

Nous sommes fières de pouvoir conclure cette partie par le fait que l'onglet $FIRST\_POP$ de notre projet présente pour chaque instance et paramètres une population initiale d'une grande diversité, d'une qualité plutôt bonne et ceux sans répétition d'individus.

3.3 Anti-convergence prématurée

La convergence prématurée est la bête noire de l'algorithme génétique, l'excès d'élitisme peut conduire à rapidement converger vers une solution à sous prétexte qu'elle est bonne par rapport aux autres solutions des premières populations qui, rappelons-le, sont diversifiés et de qualité moyenne. À ce titre, bien que sélectionnant une bonne solution dans ce panel, l'algorithme ne se donnera pas les moyens d'explorer les autres typologies d'individus et de gènes intéressants et cela peut sérieusement affecter la qualité des solutions finales. L'une des premières solutions proposées est la mutation. Au cours des itérations chaque individu de la population a une probabilité *proba_mut* de voir son code génétique muter. Nous souhaitons rappeler que notre algorithme présente 3 types de mutation : l'échange, l'insertion et l'inversion, c'est trois mutations sont équiprobables dans notre algorithme.

Notre choix a été de rendre cette probabilité de mutation dynamique. La première moitié de l'algorithme se fait avec $proba\_mut = 0.05$. Cette probabilité est justement bien dosée et s'accorde avec la recherche scientifique, pour ne pas empêcher la convergence sans non plus écrémer trop rapidement la diversité tant recherchée de la population initiale. La seconde partie de l'algorithme fait évoluer cette probabilité comme suit :

```
'Increasing Mutation Rate To Avoid Early Convergence'

If Iteration > Trial - (Trial / 10) Then
    proba_mut = 0.4
Else
    If Iteration > Trial / (1.3) Then
        proba_mut = 0.2
    Else
        If Iteration > Trial / 2 Then
            proba_mut = 0.1
        End If
    End If
  End If
```

En effet de 50 à 77% la probabilité est doublée et donne 0.1, de 77 à 90% elle double à nouveau pour donner 0.2 et enfin terminer à 0.4 pour les dernières itérations. En effet, plus l'algorithme avance dans les iterations plus la qualité de sa population est bonne et plus il convient de perturber cette solution afin de sortir d'un éventuel optimum locale. En plus de cette étape de perturbation croissante il existe de nombreux principes d'anti-élitisme.

L'algorithme du recuit simulé est dans sa structure anti-élitiste et c'est en s'inspirant de ce dernier et de sa probabilité d'accepter une solution non améliorante que nous avons fait le choix suivant. La partie de l'algorithme génétique la plus à même de provoquer la convergence prématurée est la phase de sélection. Pour éviter cela le processus de choix des individus à sélectionner se base sur un process de *Slection par tournois*. Le principe est simple, on sélectionne aléatoirement deux individus dans la population et "le meilleur gagne". Notre choix a été de modifier légèrement cette fonction et forcer le fait que dans 5% des cas c'est en fait le "moins bon" que nous allons faire gagner et donc sélectionner, cela permet de ne pas pleinement orienter la sélection vers une population totalement d'élite ou du moins trop rapidement

```
TBS1 = TournamentSelection(Population, Cost, SizeNextGen)
TBS2 = TBS1
While TBS2 = TBS1
    TBS2 = TournamentSelection(Population, Cost, SizeNextGen)
Wend
RndChoice = Int(100 * Rnd) + 1
If Cost(TBS1) <= Cost(TBS2) Then
    If RndChoice < 5 Then
        Best = TBS2
    Else
        Best = TBS1
    End If
Else
    If RndChoice < 5 Then
        Best = TBS1
    Else
        Best = TBS2
    End If
End If
```

3.4 Recherche locale

L'étape de mutation d'un algorithme génétique a vocation à perturber la population mais ne présente toutefois aucune forme d'intelligence. En recherche opérationnelle la recherche locale est une étape importante, certaines heuristiques basent même leur fonctionnement sur cette dernière. Cela consiste à explorer le voisinage d'une solution via des mouvements améliorants ou non. Pour le problème tu TSP la recherche locale par $2-Opt$ est très répandue. Elle consiste à intervertir l'ordre de visite de deux clients et ainsi inverser la chaîne de noeud entre ces deux clients. Le mouvement choisi peut ensuite être le meilleur mouvement améliorant ou le premier.

Dans le cas d'un TSP classique la fonction est simple et l'évaluation d'un mouvement aussi, il ne s'agit que de comptabiliser la différence de distance entre deux jeux de deux arcs. Dans le cas du TSPTW les time-windows requierent une fois l'inversion faite de mesurer le retard causé ou non sur tous les noeuds en aval. Cela rend la fonction beaucoup plus complexe d'un point de vue algorithmique puisque l'évaluation des gains de chaque mouvement implique un parcours de matrice et un calcul très allongé. Nous avons toutefois codé la fonction $2-Opt$ pour le TSPTW en faisant délibérément le choix de la faire selon le meilleur mouvement améliorant. Une version "premier mouvement améliorant" est alors une ouverture possible pour notre projet. La structure de notre fonction est dans les grandes lignes comme suit (Nous avons fait quelques erreurs de syntaxe volontaires afin de simplifier la lecture du code) :

```
'Current Time_Visit calculation for each node
For k = 2 To n
    TimeVisit(InitSolution(k)) = TimeVisit(InitSolution(k - 1)) + t(InitSolution(k -
    1), InitSolution(k))
    'We wait if we arrive before early date of the time window
    If TimeVisit(InitSolution(k)) < SD(InitSolution(k)) Then
        TimeVisit(InitSolution(k)) = SD(InitSolution(k))
Next k
```

```
Do
    Opt = Opt + 1, Max_Improve = 0
    For i = 1 To n - 2 'For all combination
        a = InitSolution(i), b = InitSolution(i + 1)
        'Swap plus or mines value calculation for all nodes j (c) scheduled later
        For j = i + 2 To n - 1
            c = InitSolution(j), dback = InitSolution(j + 1)
            'Distance calculation and plus or mines distance calculation
            d_ab_cd = d(a, b) + d(c, dback), d_ac_bd = d(a, c) + d(b, dback)
            Delta = d_ac_bd - d_ab_cd
            'Current Lateness calculation (Lateness1)
            For k = i + 1 To n
                 'Lateness penalization
                If (TimeVisit(InitSolution(k)) - DD(InitSolution(k))) > 0 Then
                    Lateness1 = Lateness1 + alpha * (TimeVisit(InitSolution(k)) - DD(
    InitSolution(k)))

            'We do not forget the depot in the calculation
            [. . .]
            'Lateness calculation in case of a and c swap move (Lateness2)
            For k = j To i + 1 Step -1
                'Time implementation And Lateness penalization
                'Same process ase before [. . .]

            'Dont forget to calculate Lateness even after the d point
            For k = j + 1 To n
            [. . .]
            'Plus or minus LATENESS calculation
            Deltatime = Lateness2 - Lateness1
            'Overall delta calculation
            Delta = Delta + Deltatime
            'Research of the maximum improvement move
            If -Delta > Max_Improve Then
                [. . .]

    'If we found a move that can make the objective function improve then :
    If Max_Improve > 0 Then
        'We make the move
        [. . .]

'We repeat this process if we can still improve but not more than 10 times
Loop While Max_Improve > 0 And Opt < 10
```

Cette fonction de recherche locale une fois codée doit ensuite être appelée par l'algorithme. Où ? Comment ? Combien de fois ? Sont autant de questions auxquels nous avons dû répondre, il s'agit là de nouveau de choix personnels. Nous sommes partie du constat que notre algorithme donnait d'ores et déjà d'extrêmement bonnes solutions sans cette fonction sur les instances testées. Or sachant que la complexité du $2-Opt$ est bien plus grande que celles de nos trois fonctions de mutation nous avons délibérément choisi de sortir la $2-Opt$ de l'algorithme génétique et de ne l'appeler qu'à la fin en guise de "dernier recours améliorant" cela nous permet ainsi de garder un algorithme performant au temps de résolution plus que raisonnable.

De toute évidence il convient de modifier tous les choix présentés ci précédemment en corrélation avec les performances de l'algorithme sur une instance donnée. De nouveau notre algorithme fonctionne bien en l'état mais pour d'autres types d'instance en terme de taille et typologie il conviendrait d'adapter sa structure.

4 Résultats

Dans cette section, nous allons présenter les différents résultats obtenus grâce au modèle mathématique et à l'heuristique précédemment présentés. Nous allons dans un premier temps exposer les résultats obtenus à l'aide des différentes méthodes de résolution. Puis dans une seconde partie nous allons comparer les deux méthodes et déterminer laquelle est la plus performante.

4.1 Présentation

Pour la partie qui suit, nous allons afficher les résultats de notre modèle mathématique obtenus via GUSEK et l'heuristique. Tous nos tests ont été effectués sur un ordinateur sous Windows 10 avec 8 Go de RAM et un processeur Intel Core i5 à 1.6 GHz. Pour la partie programmation de l'algorithme génétique, nous avons choisi le language VBA.

Notre dataset est constitué des instances qui nous ont été communiquées par notre professeur. Les caractéristiques sont résumées dans le tableau ci-dessous :

Paramètres	Dataset
Nombre de clients N	40
Coordonnées (CoX, CoY)	(1,50)
Fenêtre de temps $[a_i, b_i]$	[50,1000]
Capacité $Capa$	INF
Pénalité $penalite$	0.5

Nous avons choisi de tester notre modèle mathématique avec 15 clients au lieu de 40 car GUSEK est incapable d'afficher une solution avec le dataset complet. Cependant, nous avons pu tester notre algorithme génétique avec le dataset complet.
Dans les parties qui vont suivre nous allons présenter les différents résultats obtenus.

4.1.1 Modèle optimal

L'objectif de notre modèle est de déterminer un plus court chemin qui visite chaque client et qui termine par le dépôt, tout en respectant des fenêtres de temps. Notre fonction objectif est formulée comme suivant : $Minimize \sum_{i=1}^{n} \sum_{j=1}^{n} dist_{ij} * x_{ij} + \sum_{i=2}^{n} penalite_i * (k_i - b_i)$. Dans notre cas on considére que la capacité du véhicule est supérieure à la demande, nous n'avons donc pas de contrainte sur la capacité. Pour 15 clients on obtient les résultats ci-dessous :

```
INTEGER OPTIMAL SOLUTION FOUND
Time used:   172.5 secs
Memory used: 41.5 Mb (43558689 bytes)

La solution optimale trouvée pour ce VRPTW est de 280.860294494923 Km

La tournée trouvée et les temps de visites sont donnés ci-dessous:

1 ->3, t[3]=300
2 ->5, t[5]=773.345235059853
3 ->12, t[12]=350
4 ->10, t[10]=700
5 ->15, t[15]=825
6 ->7, t[7]=563.453624047077
7 ->4, t[4]=650
8 ->9, t[9]=854.752923956694
9 ->13, t[13]=875
10 ->2, t[2]=750
11 ->6, t[6]=550
12 ->14, t[14]=373.021728866443
13 ->1, t[1]=0
14 ->11, t[11]=541.514718625767
15 ->8, t[8]=833.944271910004
```

FIGURE 1 – Résultats du modèle optimal obtenu via GUSEK

Ainsi pour ce modèle avec 15 clients on obtient une distance de 280 km avec un temps d'exécution de 172.5 secondes. Le temps de visite de chaque client t[j] est affiché. On constate que tous les clients sont visités à temps, nous n'avons donc pas de pénalité. Cependant, le temps de visite du dépôt, t[1], n'est pas mis à jour.

Par ailleurs, comme dit précédemment, le temps de calcul étant très long nous n'avons pas pu calculer pour un nombre plus grand de clients. Pour proposer un résultat satisfaisant en un temps d'exécution raisonnable nous avons alors testé notre modèle heuristique.

4.1.2 Modèle heuristique : Algorithme génétique

Une fois notre modèle testé sur GUSEK, dans une optique de gain de performance et de temps d'exécution, il est assez courant de programmer le modèle heuristique à l'aide d'un langage informatique. Nous avons donc décidé de programmer une heuristique sous VBA. Ceci dit, l'algorithme génétique nous a permis de tester le dataset au complet.

La solution finale obtenue par l'heuristique correspond ici à une minimisation de la distance parcourue pour visiter tous les clients, tout en tenant en compte des fenêtres de temps.

Dans la suite nous allons vous présenter les résultats obtenus avec un nombre de clients plus grand. On précise que par défaut de temps, nous n'avons pas intégré le SPLIT dans notre algorithme génétique. Ainsi, on ne peut prendre en compte la capacité du véhicule. Le résultat que nous obtenons sera donc une tournée dite géante qui respecte les fenêtres de temps.

Après plusieurs exécutions pour stabiliser le résultat, on obtient une distance d'environ 472 km. L'onglet "MEMORY" de notre projet reflète qu'à ce jour notre meilleur résultat pour les 40 clients est 464.96. Le temps de résolution est de 19 secondes. On peut constater que ce temps est inférieur à celui obtenu via GUSEK pour seulement 15 clients. Le seul problème qu'on rencontre ici est que nous ne pouvons pas vérifier la tournée obtenue. En contraste avec le modèle sur GUSEK, l'algorithme génétique nous permet de résoudre des problèmes de grande taille avec un temps d'exécution acceptable.

4.2 Analyse numérique

Afin de comparer les deux méthodes de résolution nous avons fait des analyses de résultats en fonction du nombre de clients N qui doit être visité par notre véhicule. Nous comparons le temps de résolution et le résultat obtenu par chaque méthode. Notons que pour la suite de cette section nous considérons que la capacité du véhicule est infinie.

Pour cela, nous utilisons pour les deux méthodes les mêmes données et paramètres provenant du dataset décrit précédemment. Nous calculons le Gap Résulat et le Gap Time en poucentage, entre ces deux méthodes de la manière suivante :

$$Gap\% = \frac{(AG - GUSEK)}{(GUSEK)}$$

$$GapTime\% = \frac{(GUSEKTime - AGTime)}{(AGTime)}$$

N	GUSEK	AG	Gap Resulat%	GUSEK TIME (s)	AG TIME (s)	Gap Time%
15	280,86029	280,86029	0	172,5	5	33,5
14	267,51427	267,51427	0	15,2	5	2,04
13	224,00332	224,00332	0	2,9	5	-0,42
12	223,63285	223,63285	0	1,9	4	-0,52
11	208,07817	208,07817	0	0,9	4	-0,77
10	196,76770	196,76770	0	0,5	3	-0,83
9	177,88156	177,88156	0	0,1	3	-0,96
8	177,75893	177,75893	0	0,1	2	-0,95

TABLE 1 – Evolution du temps de résolution et du résultat pour les deux méthodes en fonction de N

Il est à noter que, le grand avantage que présente l'heuristique face à la programmation linéaire est le temps d'exécution. En effet, grâce à l'algorithme on a pu diviser le temps d'exécution par 30. La table ci-dessus nous montre que les résultats trouvés par les deux méthodes pour différents N testés, sont identiques. On peut remarquer que le modèle via GUSEK est plus rapide que l'heuristique pour un nombre de clients inférieur ou égal à 13. Nous supposons sereinement que c'est uniquement parce que nous avons délibérément décidé de laisser l'algorithme génétique tourner 600 fois pour une population

de 100 individus afin d'assurer sur chaque instance la meilleure qualité de solution. Toutefois, dès que ce nombre est dépassé on constate que l'algorithme génétique est nettement plus rapide. Pour N=15, on obtient un résultat de 280 km pour un temps d'exécution de 172.5 secondes via GUSEK contre 5 secondes avec l'AG.

Naturellement, nous souhaitons souligner le réel avantage de l'heuristique pour les grandes instances. Le dataset raccourci sur lequel tous ces tests ont été effectués ne représente pas souvent la réalité. Il n'a été choisi que pour sa capacité à être exploité sur GUSEK. D'après notre bon sens et par analogie, notre code (en VBA) nous fournit des solutions correctes en un temps moyen d'environ 20 secondes, là où il faudrait probablement plus que des heures à un solveur pour trouver la solution optimale. Il est important à dire que c'est ce temps d'exécution qui fait de l'heuristique une solution exploitable et viable.

5 Conclusion

Ce travail nous a démontré que les capacités d'un solveur (ici GUSEK), sont limitées en terme de temps d'exécution. Nous obtenons des résultats satisfaisants pour des petits datasets. Mais la résolution est quasi impossible à partir d'un certain seuil car le temps d'exécution devient colossal. C'est dans ce contexte-là que l'algorithme génétique est très utile. Nous avons pu obtenir de bons résultats pour un dataset complet en un temps quasiment négligeable. Cependant, nos deux méthodes de résolution ne prennent pas en considération la capacité du véhicule. Dans une démarche d'amélioration nous pourrons intégrer la procédure SPLIT à notre algorithme génétique afin d'obtenir des sous-tournées réalisables. Cela nous permettra d'avoir des résultats proches de la réalité. En effet, la tournée géante qu'on obtient pour 40 clients n'est pas applicable dans tous les cas industriels.

Aussi, l'algorithme génétique peut aisément se glisser dans la case des algorithmes de Machine Learning. Si nous devions proposer une amélioration à notre projet nous souhaitons aborder un point à ce sujet. Très récemment le sujet du Auto-ML fait énormément parler de lui dans le monde de la recherche. Nous le savons, pour améliorer l'algorithme génétique il faut principalement adapter ses paramètres et les combinaisons sont nombreuses, on pourrait y voir là un tout autre problème. L'une des manières de le faire serait de faire des études statistiques sur différentes instances et différents paramètres. Quoi de mieux pour faire des statistiques que du machine learning ? En effet le Auto-ML est un choix des chercheurs qui consisterait à faire du machine learning dans du machine learning. Cette mise en abyme revient à laisser à un algorithme le choix du modèle et des paramètres les plus appropriés pour la résolution d'un problème donné. Cette double couche d'intelligence artificielle s'avère être extrêmement prometteuse et d'ores et déjà plus qu'efficace. On pense que cette aire de recherche conviendrait parfaitement pour trouver les paramètres optimaux de notre algorithme et ainsi le rendre très performant.