Optimisation du VRP

Rapport SAE 6.01

Groupe 2

I. Formulation mathématique

A. Modélisation Mathématique

1. Rappels des données

Le problème abordé dans ce sujet est basé sur le problème de tournées de véhicules (plus connu sous le nom de “Vehicle Routing Problem” ou “VRP” en anglais) . Pour répondre à ce problème, différentes données sont disponibles, il est important de rappeler celles que nous utilisons :

a) Les différentes données

- V, ensemble des véhicules.
- C, ensemble des clients.
- D, dépôt (et son id).
- $Dist_{ij}$, distance entre le nœud i et j.
- $demande_{i}$, demande du client i.
- $Qmax$, capacité maximum des véhicules
- N, ensemble des nœuds, C ∪ {D}.

b) Les variables de décision

- x_{ij}^{v}, tableau des chemins pris ou pas.
- $nbreVehicules$, Nombre de véhicules utilisés
- $Qaprestour$, Capacité restante du véhicule v au retour au dépôt
- Q_{iv}, Capacité du véhicule v au nœud i
- u, variable additionnelle pour la méthode MTZ d’élimination des sous tours

2. Fonction objectif

$$Z = \sum_{v \in V} \sum_{i \in N} \sum_{j \in C} (Dist_{ij} * x_{ij}^{v})$$

Ici, nous cherchons à minimiser la somme des distances des trajets (i et j) de chaque véhicule (v). Pour cela, il suffit donc de calculer la somme des trajets ($\sum_{v \in V} \sum_{i \in N} \sum_{j \in C} (Dist_{ij} * x_{ij}^{v})$) réalisés pour chacun des véhicules.

3. Contraintes :

Notre solution doit prendre en compte les contraintes ci-dessous représentées par leurs modélisations mathématiques:

1. Un véhicule qui quitte le dépôt doit retourner au dépôt à la fin de sa tournée.

$$\forall v \in V, \sum_{i \epsilon C} x_{iD}^{v} = \sum_{j \epsilon C} x_{Dj}^{v}$$

2. Un client doit être visité une et une seule fois par un véhicule.

$$\forall\, i \epsilon C, \sum_{v \epsilon V} \sum_{j \epsilon N,\, j \neq i} (x_{ij}^{v}) = 1$$

3. Au moins un véhicule est utilisé pour la construction des tournées.

$$\sum_{v \epsilon V} \sum_{j \epsilon C} x_{Dj}^{v} >= 1$$

4. L'élimination des sous-tours grâce à la méthode MTZ, c'est-à-dire éviter les boucles et obtenir un circuit hamiltonien.

$$\forall v \in V,\ \forall i \in C, \forall j \in C,\ u_{j}^{v} >= u_{i}^{v} + demande_{j} - Qmax \times (1 - x_{ij}^{v})$$

5. La capacité du véhicule ne peut pas être dépassée.

$$\forall v \in V, \sum_{i \epsilon C} \sum_{j \epsilon N} demande_{i} \times x_{ij}^{v} \leq Qmax$$

B. CPLEX

1. Fonction objective

Ci-dessous est l'implémentation CPLEX de notre modélisation de la fonction objectif à minimiser.

```
// Fonction objectif : Minimiser la distance totale
minimize sum(v in Vehicules, i in Noeuds, j in Noeuds : i != j) Distance[i][j] * x[i][j][v];
```

2. Contraintes

Voici ci-dessous nos implémentations du CPLEX qui représentent nos contraintes..

1. Un véhicule qui quitte le dépôt doit retourner au dépôt à la fin de sa tournée

```
// Chaque véhicule doit revenir au dépôt s'il sort
forall(v in Vehicules, i in Noeuds : i != idDepot) {
  sum(j in Noeuds : j != i) x[j][i][v] == sum(j in Noeuds : j != i) x[i][j][v];
 }
```

2. Un client doit être visité une et une seule fois par un véhicule

```
// Tous les clients doivent être visités une fois
forall(i in Noeuds : i != idDepot)
  sum(v in Vehicules, j in Noeuds : j != i) x[i][j][v] == 1;
```

3. Au moins un véhicule est utilisé pour la construction des tournées

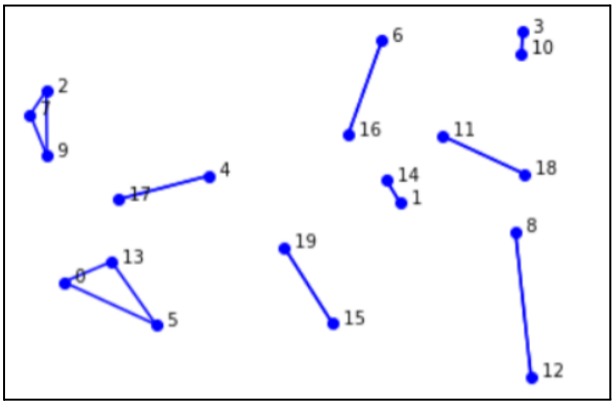
```
// Au moins un véhicule utilisé pour les tournées
sum(v in Vehicules, j in Noeuds : j != idDepot) x[idDepot][j][v] >= 1;
```

4. Elimination des sous tours

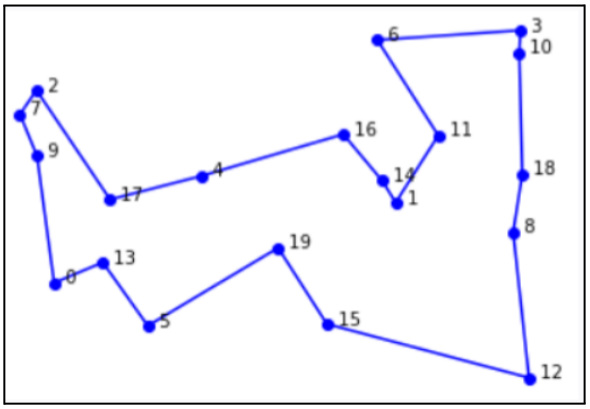
```
// Contraintes MTZ pour éviter les sous-tours
forall(v in Vehicules, i in Noeuds : i != idDepot, j in Noeuds : j != idDepot && i != j) {
  u[j][v] >= u[i][v] + Demande[j] - Qmax * (1 - x[i][j][v]);
}
```

Explication :
La méthode MTZ (Miller-Tucker-Zemlin) permet d'éliminer les sous tours afin d'obtenir un circuit hamiltonien (chaque point est parcouru une seule fois en un seul parcours).

Pour cela on utilise la variable `u[i][v]` qui représente l'ordre de visite d'un client `i` par un véhicule `v`.
La contrainte dit :
Si le véhicule va de `i` vers `j` (`x[i][j][v]=1`), alors `u[j]` doit être strictement plus grand que `u[i]` (on ajoute la demande de `j`).
Si l'arc n'est pas utilisé, la contrainte ne fait rien (grâce à `Qmax`).
Du coup, dans un sous-tour (un cycle sans dépôt), on serait obligé d'augmenter `u` en tournant en rond, ce qui est impossible.



Résolution avec sous tours



Résolution sans sous tours

5. La capacité du véhicule ne peut pas être dépassée.

```
// Capacité restante doit rester dans les limites [0, Qmax]
forall(v in Vehicules) {
  sum(i in Noeuds, j in Noeuds) Demande[i] * x[i][j][v] <= Qmax;
}
```

Nous avons également vérifié que la demande totale peut être satisfaite par la capacité totale des véhicules.

```
execute
{
  var totalDemande = 0;
  var totalCapacity = Qmax * nbVehicules;

  for (var i in Noeuds)
    {
      if (i != idDepot)
      {
       totalDemande += Demande[i];
      }
    }

    if (totalDemande > totalCapacity)
    {
      writeln("Aucune solution : La demande dépasse la capacité totale des véhicules");
      writeln("Détails :");
      writeln("Nombre de véhicules : " + nbVehicules);
      writeln("Capacité totale des véhicules " + totalCapacity);
      writeln("Demande totale : " + totalDemande);
    }
}
```

Voici donc le résultat obtenu en cas où l'on n'as pas assez de véhicules pour la demande :

```
Aucune solution : La demande dépasse la capacité totale des véhicules
Détails :
Nombre de véhicules : 2
Capacité totale des véhicules 200
Demande totale : 263
```

3. Exécution

Nous avons utilisé les logiciels et machines de l'iut.
Notre CPLEX met environ moins de 5 secondes avant de se résoudre.
Les fichiers .mod et .dat que nous avons écrits sont disponibles sur Github.

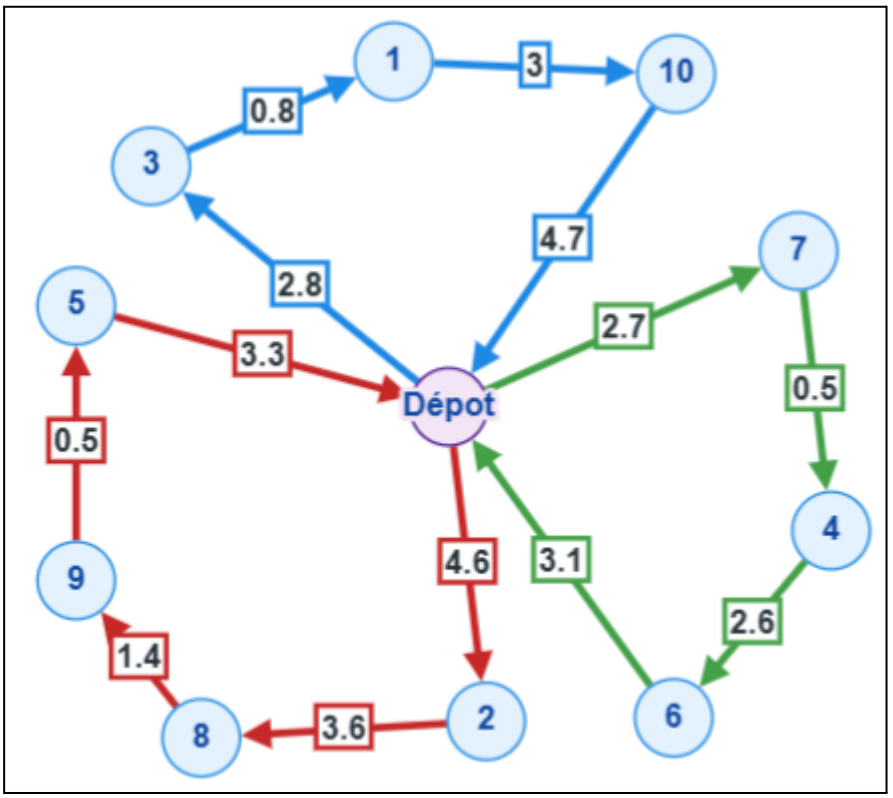
a) Solution optimale du problème des cantines

Nous avons exécuté notre programme CPLEX avec les données de la livraison des cantines, le résultat obtenu se trouve ci-dessous :

```
// solution (optimal) with objective 31.3
Résultats de l'optimisation:
==============================
 Véhicule 1:
 ------------

Ce véhicule n'a pas été utilisé
==============================
 Véhicule 2:
 ------------

  Tournée      : Dépôt -> 3 -> 1 -> 10 -> Dépôt
  Qté déposées : 100 -> 89 -> 29 -> 3 -> 3
  Distance parcourue : 04.7 -> 3 -> 0.8 -> 2.8
  Capacité utilisée : 97
==============================
 Véhicule 3:
 ------------

  Tournée      : Dépôt -> 2 -> 8 -> 9 -> 5 -> Dépôt
  Qté déposées : 100 -> 56 -> 29 -> 19 -> 1 -> 1
  Distance parcourue : 03.3 -> 0.5 -> 1.4 -> 1.3 -> 4.6
  Capacité utilisée : 99
==============================
 Véhicule 4:
 ------------

  Tournée      : Dépôt -> 7 -> 4 -> 6 -> Dépôt
  Qté déposées : 100 -> 68 -> 53 -> 33 -> 33
  Distance parcourue : 03.1 -> 2.6 -> 0.5 -> 2.7
  Capacité utilisée : 67

==============================
==============================
Statistiques globales :
  Nombre total de véhicules utilisés : 3 / 4
  Distance totale parcourue : 31.3
```

Notre programme CPLEX nous renvoie le résultat de 31.3 km. Nous pouvons noter qu'il n'utilise que 3 véhicules sur les 4 disponibles.

b) Graphe de la solution



Nous avons réalisé cette représentation graphique de notre solution CPLEX sur l'application web graphonline afin de mieux visualiser notre solution.
Nous avons donc trois véhicules, ici représentés en bleu, vert et rouge, qui partent tous du dépôt pour visiter une liste de client, une fois leur tournée de client fini ils retournent tous au dépôt.
La valeur sur chaque arête représente la distance entre les lieux, les arrêtés sont orientés pour indiquer le sens de circulation des véhicules.

C. Recuit simulé

1. Algo

Le recuit simulé est un algorithme d'optimisation probabiliste inspiré du phénomène physique de recuit en métallurgie. Il est principalement utilisé pour résoudre des problèmes de minimisation complexes, pour lesquels les méthodes classiques risquent de se bloquer dans des minima locaux (c'est-à-dire des solutions qui semblent optimales par rapport à leurs voisines) .

Son principe repose sur une exploration large de l'espace des solutions au début, puis d'un resserrement progressif vers les solutions les plus performantes.
On considère une solution SSS, qui représente un état possible du problème. Une solution initiale S0 est choisie de manière aléatoire. À chaque solution est associée une fonction objectif f(S)f(S)f(S), appelée fonction de coût, que l'on cherche à minimiser. Plus la valeur de cette fonction est faible, meilleure est la solution. L'algorithme manipule une solution courante qui évolue au fil des itérations.

À partir de la solution courante, une solution voisine est générée. Cette solution est obtenue par une modification locale de la solution courante, selon une règle de voisinage propre au problème étudié. Dans ce cas ci, cette modification consiste en une inversion, ou un changement du chemin de manière aléatoire On calcule ensuite la variation du coût entre les deux solutions à l'aide de la quantité Δf. Cette valeur permet de savoir si la solution voisine est meilleure ou moins bonne que la solution actuelle.

L'algorithme peut être résumé par le pseudo-code suivant :

```
Initialiser S_courant ← S_0
Initialiser T ← T_0

tant que T > T_min faire :
   Générer une solution voisine S_voisin
   Calculer Δf ← f(S_voisin) - f(S_courant)

   si Δf ≤ 0 alors
      S_courant ← S_voisin
   sinon
      p ← exp(-Δf / T)
      r ← nombre aléatoire dans [0,1]
      si r < p alors
         S_courant ← S_voisin
      fin si
   fin si

   T ← α × T
fin tant que

Retourner S_courant
```

La température T joue un rôle central dans l'algorithme. Elle est initialisée à une valeur élevée T0, ce qui permet d'accepter facilement des solutions moins bonnes au début de l'exécution. À chaque itération, la température est diminuée selon un facteur de refroidissement **α**, compris entre 0 et 1. Cette diminution progressive contrôle la transition entre exploration globale et recherche locale. Lorsque la solution voisine est meilleure que la solution courante, c'est-à-dire lorsque **Δ**f≤0, elle est toujours acceptée. En revanche, si la solution est moins bonne, elle peut tout de même être acceptée avec une probabilité.ce qui permet à l'algorithme de sortir de minima locaux en autorisant temporairement des dégradations de la fonction objectif, surtout lorsque la température est élevée.

À mesure que la température diminue, la probabilité d'accepter une solution moins bonne devient de plus en plus faible. L'exécution s'arrête lorsque la température devient inférieure à une valeur minimale prédéfinie ou lorsqu'un nombre maximal d'itérations est atteint. La solution obtenue est alors considérée comme une approximation satisfaisante de l'optimum global.

2. Java

L'implémentation du recuit simulé est réalisée dans la classe `RecuitSimuleCVRP`.

a) Initialisation

```
Solution actuelle = genererSolutionInitiale();
Solution meilleure = actuelle.copie();
double temperature = tempInit;
```

Une solution initiale respectant les contraintes est générée.
La température est initialisée avec la valeur donnée par l'utilisateur.

b) Boucle principale

```
while (temperature > seuilArret && iterationsSansAmelioration
< nbIttSansAmelioration)
```

Deux critères d'arrêt sont utilisés :

- Température minimale atteinte
- Nombre maximal d'itérations sans amélioration

c) Génération d'un voisin

```
Solution voisin = genererVoisin(actuelle);
```

Un mouvement aléatoire est choisi parmi :

- Relocate
- Swap
- 2-opt intra
- 2-opt inter
- Or-opt

d) Calcul du delta

```
double delta = voisin.getDistanceTotale() -
actuelle.getDistanceTotale();
```

e) Règle d'acceptation

```
if (delta <= 0 || Math.exp(-delta / temperature) >
random.nextDouble())
```

- Si amélioration → acceptation
- Sinon → acceptation probabiliste

f) Mise à jour de la meilleure solution

```
if (actuelle.getDistanceTotale() <
meilleure.getDistanceTotale())
```

g) Refroidissement

```
temperature = alpha * temperature;
```

h) Retour du résultat

Un objet `ResultatRecuit` est retourné contenant :

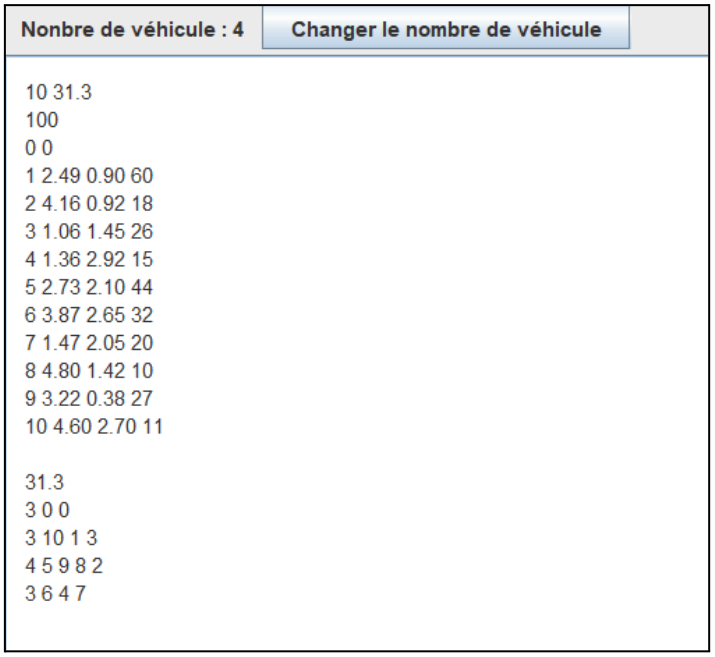
- meilleure solution
- snapshots
- temps d'exécution
- nombre d'itérations

Les snapshots sont pris toute les 5000 itération.

3. Cas pratique

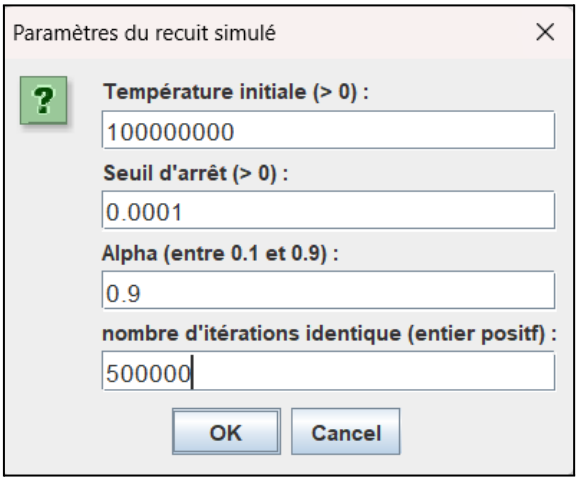
a) Livraison des cantines

Nous avons testé notre programme sur les données de cantine avec les 10 clients que nous avons écrit.



Voici ce que donne notre cantine.txt charger dans CENOFM.

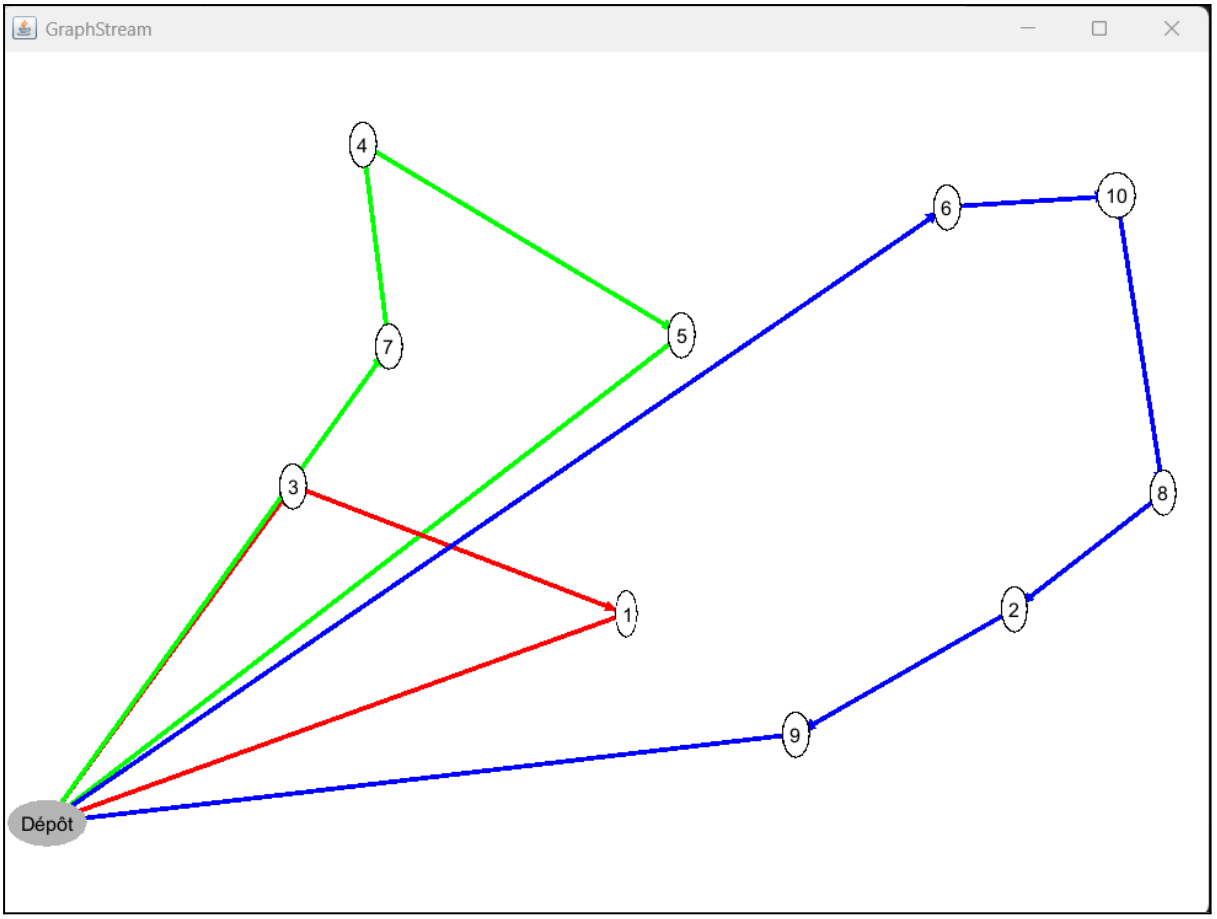
Nous avons choisi ces paramètres pour le recuit.



Et nous obtenons donc:

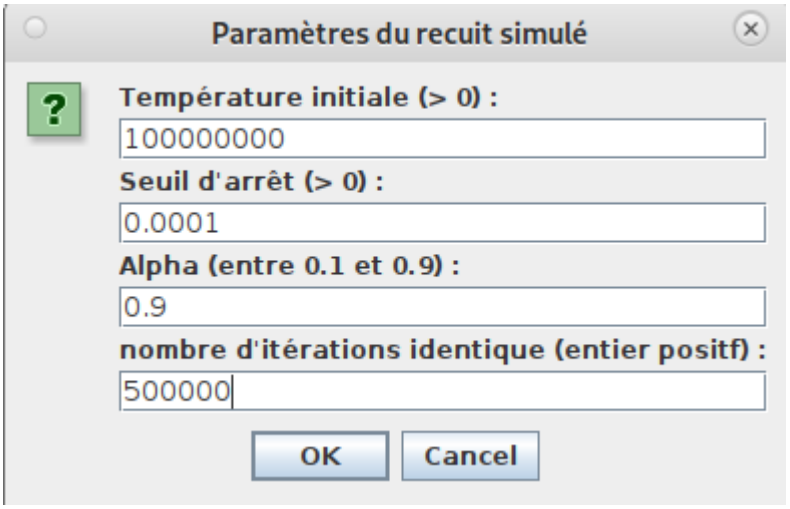
```
===== Solution Initiale =====
Distance totale : 32,94
Nombre de véhicules : 4
Véhicule 1 : Dépôt -> 1 -> 4 -> 7 -> Dépôt
Véhicule 2 : Dépôt -> 2 -> 8 -> 9 -> Dépôt
Véhicule 3 : Dépôt -> 3 -> Dépôt
Véhicule 4 : Dépôt -> 5 -> 6 -> 10 -> Dépôt

===== Solution à l'itération 50000 =====
Distance totale : 40,72
Nombre de véhicules : 3
Véhicule 1 : Dépôt -> 2 -> 1 -> 8 -> Dépôt
Véhicule 2 : Dépôt -> 5 -> 7 -> 10 -> Dépôt
Véhicule 3 : Dépôt -> 4 -> 3 -> 9 -> 6 -> Dépôt

===== Solution à l'itération 100000 =====
Distance totale : 35,30
Nombre de véhicules : 3
Véhicule 1 : Dépôt -> 1 -> 4 -> Dépôt
Véhicule 2 : Dépôt -> 2 -> 10 -> 7 -> 5 -> Dépôt
Véhicule 3 : Dépôt -> 3 -> 9 -> 6 -> 8 -> Dépôt

===== Solution à l'itération 150000 =====
Distance totale : 38,68
Nombre de véhicules : 3
Véhicule 1 : Dépôt -> 6 -> 10 -> 3 -> 2 -> Dépôt
Véhicule 2 : Dépôt -> 7 -> 1 -> Dépôt
Véhicule 3 : Dépôt -> 9 -> 5 -> 4 -> 8 -> Dépôt

===== Solution à l'itération 200000 =====
Distance totale : 37,41
Nombre de véhicules : 3
Véhicule 1 : Dépôt -> 2 -> 7 -> 6 -> 8 -> 4 -> Dépôt
Véhicule 2 : Dépôt -> 3 -> 1 -> Dépôt
Véhicule 3 : Dépôt -> 5 -> 9 -> 10 -> Dépôt

===== Solution à l'itération 250000 =====
Distance totale : 26,27
Nombre de véhicules : 3
Véhicule 1 : Dépôt -> 3 -> 1 -> Dépôt
Véhicule 2 : Dépôt -> 5 -> 4 -> 7 -> Dépôt
Véhicule 3 : Dépôt -> 6 -> 10 -> 8 -> 2 -> 9 -> Dépôt
```

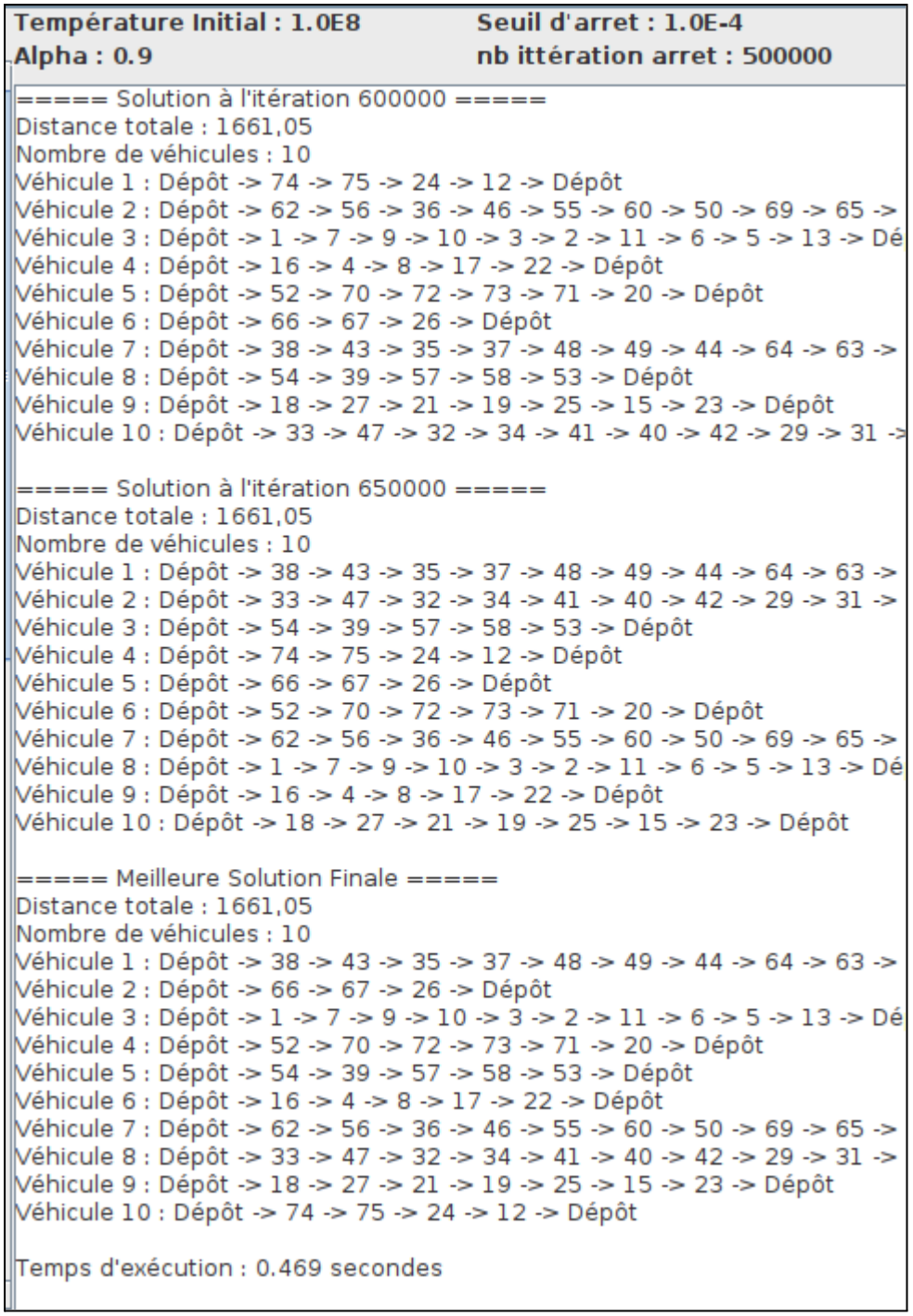
```
===== Solution à l'itération 300000 =====
Distance totale : 26,27
Nombre de véhicules : 3
Véhicule 1 : Dépôt -> 6 -> 10 -> 8 -> 2 -> 9 -> Dépôt
Véhicule 2 : Dépôt -> 5 -> 4 -> 7 -> Dépôt
Véhicule 3 : Dépôt -> 1 -> 3 -> Dépôt

===== Meilleure Solution Finale =====
Distance totale : 26,27
Nombre de véhicules : 3
Véhicule 1 : Dépôt -> 1 -> 3 -> Dépôt
Véhicule 2 : Dépôt -> 5 -> 4 -> 7 -> Dépôt
Véhicule 3 : Dépôt -> 6 -> 10 -> 8 -> 2 -> 9 -> Dépôt

Temps d'exécution : 0.106 secondes
```



b) Fichier tai75a

Le fichier sélectionné est “tai75a.txt” avec une solution optimale de 1618.352.





Température Initial : 1.0E8 **Seuil d'arret : 1.0E-4**
Alpha : 0.9 **nb ittération arret : 500000**

```
===== Solution à l'itération 600000 =====
Distance totale : 1661,05
Nombre de véhicules : 10
Véhicule 1 : Dépôt -> 74 -> 75 -> 24 -> 12 -> Dépôt
Véhicule 2 : Dépôt -> 62 -> 56 -> 36 -> 46 -> 55 -> 60 -> 50 -> 69 -> 65 ->
Véhicule 3 : Dépôt -> 1 -> 7 -> 9 -> 10 -> 3 -> 2 -> 11 -> 6 -> 5 -> 13 -> Dé
Véhicule 4 : Dépôt -> 16 -> 4 -> 8 -> 17 -> 22 -> Dépôt
Véhicule 5 : Dépôt -> 52 -> 70 -> 72 -> 73 -> 71 -> 20 -> Dépôt
Véhicule 6 : Dépôt -> 66 -> 67 -> 26 -> Dépôt
Véhicule 7 : Dépôt -> 38 -> 43 -> 35 -> 37 -> 48 -> 49 -> 44 -> 64 -> 63 ->
Véhicule 8 : Dépôt -> 54 -> 39 -> 57 -> 58 -> 53 -> Dépôt
Véhicule 9 : Dépôt -> 18 -> 27 -> 21 -> 19 -> 25 -> 15 -> 23 -> Dépôt
Véhicule 10 : Dépôt -> 33 -> 47 -> 32 -> 34 -> 41 -> 40 -> 42 -> 29 -> 31 ->

===== Solution à l'itération 650000 =====
Distance totale : 1661,05
Nombre de véhicules : 10
Véhicule 1 : Dépôt -> 38 -> 43 -> 35 -> 37 -> 48 -> 49 -> 44 -> 64 -> 63 ->
Véhicule 2 : Dépôt -> 33 -> 47 -> 32 -> 34 -> 41 -> 40 -> 42 -> 29 -> 31 ->
Véhicule 3 : Dépôt -> 54 -> 39 -> 57 -> 58 -> 53 -> Dépôt
Véhicule 4 : Dépôt -> 74 -> 75 -> 24 -> 12 -> Dépôt
Véhicule 5 : Dépôt -> 66 -> 67 -> 26 -> Dépôt
Véhicule 6 : Dépôt -> 52 -> 70 -> 72 -> 73 -> 71 -> 20 -> Dépôt
Véhicule 7 : Dépôt -> 62 -> 56 -> 36 -> 46 -> 55 -> 60 -> 50 -> 69 -> 65 ->
Véhicule 8 : Dépôt -> 1 -> 7 -> 9 -> 10 -> 3 -> 2 -> 11 -> 6 -> 5 -> 13 -> Dé
Véhicule 9 : Dépôt -> 16 -> 4 -> 8 -> 17 -> 22 -> Dépôt
Véhicule 10 : Dépôt -> 18 -> 27 -> 21 -> 19 -> 25 -> 15 -> 23 -> Dépôt

===== Meilleure Solution Finale =====
Distance totale : 1661,05
Nombre de véhicules : 10
Véhicule 1 : Dépôt -> 38 -> 43 -> 35 -> 37 -> 48 -> 49 -> 44 -> 64 -> 63 ->
Véhicule 2 : Dépôt -> 66 -> 67 -> 26 -> Dépôt
Véhicule 3 : Dépôt -> 1 -> 7 -> 9 -> 10 -> 3 -> 2 -> 11 -> 6 -> 5 -> 13 -> Dé
Véhicule 4 : Dépôt -> 52 -> 70 -> 72 -> 73 -> 71 -> 20 -> Dépôt
Véhicule 5 : Dépôt -> 54 -> 39 -> 57 -> 58 -> 53 -> Dépôt
Véhicule 6 : Dépôt -> 16 -> 4 -> 8 -> 17 -> 22 -> Dépôt
Véhicule 7 : Dépôt -> 62 -> 56 -> 36 -> 46 -> 55 -> 60 -> 50 -> 69 -> 65 ->
Véhicule 8 : Dépôt -> 33 -> 47 -> 32 -> 34 -> 41 -> 40 -> 42 -> 29 -> 31 ->
Véhicule 9 : Dépôt -> 18 -> 27 -> 21 -> 19 -> 25 -> 15 -> 23 -> Dépôt
Véhicule 10 : Dépôt -> 74 -> 75 -> 24 -> 12 -> Dépôt

Temps d'exécution : 0.469 secondes
```

Notre programme arrive à une solution de 1661,05.

II. Développement de l'application

A. Description de l'app

Notre application CENOFM (CPLEX Est Nul, On Fait Mieux) permet la conversion de fichier .txt vers des fichiers .dat afin d'être utilisé par CPLEX (principalement dans le but de rendre utilisable les fichiers Mistic). Elle permet également de résoudre des problèmes qui seraient insolvables par la version de CPLEX des machines de l'IUT grâce au recuit simulé.
Nous avons donc combiné sur une seule interface la conversion de fichier et la résolution par le recuit simulé.

B. Lancement de l'application

1. Langage choisi

Nous avons réalisé notre api en Java avec la version JDK 24 au minimum.

2. Lancement

Pour lancer l'application, il faut exécuter les scripts de démarrage (`setup.bat` pour Windows et `setup.sh` pour Linux).
Pour Linux :

- Executer `chmod u+x setup.sh` si c'est la première fois.
- Executer la commande `./setup.sh`

Pour Windows :

- Exécuter la commande `./setup.bat`

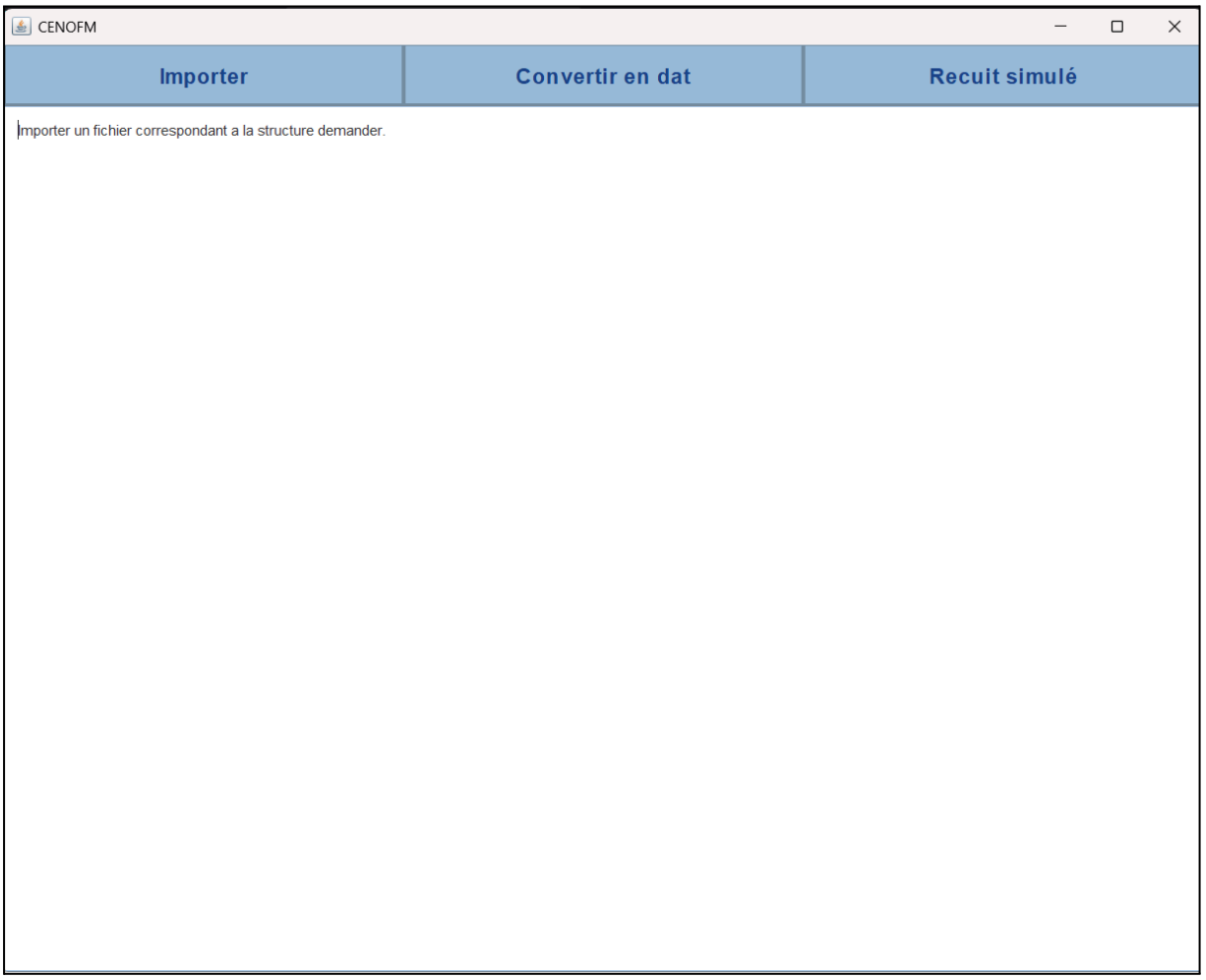
C. Import de fichier

Notre application doit pouvoir transformer des txt en .dat ainsi que résoudre un problème de VRP avec des données conséquentes. Nous avons choisi de pouvoir importer le fichier txt pour pouvoir en extraire les données, cela nous permet

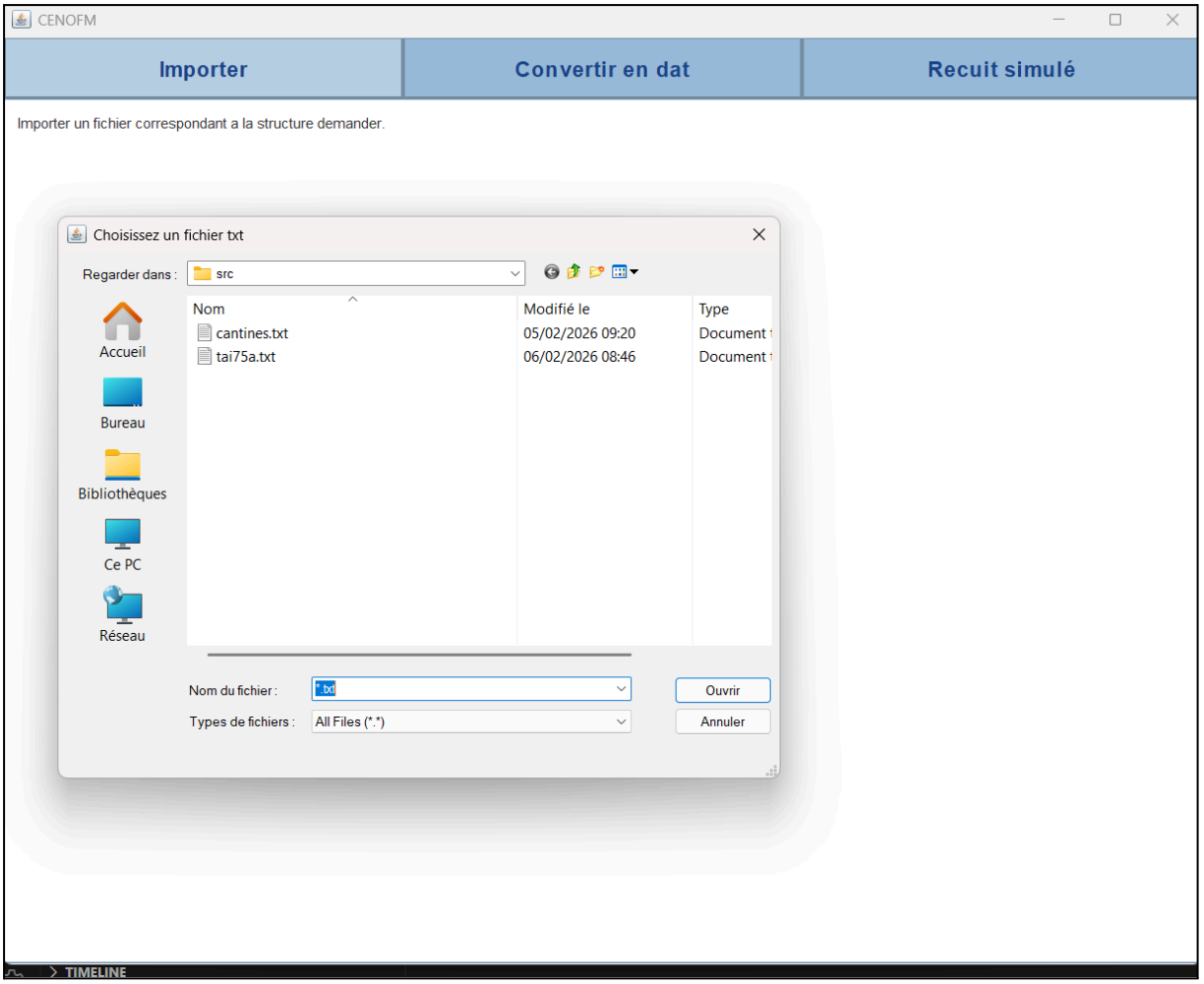
après de pouvoir convertir ce fichier en .dat ou de le résoudre sans parcourir le fichier a chaque fois.

1. Interface graphique

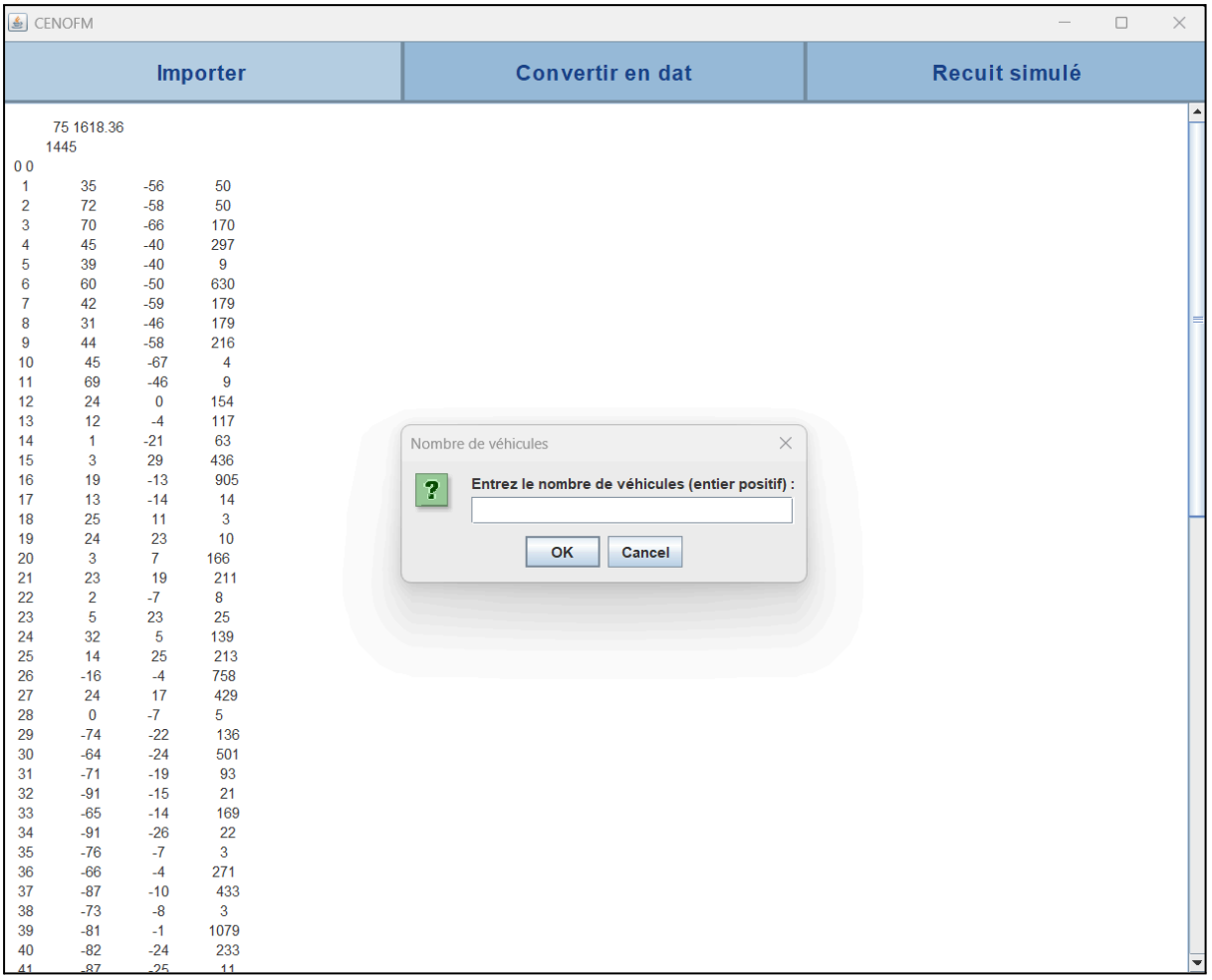
Une fois notre application démarrée, nous arrivons sur cette page. La première étape est donc d'importer un fichier.



Une fois le bouton “Importer” sélectionné, un explorateur de fichier apparaît, nous n’avons plus qu’à sélectionner un fichier mistic sous forme de txt.



Voici le fichier “tai75a.txt” ouvert et importé dans notre application, la dernière étape pour l’import de fichier est de renseigner le nombre de véhicule maximum pour les données chargées, cette information n’est pas renseigné dans les fichiers mistic et elle est nécessaire pour la conversion ainsi que le recuit simulé.



2. Code

a) Clic sur "Importer"

Dans `PanelImport.actionPerformed()` :

```
if ( e.getSource() == this.btnImporter ) {
     String cheminFichier =
this.selectionnerFichier("Choisissez un fichier txt");
```

➡ Une boîte de dialogue s’ouvre.

b) Lecture du fichier

```
try ( BufferedReader br = new BufferedReader(new
FileReader(cheminFichier)) ) {
    this.txtVrp.read(br, null);
    this.txtVrp.setCaretPosition(0);
```

Le contenu du fichier est affiché dans la zone de texte.

c) Demande du nombre de véhicules

```
NombreVehi();
this.lblNbVehi.setText("Nombre de véhicule : " +
this.nbVehicules);
```

Méthode appelée :

```
this.nbVehicules = Integer.parseInt(champ.getText().trim());
```

d) Vérification du format

```
if (verifierFormatFront(this.txtVrp.getText())) {
    this.frame.extractionDonnee(this.txtVrp.getText(),
this.nbVehicules);
}
```

Dans `verifierFormatFront()` :

```
String[] premiereLigne = lignes[0].trim().split("\\s+");
int nbClients = Integer.parseInt(premiereLigne[0]);
Double.parseDouble(premiereLigne[1]);
```

Vérification :

- nbClients
- bestSolution
- Qmax
- coordonnées dépôt
- clients

e) Extraction réelle des données

Dans `Controleur` :

```
public void extractionDonnee(String txt, int nbV) {
    this.donnee = this.lect.charger(txt, nbV);
}
```

Dans `LectureVrp` :

```
int nbClients = sc.nextInt();
double bestSolution = sc.nextDouble();
int qMax = sc.nextInt();
```

Puis :

```
donnees.setDepot(new Noeud(0, depotX, depotY, 0));
```

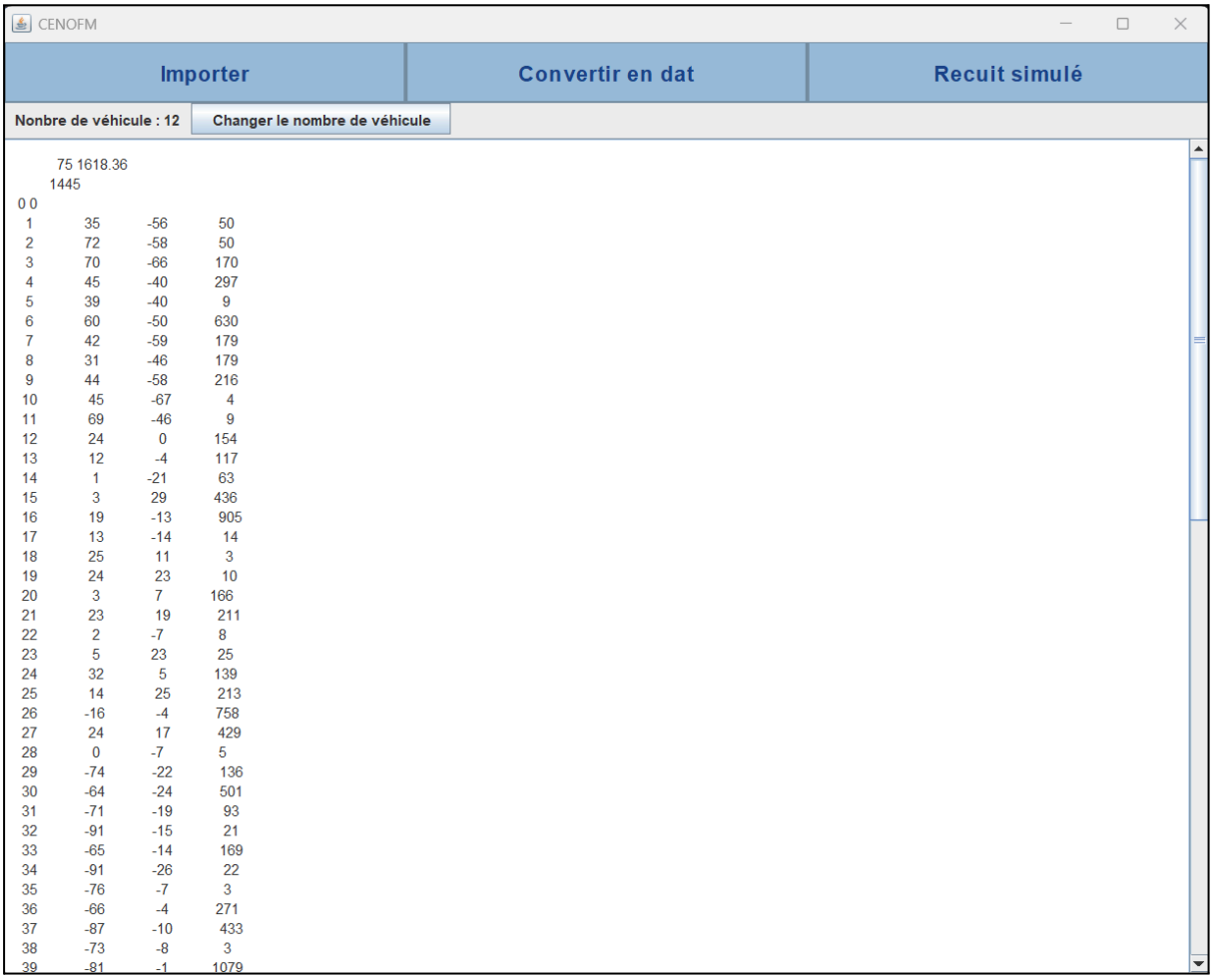
Et lecture des clients :

```
donnees.getClients().add(new Noeud(id, x, y, demande));
```

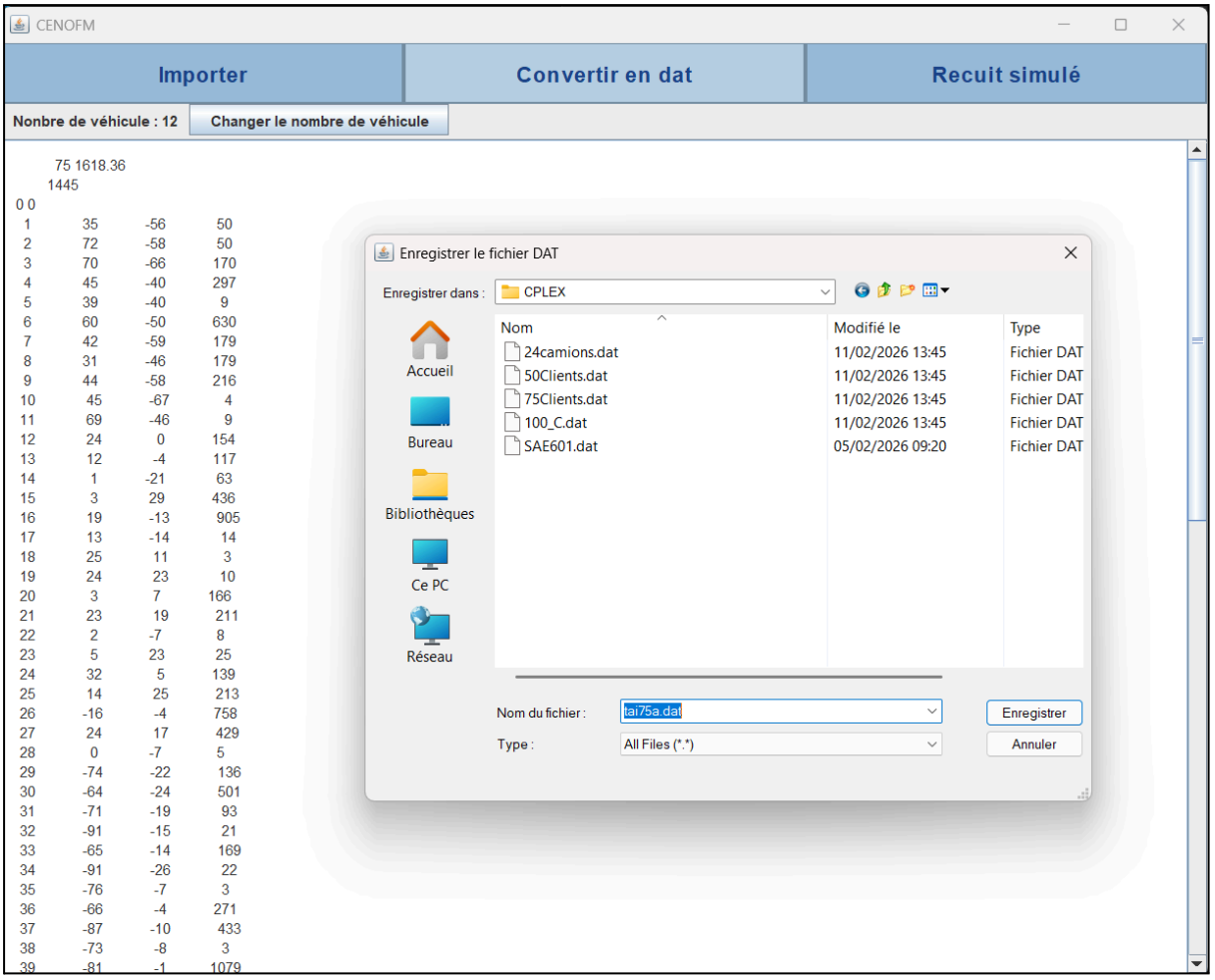
D. Conversion des fichiers .txt en .dat

1. Interface graphique

Pour la conversion en .dat il faut qu'un fichier .txt soit déjà importé



Une fois cette condition remplie, nous pouvons utiliser le bouton “Convertir en dat”. Un explorateur de fichier s’ouvre pour pouvoir enregistrer le nouveau fichier avec l'extension “.dat” déjà renseignée.



2. Code

a) Clic sur "Convertir en dat"

Dans `PanelImport` :

```
if ( e.getSource() == this.btnConvertir ) {
    String cheminSortie = enregistrerNouvFichier("Enregistrer
le fichier DAT", ".dat");
```

b) Appel contrôleur

```
this.frame.convertir(cheminSortie);
```

Dans `FrameMain` :

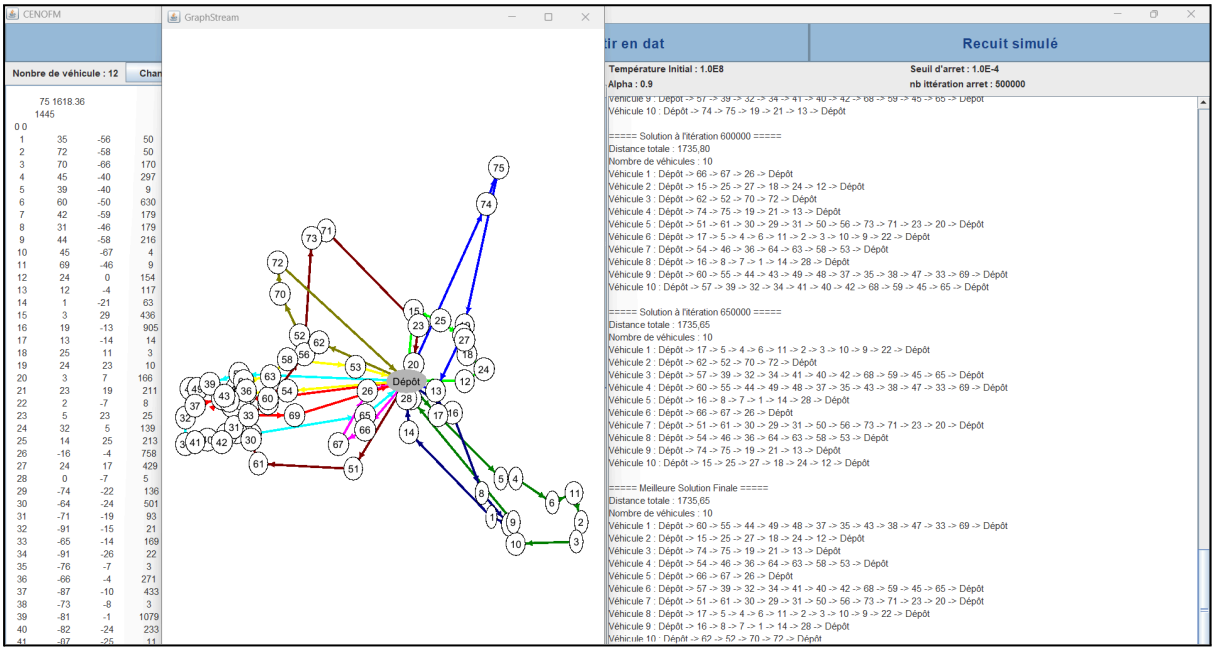
```
public void convertir(String outputPath) {
    this.ct.convertir(outputPath);
}
```

c) ConversionVrpDat

```
writer.println( "nbClientDep = "  + donnees.getTableauNoeudsComplet().length + ";" );
writer.println( "Qmax = "        + donnees.getqMax() + ";" );
writer.println( "nbVehicules = " + donnees.getNbVehicules() + ";" );
writer.println();

writer.print( s: "Demande = [" );
for ( int i = 0; i < donnees.getTableauNoeudsComplet().length; i++ )
{
    writer.print( donnees.getNoeud(i).demande + ( i == donnees.getTableauNoeudsComplet().length - 1 ? "" : "," ) );
}
writer.println( x: "];" );
writer.println();

writer.println( x: "Distance = [" );
for ( int i = 0; i < donnees.getTableauNoeudsComplet().length; i++ )
{
    writer.print( s: " [" );
    for ( int j = 0; j < donnees.getTableauNoeudsComplet().length; j++ )
    {
        double dist = donnees.getNoeud(i).distance( donnees.getNoeud(j) );
        writer.printf( Locale.US, format: "%.2f", dist );
        if ( j < donnees.getTableauNoeudsComplet().length - 1 ) { writer.print(s: " "); }
    }
    writer.println( i == donnees.getTableauNoeudsComplet().length - 1 ? "]" : "]," );
}
writer.println( x: "];" );
writer.close();
```

E. Recuit simulé

1. Interface graphique



2. Code

a) Clic sur "Recuit simulé"

```
if (e.getSource() == this.btnRecuit)
```

b) Saisie des paramètres

```
Object[] contenu = {
    "Température initiale (> 0) :", ch1,
    "Seuil d'arrêt (> 0) :", ch2,
    "Alpha (entre 0.1 et 0.9) :", ch3,
    "nombre d'itérations identique :", ch4
};
```

Validation :

```
tInit = Double.parseDouble(ch1.getText().trim());
a = Double.parseDouble(ch3.getText().trim());
if (a < 0.1 || a > 0.9)
```

c) Lancement du recuit

```
ResultatRecuit r = rs.resoudre(temperature, temperatureMin,
alpha, nbIttArret, interval);
```

d) Initialisation RecuitSimuleCVRP

```
Solution actuelle = genererSolutionInitiale();
Solution meilleure = actuelle.copie();
double temperature = tempInit;
```

e) Boucle principale

```
while (temperature > seuilArret && iterationsSansAmelioration
< nbIttSansAmelioration)
```

f) Génération d'un voisin

```
Solution voisin = genererVoisin(actuelle);
```

Dans :

```
int choix = random.nextInt(5);
case 0 -> mouvementRelocate(voisin);
case 1 -> mouvementSwap(voisin);
case 2 -> mouvement2OptIntra(voisin);
case 3 -> mouvement2OptInter(voisin);
case 4 -> mouvementOrOpt(voisin);
```

g) Calcul du delta

```
double delta = voisin.getDistanceTotale() -
actuelle.getDistanceTotale();
```

h) Critère d'acceptation

```
if (delta <= 0 || Math.exp(-delta / temperature) >
random.nextDouble())
```

Acceptation déterministe si amélioration
Acceptation probabiliste sinon

i) Mise à jour meilleure solution

```
if (actuelle.getDistanceTotale() <
meilleure.getDistanceTotale()) {
    meilleure = actuelle.copie();
}
```

j) Refroidissement

```
temperature = alpha * temperature;
```

k) Fin du programme

```
double temps = (fin - debut) / 1000.0;
return new ResultatRecuit(meilleure, snapshots, temps,
iteration);
```

3. Interprétation du résultat

Quand on clique sur recuit simulé, le programme affiche une solution initiale générée aléatoirement, les solutions générées tous les 5000 itérations, puis la solution la plus optimale, et en dernier le temps d'exécution de l'algorithme du recuit simulé.

Les solutions sont affichées de la façon suivante:

- distance total parcourue dans cette solution
  - C'est la somme des distances :
    - dépôt → premier client
    - entre clients
    - dernier client → dépôt
- Nombre de véhicules utilisés
  - Peut être inférieur ou égal au nombre demandé.
- Détail des tournées
  - Exemple : `Véhicule 1 : Dépôt -> 5 -> 12 -> 7 -> Dépôt`
  - Cela signifie que le véhicule part du dépôt, visite les clients 5, 12 et 7, puis retourne au dépôt.

Exemple :

```
Distance totale : 1365.42

Nombre de véhicules : 8

Véhicule 1 : Dépôt -> 5 -> 12 -> 7 -> Dépôt

Véhicule 2 : Dépôt -> 3 -> 14 -> 9 -> Dépôt

...

Temps d'exécution : 3.24 secondes
```