

Rapport Optimisation Discrète - Printemps 2018

Étudiants

- **JACOUD Bastien**
- **REMOND Victor**
- Lien [dépôt GitHub C++ classique](#)
- Lien [dépôt GitHub Qt](#)
- Téléchargement [Version Stable C++ classique](#)
- Téléchargement [Version Stable Qt](#)

Table des matières

- I - Introduction
- II - Représentation d'un graphe
 - 1 - Représentation du graphe
 - A - Nœuds et Distances
 - B - Parcours du Graphe
 - 2 - Visualisation
 - A - Mode console
 - B - Interface graphique

- 3 - Génération d'une solution aléatoire
- III - Algorithme Tabou
 - 1 - Implémentation
 - 2 - La récupération du voisinage
 - 3 - Résultats obtenu
- IV - Algorithme Génétique
 - 1 - Implémentation
 - 2 - Améliorations

I - Introduction

Dans le cadre d'un projet, nous avons eu à traiter le cas du *Capacited Vehicle Routing Problem*, ou *CVRP*. Le but de cet exercice est de déterminer le meilleur ensemble d'itinéraires, commençant et finissant tous au même point, aussi appelé *dépot*. Cet ensemble d'itinéraires doit être capable de desservir chaque des point préalablement défini. Il faut alors non seulement déterminer un trajet solution capable de desservir chacun des points, mais aussi en **minimisant le coût**, c'est à dire en **minimisant la somme des distances** effectuées par les camions.

Pour ce projet, nous avons eu à réaliser deux étapes principales :

- La modélisation du problème sous forme de graphe.
- L'implémentation des deux algorithmes permettant de diminuer le coût de la solution.

Nous avons donc eu à réaliser un algorithme à base de voisinage et un algorithme à base de population. Nous avons finalement fait le choix de

réaliser une **Recherche Tabou** comme algorithme à base de voisinage ainsi qu'un **Algorithme Génétique** comme algorithme à base de population.

Afin de minimiser le temps d'exécution de notre application, nous avons fait le choix de réaliser ce projet avec le langage *C++*. Dans un premier temps, nous avons utilisé l'IDE *CLion*, puis nous avons par la suite utilisé *QT* afin de réaliser une interface Graphique.

Pour générer nos graphes, nous utilisons *Graphviz*, comme mentionné dans notre [Readme](#).

II - Représentation d'un graphe

Dans un premier temps, nous avons dû convenir d'un objet représentant un graphe afin de pouvoir manipuler nos données. L'une de nos principales préoccupations était de pouvoir l'utiliser de façon relativement rapide, nous avons donc essayé de créer des fonctions de manipulation un minimum optimisées. La structure interne de notre classe **Graph** peut se décomposer en deux grosses partie :

- La représentation des Nœuds et des Distances.
- La représentations des chemins empruntés pour l'instance actuelle du Graphe.

Nous aborderons ensuite les choix que nous avons fait en ce qui concerne la visualisation de nos résultats :

- Visualisation en console.
- Graphe au format SVG.

Enfin, nous expliqueront comment est construite la solution initiale.

1 - Représentation du graphe

A - Nœuds et Distances

Nous chargeons tout d'abord les nœuds depuis le fichier de sources, en sachant que chaque nœud possède un id, une quantité de produits à livrer, et une position représentée par un couple $\{x,y\}$.

Une fois ces nœuds chargés en mémoire, on génère la matrice des distances. Comme la distance entre deux nœuds A et B est la même qu'entre B et A, nous ne manipulons pas une matrice carrée complète mais simplement une matrice triangulaire, avec une vérification sur les indices demandés :

- La fonction `getDistance(index1, index2)` réorganise les indices afin de lire `distances[i][j]` avec $i = \max(\text{index1}, \text{index2})$ et $j = \min(\text{index1}, \text{index2})$.

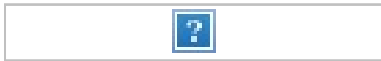
B - Parcours du Graphe

Pour représenter les chemins empruntés par les camions, nous avons créé une classe `Truck`, qui est une liste chaînée améliorée de nœuds. A chaque ajout d'un nœud sur le parcours d'un camion, ce dernier est marqué comme utilisé. De plus, avant d'ajouter un nœud au parcours d'un camion

on vérifie que cela ne cause pas de dépassement de capacité, afin de ne pas générer de graphe qui ne serait pas une solution.

Cette classe permet aussi de calculer la distance parcourue par le camion, ce qui simplifie le calcul du coût total du graph : il nous suffit de sommer la distance parcourue par chaque camion.

Enfin cette classe nous permet d'obtenir un vecteur d'entiers représenatant le parcours du camion. Exemple avec le graph ci-dessous :



Le parcours bleu sera représenté par le vecteur $\{ 0, 1, 2, 3, 4, 5, 6, 0 \}$ et le rouge par le vecteur $\{ 0, 14, 12, 11, 10, 9, 8, 7, 0 \}$.

Ces vecteurs ne permettent pas de connaître la distance du parcours, mais restent utiles pour se faire une idée du parcours et comparer rapidement deux graphes à la console.

2 - Visualisation

A - Mode console

Au début du projet, nous avions à travailler en mode console, ce qui nous a obligé à trouver un affichage correct d'une solution. Pour cela, nous avons choisi d'afficher les vecteurs représentant les différents camions, puis le coût du graphe.

Un exemple de solution affichée serait :

Graphe de base

Truck 1 :

0 -> 8 -> 18 -> 21 -> 22 -> 27 -> 28 -> 31 -> 0

Truck 2 :

0 -> 1 -> 6 -> 11 -> 15 -> 24 -> 0

Truck 3 :

0 -> 9 -> 13 -> 20 -> 25 -> 26 -> 29 -> 0

Truck 4 :

0 -> 3 -> 4 -> 14 -> 17 -> 19 -> 30 -> 0

Truck 5 :

0 -> 2 -> 5 -> 7 -> 10 -> 12 -> 16 -> 23 -> 0

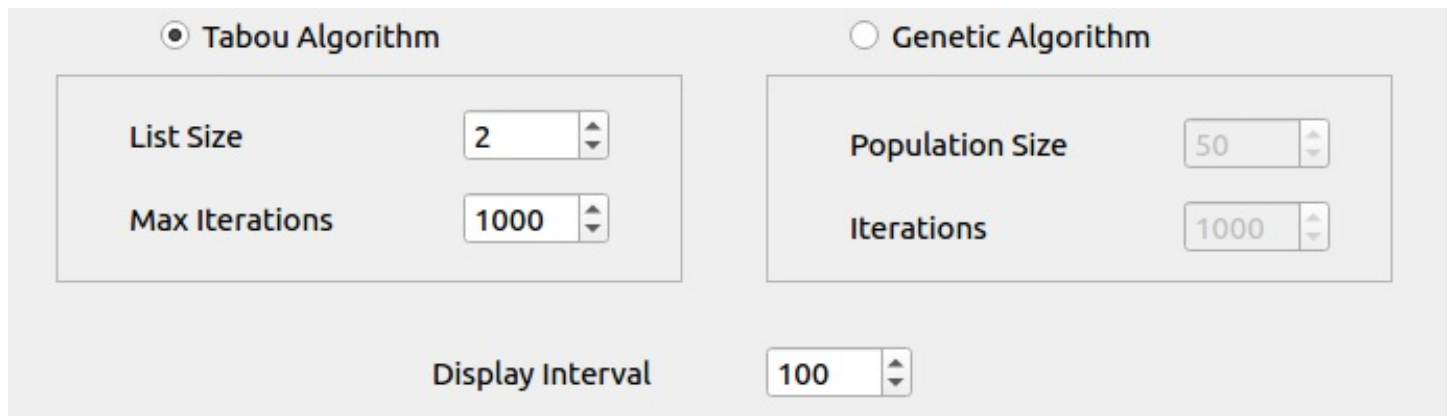
Solution's cost : 2215.092281

B - Interface graphique

Nous avons ensuite décidé de trouver un moyen pour afficher nos graphes sous leur véritable apparence. Nous avons ainsi choisi d'utiliser *GraphViz*, et plus précisément *dot* pour générer des graphes à partir de simples fichiers texte.

Tous les graphes servant d'exemples dans ce rapport ont été générés par *dot*. Nous générons les fichiers grâce à notre classe `GraphViz`, puis nous exécutons la ligne de commande permettant de générer les graphes. C'est pour cela que l'ordinateur sur lequel notre programme est exécuté doit avoir *dot* d'installé. Cette méthode fonctionne parfaitement sous Linux (testé sur plusieurs ordinateurs), mais son fonctionnement sur d'autres OS n'est pas assuré.

Nous avons ensuite décidé de changer de base pour notre programme : nous avons commencé en C++ pur avec CLion comme IDE, mais nous sommes ensuite passé sous Qt afin de pouvoir créer une interface graphique rendant l'utilisation du programme plus intuitive. Parmi les améliorations apportées lors du passage sous Qt se trouve l'apparition d'une fenêtre *Settings* permettant de choisir quel algorithme utiliser et d'en renseigner les paramètres.



The screenshot shows a settings dialog with two main sections. The first section, 'Tabou Algorithm', is selected with a radio button and contains two parameters: 'List Size' with a value of 2 and 'Max Iterations' with a value of 1000. The second section, 'Genetic Algorithm', is unselected and contains two parameters: 'Population Size' with a value of 50 and 'Iterations' with a value of 1000. At the bottom, there is a 'Display Interval' parameter with a value of 100. All values are displayed in spinners.

Un objectif de notre passage sous Qt était aussi la création d'une fenêtre de monitoring de l'algorithme (cf Exemple1.svg).

Cependant la partie gauche de la fenêtre qui reste blanche est dûe à une erreur que nous n'avons pas réussi à régler : à cet endroit aurait dû être affiché le graphe de la meilleure solution rencontrée par l'algorithme jusqu'à maintenant.

Les graphes générés restent cependant disponibles dans le dossier `graphviz/`.

3 - Génération d'une solution aléatoire

Afin de pouvoir lancer nos algorithmes, ils nous fallait la possibilité de

créer une solution initiale (une seule pour l'algorithme Tabou, une par membre de la population dans le cas de l'algorithme génétique).

Nous avons décidé de générer une simple solution aléatoire comme suit :

```
Soit N* une copie de l'ensemble des noeuds du graphe
Soit T& une liste de références vers les camions du graphe
```

```
Faire :
```

```
    go <- true
```

```
    Pour N dans N* :
```

```
        |   T* <- copie(T&)
```

```
        |   i  <- entier aléatoire dans [0;taille(T*)]
```

```
        |   ok <- false
```

```
        |   Faire :
```

```
            |   |   Si ( quantite(N) <= placeLibre(T*[i])
```

```
                |   |   |   ajouter N à T*[i]
```

```
                |   |   |   ok <- true
```

```
                |   |   Sinon
```

```
                |   |   |   T* <- T* privé de T[i]
```

```
                |   |   |   Si ( taille(T*) == 0 )
```

```
                    |   |   |   |   //Plus aucun camion disponible
```

```
                    |   |   |   |   //Cette solution n'est pas viable
```

```
                    |   |   |   |   go <- false
```

```
                |   |   |   Fin Si
```

```
                |   |   Fin Si
```

```
            |   Tant Que ( ok == false ET go == true )
```

```
    Fin Pour
```



```
Tant Que ( go == false )
```

```
//Tous les nœuds sont placés, la solution est viable
```

III - Algorithme Tabou

1 - Implémentation

L'implémentation de l'Algorithme Tabou n'est pas des plus compliqués. Pour ce faire, nous utilisons une classe `TabouAlgorithm`, contenant tous les éléments indispensables au bon fonctionnement de l'algorithme.

Cette classe est donc principalement composée de trois attributs :

- La liste `m_T`, représentant notre liste Tabou, qui comportera les transformations élémentaires à ne pas reproduire.
- La solution `m_xmin`, représentant la solution avec le plus petit coût actuellement trouvé par l'algorithme.
- Le coût `m_fmin`, représentant le coût minimum trouvé par l'algorithme.

Viennent alors s'ajouter 3 paramètres, modifiables par l'utilisateur. Le premier, et le plus important est la taille de la liste tabou, `M_TAILLE`. Les deux autres, `M_NMAX` et `M_DISPLAY` représentent respectivement le nombre d'itérations à effectuer ainsi que le nombre d'itérations avant l'affichage en mode console d'un graphe.

Nous avons alors pu réaliser l'algorithme de la recherche tabou, tel que

décrit ci-dessous :

```
Génération de la solution aléatoire dans m_graph
m_xmin <- solution associée au graphe actuel;
m_fmin <- coût de la solution m_xmin;
Faire :
|   V = <- récupération du voisinage, sans sélectionner
les éléments de la liste tabou
|   Si (V n'est pas vide)
|   |   y_jmin <- indice pour laquelle la valeur de la
solution contenue dans V est minimale
|   |   y_fmin <- valeur de cette solution minimale de
V
|   |   delta_f <- y_fmin - cout de la solution actuell
e (cout de m_graph);
|   |   Si (delta_f >= 0)
|   |   |   On ajoute à notre liste tabou m_T la tr
ansformation élémentaire permettant d'obtenir la solution V[y
_jmin]
|   |   Fin Si
|   |   Si (y_fmin < m_fmin)
|   |   |   m_fmin <- y_fmin;
|   |   |   m_xmin <- Solution correspondant à V[y_
jmin];
|   |   Fin Si
|   |   m_graph <- Graph correspondant à la solution m_
xmin
|   |   Affichage en mode console de m_graph tout les M
```

```
_DISPLAY
```

```
|      Fin Si
```

```
Tant que (i < M_NMAX);
```

Afin d'optimiser au maximum cet algorithme, nous avons fait le choix de récupérer uniquement l'indice `y_jmin` du minimum contenu dans le voisinage au lieu de récupérer un objet.

De même, toutes les transformations élémentaires sont récupérées au moment de la génération du voisinage, étant donné que lorsqu'un graphe est ajouté au voisinage, on a réalisé au préalable la transformation élémentaire permettant de le générer.

2 - La récupération du voisinage

Nous allons maintenant approfondir la fonction

`getVoisinage(listeTabou, listTransformation)`, définie dans notre classe `Graph`.

Cette fonction est une des plus importantes de notre algorithme, puisqu'elle permet à la fois de générer et de nous retourner le voisinage du graphe actuel, mais aussi une liste de toutes les transformations élémentaires associées aux graphes contenus dans le voisinage. Cette fonction prend également en compte notre liste tabou, pour ne pas réaliser de transformation élémentaire contenue dans cette dernière, et donc non autorisée.

En premier lieu, nous avons défini nos transformations élémentaires. Nous nous sommes rapidement aperçu que les transformations peuvent être de deux types :

- Une permutation entre deux sommets du graphe, au niveau du passage d'un camion, ce qui correspond à la fonction `invertNodes(node1, node2)`.
- Un déplacement de n'importe quel sommet du graphe, au niveau du passage d'un camion, ce qui correspond à la fonction `moveNode(node, old_truck, new_truck, new_position)`.

Grâce à ces deux transformations élémentaires, nous avons alors pu identifier tous les graphes possibles, composant notre voisinage. Voici notre algorithme nous permettant de générer notre voisinage :

```
Pour (Chaque sommet de notre graphe, nommé node1)
| // Make all elementary OP based on node inversion
| Pour (Chaque sommet suivant node 1, nommé node2)
| | pair_tabou <- transformation élémentaire d'invers
ion de node1 et node2
| | Si (pair_tabou n'est pas dans la liste tabou)
| | | new_graph <- Inversion de node1 et node2
| | | Si (new_graph est solution){
| | | | On ajoute new_graph dans notre voisinage
| | | | On ajoute pair_tabou à notre liste de tra
nsformations
| | | Fin Si
| | Fin Si
| Fin Si
```

```

|   Fin Pour
|   // Make all elementary OP mooving just one node
|   Pour (Chaque sommet autre que node 1, nommé node2)
|   |   pair_tabou <- transformation élémentaire de chang
ement de place de node1
|   |   Si (pair_tabou n'est pas dans la liste tabou)
|   |   |   new_graph <- déplacement de node1
|   |   |   Si (new_graph est solution){
|   |   |   |   On ajoute new_graph au voisinage
|   |   |   |   On ajoute pair_tabou à notre liste de tra
nsformations
|   |   |   Fin Si
|   |   Fin Si
|   Fin Pour
Fin Pour
On retourne notre voisinage

```

De cette manière, nous récupérons notre voisinage, composé de toutes les transformations élémentaires possibles. De plus, nous éliminons directement les transformations contenues dans notre liste tabou. Nous récupérons également toutes les transformations effectuées dans une liste, afin de pouvoir ajouter directement la prochaine transformation qui va être effectuée à notre liste tabou.

3 - Résultats obtenu

Nous avons donc réalisé de nombreux tests sur notre algorithme de

recherche tabou, en variant les paramètres ainsi que les différents jeux de données. Chaque fois, nous avons paramétré notre algorithme pour qu'il **effectue 100 itérations et affiche le résultat toutes les 20 itérations.**

1. Dans un premier temps nous avons utilisé le jeu de données *Data01.txt*. Nous obtenons une solution initiale générée qui possède un coût d'environ **2000**. Notre algorithme tabou nous permet de diminuer considérablement le coût, qui va se rapprocher de **800** et même parfois va devenir inférieur. Voici un exemple de résultat obtenu pour le premier jeu de données :
cf (graphviz/Tabou_01_OK.svg)
2. Nous avons ensuite utilisé le second jeu de données : *data02.txt*. Cette fois, la solution initiale générée est d'environ **3800** et diminue jusqu'à **1500**. Voici un exemple obtenu avec ce jeu de données :
cf (graphviz/Tabou_02_OK.svg)
3. Pour le troisième jeu de données, le coût débute à environ **4500** et diminue jusqu'à **2000**. Voici un exemple de graphe obtenu en sortie de notre algorithme :
cf (graphviz/Tabou_03_OK.svg)
4. Cette fois le coût varie entre **3300** à la génération de la solution initiale et **1480** à la fin de l'algorithme. Voici un exemple de graphe généré :
cf (graphviz/Tabou_04_OK.svg)

5. Pour finir, avec le dernier jeu de données, notre coût varie entre **3700** au lancement et **1600** en fin d'algorithme. Voici un exemple :
cf (graphviz/Tabou_05_OK.svg)

Au niveau du paramétrage de notre liste tabou, la taille que nous avons fixé pour notre liste varie entre 10 et 20 suivant le test effectué et le jeu de données utilisé. Comme nous effectuons ici seulement 100 itérations, il est inutile d'augmenter sa taille.

IV - Algorithme Génétique

1 - Implémentation

Avant de se lancer dans l'implémentation de l'algorithme, nous avons déjà dû choisir une représentation pour les membre de la population qui soit relativement facile à manipuler dans le fonctions de reproduction ou de mutation.

Nous avons ainsi fait le choix de voir un ensemble de trajets comme un seul et unique trajet en omettant les passage par le noeud racine. Exemple avec le graph suivant.

cf graphviz/exemple1.svg

Dans ce graph, on remarque deux chemins :

Bleu : 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 0

Rouge : 0 -> 14 -> 13 -> 12 -> 11 -> 10 -> 9 -> 8 -> 7 -> 0

On modélisera ces deux chemins par un seul vecteur d'entiers comme ci-dessous :

Vecteur : { 1, 2, 3, 4, 5, 6, 14, 13, 12, 11, 10, 9, 8, 7 }

La première implémentation de l'algorithme génétique était relativement basique, mais comportait plusieurs défauts. Par exemple, la population était entièrement renouvelée d'une itération à la suivante.

Les mutations et fonctions de reproduction déjà implémentées étaient cependant déjà bien choisies, avec par exemple une implémentation d'un algorithme de recombinaison des nœuds. Pour fonctionner, celui-ci crée une matrice d'adjacence pour chacun des deux parents choisis puis les met en commun et crée un fils. Source de l'algorithme

edgeRecombination : [Wikipedia](#).

Une seconde fonction de reproduction utilisée était la fonction *NodeSwap*, qui permettait d'intervertir deux nœuds ou plus dans le schéma de représentation d'un membre de la population.

2 - Améliorations

Suite aux performances peu convaincantes de notre algorithme, nous avons modifié son fonctionnement en ajoutant une fonction de mutation, appelée *Partially Mapped Crossover*, trouvée sur [ce site](#). Nous avons aussi

modifié le passage d'une génération à une autre : dans notre précédente version, la totalité de la population était changée. Cette fois, nous avons fait en sorte de n'en modifier qu'une partie, afin de ne pas avoir un algorithme qui dérive totalement.