Rapport de programmation fonctionnelle 2018 - 2019

Recherche d'un planificateur de tournée efficace

Adrien BLASSIAU

ENSIIE - 11 janvier 2019

Sommaire

1. INTRODUCTION

L' objectif du projet est d'implémenter un planificateur de tournée efficace pour un assez grand nombre de villes. On veut donc déterminer le plus court chemin possible qui visite toutes les villes et qui termine dans la ville de départ.
Ce problème s'apparente au **Problème du voyageur de commerce**, un problème d'optimisation NP-complet, dans lequel les villes doivent être traversées une et une seul fois. Cela revient à rechercher le plus court cycle hamiltonien dans le graphe G. Cependant, dans notre problème, les villes peuvent être traversées plus d'une fois.
On nous propose une méthode de résolution en trois phases et divisées en deux temps que voici :

- Dans un premier temps, les villes seront toutes reliées entre elles. Le graphe représentant la situation sera **complet**.
- Dans un second temps, les villes ne seront plus toutes reliées entre elles, à part celles de l'enveloppe convexe. Le graphe **ne sera plus complet** mais devra rester **connexe**.

Les trois phases de résolution sont données dans l'énoncé et seront chacune présentées dans ce rapport dans 3 parties distinctes. On mettra en avant à chaque fois la démarche mis en oeuvre et les choix effectués ainsi que les différences entre la résolution pour le graphe complet (1er temps) et le graphe non complet (2e temps).

2. Organisation du code source et du projet

2.1. Foncteurs et modules

Le projet est divisé en plusieurs fichiers qui implémentent pour la plupart des foncteurs. Tous les fichiers de base sont construits de manière à être le plus indépendants possibles les uns des autres, pour pouvoir être réutilisés dans un autre projet par exemple. On retrouve dans le fichier `.depend` l'ensemble des dépendances.

Le module principale `Tournee` est créé dans les différents main du projet à partir du foncteur de tournée `MakeTournee` contenu dans `tournee.ml`, et d'un module de villes, `OrderedTypeVille`, contenu dans `ville.ml` et passé en argument.

Dans `MakeTournee`, on crée et utilise un ensemble de modules à partir de foncteurs présents dans les différents fichiers du projet. Tous ces foncteurs, sauf `CityGraph` qui prend un OrderedType de type string, acceptent `OrderedTypeVille` en argument.
On utilise ainsi les modules suivants dans le foncteur `MakeTournee` de `tournee.ml` :

- `CityGraph` construit à partir de `MakeGraph(String)` définit dans `graph.ml`, est utilisé pour stocker les routes uniquement pour la partie 2 de l'énoncé
- `Drawer` construit à partir de `MakeDrawer(OrderedTypeVille)` définit dans `draw.ml`, est utilisé pour afficher graphiquement la tournée
- `ConvexHull` construit à partir de `MakeConvexHull(OrderedTypeVille)` définit dans `convex_hull.ml`, est utilisé pour trouver l'enveloppe convexe d'une tournée
- `CitySet` construit à partir de `MakeCitySet(OrderedTypeVille)` définit dans `cit_set.ml`, est utilisé pour stocker les villes à placer dans la tournée
- `KDTree` construit à partir de `MakeKDTree(OrderedTypeVille)` définit dans `kd_tree.ml`, est utilisé pour stocker les coordonnées des villes et faciliter la recherche du plus proche voisin
- `Reader` construit à partir de `MakeReader(OrderedTypeVille)` définit dans `read.ml`, est utilisé pour lire les différents fichiers en entrée du programme et créer les premières structures utiles.

Deux autres modules sont utilisés dans l'ensemble du projet : `Useful` (`useful.ml`) qui possède un ensemble de fonctions utiles et inclassables et `Stack` (`stack.ml`) qui implémente des piles génériques.

Enfin, d'autres modules sont spécifiques aux tests et sont tous utilisés dans `main_test.ml`.

2.2. Structures de données et type principal

Dans cette partie, on précise les différentes structures de données et types principaux utilisés tout au long du projet.

2.2.1 Le type ville

Le type ville est le type qui représente une ville. C'est un type somme, constitué de trois champs : un pour le nom de la ville, un pour sa coordonnée x (longitude) et un pour sa coordonnée y (latitude). Le type est défini dans le fichier `ville.ml`. Dans ce fichier, on retrouve aussi la définition du module `OrderedTypeVille` qui sera ensuite utilisé en entrée du foncteur principal du projet : `Tournee`. Le module `OrderedTypeVille`, sous le nom de `N`, sera ensuite utilisé dans l'ensembe des modules créés dans le module de Tournée. `OrderedTypeVille` contient un ensemble de fonctions utiles : des getters, des setters, des constructeurs, ... qui permettent d'effectuer des opérations sur les villes.

2.2.2 Le stockage des villes à placer dans la tournée

Pour stocker les villes ne se trouvant pas dans la tournée, on choisit d'utiliser un Set. Cette structure est plus rapide qu'une simple liste pour certaines opérations comme le retrait d'un élément par exemple, car elle est implémentée avec un avl. Un foncteur implémente le Set de ville. Il se trouve dans `city_set.ml`. On utilise un module qui implémente ce foncteur avec comme module en entrée du foncteur le module de ville. La plupart des fonctions utilisent directement les fonctions du module Set d'OCaml, j'ai rajouté certaines fonctions d'affichage et de construction particulières.

2.2.3 Le stockage des villes de la tournée

Pour stocker les villes dans la tournée et donc le chemin courant, j'utilise une simple List. Le Set n'était pas approprié car certaines villes peuvent se retrouver en double dans la tournée, ce que le Set ne permet pas de gérer directement.
L'utilisation de la liste est simple. On y stocke la tournée courante, ainsi chaque ville qui se suit dans la liste se suit dans la tournée.

2.2.4 Le stockage des différentes routes

Pour la partie 2 du projet, certaines routes n'existent plus. Il faut donc une structure qui rende compte de l'ensemble des routes existantes. La structure choisie est une structure classique de graph, c'est-à-dire un Map de Set. On implémente les graphes dans un foncteur se trouvant dans `graph.ml`. On utilise un module qui implémente ce foncteur avec comme module en entrée du foncteur un simple module de chaîne de caractères car on ne souhaite stocker que le nom des villes. Un noeud de notre structure est donc le nom d'une ville.

2.2.5 Le stockage de la position des villes

Comme on le verra par la suite, une recherche du plus proche voisin est nécessaire dans la phase 2 du projet. Or cette algorithme est assez coûteux, en $\mathcal{O}(n)$. Un des défis du projet est de réduire cette complexité pour accélérer la recherche de la tournée la plus optimale possible. On utilise pour cette étape un arbre k-d qui permet de réduire la recherche du plus proche voisin à une complexité en $\mathcal{O}(\log n)$. L'arbre est implémenté dans un foncteur se trouvant dans `kd_tree.ml`. On utilise un module qui implémente ce foncteur avec comme module en entrée du foncteur un module de ville. La plupart des fonctions offertes par le foncteurs permettent notamment la

construction de cette arbre et la recherche du plus proche voisin à partir de cette arbre. On verra par la suite le fonctionnement de cette arbre.

2.2.6 Le stockage des villes lors de la construction de l'enveloppe convexe

On utilise une pile lors de la construction de l'enveloppe convexe. Un module de pile est implémenté dans `my_stack.ml`. Contrairement au module de pile `Stack` d'OCaml, qui utilise des concepts **impératifs**, ici le module est **purement fonctionnel**. Les fonctions sont classiques et assez sommaires.

2.3. Makefile

Un `Makefile` permet de compiler les fichiers et générer un exécutable pour chaque partie du problème (graphe complet et graphe non complet) ainsi que des exécutables de tests. Le fichier `MANUEL.md` donne les possibilités du Makefile.

3. PHASE 1

Cette phase correspond à la recherche d'une tournée sur un petit nombre de villes.
Le processus est donné dans l'énoncé, on tire soit une ville unique soit on récupère l'enveloppe convexe de la tournée. Le tirage de la ville unique ne possède pas de soucis, on prend le premier élément du Set de ville. On s'intéresse ici à la construction de l'enveloppe convexe qui s'effectue grâce aux fonctions du fichier `convex_hull.ml`

3.1. Construction de l'enveloppe convexe avec la méthode de Graham

On utilise la méthode du parcours de Graham qui effectue la recherche de l'enveloppe convexe avec une complexité en $\mathcal{O}(n \log n)$ où n est le nombre de points (ici de villes).
Cette méthode est assez simple et est constituée de trois étapes que l'on détaille rapidement ici :

- La première étape consiste à trouver le point pivot, c'est à dire le point d'ordonnée la plus faible ou encore le point "le plus bas". À deux points d'ordonnées égales, on prend celui d'abscisse le plus faible donc le point "le plus à droite". Cette recherche se fait simplement en parcourant la liste des points. Par exemple, le pivot est ici 61 (voir figure plus bas). On passe à l'étape 2.
- Cette deuxième consiste à trier les points selon l'angle qu'ils font avec l'axe des abscisses relativement au pivot. On utilise pour cela la fonction `atan2` qui permet de trouver rapidement cet angle. On les stocke dans une liste sans prendre en compte le pivot. On passe à la dernière étape.
- Cette dernière étape est l'algorithme en lui même. Il utilise une pile et permet de construire l'enveloppe itérativement. On commence avec le pivot que l'on place au sommet de la pile. On place ensuite dans la pile le premier élément de la liste triée. On itère sur la liste triée. Pour chaque élément de cette liste, tant que la pile est de hauteur supérieur à 2 et que l'élément en question est à droite du segment formé par les deux éléments au sommet de la pile, on dépile la pile. Une fois que cette condition n'est plus respectée, on empile l'élément et on recommence le processus avec le prochain élément du tableau. On trouve ainsi l'enveloppe convexe.

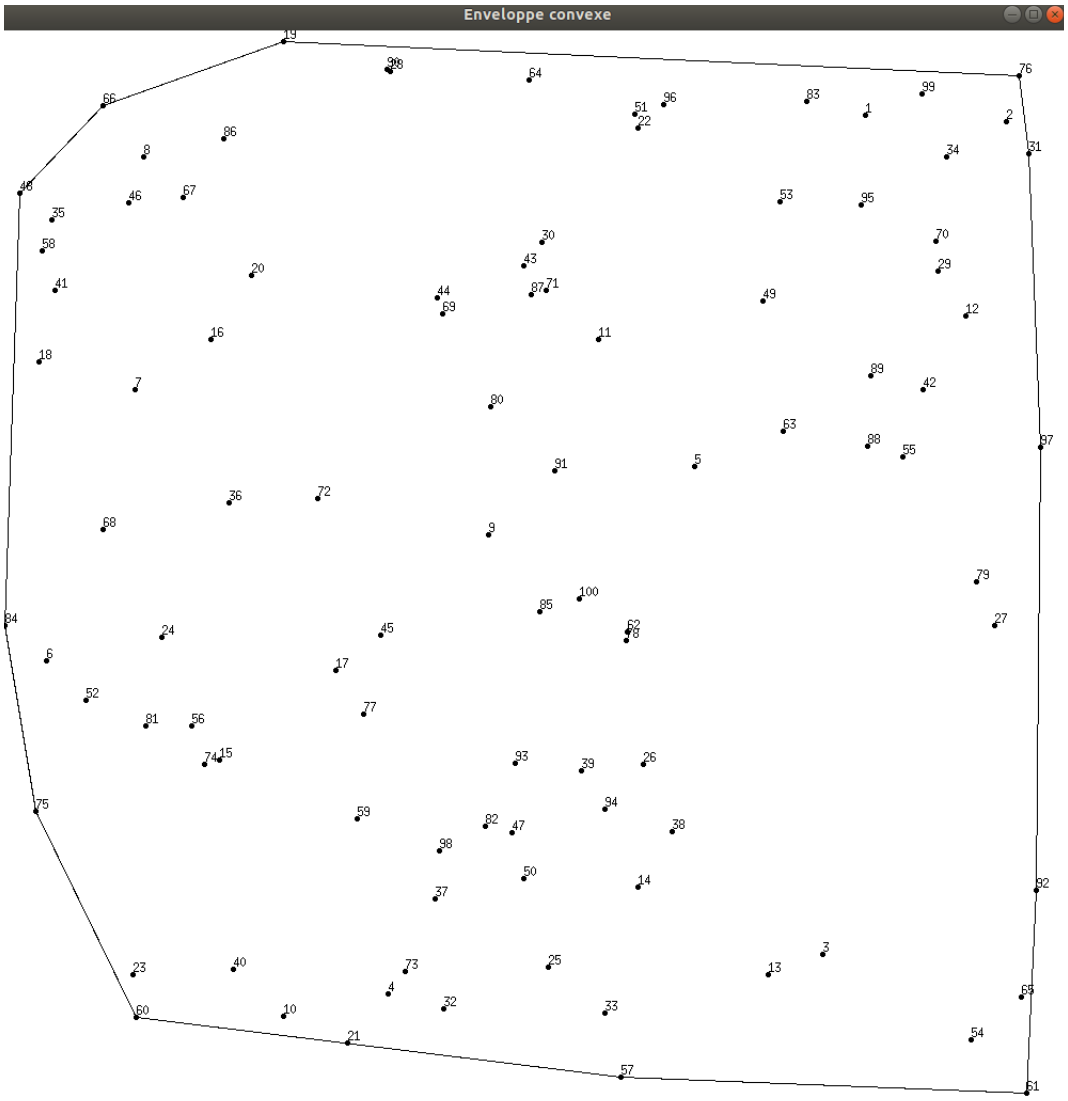

FIGURE 1 – Enveloppe convexe sur 100 points avec le module `Graphics`

3.2. Remarques et idée d'améliorations

On remarque que pour la partie 2, c'est-à-dire dans le cas où toutes les villes ne sont pas reliées, on considère que celles de l'enveloppe convexe le sont. Une autre option envisagée était de construire l'enveloppe convexe puis ensuite de réduire le chemin obtenu, jusqu'à ce qu'il vérifie les contraintes d'existence de chemin.

Comme idée d'amélioration, on aurait pu utiliser **l'algorithme de Chan** qui a une complexité en $\mathcal{O}(n \log h)$ où h est le nombre de points de l'enveloppe convexe et n le nombre de points total. Cet algorithme est plus lourd à implémenter. Il calcule des enveloppes convexes de sous groupes de points avec par exemple **la méthode de Graham**.

4. PHASE 2

Cette phase correspond au corps de l'algorithme qui correspond à prendre des villes non présentes dans la tournée de manière à minimiser sa taille totale. Avant de continuer, le sujet soulevait une légère ambiguïté dans les points b et c. En effet, la phrase "... la distance minimale aux villes de la tournée ...", peut se comprendre de deux manières :

- comme une recherche de la distance minimale à chacune des villes de la tournée, ce qui revient à trouver la distance au plus proche voisin pour chaque ville non présente dans la tournée puis prendre la plus proche (ou la plus loin), ce qui à première vue a une complexité en $\mathcal{O}(n^2)$ mais est **exacte**
- comme une recherche de la distance minimale au barycentre des villes de la tournée, le "aux" est donc à comprendre comme "moyenne", ce qui à première vue a une complexité en $\mathcal{O}(n)$ mais est **approché**

J'ai choisi la première interprétation car je trouvais que la seconde trahissait un peu l'esprit de l'exercice (notamment de certains travaux de recherche où l'on peut voir que c'est la première méthode qui est suivie ...).

La méthode choisie est cependant longue, c'est pour cela que j'ai décidé de faire subir au villes de la tournée un pré-traitement particulier qui m'a permis ensuite d'abaisser la complexité de cette recherche.

4.1. Arbre k-d et recherche du plus proche voisin

J'ai décidé de stocker mes points dans un arbre binaire que l'on appelle **arbre k-d** (ou k-d tree en anglais) sur lequel j'ai pu ensuite effectuer des recherches du plus proche voisin avec une complexité en $\mathcal{O}(\log n)$, ce qui fait descendre la complexité de la partie d'un $\mathcal{O}(n^2)$ à un $\mathcal{O}(n \log n)$ moyennant un léger pré traitement des points par leur insertion dans l'arbre. Je vais expliquer sa construction ainsi que le fonctionnement de l'algorithme de recherche du plus proche voisin dessus. Le code liée à cette partie se trouve dans `kd_tree.ml`.

4.1.1 Construction

L'arbre k-d est un arbre binaire assez simple à construire. Il repose sur le principe de partitionnement de l'espace. Ici, on travaille en deux dimensions donc l'espace est le plan. Chaque noeud de l'arbre contient un point. Chaque noeud non terminal divise l'espace en deux sous espaces. Ainsi les sous branches droite et gauche d'un noeud représente deux demi espaces. Si le point d'un noeud divise l'espace selon l'axe (Ox) par exemple, il correspond à la médiane de l'ensemble des points dont il fait partie. Ainsi tous les points de coordonnée x inférieure à la coordonnée du point du noeud père se retrouve dans la sous branche gauche. Les autres se retrouvent dans la sous branche droite. Pour chaque noeud, on alterne l'axe selon lequel la division se fait. L'exemple suivant montre une partitionnement de l'espace effectué sur un ensemble de 10 points et son stockage dans l'arbre.

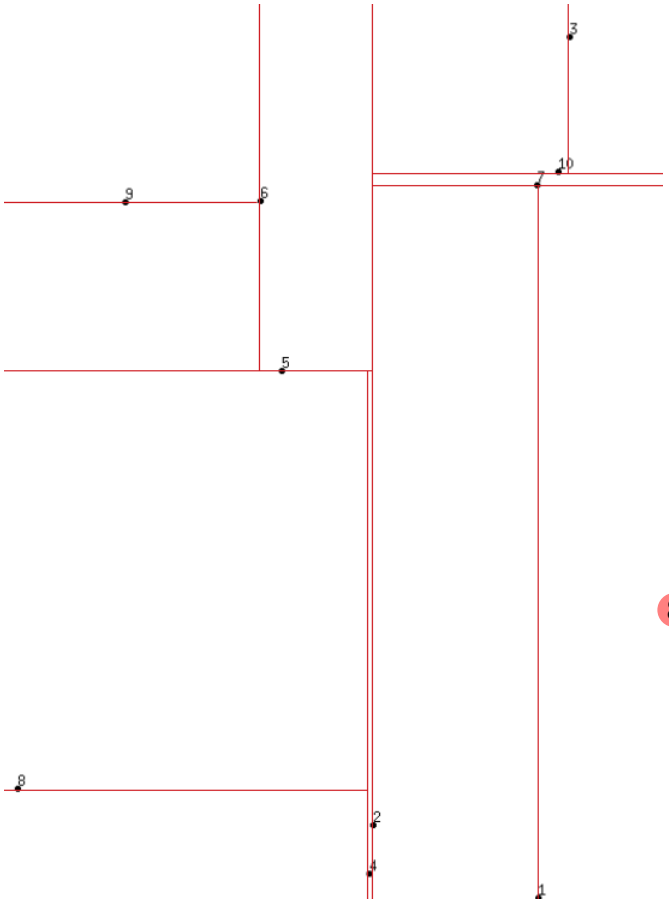

FIGURE 2 – Partionnement de l'espace

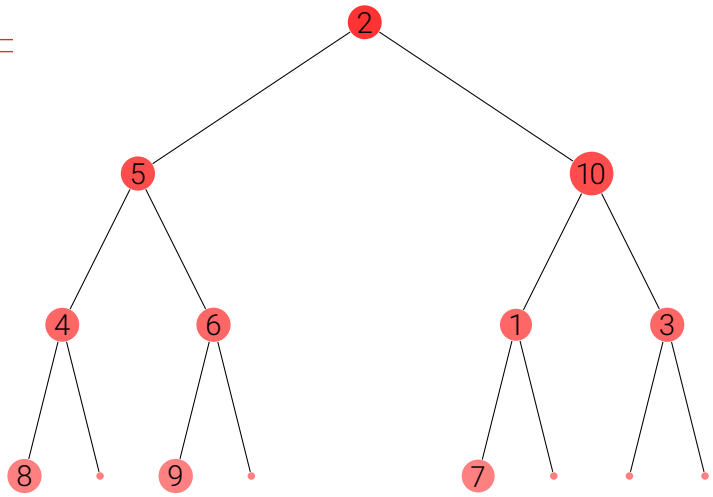

FIGURE 3 – Stockage dans l'arbre k-d

Ici, 2 découpe l'axe (Ox) en deux groupes : `[8;9;6;5;4]` (ensemble A) à gauche et `[7;1;10;3]` (ensemble B) à droite. 5 découpe l'axe (Oy) en deux groupes : `[9;6]` (ensemble C) et `[8;4]`. 10 découpe l'axe (Oy) en deux groupes : `[7;1]` (ensemble C) et [3]. La construction se fait de cette manière.

L'arbre implémenté en OCaml doit aussi indiquer si le noeud se trouve dans la tournée ou non. Ainsi, chaque noeud possède un booléen qui indique si le noeud appartient à la tournée et deux autres booléens qui indiquent si les branches gauche et droite en contiennent, ce qui permettra par la suite de couper des branches de l'arbre lors de la recherche.

4.1.2 Recherche du plus proche voisin avec l'arbre k-d

L'algorithme de recherche du plus proche voisin d'un noeud N opère selon un processus récursif dont voici les grandes étapes :

- On insère le noeud dont on recherche le plus proche voisin dans l'arbre
- Une fois inséré, on remonte l'arbre.
- On examine le noeud père. Si il est le noeud le plus proche du noeud N, il devient le meilleur noeud N_m. Ensuite, on cherche à savoir si un meilleur noeud peut se trouver dans la branche

fille non explorée de ce noeud. Pour cela, on regarde si le cercle de rayon N_mN coupe l'axe donné par le noeud père. Si c'est le cas, on explore la branche en suivant le même processus : insertion, puis remontée et comparaison. Si ce n'est pas le cas, on coupe la branche et on continue la remontée de l'arbre.

- On stoppe le processus à la racine

Certaines petites subtilités sont présentes dans l'algorithme comme par exemple la nécessité d'explorer un noeud, même s'il ne vérifie pas la règle de l'intersection entre l'axe et le cercle, dans certains cas où l'on a pas encore trouvé de plus proche voisin (partie 2 du projet).

4.2. L'insertion d'une ville aléatoire

Beaucoup plus rapidement, la méthode utilisée pour choisir un élément quelconque dans le Set est la suivante : on transforme le Set de villes en liste, on mélange la liste de manière aléatoire (grâce la fonction shuffle présente dans `useful.ml`), et on prend le premier élément de la liste pour la partie 1 ou bien on itère dessus jusqu'à trouver un point insérable dans le cas de la partie 2 du projet.

4.3. Remarques et idée d'améliorations

On remarque que le choix de l'arbre k-d est judicieux, surtout pour les tournées où le nombre de villes est assez important. Pour une tournée de 1000 villes par exemple, avec **HULL**, **NEAREST** et **REPOSITIONNEMENT**, on obtient une solution en 8 secondes en utilisant l'arbre et le double sans l'utiliser, c'est à dire avec la méthode naïve. Un exemple de rapidité est présent dans l'exécutable de test unitaires des fonctions. Le temps de pré traitement des points, c'est à dire de création de l'arbre est largement compensé par l'algorithme de recherche du plus proche voisin.

5. Phase 3

La phase 3 du projet ne pose pas de problème majeur. Elle propose d'améliorer le résultat obtenu en phase 2 en appliquant un algorithme au choix entre celui de repositionnement et celui d'inversion. Pour chacun des algorithmes, on prend en compte la non-présence de chemin pour la partie 2 de l'énoncé.

5.1. Repositionnement

Soit ABCDEFGA notre tournée. Pour le repositionnement, on choisit un triplet ABC, on enlève B, puis on itère sur les triplets qui suivent en essayant d'insérer B de manière à minimiser le plus possible la taille de la tournée. On continue ensuite avec CDE et ainsi de suite. On fait cela sur chaque triplet puis on recommence. On s'arrête lorsqu'il n'y a plus d'insertions bénéfiques possible dans la tournée.

5.2. Inversion

Soit ABCDEFGA notre tournée. Pour l'inversion, on choisit un doublet AB, puis on itère sur les doublets qui suivent CD, DE, etc en cherchant le premier couple avec lequel l'inversion est bénéfique pour la tournée. On fait cela sur chaque couple puis on recommence. On s'arrête lorsqu'il n'y a plus d'insertion améliorante possible dans la tournée.

5.3. Remarques et idées d'améliorations

On remarque que la méthode de repositionnement est plus souvent efficace que la méthode d'inversion. Pour rendre la méthode d'inversion plus efficace, on pourrait ne pas choisir pour chaque couple le premier qui permet une inversion bénéfique mais plutôt celui qui permet la plus bénéfique à chaque fois. Cela permettrait sûrement à l'algorithme d'être moins long.

6. Conclusion

L'objectif de ce projet était la mise en place d'un planificateur de tournée efficace. Au cours des différentes phases, le système est devenu de plus en plus efficace, jusqu'à pouvoir fournir un chemin le plus optimal possible, sans prendre trop de temps.

Toutefois, sur des tournées longues de l'ordre de 1000 villes, l'algorithme prend plusieurs secondes : moins de 10 dans le cas d'un graphe complet mais souvent bien plus lorsque l'on passe à un graphe non complet et que l'on utilise la structure de graphe.

Ainsi, on pourrait trouver d'autres idées d'améliorations et d'approximation afin d'augmenter le plus possible le ratio précision/temps de calcul qui est important à contrôler dans ce genre de problème.