AUFFRAY Baptiste
DROMIGNY--CHEVREUIL Ivan

Rapport de Recherche Opérationnelle

Méthode de résolution exacte d'un problème de tournée de véhicule avec capacités et profits

2016-2017

I – Introduction

Membres scientifiques au sein de la Communauté Anarchiste Nationaliste Ancrer sur la Recherche et le Développement (alias C.A.N.A.R.D), il nous a été confié la mission d'implémenter un programme permettant d'organiser les tournées de récupération d'eau de nos drones, et ceux de la manière la plus optimisée possible. Afin d'atteindre les objectifs qui nous ont été confiés, nous avons divisé notre programme en trois grandes sous-parties :

- Pour une instance de lieux données, nous recherchons dans un premier temps tous les circuits possibles que notre drone peut effectuer en fonction de sa capacité de stockage.
- Dans un second temps, nous devons, pour chacun des circuits, trouver le chemin le plus court et ceux en passant par tous les lieux. Nous avons, pour ce point-ci implémenté trois méthodes de résolutions différentes.
- Pour finir, nous fournissons nos données précédemment calculées à un solveur (GLPK) qui résout automatiquement les différents circuits à effectuer et la taille du parcours total.

L'ensemble de notre programme a été effectué en C++, et nous faisons ainsi appelle au solveur GLPK en tant que bibliothèque de fonction afin d'aboutir à la résolution du problème.

Nous avons également effectué des jeux de données afin d'analyser nos différentes implémentations, et de déterminer laquelle employer en fonction du contexte pour obtenir un résultat optimal.

Pour finir, des propositions d'améliorations ont été fournis afin d'optimiser le temps d'exécution des programmes.

La compilation ainsi que l'exécution de notre programme s'effectuent au moyen de la commande suivante à effectuer dans un terminal :

- g++ -std=c++11 -o2 hpp-cpp/*.cpp -lglpk -lm -o projet && ./projet arg1 arg2 arg3

où arg1 : obligatoire, nom de l'instance numérique
arg2 : obligatoire, numéro du TSP que l'on veut utiliser pour la résolution (1, 2 ou 3)
arg3 : optionnel, une chaine de caractère quelconque suffit afin de lancer la fonction analyse (dans ce cas arg1 et arg2 ne seront pas utilisé donc l'on peut également leur attribuer une chaine de caractère quelconque)

II – Description des algorithmes

Dans les descriptions qui vont suivre afin de simplifier les descriptions des algorithmes, et d'alléger le texte, nous admettrons les postulats suivants :

- un vecteur de vecteur d'entier sera raccourcis en 'VVE'
- le nombre de lieux de l'instance sera abrégé en 'nbL'

II-1 – Structures de données

Dans le but d'optimiser et de simplifier notre implémentation, plusieurs structures de données ont été utilisées :

- Une structure, nommée *donnees*. Cette dernière contient :
  - 3 entiers : *nbLieux*, qui est le nombre de lieux dans l'instance (dont la base) ; *capacité*, qui est la capacité du drone ; *nbCirc*, qui est le nombre total de circuits possibles dans l'instance.
  - 2 vecteurs d'entiers : *quantite*, qui est la quantité d'eau présente à chaque lieu ; *tailleCirc*, qui est la distance minimum de chacun des circuits possibles.
  - 3 VVE : *distances*, qui contient les distances entre chaque lieu ; *circuit*, qui contient tous les circuits possibles ; *contr*, qui permet de générer par la suite la matrice des contraintes plus aisément.

- Une structure, nommée *voyage*. Cette dernière contient :
    - 3 entiers : *nbLieuxCirc*, qui correspond au nombre de lieux dans le circuit actuel ; *nbLieuxCircPow*, qui correspond à $2^{nbLieuxCirc}$; *circ*, qui permet d'accéder au distancier et ainsi d'effectuer les calculs nécessaires à la résolution du problème.
    - 2 VVE : *dist*, contenant les distances entre les différents lieux ; *parcours*, permettant de connaitre le dernier lieu visité.

- Plusieurs vecteurs ont été également utilisés un peu partout

Les différentes structures ont été implémentées afin de permettre le stockage au même endroit de plusieurs variables utiles à la résolution d'un problème commun à celle-ci. De plus cela permet de réutiliser directement la structure quand l'utilisation de toutes ces variables se fait sentir et ainsi d'éviter l'initialisation de toutes ces dernières.

Par ailleurs, nous avons favorisé l'utilisation des vecteurs quand cela était possible. En effet, ces derniers se trouvent êtres très utiles. L'allocation dynamique, et les différentes fonctions (inhérente ou non à la structure) se trouvent êtres d'une utilité et d'une efficacité remarquable. Nous pouvons ici prendre comme exemple les fonction *vector.front()* permettant d'accéder au premier élément du vecteur quel que soit sa taille de celui-ci ; *vector.size()* permettant d'obtenir le nombre d'élément dans le vecteur ; ou encore le next_permutation(*vector.begin(), vector.end())* dont l'explication du fonctionnement sera fournis par la suite.

II-2 – Énumération des regroupements

Afin de rechercher tous les regroupements possibles, nous avions dans un premier temps implémenté une fonction en Brute Force. Nous avions un tableau de booléen ayant pour taille le nombre de lieux (excepté la base). Au moyen d'une boucle allant de 0 à 2^{nbL}, on effectuait tous les regroupements possibles en modifiant les valeurs du tableau de booléen à chaque fin de tour de boucle. Une telle méthode se trouvait avoir une complexité temporelle exponentielle, ce qui n'est pas optimisé. Nous avons décidé de modifier notre algorithme.

Pour le nouvel algorithme, on crée un vecteur vide qu'on "push_back()" dans notre VVE final. Ce dernier est le VVE d'entier nommé *circuit* présent dans la structure *donnees*. Puis, on test tous les circuits possibles au moyen de 3 boucles : la première va pour *i* de 0 à nbL (toujours sans compter la base). Dans cette première boucle on crée un VVE temporaire qui devient égal au VVE final. Au moyen d'une boucle pour *j* de 0 à la taille du VVE temporaire, on "push_back()" dans ce dernier pour chacun de ses vecteurs d'entier le lieu en cours (itérateur *i*). Une fois cela effectué, on a créé tous les nouveaux circuits possibles contenants le lieu *i*, donc au moyen d'une seconde boucle pour *j* allant de 0 à la taille du VVE temporaire, on test si la quantité d'eau totale de chacun des circuits est inférieure ou égale à la capacité du drone. Si c'est le cas on rajoute le circuit (donc le vecteur d'entier) dans le VVE final.

Ainsi tous les circuits possibles sont testés, et dans le cas d'un circuit impossible, on évite de tester tous les circuits suivant existants (si 1-2-3 impossible : 1-2-3-4, 1-2-3-5 etc. ne seront pas testés) ce qui réduit grandement la complexité.

On supprime le tout premier vecteur d'entier du VVE final, car celui-ci est vide et est donc inutile. La variable *nbCirc* de la structure *donnees* devient le nombre de vecteur d'entier de notre VVE final. Par la suite, à chacun de ces circuits on ajoute le nombre 0 (la base), et on utilise un algorithme de *sort* afin de trier les circuits dans l'ordre croissant (cela est par la suite utile pour l'algorithme du NUS d'avoir le 0 en première position des vecteurs). Finalement, on parcourt le VVE final afin de créer le VVE *contr* présent dans la structure *donnees*, ce dernier permettant par la suite de créer la matrice creuse des contraintes que l'on fournit à GLPK pour la résolution du problème principal.

Cet algorithme nous permet donc en un temps optimal de déterminer tous les circuits existants, le nombre de ces circuits ainsi que la future matrice des contraintes.

II-3 – Énumération des tournées (regrouper)

Une fois tous les circuits possibles calculés, nous devions pour chacun d'entre eux calculer le plus court chemin possible en passant par tous leurs points respectifs. Ce problème est plus communément appelé le Problème du Voyageur de Commerce (Travelling Salesman Problem en anglais, abrégé en TSP). Afin de résoudre ce problème nous avons effectué plusieurs méthodes de résolution :
- un algorithme en Brute Force
- un algorithme en Programmation Dynamique
- une résolution grâce à GLPK.

II-3.1 Brute Force (TSP1)

Ayant l'habitude de faire de la programmation impérative, la première idée qui nous est venu à l'esprit etait une résolution algorithmique du problème.

La manière la plus simple afin de résoudre ce problème est d'essayer tous les chemins possibles (ce qui correspond à toutes les permutations possibles du chemin, donc *nbL!* tests). Coup de chance, il existe en C++ une fonction permettant de faire automatiquement une permutation sur un vecteur : *next_permutation(Iterator first, Iterator last)*, contenu dans la librairie <*algorithm*>. Cette fonction va réarranger les éléments du vecteur entre *first* et *last* dans le prochain plus grand ordre lexicographique. Il suffit donc d'effectuer *nbL!* fois la fonction *next_permutation()* et de calculer la somme du parcours à chaque itération. On stocke dans une variable la plus petite valeur trouvée parmi toutes ces permutations et on la retourne une fois que toutes ces dernières ont été testées.

Nous avons ici une approche de résolution naïve, qui s'avère être très peu efficace lorsque le nombre de lieux dans le circuit augmente. Nous avons toutefois tenté d'optimiser quelques peu cet algorithme. Pour expliquer cela, prenons un exemple :

Nous disposons des lieux 0, 1, 2 et 3, et le point 0 est le point de départ (et donc d'arrivée). *next_permutation()* va nous fournir les permutations suivantes :

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
Arrêt

(le passage de la base vers le premier lieux et du dernier vers la base est ici implicite, nous ne prenons donc pas la peine de l'afficher)

On sait qu'un parcours et son inverse font exactement la même distance (1→2→3 = 3→2→1) car nous utilisons une matrice des distances symétriques. Donc une fois que nous avons effectué *(nbL-1)!* permutations, nous avons effectué toutes les permutations commençant par un des nombres du circuit. Une fois ces permutations et les calculs de distances effectués on peut donc être certain, que tous les autres parcours finissant par ce nombre ont déjà été calculés. On ajoute donc ce nombre dans un tableau, et ainsi à chaque prochaine permutation on teste si un des nombres contenus dans le tableau est le dernier du parcours, si c'est le cas, on ne prend pas la peine de calculer la distance et on passe directement à la permutation suivante.

1 2 3

1 2 3
1 3 2

1 2 3
1 3 2
2 1 3

1 2 3
1 3 2
2 1 3
~~2 3 1~~

1 2 3
1 3 2
2 1 3
~~2 3 1~~
~~3 1 2~~
3 2 1

1 2 3
1 3 2
2 1 3
~~2 3 1~~
~~3 1 2~~
~~3 2 1~~
Arrêt

(Les sous-circuits barrés sont les sous-circuits non testés)

Malgré cette optimisation la fonction reste très peu efficace et nous nous trouvons avec une complexité en temps qui se trouve être exponentielle. Nous avons donc effectué des recherches, et avons décidé de trouver une nouvelle approche à la résolution de ce problème.

II-3.2 Branch and Bound (TSP2, TSPfromNUS)

Le crédit de l'implémentation de cet algorithme ne nous revient absolument pas. Il a été implémenté par « National University of Singapore ». Cet algorithme utilise le principe de programmation dynamique afin de résoudre le problème. Nous avons fourni en même temps que ce rapport et l'implémentation de notre programme le fichier pdf source duquel nous tirons ce programme. Bien que nous n'ayons pas nous même implémenté cet algorithme, et que nous ayons encore une zone de flou en ce qui concerne son fonctionnement, nous avons tous de même dans l'ensemble compris son implémentation, et nous trouvions cela dommage de ne pas l'utiliser afin de montrer les différences de complexité avec la résolution Brute Force et la résolution GLPK. De plus nous avons tout de même modifié l'algorithme afin que celui-ci s'adapte à notre propre programme. Un travail non négligeable a donc été effectué sur cet algorithme.

Dans un premier temps une explication de ce qu'est la programmation dynamique s'impose :

C'est un paradigme de programmation, qui est avant tout utilisé afin de résoudre les problèmes d'optimisation. Cela revient à décomposer son problème en sous-problèmes, ces mêmes sous-problèmes en sous-problèmes etc. On résolve ensuite les problèmes des plus petits aux plus grands en stockant les différents résultats intermédiaires afin de parvenir à une solution optimale. Ce type de programmation s'appuie sur le principe d'optimalité de Bellman : qui est que toute solution optimale est constituée de sous-problèmes eux-mêmes résolus de manière optimale. La résolution de manière optimale de ces sous-problèmes, et le stockage progressif des réponses évitent les recalculs, ce qui nous conduit donc à une solution optimale pour notre problème et ceux de manière efficace.

Dans un second temps, une explication de ce qu'est un algorithme par *séparation et évaluation* (branch and bound en anglais) ne serait pas de trop :

C'est une méthode générique de résolution de problème d'optimisation combinatoire, qui consiste à trouver un point minimisant une fonction, appelée coût, dans un ensemble dénombrable. Cette méthode se divise en deux sous-partie, la séparation et l'évaluation:

- Séparation : cette phase consiste en la division en sous-problèmes du problème principal, ayant chacun leur ensemble de solutions réalisables. En résolvant tous les sous-problèmes et en sélectionnant la meilleure solution trouvée, on finit par s'assurer de résoudre le problème principal. On applique principalement ce principe de manière récursive à tous les sous-ensembles de solutions obtenues, et ceux tant que des ensembles contiennent plusieurs solutions. Cela revient en la construction d'un arbre de recherche/décision, car l'ensemble des solutions s'organise selon une hiérarchie naturelle en arbre.

- Évaluation : Cette étape consiste en l'évaluation d'un nœud de l'arbre, et a pour objectif de déterminer l'optimum de l'ensemble des solutions réalisables associé au nœud en question, ou qu'il n'existe aucune solution optimale. Si un nœud ne contenant pas de solution optimale est détecté, on ne prend pas la peine d'effectuer la séparation de son espace de solution.

Revenons maintenant à notre algorithme. Afin de le faire fonctionner plus aisément, nous avons rajouté une structure appelé *voyage*, permettant de stocker les informations propres à l'algorithme. Cette structure est composée de deux VVE. L'un permet de stocker les distances des différents nœuds de l'arbre (*dist)*, tandis que l'autre permet de stocker le dernier lieu visiter (*parcours*). Elle contient également 3 entiers, l'un permet de connaître le nombre de lieux dans le circuit, l'autre permet de connaître la valeur de 2^{nbL}, et le dernier permet de savoir dans quel circuit on se trouve actuellement afin d'accéder aux valeurs du distancié plus aisément.

Déroulement de l'algorithme :

- TSP2(donnees &p) : Dans un premier temps, on instancie une structure *voyage* dans laquelle on entre les informations nécessaires (nombre de lieux, le circuit dans lequel on est, initialisation des VVE à -1, *nbCirclePow* qui correspond à $2^{nbLCirc}$). On exécute ensuite la fonction TSPfromNUS(0, *nbCirclePow*-2, *v*, *p)* et on stocke le résultat dans le vecteur de taille des circuits (*tailleCirc* de la structure *donnees*). Cette opération est ainsi répétée pour tous les circuits.

- TSPfromNUS(int start, int set, voyage &v, donnees &p) : Dans un premier temps on vérifie si la case sur laquelle on est situé (avec pour ordonnée le paramètre *start* et pour abscisse le paramètre *set*) est différente de -1. Si c'est le cas, c'est qu'on est situé sur un nœud dont on connaît déjà la distance, donc on retourne cette dernière.

Sinon pour chacun des lieux du circuit, un *mask* est calculé en fonction de la valeur de *i* (l'itérateur de la boucle actuelle) et de la valeur *nbLieuxCircPow*. Puis on calcule *masked* en effectuant un 'et bit à bit' entre *mask* et *set*.

C'est ce point précis que nous n'arrivons justement pas à cerner. Par la suite, si *masked* est différent de *set*, on calcule une valeur temporaire qui est la distance entre le lieu *start* et le lieu *i,* à laquelle on additionne TSPfromNUS(i, masked, &v, &p) (exécution récursive de la fonction avec pour point de départ *i*). Cette valeur temporaire est ensuite stockée dans le tableau *dist* et la valeur de *i* est stockée dans le tableau *parcours* (pour les deux tableaux cette valeur est stockée aux coordonnées (*start,set*)).

Ce que nous n'arrivons pas à comprendre est comment cette opération (le 'et bit à bit') permet de déterminer si oui ou non le nœud actuel est un nœud viable à la résolution du problème (car il est bien ici question de cela) ou si l'on doit passer au nœud suivant.

Par ailleurs, nous avons remarqué que pour les deux tableaux *dist* et *parcours* seul les colonnes à coordonnées pairs sont utilisées. En effet, les colonnes impaires sont constamment à -1. Nous pensons que cela est dû au fait de l'opération 'et bit à bit', qui ne retourne que des nombres pairs. L'accès au tableau à des coordonnées en abscisse pairs se trouve ainsi plus efficace et facile que l'accès à des coordonnées en abscisse impairs.

Cet algorithme se trouve être beaucoup plus optimisé que le précédent pour les instances de grandes tailles. La complexité est ici plus dure à déterminer que l'algorithme précédent, mais selon la page wikipédia du Voyageur de Commerce, « Held et Karp ont montré que la programmation dynamique permettait de résoudre le problème en $O(n^2 2^n)$ ». On peut donc supposer que la complexité de cet algorithme se rapproche grandement de celle énoncée précédemment. De plus, les différents tests effectué (cf. III_-_Analyse_des_résultats) vérifie cette supposition.

II-3.3 GLPK (TSP3, voyageur, resolve_voyageur)

Afin d'avoir un panel de résolutions plus étendu, nous avons décidé d'implémenter une résolution du voyageur de commerce grâce à la bibliothèque C, GLPK. Nous pourrons alors comparer la vitesse de cette résolution avec les autres.

Pour effectuer une résolution en GLPK, nous devons d'abord poser un programme linéaire qui résout le problème du voyageur de commerce. Nous reprenons dans notre cas le programme linéaire qui nous a été donné lors des cours magistraux de Recherche Opérationnelle.

Variables de décision

$$x_{ij} = \begin{cases} 1 & \text{si on va de la ville } i \text{ à la ville } j \\ 0 & \text{sinon} \end{cases}$$

$$(i, j \in \{1, \ldots, n\})$$

$$
\begin{aligned}
\min\ z \quad &= \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\
s.c. \quad &\sum_{j=1}^{n} x_{ij} \quad = \quad 1 \qquad \forall i \in \{1,\ldots,n\} \\
&\sum_{i=1}^{n} x_{ij} \quad = \quad 1 \qquad \forall j \in \{1,\ldots,n\} \\
&\sum_{i,j\in S} x_{ij} \quad \leq \quad |S|-1 \quad \forall S \text{ avec } |S| \leq n-1 \\
&x_{ij} \in \{0,1\} \qquad\qquad \forall i,j \in \{1,\ldots,n\}
\end{aligned}
$$

où les coûts c_{ij} sont non-négatifs

Pour résoudre nous posons des variables de décisions binaires *x* qui correspondent au fait d'aller d'un lieu à un autre. Elles représentent donc tous les chemins possibles depuis un lieu vers un autre. On obtient alors nbL^2 variables binaires.

Sur la représentation ci-dessus, les variables sont indexées par i et j, i correspond au lieu de départ et j au lieu d'arrivée.

On pose ensuite une fonction objective, dans notre cas on souhaite minimiser la distance parcourue pour explorer chacun des lieux. Pour cela on additionne toutes les variables *x* multipliées avec les distances qui leurs correspondent. Ainsi, seule les distances pour lesquels les *x* égaux à 1 seront conservées dans l'opération, autrement dit on additionne toutes les distances des chemins parcourues.

Il faut ensuite ajouter des contraintes sur nos variables de décision, ceci permet de donner une certaine marche à suivre au solveur GLPK afin de d'obtenir une solution qui corresponde à nos attentes.

Dans un TSP, nous souhaitons parcourir tous les lieux une seule fois. Sachant que les variables *x* représentent les chemins entre deux lieux, on peut alors se dire qu'on souhaite arriver dans chaque lieu une seule fois et qu'on souhaite partir de chaque lieu une seule fois.

Tout d'abord, pour représenter les contraintes « d'arriver une fois dans un lieu », on va additionner tous les *x* qui représentent les chemins arrivant dans un même lieu et on va poser cette addition comme égale à 1. On effectue cela pour chacun des lieux, ce qui nous donne *nbL* contraintes.

Ensuite on veut représenter les contraintes « partir d'un lieu une seule fois ». Ces contraintes sont très similaires aux précédentes, il faut additionner tous les *x* qui partent d'un même lieu et poser cette addition à 1. On répète cette contrainte pour chaque lieu et on obtient *nbL* contraintes supplémentaires. Nous avons donc déjà 2**nbL* contraintes mais celles-ci ne sont pas encore suffisantes pour résoudre correctement notre problème.

En effet, si on lance la résolution comme ceci, on obtiendrait plusieurs petits chemins qui boucleraient entre eux. Il faut donc ajouter des contraintes qui empêchent les sous-boucles dans le parcours des lieux. Cette contrainte revient à additionner les chemins mis en cause dans une sous-boucle possible et d'indiquer que cette addition doit être strictement inférieure au nombre de lieux qui entrent en jeu dans cette sous-boucle. Ainsi tous les *x* de la possible sous-boucle ne sont pas tous égaux à 1 et elle ne peut pas se former.

Ceci donne un nombre très important de contraintes. Ce nombre correspond au nombre de partitions d'un ensemble (nombre de Bell) auquel on ajoute le nombre de lieux qu'on doit explorer et auquel on retire 2 (pour la partition qui contient tous les lieux et celle qui contient toutes les partitions de 1). Ceci nous donne donc B(*nbL*)+*nbL*-2 contraintes.

Dans un premier temps nous trouvions très compliqué de calculer toutes les sous-boucles possibles d'un ensemble. On nous a alors conseillé de ne pas ajouter ses contraintes dans un premier temps, d'exécuter la résolution, et dans le cas d'un passage par une sous-boucle, de rajouter la contrainte correspondante et de relancer la résolution. Nous pensions que cette méthode serait plus optimisée, c'est pour cela que nous avons opté pour celle-ci.

- Explication de l'implémentation de GLPK :

Tout d'abord, nous devons définir notre problème, nous indiquons qu'on souhaite minimiser une valeur et nous initialisons le nombre de contraintes que nous allons ajouter à la matrice des contraintes et on indique qu'on souhaite que ces contraintes soit égales à 1. On va ensuite définir le nombre de variables de décisions dont nous avons besoin pour ce problème ainsi que leurs types. Ici, on a nbL^2 variables binaires comprissent entre 0 et 1. On ajoute ensuite le coefficient qui correspond à chaque variable, c'est-à-dire la distance entre deux lieux.

On va maintenant ajouter les 2**nbL* premières contraintes, celles sur le fait d'arriver une fois dans un lieu et d'en partir une seule fois. Pour cela on va utiliser trois tableaux. Pour le même indice dans ces tableaux, le premier va contenir la position des variables de décision qui entre en jeu dans la contrainte. Le deuxième va contenir le numéro de la contrainte à laquelle appartient cette variable de décision. Puis le troisième contient la valeur du coefficient avec lequel on souhaite multiplier la variable de décision en question, ici ça sera toujours 1.

Maintenant nous n'allons pas ajouter les contraintes sur les sous-boucles, nous allons lancer la résolution du simplexe comme ceci. Une fois notre résolution terminée, nous allons récupérer les variables de décisions qui sont à 1, c'est-à-dire les chemins qui sont parcourues. Nous analysons ce résultat afin de déterminer s'il n'y pas de sous-boucles et si une est détectée, nous allons ajouter à notre problème une contrainte qui permet de « briser » cette sous-boucle. Puis on réitère cette étape jusqu'à ce qu'il n'y ait plus de sous-boucles, et qu'on obtienne le résultat attendu.

Une fois ceci mis en place, on a juste à appliquer cet algorithme pour chaque regroupement calculé précédemment.

Cette solution peut paraître plutôt pratique dans un premier temps, puisque toutes les contraintes sur les sous-boucles qu'on ajoute son utile à la résolution du problème. Ceci ne nous oblige pas à ajouter un nombre extrêmement grand de contraintes. Malheureusement ceci nous oblige à relancer plusieurs fois l'algorithme du simplexe et calculer à chaque fois s'il y a une sous-boucle, ce qui au final prend beaucoup de temps.

En effet pour une instance de taille 25 de A, il faut 54 secondes pour calculer tous les plus courts chemins pour chaque regroupement. Mais pour une instance de taille 30 de A, il faut un peu plus de 20 minutes pour calculer le plus court chemin de tous les regroupements. Le temps que met cet algorithme pour résoudre le problème nous a alors semblé extrêmement long. Surtout qu'en faisant quelque recherche sur le problème de voyageur de commerce sur internet, on apprend que : « Les méthodes d'optimisation linéaire sont à ce jour parmi les plus efficaces pour la résolution du problème de voyageur de commerce et permettent désormais de résoudre des problèmes de grande taille (à l'échelle d'un pays). » (Wikipédia).

A la lueur de ces mots, on comprend que la méthode que nous avons choisie n'est pas là plus optimiser, loin de là, surtout par rapport aux temps d'exécution des autres voyageurs de commerce que nous avons implémenté.

Nous avons alors décidé d'implémenter un voyageur de commerce en GLPK et en initialisant dès le début toutes les contraintes sur les sous-boucles. Pour cela nous initialisons notre problème comme précédemment jusqu'aux contraintes sur l'arrivée et le départ d'un lieu.

Maintenant, il nous faut calculer toutes les sous-boucles possibles afin de pouvoir les ajouter aux contraintes. Pour cela, nous devons déterminer les sous-regroupements des lieux possibles, on utilise alors la fonction *regrouper_sc(vector<int>)* qui va faire les mêmes regroupements que la fonction *regrouper(donnees &p)* mais sans supprimer les circuits qui dépassent la limite de stockage du drone. Ainsi nous obtenons un tableau qui contient tous les sous-regroupements.

Ensuite, à partir de ces regroupements, nous devons déterminer les sous-boucles possibles. Afin de réaliser cela, nous devons à partir de chaque paire d'éléments contigus de l'ensemble d'un regroupement déterminer les variables de décisions qu'on va utiliser dans la contrainte qu'on va ajouter. Puis on répète cette opération pour les (*nbLCirc*-1)! prochaines permutations avec la fonction next_permutation() du sous-regroupement en cours d'analyse.

Pour finir, nous n'avons plus qu'à lancer la résolution du problème. On observe alors une réelle amélioration pour le temps d'exécution de notre problème. En effet pour une instance de 25 de A, il ne faut plus

que 1,01 secondes au lieu de 54 secondes précédemment. Puis seulement 8,03 secondes pour l'instance de 30 de A au lieu de 20 minutes précédemment, la différence est extrêmement importante.

Ceci nous confirme que notre première idée était complètement fausse, que le fait d'ajouter les contraintes des sous-boucles petit à petit prenait trop de temps par rapport au calcul des toutes les sous-boucles possibles.

Malgré les améliorations que nous avons apportées à TSP3 par rapport à la première version, son temps d'exécution reste toujours plus élevé que TSP1 et TSP2. On pense que ceci est dû au fait que le TSP3 doit effectuer le calcul de toutes les sous-boucles possibles par rapport aux autres algorithmes. Ce qui reste un temps de calcul non négligeable, cela représente tout de même B(*nbL*)+*nbL*-2 sous-boucles à calculer, avec le nombre de Bell qui augmente de façon exponentielle. Ainsi plus le nombre de ville augmente plus son temps d'exécution augmente. C'est pour cela que dans un premier temps on pensait que le fait d'ajouter juste les contraintes de sous-boucle active serait plus rapide. Mais au final, la ré-exécution multiple du simplexe prend un temps très important.

Ainsi pour améliorer TSP3 il faudrait trouver une solution plus simple et plus rapide pour déterminer les sous-boucles, une solution que nous n'avons malheureusement pas trouvée.

Sachant que l'optimisation linéaire serait la méthode de résolution du TSP la plus efficace, ceci nous pousse à penser qu'il existe une méthode de résolution par GLPK plus rapide. Même si celle que nous avons implémenté s'exécute dans des temps raisonnables.

III – Analyse des résultats

Afin d'analyser et de comparer nos différentes implémentations, nous avons, au moyen de la fonction *analyse()* que nous avons implémenté et de « gnuplot », réalisé des diagrammes. Ces derniers ont tous été mis en annexes.

Dans un premier temps la comparaison du nombre de circuits entre les instances du dossier A et les instances du dossier B nous montre que celles du dossier B contiennent beaucoup plus de circuits que pour celles de A (cf. Figure_7-Figure_8). Cela est dû au fait que pour les instances de B, le drone détient une capacité de stockage un peu supérieur à celles de A (30 pour les A contre 33 pour les B). De plus les lieux des instances de B détiennent une quantité d'eau en moyenne plus basse que pour A.

Ainsi, on remarque pour la fonction *regrouper(donnees &p)* (cf. Figure_3), que le nombre de lieux n'a pas vraiment d'incidence sur le temps d'exécution de la fonction. Le facteur majeur se trouve être le nombre de circuits possibles. Cela semble cohérent vis à vis de l'implémentation de la fonction, car cette dernière a été implémenté de manière à éviter le calcul des circuits impossibles.

En ce qui concerne l'instance A, on observe que quel que soit le nombre de lieux, le TSP1 est toujours le plus optimisé (cf. Figure_1). Du fait de l'explication ci-dessus (peu de circuits possibles pour les instances de A), et du fait que la complexité de l'algorithme est en O(*nbL!*), la résolution du TSP1 s'exécute plus rapidement que pour le TSP2 ou le TSP3.

En revanche, on remarque que pour les instances de B (cf. Figure_2), le TSP2 est le plus adéquate afin de résoudre le problème. Cela vient du fait que les instances du dossier B (cf. Figure_8) contiennent d'une part beaucoup plus de circuits possibles, mais surtout de beaucoup plus de circuits contenant beaucoup de lieux (pour une instance de 10 lieux nous avons 34 circuits de taille 4 pour A contre 34 de taille 6 pour B).

On peut donc en déduire que pour des instances contenants peu de lieux, ainsi que des circuits de petites tailles, TSP1 est plus adéquate, tandis que pour la résolution d'instances contenant une grande quantité de lieux et des circuits de grandes tailles, TSP2 est plus optimisé. Il existe donc un point limite à partir duquel TSP2 sera plus adéquate à la résolution du problème que TSP1.

La comparaison de TSP en fonction des instances du dossier A ou B (cf. Figure_4 - Figure_5 - Figure_6) nous montre que pour les trois algorithmes, le facteur majeur se trouve être le même que celui de la fonction *regrouper(donnees &p)*. En effet le nombre de lieux n'est pas un facteur majeur du temps d'exécution des fonctions, qui se trouve en réalité être le nombre de circuits possibles.

Une fois la fonction *analyse()* exécuté, et les informations nécessaires récupérés, nous nous sommes rendu compte que nous ne calculions pas le temps total de résolution du problème. Ne voulant pas complètement modifier notre fonction *analyse()* et surtout ne pas à avoir à réexécuter la fonction pour toutes les instances (et donc devoir attendre l'exécution de chaque fonction indépendamment plus l'exécution de la résolution complète de chaque problème), nous avons décidé d'exécuter la résolution d'une partie des problèmes au moyen du *main(int argc, char *argv[])* et ainsi de stocker les résultats obtenus pour les analyser (cf. Figure_9).

Ainsi, on remarque que pour les instances ayant peu de circuits possibles pour la résolution du problème (instance A), c'est la résolution finale par GLPK (colonne *diff*) qui va occuper la majorité du temps. Cela n'est toutefois pas vrai pour la résolution par TSP3. Dans ce cas-là, c'est justement la résolution du TSP (via GLPK) qui va occuper la majorité du temps.

On remarque que pour des instances ayant beaucoup de circuits possibles (instance B), c'est également la résolution finale qui va prendre le plus de temps. Ceci n'est toutefois pas vrai pour l'instance de taille 30, mais cela provient du fait de la complexité de TSP1 qui se trouve être exponentielle.

On peut donc en conclure que plus le nombre de circuits est important, et plus la résolution finale par GLPK sera longue. Cela semble logique, car plus nous avons de circuits, et plus nous avons de variables qui rentrent en compte dans la résolution, il est donc normal que GLPK soit plus lent à la résolution du problème dans ces cas-là.

GLPK prend donc une part non négligeable du temps de résolution du problème, et cela peut également expliquer pourquoi la résolution du TSP via GLPK est nettement plus lente qu'avec les autres algorithmes.

Par ailleurs, nous avons décidé de ne pas exécuter le calcul du temps d'exécution de TSP3 pour les instances de B contenant plus de 25 lieux. En effet, le temps d'exécution de TSP3 pour l'instance de B de taille 20 étant déjà de 3h, et les instances supérieures à 25 contenant significativement plus de circuits de grandes tailles que cette dernière, il aurait sans doute fallu attendre plusieurs jours avant la fin de l'exécution du programme pour les dernières instances. Cela ne se serait traduit qu'en une perte de temps conséquente, car les données que nous avons obtenu jusqu'à présent ont été amplement suffisants à l'analyse de l'implémentation des différentes fonctions.

IV – Améliorations possibles

En ce qui concerne la résolution du TSP3, nous n'avons, comme expliqué dans le chapitre correspondant, trouvé aucunes améliorations susceptibles d'améliorer l'exécution de la résolution.

Par ailleurs, en ce qui concerne TSP1 et TSP2, on pourrait, en fonction du nombre de circuits possibles pour l'instance, tenter de calculer le point limite auquel TSP2 devient plus efficace que TSP1, et ainsi exécuter l'un ou l'autre en fonction de la situation. Cela permettrait d'optimiser quelques peu la résolution, mais le gain de temps ne serait pas conséquent. En effet le point limite d'optimalité entre TSP1 et TSP2 est situé à une échelle peu élevé et, de plus, la résolution par TSP2 est plus lente, de très peu, que celle de TSP1 sur les instances ayant peu de circuits.

V – Conclusion

Après près de trois semaines de recherches, notre groupe de chercheurs de la communauté C.A.N.A.R.D a pu mettre au point un système permettant de calculer le chemin que doit emprunter notre drone afin de récupérer un maximum d'eau. La rentabilité du C.A.N.A.R.D point de vue production d'eau a permis un développement sans précédent, une meilleure hygiène de vie et la mise en place de dérogations sur le contrôle de la démographie. Certains couples ont alors pu avoir plus de deux enfants, chose qui n'est pas arrivée depuis près de 10 ans dans notre communauté.

Nos recherches ont permis de mettre au point trois systèmes de calcules différents qui s'adaptent mieux selon certaines situations, bien que des optimisations sont sans aucun doute encore possible pour certains. Par

exemple, le système utilisant TSP1 sera plus adapté pour des parcours avec peu de lieux à visiter. Tandis que pour un parcours avec un grand nombre de lieux, TSP2 le sera plus.

Malheureusement, ce système peut prendre un temps de calcule très long si on souhaite visiter énormément de lieux. En effet, selon le nombre de lieux, le temps de calcule est exponentiel. Ainsi dès l'arrivée de la pluie, nous lançons les calculs sur de petits regroupements de lieux. Ceci nous permet en général à la fin des averses, d'obtenir un chemin optimal. Nous utilisons notre système sur un très grand nombre de lieux lorsque les prévisions météo du lendemain annoncent d'importantes intempéries.

Les résultats étaient probants, mais nous remarquions que les écarts entre les attentes d'apport d'eau et la quantité réelle apportée par les drones étaient importants. Trop important pour que ces écarts proviennent de l'incertitude sur la quantité disponible aux points de pompages. Cela ne signifiait qu'une chose, certains points de pompages étaient déjà visités par les drones des communautés voisines. En effet nous n'avions pas pensé dans nos différentes méthodes de calcul à prendre en compte les tournées des drones voisins.

Notre communauté, préférant utiliser le savoir que les armes, a décidé de mettre notre équipe de recherche en poste pour la conception future d'un système amélioré.

VI – Références bibliographiques

- https://fr.wikipedia.org/wiki/Partition_d%27un_ensemble
- http://kam.mff.cuni.cz/~elias/glpk.pdf
- https://fr.wikipedia.org/wiki/Nombre_de_Bell
- http://villemin.gerard.free.fr/aNombre/TYPDENOM/Bell.htm#liste
- https://github.com/evandrix/SPOJ/blob/master/DP_Main112/Solving-Traveling-Salesman-Problem-by-Dynamic-Programming-Approach-in-Java.pdf (le pdf a également été fournis)
- https://fr.wikipedia.org/wiki/Programmation_dynamique
- https://fr.wikipedia.org/wiki/S%C3%A9paration_et_%C3%A9valuation
- Recherche opérationnelle (X6I0030) : - CM Algorithme du simplexe
  - TP3: Utilisation de GLPK en tant que bibliothèque de fonctions

VI – Annexes

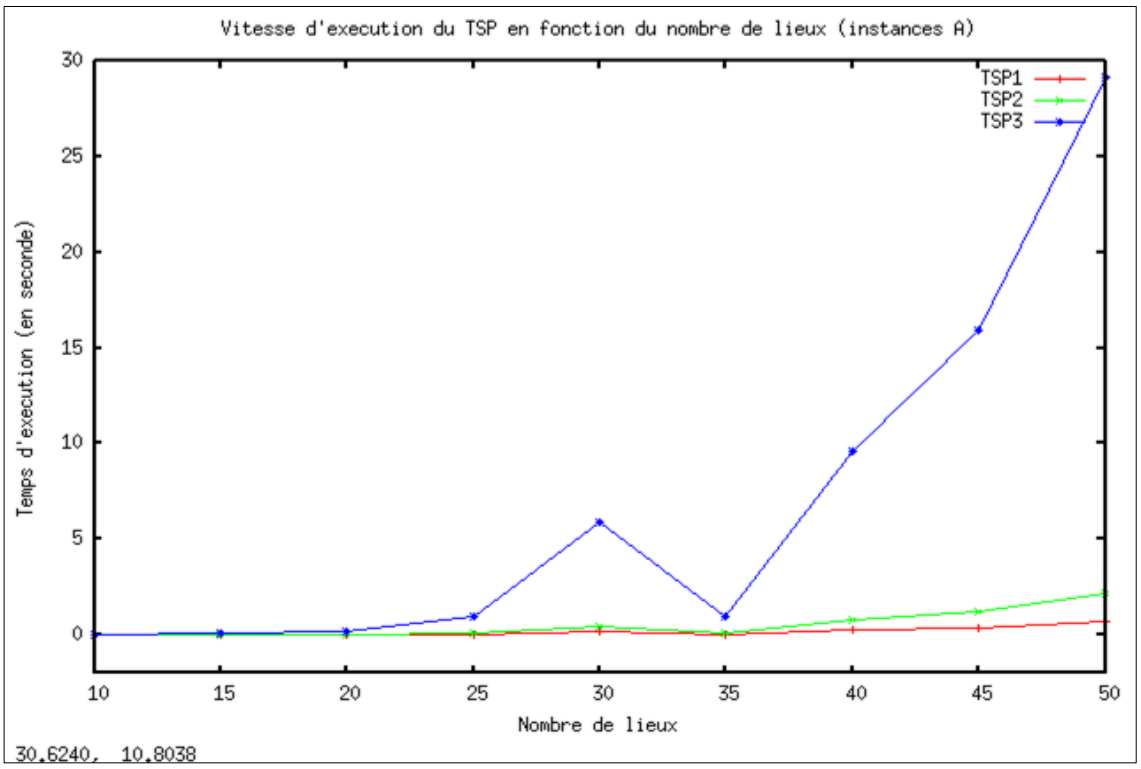

Figure I

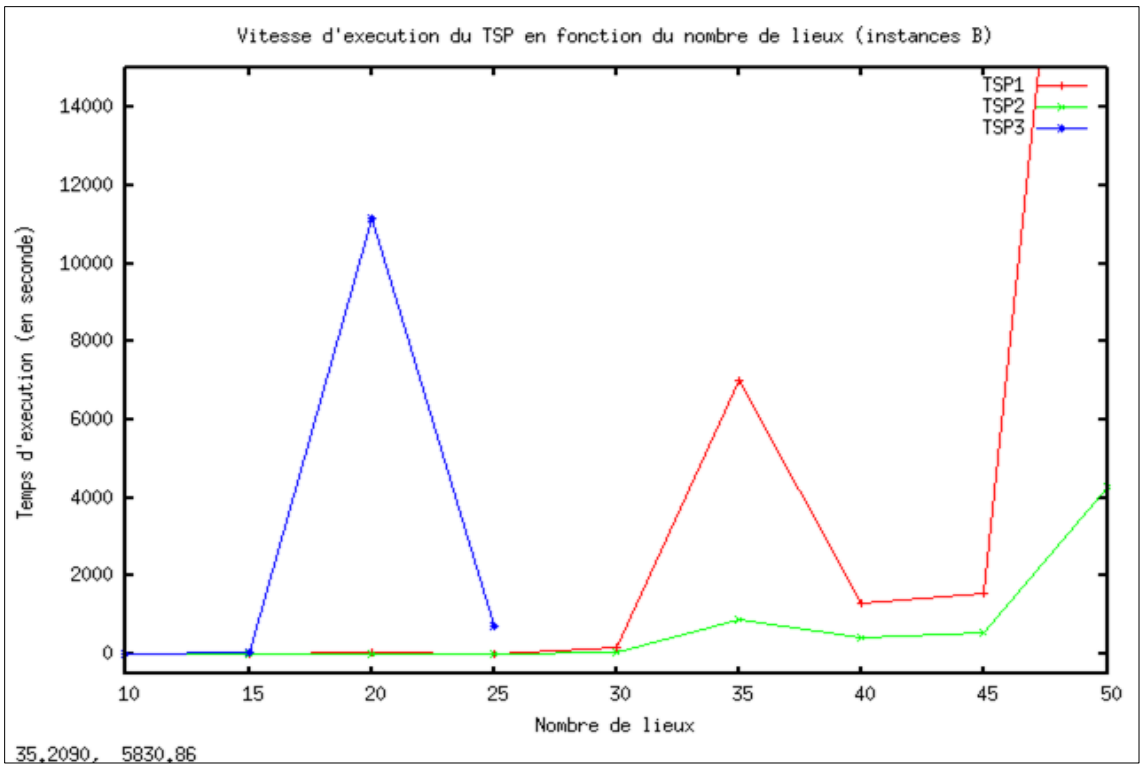

Figure 2

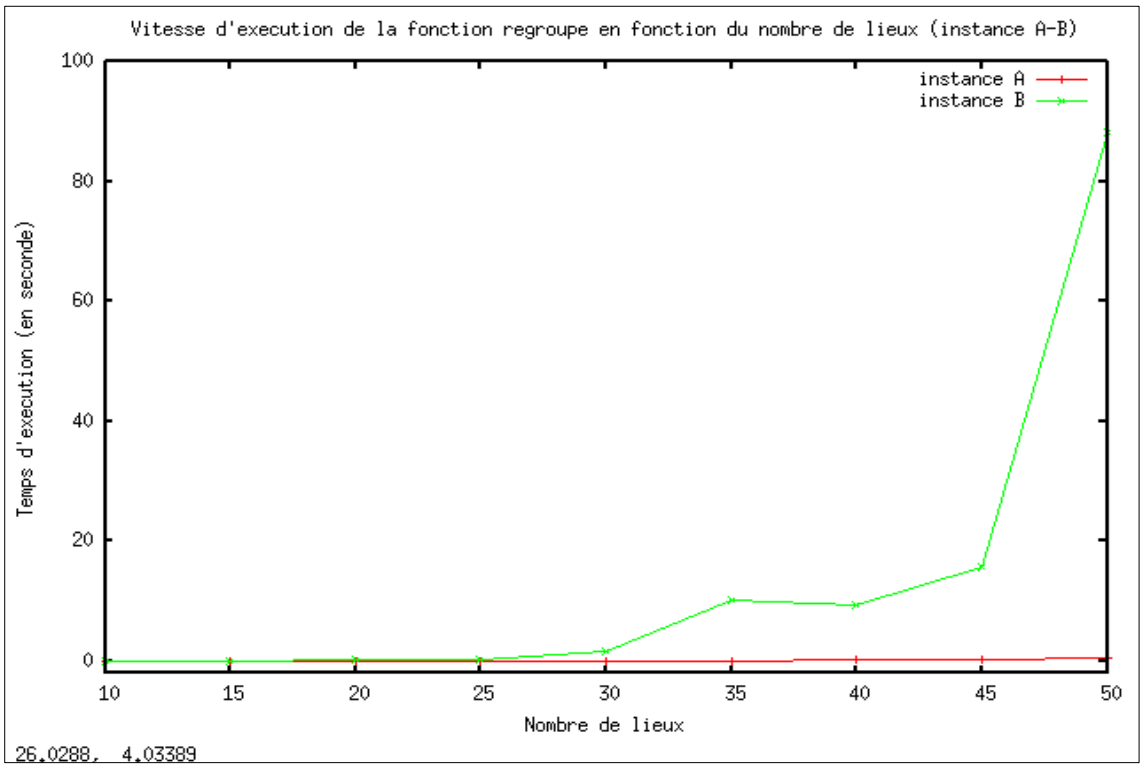

Figure 3

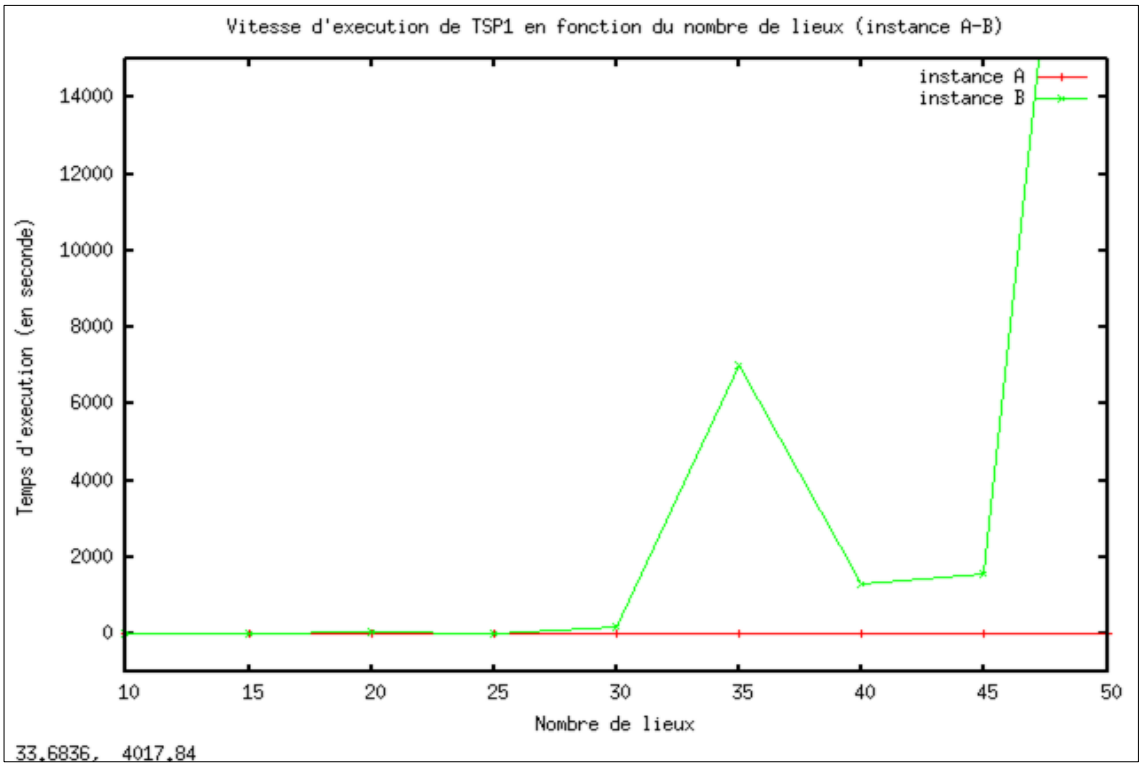

Figure 4

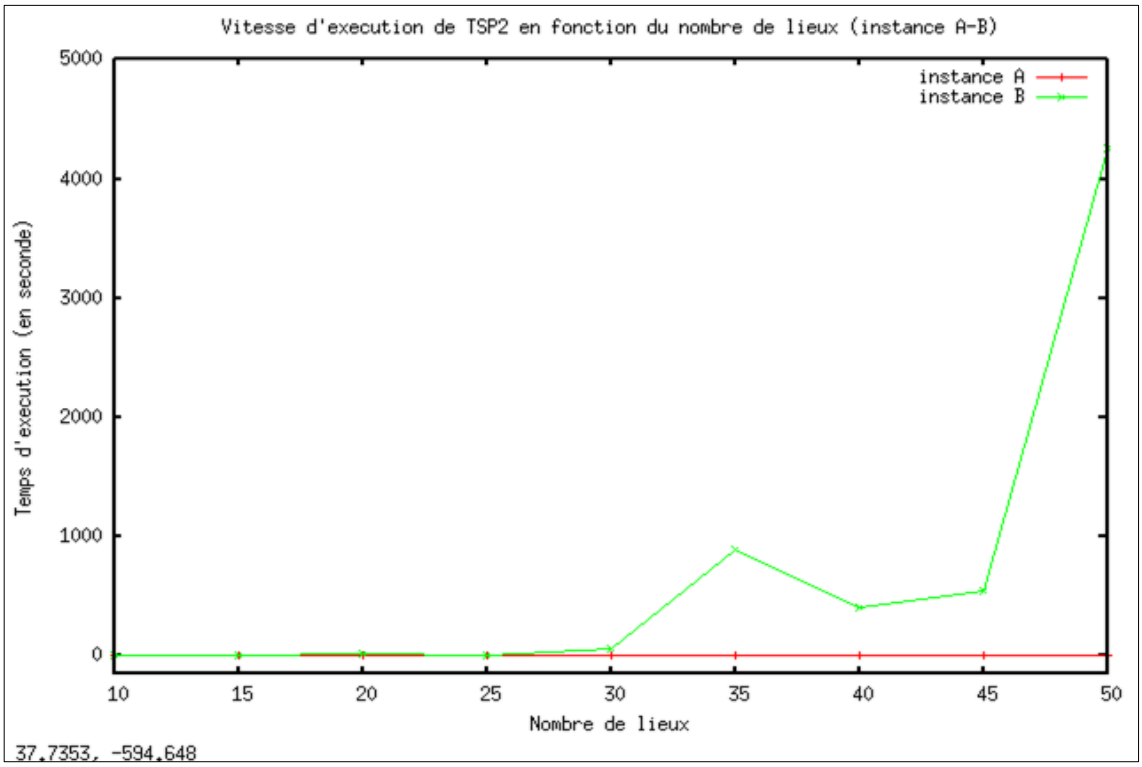

Figure 5

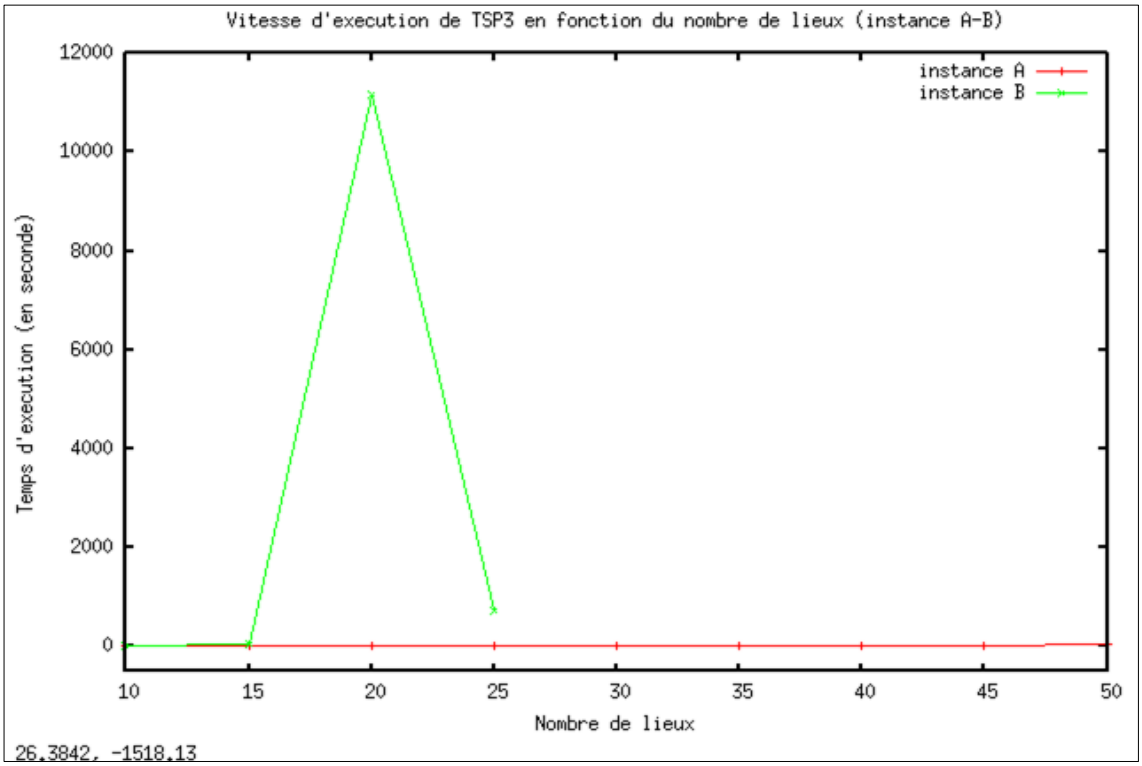

Figure 6

Instance A

Temps d'excécution:

nb Lieux	regrouper()	TSP1()	TSP2()	TSP3()
10	0	0	0	0.012
15	0.004	0	0.004	0.036
20	0.004	0.004	0.02	0.144
25	0.02	0.02	0.096	0.92
30	0.052	0.132	0.428	5.872
35	0.032	0.02	0.112	0.972
40	0.136	0.228	0.776	9.632
45	0.308	0.356	1.208	15.96
50	0.424	0.64	2.124	29.32

Regroupements :

nb lieux	10	15	20	25	30	35	40	45	50
nb regroupement	79	240	694	2948	9126	4195	17626	26984	44145
1	0	0	0	0	0	0	0	0	0
2	9	14	19	24	29	34	39	44	49
3	36	91	171	276	406	561	741	946	1176
4	34	131	425	1588	2712	2981	6094	8693	12114
5	0	4	79	1041	4673	619	8960	14448	25189
6	0	0	0	19	1305	0	1785	2852	5617
7	0	0	0	0	1	0	7	1	0
8	0	0	0	0	0	0	0	0	0

Figure 7
(écriture dans un fichier texte de la fonction *analyse()*)

Instance B

Temps d'excécution:

nb Lieux	regrouper()	TSP1()	TSP2()	TSP3()
10	0	0.004	0.012	0.172
15	0.016	0.68	0.56	35.824
20	0.148	37.044	10.592	11151.01
25	0.288	10.18	8.408	718.804
30	1.46	149.26	58.936	
35	10.04	7025.07	894.536	
40	9.428	1297.06	409.592	
45	15.676	1541.65	543.796	
50	88.332	31734.52	4258.34	

Regroupements :

nb lieux	10	15	20	25	30	35	40	45	50
nb regroupement	288	4509	42299	62714	270307	1954584	1567966	2274695	10124740
1	0	0	0	0	0	0	0	0	0
2	9	14	19	24	29	34	39	44	49
3	36	91	171	276	406	561	741	946	1176
4	84	364	969	2024	3654	5984	9139	13244	18424
5	125	1001	3876	10600	23656	46291	81453	134413	211371
6	34	1780	11167	30075	90260	251430	418329	707195	1512288
7	0	1143	17072	18532	114597	701247	722680	1037643	4083062
8	0	116	8292	1183	35831	728642	313778	358209	3431809
9	0	0	733	0	1871	212460	21797	22896	822765
10	0	0	0	0	3	7935	10	105	43597
11	-1	0	0	0	0	0	0	0	199
12	-1	0	0	0	0	0	0	0	0
13	-1	0	0	0	0	0	0	0	0

Figure 8
(écriture dans un fichier texte de la fonction *analyse()*)

Instance A :

nbL	TSP1			TSP2			TSP3		
	regroupe+TSP	main	diff	regroupe+TSP	main	diff	regroupe+TSP	main	diff
40	0.364	1.036	0.672	0.912	1.644	0.732	9.768	10.548	0.78
45	0.664	13.18	12.516	1.516	13.604	12.448	16.268	28.112	11.844
50	1.064	22.16	21.096	2.548	23.148	20.6	29.744	51.184	21.44

Instance B :

nbL	TSP1			TSP2		
	regroupe+TSP	main	diff	regroupe+TSP	main	diff
20	37.192	183.156	145.964	10.74	163.764	153.024
25	10.468	28.36	17.892	8.696	28.108	19.412
30	150.72	319.632	168.912	60.396	221.752	161.356

Figure 9