Introduction aux techniques décisionnelles

Problème du voyageur de commerce

HARRACH Benjamin, HARTKE David, MACHECOURT Benjamin

Le problème du Voyageur de Commerce qui consiste à parcourir les N villes d'un réseau en effectuant le trajet le plus court est un problème tout aussi intéressant que complexe. En effet, trouver une solution optimale à un tel problème semble facile en théorie : il suffit d'essayer toutes les solutions et de comparer leurs longueurs pour voir la meilleure. Cependant, en pratique, l'énumération de toutes les solutions peut prendre trop de temps. Or, le temps de recherche de la solution optimale est un facteur très important et c'est à cause de lui que les problèmes d'optimisation combinatoire sont réputés si difficiles. Pour pallier à ce problème de temps, de nombreuses méthodes approchées ont été inventées. Tout au long de ce TP nous avons donc essayé de trouver et d'implémenter les plus efficaces.

I - OPTIMISATION EN PROFONDEUR

Comme première approche du *Traveling Salesman Problem*, nous avons commencé par coder **l'algorithme heuristique du plus proche voisin** (dans la classe Solution) car simple à mettre en place. Celui-ci nous a permis d'obtenir des résultats efficaces mais largement améliorables (voir annexe 1). A chaque itération on recherche le nœud le plus proche du nœud où l'on se trouve, sans jamais repasser par un nœud déjà inséré dans le chemin courant. Dans notre algorithme nous partions à chaque fois d'une ville de départ aléatoire. Ainsi, afin de limiter la part de l'aléatoire dans le résultat final, nous avons décidé de coder une boucle qui, tant que le temps imparti à l'exécution de notre algorithme (à savoir m_time) n'est pas atteint, relance notre algorithme à partir d'une nouvelle ville et compare le résultat aux précédents. A la fin, il suffisait juste de récupérer la meilleure solution. Cette méthode de sélection de la solution finale sera d'ailleurs utilisée dans tous nos autres algorithmes par la suite car extrêmement efficace.

Nous avons ensuite voulu optimiser cette heuristique. Afin d'améliorer les résultats obtenus, nous avons donc ajouté **l'algorithme du 2-opt** (en boucle jusqu'à ce que l'on n'observe plus d'amélioration) à la suite de notre **algorithme du plus proche voisin**. Il s'agit d'un algorithme de recherche locale qui à chaque itération supprime deux arrêtes sécantes de la solution courante pour ensuite les reconnecter en évitant le croisement. Le **2-opt** (dans la classe Solution également) nous fournissait de bonnes solutions assez rapidement.

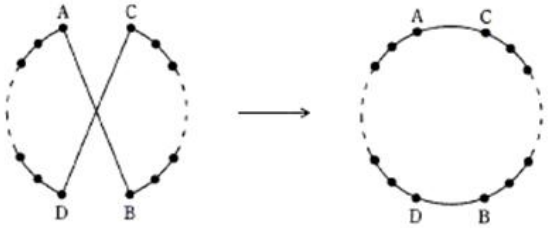

fig.1 – amélioration de la solution initiale via l'algorithme 2-opt

Nous avons ensuite codé le **nodeInsertion** : on insère chaque nœud à chaque position de la solution et si cela améliore la solution on garde ce nouveau chemin.

Nous avons ensuite voulu combiner les 2 méthodes d'optimisation. Après quelques essais « à l'aveugle », nous avons finalement opté pour un algorithme méta-heuristique de la littérature : le **Variable Neighborhood Descent** (VND). Il s'agit de deux boucles "while" imbriquées dans lesquelles on utilise le **2-opt** et le **nodeInsertion**. Cet algorithme donne de très bon résultats, même s'il est assez lent.

Les résultats étaient meilleurs (gain de 2% en moyenne) mais nous étions encore à 3% de l'optimum en moyenne sur les 10 instances test (voir annexe 1).

Nous avons alors codé dans la continuité le **edgeInsertion** : on insère chaque chemin entre deux villes à une autre position et si l'on constate l'amélioration on réalise le changement. Cet algorithme a ensuite été rajouté au **VND :**

```
public void VND() throws Exception {
    double avant, apres;
        do {
            do {
                do {
                    avant = this.evaluate();
                    this.deux_opt();
                    apres = this.evaluate();
                } while (avant > apres);
                avant = this.evaluate();
                this.nodeInsertion();
                apres = this.evaluate();
            } while (avant > apres);
            avant = this.evaluate();
            this.edgeInsertion();
            apres = this.evaluate();
        } while (avant > apres);
}
```

Les résultats obtenus suite à ce rajout ont été extrêmement positifs puisque nous avons alors atteint une moyenne de 1,3% d'écart à l'optimum en moyenne sur les 10 instances test (attention les résultats sont à nuancer car cette méthode impliquait un dépassement des 60 secondes autorisées pour les instances d657 et rat575).

fig.2 – notre implémentation du VND

II - L'ALGORITHME GENETIQUE (GA)

Rapidement, sentant que nous arrivions à un stade où les méthodes d'optimisation "traditionnelles" ne nous aideraient plus à nous extraire des optimums locaux, nous avons mis en place le développement en parallèle d'un algorithme génétique. Les algorithmes génétiques sont basés sur le principe de la sélection naturelle. Au départ nous avons une population d'individus aléatoires, qui va évoluer et converger vers la meilleure solution possible au fil des générations.

Nous avons dans un premier temps codé l'algorithme « de base » à partir d'une documentation internet. Pour cela nous avons rajouté deux classes : Population (qui permet de traiter les populations de solutions) et GA (qui contient toutes les méthodes relatives à l'évolution de la population). Tout d'abord la population initiale est créée par la méthode ***generateIndividual()*** (dans la classe Solution). On construit ensuite l'hérédité dans le ***evolvepopulation1()*** : on prend le meilleur individu *parent1* et un individu « correct » *parent2* (sélectionné parmi un ensemble de 10 solutions initiales) qui donnent naissance à un enfant. Cet enfant est créé dans le ***crossover1()*** : on recopie un sous-chemin aléatoire du parent1 puis on complète avec le parent2 avec le reste des villes non utilisées, dans l'ordre, du chemin du parent2. On remplace alors la population entière sauf le meilleur parent. On applique ensuite une mutation aléatoire ***mutationRate()*** qui échange deux villes aléatoirement dans chaque enfant. Puis on répète l'opération complète jusqu'à obtenir une bonne solution pour notre problème.

L'algorithme génétique nous fournissait des résultats très proches du **VND** et était plus rapide. Nous avons alors voulu coupler le génétique (**GA**) avec une optimisation : on utilise le **VND** sur chaque fils après sa création. Ceci donne les optimums pour eil51, eil101, kroA100 et kroA150, mais dépasse les 60s pour les schémas avec plus de villes.

Nous avons donc optimisé le GA avec un 2opt plutôt que le VND complet pour les instances avec plus de 160 villes. Ceci fournissait des résultats très corrects, mais nous avons voulu étoffer notre GA et le rendre le plus performant possible, nous avons donc essayé plusieurs variantes. Deux nouvelles mutations ont été créées : le ***mutationNearest*** (qui est un taux de mutation en %), qui remplace une ville par sa plus proche voisine, et le ***mutationReverse*** (%), qui renverse le chemin entre deux villes. Nous avons également introduis le ***initialNearest*** (%) : la population initiale n'était plus aléatoire mais créée avec un certain pourcentage de chemins "plus proche voisin".

Nous avons remarqué que les meilleurs résultats étaient obtenus en prenant un initialNearest élevé, de l'ordre de 0.9, sans atteindre 1 pour introduire de nouvelles combinaisons. Nous avons également implémenté une méthode de test ***quelsParamètres()***, qui teste toutes les combinaisons de taux de mutation possibles et qui renvoie la meilleure. Étonnamment, cette méthode donne quasiment toujours 0% de mutation pour les 3

mutations possibles. La mutation, censée nous permettre de sortir des minimums locaux, ne nous est donc pas utile dans notre implémentation du GA.

Nous nous sommes alors demandé si c'était à cause de notre manière de faire évoluer la population ou de notre crossover. Nous avons donc implémenté un nouveau ***evolvePopulation2()*** qui prend cette fois-ci toute l'ancienne génération (sauf le meilleur) comme parent2 pour faire le crossover. Nous avons également mis en place un ***crossover2()*** qui garde tous les edges en commun des parents 1 et 2 puis complète avec le parent2. Cependant ces deux méthodes ne changent pas la valeur des paramètres de mutation (on doit toujours les prendre nuls), et elles n'améliorent pas vraiment le résultat en général.

Nous avons donc fixé l'initialNearest à 0.9, le crossover1(), le evolvePopulation1() et les mutations à 0 pour le GA. Nous pouvions encore jouer sur le nombre d'individus de la population et son nombre d'évolution. Nous mettions environ 30 individus et 15 évolutions pour que ce ne soit pas trop long et qu'il puisse y avoir plusieurs itérations de l'algorithme génétique de manière à prendre la meilleure et toujours limiter la partie aléatoire du résultat obtenu à la fin.

III - RETOUR A L'OPTIMISATION DIRECTE

Malgré des résultats corrects pour le GA, nous avons tout de même voulu savoir s'il n'était pas possible d'améliorer encore notre optimisation directe en profondeur. Pour cela nous avons codé en plus le **3-opt** qui permet de déconnecter trois edges qui généralement se croisent afin de les décroiser et diminuer encore la longueur du chemin. Nous avons ensuite voulu intégrer le **3-opt** à notre **VND** mais il s'avère que ce dernier est beaucoup trop long sur les grosses instances pour être utilisé de manière répétée et donc efficace. D'ailleurs, sur les petites instances l'apport du 3-opt était nul. Seules les plus grosses instances ont vu leurs résultats s'améliorer grâce à cette méthode (voir la comparaison des deux dernières colonnes de l'annexe 1).

Suite à différents tests visant notamment à déterminer à partir de quel nombre de nœuds l'algorithme génétique perd de son intérêt (voir dans l'annexe 1 : au début le GA était optimisé jusque 150 villes puis nous avons réussi à trouver des paramètres qui améliorent les solutions jusque 250 villes), nous avons donc finalement décidé de différencier plusieurs situations :

- les instances comprenant moins de 250 villes pour lesquelles on utilise **l'algorithme génétique associé au VND** (avec une population de 100 chemins jusque 160 villes, puis 30 chemins entre 160 et 250 villes)
- celles entre 250 et 400 villes pour lesquelles ont applique le **nearest neighbor** puis un **VND contenant le 3-opt** sur la meilleure solution en **nearest puis VND**
- celles entre 400 et 800 villes où l'on fait juste un **3-opt** sur la meilleure solution en **nearest puis VND**
- pour celles de plus de 800 villes on prend juste la meilleure solution en **nearest puis VND** pour assurer un résultat correct.
- enfin, au dessus de 2200 villes on prend la meilleure solution en **nearest**, sans aucune optimisation, afin d'être certains d'avoir un résultat peu importe la taille de l'instance fournie.

Avec cette implémentation, nous arrivons à un écart de 0,9% en moyenne à l'optimum sur les 10 instances test.

Remarque : afin de respecter la limite de temps pour les grosses instances, nous avons deux méthodes (dans la classe TSPSolver) qui permet de réaliser autant de fois soit l'algorithme génétique, soit le nearest puis VND tant que m_time n'est pas dépassé. Puis cette méthode sélectionne le meilleur résultat et le renvoie comme solution finale.

En conclusion, l'algorithme génétique semble extrêmement puissant puisqu'il nous a permis d'obtenir 4 optimums sur 10. Néanmoins, plus la taille de l'instance augmente, plus cet algorithme est mis en défaut. De plus, nos mutations nous ont finalement été inutiles, ce qui semble illogique au vu de leur objectif qui est de sortir des minimas locaux. Une piste de recherche pour améliorer notre algorithme serait donc de comprendre l'échec de ces mutations et essayer d'en implémenter de plus efficaces. Egalement, un travail pourrait être fait au niveau de la sélection de chaque population enfant (quels chemins garder de la population parente…) car nous n'avons pas réellement tenté d'optimiser cet aspect de l'algorithme génétique. Pour les grosses instances, nous sommes néanmoins assez satisfaits des techniques d'optimisation classiques (2-opt, 3-opt, edge et node insertion, mis sous la forme du VND) qui ont donné de bons résultats (moins de 3% de l'optimum) sur des chemins de départs assez basiques puisque créés par la méthode du plus proche voisin. Il serait d'ailleurs intéressant de comparer l'influence de la méthode de départ (avec un farthest ou best insertion par exemple) sur les résultats de ce VND.

Sources :

- The Traveling Salesman - Gerhard Reinelt
- Handbook of Metaheuristics - Michel Gendreau et Jean-Yves Potvin
- https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
- https://fr.wikipedia.org/wiki/2-opt
- http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5
- http://labo.algo.free.fr/pvc/algorithme_genetique.html

ANNEXE 1 – Tableau évolutif comparatif des résultats des différents algorithmes

{Tableau du haut : distances ; tableau du bas : pourcentages d'écart à l'optimum}

instances	optimal	nearest neighbour	2-opt	VND	VND avec edge insertion	Genetique jusque 150	Genetique jusque 250	Genetique jusque 250 et 3-opt
d198	15780	17620	16134	16049	15842	15830	15822	15812
d657	48912	60174	52472	51314	50228	50457	50457	50351
eil51	426	482	437	432	428	426	426	426
eil101	629	746	675	648	630	629	629	629
kroA100	21282	24698	21748	21415	21639	21282	21282	21282
kroA150	26524	31479	27748	27468	26717	26524	26524	26524
kroA200	29368	34543	30895	30338	29509	29541	29478	29486
lin318	42029	49201	44317	43549	43017	42762	42762	42459
pcb442	50778	58950	53920	52828	51590	51494	51494	51491
rat575	6773	7993	7250	7050	6932	6983	6983	6946

instances	nearest neighbour	2-opt	VND	VND avec edge insertion	Genetique jusque 150	Genetique jusque 250	Genetique jusque 250 et 3-opt
d198	11,7	2,2	1,7	0,4	0,3	0,3	0,2
d657	23,0	7,3	4,9	2,7	3,2	3,2	2,9
eil51	13,1	2,6	1,4	0,5	0,0	0,0	0,0
eil101	18,6	7,3	3,0	0,2	0,0	0,0	0,0
kroA100	16,1	2,2	0,6	1,7	0,0	0,0	0,0
kroA150	18,7	4,6	3,6	0,7	0,0	0,0	0,0
kroA200	17,6	5,2	3,3	0,5	0,6	0,4	0,4
lin318	17,1	5,4	3,6	2,4	1,7	1,7	1,0
pcb442	16,1	6,2	4,0	1,6	1,4	1,4	1,4
rat575	18,0	7,0	4,1	2,3	3,1	3,1	2,6
Moyenne	17,0	5,0	3,0	1,3	1,0	1,0	0,9

OPTIMUM
[0 2[
[2 4[
[4 6[
[6 8[
[8 10[
>10
* temps dépassé