Rapport TP ITD groupe 4

Ajaykrishna Mouttoucomarassamy - Quentin Séité - Romain Vo

Sommaire :

I) Contexte

Le Traveling Salesman's Problem

Le traveling Salesman's problem (abrégé en TSP) est un problème d'optimisation célèbre qui, malgré la simplicité de son énoncé, ne connaît aucun algorithme permettant de trouver une solution exacte en temps polynomial. Le problème est le suivant : On cherche à trouver le plus court chemin passant par N villes (définies au préalable) et retournant à son point de départ. C'est dans cette dernière condition que réside le cœur de la complexité du problème.

Dans ce TP, notre objectif est de coder un algorithme capable de donner un résultat le plus proche possible de la solution optimale au TSP en moins de 60 secondes, et ce pour 10 versions du problème (instances) de nombre de villes différent.

II) Heuristiques envisagées et implémentée

a) Algorithme de Recherche locale

En partant d'une solution initiale, on parcourt le voisinage de cette solution en appliquant une transformation permettant de parcourir tout l'espace des solutions dans le voisinage. On sélectionne la meilleure des solutions comme nouvelle solution initiale et on réitère le processus jusqu'à ce qu'on ne trouve plus de meilleure solution dans le voisinage de notre solution initiale, c'est-à-dire qu'on trouve un minimum local.

Swapping

Ici, l'opération de transformation choisie est le « swapping », on inverse l'ordre de parcours de deux villes dans la solution initiale. Par exemple, dans un TSP pour les villes 1,2 et 3 et avec la solution initiale « 1->2->3->départ », le voisinage est composé des solutions

« 1->2->3->départ » / « 2->1->3-> départ » / « 3->2->1-> départ » / « 1->3->2-> départ »
« 1->2->3-> départ »

Pour tenter de trouver une meilleure solution après avoir trouvé un premier minimum local, on peut perturber cette solution en lui appliquant quelques « swap » et chercher un nouveau minimum local en espérant que celui-ci sera meilleur ; dans le cas contraire on garde l'ancienne solution.

2-opt

On peut aussi utiliser une transformation dite 2-opt, qui consiste entre deux villes d'index différents à renverser le chemin parcouru, contrairement au swap on ne casse ici que 2 arêtes, les arêtes "extérieures". Par exemple dans un TSP pour les villes 1,2,3,4,5, appliquer une transformation 2-opt entre 1 et 5 reviendrait à faire :

« 1->2->3->4->5 » ----> « 1->4->3->2->5 »

Le 2-opt est un dérivé de la k-optimisation qui consiste à tester pour chaque ensemble de k points les meilleurs renversement et échanges de villes.

b) Algorithme de colonies de fourmis

Dans une colonie de fourmis, les éclaireuses vont, pour aller jusqu'à un point de nourriture, parcourir un chemin aléatoire tout en laissant des phéromones derrière elles. Attirées par ces phéromones, d'autres fourmis vont suivre le chemin jusqu'à la nourriture. S'il existe plusieurs chemins vers la nourriture, c'est le chemin le plus court qui sera le plus chargé en phéromones et il sera donc privilégié par les fourmis. Au contraire, un chemin qui est au départ utilisé par hasard mais qui est mauvais va être délaissé et, par évaporation des phéromones au cours du temps, ne plus être utilisé car ne contenant plus de phéromones.

L'algorithme de colonies de fourmis s'inspire par biomimétisme du comportement de ces colonies. Les fourmis sont des objets comportant une liste de villes parcourues, une liste de ville à parcourir et une mémoire du chemin parcouru. Elles peuvent déposer des phéromones sur les différents arcs de parcours du graphe. Ces phéromones s'évaporent en partie à la fin de N itérations suivant leur dépôt.

On pose un certain nombre de « fourmis » réparties aléatoirement sur les différentes villes. A chaque itération d'une boucle, une fourmi va aller visiter une ville qu'elle n'a pas encore visité. Le choix de cette ville se fait en fonction de la quantité de phéromones présentes sur le chemin entre les deux villes, et la visibilité de la ville, c'est-à-dire sa distance à la ville précédente. Une fois que les fourmis ont chacune visitées les N villes de l'instance et sont revenu à leur point de départ (donc au bout de N itérations), Chaque fourmi a alors en mémoire son parcours ainsi que la distance que représente ce parcours. Plus cette distance est petite, plus la fourmi dépose de phéromone sur ce chemin, c'est-à-dire qu'on augmente la probabilité que ce chemin soit pris par les autres fourmis inversement proportionnellement à la taille du chemin. Le plus petit chemin après N itérations est stocké s'il est inférieur à la solution actuelle (quand il y en a une). On vide alors la mémoire des fourmis et on recommence le processus. L'algorithme s'arrête après un nombre défini de boucle, ou bien si toutes les fourmis prennent le même chemin.

Afin d'améliorer la rapidité de l'algorithme, on définit un nombre E de fourmis dites élitistes, c'est-à-dire les fourmis ayant parcouru les meilleurs chemins, qui vont déposer plus de phéromones sur ces solutions potentiellement meilleures et ainsi augmenter la vitesse de convergence.

c) Algorithme génétique

L'algorithme génétique s'inspire de la théorie de l'évolution qui explique l'évolution d'une population d'individus à travers des mécanismes tels que la **reproduction et la mutation**.

En assimilant un individu à un chemin, c'est à dire un ordre de visite des villes, nous pouvons utiliser cet algorithme pour obtenir une solution au TSP. L'algorithme peut se résumer comme ceci :

Initialiser une population d'individus générer aléatoirement
Pour n itérations :
    Parent 1, Parent 2 = selectionParents()
    Enfant = crossover(Parent1, Parent2)
    Si hasard > seuilMutation :
        Appliquer mutation à Enfant
    Si hasardBis > seuilOptimisation :
        Optimiser Enfant
    Si Enfant acceptable
        Insérer Enfant dans la population
Retourner le meilleur individu de la population

Pour l'initialisation, nous privilégions une **génération aléatoire d'individus** pour avoir une **grande diversité**. Ensuite, la sélection de ceux qui vont se reproduire peut être **aléatoire**, **élitiste** (un des parents est le meilleur de la population) ou "**à la roulette**" qui attribue à chaque individu une probabilité d'être sélectionné proportionnelle à sa valeur (distance).

Puis, les parents se **reproduisent pour créer un ou plusieurs enfants potentiellement meilleurs que toute la population actuelle**. Pour cela, nous avons 3 méthodes de "crossover", c'est à dire de recombinaison :

- crossover à 1 ou 2 points : on recopie les villes d'un parent dans l'enfant entre 0 et un index (crossover1) ou entre 2 index (crossover2), puis on complète avec les villes, non déjà présentes dans l'enfant, de l'autre parent selon un ordre défini.
- crossoverOX (ordered crossover) : variante du crossover2 dans lequel l'ordre de recopie du 2ème parent est dans l'ordre dans lequel il apparaît dans celui-ci.

Ensuite, les enfants ont une probabilité (fixée par nous et déterminée empiriquement) de subir une **mutation** qui consiste à changer l'ordre aléatoirement par swap de 2 villes ou de morceaux de villes, ce qui permet de **sortir des optima locaux**. Nous devons également **optimiser** l'enfant en lui appliquant un algorithme de recherche local, nous avons choisis le **2-opt**.

Enfin, il faut **insérer l'enfant dans la population** en enlevant le moins bon ou le plus ressemblant en prenant soin de ne pas éliminer le meilleur.

Après avoir codé et testé les différentes méthodes pour chaque étape de l'algorithme, nous avons choisi **empiriquement** les meilleurs : **sélection élitiste** pour les premières itérations puis **aléatoire**, **crossoverOX**, **mutation par morceaux** et **insertion en enlevant le moins bon**.

III) Comparaison

Comparaison des solutions (NT = Non testé)

Circuit	Optimal	2-opt	Temps d'éxécution	Swap	Temps d'éxécution	Ant Colony	Temps d'éxécution	**Génétique + 2-opt**	**Temps d'éxécution**
brazil58	inconnu	25726	111 ms	37890	111ms	NT	> 60s	**25395**	**<1s**
d198	15780	16176	244 ms	36422	513ms	NT	> 60s	**15781**	**8 s**
d657	48912	53154	469 ms	156729	18,45 s	NT	> 60s	**49516**	**60s**
eil10	inconnu	159	74 ms	159	72ms	159	<1s	**159**	**<1s**
eil51	426	437	116 ms	534	128ms	435	<1s	**426**	**<1s**
eil101	629	662	147 ms	951	225ms	706	> 60s	**629**	**<2s**
kroA100	21282	22121	145 ms	41389	243ms	NT	> 60s	**21282**	**<2s**
kroA150	26524	27873	169 ms	50574	346ms	NT	> 60s	**26524**	**4,5s**
kroA200	29368	30470	194 ms	73356	490ms	NT	> 60s	**29368**	**<5s**
lin318	42029	44565	262 ms	118244	1,160s	NT	> 60s	**42051**	**30s**
pcb442	50778	55977	334 ms	141013	4,563s	NT	> 60s	**51223**	**<35s**
rat575	6773	7416	395 ms	19341	11,095s	NT	> 60s	**6926**	**60S**

Au vu des résultats des différentes approches testées, l'utilisation de **l'algorithme génétique optimisé via 2-opt se démarque** dans l'objectif de trouver l'optimal en moins de 60 secondes et c'est donc celui que nous utiliserons, en stoppant l'algorithme juste avant 60 secondes pour éviter une exécution trop longue sur de très grosses instances.

Quant aux **paramètres** utilisés dans l'algorithme génétique, nous avons fait de nombreux tests en les faisant varier (cf l'annexe) et bien sûr pris les meilleurs :

Nb individus	**Nb itérations élitistes**	**Nb itérations aléatoires**	**Seuil de mutation**	**Seuil d'optimisation**
100	4 000	100 000	0.1	0.5

IV) Conclusion

Ainsi, en combinant les meilleurs heuristiques et en écartant des fourmis, nous sommes parvenu à des résultats satisfaisants (écart à l'optimal de 2.2% dans le pire des cas et le plus souvent <1%) en moins de 60 secondes, nous remplissons donc le cahier des charges.

github: https://github.com/qseite/ITD_Mouttoucomarassamy_Seite_Vo

Bibliographie :

Algorithme génétique :
http://labo.algo.free.fr/pvc/algorithme_genetique.html
http://www.dmi.unict.it/mpavone/nc-cs/materiale/moscato89.pdf

Algorithme de colonie de fourmis :
Optimisation par colonies de fourmis
COSTANZO Andrea LUONG Thé Van MARILL Guillaume
http://www.i3s.unice.fr/~crescenz/publications/travaux_etude/colonies_fourmis-200605-rapport.pdf

Algorithme 2-opt :

http://labo.algo.free.fr/pvc/algorithme_2opt.html

ANNEXE

Résultats finaux :

Moyenne des écarts : 0.44%

Circuit	Optimal	Génétique + 2-opt	Ecart
brazil58		25395	
d198	15780	15781	0,01%
d657	48912	49516	1,23%
eil10		159	
eil51	426	426	0,00%
eil101	629	629	0,00%
kroA100	21282	21282	0,00%
kroA150	26524	26524	0,00%
kroA200	29368	29368	0,00%
lin318	42029	42051	0,05%
pcb442	50778	51223	0,88%
rat575	6773	6926	2,26%

Améliorations - recherche de paramètres optimaux dans génétique + 2-opt

Mutation&Optimisation enfant1 et enfant2 = lié			
100 individus	4000 repro élitistes	6000 repro aléatoires	
seuilMutation=0.1	seuilOptimisation=0.5		écart
itération finale=4201	d657	49563	1,33%
itération finale=3689	d657	49622	1,45%
itération finale=3731	d657	49739	1,69%
itération finale=4022	d657	49786	1,79%
itération finale=5684	rat575	6920	2,17%
itération finale=5195	rat575	6950	2,61%
itération finale=5876	rat575	6935	2,39%

itération finale=9215	pcb442	51106	0,65%
itération finale=10000	pcb442	51425	1,27%
itération finale=10000	pcb442	51001	0,44%

Mutation&Optimisation enfant1 et enfant2 = indépendante			
100 individus	4000 repro élitistes	6000 repro aléatoires	
seuilMutation=0.1	seuilOptimisation=0.5		écart
itération finale=3551	d657	49481	1,16%
itération finale=4106	d657	49586	1,38%
itération finale=3834	d657	49766	1,75%
itération finale=5218	rat575	6937	2,42%
itération finale=5363	rat575	6930	2,32%
itération fiale=5495	rat575	6956	2,70%
itération finale=9862	pcb442	51072	0,58%
itération finale=10000	pcb442	51002	0,44%
itération finale=10000	pcb442	51283	0,99%
itération finale=10000	pcb442	51375	1,18%

Mutation&Optimisation enfant1 et enfant2 = indépendante			
40 individus	4000 repro élitistes	6000 repro aléatoires	
seuilMutation=0.1	seuilOptimisation=0.5		écart
itération finale=4619	d657	49648	1,50%
itération finale=4333	d657	49590	1,39%
itération finale=3910	d657	49758	1,73%
itération finale=3907	d657	49559	1,32%
itération finale=5450	rat575	6924	2,23%
itération finale=5545	rat575	6864	1,34%
itération finale=5233	rat575	6891	1,74%
itération finale=6579	pcb442	51496	1,41%
itération finale=8340	pcb442	51020	0,48%
itération finale=9703	pcb442	51019	0,47%
itération finale=10000	pcb442	51166	0,76%