2015
2016

Rapport de projet

METAHEURISTIQUE

SALETTI BENJAMIN / GALLO MATHIEU

Table des matières

Le problème TSP

Le problème du voyageur de commerce (ou Traveling-Salesman Problem) est un problème récurrent NP complet. Le but est le suivant : Un marchand doit passer par un ensemble de villes, une seule fois par ville, tout en minimisant le coût de déplacement.

Ce problème étant NP complet, il n'est pas envisageable de tester toutes les entrées différentes possibles ; le temps d'exécution serait exponentiel selon la taille du problème. Si l'on prend en entrée un ensemble de seulement 10 villes à visiter, il y a 3.628.000 combinaisons possibles.

Il existe cependant d'autres méthodes pour approcher un résultat optimal, avec un temps d'exécution acceptable. Dans le cadre du projet donné, il nous est demandé d'utiliser un algorithme génétique afin de trouver une solution approchant l'optimale.

Les algorithmes génétiques

La logique des algorithmes génétiques a trouvé son inspiration dans le comportement du vivant, comme par exemple le Darwinisme qui est l'étude de l'évolution. Au fil du temps, toutes les espèces évoluent par leur ADN de deux manières différentes : la mutation, et le croisement. Ces changements d'ADN ayant un impact sur les capacités des individus, une sélection naturelle se fait donc continuellement : les individus les plus propices à vivre deviennent une nouvelle génération, tandis que les autres disparaitront peu à peu.

Une mutation est une modification de l'ADN d'un individu, tandis qu'un croisement est le résultat d'une reproduction. Dans le cas d'un croisement, il est possible de créer deux individus en fonction de l'ADN des parents, sans pour autant que tous les individus fils ne se soient identiques.

Dans le cadre du problème du TSP, un individu sera représenté par solution au problème (qui correspond à une suite de villes), et nous simulerons une population par un ensemble de solutions différentes.

1) Les algorithmes testés

Pour mettre en place la recherche d'une solution par un algorithme génétique, nous avons choisi d'utiliser une mutation et un croisement, et de faire une sélection par tournoi. Nous avons donc testé plusieurs mutations et croisements afin de déterminer lequel d'entre eux pouvait nous fournir le meilleur résultat.

Pour chacune de ces méthodes, nous avons eu besoin de choisir un (pour la mutation) ou deux individu(s) (pour le croisement). Cette sélection s'est faite de manière « naturelle », par un tournoi. Sur toute la population, nous choisissons initialement 4 individus aléatoirement à mettre dans le tournoi, qui nous renvoie le meilleur des 4.

a. Les différentes mutations implémentées :

MAT.MUTATION1 : On choisit deux éléments aléatoires, et on inverse leur position. On réitère l'opération n fois.

Exemple de transformation :

1-2-3-4-5-6	**1-4-2-3-5-6**	**1-6-2-3-5-4**	**5-6-2-3-1-4**	**5-6-2-4-1-3**
Initiale	Inversion(2,4)	Inversion(4,6)	Inversion(1,5)	Inversion(3,4)

MAT.MUTATION2 : On retire N% des chemins de la liste initiale, et les réinsère aléatoirement.

Exemple de transformation :

1-2-3-4-5-6	**1-_-_-4-5-_**	**1-3-_-4-5-_**	**1-3-6-4-5-_**	**1-3-6-4-5-2**
Initiale	Liste = **2-3-6**	Liste = **2-6**	Liste = **2**	Finale

MAT.MUTATION3 : On retourne une liste constituée successivement de la tête et de la queue de liste.

Exemple de transformation :

1-2-3-4-5-6	**1-6-2-5-3-4**
Initiale	Finale

BEN.MUTATION : On tire deux éléments aléatoires, puis on inverse le chemin entre les deux.

Exemple de transformation :

1-2-3-4-5-6	**1-6-5-4-3-2**
Tirage : 2, 6	Finale

BEN.MUTATIONPOURCROISEMENT : On tire deux nombres aléatoires a et b, puis on décale l'élément en position a jusqu'à la position b.

Exemple de transformation :

1-2-3-4-5-6	**1-3-4-5-6-2**
Tirage : 1, 5	Finale

OBSERVATIONS EFFECTUEES :

Sur toutes ces mutations, la plus performante est : **Ben.Mutation**. Toutes les autres renvoient une solution avec un écart non négligeable. Nous supposons donc que la manière de transformer la solution initiale a un impact sur l'amélioration au fil des générations.

En effet, cette mutation est la seule à conserver des bouts de chemins lorsqu'on l'effectue. Les autres mutations ne laissent que très peu de villes successives dans le même ordre lorsqu'elles mutent un individu. Une bonne mutation ne doit donc que morceler la solution initiale, et non la réordonner complètement.

b. Les différents croisements implémentés :

MAT.CROISEMENT : On retire N% des chemins des deux parents, puis on les réinsère selon l'ordre de l'autre parent.

Exemple de transformation :

1-2-3-4-5-6-7-8	**1-2-_-_-5-_-7-_**	**1-2-4-6-5-3-7-8**
Initiale	Solution coupée	Finale
2-4-6-8-1-3-5-7	**2-_-_-8-1-_-5-7**	**2-3-4-8-1-6-5-7**

BEN.CROISEMENT :

« Order Search »

PAS	ACTION	EXEMPLE
0	Sélectionner P1 et P2 de la population	P1 = 1-2-**5-4-3**-7-6 P2 = 5-4-**2-6-3**-1-7
1	Sélectionner aléatoirement deux points c_1 et c_2 avec $c_1 < c_1$	random = 0.44; alors c_1 = int(7x0.44)+1 = 3 random = 0.71; alors c_2 = int(7x0.71)+1 = 5
2	Changer les positions (c_1, c_1+1,...c_2) dans P1 et P2 pour former partiellement C1 et C2.	C1 = ?-?-**2-6-3**-?-? C2 = ?-?-**5-4-3**-?-?
3	Créer une liste L1(L2) en réaccommodant les éléments de P1(P2) dans le sens des aiguillés de l'horloge $c_2 + 1$, $c_2 + 2$,... 1, 2,...c_2	L1 = (7, 6, 1, 2, 5, 4, 3) L2 = (1, 7, 5, 4, 2, 6, 3)
4	De L1(L2), créez L1' (L2') en éliminant les noedes déjà attribues à C1(C2) dans le pas 2 et au même temps nous conservons l'ordre dans L1 et L2	L1' = L1 - (2, 6, 3) = (7, 1, **5, 4**) L2' = L2 - (5, 4, 3) = (1, 7, **2, 6**)
5	Attribuer les éléments de L1'(L2') aux éléments qui manquent dans C1(C2) avec l'ordre suivant $c_2 + 1$, $c_2 + 2$,... 1, 2,...$c_1 - 1$	C1 = **5-4**-2-6-3-7-1 C2 = **2-6**-5-4-3-1-7

c. Les autres techniques testées :

LA DOUBLE MUTATION : Nous avons testé d'utiliser deux mutations d'affilée pour avoir un individu encore plus différent du premier. Cette solution n'a pas été retenue à cause des mauvais résultats obtenus.

TCHERNOBYL ROLLBACK: A la place du tournoi, nous avons tenté d'effectuer un croisement sur toute la population, de la muter, et de garder uniquement les meilleures solutions parmi les parents et les enfants. Cette méthode n'était pas très efficace, et la population se mettait très vite à stagner.

LA POPULATION LEGENDAIRE: A chaque itération, nous ne gardions que les meilleurs individus des parents, et des enfants. Cette méthode n'a pas non plus été retenue, car comme la précédente, la population stagnait très vite.

2) Les algorithmes sélectionnés

Parmi toutes les solutions citées, nous avons choisi d'utiliser la mutation *Ben.Mutation*, et le croisement *Ben.Croisement*. Une fois les algorithmes choisis, nous avons testé les différentes techniques listées pour améliorer notre résultat. Chaque technique nous donnait une amélioration d'environ 5%.

a. Les autres techniques utilisées :

L'EVALUATION D'UNE SOLUTION : A chaque nouvelle génération, il faut utiliser la méthode *evaluate()* pour connaître le coût d'une solution, et mettre à jour la meilleure solution trouvée si tel est le cas.

Pour gagner quelques exécutions en plus, nous avons créé notre propre méthode *fakeEvaluate()* moins coûteuse que l'ancienne, et appelons l'ancienne méthode uniquement lorsqu'on sait qu'une meilleure solution a été trouvée.

LA TAILLE DE LA POPULATION: Par défaut, nous avions mis une population de taille 40. Par la suite, nous avons placé une population de taille dynamique selon le problème : la population contient autant d'individus que de nombre de villes dans le problème.

LE RATIO DE MUTATION: Par défaut, le ratio de mutation était placé à 10% car c'est ici que nous obtenions les meilleurs résultats. Par la suite, nous avons mis en place un ratio dynamique qui varie en fonction de la recherche de solution : plus la population stagne, plus le ratio augmente (avec un plafond à 100%).

LE TOURNOI: Le tournoi s'effectue entre 4 participants distincts, et renvoie le meilleur des 4. Nous l'avons aussi rendu dynamique en diminuant de 1 la taille du tournoi à chaque fois que la population stagne pendant 60 itérations (avec un minimum à 1).

INITIALISATION INTELLIGENTE: L'initialisation basique renvoie un chemin totalement aléatoire qui passe par toutes les villes. Ce chemin n'est pas du tout optimisé, et le résultat est quasiment tout le temps très mauvais.

Nous avons donc réalisé une initialisation qui choisit une suite de ville selon la distance entre les villes. Cette initialisation retourne donc une solution initiale très optimisée, mais qui tombe sans doute un optimum local très rapidement. Nous avons donc couplé les deux initialisations (50% chacune) pour avoir des chemins optimisés et aléatoires en même temps.

LE BIG BANG : Lorsqu'une population stagne trop longtemps, c'est que nous nous retrouvons dans un minimum (peut-être local). Pour éviter d'être mal tombé, nous avons mis en place le big bang : Lorsque l'on stagne pendant 10x la taille du problème, on efface une partie de la population actuelle pour en créer une nouvelle. Cette nouvelle population est constituée d'¼ de survivants de l'ancienne population, ¼ initialisée avec *smartInit()*, et ½ initialisées totalement aléatoirement.

Pour chaque méthode (Mutation ou Croisement) de transformation, nous devions sélectionner un ou deux individus à muter ou croiser. Pour cette sélection, nous avons utilisé la méthode de tournoi.

Conclusion

Ce projet nous a permis de constater qu'il existe une très large variété d'algorithmes génétiques, et beaucoup de variantes possibles. Les algorithmes les plus utiles peuvent varier d'un problème à un autre, ainsi que les optimisations à effectuer.

L'utilisation d'algorithmes génétiques pour résoudre des problèmes NP-complet reste non négligeable, car il nous permet d'atteindre une solution plus optimisée en moins de temps.

Nous avons aussi pu remarquer l'importance que les paramètres pouvaient avoir dans l'algorithme, et l'avantage de les rendre dynamiques pour avoir des solutions encore plus performantes. Le plus gros défaut de ces méthodes reste l'échelonnage pour atteindre les paramètres optimaux, qui doivent être revus à chaque modification de la résolution du problème.