Mathis Saillot

PROJET MÉTAHEURISTIQUE

Voyageur de Commerce

Je Descends de la Colline à Cheval

Université de lorraine M1 Informatique UFR MIM 19/20

Introduction

Dans ce rapport je vais présenter les différents algorithmes métaheuristiques que j'ai mis en place dans le but de trouver la meilleure solution possible à certaines instances du problème du voyageur de commerce (TSP). Ce problème - peut-être l'un des problèmes d'optimisation les plus célèbres avec le problème du sac à dos - consiste à trouver un cycle Hamiltonien de longueur minimal dans un graphe non orienté complet. Pour évaluer les algorithmes, 4 instances du TSP sont utilisés : a280, bier127, ch130 et une dernière non communiquée. Le score de chaque algorithme sur chaque instance est la moyenne des meilleures solutions obtenues sur 10 exécutions d'une minute.

Dans ce rapport je vais commencer par expliquer comment sont représentées les données du TSP dans mon code ainsi que les différents opérateurs qui sont à la base de la plupart des algorithmes.

Ensuite je vais présenter les différents algorithmes que j'ai implémenté afin de résoudre ce problème. Puis je vais comparer les performances de ces algorithmes.

Les algorithmes que j'ai essayé sont les suivants : Recuit Simulé (SA), Hill Climbing (HC), algorithme évolutionnaire (GA), Monte Carlo Tree Search (MCTS) et Ant Colony Optimisation (ACO).

Pour conclure je parlerais des idées que je n'ai pas eu le temps d'implémenter dans le but d'améliorer les résultats.

I. Représentation des données et opérateurs de voisinage

1. Représentation des données

Une solution du TSP est représentée par une suite d'entiers représentant l'ordre dans lequel les villes sont visités. Une solution valide pour un problème de taille N est donc une suite de N entiers où chaque entier de 0 à N-1 apparait une et une seule fois.
L'évaluation de la qualité d'une solution est la distance totale du chemin, à savoir la somme des distances de deux villes consécutives du chemin, plus la distance entre la première et la dernière ville.

Afin de réduire l'espace de recherche, la dernière ville du trajet ne change jamais, tous mes opérateurs modifiants des solutions ne changent pas la dernière ville du trajet.

2. Opérateurs de voisinages

Pour construire une solution, une possibilité est d'utiliser un opérateur de voisinage. Deux solutions sont dites voisines si il existe un opérateur permettant de passer de l'une à l'autre. J'ai codé quatre opérateurs de voisinage différents pour le TSP. Ces quatre opérateurs sont ceux définis dans le cours de métaheuristiques.

a. Voisin Swap

Cet opérateur intervertit la position de deux villes dans le chemin.

(1 2 3 4 5) -> (1 4 3 2 5)

b. Voisin Zone

Cet opérateur inverse l'ordre des villes entre deux indices.

(1 2 3 4 5) -> (1 4 3 2 5)

c. Voisin Décalage

Cet opérateur change la position d'une ville dans le chemin, mais l'ordre des autres villes reste inchangé.

(1 | 2 3 4 5) -> (1 4 2 3 5)

d. Voisin Adjacent

Cet opérateur intervertit la position de deux villes adjacentes.

(1 2 3 4 5) -> (1 2 4 3 5)

Ces opérateurs n'ont pas la même efficacité suivant l'algorithme dans lequel ils sont utilisés.

3. Heuristique Greedy

Cette heuristique construit une solution en commençant par la ville 0 et en ajoutant la ville la plus proche au fur et à mesure. J'ai utilisé cette heuristique dans différents algorithmes pour avoir une solution initiale par exemple.

II. Mes algorithmes

Dans cette section je vais présenter les différents algorithmes que j'ai implémenter afin de résoudre le problème du TSP.

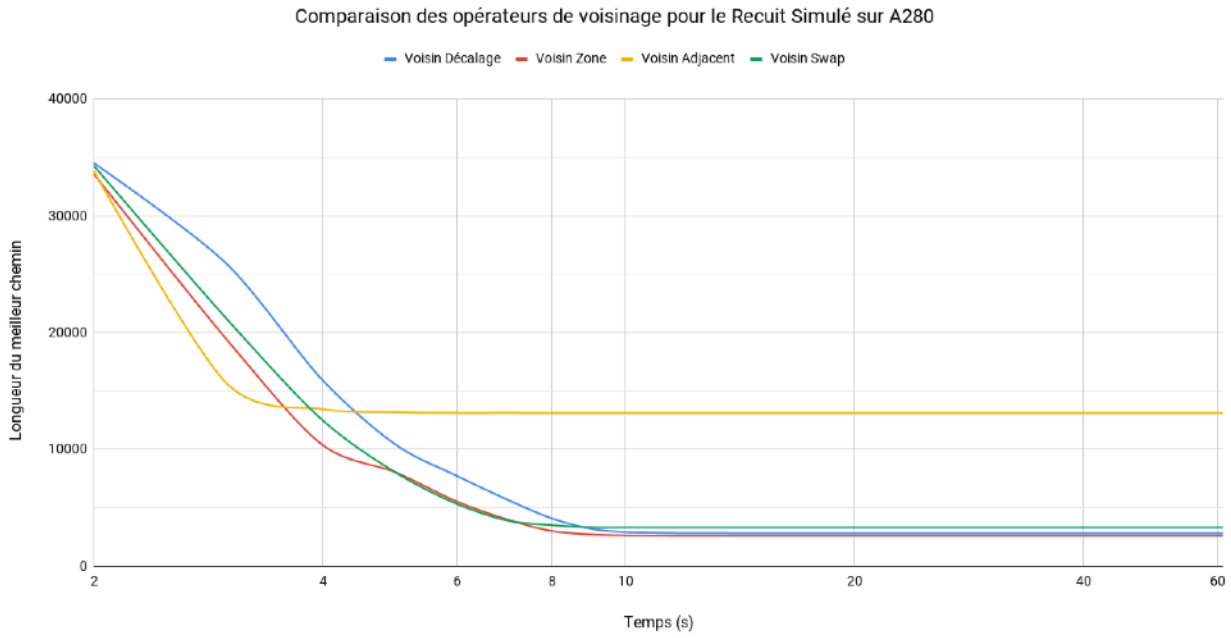
1. Recuit Simulé (SA)

a. Description

Le recuit simulé (SA) - ou Simulated Annealing en anglais - est une catégorie d'algorithme métaheuristiques s'inspirant de la métallurgie. C'est un algorithme utilisant les opérateurs de voisinage. Une solution est initialisée aléatoirement, puis un voisin aléatoire est sélectionné, si il est meilleur il remplace l'état actuel, mais si il est moins bon il a un certain pourcentage de chance de remplacer quand même l'état actuel. Ce pourcentage est déterminé en fonction de la température actuelle et de l'écart entre les deux solutions. La température baisse au fur et à mesure du temps, ce qui fait baisser les chances d'accepter une solution moins bonne. C'est la température qui représente le compromis entre exploration et exploitation. Au début, l'exploration est privilégié, et plus on avance dans le temps c'est l'exploitation qui prend le dessus.

b. Paramètres de l'algorithme

Plusieurs paramètres peuvent être modifiés, ce qui à un impact sur l'efficacité de l'algorithme. Le plus important est l'opérateur de voisinage utilisé. Comme expliqué en I.2, quatre opérateurs ont été testés. La courbe de progression de la valeur de l'état actuel en fonction du temps d'exécution pour chaque opérateur sur le problème a280 est indiqué dans le graphique 1. On peut voir que le meilleur résultat est obtenu avec l'opérateur Voisin Zone et Voisin Décalage. L'opérateur Voisin Adjacent donne les pires résultats.



Les autres paramètres du SA sont liés à la diminution de la température (vitesse de diminution et paliers de diminutions). J'ai effectué de nombreux tests afin d'estimer les valeurs qui donnent les meilleurs résultats.

Tableau 1

Instance TSP	A280	Bier127	Ch130
Nombre de répétitions moyen du SA sur 60 secondes	6	28	26

c. Améliorations

Au fur et à mesure du développement, j'ai modifié le SA afin d'améliorer ses résultats. En observant la courbe d'évolution de la meilleure solution en fonction du temps, on remarque une stagnation du résultat au bout d'un certains temps. Pour contrer ce problème j'ai décidé de relancer le SA depuis une nouvelle solution aléatoire quand l'algorithme stagne pendant trop longtemps. L'algorithme étant stochastique, les résultats varient d'une exécution à l'autre, la répétition permet de diminuer la part d'aléatoire en gardant le résultat du meilleur recuit.

Le nombre de répétition moyen par instance est résumé dans le tableau 1. On peut voir l'impact du nombre de ville sur le nombre de répétition du SA.

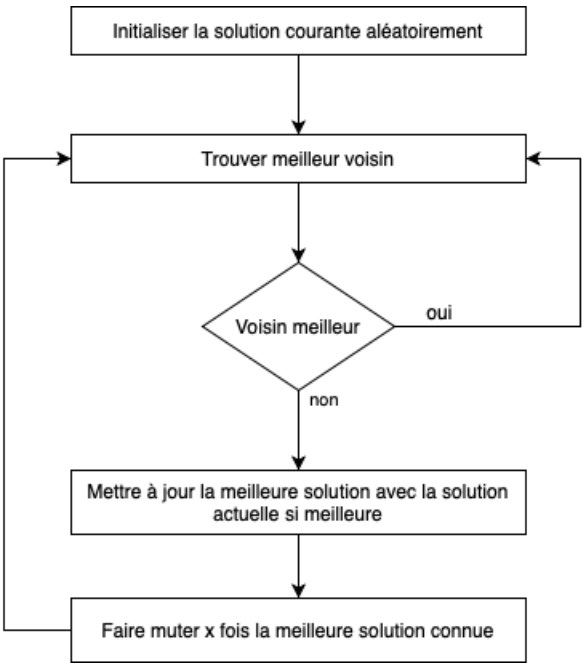
2. Hill Climbing (HC)

a. Description

Le principe de base du HC est plutôt simple, à partir d'une solution, on cherche le voisin qui apporte la plus grande amélioration, dans le cas du TSP c'est le voisin qui diminue le plus la longueur du trajet. Puis on répète l'opération jusqu'à qu'il ne soit plus possible d'améliorer la solution. Un avantage du HC est qu'il va pouvoir très vite converger vers un optimum local mais il est incapable d'en sortir. Pour cela on répète l'algorithme depuis un point de départ différent afin d'essayer d'avoir un meilleur résultat. J'ai dans un premier temps codé le HC dans le but de le combiner avec le SA afin de converger plus vite vers une solution tout en gardant l'avantage de l'exploration du SA. Mais cela n'a pas donné de résultats satisfaisants. J'ai ensuite eu l'idée de partir du HC et de le modifier pour ajouter une phase d'exploration.

b. Mutating Hill Climbing (MHC)

Le fonctionnement de l'algorithme du MHC est décrit par le Diagramme 2. Cette version de l'algorithme permet d'allier exploration et exploitation, ce qui est essentiel dans tout algorithme métaheuristique. Si l'algorithme stagne dans un optimum local pendant trop longtemps, je le relance afin d'essayer d'améliorer la solution en suivant un chemin différent.



Comme pour le SA, les opérateurs de voisinage utilisés pour déterminer le meilleur voisin ainsi que pour appliquer les mutations ont un impact important sur les performances générales de l'algorithme. Les meilleurs performances sont obtenus à l'aide des opérateurs de voisinages Zone et Décalage combinés. Tous les voisins donnés par ces deux opérateurs sont testés et le meilleur est choisi. Pour la mutation, l'opérateur le plus efficace est le Swap. Je pense que ces résultats s'expliquent par le fait que l'opérateur de Zone et de Décalage modifient peu la solution en gardant l'ordre de la plupart des villes intacts. Alors que l'opérateur Swap permet en quelques itérations de mélanger très vite une solution, ce qui est idéal pour de l'exploration.

c. Variantes

Une variante consiste à commencer l'algorithme avec le résultat de l'heuristique Greedy. Pour les instances actuelles du TSP cette variante n'a pas donné de bons résultats. Mais pour des problèmes avec plus de villes, cette variante est essentielle, car avec l'augmentation du nombre de ville, le nombre d'itérations de l'algorithme diminue drastiquement et donc son efficacité.

Une deuxième variante consiste a ne pas repartir obligatoirement de la meilleure solution à qui on applique les mutations, mais parfois depuis le résultat de l'itération précédente à qui on applique aussi des mutations. Cette variante a pour but d'augmenter l'exploration. Comme pour la variante précédente, les résultats sont légèrement inférieur avec ces instances de TSP.

3. Monte Carlo Tree Search (MCTS)

a. Description

La famille d'algorithme correspondant aux MCTS est très large et il existe un très grand nombre de variantes plus ou moins adaptés à différents types de problèmes. J'ai décidé de partir avec un algorithme basé sur UCB (Upper Confident Bound). Le but de l'algorithme est de construire un arbre de décision de façon asymétrique. L'arbre est construit dans la direction la plus encourageante. Ici chaque noeud de l'arbre représente un morceau de trajet, chacun des fils d'un noeud étant ce même trajet auquel on ajoute une ville. La racine de l'arbre est un trajet contenant la ville de départ (la ville 0 par exemple). La racine possède donc N - 1 fils.

L'algorithme se déroule en 4 étapes : sélection, expansion, simulation et backpropagation. Lors de la sélection, on parcourt l'arbre depuis la racine jusqu'à un noeud dont tous les fils ne sont pas encore construit. Pour savoir comment l'arbre est parcouru, on applique une certaine politique.

Une fois le noeud choisi, on créé le fils qui n'a pas encore été exploré, c'est l'expansion. Puis à partir de ce noeud on effectue une simulation jusqu'à la racine de l'arbre. Dans notre cas, on complète le chemin pour qu'il soit complet. Là aussi il faut définir une certaine politique de simulation. Une fois le résultat de la simulation obtenu, la backpropagation remonte le chemin depuis le noeud que l'on a ajouté jusqu'à la racine en ajoutant le score obtenu aux noeuds.

b. Politiques utilisées

La politique de sélection UCB choisi le fils maximisant la valeur de l'équation 1.

$$(1) \quad \frac{score_i}{n_i} + c * \sqrt{\frac{\ln N_i}{n_i}}$$

$score_i$ est le score total du i ème fils.

n_i est le nombre de fois que le i ème fils à été choisi lors de la sélection.

N_i est le nombre de fois où le noeud actuel à été choisi lors de la sélection.

c est une constante correspondant au taux d'exploration.

La partie gauche de l'équation correspond à l'exploitation, plus le score moyen d'un noeud est élevé plus on a envie de le sélectionner. La partie droite correspond à l'exploration. Le taux d'exploration permet de régler l'importance de l'exploration par rapport à l'exploitation. Moins un noeud a été exploré par rapport aux autres fils, plus la valeur augmente.

La politique de simulation que j'ai choisi est basée sur la distance vers la ville suivante. Les villes les plus proches ont plus de chances d'être sélectionnées, mais ce n'est pas forcément la plus proche. Cela permet d'avoir un compromis entre déterminisme et aléatoire.

c. Nested Rollout Policy Adaptative (NRPA)

En effectuant des recherches sur MCTS, j'ai trouvé cet article **[1]** présentant le NRPA. Le but de cet algorithme est de faire des appels récursifs sur un certain nombre de niveaux. Une politique de simulation est mise à jour au fur et à mesure des itérations à l'aide de la meilleure solution trouvée jusque là. La politique associe à chaque coup un poids qui permet de calculer la probabilité de jouer ce coup lors de la simulation. Les poids des coups sont mis à jour en fonction de leur apparition ou non dans la meilleure solution.

Bien qu'étant très intéressante, je n'ai pas réussi à l'implémenter de façon à ce qu'un résultat probant soit donné par l'algorithme.

4. Ant Colony Optimisation (ACO)

a. Description

L'algorithme ACO s'inspire du comportement des fourmis lors de la recherche de nourriture. Les fourmis se déplace le long d'un graphe en laissant des phéromones sur leur trajet. Plus le trajet est intéressant, plus la quantité de phéromones déposés est importante. Les phéromones s'évapore avec le temps. Dans le cas du voyageur de commerce, les fourmis sont placées sur les différentes villes, puis elles construisent une solution en explorant le graphe de ville en ville. Pour savoir vers où elles se dirigent, les fourmis choisissent aléatoirement la ville suivante en fonction de sa distance et de la quantité de phéromone sur le chemin. Les paramètres α et β permettent de contrôler l'importance de l'exploration et de l'exploitation.

Une fois que les fourmis sont revenues à la ville de départ, les phéromones sont mis à jour et une nouvelle génération de fourmis recommence le travail.

b. Améliorations

Afin d'améliorer les performances d'ACO, j'ai essayé d'apporter des variantes à l'algorithme de base. Une fois que les fourmis ont fini leur trajet, celui-ci est amélioré à l'aide d'un des opérateurs de voisinage. Cela permet de converger plus rapidement vers des chemins plus courts.

5. Algorithmes évolutionnaire (GA)

a. Description

Les algorithmes génétiques sont des algorithmes s'inspirant des théories de l'évolution. Dans ces algorithmes, une population d'individus évolue afin d'optimiser un problème. Le génome de chaque individus sert à représenter une solution au problème. Des opérateurs de sélection, de croisement et de mutation permettent de renouveler cette population avec de nouveaux individus ayant en théorie un meilleur génome. Dans le cadre de ce projet j'ai essayé plusieurs opérateurs différents afin de les comparer dans le but de trouver l'opérateur le plus adapté.

Le fitness d'un individus est une valeur permettant de classer les individus au sein d'une population. Dans le cas du TSP le fitness est la longueur du chemin représenté par le génome de l'individu. Du coup plus le fitness est faible, mieux il est classé.

b. Opérateurs de sélection

Afin de sélectionner les individus qui vont être utilisé pour effectuer un croisement, plusieurs solutions sont possible, plus ou moins efficace suivant le contexte :

- La sélection par tournoi. Plusieurs individus sont choisis aléatoirement dans la population et seul l'individus avec la meilleure valeur de fitness est sélectionné. Le nombre d'individus participant au tournoi est un paramètre important permettant de régler les chances qu'un individus de moins bon fitness gagne le tournoi.
- La sélection élitiste. Les x meilleurs individus de la population sont sélectionnés.
- La sélection aléatoire. Un individus est tiré au sort dans la population.
- Une combinaison de plusieurs de ces opérateurs, par exemple un individus est tiré aléatoirement parmi les 10 meilleurs de la population.

c. Opérateurs de mutation

Les opérateurs de mutations que j'ai utilisé sont simplement les quatre opérateur de voisinage.

d. Opérateurs de croisement

Le croisement est une étape très importantes d'un algorithme génétique. C'est cet opérateur qui créer la descendance qui va remplacer la population actuelle. Un opérateur de croisement créer un ou plusieurs enfants à partir de deux parents ou plus. J'ai eu le temps de codé trois opérateurs différents pour mon GA. Les opérateurs sont issus de **[2]**.

Le premier opérateur est le Modified Order Crossover (MOC). Un indice aléatoire est choisi afin de séparer en deux parties les gènes de parents. Les villes de la partie gauche du parent 1 sont placés à leur position dans le parent 2 pour le fils 2 et inversement. Ensuite les villes restantes sont placés dans leur ordre d'apparition dans la partie droite du parent 1.

Parent 1 = (1 2 3 | 4 5 6)
Parent 2 = (5 3 1 | 2 6 4)

La partie gauche est placée :

Enfant 1 = (1 * 3 * 5 *)
Enfant 2 = (* 3 1 2 * *)

Les enfants sont complétés avec les villes restantes dans l'ordre :

Enfant 1 = (1 2 3 6 5 4)
Enfant 2 = (4 3 1 2 5 6)

Le second opérateur est le Partially-Mapped Crossover (PMX). Dans un premier temps, deux indices aléatoires sont choisis afin de séparer les parents en trois parties.
Par exemple P1 = (1 2 | 3 4 | 5 6) et P2 = (5 3 | 1 2 | 6 4). La partie centrale des parents est la carte d'association. Dans l'exemple on a 3 <-> 1 et 4 <-> 2. La partie centrale du parent 1 est recopié dans le fils 2 et inversement. Ensuite on recopie dans le fils 1 les éléments du parents en remplaçant si besoin les villes déjà présente par leur ville associée dans la carte d'association. Et la même chose est effectué avec le fils 2 et le parent 2. Le résultat est le suivant :

Enfant 1 = (3 4 1 2 5 6)
Enfant 2 = (5 1 3 4 6 2)

Le dernier opérateur que j'ai essayé d'implémenter est le Complete Subtour Exchange Crossover (CSEX). Cet opérateur identifie les séquences de villes présentent chez les deux parents. Toutes les permutations d'inversions de ces séquences sont ensuite essayé chez les enfants et seul le meilleur résultat est conservé.

Parent 1 = (1 2 3 5 4 6) et Parent 2 = (5 6 3 2 1 4). Il y a une seule séquence commune : 1 2 3.

Les enfants possibles sont : Enfant 1 = (3 2 1 5 4 6) et Enfant 2 = (5 6 1 2 3 4). Seul le meilleur des deux est gardé.

e. Les variantes

Avec tous ces opérateurs, de nombreuses variantes sont possible. J'ai essayé différentes combinaison d'opérateur de sélection, mutation et croisement ainsi que de taille de population et de taux de mutation. L'opérateur de croisement MOC m'a donné les meilleurs résultats, suivi de peu par PMX, mais CSEX n'a pas du tout fonctionné. Je pense avoir mal utilisé l'opérateur CSEX, sans doute pas avec les bons opérateur de sélection et de mutation car l'opérateur CSEX est sensé donné de meilleurs résultats que les autres d'après **[3]**.

Mon algorithme génétique donnant les meilleurs résultats a une population de 50 individus, dont le meilleur survit à la génération suivante. L'opérateur de mutation est l'opérateur Voisin Zone, avec un taux de mutation à 15 %. Ce qui signifie que 7 individus sur 50 sont issus de la mutation du meilleur individus. L'opérateur de croisement est l'opérateur MOC, la sélection se fait aléatoirement parmi les 10 meilleurs individus.

III. Comparaisons des algorithmes

Afin de savoir quel algorithme rendre, j'ai effectué de nombreux tests sur les instances fournies du TSP. Le tableau 2 résume les résultats de ces tests, les scores sont arrondis pour plus de lisibilité. On peut voir que les meilleurs résultats sont obtenus par le MHC.

Tableau 2
Comparatif de la moyenne des résultats des différents algorithmes sur 10 exécutions de 60 secondes

	A280	Bier127	Ch130
Optimum connu	2579	118282	6110
Mutating Hill Climbing (MHC)	2589	125999	6110
Hill Climbing (HC)	2688	126509	6153
Simulated Annealing (SA)	2636	127093	6157
Ant Colony Optimisation (ACO)	2954	131336	6524
Monte Carlo Tree Search (MCTS)	3002	135682	6746
Algorithme Génétique (GA)	2973	140843	6803
Greedy	3182	147312	7575

Tous les algorithmes donnent de meilleurs résultat que l'heuristique greedy, ce qui montre bien qu'il y a une optimisation effectuée. Les résultats montre aussi que mes algorithmes peuvent être améliorer, avec plus de recherche et de temps.

IV. Conclusion

Au vu des résultats, c'est l'algorithme Mutating Hill Climbing que je rends.

Ce projet m'a énormément plu et j'ai essayé de faire de mon mieux pour avoir les meilleurs algorithmes possibles. Je n'ai malheureusement pas eu le temps d'implémenter ni de tester tout ce que j'avais envie de tester au début du projet. Je n'ai par exemple pas eu le temps d'implémenter cette variante de l'ACO avec des fourmis volante **[4]**. Au début du projet j'étais convaincu que j'allais rendre un algorithme évolutionnaire adaptatif contenant différents opérateurs dans un genre de melting pot des différents algorithme. Mais j'ai été énormément déstabilisé par la difficulté de codé un algorithme évolutionnaire égalant les résultats du recuit simulé. J'ai donc mis ce projet de côté afin de pousser le recuit dans ses retranchements avant de coder le Hill Climbing et d'être choquer par ses résultats. Je pense que je rends aujourd'hui mon meilleur algorithme pour résoudre, non pas n'importe quels instances du voyageur de commerce, mais ces trois instances particulières du voyageur de commerce.

Pourquoi faire simple quand on peut faire compliquer ? Peut être que finalement, dans certains cas, simple ça marche aussi.

V. Bibliographie

[1] Christopher D. Rosin, « Nested Rollout Policy Adaptation for Monte Carlo Tree Search », in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, Barcelona, Catalonia, Spain, July 16-22, 2011

[2] A.J. Umbarkar, P.D. Sheth, « Crossover Operators In Genetic Algorithms : A Review », *ICTACT Journal On Soft Computing,* October 2015, Volume 06, Issue 01

[3] K. Katayama, H. Hirabayashi, H. Narihisa, « Performance Analysis for Crossover Operators of Genetic Algorithm », *Systems and Computers in Japan*, Vol. 30, No. 2, 1999

[4] F. Dahan, K. El Hindi, H. Mathkour, H. AlSalman, « Dynamic Flying Ant Colony Optimization (DFACO) for Solving the Traveling Salesman Problem », on *Sensors 2019*, 19, 1837 (**www.mdpi.com/journal/sensors**)