Université de Nantes
Master 1 ORO

Projet Métaheuristiques 2015
Battle of Metheuristics

Janvier 2016

Guillaume Legru
Florian Delavernhe

Introduction

Pour résoudre un même problème, on peut utiliser plusieurs métaheuristiques différentes. Ces dernières ont des caractéristiques différentes et peuvent fournir une meilleur solution finale que les autres, fournir une solution bonne plus rapidement ou bien être plus rapide en fonction du problème. Nous allons donc, dans ce projet, comparer deux métaheuristiques pour des problèmes similaires de **SPP**. Nous utiliserons dans notre comparaison, le **GRASP** présenté dans notre projet précèdent, ainsi que un algorithme de **Colonie de fourmis**. Dans une première partie, nous présenterons notre implémentation de l'algorithme **ACO**. Et dans une deuxième partie, nous comparerons les résultats de nos deux algorithmes.

1 Ant Colony Optimization

1.1 Construction d'une solution par une fourmi

Dans les problèmes **SPP**, les variables de la solution sont binaires. Notre solution est donc une suite de bits.

Une fourmi construit une solution bit par bit. Nous décidons de commencer par le premier bit et de continuer sur les indices croissants. Pour fixer un bit, nous commençons par vérifier que notre bit n'est pas déjà fixé (c'est à dire qu'il est déjà présent dans une contrainte saturé), puis nous utilisons un random pour choisir entre **0** et **1**, le choix est biaisé en fonction des "phéromones" déposées précédemment. Si nous fixons à **0** alors on passe au bit suivant, sinon nous fixons à **0** toutes les variables ayant au moins une contrainte en commun avec la variable nouvellement fixée.

Nous créons ainsi une solution que nous garantissons réalisable car le premier bit fixé peut l'être à **1** de par l'absence de contraintes saturées, et, notre algorithme fixant à **0** les variables des contraintes nouvellement saturé, il n'est pas possible de fixé à **1** une variable déjà fixée à **0**.

Ainsi, toutes nos fourmis produisent des solutions réalisables, et influencée a chaque bit par les "phéromones", c'est-à-dire les valeurs des solution précédemment trouvées.

1.2 Dépôt des phéromones

Dans un problème de type **SPP** résolu par un algorithme **ACO**, nous souhaitons que les valeurs des solutions trouvées précédemment influent sur la valeur de la nouvelle solution produite. Les valeurs des solutions dépendent des variables fixées, il faut donc influencer sur le choix de valeur **1** ou **0** de chaque bit de la solution. Nous avons donc attribué pour chaque bit une probabilité pour **1** et **0** qui permettra de choisir la valeur du bit : les phéromones. Dans notre algorithme, il y à déjà des phéromones initiaux pour que la fourmi puisse choisir.

Il existe plusieurs paramètres pour régler le dépôt de phéromones : le nombre de phéromone initial, la valeur déposée par chaque fourmi en fonction de ses résultats, ainsi que le taux d'évaporation.

En fonction du nombre de fourmi, une quantité initiale de phéromone trop faible peut diminuer notre espace de recherche et nous confiner dans des maxima locaux : les

premières fourmis influent trop sur le choix des suivantes. Cependant, si ce nombre est trop élevé, nous explorons trop l'espace de recherche, et ne développons pas assez les solutions intéressantes : les phéromones déposés pour une solution intéressante peuvent ne pas être assez signifiante par rapport aux quantités de phéromones initiales. Dans notre algorithme, nous avons donc choisi une quantité de phéromones qui s'ajuste au nombre de fourmi dans chaque itération.

Pour le dépôt de phéromone, chaque fourmi dépose une quantité de phéromone comprise entre **0** et **1** d'après une fonction utilité linéaire : la fourmis avec la meilleur solution dépose **1** phéromone, et celle avec la pire **0**. Les autre déposent en fonction de leurs position entre la distance de la pire à la meilleur : $\frac{x_{meilleur}-x_i}{x_{meilleur}-x_{pire}}$.

Le taux d'évaporation permet que les nouvelles solutions produites par les fourmis à l'itération i ait un impact plus ou moins important sur le fourmis à l'itération $i+1$: sans évaporation, notre espace de recherche se limite trop rapidement, et avec trop d'évaporation l'espace de recherche reste trop vaste et on explore pas assez les bonnes solutions. Le taux d'évaporation est donc à entrer dans la ligne de commande lors de l'exécution.

1.3 Amélioration envisagé

Ils existent de nombreuses variantes d'implémentation d'un algorithme de fourmis. Nous avons donc eu l'occasion de réfléchir a différentes façons d'améliorer notre algorithme.

Pour la construction des solutions des fourmis, nous pensions fixer les bits dans un ordre aléatoires, on peut ainsi sortir de maxima locaux plus facilement.

On peut aussi modifier le dépôt de phéromone initial, et biaisé dés le début le choix afin de favorisé le choix de variable **1**, nous évitons ainsi d'explorer dans les premières itérations de très mauvaise solutions.

2 Comparaison entre ACO et GRASP

Nous allons donc maintenant comparer les valeurs que nous obtenons par notre algorithme ACO présenté ci-dessus avec notre algorithme GRASP présenté dans le projet précédent. Pour cela, nous utilisons les jeux de données fournis par Xavier Delorme de l'école des mine de Saint-Etiènne sur sa page personnelle.

2.1 Résultats

pb_100rnd0100.dat

Temps	GRASP	ACO
1	372	357
5	372	363
10	372	369
30	372	365
60	372	371

pb_100rnd0200.dat

Temps	GRASP	ACO
1	30	32
5	30	32
10	30	32
30	30	32
60	30	32

pb_100rnd0300.dat

Temps	GRASP	ACO
1	169	178
5	176	176
10	176	178
30	176	178
60	176	187

pb_100rnd0400.dat

Temps	GRASP	ACO
1	14	14
5	14	14
10	14	15
30	14	15
60	14	15

pb_100rnd0900.dat

Temps	GRASP	ACO
1	451	385
5	451	395
10	453	390
30	453	400
60	453	401

2.2 Interprétation résultat

En comparant nos résultat reporté ici ainsi que nos expérimentation, on remarque que l'algorithme **ACO** nécessite un temps plus important pour avoir de meilleur résultat, en effet ces meilleurs résultats sont toujours obtenu avec des temps importants. Contrairement à **GRASP** qui trouve toutes ces bonne solution très rapidement. Donc si nous comparons les **GRASP** et **ACO** sur de cours temps, **GRASP** est beaucoup plus efficace. Cependant, on remarque que pour des temps importants notre algorithme des fourmis trouve mieux que notres **GRASP**.

Donc sur des courtes durées, **GRASP** est beaucoup plus efficace, cependant il ne s'améliore que très peu, contrairement a **ACO** qui fourni des solutions moins bonne que **GRASP** pour un temps cours mais qui si on lui laisse plus de temps, fini par fournir une solution qui sera meilleur que celle de **GRASP** pour ce même temps.

Conclusion

L'algorithme des Colonies de fourmis est un algorithme qui à besoin de temps pour trouver une solution convenable. En effet, sur un temps trop court, l'algorithme renvoie un résultat aléatoire, le mécanisme de phéromone ne pouvant pas faire effet. Si on lui laisse un temps suffisant, il trouve de bonne solution pour un nombre important de fourmis (l'espace de recherche est mieux exploré) et un taux d'évaporation moyen.