Métaheuristique pour l'optimisation TP7

Deniz Sungurtekin

December 2020

1 Introduction

Le but de ce travail pratique est d'utiliser le "Genetic Algorithms" pour maximiser une fonction donnée. Dans notre cas, elle est décrite par une table de vérité qui donne la valeur de l'output selon chaque combinaison d'input possible. L'idée est de trouver l'expression booléenne correspondante au mieux à la table de vérité donnée. La méthode utilisée est une métaheuristique de population qui consiste à faire évoluer des individus à travers des générations qui subiront une phase de sélection où les meilleurs candidats seront sélectionnés, une phase de croisement consistant à concevoir la nouvelle génération dont les individus posséderont, selon une certaine probabilité, une partie de l'information de ses deux parents, puis finalement chaque individus aura une certaine chance de subir des mutations. Dans un premier temps, nous allons expliquer la méthodologie utilisée afin d'implémenter la simulation de cette évolution darwiniste puis nous analyserons et discuterons les résultats obtenus.

2 Méthodologie

Pour résoudre le problème donné, l'idée est de simuler l'exécution d'un programme utilisant un langage "post-fix" à travers une machine à pile. Cette machine est représentée par une classe CPU, qui possède une pile prenant les instructions se trouvant dans le programme. Celle-ci est représentée par une séquence de longueur 5 constituée d'opérateur booléen (NOT, OR, XOR, AND) et de quatre variables (X_i). On veut donc trouver le meilleur programme qui suit au mieux la table de vérité donnée. L'espace de recherche est de 8^5 puisque pour les cinq instructions du programme, il y a 8 possibilités puisque les répétitions sont autorisées. Cependant, il existe des séquences invalides puisque nos opérateurs consume une à deux variables, par conséquent il est nécessaire d'avoir des données sur le haut de la pile avant d'utiliser un opérateur et donc d'avoir au préalable une instruction appelant une de nos variables.

Il y a deux donc deux types de comportements lorsqu'une instruction est lu lors de l'exécution du programme. Soit l'instruction est une des variables et la donnée à la position correspondante de la sous-liste data est ajoutée à la pile,

soit l'instruction est une opération. Si c'est le cas on vérifie si la taille de la pile est plus petite que le nombre de variable consumé par l'opérateur. Dans le cas où la condition est vérifiée la pile est vidée et l'exécution s'arrête en retournant la valeur 2, cela permet de faire en sorte que la fitness de ce genre de programme soit de 0 est donc éliminé au fur et à mesure de la sélection. Si la condition n'est pas vérifiée, on "pop" la valeur ou les deux valeurs présentent sur la pile, on effectue l'opération et on ajoute cette valeur au dessus de la pile.

En plus de cette restriction, j'ai décidé d'éviter qu'il soit possible d'avoir deux mêmes déclarations de variable dans le programme, afin d'éliminer les programmes qui possèdent plusieurs déclaration de la même variable pour restreindre l'espace de recherche. Pour ce faire, lors de l'exécution d'un programme je vérifie simplement si celui-ci possède deux mêmes variables, si oui alors l'exécution s'arrête et retourne la valeur 2. Finalement, tout les programmes qui sont valides mais ne consume pas l'entièreté de ces variables sont également éliminé, en d'autre mot si la pile n'est pas de longueur 1 à la fin de l'exécution celui-ci renvoi la valeur 2.

Pour tous les autres programmes, l'exécution effectue donc un "switch statement" sur les instructions du programme et retourne la valeur obtenue. Ensuite, il est donc nécessaire de calculer la fitness du programme qui consiste simplement à compter le nombre de fois où la valeur de l'exécution du programme avec les éléments du "DataSet" est valide. Celui-ci est valide si le dernier élément de la sous-liste "data" est égale à la valeur renvoyé par l'exécution. Avec tout cela, il est donc possible d'utiliser l'algorithme déjà implémenté pour trouver les expressions (programme de longueur 5 généré aléatoirement) qui suit au mieux la table de vérité donné par le "DataSet".

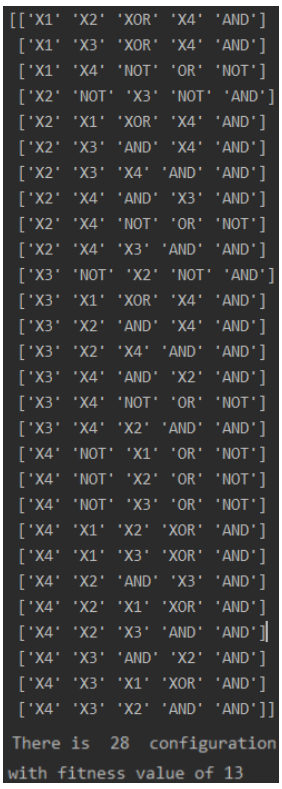
3 Résultat

Premièrement, voici un exemple de résultat obtenue avec une population de 100 individus, 40 générations, $p_c = 0.6$ et $p_m = 0.01$:

```
(['X2', 'X3', 'X4', 'AND', 'AND'], 13
```

Effectivement, cette expression possède une fitness de 13 et donc est valide pour 13 sous-liste data sur les 16 existantes. C'est la fitness maximal que j'ai pu trouver en utilisant mon implémentation. Cependant, il existe plusieurs programme possédant cette fitness, afin de toute les obtenir j'ai lance 3000 fois l'algorithme et stocker l'ensemble des solutions dans une liste. Ensuite, j'ai simplement pris les éléments uniques dans celle-ci.

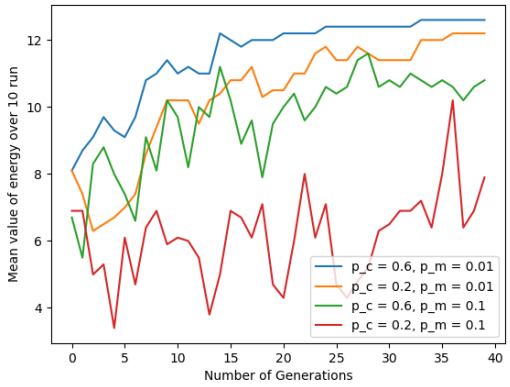
Voici le résultat obtenu:

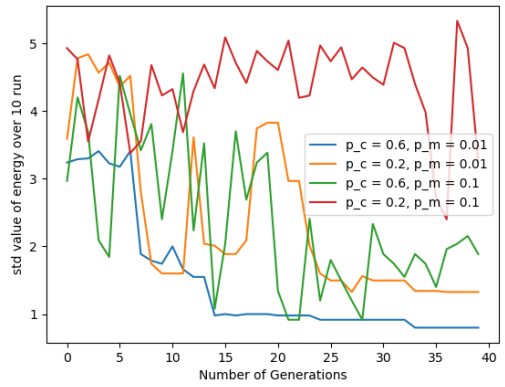


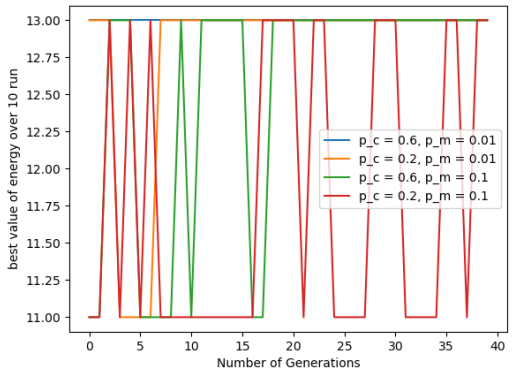
Il est important de noté que dans le lot il y a des configurations qui peuvent être jugées comme étant équivalentes par exemple ["X1","X4","Not","Or","Not"] et ["X4","Not","X1","Or","Not"] ou même ["X2","X4","Not","Or","Not"], puisque dans ce cas la relation entre X1 et X4 est pareil que celle entre X2 et X4 dans les sous-listes "data".

Ensuite, il est important d'analyser l'impact de nos paramètres dans la résolution, particulièrement de p_c et p_m puisque nous avons vu dans le dernier travail pratique que ces deux paramètres ont un grand impact sur le comportement de l'algorithme et sa façon d'explorer l'espace de recherche. Pour cela, j'ai lancé 10 fois l'algorithme avec quartes configurations différentes: $p_c = 0.6$ et $p_m = 0.01$, $p_c = 0.2$ et $p_m = 0.01$, $p_c = 0.6$ et $p_m = 0.1$, $p_c = 0.2$ et $p_m = 0.1$, puis j'ai relevé la valeur moyenne, l'écart-type et la meilleur fitness des 10 runs à chaque générations. J'ai choisis de fixer la taille de la population à 100 et le nombre de génération à 40, puisqu'elle permet déjà d'obtenir une stabilité suffisante dans les résultats.

Voici donc le graphique de chacune de mes mesures:







En observant la moyenne de la fitness, on remarque clairement que le meilleur résultat est obtenu avec $p_c = 0.6$ et $p_m = 0.01$, l'algorithme se diversifie assez pour trouver rapidement une bonne solution, malgré la petite probabilité de mutation. De plus, puisque cette valeur est petite, elle ne suffit pas pour que les individus subissant des mutations négatifs prenant le dessus sur les meilleurs individus lors de la sélection. On peut observer ce phénomène lorsque $p_m = 0.1$, on voit clairement que ces deux configurations ont tendance à obtenir des valeurs de fitness qui augmente et diminue beaucoup plus fréquemment. En outre, cela montre que la probabilité de crossover à un impact beaucoup plus grand que la probabilité de mutation sur la diversification des résultats.

Cette observation se confirme en observant la valeur des écarts-types, on distingue clairement que les configurations avec une probabilité de mutation à 0.01 possèdent les écarts-type les plus petits à la fin des 40 générations. Il est également important de noter que l'écart-type de la configuration avec $p_c = 0.6$ et $p_m = 0.01$ converge beaucoup plus rapidement vers 0, cela exprime la stabilité dans les résultats. Finalement, on voit dans le dernier graphe que le meilleur individu de toute les générations possède une fitness de 13. Par conséquent, la configuration avec $p_c = 0.6$ et $p_m = 0.01$ est la plus efficace.