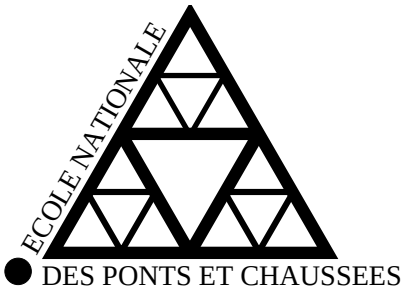

Optimisation - Algorithmes évolutionnaires

Fourmi artificielle
Piste de Santa Fe

Aurélien Boffy
Flavien Billard
Thomas de Soza

23 juin 2005

Introduction

L'objet de ce projet est l'étude du problème de la piste de Santa Fe à partir d'un article [Langdon et Poli, avril 1998]. Notre premier travail a donc été tout naturellement de lire ce dernier et différents autres articles qui faisaient référence à ce même problème de fourmi artificielle, comme [Langdon et Poli, janvier 1998] (notamment car l'article fourni ne redéfinit pas les données du problème).

Une *toolbox* pour MATLAB a été développée pour pouvoir utiliser des algorithmes de programmation génétique : GPLAB. Le problème de la piste de Santa Fe étant très standard, il est même déjà implémenté à titre d'exemple dans la bibliothèque. La deuxième étape du projet consistait donc à lire et à comprendre le code fourni.

Le but de notre projet était à l'origine de modifier ce code pour implémenter les différentes idées présentées dans l'article qui nous avait été fourni pour voir quelles en étaient les répercussions. Nous pouvions aussi essayer de jouer sur les différents paramètres d'évolution (méthode d'initialisation, de croisement, taux de mutation, etc.) pour voir lesquels avaient la plus forte incidence sur les résultats. A dire vrai, le sujet était assez vague et nous laissait beaucoup de liberté.

1 Le problème

Le problème de la fourmi artificielle qui doit suivre la piste de Santa Fe [Koza, 1992] est très connu et a souvent été utilisé pour évaluer différents algorithmes évolutionnaires.

L'environnement Une fourmi artificielle est placée sur une grille de 32×32 cellules (cf. Figure 1). Cet environnement est toroïdal, en ce sens que la fourmi ne tombe jamais : lorsqu'elle arrive sur un bord, elle peut continuer à se déplacer comme si la grille était en fait une petite sphère.

Chaque case peut contenir une pièce de nourriture. Au total, la grille contient 89 pièces de nourriture disposées sur une piste qui comporte 21 tournants. La difficulté réside dans le fait que les pièces de nourriture ne sont pas toutes contiguës : la piste présente des trous. Elle est en effet longue de 144 cases donc il y a 55 cases sur cette piste qui ne contiennent pas de nourriture. A noter que la difficulté de suivre la piste augmente vers la fin car les trous deviennent de plus en plus nombreux.

La fourmi La fourmi est initialement placée en haut à gauche de la grille et elle regarde vers la droite. Son objectif est de manger le plus de nourriture possible. Pour ce faire, elle dispose de 3 mouvements de base : `move` pour avancer d'une case vers l'avant, `left` pour faire un quart de tour vers la gauche et `right` pour faire un quart de tour vers la droite.

Lorsque la fourmi entre dans une case où se trouve de la nourriture, elle la mange.

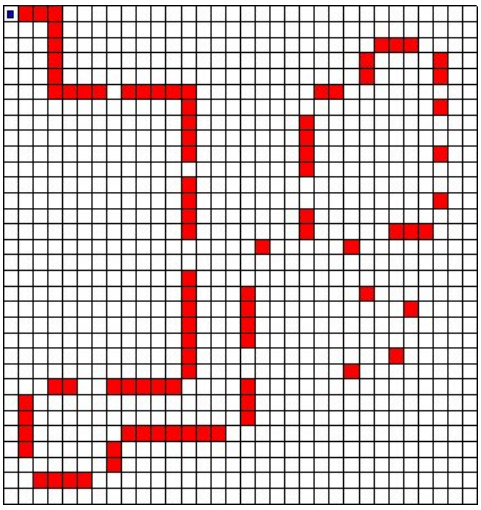

FIG. 1 – La piste de Santa Fe

Le gros problème de la fourmi est qu'elle est presque aveugle. Elle n'est en fait capable de ne voir que la case qui est juste devant elle. Pour voir ce qu'il y a à sa gauche, elle a besoin d'abord de se tourner et pour voir ce qu'il y a plus loin, il faut qu'elle avance.

L'*intelligence* de la fourmi Peut-on vraiment parler d'*intelligence* ? Voici comment raisonne la fourmi : assimilons son cerveau à un arbre dont les feuilles sont les 3 actions de base (`move`, `left` et `right`). Il existe 3 nœuds internes possibles :

- `IfFoodAhead` : ce nœud a toujours 2 fils. Arrivé à ce nœud, s'il y a de la nourriture devant la fourmi, on descend vers la branche gauche, sinon on descend vers la branche droite.
- `Progn2` : lorsqu'on arrive à ce nœud qui a aussi toujours 2 fils, on passe dans un premier temps dans le sous-arbre gauche, puis passe dans le sous-arbre droit.
- `Progn3` : idem sauf que ce nœud est d'arité 3 : on parcourt successivement les 3 sous-arbres.

Ainsi, en parcourant une unique fois l'arbre, la fourmi peut exécuter plusieurs actions différentes pour peu que l'on soit passé par `Progn2` ou `Progn3`.

Par exemple avec l'arbre de la figure 2, la fourmi commence par regarder s'il y a de la nourriture devant elle. Si c'est le cas, on part vers le sous-arbre gauche et la fourmi va en fait obligatoirement avancer d'une case. Si la case devant la fourmi est vide, alors la fourmi va avancer d'une case, se tourner vers la gauche puis avancer d'une nouvelle case.

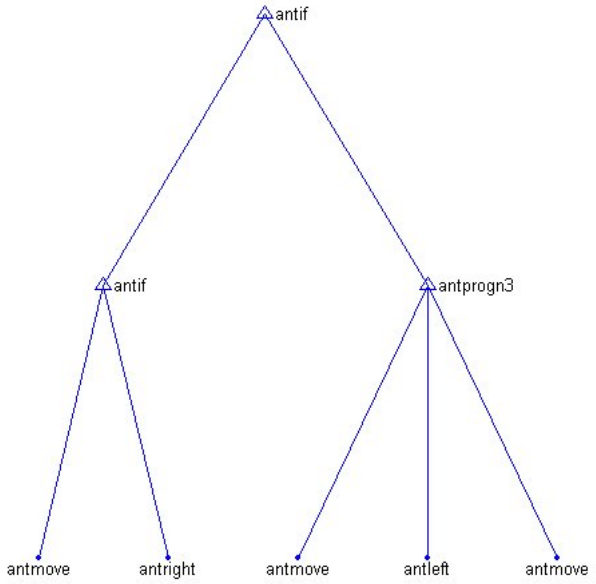

FIG. 2 – Exemple d'arbre

Ce qu'il faut optimiser Le but de l'algorithme est de trouver l'arbre qui permet à la fourmi de manger le plus de nourriture possible. Pour ce faire, elle dispose d'un nombre N de pas de temps limité (généralement 400 ou 600). Chaque action de base (`move`, `left` et `right`) utilise un pas de temps. La *fitness* (ou performance) de la fourmi est donc le nombre de pièces de nourriture qu'elle a mangé pendant les N pas de temps.

2 Amélioration (et correction) du code initial

Une fois le problème bien compris, nous avons essayé de voir ce que donnait l'algorithme de GPLAB. Avec une simple commande, on peut suivre la progression de la *fitness* du meilleur individu ainsi que de nombreux autres paramètres (*fitness* moyenne, écart-type, complexité des arbres, etc.). Lorsque l'algorithme a terminé 10 générations, une fenêtre s'ouvre et affiche l'arbre de la meilleure fourmi obtenue.

Cependant, à la simple lecture de cet arbre, il est difficile de savoir comment se comporte la fourmi. Nous avons donc décidé d'améliorer le code afin de pouvoir observer la trajectoire que suit la fourmi sur la grille. Ceci est en effet impossible avec la version disponible de GPLAB.
Ce dessein nous a permis de nous plonger dans le code et de le comprendre. Lorsqu'enfin nous sommes arrivés à afficher une trajectoire, nous nous sommes aperçus que la fourmi ne mangeait pas le nombre de pièces de nourriture qu'elle était censé (c'est-à-dire autant que sa fitness). De plus, en comparant étape par étape la trajectoire de la fourmi avec l'arbre, nous pouvions remarquer que la fourmi ne réagissait pas comme elle devait le faire. Après de nombreuses vérifica-

tions de notre code, nous nous sommes posés quelques questions sur la justesse de l'algorithme implémenté dans GPLAB.
Nous avons alors relu le code précisément et nous avons remarqué plusieurs erreurs. Certaines étaient même assez grossières (inversion des coordonnées x et y par exemple).

Le temps nécessaire pour tout corriger et pour aboutir à une version qui marche a été assez important pour plusieurs raisons :

- Il a été difficile de se persuader que cette bibliothèque, pourtant amplement utilisée, puisse contenir des erreurs. On a dans ce cas plutôt tendance à croire que c'est notre code qui contient des erreurs.
- Nous ne connaissions pas bien la programmation en MATLAB et nous avons trouvé qu'il était difficile de debugger un programme, notamment car le langage n'est pas typé.
- Il est toujours plus difficile de corriger un programme qui a été écrit par quelqu'un d'autre.
- Nous n'avons pas uniquement corrigé des lignes de code qui contenait des petites erreurs. Nous avons aussi reprogrammé quelques fonctions importantes comme celle qui permet de parcourir l'arbre.

Il est vrai qu'il était quasiment impossible de découvrir ces erreurs sans tracer la trajectoire de la fourmi car on n'a aucun moyen de vérification de l'algorithme. En effet, le code qui était implémenté optimisait évidemment un arbre, mais cet arbre avait d'autres caractéristiques (par exemple, le nœud `IfFoodAhead` n'avait pas le comportement désiré) et la piste n'était pas celle attendue.

Finalement, nous avons réussi à corriger le programme et nous pouvons désormais observer la trajectoire des meilleures fourmis, ce qui permet d'observer quels comportements sont obtenus. Ainsi, il est intéressant d'observer comment se comporte un individu lorsqu'il a perdu la piste. Cherche-t-il à la retrouver ou alors part-il au hasard jusqu'à ce qu'il la recroise ?

3 Réglage des paramètres

Afin de pouvoir comparer nos résultats avec ceux présentés dans l'article, nous devions utiliser les mêmes paramètres. Nous nous sommes donc plongés dans l'aide de GPLAB pour pouvoir être dans les mêmes conditions. Voici quelques exemples des paramètres que nous utilisons :

- Procédure *Ramped half and half* pour l'initialisation des arbres, ce qui permet de favoriser la diversité.
- Une profondeur d'arbre maximale de 6 lors de l'initialisation des arbres. Pendant l'évolution, il n'y a plus de limite sur la taille des arbres.
- Taux de *reproduction* de 10%. Ceci signifie que 10% de la population ne subissent pas les effets des différents opérateurs (croisements, mutations) et sont directement copiés dans la génération suivante.
- Sélection par le mode des tournois entre 7 individus.
- Aucun élitisme
- 90% de croisement

– 10% de mutation

Ceci nous a permis de comprendre quels étaient la signification des différents paramètres utilisés pour les algorithmes de programmation génétique, notamment les procédures d'initialisation, de croisement et de mutation des arbres.

Nous avons aussi dû lancer des batteries de tests pour voir quelle était l'influence des modifications que nous pouvions apporter. Pour avoir des résultats exploitables, il est évidemment impossible de se contenter de lancer une fois le programme. Une véritable étude nécessite des dizaines de *runs* suivis de calculs de moyenne et d'écarts-types notamment.
Il est précisé dans l'article que les tests ont été effectués avec une population de 500 individus et pendant 50 générations. De plus, le nombre de pas de temps N utilisé est 600.
Le problème est que l'exécution d'un tel *run* nécessite un temps bien trop long pour qu'on puisse se permettre des tests en batterie (une telle exécution dure au moins une heure). Nous avons donc effectué tous nos tests avec 30 générations qui comportent chacune 100 fourmis, lesquelles vivent pendant $N = 200$ pas de temps.

4 Obtention de meilleures fourmis

Un des aspects qui a valu au problème de Santa Fe d'être souvent utilisé comme *benchmark* des algorithmes de programmation génétique, est le fait que ces derniers font en général sur ce problème à peine mieux que les algorithmes qui tirent au hasard des arbres (en utilisant des procédures de *Ramped half and half* par exemple) avec une méthode de *Hill Climbing* .
Ce point un peu gênant et d'autres ont été largement décrits dans l'article [Langdon et Poli, janvier 1998] sous le titre "*Why ants are hard ?*".
Un des autres points gênants du problème de Santa Fe est le fait qu'il ne pénalise pas les solutions qui, bien que réalisant des *fitness* élevées, ne sont pas optimales au sens où elles ne suivent pas la piste correctement (par exemple une fourmi qui ne ferait que couper la piste en effectuant des allers-retours sur la grille).

Pour éliminer ces problèmes, [Langdon et Poli, avril 1998] propose, soit de modifier le problème de Santa Fe pour favoriser les algorithmes de programmation génétique, soit de modifier les algorithmes eux-mêmes par exemple en changeant la *fitness* des fourmis.
Nous avons donc implémenté ces deux changements et avons conduit des batteries de tests sur le problème original et sur ces deux nouveaux cas que nous présentons ci-dessous.

4.1 Limitation de la nourriture devant les fourmis

Comme nous l'avons vu, la *fitness* d'une fourmi est le nombre de pièces de nourriture mangées lors de l'exécution de son arbre. Cependant, pour une même *fitness* donnée, deux arbres peuvent être de "qualité" inégale. Par exemple on préferera une fourmi qui suit bien la piste, plutôt qu'une fourmi qui tombe par hasard sur des pièces de nourritures.

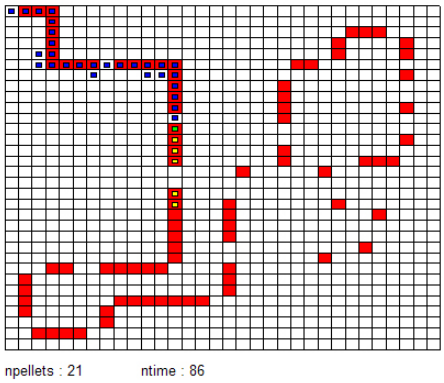

FIG. 3 – Seules 5 pièces de nourritures (en jaune) sont placées sur la piste devant la fourmi (en vert)

Pour pénaliser ce genre de fourmis "opportunistes", on utilise l'amélioration dite *limiting food ahead* suggérée par [Langdon et Poli, avril 1998] : initialement, seules les x premières pièces de nourritures sont placées sur la piste. Les nouvelles pièces de nourritures apparaissent les unes aprés les autres (dans l'ordre du parcours de la piste) lorsque la fourmi mange des pièces déjà présentes, si bien qu'il n'y a à chaque fois que x pièces sur la piste. Cela oblige la fourmi à suivre la piste dans le sens normal du parcours pour manger les pièces (elle ne peut pas attaquer la piste au milieu ou à la fin).
Plus précisément une fourmi qui a perdu la piste ne devrait plus chercher à retrouver celle-ci au hasard en suivant une direction quelconque mais devrait effectuer une recherche locale autour du point où elle l'a perdu (car c'est le seul endroit où elle pourra retrouver des pièces de nourriture).

Avec cette amélioration les arbres retenus à chaque génération ont tendance à donner des fourmis qui suivent mieux la piste et donc récoltent plus de nourriture. Nous avons fait des simulations dans le cas où $x = 5$ pour une série de 10 tests de 30 générations de 100 fourmis.

4.2 Prise en compte de la vitesse dans la *fitness*

L'article qui nous était fourni, préconisait pour modifier la *fitness*, de prendre en compte la vitesse de prise de pièces de nourriture par la fourmi. Pour modéliser cette vitesse sur un *run*, l'article suggérait de définir le rapport entre la distance de la dernière pièce mangée à l'origine (mesurée avec l'abscisse curviligne s de la piste), et le nombre de pas de temps T_l utilisés pour manger cette dernière pièce. Soit $v = s/T_l$ où $1 \leq s \leq 144$ et $1 \leq T_l \leq T_{max}$.

Afin de normaliser cette vitesse, elle était rapportée à la vitesse dans le cas où la dernière pièce est mangée au bout de la piste et au bout du nombre maximum de pas de temps (par exemple 144/200).

Cette définition étant lourde à implémenter et ne fournissant selon l'article

que peu d'améliorations, nous avons préféré implémenter notre propre définition de la vitesse. Notre idée a été de pénaliser une fourmi qui laisse s'écouler trop de pas de temps entre deux pièces mangées consécutivement. Ainsi, à chaque pas de temps, nous retirons 1 à la *fitness* de la fourmi si celle ci ne mange pas une pièce de nourriture.

L'inconvénient de cette technique étant que la *fitness* n'a plus de sens, elle est même souvent négative. Cependant, ce qu'on a pu observer, c'est que cette modification astreint la fourmi à rester sur la piste ou plutôt l'oblige, dès qu'elle sort de la piste, à rebrousser chemin pour la retrouver.

5 Résultats

Nous avons décidé, pour comparer nos améliorations, de les confronter au problème original. Pour les paramètres donnés plus haut, nous avons obtenu les résultats suivants (sur 10 lancers) :

- Pour le problème original : une moyenne de $33,8$ et un écart-type de $7,74$
- Pour le problème avec limitation de la nourriture devant la fourmi : une moyenne de $35,9$ et un écart-type de $13,89$.
- Pour la prise en compte de la vitesse dans la performance : une moyenne de $40,4$ et un écart type de $9,97$.

Il a fallu limiter de nombreux paramètres et nous n'avons donc pas obtenu de solution au problème.

Cependant, en observant les courbes d'évolution de la *fitness* au cours des générations, nous pouvions remarquer qu'elles continuaient souvent à croître à la fin d'un *run*. Ceci laissait donc présager à un meilleur résultat si nous avions pu utiliser plus de générations.

De plus, en limitant le nombre de pas de temps à 200, il était évidemment beaucoup plus difficile d'obtenir une fourmi qui mange les 89 pièces de nourriture, même avec un arbre performant (la fourmi suit bien une partie de la piste mais s'arrête parceque les pas de temps sont écoulés).

Du point de vue des améliorations implémentées, nous remarquons que les tests donnent des résultats plutôt concluants, surtout pour la prise en compte de la vitesse dans la *fitness*. Cependant, vu les écarts-type et le nombre de tests réalisés, il est certain qu'il faut faire attention de ne pas tirer de conclusions trop hatives.

Conclusion

Ce projet nous aura permis de nous familiariser avec la programmation en MATLAB (correction et amélioration du programme initial) et avec les algorithmes génétiques (réglage des paramètres).

Malheureusement, la durée de chaque exécution ne nous a pas permis de réaliser autant de tests que nous l'aurions souhaité et nous n'avons pas pu tester d'autres idées que nous avons eu pendant la durée de ce projet.

Références

[Koza, 1992] Koza, John R. *Genetic Programming : On the Programming of Computers by Natural Selection.* MIT Press, 1992.

[Langdon et Poli, avril 1998] Langdon, W. B. et Poli, R. *Better Trained Ants for Genetic Programming.* University of Birmingham, 1998.

[Langdon et Poli, janvier 1998] Langdon, W. B. et Poli, R. *Why Ants are Hard.* University of Birmingham, 1998.