

Methodes Formelles

TP Artificial Immune Systems

Programmation du problème du voyageur de commerce

19 novembre 2006

1 Objectif du TP

On va compléter le code d'un programme permettant l'approximation d'une solution optimale du problème du voyageur de commerce. Ce programme utilise un système immunitaire artificiel.

2 Fonctions clés de l'algorithme

2.1 Fonction de mutation d'un anticorps

La fonction `muteAc` du fichier `anticorps.c` effectue la mutation d'un anticorps passé en paramètre. Le paramètre `nbMutations` permet de déterminer le nombre de mutation à effectuer dans l'anticorps. Pour des questions de simplicité, on se contentera de faire des permutations individuelles.

```
/* Mutation d'un Anti-Corps */
void muteAc(Ac *ac,int nbMutations)
{
    int ville_1, ville_2, ville_temp, nb_iter;
    /* On nous donne un Anti-Corps, on permutte nbMutations indices du tableaux */

    for(nb_iter = 0; nb_iter < nbMutations; nb_iter++)
    {
        ville_1 = myRandomMinMax(0, ac->nbVilles-1); /* Choix d'une ville au hasard */
        /* assurons nous que l'on permutte pas la meme chose */
        for(ville_2 = ville_1;
            ville_2 == ville_1;
            ville_2 = myRandomMinMax(0, ac->nbVilles-1))
            {;}
        /* permutation */
        ville_temp = ac->parcours[ville_1];
        ac->parcours[ville_1] = ac->parcours[ville_2];
        ac->parcours[ville_2] = ville_temp;
    }
    calculCoutAc(ac);
}
```

2.2 Mutation des clones

Le nombre de permutations à effectuer sur l'individu doit rester raisonnable. Il serait absurde (et inefficace) d'effectuer une mutation tellement forte que le clone n'aurait plus rien à voir avec l'individu original. Le facteur de $(i+1)*NBVILLES/(3*nbClones)$ permet d'assurer une mutation d'au plus un tiers du parcours.

```
/** Il faut un clonage avant, les individus les plus a droite    */
/** doivent correspondre aux clones.                               */
```

```

void mutationClones(Population *population, int nbClones)
{
    /* Plus un clone est a droite, meilleur il est.          */
    /* On peut ainsi faire varier le nombre de mutations a effectuer. */
    /* Dans un premier temps, le nombre de mutations peut etre fixe. */
    int i;
    for(i=0; i<nbClones; i++)
    {
        muteAc(population->individus[population->nbIndividus-i-1], (i+1)*NBVILLES/(3*nbClones));
    }
}

```

2.3 Selection BestOfMeilleurs

```

/** Les meilleurs de (Meilleurs + Clones) sont gardes et sont les    */
/** les plus a droite.                                              */
/** Pour cela, on trie une sous population (fct triSousPopulation)  */
/** de 2*nbClones individus (Meilleurs + Clones) et on garde les   */
/** nbClones meilleurs.                                            */
/** ==> La population diminue de nbClones individus                */
void selectionBestOfMeilleursEtClones(Population *population,
                                     int nbClones)
{
    int i;
    Ac * temp;
    /* Tri de la sous population de taille 2*nbClones, en partant de la droite */
    triSousPopulation(population, population->nbIndividus - 2*nbClones, 2*nbClones);
    /* On copier 2*nbClones Ã partir de la droite sur la gauche */

    for(i=0; i < nbClones; i++)
    {
        temp = population->individus[population->nbIndividus - 2*nbClones + i];
        population->individus[population->nbIndividus - 2*nbClones + i] =
            population->individus[population->nbIndividus - nbClones + i];
        population->individus[population->nbIndividus - nbClones + i] = temp;
    }
    population->nbIndividus = population->nbIndividus - nbClones;
}

```

2.4 Remplacement des moins bons

```

/** Les moins bons doivent etre a gauche (apres un tri par exemple) */
void remplacementMoinsBons(Population *population, int nbNouveaux)
{
    int i;

    for(i=0; i<nbNouveaux; i++)
    {
        genereAc((population->individus[i]));
    }
}

```

2.5 Mutation des moins bons

De même que pour la mutation des clones, on ne mute au plus qu'un tiers du parcours.

```

/** Les moins bons doivent etre a gauche (apres un tri par exemple) */
void mutationMoinsBons(Population *population, int nbMoinsBons)
{
    /* Plus un mauvais est a droite, meilleur il est ! */
    /* On peut ainsi faire varier le nombre de mutations a effectuer. */
    /* Dans un premier temps, le nombre de mutations peut etre fixe. */
    int i;
    for(i=0; i<nbMoinsBons; i++)
    {
        muteAc(*(population->individus + (population->nbIndividus) - i - 1),
                (i+1)*NBVILLES/(3*nbMoinsBons));
    }
}

```

2.6 Programme principal

```

/***** Les choses serieuses commencent *****/
/* Parametres : */
/* &p(p :population), */
generePopulation(&p,nbIndividus,nbIndividus+nbClones); /* nbIndividus, */
/* maxNbIndividus */

/* Tri d'une Population avec un COUT DECROISSANT => */
/* les moins bons sont a gauche, les meilleurs sont a droite */
triPopulation(&p);
tour=0;

while (tour !=nbGenerations)
{
    tour++;

    /* Au debut de la boucle, la population doit etre deja trie */
    /* On clone les meilleurs, on les muttes,
       et on garde les meilleurs des meilleurs + clones */
    clonageMeilleurs(&p, NBCLONES);
    mutationClones(&p, NBCLONES);
    selectionBestOfMeilleursEtClones(&p, NBCLONES);
    /* On injecte du "sang neuf" régulièrement, mais pas tout le temps;-) */
    /* Sinon, on mute les mauvais élèves ... */
    if( (tour % NBGENERATIONSINJECTION) == 0)
    {
        remplacementMoinsBons(&p, NBNouveaux);
    }
    else
    {
        mutationMoinsBons(&p, NBCLONES);
    }

    triPopulation(&p); /* Pour trouver le meilleur, la */
    meilleur=meilleurIndividu(&p); /* population doit etre deja trie */
    if (LeMeilleur.cout>meilleur->cout)
    {
        dessineParcoursAc(fdGnuplot,meilleur);
        printCoutAc(meilleur);
        cloneAc(meilleur,&LeMeilleur);
        ecrireCout(fdCout,tour,meilleur->cout);
    }
}

```

```

    visualiserCout(fdGnuplotCout,fileNameCout);
}
#if NBVILLES==8 || NBVILLES==16 || NBVILLES==30
    if (compareAc(&LeMeilleur,&BestOf)==0) {
        printf("Stop! (meilleure solution trouvee)\n");
        break;
    }
#endif
}

```

3 Résultats obtenus, influence des paramètres

3.1 16 villes

Observons le comportement de notre algorithme avec 16 villes. Les 16 villes, toujours placées au même endroit, nous permettent d'étudier l'influence des paramètres.

3.1.1 Influence de la taille de la population (50% de selection, 20% d'injection, toutes les 20 générations)

Avec 10 individus, la solution n'est pas trouvée en 100000 générations.

Avec 100 individus, la solution est trouvée en environ 150 générations.

Avec 1000 individus, la solution est trouvée en environ 40 générations.

Avec 10000 individus, la solution est trouvée en environ 30 générations.

Avec 100000 individus, le résultat est obtenu après 30 générations, mais le temps de traitement est bien plus long!

3.1.2 Influence des autres paramètres

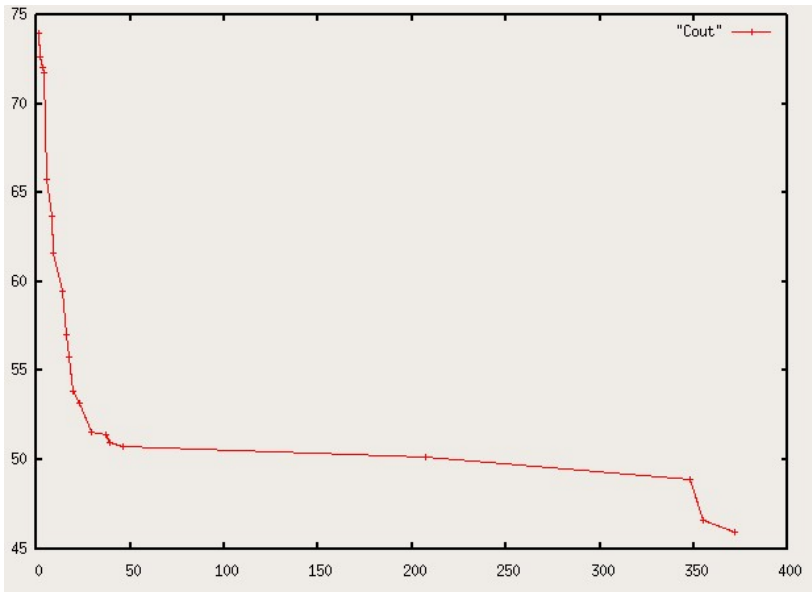
Il est assez difficile d'évaluer l'influence des autres paramètres de façon objective. En effet, la trivialité du problème ne nous permet pas d'observer des variations significatives dans les résultats. En testant avec une trentaine de ville (mais à chaque fois, avec une disposition différente), on remarque cependant :

- Qu'une fréquence élevée d'injection (au plus, tout les 5 tours), permet d'accélérer l'obtention de l'approximation. Au delà, les anticorps ne semble pas avoir le temps de s'améliorer. Des injections trop rares mènent à des extrema locaux.
- Qu'un taux trop faible de clones ralentit considérablement la bonne marche de l'algorithme. Un taux de 50% semble adéquat.
- Qu'un taux trop élevé d'injection (remplacement des moins bons) amène au remplacement des bons, ou des mauvais en progrès. 10 à 20 % semble être un bon taux.

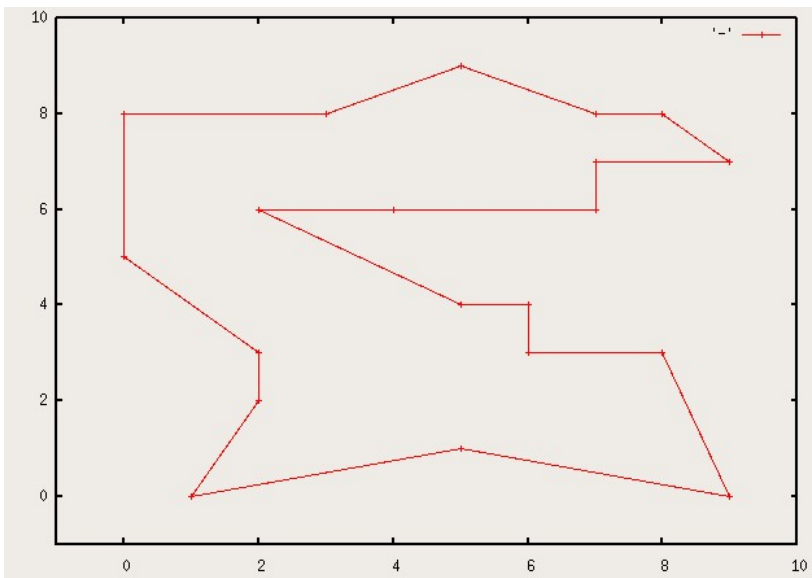
L'influence du nombre de génération est mis en évidence sur l'exemple suivant.

3.2 20 villes, une population de 1000 individus, sur 1000 générations

L'aspect de la courbe de coût est typique de l'ensemble des essais effectués : le nombre de générations nécessaires pour obtenir une amélioration du résultat est exponentiel. Sur l'exemple pris ici, le système immunitaire a globalement donné une approximation acceptable de la solution optimale un peu avant la cinquantième génération. Il a fallu attendre quelques 300 générations supplémentaires pour obtenir une amélioration significative (en l'occurrence, la solution optimale).



Evolution du cout de la meilleurs solution, en fonction du nombre de générations



Solution optimale trouvée.