Developpez.com
Club des développeurs

Application d'un algorithme génétique au problème du voyageur de commerce

par Pierre Schwartz (retour aux articles)

Date de publication : 25 juin 2008

Dernière mise à jour :

Le but de cet article est d'illustrer l'implémentation d'un algorithme génétique sur un exemple concret de recherche opérationnelle : le problème NP complet dit du voyageur de commerce. Prérequis : algorithmes génétiques, programmation objet, C++

I - Introduction..3
II - Déroulement global de l'algorithme..4
II-1 - Evolution des générations..4
II-2 - Eviter la régression à tout prix..5
II-3 - Création des individus..6
III - Définition du problème..7
IV - Spécialisation suivant le type d'individus..8
V - De l'art de ne pas laisser la machine chercher naïvement..11
VI - Diagrammes de convergence..12
VI-1 - Recherche sur 10 villes..12
VI-2 - Recherche sur 15 villes..13
VI-3 - Recherche sur 50 villes..14
VII - Conclusion..15

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - Introduction

Rappel : Le problème du voyageur de commerce est le suivant :

- soient N villes séparées chacune d'elles par une distance D(i,j)
- trouver le chemin le plus court passant par toutes les villes et retournant à son point de départ.

En d'autres termes, il s'agit de déterminer le plus court chemin hamiltonien sur un graphe donné. La taille de l'espace de solutions est 1/2(N-1)!.

L'utilisation d'un algorithme génétique est particulièrement adaptée à cet exemple vu l'indépendance des solutions potentielles, la simplicité de créer des solutions potentielles et vu sa formalisation 'relativement' simple. Dans cet article, je vais proposer une implémentation en C++, la plus générique possible, adaptable à d'autres problèmes.

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II - Déroulement global de l'algorithme

II-1 - Evolution des générations

Une approche objet nous fournit une abstraction du déroulement de l'algorithme pour ne plus lier les spécificités du problème qu'au type des individus considérés. De manière tout à fait transparente, le déroulement est le suivant :

- création d'une population aléatoire
- pour chaque génération
- déterminer une liste d'individus à muter
- faire muter ces individus
- déterminer une liste d'individus à croiser
- croiser ces individus
- injecter ces 2 nouvelles listes d'individus dans la population
- choisir les individus pour la génération suivante

L'élément principal, auquel bénéficiera l'abstraction du type des individus, possède la logique macroscopique de l'algorithme. Il est à la fois central et très simple : il ne possède comme seul membre que la population en cours de traitement. Cependant, libre à vous de l'enrichir si le besoin s'en fait sentir.

```
void runNgenerations(int nbGenerations){
    int size = int(population.size());

    int itmp = 0;
    for (int i = 0; i<nbGenerations; i++){
        // sélection des meilleurs éléments
        std::set<individual> toMutate;
        selectIndividuals( int(size /1.5), toMutate);

        // croisements / mutations
        std::set<individual> mutated;
        mutate(toMutate, .4, mutated);

        std::set<individual> toCross;
        selectIndividuals( size /2, toCross);

        std::set<individual> crossed;
        cross(toCross, .8, crossed);

        // insertion dans population
        injectIntoPopulation(mutated);
        injectIntoPopulation(crossed);

        troncatePopulation(size);

        individual &tmp = *population.begin();
        std::cout << i << " " << tmp.fitness << " " << tmp.display() << std::endl;
    }
}
```

Le conteneur d'individus sera très souvent sollicité pour ajouter, rechercher et supprimer de nouveaux éléments. L'utilisation d'un conteneur trié s'avèrera des plus utiles. L'unicité des individus n'est pas nécessaire, cependant des doublons risquent d'augmenter le volume de données et le temps de calcul, bien qu'ils puissent aussi favoriser certains individus lors des choix aléatoires. J'ai choisi d'utiliser un std::set<individual>

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Cet élément central, qui sera le moteur de l'algorithme génétique possède toutes les fonctions générales de manipulation de la population : création, sélection, insertion, suppression.

```
void injectIntoPopulation(std::set<individual>& l){
        for (std::set<individual>::iterator i = l.begin(); i != l.end(); i++)
                population.insert(*i);

    }

    void troncatePopulation(int n){
        int nbFound = 0;
        std::set<individual>::iterator i = population.begin();
        while (i != population.end() && nbFound < n){
            i++;
            nbFound++;
        }

        if (i != population.end())
            population.erase(i, population.end());
    }
```

II-2 - Eviter la régression à tout prix

Le but étant quand même d'améliorer les solutions proposées, nous devons nous prémunir d'éventuelles régressions. J'ai choisi de me baser sur le meilleur élément d'une génération pour juger de l'avancement de la résolution. Bien qu'il puisse s'agir d'un extremum local sans chance d'amélioration, cet individu correspond à la meilleure solution trouvée au problème et en est la réponse si l'exécution devait s'arrêter.

La méthode basique consiste à toujours reprendre les meilleurs individus d'une génération sur l'autre, on est sûr que la génération suivante sera au moins aussi bonne que la précédente. On évite le risque de perdre les meilleurs individus par une sélection malchanceuse les oubliant, malgré leur meilleure qualité. Oui, mais qu'en est-il des autres individus moins bons ? Ils sont peut-être la clef de la convergence optimale sans que nous le sachions encore. C'est pourquoi la solution de sécurité consiste à ne jeter aucun individu d'une génération sur l'autre. Evidemment je vous laisse imaginer le temps de calcul qui va croître de génération en génération.

La solution naïve serait de faire une simple sélection (roulette?) pour déterminer les individus de la prochaine génération, mais on amène le risque de régression. Cette solution a l'avantage de rester vraiment neutre pour ne mettre en avant que les individus qui le méritent, sans a priori sur la solution finale. Un compromis pourrait être

Nouvelle génération = sélection[roulette?] (ancienne génération + éléments mutés + éléments croisés) + échantillon des meilleurs individus de la génération courante

Dans tous les cas nous sommes obligés de jeter certains individus, autant prendre le moins de risques.

La méthode de sélection est cependant libre. Je propose un système de roulette pour privilégier les meilleurs individus. Les meilleurs fitness étant décroissantes, j'ai juste ajouté un artifice pour inverser cette tendance.

```

    void selectIndividuals(int nbToSelect, std::set<individual>& selected){
        int globalSum = 0;

        std::set<individual>::iterator it = population.end(); it--;
        int maxFitness = it->fitness;

        for (std::set<individual>::iterator i = population.begin(); i!=population.end(); i++)
            globalSum += maxFitness  - i->fitness;    // inversion du sens des fitness

        std::set<individual>::iterator ii = population.begin();
        // conservation des 5 meilleurs individus
        for (int i=0; i<5; i++, ii++)
```


Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
            selected.insert(*ii);

   // roulettes de sélection
        for (int i = 0; i<nbToSelect ; i++){
            int found = rand()%globalSum;
            int a=0;
            std::set<individual>::iterator ite = population.begin();
            a += maxFitness  - ite->fitness;

            while (a<found){
                ite++;
                a += maxFitness  - ite->fitness;
            }

            if (selected.find(*ite) == selected.end())
                selected.insert(*ite);
        }
    }
```

On peut aussi simplement conserver les meilleurs éléments pour la sélection. Le conteneur est trié, il suffit de prendre les premiers éléments :

```
        std::set<individual>::iterator ii = population.begin();
        for (int i=0; i<nbToSelect; i++){
            ii++;
            selected.insert(*ii);
        }
```

II-3 - Création des individus

La création d'une population dépend fortement du problème, c'est encore un élément à basculer dans l'individu :

```
void individual::createRandomly(){
    int nbValues = int(data.distances.size());

    // on commence à la position 0
    solution.push_back(0);
    std::vector<int> remainingPossibilities;
    for (int i=1; i<nbValues; i++)
        remainingPossibilities.push_back(i);

    int a=1;

 // on recherche les villes suivantes
    while (a < nbValues){
        solution.push_back(data.selectNearCity(a));
        a++;
    }

 // on rend la solution cohérente avec le problème
    makeCoherent();

    calculateQuality();
}
```


Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III - Définition du problème

Pour rester le plus simple possible, j'ai choisi de représenter le problème par le graphe des villes : une matrice aléatoire, symétrique et de diagonale nulle. Ainsi les distances sont équivalentes selon leur orientation. Vous pouvez bien sûr supprimer la symétrie, l'algorithme se débrouillera avec la matrice donnée.

Chaque problème vient avec sa définition de règles, ses données et ses contraintes. Le moteur d'algorithmes n'a pas besoin directement de connaître le problème à résoudre, seuls les individus doivent pouvoir vérifier les contraintes.

Chaque individu doit simplement avoir une référence ou un pointeur sur le jeu de données à résoudre. Dans notre exemple du voyageur de commerce, chaque individu possède une simple référence sur l'instance des règles.

Le problème peut simplement se décrire :

```
class problem{
public:
    problem(int);
    int selectNearCity(int);

    std::vector<std::vector<int> > distances;

protected:
    std::vector< std::map<int, int> > dual;
    std::vector< int > maxDistances;
    std::vector<int > globalSums;
};
```

On y trouve bien sûr les distances entre les villes (distances) mais aussi d'autres structures telles que les données duales ou encore les éléments de base des roulettes de sélection.

On retrouve aussi une petite fonction permettant de choisir une ville à proximité d'une ville donnée en tenant compte des données duales, implémenté sous la forme d'une roulette.

```
int problem::selectNearCity(int from){
    int f = rand()%globalSums[from];
    int i = 0;
    std::map<int, int>::iterator it = dual[from].begin();
    i += maxDistances[from] - it->first;

    // roulette de sélection
 while (i<f && it!=dual[from].end()){
        i += maxDistances[from] - it->first;
        it++;
    }

 if (it == dual[from].end())
  return dual[from][ int(dual[from].size()-1) ];

    return it->second;
}
```


Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

IV - Spécialisation suivant le type d'individus

Un individu correspond à une solution possible au problème. Quel que soit le problème traité, on devra être en mesure d'attribuer une fitness ou une qualité à chacun de nos individus.
Le problème traité déterminera la logique microscopique : la création aléatoire, les méthodes de calcul de qualité, de mutations et de croisements des individus. Il suffit de les énoncer pour voir apparaître des fonctions virtuelles pour ces opérations.

- createRandomly
- calculateQuality
- mutate
- cross

Voilà, le squelette est posé, reste maintenant à spécifier les traitements relatifs à notre problème du voyageur de commerce.

Dans le cas du voyageur de commerce, chaque individu est composé d'une liste de villes. Sa qualité est donnée par la distance totale du chemin qu'il représente. Ce problème est mono critères, la fitness de chaque individu reflète directement sa résolution du problème, ce qui simplifie la résolution et l'implémentation.

```
class individual{
public:
    individual(problem &);     // constructeur à partir du jeu de données
    individual(const individual&);
    individual& operator =(const individual&);
    ~individual();

    void createRandomly();     // création aléatoire
    void mutate();                 // mutation
    individual operator*(individual&);    // croisement

    std::vector<int> solution;        // chemin retenu dans le graphe
    int fitness;                            // qualité de la solution
    problem &data;         // référence sur les données du problème

    std::string display();
    void makeCoherent();     // vérifier que this répond bien au problème

protected:
    void calculateQuality();
};
```

La mutation d'un individu peut être toute opération aléatoire qui le fait passer indépendamment à un autre chemin hamiltonien. On peut par exemple permuter un ou plusieurs couples de villes pour retomber sur un chemin hamiltonien potentiellement meilleur. On peut considérer la mutation de deux manières : elle peut créer un nouvel individu qui correspondra à l'individu muté ou bien elle pourra directement modifier l'individu muté. La première approche permet de conserver l'individu originel au cas où sa mutation l'aurait rendu moins bon. C'est ce que j'ai choisi dans mon implémentation.

```
void individual::mutate(){
    // intervertir 2 villes
    int l = int(solution.size());

 // on détermine le nombre de couples à intervertir
    int nb = rand()%5;
    for (int i=0; i<nb; i++){
        int a = rand()%l;
        int b = rand()%l;
```


Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        int tmp = solution[a];
        solution[a] = solution[b];
        solution[b] = tmp;
    }

   // on détermine la fitness de la solution créée
    calculateQuality();
}
```

Le croisement de solutions peut être toute opération permettant d'obtenir une nouvelle solution à partir de 2 ou plus individus existants. J'ai choisi d'effectuer un croisement multipoints en reprenant des parties de chaque individu parent pour créer un nouvel individu. Il faut cependant toujours conserver la cohérence des solutions créées en rétablissant le caractère hamiltonien d'un individu créé. De manière plus évidente que pour la mutation, le croisement crée un nouvel élément. On aurait cependant pu modifier l'un des parents, amenant le même risque que dans la mutation sans conservation d'historique.

```
individual individual::operator*(individual& i){

    int l = int(i.solution.size());

    // croisement multipoints
    int a = rand()%l;
    int b = rand()%l;
    int min1, min2;

    min1 = (a<b)?a:b;
    min2 = (a<b)?b:a;

   // pour la première inversion de composantes
    for (int ii=0; ii<min1; ii++){
        int tmp =  solution[ii];
        solution[ii] = i.solution[ii];
        i.solution[ii] = tmp;
    }

 // pour la seconde inversion de composantes
    for (int ii=min1; ii<min2; ii++){
        int tmp =  i.solution[ii];
        i.solution[ii] = solution[ii];
        solution[ii] = tmp;
    }

 // pour la troisième inversion de composantes (2 points de croisement = 3 composantes)
    for (int ii=min2; ii<l; ii++){
        int tmp =  solution[ii];
        solution[ii] = i.solution[ii];
        i.solution[ii] = tmp;
    }

 // on rend les individus cohérents
    makeCoherent();
    i.makeCoherent();

    calculateQuality();
    i.calculateQuality();

    return *this;
}
```

Pour profiter du tri de la population manipulée par le moteur d'algorithme génétique, on pensera à spécifier un opérateur de tri pour la classe d'individus manipulée :

```
bool operator<(individu& i1, individu& i2){
    return i1.fitness< i2.fitness;
}
```


Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Note : on remarque que les traitements aléatoires sont pléthore dans un algorithme génétique, n'hésitez pas à utiliser un autre générateur que rand() si le besoin s'en fait sentir.

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

V - De l'art de ne pas laisser la machine chercher naïvement

Si déjà on a réussi à reléguer les données du problème dans la classe individual pour abstraire le moteur d'algorithmes, autant faire que les individus utilisent effectivement ces données, et pas que pour calculer leur propre fitness.

Chacun des traitements des individus pourrait être orienté pour résoudre plus rapidement les contraintes du problème. Par exemple, la création aléatoire des individus pourrait privilégier des éléments consécutifs proches, de même les mutations pourraient essayer de relier des villes proches. La considération de la matrice des villes fait ainsi apparaître sa matrice duale, comportant pour chaque ville, celles qui sont les plus proches. Il suffit pour ça de chercher le successeur d'une ville par une roulette de sélection sur la matrice duale, en tenant toujours compte des villes déjà utilisées.

Ce processus de sélection par roulette peut être rattaché directement aux données du problème, par l'introduction d'un objet 'problème' à part entière, contenant toutes les données et les contraintes de notre graphe, y compris la fameuse matrice duale. Il suffit ensuite que les individus s'y réfèrent lorsqu'ils ont besoin de connaître une ville qui peut potentiellement suivre une ville donnée.

En agissant ainsi, on va s'économiser de nombreuses mutations ou croisements perdus d'avance, ou des individus visiblement trop mauvais. Mais évidemment, le revers de la médaille concernera le comportement de l'algorithme. En élaguant trop largement des solutions mauvaises au premier abord, on prendra le risque de passer à côté de la solution optimale en s'enfermant dans un extremum local.

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

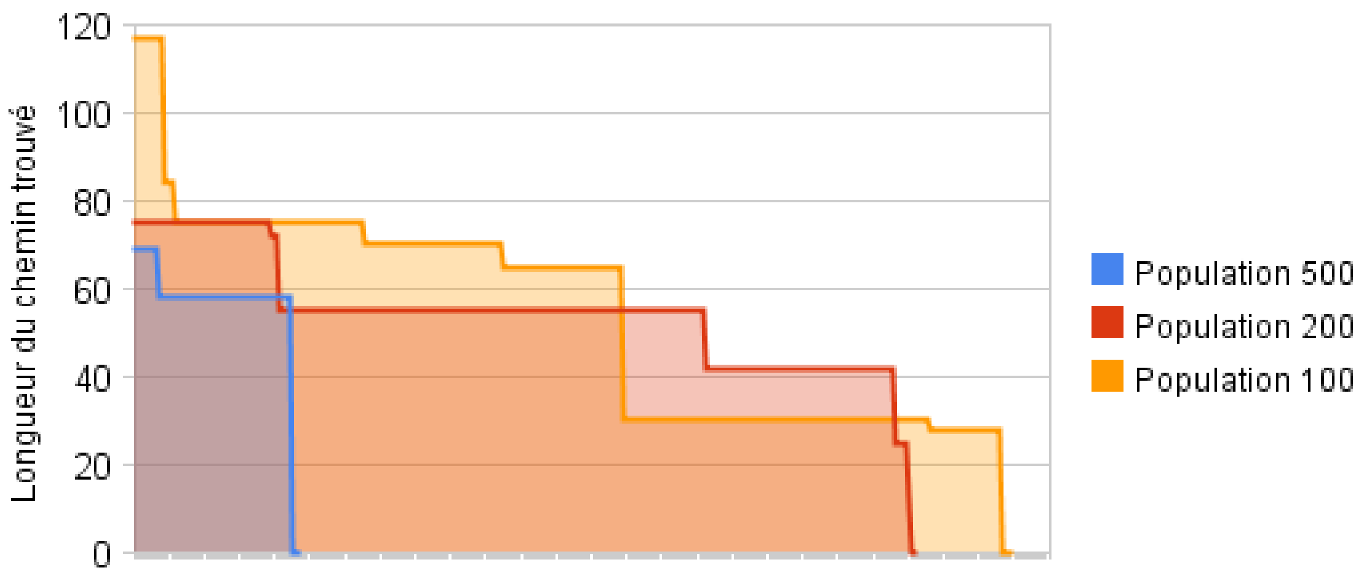
VI - Diagrammes de convergence

Le programmeur a la main sur plusieurs paramètres pour affiner la recherche de solutions :

- La taille de la population
- L'évolution de la taille de la population

Le hasard étant omniprésent, les exécutions ne démarrent pas toutes avec le même meilleur individu, le départ des courbes n'est donc pas le même partout. J'ai choisi de manipuler des populations de taille fixe, voici les résultats que j'ai pu obtenir.

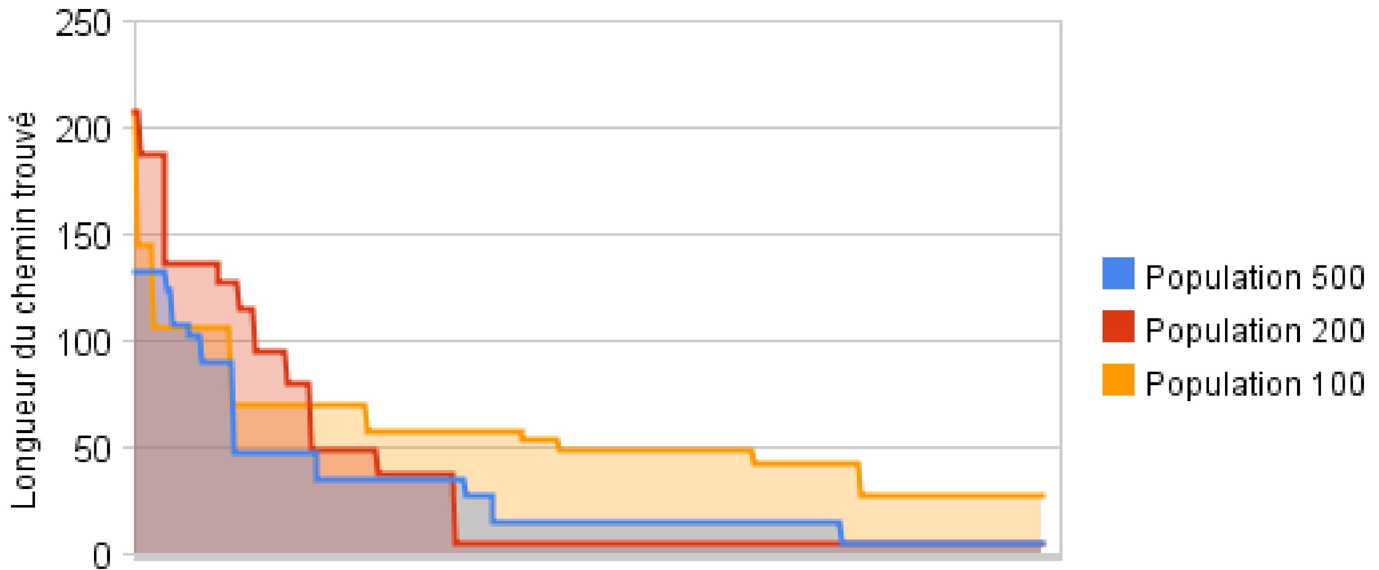
VI-1 - Recherche sur 10 villes



Différentes exécutions avec des populations de 100, 200 et 500 ont toutes convergé assez rapidement. On remarquera cependant que la convergence avec 500 individus a nécessité moins de générations. Autre point notable : l'exécution avec une population de 100 a été meilleure que l'exécution avec 200 individus pendant un court instant, le hasard est facétieux.

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

VI-2 - Recherche sur 15 villes

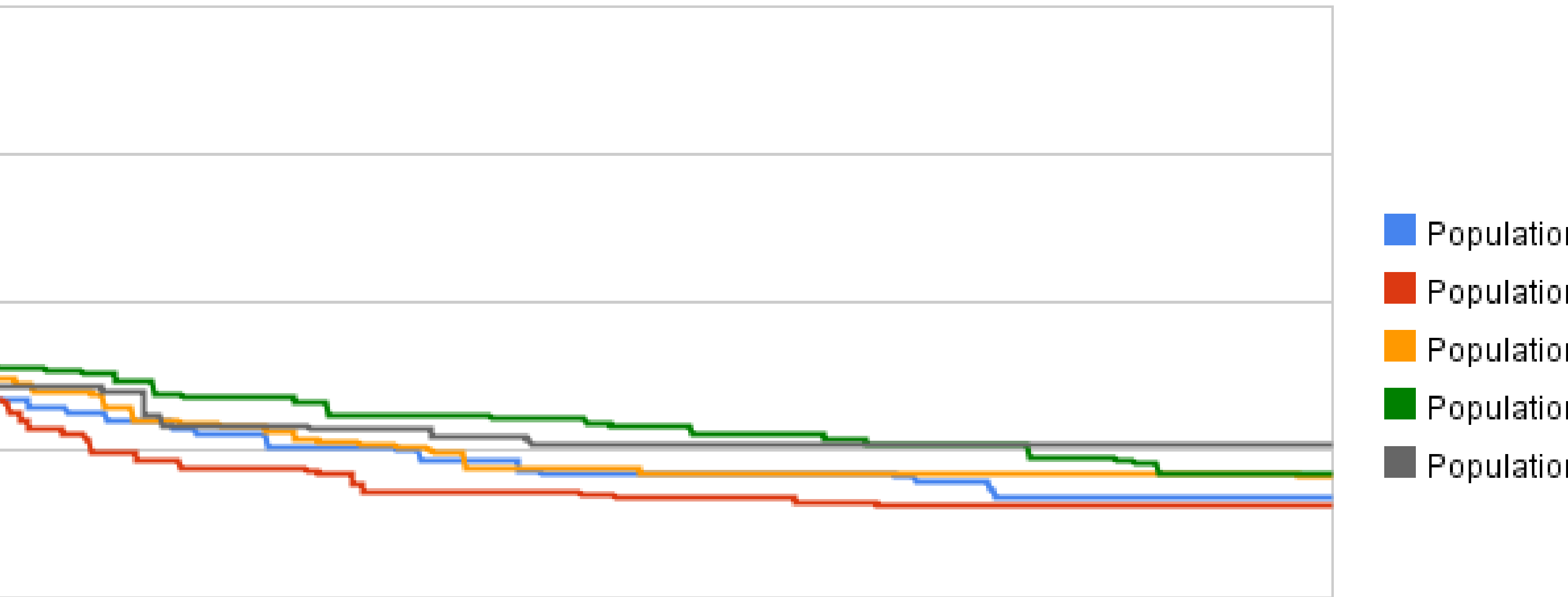


Pour un espace de recherche 200.000 fois plus grand que précédemment, les exécutions n'ont pas convergé en 2000 générations, cependant les solutions proposées sont très très proches de la solution optimale (longueur 20 contre une longueur optimale de 15). Comme précédemment, la convergence est plus lente avec une population moins importante, même si on remarque que les exécutions avec 500 et 200 individus sont tour à tour meilleures.

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

VI-3 - Recherche sur 50 villes

50 villes



Synthèse sur 10.000 générations

Pour un espace de recherche près de 7.10^51 fois plus grand que précédemment, les exécutions n'ont pas convergé au bout de 10.000 générations. L'algorithme a réussi à diviser par 5 la taille du chemin hamiltonien depuis les individus aléatoires. J'ai stoppé l'exécution à 10.000 générations, mais la courbe de progression montre bien une amélioration continue des solutions.

Sur les diagrammes de convergence, les vitesses de progression sont rapides au début de l'exécution puis de plus en plus lente, ce qui peut justement nous inciter à arrêter prématurément l'exécution pour garantir un rapport qualité/temps optimal. Et sur cet exemple encore plus que sur les précédents, on se rend compte du caractère improbable des tailles des populations : les 5 exécutions fournissent des résultats similaires. D'un point de vue statistique, il aurait fallu d'un seul individu pour résoudre le problème, en une seule génération, mais on s'aperçoit quand même de l'avantage d'utiliser des populations nombreuses.

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

VII - Conclusion

Bien que nécessitant beaucoup de temps à l'exécution, les algorithmes génétiques n'ont pas à avoir honte de leur prestation dans la résolution du problème du voyageur de commerce. N'étant pas faits pour fournir des solutions optimales, ils ont néanmoins réussi à en déterminer certaines. Bien que non optimales, les solutions proposées pour la résolution à 50 villes n'ont cessé de s'améliorer et s'amélioreraient encore si on poursuivait l'exécution.

Télécharger les sources C++ : **implementation.zip (7 Ko)**

Merci à toute l'équipe de la rubrique Algo pour la relecture technique et à **raptor70** pour la relecture orthographique.

Copyright © 2006 - Pierre Schwartz. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.