Algorithmes évolutionnistes

Description. La conception d'un bon algorithme d'optimisation ne consiste pas seulement à trouver les bons mécanismes de recherche mais encore faut-il trouver les bons paramètres qui permettent de réguler l'influence de chaque mécanisme : les tendances à l'exploration, à l'exploitation, la convergence de la population, la stabilité des résultats (taux de succès)…
Les algorithmes évolutionnistes dont les algorithmes génétiques sont des algorithmes fortement paramétrables : taille de la population, nombre de générations limite, taux de croisement et de mutation, type d'opérateur de sélection, de croisement, de mutation et de remplacement.
Dans ce TP, nous nous proposons d'effectuer une étude expérimentale sur le paramétrage d'un algorithme évolutionniste. L'objectif est d'analyser l'impact des différents paramètres sur la recherche et ceci face à des fonctions fitness de différentes natures.

I. Introduction

1. Copiez le fichier TP_AlgorithmeEvolutionniste.zip sur votre compte local à partir Moodle (AG41, https://moodle.utbm.fr/) et procédez à sa décompression.
2. Chargez le programme dans un environnement de programmation C++, compilez (en tapant 'make' en ligne de commande ou en utilisant Code::Blocks par exemple) et exécutez (en tapant 'algo_evo.exe' en ligne de commande).
3. Le but est de cet algorithme évolutionniste est de trouver le plus court chemin passant par 10 et 50 villes (problème du voyageur de commerce).
   Les meilleures solutions trouvées pour le problème à 10 villes et 50 villes sont 3473km et 5644km.

II. Présentation du squelette du programme

4 classes (chromosome, population, random, ae avec des fichiers .h et .cpp), une fonction main.cpp, 2 fichiers de distances et un fichier 'make'

1. La classe chromosome
   a. attributs : genes, taille, fitness
   Comment est codée une solution ?
   b. méthodes : constructeur chromosome(), evaluer(), afficher(), ordonner(), echange_2_genes(), identique()…

3. La classe population
   a. attributs : individu, taille_pop
   b. méthodes : constructeur population(), nb_chromosomes_similaires()

4. La classe Radom : 2 méthodes : randomize() et aleatoire()

5. La classe Algorithme Génétique
   a. attributs : nbgenerations, taille_pop, taux_mutation, taille_chromosome, pop, ordre, les_distances
   b. méthodes : constructeur ae(), optimiser(), selection_roulette(), croisement1X(),…

IV. Analyse de paramétrage

A. Nombre de générations

1. Lancez l'algorithme d'optimisation plusieurs fois (par exemple 5 fois pour chaque valeur des paramètres) avec les paramètres suivants :
   Taille de la population : 20
   Probabilité de croisement : 0.8
   Probabilité de mutation : 0.5
   Taille du chromosome : 10 (distances_entre_villes_10.txt)
   Nombre de générations de : 50, 100, 500, 1000 et 100 000.

L'exécutable algo_evo prend 6 arguments :

```
algo_evo.exe 50 20 0.8 0.5 10 distances_entre_villes_10.txt
```

2. Reportez les résultats et calculer les statistiques (moyenne, écart-type, meilleur résultat) obtenue par chaque groupe de tests. Vous pouvez utilisez OpenOffice ou Excel pour calculer moyenne et écart-type (des fonctions existent déjà).

nb générations	Meilleur résultat					moyenne	écart-type
	exé 1	exé 2	exé 3	exé 4	exé 5		
50							
100							
500							
1000							
100000							

3. Quelles conclusions en tirez-vous ? Qu'es ce que la convergence d'un algorithme ? Converge-t-il ? Quels autres critères d'arrêt proposez-vous ?

B. Taille de la population

1. Exécutez l'algorithme de recherche avec les paramètres suivants
   Taille de la population : 5, 50, 100, 200
   Probabilité de croisement : 0.8
   Probabilité de mutation : 0.5
   Taille du chromosome : 10 (distances_entre_villes_10.txt)
   Nombre de générations : 1000.

2. Complétez le tableau.

taille population	Meilleur résultat					moyenne	écart-type
	exé 1	exé 2	exé 3	exé 4	exé 5		
5							
50							
100							
200							

3. Quelles conclusions vous en tirez ?

4. Une fois ces paramètres bien réglés (nombre de générations et taille de la population) pour résoudre le problème à 10 villes (optimal à 3473km) traiter le problème à 50 villes (distances_entre_villes_50.txt) sur lequel on travaillera jusqu'à la fin du TP. La meilleure solution connue pour ce problème est de 5644km. Quelles conclusions vous en tirez ?

V. Développement de nouveaux opérateurs de mutation et de croisement

Il est nécessaire pour améliorer l'algorithme d'utiliser d'autres opérateurs de mutation et de croisement. L'objectif de cette partie du TP est d'implémenter puis de comparer les performances de plusieurs opérateurs de mutation et de croisement.

Voir sur le site 'Orchids' suivant les différents opérateurs de mutation et de croissement : http://orchids.loria.fr/zmcpgen/page_principale_algo_genetique.htm

Opérateurs de mutation

1. Remarque préliminaire sur le taux de croissement. Si on fixe le taux de mutation à zéro, que se passe-t-il en termes de diversité dans la population ?
   ```
   algo_evo.exe 50000 200 0.8 0.0 50 distances_entre_villes_50.txt
   ```
   et s'il vaut 1 ? Quelles sont vos conclusions ?
2. En plus de l'opérateur de mutation proposé dans le squelette initial de votre programme (échange de 2 gènes consécutifs). Implémentez deux autres opérateurs de mutation de votre choix présentés sur le site 'Orchids'.
3. Exécutez l'algorithme d'optimisation avec les mêmes paramètres que la section précédente afin de comparer les opérateurs de mutation.
4. Quelles sont vos conclusions ?
5. Quels opérateurs de mutation peut-on inventer ?

Opérateurs de croissement

1. Si on met le taux de croisement à zéro, que se passe-t-il ?
2. En plus de l'opérateur de croisement mono-point proposé dans le squelette initial de votre programme (croissement à 1 point). Implémentez un autre opérateur de croisement de votre choix 'Orchids'.
3. Exécutez l'algorithme d'optimisation avec les mêmes paramètres que la section précédente afin de comparer les opérateurs de croisement.
4. Quelles sont vos conclusions ?
5. Quels opérateurs de croisement peut-on inventer ?

VI. Opérateurs de sélection

1. Quel est l'opérateur de sélection implémenté dans le code initial ?
2. Que se passe-t-il si on utilise une sélection et un remplacement aléatoire ? Vous pouvez utiliser les fonctions selection_aleatoire() et remplacement_aleatoire().
3. Utiliser les fonctions selection_ranking() et remplacement_ranking() pour tester la sélection par ranking. Faites varier la valeur du taux de ranking entre les valeurs : 0.0, 1.0, 5.0, 100.0, 1000.0. Quelles sont vos conclusions ?

VII. Questions ouvertes sur des améliorations possibles :

1. Comment faire varier les taux de mutation et de croissement au cours de l'algorithme ?
2. Comment utiliser plusieurs opérateurs de mutation ou/et de croisement simultanément ?
3. Comment travailler avec une population variable ? et quel intérêt ?
4. Comment et pourquoi faire varier la pression de sélection au cours de l'algorithme ?

```
// Procédure principale de la recherche
chromosome* Ae::optimiser()
{
	int amelioration = 0;
	chromosome *fils1 = new chromosome(taille_chromosome);
	chromosome *fils2 = new chromosome(taille_chromosome);
	chromosome *pere1;
	chromosome *pere2;
	int best_fitness;

	// Évaluation des individus de la population initiale
	for(int ind=0; ind<taille_pop; ind++)
		pop->individus[ind]->evaluer(les_distances);

	// On ordonne les individus selon leur fitness
	pop->ordonner();

	best_fitness = pop->individus[pop->ordre[0]]->fitness;
	//  On affiche les statistiques de la population initiale
	cout << "Quelques statistiques sur la population initiale" << endl;
	pop->statiatiques();

	// Tant que le nombre de générations limite n'est pas atteint
	for(int g=0; g<nbgenerations; g++)
	{
		// Sélection de deux individus de la population courante
		pere1 = pop->selection_roulette();
		pere2 = pop->selection_roulette();

		// On effectue un croisement avec une probabilité "taux_croisement"
		if(Random::aleatoire(1000)/1000.0 < taux_croisement)
		{
			croisement1X(pere1, pere2, fils1, fils2);
		}
		else
		{
			fils1->copier(pere1);
			fils2->copier(pere2);
		}

		// On effectue la mutation de l'enfant 1 avec une probabilité "taux_mutation"
		if(Random::aleatoire(1000)/1000.0 < taux_mutation)
			fils1->echange_2_genes_consecutifs();

		// On effectue la mutation de l'enfant 2 avec une probabilité "taux_mutation"
		if(Random::aleatoire(1000)/1000.0 < taux_mutation)
			fils2->echange_2_genes_consecutifs();

		// Évaluation des deux nouveaux individus générés
		fils1->evaluer(les_distances);
		fils2->evaluer(les_distances);

		// Insertion des nouveaux individus dans la population
		pop->remplacement_roulette(fils1);
		pop->remplacement_roulette(fils2);

		// On réordonne la population selon la fitness
		pop->reordonner();

		// Si l'un des nouveaux individus-solutions est le meilleur jamais rencontré
		if (pop->individus[pop->ordre[0]]->fitness < best_fitness)
		{
			best_fitness = pop->individus[pop->ordre[0]]->fitness;
			cout << "Amelioration de la meilleure solution a la generation "
			     << g << " : " << best_fitness << endl;
			amelioration = g;
		}
	}
	//  on affiche les statistiques de la population finale
	cout << "Quelques statistiques sur la population finale" << endl;
	pop->statiatiques();
	//  on affiche la consanginité de la population finale
	pop->similitude();

	//retourner le meilleur individu rencontré pendant la recherche
	return pop->individus[pop->ordre[0]];
}
```