TP 3 – Algorithmes génétiques.

Q. Giorgi.

Recommandations

Lisez très attentivement le sujet

ATTENTION : La présentation, la lisibilité, ainsi que le respect des consignes seront prises en compte dans la notation.

Comme pour chaque TP, vous :

- Réaliserez un makefile pour la compilation des fichiers sources.
- Utiliserez les options de compilation -Wall -g
- Devrez rendre un code source testé et qui se compile correctement !

Travail préparatoire, rappels et compléments de cours.

Lecture préparatoire :

- Lisez (et comprenez !) le document "Algorithmes génétiques.pdf »

- Lisez (et comprenez !) les codes sources fournis.

Travail préparatoire à rendre en début de TP (ce travail sera évalué) :

- Ecrirez en français, l'algorithme DETAILLE utilisé pour réaliser la fonction d'évaluation, notamment le calcul de l'expression arithmétique.

Si vous vous y prenez mal, cette fonction peut vous prendre beaucoup de temps, il est STRICTEMENT nécessaire, d'y avoir réfléchi avant, même d'avoir essayer de la coder.

- Préparez des tests unitaires de la fonction « calcul » avec des serpents de 64-1 gènes.

Le diable veut créer des serpents maléfiques !

Avant propos.

Le diable se prend pour Darwin

Il a remarqué que certains serpents aux portes des enfers possèdent des caractères étranges sur leur corps… En fait ces caractères ressemblent à une expression arithmétique avec des chiffres héxadécimaux… et oui le binaire cest divin, l'héxadécimal, c'est mal….

Il regarde attentivement les serpents, puis au bout d'une minute ou deux, tout d'un coup, il jure sur tous les maitres des enfers……. « Saperlipopette….666 »

Introduction

Un serpent est constitué d'éléments (opérateurs ou opérandes) qui forment une expression arithmétique . De la forme :

operande1 operateur1 *operande2* operateur2 *operande3* operateur3 *operande4*

(vous remarquerez qu'il y a un nombre impair d'éléments)

Un serpent est dit maléfique quand l'évaluation de l'expression retourne 0x666 (en hexadécimal)

Attention à l'évaluation de l'expression !! comme en mathématiques on a une priorité plus importantes des opérateurs * et /

3 + 4 * 5 * 3 - 2 = 3 + ((4*5)*3) -2 = 61

Une erreur serait d'évaluer de la gauche vers la doite chaque opérateur ce qui donnerait ici 103….

Attention à un autre problème, la division par 0…

Si l'expression contient une division par zéro, on ne peut pas effectuer le calcul, le score du serpent est alors fixé par convention à MAX (ici 200000)

Un serpent a donc un score = | resultat – 0x666 | (on appelle cela aussi un fitness), plus le score est pétit, plus le serpent est proche d'être maléfique…

Le diable ne sachant pas comment faire, il va utiliser des algorithmes génétiques pour essayer de produire des serpents maléfiques. En effet, chaque membre de l'expression (opérateur ou opérande) correspond à un gène de notre serpent.

Structure de données utilisée

Chaque gène est codé, dans la structure de données qui nous intéresse, sur 4 bits, ainsi pour :

un gène d'opérande: la valeur du gène correspond à la valeur de l'opérande en héxadécimal.

Un gène d'opérateur: seul les 2 bits de poids faible on un sens, on ignore les 2 autres bits de poids forts :

xx00 → '+'
xx01 → '-'
xx10 → '*'
xx11 → '/' (division entière)

En langage C la plus petite structure de données étant le char (8bits), on regroupera les gènes dans un tableau de char, chaque char contiendra 2 gènes (un opérande et un opérateur). A noter que les ***4 derniers bits ne seront pas utilisés.***

Ainsi on définira les structures de données suivantes :

```
#define NBGENE 4
typedef struct {
        unsigned char gene[NBGENE/2];
        int score; /* sera calculé plus tard */
} serpent;
```

Pour un objet « s » de type serpent si le tableau de gènes vaut "\xA0\x80"

Les 3 gènes utiles pour l'expression sont 0xA, 0x0 et 0x8

alors l'expression arithmétique est 0xA + 0x8, soit en décimal 10+8 (et resultat =18)

Pour un objet « s » de type serpent, si le tableau de gènes vaut "\xD3\x43"

Les 3 gènes utiles pour l'expression sont 0xD, 0x3 et 0x4

alors l'expression arithmétique est 0xD / 0x4, soit en décimal 13/4 (et resultat =3)

Un serpent tout seul, ca ne sert pas à grand-chose, on va donc créer des groupes de serpents :

```
typedef struct {
	serpent *membres;
	int nombre;
} groupe;
```

On donne aussi une fonction permettant d'afficher l'expression arithmétique.

```
#define lire(gene,i)    (i%2)?(gene[i/2]&0xF):(gene[i/2]>>4);
void affiche(unsigned char *gene)
{
	char code[]="+-*/";
	int i=0,res;
	// the last gene is useless
	while (i<(NBGENE-1)) {
		res=lire(gene,i);
		if (i%2)
			printf("%c ",code[res%4]);
		else
			printf("0x%x ",res);
		i=i+1;
	}
	printf("\n");
}
```

Fonctionnement de l'algorithme :

On crée une population, de manière aléatoire, correspondant à la première génération de N serpents
Tant que l'évaluation de la population ne détecte pas un serpent maléfique, faire
	on selectionne les parents (P serpents) parmis cette population de N serpents
	on reproduit les P parents entre eux pour créer une deuxième génération de N serpents fils
	On introduit une mutation des fils
fin tant que

Génération aléatoire :

Cette phase est la plus simple, il suffit de générer N tableaux de NBGENE/2 char de manière aléatoire (utilisation de srand() et rand())

prototype de la fonction :

```
void generationAleatoire(groupe *population) ;
```

Evaluation

Cette phase consiste à évaluer les expressions arithmétiques de chaque serpents et de mettre à jour la valeur de score. La difficulté ici réside dans l'implémentation de la méthode de calcul. La reflexion algorithmique doit avoir lieu avant le TP et fait partie du travail préparatoire.

Prototype de la fonction : (retourne 0 si détecte un serpent maléfique, 1 sinon)
int evaluation(groupe *population) ;

Cette fonction utilisera la fonction de calul de l'expression dont le prototype est :
void calcul(serpent *g) ;

Cette fonction pourra aussi afficher la moyenne des scores de la génération, ainsi que l'écart-type pour voir si il y a une sorte de convergence, ou une amélioration de génération en génération.

Selection

La méthode de selection demandée ici est la plus simple à implémenter, il s'agit de la méthode élitiste, qui consiste à selectionner les P premiers serpents dont le score est le plus petit parmis les N serpents.

Ici on copiera dans parents, les serpents sélectionnés depuis population.

A noter que cette méthode n'est pas la plus éfficace car elle tend à créer des populations très homogènes qui peuvent atteindre un « minimum local », mais le manque de diversité ne permet pas parfois d'atteindre le résultat cherché. Vous pourrez une fois celle-ci implémentée, en implémenter une autre.

Protoype de la fonction :
void selection(groupe *population,groupe *parents) ;

Reproduction

La méthode de reproduction demandée ici consiste à faire un cross-over à 1 point entre les 2 parents ***différents selon la méthde suivante :*** La frontière du cross-over peut être entre des « char » ou entre des « gènes » de 2 parents (mais pas au milieu d'un gène)

Ici on remplira dans population, les serpents fils issus de parents.

Protoype de la fonction :
void reproduction(groupe *population,groupe *parents) ;
void mutation (groupe *population)

Mutation

La phase de mutation consiste à faire muter quelques gènes des fils créer lors de la reproduction, selon un pourcentage de mutation que vous fixerez. Ici, dans le cas de la selection elitiste, la mutation peut être très bénéfique car elle rapporte une diversité génétique qui peut permettre d'atteindre le résultat cherché. La mutation peut se faire sur un gène, mais est plus efficace sur un couple de gène (opérande/opérateur)

Protoype de la fonction :
void mutation (groupe *population) ;

Question : Implémenter l'algorithme décrit ci-dessus en utilisant les fichiers sources fournis, et la compilation séparée comme indiquée par l'enseignant.