Федеральное государственное бюджетное образовательное учреждение высшего образования

"Уфимский государственный авиационный технический университет"

Кафедра Высокопроизводительных вычислительных технологий и систем

Дисциплина: Интеллектуальные системы

Отчет по практической работе № 5

Тема: «Генетический алгоритм»

Группа ПМ-453	Фамилия И.О.	Подпись	Дата	Оценка
Студент	Шамаев И.Р.			
Принял	Казакова Т.Г.			

Уфа 2022

Теоретический материал

Генетический алгоритм — это в первую очередь эволюционный алгоритм, другими словами, основная фишка алгоритма — скрещивание (комбинирование). Так вот, путем перебора и самое главное отбора получается правильная «комбинация».
Алгоритм делится на три этапа:

- Скрещивание
- Селекция (отбор)
- Формирования нового поколения

Если результат нас не устраивает, эти шаги повторяются до тех пор, пока результат нас не начнет удовлетворять или произойдет одно из ниже перечисленных условий:

- Количество поколений (циклов) достигнет заранее выбранного максимума
- Исчерпано время на мутацию

Более подробно о шагах

Создание новой популяции. На этом шаге создается начальная популяция, которая, вполне возможно, окажется не идеальной, однако велика вероятность, что алгоритм эту проблему исправит. Главное, чтобы они соответствовали «формату» и были «приспособлены к размножению».

Размножение. Для получения потомка требуется два родителя. Главное, чтобы потомок (ребенок) мог унаследовать у родителей их черты. При это размножаются все, а не только выжившие (эта фраза особенно абсурдна, но так как у нас все в сферическом вакууме, то можно все), в противном случае выделится один альфа, гены которого перекроют всех остальных, а нам это принципиально не приемлемо.

Мутации. Мутации схожи с размножением, из мутантов выбирают некое количество особей и изменяют их в соответствии с заранее определенными операциями.

Отбор. Тут начинается самое интересное, мы начинаем выбирать из популяции долю тех, кто «пойдет дальше». При этом долю «выживших» после нашего отбора мы определяем заранее руками, указывая в виде параметра. Как ни печально, остальные особи должны погибнуть.

Практика

Рассмотрим на примере Диофантовых уравнений (Уравнения с целочисленными корнями).
Наше уравнение: a+2b+3c+4d=30

Корни данного уравнения лежат на отрезке [1;30], поэтому мы берем 5 случайных значений a,b,c,d. (Ограничение в 30 взято специально для упрощения задачи)
И так, у нас есть первое поколение:

1. (1,28,15,3)
2. (14,9,2,4)
3. (13,5,7,3)
4. (23,8,16,19)
5. (9,13,5,2)

Для того чтобы вычислить коэффициенты выживаемости, подставим каждое решение в выражение. Расстояние от полученного значения до 30 и будет нужным значением.

1. |114-30|=84
2. |54-30|=24
3. |56-30|=26
4. |163-30|=133
5. |58-30|=28

Меньшие значения ближе к 30, соответственно они более желанны. Получается, что большие значения будут иметь меньший коэффициент выживаемости. Для создания системы вычислим вероятность выбора каждой (хромосомы). Но решение заключается в том, чтобы взять сумму обратных значений коэффициентов, и исходя из этого вычислять проценты. (*P.S. 0.135266 — сумма обратных коэффициентов*)

1. (1/84)/0.135266 = 8.80%
2. (1/24)/0.135266 = 30.8%
3. (1/26)/0.135266 = 28.4%
4. (1/133)/0.135266 = 5.56%
5. (1/28)/0.135266 = 26.4%

Далее будем выбирать пять пар родителей, у которых будет ровно по одному ребенку. Давать волю случаю мы будем давать ровно пять раз, каждый раз шанс стать родителем будет одинаковым и будет равен шансу на выживание.
3-1, 5-2, 3-5, 2-5, 5-3
Как было сказано ранее, потомок содержит информацию о генах отца и матери. Это можно обеспечить различными способами, но в данном случае будет использоваться «кроссовер». (| = разделительная линия)

- **Х.-отец:** a1 | b1,c1,d1 **Х.-мать:** a2 | b2,c2,d2 **Х.-потомок:** a1,b2,c2,d2 or a2,b1,c1,d1
- **Х.-отец:** a1,b1 | c1,d1 **Х.-мать:** a2,b2 | c2,d2 **Х.-потомок:** a1,b1,c2,d2 or a2,b2,c1,d1
- **Х.-отец:** a1,b1,c1 | d1 **Х.-мать:** a2,b2,c2 | d2 **Х.-потомок:** a1,b1,c1,d2 or a2,b2,c2,d1

Есть очень много путей передачи информации потомку, а кроссовер — только один из множества. Расположение разделителя может быть абсолютно произвольным, как и то, отец или мать будут слева от черты.
А теперь сделаем тоже самое с потомками:

- **Х.-отец:** (13 | 5,7,3) **Х.-мать:** (1 | 28,15,3) **Х.-потомок:** (13,28,15,3)
- **Х.-отец:** (9,13 | 5,2) **Х.-мать:** (14,9 | 2,4) **Х.-потомок:** (9,13,2,4)
- **Х.-отец:** (13,5,7 | 3) **Х.-мать:** (9,13,5 | 2) **Х.-потомок:** (13,5,7,2)
- **Х.-отец:** (14 | 9,2,4) **Х.-мать:** (9 | 13,5,2) **Х.-потомок:** (14,13,5,2)
- **Х.-отец:** (13,5 | 7, 3) **Х.-мать:** (9,13 | 5, 2) **Х.-потомок:** (13,5,5,2)

Теперь вычислим коэффициенты выживаемости потомков.

- (13,28,15,3) — |126-30|=96(9,13,2,4) — |57-30|=27
  (13,5,7,2) — |57-30|=22
  (14,13,5,2) — |63-30|=33
  (13,5,5,2) — |46-30|=16

Печально так как средняя приспособленность (fitness) потомков оказалась 38.8, а у родителей этот коэффициент равнялся 59.4. Именно в этот момент целесообразнее использовать мутацию, для этого заменим один или более значений на случайное число от 1 до 30.
Алгоритм будет работать до тех, пор, пока коэффициент выживаемости не будет равен нулю. Т.е. будет решением уравнения.
Системы с большей популяцией (например, 50 вместо 5-и) сходятся к желаемому уровню (0) более быстро и стабильно.

Программа

Класс на C++ требует 5 значений при инициализации: 4 коэффициента и результат. Для вышеприведенного примера это будет выглядеть так: **CDiophantine dp(1,2,3,4,30);**

Затем, чтобы решить уравнение, вызовите функцию Solve(), которая возвратит аллель, содержащую решение. Вызовите GetGene(), чтобы получить ген с правильными значениями a, b, c, d.

```
#include <stdlib.h>
#include <time.h>
#include <iostream>
using namespace std;

#define MAXPOP      25

struct gene {
	int alleles[4];
	int fitness;
	float likelihood;

	// Test for equality.
	int operator==(gene gn) {
		for (int i = 0; i < 4; i++) {
			if (gn.alleles[i] != alleles[i]) return false;
		}

		return true;
	}
};

class CDiophantine {
public:
	CDiophantine(int, int, int, int, int);// Constructor with coefficients for
a,b,c,d.
	int Solve();// Solve the equation.

	// Returns a given gene.
	gene GetGene(int i) { return population[i]; }

protected:
	int ca, cb, cc, cd;// The coefficients.
	int result;
	gene population[MAXPOP];// Population.

	int Fitness(gene&);// Fitness function.
	void GenerateLikelihoods();// Generate likelihoods.
	float MultInv();// Creates the multiplicative inverse.
	int CreateFitnesses();
	void CreateNewPopulation();
	int GetIndex(float val);

	gene Breed(int p1, int p2);

};
```

```
CDiophantine::CDiophantine(int a, int b, int c, int d, int res) : ca(a), cb(b), cc(c),
cd(d), result(res) {}

int CDiophantine::Solve() {
	int fitness = -1;

	// Generate initial population.
	srand((unsigned)time(NULL));

	for (int i = 0; i < MAXPOP; i++) {// Fill the population with numbers between
		for (int j = 0; j < 4; j++) {// 0 and the result.
			population[i].alleles[j] = rand() % (result + 1);
		}
	}

	if (fitness = CreateFitnesses()) {
		return fitness;
	}

	int iterations = 0;// Keep record of the iterations.
	while (fitness != 0 || iterations < 50) {// Repeat until solution found, or over
50 iterations.
		GenerateLikelihoods();// Create the likelihoods.
		CreateNewPopulation();
		if (fitness = CreateFitnesses()) {
			return fitness;
		}

		iterations++;
	}

	return -1;
}

int CDiophantine::Fitness(gene& gn) {
	int total = ca * gn.alleles[0] + cb * gn.alleles[1] + cc * gn.alleles[2] + cd *
gn.alleles[3];

	return gn.fitness = abs(total - result);
}

int CDiophantine::CreateFitnesses() {
	float avgfit = 0;
	int fitness = 0;
	for (int i = 0; i < MAXPOP; i++) {
		fitness = Fitness(population[i]);
		avgfit += fitness;
		if (fitness == 0) {
			return i;
		}
	}

	return 0;
}

float CDiophantine::MultInv() {
	float sum = 0;

	for (int i = 0; i < MAXPOP; i++) {
		sum += 1 / ((float)population[i].fitness);
	}

	return sum;
}
```

```
void CDiophantine::GenerateLikelihoods() {
	float multinv = MultInv();

	float last = 0;
	for (int i = 0; i < MAXPOP; i++) {
		population[i].likelihood = last = last + ((1 /
((float)population[i].fitness) / multinv) * 100);
	}
}

int CDiophantine::GetIndex(float val) {
	float last = 0;
	for (int i = 0; i < MAXPOP; i++) {
		if (last <= val && val <= population[i].likelihood) return i;
		else last = population[i].likelihood;
	}

	return 4;
}

gene CDiophantine::Breed(int p1, int p2) {
	int crossover = rand() % 3 + 1;// Create the crossover point (not first).
	int first = rand() % 100;// Which parent comes first?

	gene child = population[p1];// Child is all first parent initially.

	int initial = 0, final = 3;// The crossover boundaries.
	if (first < 50) initial = crossover;	// If first parent first. start from
crossover.
	else final = crossover + 1;// Else end at crossover.

	for (int i = initial; i < final; i++) {// Crossover!
		child.alleles[i] = population[p2].alleles[i];
		if (rand() % 101 < 5) child.alleles[i] = rand() % (result + 1);
	}

	return child;// Return the kid...
}

void CDiophantine::CreateNewPopulation() {
	gene temppop[MAXPOP];

	for (int i = 0; i < MAXPOP; i++) {
		int parent1 = 0, parent2 = 0, iterations = 0;
		while (parent1 == parent2 || population[parent1] == population[parent2]) {
			parent1 = GetIndex((float)(rand() % 101));
			parent2 = GetIndex((float)(rand() % 101));
			if (++iterations > 25) break;
		}

		temppop[i] = Breed(parent1, parent2);// Create a child.
	}

	for (int i = 0; i < MAXPOP; i++) population[i] = temppop[i];
}


void main() {

	CDiophantine dp(1, 2, 3, 4, 30);

	int ans;
	ans = dp.Solve();
	if (ans == -1) {
		cout << "No solution found." << endl;
```

```
	}
	else {
		gene gn = dp.GetGene(ans);

		cout << "The solution set to a+2b+3c+4d=30 is:\n";
		cout << "a = " << gn.alleles[0] << "." << endl;
		cout << "b = " << gn.alleles[1] << "." << endl;
		cout << "c = " << gn.alleles[2] << "." << endl;
		cout << "d = " << gn.alleles[3] << "." << endl;
	}
}
```

Пример работы программы