Homework 2

Genetic Algorithm

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tin Nguyen

ECE 578

11/9/2010

**I. Introduction**

The goal for this homework is be able to do genetic algorithm of labyrinth and demonstrated crossover, mutation, fitness and best solution for the robot on the labyrinth path. Normally genetic algorithm usually begins with randomly-generated genomes. It generates phenomes for each genome, then evaluates each individual’s fitness based on it’s phenome.

**II. Design**

My design is be create the labyrinth pattern and created the path for robot to moves and using crossover, mutation, fitness and best solution to evaluate the path for robot moves. My main focus is write all these code in C+++ and demonstrated my design of labyrinth.

**III. Implementation**

1. **Crossover**

Take 2 genomes, performs one-point crossover on them to produce two new genomes, which returned in a list.



**2. Mutation**

Take a genome, returns the same genome with some of the bits flipped. Flip with a small probability.



1. **Genome**

The list of a population member’s genes. Genes are 1s and 0s.

1. **Phenome**

The expression of an individual’s genome in the “world.” The phenome is a description of the behavior of the individual.

1. **Fitness**

Measure how good a genome ( or phenome) is. Genomes are tend score directly, but often they converted into a phenome and the phenome is given a score. For example, given genome/phenome below, show the the robot’s move and shortest path to exit.

**IV. Sources**

#include<iostream>

#include<fstream>

#include<vector>

#include<string>

#include<cstdlib>

using namespace std;

#define NUM\_GEN 10000

#define POPULATION 2000

#define LENGTH 50

#define CROSSOVER\_PROB 0.3

#define MUTATION\_PROB 0.01

struct position

{

 int x;

 int y;

};

char alphabet[4] = {'F', 'L', 'R', 'S'};

void input\_matrix(vector<string> & matrix, char \* file\_name);

string GA(vector<string> & matrix, int & count);

void init\_population(vector<string> & population);

int fitness(string & plan, vector<string> & matrix);

void reproduce(vector<string> & population, int \* fitness\_value);

void cross\_over(vector<string> & population, int \* fitness\_value);

void mutation(vector<string> & population, int \* fitness\_value);

int simulate(string & plan, vector<string> & matrix);

void perform\_crossover(string & str1, string & str2, vector<string> & population, int \* fitness\_value);

void perform\_mutation(string & temp, vector<string> & population, int \* fitness\_value);

void get\_action(char action, position & pos, char & robot);

void print\_pop(vector<string> population);

int main(int argc, char \*\* argv)

{

 ofstream fout;

 char c;

 string result;

 int count = 0;

 if(argc != 2)

 {

 cout << "Command should have the form : genetic -file\_name \n";

 cout << "/t -file\_name : Name of the file containing the input matrix.\n";

 return 1;

 }

 vector<string> matrix;

 input\_matrix(matrix, argv[1]);

 //result = "RFLLRLLLS";

 //cout << "Test : " << fitness(result, matrix) << "\n";

 //cin >> c;

 // cout << matrix[9] << "\n";

 // cin >> c;

 result = GA(matrix, count);

 // cout << result << "\n";

 // cin >> c;

 if(count >= NUM\_GEN)

 {

 cout << "The algorithm cannot find the solution for this instance\n";

 cin >> c;

 return 0;

 }

 cout << "The algorithm found a solution after " << count << " generations.\n";

 cout << "That solution is : " << result << "S\n";

 fout.open("solution.txt");

 fout << "The algorithm found a solution after " << count << " generations.\n";

 fout << "That solution is : " << result << "S\n";

 fout.close();

 cin >> c;

 return 0;

}

void input\_matrix(vector<string> & matrix, char \* file\_name)

{

 ifstream fin;

 fin.open(file\_name);

 matrix.resize(0);

 int row;

 fin >> row;

 matrix.resize(row);

 for(int i = 0; i < row; i++)

 {

 fin >> matrix[i];

 }

 fin.close();

}

string GA(vector<string> & matrix, int & count)

{

 char c;

 string temp = "";

 vector<string> population;

 int fitness\_value[POPULATION];

 init\_population(population); ////////

 count = 0;

 while(count <= NUM\_GEN)

 {

 //print\_pop(population);

 //cin >> c;

 for(int i = 0; i < POPULATION; i++)

 {

 fitness\_value[i] = fitness(population[i], matrix); /////////

 if(fitness\_value[i] == -1)

 {

 return population[i];

 }

 }

 reproduce(population, fitness\_value); /////////////

 cross\_over(population, fitness\_value); //////////////////

 mutation(population, fitness\_value); //////////////////

 count++;

 }

 print\_pop(population);

 return temp;

}

void init\_population(vector<string> & population)

{

 char temp;

 population.resize(POPULATION);

 for(int i = 0; i < POPULATION; i++)

 {

 population[i].resize(0);

 for(int j = 0; j < LENGTH; j++)

 {

 temp = alphabet[rand() % 4];

 population[i].push\_back(temp);

 }

 }

}

int fitness(string & plan, vector<string> & matrix)

{

 int result;

 result = simulate(plan, matrix); /////////////

 if(result == -1)

 {

 return -1;

 }

 return result;

}

void reproduce(vector<string> & population, int \* fitness\_value)

{

 vector<string> temp;

 temp.resize(0);

 float percentage[POPULATION];

 int i;

 int sum = 0;

 float prob;

 temp = population;

 for(i = 0; i < POPULATION; i++)

 {

 sum += fitness\_value[i];

 }

 percentage[0] = (float)fitness\_value[0]/sum;

 //cout << percentage[0] << "AAA\n";

 for(i = 1; i < POPULATION; i++)

 {

 percentage[i] = (float)fitness\_value[i]/sum + percentage[i - 1];

 //cout << percentage[i] << "AAA\n";

 }

 for(i = 0; i < POPULATION; i++)

 {

 prob = (float)(rand())/RAND\_MAX;

 //cout << prob << "AAA\n";

 for(int j = 0; j < POPULATION; j++)

 {

 if(prob < percentage[j])

 {

 population.push\_back(temp[j]);

 break;

 //cout << population[j] << "AAA\n";

 }

 }

 }

 /\* population.resize(0);

 for(i = 0; i < temp.size(); i++)

 population.push\_back(temp[i]);\*/

}

void cross\_over(vector<string> & population, int \* fitness\_value)

{

 float prob;

 int temp\_1;

 int temp\_2;

 int count;

 for(int i = 0 ; i < POPULATION; i++)

 {

 prob = (float)(rand())/RAND\_MAX;

 if(prob < CROSSOVER\_PROB)

 {

 // temp\_1 = rand() % POPULATION;

 temp\_1 = i;

 count = 0;

 do

 {

 temp\_2 = rand() % POPULATION;

 }

 while((population[temp\_2] == population[temp\_1]) && (count++ <= 2\*POPULATION));

 perform\_crossover(population[temp\_1], population[temp\_2], population, fitness\_value); ///////////////

 }

 }

}

void mutation(vector<string> & population, int \* fitness\_value)

{

 int temp;

 float prob;

 for(int i = 0 ; i < POPULATION; i++)

 {

 prob = (float)(rand())/RAND\_MAX;

 if(prob < MUTATION\_PROB)

 {

 temp = rand() % POPULATION;

 perform\_mutation(population[temp], population, fitness\_value); //////////////////

 }

 }

}

int simulate(string & plan, vector<string> & matrix)

{

 int count;

 position pos;

 pos.x = 0;

 pos.y = 0;

 char robot = 'n';

 int i = 0;

 int j;

 int temp[20][20];

 for(i = 0; i < matrix.size(); i++)

 {

 for(j = 0; j < matrix.size(); j++)

 {

 temp[i][j] = 0;

 }

 }

 temp[0][0] = 1;

 count = 0;

 i = 0;

 while(i < plan.length())

 {

 get\_action(plan[i], pos, robot); ///////////////////

 if(plan[i] != 'S')

 {

 count++;

 }

 if((pos.x < 0) || (pos.x >= matrix.size()) || (pos.y < 0) || (pos.y >= matrix.size()) || matrix[pos.x][pos.y] == 'X')

 {

 return count;

 }

 if(matrix[pos.x][pos.y] == 'O')

 {

 for(int k = i + 1; k < plan.length(); k++)

 {

 plan[k] = 'S';

 }

 return -1;

 }

 if(temp[pos.x][pos.y] == 0)

 {

 temp[pos.x][pos.y] = count;

 }

 else

 {

 count = temp[pos.x][pos.y];

 }

 i++;

 }

 return (count + 5);

}

void perform\_crossover(string & str1, string & str2, vector<string> & population, int \* fitness\_value)

{

 char c;

 string temp;

 temp.resize(0);

 string temp\_1;

 string temp\_2;

 temp\_1.resize(0);

 temp\_1 = str1;

 temp\_2.resize(0);

 temp\_2 = str2;

 int pos;

 int index = 0;

 int val = fitness\_value[0];

 pos = rand() % LENGTH;

 if(pos < 0 || pos >= LENGTH)

 {

 cout << "AAA\n";

 cin >> c;

 }

 temp = temp\_1;

 for(int i = pos; i < LENGTH; i++)

 {

 temp\_1[i] = temp\_2[i];

 temp\_2[i] = temp[i];

 }

 for(int j = 0; j < POPULATION; j++)

 {

 if(fitness\_value[j] < val)

 {

 index = j;

 val = fitness\_value[j];

 }

 }

 population[index].resize(0);

 for(int x = 0; x < temp\_1.length(); x++)

 population[index].push\_back(temp\_1[x]);

 fitness\_value[index] = LENGTH;

 index = 0;

 val = fitness\_value[0];

 for(int k = 0; k < POPULATION; k++)

 {

 if(fitness\_value[k] < val)

 {

 index = k;

 val = fitness\_value[k];

 }

 }

 population[index].resize(0);

 for(int y = 0; y < temp\_2.length(); y++)

 population[index].push\_back(temp\_2[y]);

 fitness\_value[index] = LENGTH;

}

void perform\_mutation(string & temp, vector<string> & population, int \* fitness\_value)

{

 int index = 0;

 int val = fitness\_value[0];

 int pos;

 char c;

 string str;

 str.resize(0);

 str = temp;

 pos = rand() % LENGTH;

 c = alphabet[rand() % 4];

 while(c == str[pos])

 {

 c = alphabet[rand() % 4];

 }

 str[pos] = c;

 for(int i = 0; i < POPULATION; i++)

 {

 if(val > fitness\_value[i])

 {

 index = i;

 val = fitness\_value[i];

 }

 }

 population[index].resize(0);

 for(int x = 0; x < str.length(); x++)

 population[index].push\_back(str[x]);

}

void get\_action(char action, position & pos, char & robot)

{

 if(action == 'S')

 {

 return;

 }

 if(robot == 'n')

 {

 if(action == 'F')

 {

 pos.x--;

 }

 else if(action == 'L')

 {

 pos.y--;

 robot = 'w';

 }

 else if(action == 'R')

 {

 pos.y++;

 robot = 'e';

 }

 return;

 }

 if(robot == 's')

 {

 if(action == 'F')

 {

 pos.x++;

 }

 else if(action == 'L')

 {

 pos.y++;

 robot = 'e';

 }

 else if(action == 'R')

 {

 pos.y--;

 robot = 'w';

 }

 return;

 }

 if(robot == 'e')

 {

 if(action == 'F')

 {

 pos.y++;

 }

 else if(action == 'L')

 {

 pos.x--;

 robot = 'n';

 }

 else if(action == 'R')

 {

 pos.x++;

 robot = 's';

 }

 return;

 }

 if(robot == 'w')

 {

 if(action == 'F')

 {

 pos.y--;

 }

 else if(action == 'L')

 {

 pos.x++;

 robot = 's';

 }

 else if(action == 'R')

 {

 pos.x--;

 robot = 'n';

 }

 return;

 }

}

void print\_pop(vector<string> population)

{

 for(int i = 0; i < POPULATION; i++)

 {

 cout << population[i] << "\n";

 }

}

1. **Test Results**

The algorithm found a solution after 31 generations.

That solution is : SRRSLRFSFFLFFRFSFSRFSFLRSFRFRSSSSSSSSSSSSSSSSSSSSSS

10 x10 matrix dimension

. . X X X X X X X X

X . . X X X X X X X

X X . . . X X X X X

X X . X X X X X X X

X X . X X X X X X X

X X . . . . X X X X

X X . . X . X X X X

. O X X X . X X X X

. X . . . . . . X X

. . . X . X X X X X

**Symbol of syntax**

O is stand for target

X is stand for wall

. is the path that robot go

S is stand for stop

R is stand for right

F is forward

1. **In conclusion**

This homework is excited because I learn how created pattern that robotic go and follow the path in the labyrinth. I have to assign direction when the robot have to turn or when robot stop then hit for target.