Alan Cheng

ECE-478

10-10-10

ID: 4691

Genetic Algorithm Maze Solving Program

 In this program, an answer to a labyrinth is solved. The program first declares classes in:

class chromozone

{

public:

 int chromo[64];

 int fitness;

};

typedef vector<chromozone> vChrom;

class GA

{

public:

 int mazeSeq[8][8];

 int mazeCopy[8][8];

 int mazeLoc[8][8];

 int populationSize;

 int maxLoop;

 int usedSpace;

 float mutationRate;

 float mutation;

 int checkDeath(int X, int Y, int &nxtX, int &nxtY);

 void init\_population(vChrom &population, vChrom &tempStore);

 void calc\_fitness(vChrom &chromozone);

 void mutate(chromozone &member);

 void mate(vChrom &population, vChrom &tempStore);

};

**Initializing the Population:**

Then it initializes the population, along with the maze layout. The maze is a 2D array which is filled with 9, 8, 7, or 0. A 9 represent a wall. A 7 represent the starting point. An 8 represent the finishing point. A 0 represent a moveable walkway. The maze layout is as following:

9 9 9 9 9 9 9 9

9 0 0 0 0 0 9 9

9 0 9 9 9 0 9 9

9 0 0 0 9 0 9 9

9 0 9 9 9 0 0 9

9 0 9 9 9 9 0 9

9 0 9 0 0 0 0 9

9 8 9 7 9 9 9 9

The code in the program is as following:

 for(i=0; i<8; i++)

 {

 mazeSeq[i][0]=9;

 }

 for(j=0; j<8; j++)

 {

 mazeSeq[0][j]=9;

 }

 for(j=0; j<8; j++)

 {

 mazeSeq[7][j]=9;

 }

 for(i=0; i<8; i++)

 {

 mazeSeq[i][7]=9;

 }

 mazeSeq[6][1] = 9;

 mazeSeq[6][2] = 9;

 mazeSeq[6][3] = 9;

 mazeSeq[3][7] = 7;

 mazeSeq[1][7] = 8;

 mazeSeq[2][2] = 9;

 mazeSeq[3][2] = 9;

 mazeSeq[4][2] = 9;

 mazeSeq[4][3] = 9;

 mazeSeq[2][4] = 9;

 mazeSeq[3][4] = 9;

 mazeSeq[4][4] = 9;

 mazeSeq[2][5] = 9;

 mazeSeq[3][5] = 9;

 mazeSeq[4][5] = 9;

 mazeSeq[5][5] = 9;

 mazeSeq[2][6] = 9;

The for loops are for setting the boundary to 9.

Two other 8 by 8 arrays are created. One for keeping track of visited squares in case of endless loops and one for telling which sequence to put the chromosome sequence in.

Afterwards, we random give sequences to a population of 1000. Before we fill on the random sequences, we find the amount of spaces in the maze and shorten each chromosome by this amount. This is done by:

 for(i=0; i<8; i++)

 {

 for(j=0; j<8; j++)

 {

 if(mazeSeq[i][j] == 0)

 usedSpace++;

 }

 }

 chromoS.fitness = 0;

 population.clear();

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

 {

 for (j=0; j<usedSpace; j++)

 {

 chromoS.chromo[j] = lowest+int(range\*rand()/(RAND\_MAX + 1.0));

 fprintf(fptr2, "%d", chromoS.chromo[j]);

 }

 fprintf(fptr2, "\n \n" );

 population.push\_back(chromoS);

 }

 tempStore.resize(populationSize);

\*The fprintf is for debugging reasons to writing to a file.
\*\*The population was defined earlier and same with the lowest/highest values to be random generator. The numbers were 1 for lowest and 4 for highest.

**Calculate Fitness:**

The next step is to calculate the fitness. We first find the starting and ending points.

 for(i=0; i<8; i++)

 {

 for(j=0; j<8; j++)

 {

 if(mazeSeq[i][j] == 7)

 {

 startX = i;

 startY = j;

 }

 }

 }

 for(i=0; i<8; i++)

 {

 for(j=0; j<8; j++)

 {

 if(mazeSeq[i][j] == 8)

 {

 endX = i;

 endY = j;

 }

 }

 }

We then reset the array which holds where the robot is going. A 1 represent a place that the robot has gone to. The maze is then filled with the sequence of the corresponding chromosome. Afterwards we check if the robot dies by this sequence and calculate the fitness using the amount of spaces remaining.

 for(i=0; i<8; i++)

 {

 for(j=0; j<8; j++)

 {

 if(mazeSeq[i][j] == 8)

 {

 endX = i;

 endY = j;

 }

 }

 }

 for(l=0; l<chromozone.size(); l++)

 {

 distT = 0;

 curX = startX;

 curY = startY-1;

 for(i=0; i<8; i++)

 {

 for(j=0; j<8; j++)

 {

 mazeCopy[i][j]=0;

 }

 }

 for(int m=0; m<usedSpace-1;m++)

 {

 for(int x=0; x<8; x++)

 {

 for(int y=0; y<8; y++)

 {

 if(mazeLoc[x][y] == m)

 mazeSeq[x][y] = chromozone[l].chromo[m];

 }

 }

 }

 do

 {

 if(checkDeath(curX, curY, nxtX, nxtY)==1)

 {

 break;

 }

 else

 {

 mazeCopy[curX][curY] = 1;

 curX = nxtX;

 curY = nxtY;

 if(mazeCopy[nxtX][nxtY] == 1)

 {

 break;

 }

 distT++;

 }

 }

 while((curY != endY));

 chromozone[l].fitness = usedSpace - distT;

 }

}

In more detail, there is a do loop which repeats along if the robot is alive. The function checkDeath checks if the robot dies and returns a 1 if it is dead. The amount of remaining spaces is the Manhattan distance, and is calculated by the total number of moveable spaces subtracted by the number of spaces the robot travelled.

The checkDeath function is as followed:

 case 1: // If the robot goes forward

 if(mazeSeq[X][Y-1] == 9)

 {

 rcode = 1;

 break;

 }

 nxtX = X;

 nxtY = Y-1;

 break;

 case 2:

 if(mazeSeq[X-1][Y] == 9) // If the robot goes left

 {

 rcode = 1;

 break;

 }

 nxtX = X-1;

 nxtY = Y;

 break;

 case 3:

 if(mazeSeq[X+1][Y] == 9) // If the robot goes right

 {

 rcode = 1;

 break;

 }

 nxtX = X+1;

 nxtY = Y;

 break;

 case 4: // If the robot goes backwards

 if(Y+1 > 7) // If the robot goes out of bounds

 {

 rcode = 1;

 break;

 }

 if((mazeSeq[X][Y+1] == 9) && (mazeSeq[X][Y+1] == 8))

 {

 rcode = 1;

 break;

 }

 nxtX = X;

 nxtY = Y+1;

 break;

 }

 return rcode;

}

This functions gets the next space in the maze and see if the robot will hit a wall using this command. If so, the function returns a 1.

**Sorting the Fitness:**

The next step goes through the population and puts the chromosones with the best fitness at the top and the worst fitness at the bottom. This way, we only need to take the top of the stack for the crossover and mutation operators. We can also display the chromosone with the best fitness by calling the top of this stack.

bool sort\_fitness(chromozone x, chromozone y);

bool sort\_fitness(chromozone x, chromozone y)

{

 return(x.fitness < y.fitness);

}

inline void sort\_by\_fitness(vChrom &population);

inline void sort\_by\_fitness(vChrom &population)

{

 sort(population.begin(), population.end(), sort\_fitness);

}

The best chromosone is displayed by:

inline void print\_best(vChrom &dir);

inline void print\_best(vChrom &dir)

{

 cout << "Direction: " << dir[0].chromo[0] << dir[0].chromo[1] <<

 dir[0].chromo[2] << dir[0].chromo[3] << dir[0].chromo[4] <<

 dir[0].chromo[5] << dir[0].chromo[6] << dir[0].chromo[7] <<

 dir[0].chromo[8] << dir[0].chromo[9] << dir[0].chromo[10] <<

 dir[0].chromo[11] << dir[0].chromo[12] << dir[0].chromo[13] <<

 dir[0].chromo[14] << dir[0].chromo[15] << dir[0].chromo[16] <<

 dir[0].chromo[17] << dir[0].chromo[18] << dir[0].chromo[19] <<

 dir[0].chromo[20] << " (" << dir[0].fitness << ") " << endl;

}

**Reproducing and crossover/mutation operators:**

In this function, the top half of all the chromosomes and randomly select two chromosomes. With these chromosomes, the program randomly calculates a crossover point and a mutation point. For mutation, a random number (1-4) is calculated and the point which the mutation pointer points to is swapped with the random number.

void GA::mate(vChrom &population, vChrom &tempStore)

{

 int spos, j, k;

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

 {

 j = rand() % (populationSize / 3); // 3 is optimal

 k = rand() % (populationSize / 3);

 spos = rand() % usedSpace;

 for(int l=0; l<spos; l++)

 {

 tempStore[i].chromo[l] = population[j].chromo[l];

 }

 for(int l=spos; l<usedSpace; l++)

 {

 tempStore[i].chromo[l] = population[k].chromo[l];

 }

 if (rand() < mutation)

 mutate(tempStore[i]);

 }

}

void GA::mutate(chromozone &member)

{

 int lowest=1, highest=4;

 int range=(highest-lowest)+1;

 int mutPos = rand() % usedSpace;

 int delta = lowest+int(range\*rand()/(RAND\_MAX + 1.0));

 member.chromo[mutPos] = delta;

}

**Swapping:**

At last, we take the newly produced chromosones and replace the old ones with them. This is done by:

inline void swap(vChrom \*&population, vChrom \*&tempStore);

inline void swap(vChrom \*&population, vChrom \*&tempStore)

{

 vChrom \*temp = population;

 population = tempStore;

 tempStore = temp;

}

**The Main Function:**

The main function first initializes the population. Then the program loops, going through the functions which calculate the fitness, sort the fitness, generate new chromosones, and swap the old with the new chromosones, until the fitness is two, since two of the spaces is unused in the maze.

int main()

{

 GA ga;

 int i=0;

 vChrom chromoPop, chromoStore, \*population, \*storeData;

 srand(unsigned(time(NULL)));

 ga.init\_population(chromoPop, chromoStore);

 population = &chromoPop;

 storeData = &chromoStore;

 for(int i=0; i<15000; i++)

 {

 ga.calc\_fitness(\*population);

 sort\_by\_fitness(\*population);

 print\_best(\*population);

 if ((\*population)[0].fitness == 2)

 break;

 ga.mate(\*population, \*storeData);

 swap(population, storeData);

 }

 \_getch();

 return 0;

}

**Analysis:**

Overall, the genetic algorithm performed very good, very little have fitness functions higher than the previous fitness. The problem was that sometimes there would be lots of generations before it found the solution, most of the time at the same fitness and not improving.

So therefore, in order to find the best value to divide the population (how many are canidates to pick for selection), the values 2, 3, and 4 where tried. Any number after that would have a limited selection, therefore leading to a greater number of generations.

 j = rand() % (populationSize / n); // n = to the values 2, 3, 4

 k = rand() % (populationSize / n);

In the results for test 1, the values of 3 and 4 did the best with a generation maxium of 21 while the value of 2 lacked behind with a generation maxium of 35.

In test 2, a value of 3 significantly did better with a generation maxium of 25. This was almost two times less than a value of 2 which got 40. A value of 4 was in the middle with a value of 34

In test 3, a value of 3 again did the best with a generation of 22 while a value of 2 severly lacked behind with a genration of 37. A value of 4 was closer to a value of 2 with a generation of 26.

In conclusion, the algorithm worked 100% of the time using any population to be selected. However a range of 500 was very ineffecient because it gets to the goal in about 35 tries. A range of 250 was better with an average of about 27 tries. A range of about 333 was optimal with an average of 23. This is a significant difference from a range of 500 and results in faster calculation.