RECORD ABOUT GA

TING LI e-mail: litingszy@qq.com

21 May, 2019

1 Genetic Programming

1.1 About the problem

Traveling Salesman Problem is a famous problem. It supposes a traveling businessman wants to visit n cities, he must choose the path to go, the path limit is each city can only visit once, and finally to return to the original departure city. The goal of path selection is to obtain the minimum path distance of all paths.

However, in this assignment, there is another condition should be considered: cost. Suppose different type of cities have different cost(cents/km) of travelling between each other. Our target is to figure out the best path which has the minimum cost.

1.2 Implementation overview

In order to implement this algorithm, those functions given in ga.cpp are altered and several necessary functions are added, such as functions for reading data, for selecting and for calculating the cost. Using these functions can we implement this algorithm to solve TSP.

Before introduce these functions, another change should be pointed out that two structures are defined: CityINFO and TSPPara. The first one is used to store city's information such as city type and coordinates. And TSPPara is consist of several parameters(or features) which are used during figuring out the best route. Therefore, some representations in given functions will change, but the overall functionality is similar.

The Implementation part contains 4 steps.

1.3 Step 1: Functions altered

First of all, we should read the data in the file. So **GetData()** function is added to read data. It can read the city number and store it, as well as store the city's type and coordinates in the CityInfo structure. These data will be used for future calculations.

There are 4 important functions given in the file: InitPop(), EvaluateFitness(), Crossover() and Mutate(). We need to alter them to varying degrees to achieve our goals.

Firstly, in this assignment, **InitPop(TSPPara &City)** is defined to load the initial population members with random cities' serial numbers. Here City is defined as TSPPara type. There is no functional change in this function, just a change in the way it is represented (using the structure).

Secondly, **EvaluateFitness(TSPPara &City)** is defined to calculate fitness. In this function, City(a structure) is called, and the detailed code is shown below.

```
void EvaluateFitness(TSPPara &City) {
  //Evaluates fitness
  int i, j, s, e, best = 0;
  for (i = 0; i < PopSize; i++) {
    City.Cost[i] = 0;
```

```
    for (j = 1; j <= cIndividualLength - 1; j++) {
      s = City.pop[i][j - 1]; e = City.pop[i][j];
      City.Cost[i] = City.Cost[i] + CityCost[s][e];  //total cost
    }
    City.Fitness[i] = FIX / City.Cost[i];
    if (City.Fitness[i] > City.Fitness[best]) {  //choose the biggest fitness value
      best = i;
    }
  }
  Copy(City.BestRoute, City.pop[best]); //copy the best one to City.BestRoute
  City.BestFitness = City.Fitness[best];
  City.BestValue = City.Cost[best]; City.BestNum = best;
}
```

As shown above, in the cycle, the total route and fitness value of each individual are calculated, and the maximum fitness value is selected in the second external cycle, that is to say, the optimal path of all individuals is selected. The City.Cost array is called in the comparison, which is a table containing travel costs between all individuals for querying and calculating the total cost. After obtaining the best individual corresponding to the maximum fitness value. This individual is copied to City.BestRout. At the same time, City.BestFitness, City.BestValue and City.BestNum are all updated.

Thirdly, crossover is a significant part in the GA algorithm as it ensure the stability of the population and evolve towards the optimal solution. It takes chromosomes from both parents and crosses them to produce offspring. **Crossover(TSPPara &City, double CR)** is defined to implement the crossover operation. The main part of the code is as follows:

```
for (i = 0; i < PopSize; i++) {
    double s = ((double)(rand() % RAND_MAX)) / RAND_MAX;
    if (s < CR) {
      cn = rand() % PopSize; cm = cn;
      if (cm == City.BestNum || cn == City.BestNum) { continue; }//If the optimal is
          encountered, the next cycle is directly carried out
      l = rand() % (cIndividualLength / 2) + 1;  //1~first part
      m = rand() % (cIndividualLength - l) + 1; //1~cIndividualLength
      memset(SHUF, 0, sizeof(SHUF));//replace SHUF's current sizeof(SHUF) bits with 0.
      Temp1[0] = Temp1[cIndividualLength] = 0;
      for (j = 1; j <= l; j++) {//Shuffling order(is random), and the selected one is marked
           as 1.
        Temp1[j] = City.pop[cn][m + j - 1]; SHUF[Temp1[j]] = 1;
      }
      for (k = 1; k < cIndividualLength; k++) {
        if (SHUF[City.pop[cm][k]] == 0) {
          Temp1[j++] = City.pop[cm][k]; SHUF[City.pop[cm][k]] = 1;
        }}
      memcpy(City.pop[cm], Temp1, sizeof(Temp1));
    }}
```

The purpose of this loop is to complete the crossover operation. In an external loop, a double type number is randomly selected first, and if it is less than CR, the crossover is started. The main process is to select a random number of cities and sort them randomly. Then use an array named SHUF to record the selected city's order number (the element corresponding to the same position in SHUF is set to 1). Then a cycle and a judgment condition are used to reorder the unselected city numbers. This steps avoid duplicate appearance of cities. In this assignment, the crossover result is: the first and the final one do not change, and those individuals between them will be disrupted and crossed.

Fourthly, mutation is also important as it can ensure the diversity of the population and avoid the

possible local convergence caused by crossover. **Mutate(TSPPara &City, double MR)** is defined to implement it. Because the code of this function is too long, here I will introduce its mian processes in words. In this function, firstly we define a random double type number and a random number, then respectively compare them with mutation rate and City.BestNum(the serial number of the best individual), to ensure the optimal individual does not mutate. Then randomly choose two points to mutate, the operation is to exchange the values of the two, and then calculate the fitness of the individual after the mutation to see if it is less than the fitness before the mutation. If so, the mutation is carried out. This process is repeated twice (the experiment result proves that the result is great after repeating process two times).

1.4 Step 2: Lookup table of traveling cost

In order to reduce the calculating cost, here defined a CalculateCost() function to calculate costs of traveling between cities and store them in a NN table. In this function, the distance between cities are calculated firstly, and then calculate traveling cost according to different types of cities. Finally, store these costs in CityCost which can be seen as lookup table of traveling cost.

The detailed code is shown as follows:

```
double CityDist[cIndividualLength][cIndividualLength];
double CityCost[cIndividualLength][cIndividualLength];//This is the lookup table
void CalculateCost(){
  int i, j;
  double temp1, temp2;
  for (i = 0; i < cIndividualLength; i++) {
    for (j = 0; j <= cIndividualLength; j++) {//The last city should be able to return to
        the departure node.
      temp1 = CityInfo[j].x - CityInfo[i].x; temp2 = CityInfo[j].y - CityInfo[i].y;
      CityDist[i][j] = sqrt(temp1 * temp1 + temp2 * temp2);
      int flag = CityInfo[j].t * CityInfo[i].t;
      switch (flag) {
      case 1: CityCost[i][j] = CityDist[i][j] * 10.0; break;
      case 2: CityCost[i][j] = CityDist[i][j] * 7.5; break;
      case 3: CityCost[i][j] = CityDist[i][j] * 5.0; break;
      case 4: CityCost[i][j] = CityDist[i][j] * 5.0; break;
      case 6: CityCost[i][j] = CityDist[i][j] * 2.5; break;
      case 9: CityCost[i][j] = CityDist[i][j] * 1.0; break;
      }}}}
```

1.5 Setp 3: Roulette wheel selection

In this assignment, a Roulette wheel selection is implemented. The Roulette is also known as the proportional selection method in which the probability of each individual being selected is proportional to its fitness. So in this function, the process can be divided into 4 steps:

1. Calculate the sum of fitness of all individuals. (The fitness of each individual has been previously calculated and stored in the structure.)

2. Calculate the probability that each individual is inherited into the next generation.

3. Then the cumulative probability of each individual is calculated. (Stored in SelectP.)

4. Generate a random number between 0 and 1, and compare it with the cumulative probability of each individual. If the cumulative probability is greater than this random number, choose it, otherwise choose individual k. Until SelectP [k-1] $<$ Rand. $<$ SelectP [k]. This step cycles PopSize times. (Population size.)

The detailed code of Roulette wheel selection is as follows:

```
void RWselection(TSPPara &City) {//Roulette wheel selection
```

```
int i, j, k;
int tpop[PopSize][cIndividualLength + 1];
double s, sum = 0;
double Assi[PopSize], SelectP[PopSize + 1];
for (i = 0; i < PopSize; i++) {
  sum += City.Fitness[i];}
for (i = 0; i < PopSize; i++) {
  Assi[i] = City.Fitness[i] / sum;}
SelectP[0] = 0;
for (i = 0; i < PopSize; i++) {
  SelectP[i + 1] = SelectP[i] + Assi[i] * RAND_MAX;}
memcpy(tpop[0], City.pop[City.BestNum], sizeof(tpop[0]));//To copy data from City.BestNum
    to tpop[0].
for (k = 1; k < PopSize; k++) {
  double ran = rand() % RAND_MAX + 1; s = (double)ran / 100.0;
  for (i = 1; i < PopSize; i++) {
    if (SelectP[i] >= s) { break; }}
  memcpy(tpop[k], City.pop[i - 1], sizeof(tpop[k]));}
for (i = 0; i < PopSize; i++) {
  memcpy(City.pop[i], tpop[i], sizeof(tpop[i]));}}
```

1.6 Step 4: Experiment with different parameters

After implementation of algorithm, then comes to experiment. In this assignment, three main parameters are tested and changed in different experiment. They are population size(PopSize), crossover rate(cCrossoverRate) and mutation rate(cMutationRate).

Firstly, population size is changed and the one which has the best performance is figured out. Then change the crossover rate and mutation rate respectively to find out the best parameters.

I test different size of population, and here pined out the main part of the result in which can we see the best performance and its corresponding population size. When the individual length is 100, its best population size can be 300, and when it is 200, the best population size is 500. However for 500 individuals, the best population size is 250, it could not perform when the population size is too large (over 250 or more). The experiment results are shown as follows:

Individual Length	100			200			500		
Population Size	100	300	500	100	300	500	100	200	250
Minimum Cost	79186	55243.1	61241.2	166700	158624	151902	397061	389418	341861
Correspond Epoch	815	3700	2225	4210	2480	2500	4450	1980	9750

Table 1: Minimum cost in different population size

Then choose the best population size for different individual length, and continue experiment with different crossover rate and mutation rate. Here I choose three different crossover rate and three mutation rate respectively, and the experiment results is shown bellow (The epoch numbers are not shown, as they are too many which may make the table difficult to read):

Individual Length		100			200			500		
Population Size		300			500			250		
Crossover rate		0.7	0.75	0.8	0.7	0.75	0.8	0.7	0.75	0.8
	0.001	55761	55243.1	63074.2	138067	138543	166112	333260	341861	385254
Mutation rate	0.01	58611.8	63769.6	58754	155197	151902	164759	326998	365898	358712
	0.05	58529.7	69875.5	69095.6	139551	151941	155014	344010	367299	355797

Table 2: Minimum cost in different crossover rate and mutation rate

As shown in the table above, we can conclude the best parameters for different individual length:

Individual Length	Population size	Crossover rate	Mutation rate	Min cost	Correspond epoch
100	300	0.75	0.001	55243.1	3700
200	500	0.7	0.001	138067	5155
500	250	0.7	0.01	326998	5710

Table 3: Best parameters

1.7 Step 5: The Best Experiment Result

Those best parameters for different individual length are as shown in the last chapter, here attach the experiment results(graphs). It should be point out that for 100 individual length, the best travelling cost is appears in the 3700th epoch, while 300 individuals is in the 5155th epoch and 500 individuals is in the 5710th epoch. By the way, the whole code is attached in the end of the report.

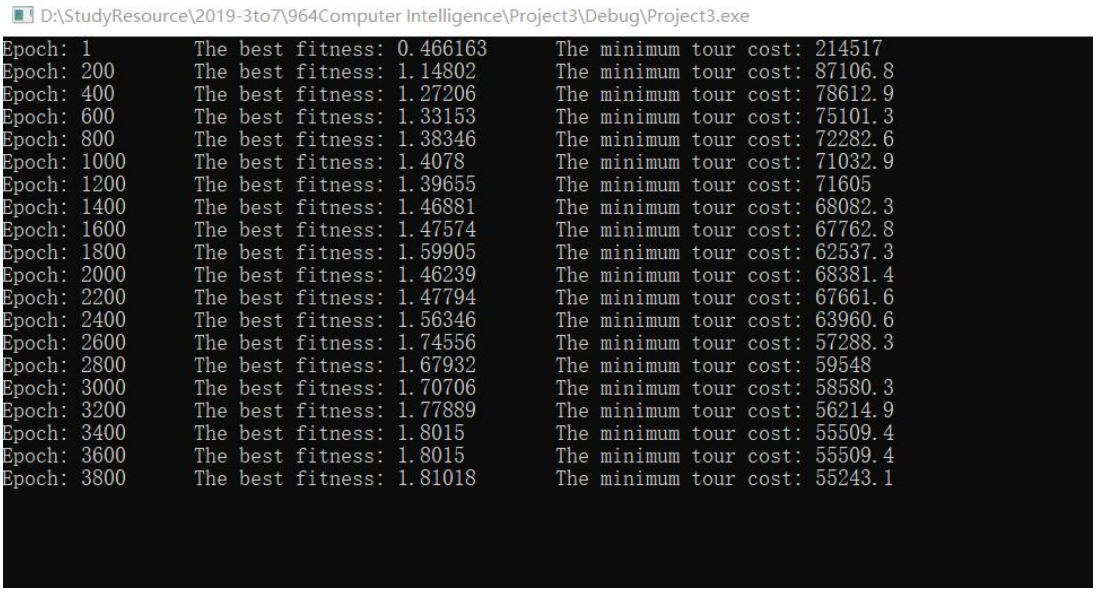

Figure 1: Experiment result of 100 individual length(print every 200th generation)

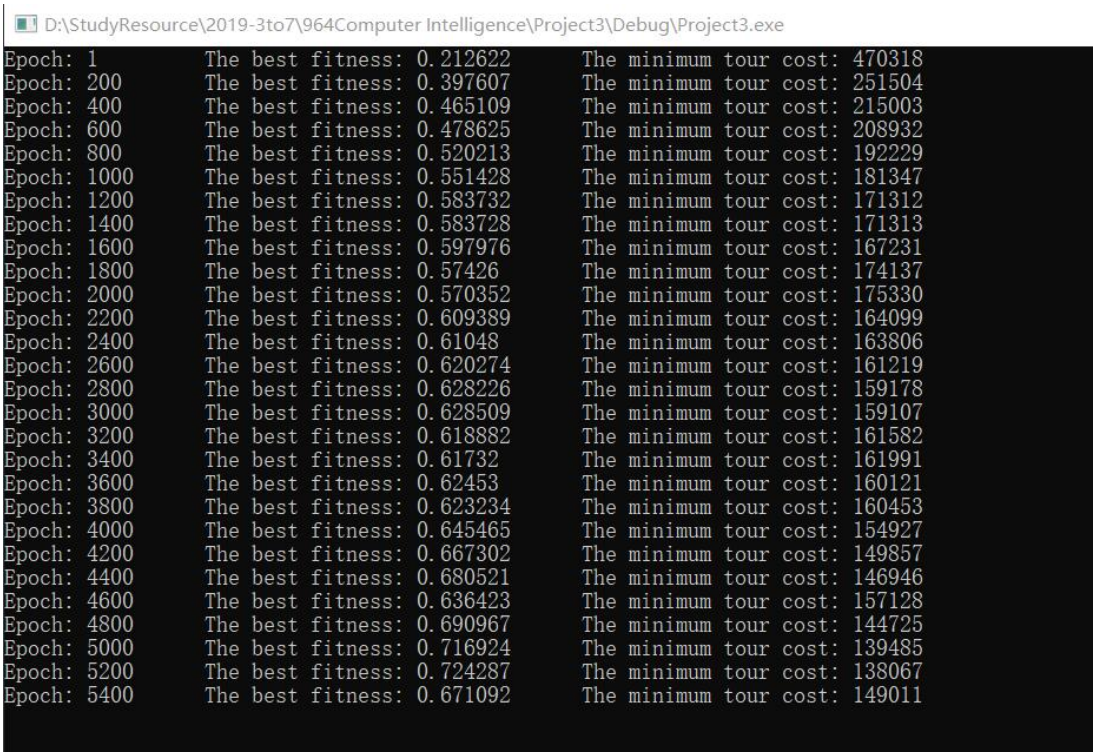

Figure 2: Experiment result of 200 individual length(print every 200th generation)

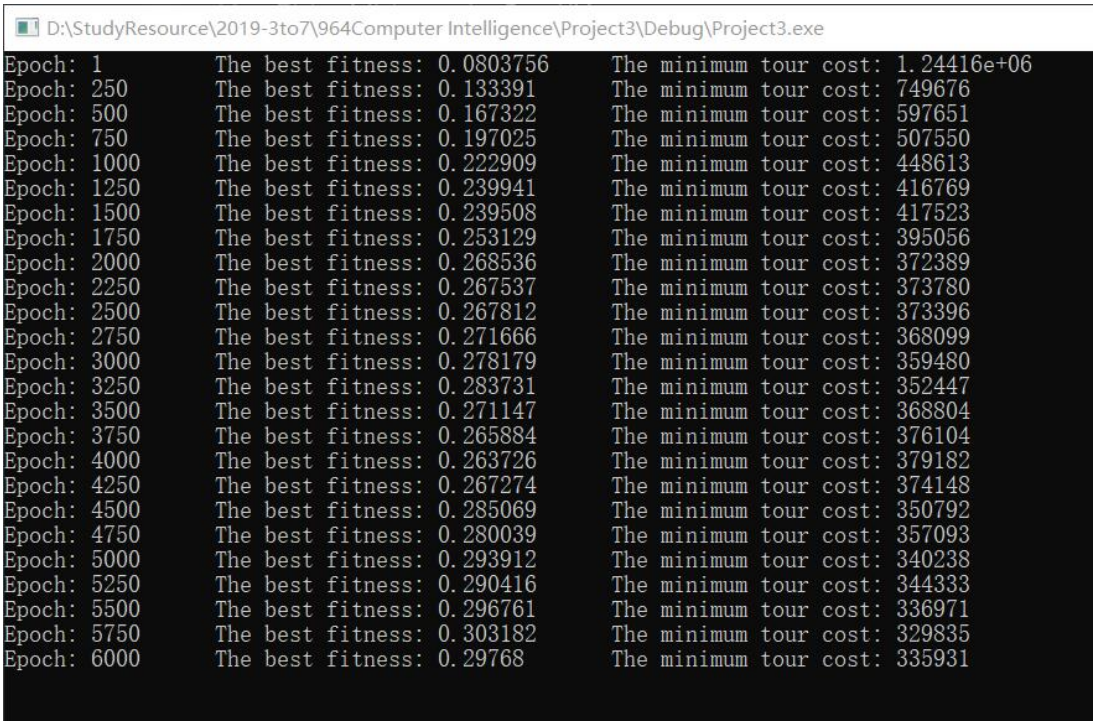

Figure 3: Experiment result of 500 individual length(print every 250th generation)