**21374389 牛鹏军 210711**

**理论分析**

|  |
| --- |
| e0624d7e48dbd24f9b2801a02e33e56 |

用遗传算法解决TSP问题，编码选择城市序号即可。

**算法设计**

|  |
| --- |
| f89be530c29ff90519d52a65ecba49d |

**编程实现**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #include <time.h>  #define NUM\_CITIES 10  #define POPULATION\_SIZE 100  #define GENERATIONS 100000  #define MUTATION\_RATE 0.05  // 城市坐标  int cities[NUM\_CITIES][2] = {  {1, 1}, {9, 9}, {8, 0}, {3, 1}, {7, 8},  {8, 1}, {1, 9}, {1, 5}, {8, 5}, {8, 6}  };  // 计算两点之间的距离  double distance(int x1, int y1, int x2, int y2) {  return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));  }  // 适应度函数：计算路径总长度的倒数  double fitness(int individual[]) {  double total\_distance = 0.0;  int visited[NUM\_CITIES] = {0}; // 记录城市是否已经访问过  visited[0] = 1; // 起点城市被访问过  for (int i = 0; i < NUM\_CITIES - 1; i++) {  int city1 = individual[i];  int city2 = individual[i + 1];  if (visited[city2]) { // 如果城市已经被访问过，说明路径出现重复  return 0.0; // 返回0表示不合法的路径  }  total\_distance += distance(cities[city1][0], cities[city1][1], cities[city2][0], cities[city2][1]);  visited[city2] = 1; // 将城市标记为已访问  }  return 1.0 / total\_distance;  }  // 选择函数：基于适应度选择一个个体（轮盘赌选择）  int select\_individual(double fitness\_values[]) {  double sum\_fitness = 0.0;  for (int i = 0; i < POPULATION\_SIZE; i++) {  sum\_fitness += fitness\_values[i];  }  double r = ((double) rand() / (RAND\_MAX)) \* sum\_fitness;  double partial\_sum = 0.0;  for (int i = 0; i < POPULATION\_SIZE; i++) {  partial\_sum += fitness\_values[i];  if (partial\_sum >= r) {  return i;  }  }  return POPULATION\_SIZE - 1; // 备用返回值  }  // 交叉函数（顺序交叉）  void crossover(int parent1[], int parent2[], int child[]) {  int start = 1 + rand() % (NUM\_CITIES - 1); // 确保起点不是0  int end = start + (rand() % (NUM\_CITIES - start));  // 复制父代1的城市到子代  for (int i = start; i <= end; i++) {  child[i] = parent1[i];  }  // 从父代2中选择未选择的城市添加到子代中  int current = (end + 1) % NUM\_CITIES;  for (int i = 0; i < NUM\_CITIES; i++) {  int candidate = parent2[(end + 1 + i) % NUM\_CITIES];  int found = 0;  for (int j = start; j <= end; j++) {  if (child[j] == candidate) {  found = 1;  break;  }  }  if (!found) {  child[current] = candidate;  current = (current + 1) % NUM\_CITIES;  }  }  child[0] = 0; // 确保起点为城市0  }  // 变异函数（交换变异）  void mutation(int individual[]) {  if (rand() % 100 < MUTATION\_RATE) { // MUTATION\_RATE 百分比的变异概率  int idx1 = 1 + rand() % (NUM\_CITIES - 1); // 确保不变异起点0  int idx2 = 1 + rand() % (NUM\_CITIES - 1);  // 确保 idx1 和 idx2 不同  while (idx1 == idx2) {  idx2 = 1 + rand() % (NUM\_CITIES - 1);  }  // 交换城市  int temp = individual[idx1];  individual[idx1] = individual[idx2];  individual[idx2] = temp;  }  }  // 初始化种群  void initialize\_population(int population[][NUM\_CITIES]) {  for (int i = 0; i < POPULATION\_SIZE; i++) {  population[i][0] = 0; // 确保起点为城市0  for (int j = 1; j < NUM\_CITIES; j++) {  population[i][j] = j;  }  // 打乱除起点城市0以外的其他城市  for (int j = 1; j < NUM\_CITIES; j++) {  int swap\_idx = 1 + rand() % (NUM\_CITIES - 1);  int temp = population[i][j];  population[i][j] = population[i][swap\_idx];  population[i][swap\_idx] = temp;  }  }  }  int main() {  srand(time(NULL)); // 初始化随机种子  int population[POPULATION\_SIZE][NUM\_CITIES];  int new\_population[POPULATION\_SIZE][NUM\_CITIES];  double fitness\_values[POPULATION\_SIZE];  // 初始化种群  initialize\_population(population);  // 进化种群  for (int generation = 0; generation < GENERATIONS; generation++) {  // 计算每个个体的适应度  for (int i = 0; i < POPULATION\_SIZE; i++) {  fitness\_values[i] = fitness(population[i]);  }  // 创建新种群  for (int i = 0; i < POPULATION\_SIZE; i += 2) {  int parent1\_idx = select\_individual(fitness\_values);  int parent2\_idx = select\_individual(fitness\_values);  crossover(population[parent1\_idx], population[parent2\_idx], new\_population[i]);  crossover(population[parent2\_idx], population[parent1\_idx], new\_population[i + 1]);  mutation(new\_population[i]);  mutation(new\_population[i + 1]);  }  // 将新种群复制到当前种群  for (int i = 0; i < POPULATION\_SIZE; i++) {  for (int j = 0; j < NUM\_CITIES; j++) {  population[i][j] = new\_population[i][j];  }  }  // 每10000代打印一次最佳适应度  if (generation % 10000 == 0) {  double best\_fitness = fitness\_values[0];  for (int i = 1; i < POPULATION\_SIZE; i++) {  if (fitness\_values[i] > best\_fitness) {  best\_fitness = fitness\_values[i];  }  }  printf("Generation %d: Best Fitness = %.5f\n", generation, best\_fitness);  }  }  // 找到最终种群中最好的个体并打印  int best\_individual\_idx = 0;  double best\_fitness = fitness\_values[0];  for (int i = 1; i < POPULATION\_SIZE; i++) {  if (fitness\_values[i] > best\_fitness) {  best\_fitness = fitness\_values[i];  best\_individual\_idx = i;  }  }  printf("Best Path Length: %.2f\n", 1.0 / best\_fitness);  printf("Best Path Sequence: ");  for (int i = 0; i < NUM\_CITIES; i++) {  printf("%d ", population[best\_individual\_idx][i]);  }  printf("\n");  return 0;  } |

**测试分析**

采用以下条件时，某次运行得到结果为{0，3，7，6，4，1，9，8，5，2}，与教材结论相同，是最短路径。

|  |
| --- |
| #define NUM\_CITIES 10  #define POPULATION\_SIZE 100  #define GENERATIONS 100000  #define MUTATION\_RATE 0.05 |
|  |

在测试时发现，有时无法得到最优解，这与个体数量、繁衍代数、变异概率的设置有关。后来发现增加繁衍代数可以解决，或者增加个体数量，调整变异概率。另外还与代码中使用了当前时间作为伪随机数种子有关，所以不同时间运行得到结果也可能不一样。

**结论**

遗传算法是一种比较经典的启发式算法，在求解复杂问题时有很大的优点。但是在编码方式与各算子的设置以及种群数量、繁衍代数、变异概率等方面需要注意。