遗传算法求解旅行商问题实验报告

22050605 杨新莹

1. 遗传算法的基本原理

遗传算法是一种模拟自然选择和遗传机制的优化算法，借鉴了达尔文的进化论原理。遗传算法模拟了生物进化中的遗传、交叉和突变的过程，以寻找最优解或近似最优解。

其基本原理包括以下几个关键步骤：

1）初始化种群：通过随机生成一组初始解来创建初始种群。

2）适应度评估：对种群中的个体按照问题的特定评价函数进行评估，以确定每个个体的适应度。适应度函数通常是问题特定的，旨在衡量个体解决问题的质量。

3）选择： 选择适应度较高的个体作为父代，以便通过遗传操作产生后代。常见的选择方法包括轮盘赌选择、锦标赛选择等。

4）交叉：从选择的父代个体中随机选取一对个体，通过交叉操作产生新的个体，以期望获得具有更好适应度的后代。交叉操作可以有不同的策略，如单点交叉、多点交叉、均匀交叉等。

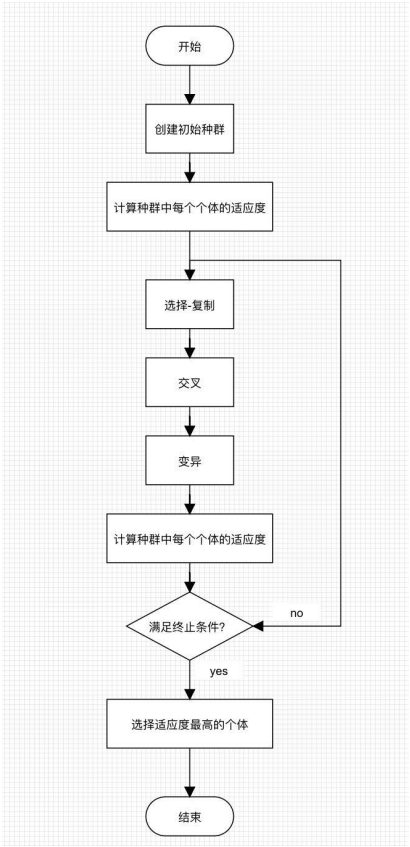
5）变异：在交叉操作之后，对部分个体进行突变操作，以保持种群的多样性。变异操作有助于避免算法陷入局部最优解。

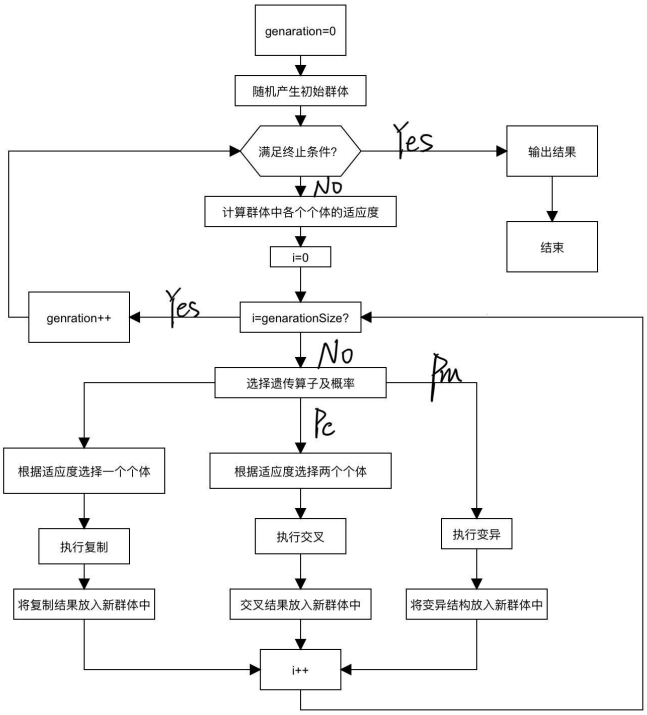
6）生成新种群：通过选择、交叉和变异操作生成新的种群，以替换原来的种群。

7）重复进化过程：重复进行选择、交叉、变异和生成新种群的过程，直到满足终止条件，比如达到最大迭代次数或找到满意的解。

通过模拟这些生物进化过程，遗传算法能够搜索大规模问题的解空间，并逐步优化解的质量，从而找到问题的较优解或最优解。

2. 遗传算法实现的流程图





3. 遗传算法的核心代码

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include<cmath>
#include <iomanip>

using namespace std;

// 城市的坐标点
struct City
{
	int x, y;
};

// 遗传算法参数
const int populationSize = 100;//种群数量
```

```
const int generations = 1000;//迭代次数
const double Pm = 0.002;//变异参数
const double Pc = 0.4;//交叉参数
const int numCities = 10;//城市数量
const int xnum = 20;//终止迭代
const double fit = 0.0000005;

int bestflag = 0;
double topfitness = -1;
double bestfitness[generations];
double minDistance[generations];

//随机城市坐标,坐标范围设为 100 内，并输出
vector<City> cities(numCities);

void initializeCities() {
	for (int i = 0; i < numCities; i++) {
		cities[i].x = rand() % 100;
		cities[i].y = rand() % 100;
	}
	cout << "随机生成的 10 个城市分别为：" << endl;
	for (int i= 0; i < numCities; i++)
	{
		cout << "City" << i	<< "(" << cities[i].x << "," << cities[i].y << ")"<<endl;
	}
}

// 计算两城市之间的距离
double calculateDistance(const City& city1, const City& city2) {
	int dx = city1.x - city2.x;
	int dy = city1.y - city2.y;
	return sqrt(dx * dx + dy * dy);
}

//适应度计算
double calculateFitness(const vector<int>& path) {
	//距离计算
	double totalDistance = 0.0;
	for (int i = 0; i < numCities - 1; i++) {
		totalDistance += calculateDistance(cities[path[i]], cities[path[i + 1]]);
	}
	totalDistance += calculateDistance(cities[path[numCities - 1]], cities[path[0]]); // 回到
起始城市
```

```
    return 1.0 / totalDistance; // 将距离转化为适应度，距离越短，适应度越高
}

// 计算路径的总距离
double calculateTotalDistance(const vector<City>& cities, const vector<int>& path) {
    double totalDistance = 0.0;
    for (int i = 0; i < path.size() - 1; ++i) {
        totalDistance += calculateDistance(cities[path[i]], cities[path[i + 1]]);
    }
    totalDistance += calculateDistance(cities[path.back()], cities[path[0]]);
    return totalDistance;
}

// 随机生成初始种群
vector<vector<int>> generateInitialPopulation(int numCities) {
    vector<vector<int>> population(populationSize);
    for (int i = 0; i < populationSize; ++i) {
        vector<int> path(numCities);
        for (int j = 0; j < numCities; ++j) {
            path[j] = j;
        }
        random_shuffle(path.begin() + 1, path.end());//随机打乱顺序
        population[i] = path;
    }
    return population;
}

// 选择父代
vector<int> selectParent(const vector<double>& fitness, const vector<vector<int>>&
population) {
    double sumFitness = 0.0;
    for (double f : fitness) {
        sumFitness += f;
    }
    double r = ((double)rand() / RAND_MAX) * sumFitness;
    //加权随机选择
    double partialSum = 0.0;

    for (int i = 0; i < populationSize; i++) {
        partialSum += fitness[i];
        if (partialSum >= r) {
            return population[i];
        }
    }
```

```
    return population[0];
}

// 交叉操作
vector<int> crossover(const vector<int>& parent1, const vector<int>& parent2) {
    int size = parent1.size();//可以进行交换的点的数量
    int start = rand() % size;
    int end = rand() % size;
    if (start > end)
    {
        swap(start, end);
    }
    vector<int> child(size, -1);//初始化为-1

    for (int i = start; i <= end; ++i) {
        child[i] = parent1[i];//每一个路径点
    }

    int index = 0;
    for (int i = 0; i < size; ++i) {
        if (child[i] == -1) {//检查 child 中的当前位置是否为-1，如果是，表示这个位置尚未
填充
            while (find(child.begin(), child.end(), parent2[index]) != child.end()) {
                //查找 child 中是否存在 parents[index],如果存在，则返回 child.end()（即
表示查找成功）,因为路径不能重复
                index = (index + 1) % size;
            }
            child[i] = parent2[index];
            index = (index + 1) % size;
        }
    }

    return child;
}

// 变异操作
void mutate(vector<int>& path) {
    int size = path.size();
    for (int i = 1; i < size; ++i) {
        if ((rand() / (double)RAND_MAX) < Pm) {
            int j = rand() % size;
            swap(path[i], path[j]);
        }
    }
```

```
}


int main() {
    srand(static_cast<unsigned>(time(nullptr)));//使用当前时间作为种子，以使每次运行程序时得到不同的随机数序列
    initializeCities();//初始化城市坐标

    vector<vector<int>> population = generateInitialPopulation(numCities);//初始化种群
    for (int generation = 0; generation < generations; ++generation) //迭代
    {

        //适应度计算
        vector<double> fitness(populationSize);
        for (int i = 0; i < populationSize; i++) {
            fitness[i] = calculateFitness(population[i]);
        }
        vector<vector<int>> newPopulation(populationSize);//新种群
        for (int i = 0; i < populationSize; ++i) {
            vector<int> child;
            if ((rand() / (double)RAND_MAX) < Pc) {
                vector<int> parent1 = selectParent(fitness, population);
                vector<int> parent2 = selectParent(fitness, population);
                child = crossover(parent1, parent2);
            }
            else
            {
                child= selectParent(fitness, population);
            }
            mutate(child);
            newPopulation[i] = child;
        }

        population = newPopulation;


        vector<int> bestPath = population[0];
        minDistance[generation] = calculateTotalDistance(cities, bestPath);
        bestfitness[generation] = fitness[0];
        for (int k = 0; k < populationSize;k++) {
            double distance = calculateTotalDistance(cities, population[k]);
            if (distance < minDistance[generation]) {
                minDistance[generation] = distance;
                bestPath = population[k];
```

```
                bestfitness[generation] = fitness[k];
            }
        }

        //输出
        cout << "Generation   " << setw(3) << setfill('0') << generation << " " << "路径为：
";
        for (int city : bestPath) {
            cout << city << "->";
        }
        cout<<"     最短路径距离是: " << minDistance[generation] << "适应度为：" <<
bestfitness[generation] << endl;

        //终止迭代条件
        //最佳的适应度和前几代差距小于设定值，进行数量的统计
        if (abs(topfitness - bestfitness[generation]) < fit) bestflag++;
        else bestflag = 0;
        if (bestflag > xnum) break;//统计值到达一定程度，终止迭代
        if (topfitness < bestfitness[generation]) topfitness = bestfitness[generation];

    }

    // 找到最佳路径
    vector<int> bestPath = population[0];
    double minDistance = calculateTotalDistance(cities, bestPath);
    for (const auto& path : population) {
        double distance = calculateTotalDistance(cities, path);
        if (distance < minDistance) {
            minDistance = distance;
            bestPath = path;
        }
    }

    // 输出结果
    cout << "最终的最佳路径为：";
    for (int city : bestPath) {
        cout << city << "->";
    }
    cout << "总距离：" << minDistance << endl;
    return 0;
}
```

4. 用遗传算法解决旅行商问题

4.1 旅行商问题描述

已知 N 个城市之间的相互距离,现有一个商人必须遍访这 N 个城市,并且每个城市只能访问一次,最后又必须返回出发城市。如何安排他对这些城市的访问次序,使其旅行路线总长度最短。

4.2 算法的执行过程和说明

1）初始化种群，创建一个初始种群，包含多个随机生成的个体（染色体）。每个个体表示一条可能的旅行路线

2）计算适应度函数，适应度为距离和的倒数，适应度越高，表示距离越短

3）根据适应度进行选择，这里使用的轮盘选择

4）在选定的个体中进行交叉，交叉时需要考虑到路径中城市不能重复

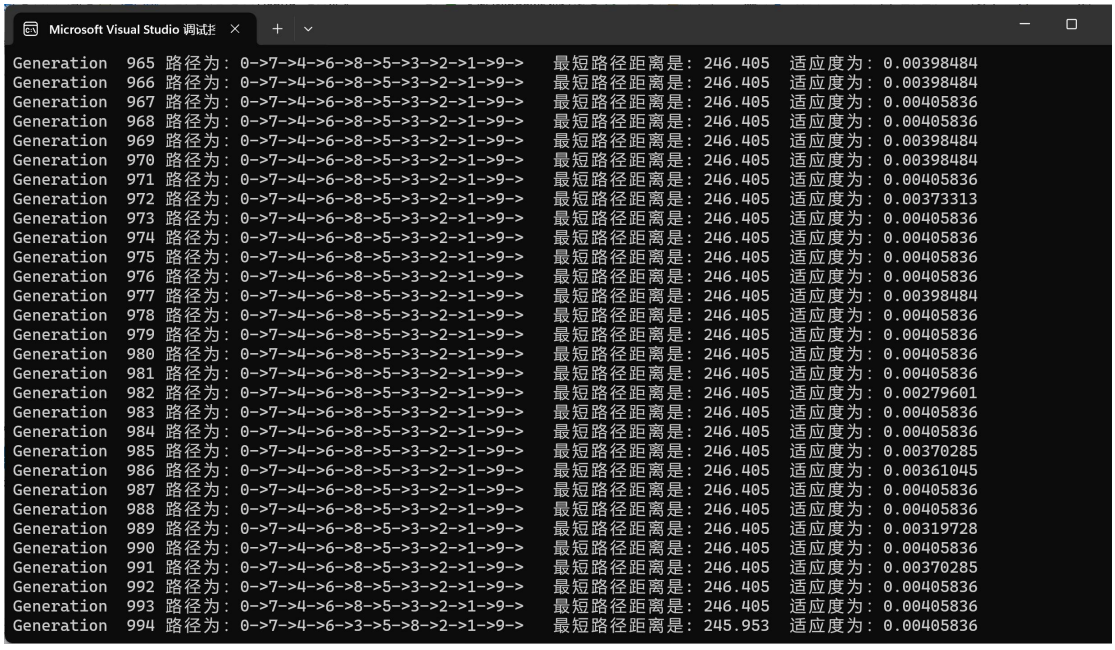
5）对后代进行变异，引入新的特性，防止遗漏

6）更新种群

7）重复 2-6，知道满足终止条件

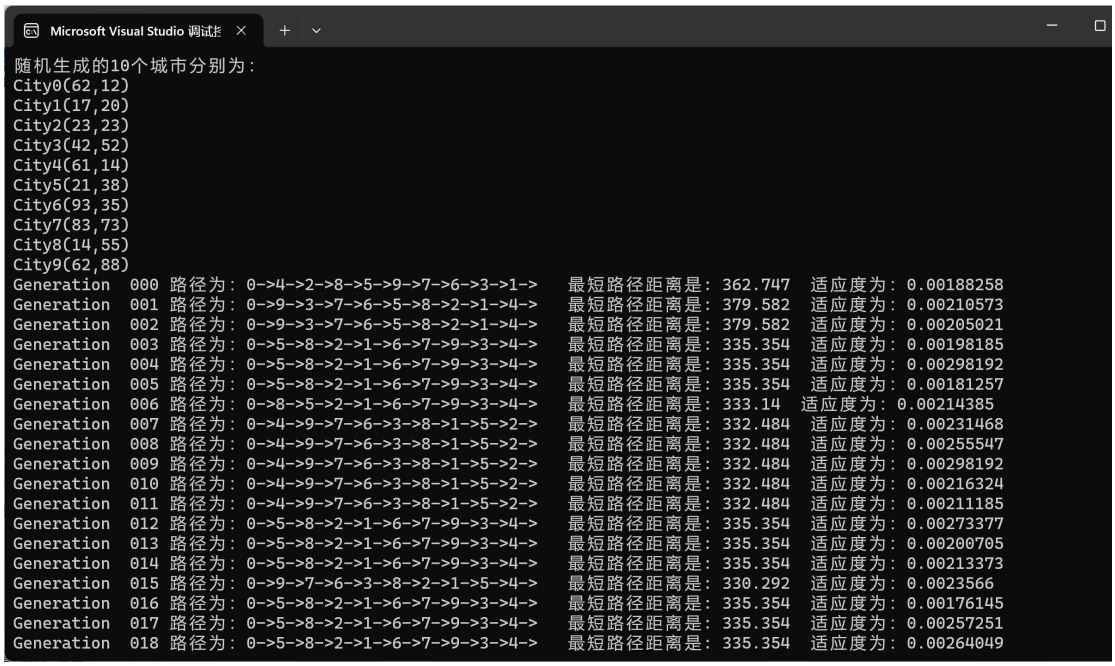
4.3 实验结果和分析

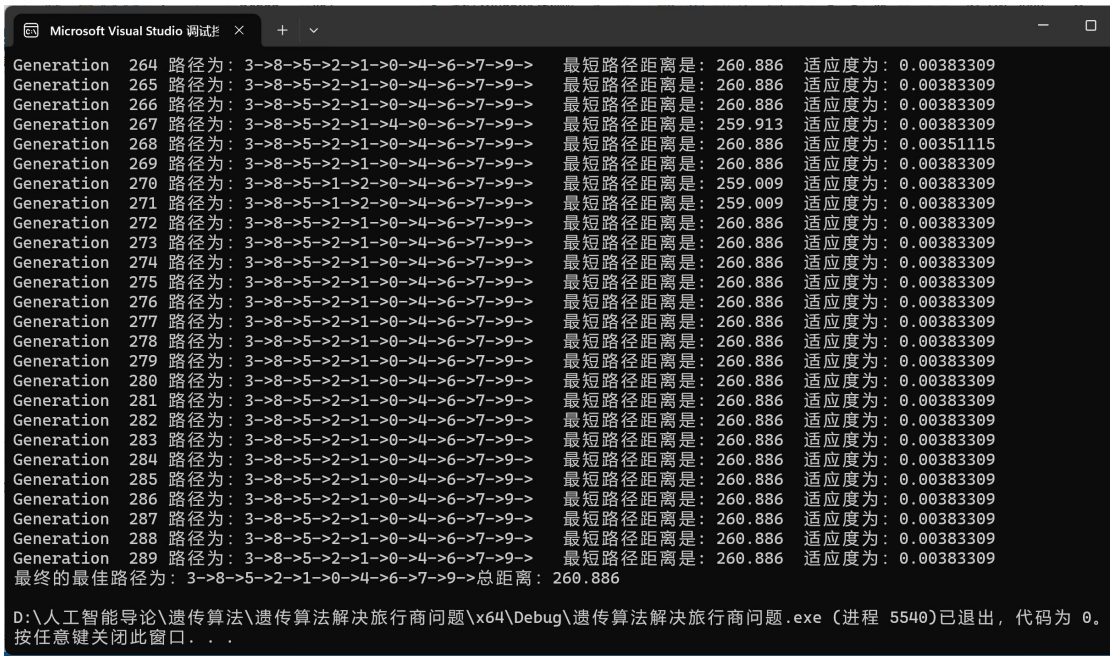
1）下面是终止条件仅为到达迭代次数，可以看到在前面（远小于 1000）代，已经达到最优解，由于次数过多，变异交叉的影响导致原本趋向于最优解的值发生了改变，明显不符合我们的的目标



分析后得出结论：当在过去的几代中，个体没有明显的改进，可以通过存储每一代获得的最佳适应度值，然后将当前的最佳值与预定的几代之前获得的最佳值进行比较来实现。如果差异小于某个阈值，则算法可以停止，即增加一个终止条件

2）下面便是增加了一个终止条件后的运行结果，可以看到，当适应度趋近于一个值时，种群停止迭代，输出最优解





4.4 不同遗传算子对解决旅行商问题的影响

不同的遗传算子对问题的解决会产生不同的影响。选择合适的遗传算子以及相应的参数，可以提高遗传算法的搜索效率和搜索质量。

1. 选择算子：选择算子决定哪些个体能够进入下一代。如果选择算子过于严格，可能导致优秀的个体被淘汰，从而降低搜索效率；如果选择算子过于宽松，可能导致劣质个体在种群中占主导地位，降低搜索质量。在旅行商问题中，常用的选择算子包括轮盘赌选择、锦标赛选择和自适应选择等。

2. 交叉算子：交叉算子负责在父母个体之间产生新的后代。不同的交叉策略会影响后代个体的多样性，从而影响搜索过程。例如，单点交叉可能导致局部最优解的产生，而多点交叉可以增加搜索空间，有助于找到全局最优解。交叉算子决定了如何将两个父代个体的基因重组，形成新的子代个体。在旅行商问题中，常用的交叉算子包括部分映射交叉（PMX）、顺序交叉（OX）和循环交叉（CX）等。

3. 变异算子：变异算子负责在个体中引入新的基因。变异算子的强度和频率会影响种群的多样性。较弱的变异算子可能导致算法陷入局部最优解，而较强的变异算子可以增加搜索空间，提高算法跳出局部最优解的概率。在旅行商问题中，常用的变异算子包括单点变异、插入变异和反转变异等。

5. 遗传算法的优缺点及解决方法

5.1 遗传算法的优点

1）全局最优

在许多情况下，优化问题具有局部最大值和最小值。这些值代表的解比周围的解要好，但并不是最佳的解。大多数传统的搜索和优化算法，尤其是基于梯度的搜索和优化算法，很容易陷入局部最大值，而不是找到全局最大值。遗传算法更有可能找到全局最大值。这是由于使用了一组候选解，而不是一个候选解，而且在许多情况下，交叉和变异操作将导致候选解与之前的解有所不同。只要设法维持种群的多样性并避免过早趋同，就可能产生全局最优解。

2）处理复杂问题

由于遗传算法仅需要每个个体的适应度函数得分，而与适应度函数的其他方面（例如导数）无关，因此它们可用于解决具有复杂数学表示、难以或无法求导的函数问题。

3）处理缺乏数学表达的问题

遗传算法可用于完全缺乏数学表示的问题。这是由于适应度是人为设计的。例如，想要找到最有吸引力颜色组合，可以尝试不同的颜色组合，并要求用户评估这些组合的吸引力。使用基于意见的得分作为适应度函数应用遗传算法搜索最佳得分组合。即使适应度函数缺乏数学表示，并且无法直接从给定的颜色组合计算分数，但仍可以运行遗传算法。

只要能够比较两个个体并确定其中哪个更好，遗传算法甚至可以处理无法获得每个个体适应度的情况。例如，利用机器学习算法在模拟比赛中驾驶汽车，然后利用基于遗传算法的搜索可以通过让机器学习算法的不同版本相互竞争来确定哪个版本更好，从而优化和调整机器学习算法。

4）耐噪音

一些问题中可能存在噪声现象。这意味着，即使对于相似的输入值，每次得到的输出值也可能有所不同。例如，当从传感器产生异常数据时，或者在得分基于人的观点的情况下，就会发生这种情况。尽管这种行为可以干扰许多传统的搜索算法，但是遗传算法通常对此具有鲁棒性，这要归功于反复交叉和重新评估个体的操作。

5）并行性

遗传算法非常适合并行化和分布式处理。适应度是针对每个个体独立计算的，这意味着可以同时评估种群中的所有个体。另外，选择、交叉和突变的操作可以分别在种群中的个体和个体对上同时进行。

6）持续学习

进化永无止境，随着环境条件的变化，种群逐渐适应它们。遗传算法可以在不断变化的环境中连续运行，并且可以在任何时间点获取和使用当前最佳的解。但是需要环境的变化速度相对于遗传算法的搜索速度慢。

5.2 遗传算法的缺点及解决方案

1）需要特殊定义

将遗传算法应用于给定问题时，需要为它们创建合适的表示形式——定义适应度函数和染色体结构，以及适用于该问题的选择、交叉和变异算子。

2）超参数调整

遗传算法的行为由一组超参数控制，例如种群大小和突变率等。将遗传算法应用于特定问题时，没有标准的超参数设定规则。

3）计算密集

种群规模较大时可能需要大量计算，在达到良好结果之前会非常耗时。可以通过选择超参数、并行处理以及在某些情况下缓存中间结果来缓解这些问题。

4）过早趋同

如果一个个体的适应能力比种群的其他个体的适应能力高得多，那么它的重复性可能足以覆盖整个种群。这可能导致遗传算法过早地陷入局部最大值，而不是找到全局最大值。为了防止这种情况的发生，需要保证物种的多样性。

5）无法保证的解的质量

遗传算法的使用并不能保证找到当前问题的全局最大值（但几乎所有的搜索和优化算法都存在此类问题，除非它是针对特定类型问题的解析解）。

6. 总结和讨论

遗传算法是一种基于自然选择和生物进化原理的优化算法，可以用于解决各种优化问题，如旅行商问题(TSP)。在遗传算法中，个体被视为染色体，每个染色体包含一组基因，这些基因表示问题的解决方案。算法通过模拟自然选择和生物进化过程，对个体进行选择、交叉和变异等操作，不断优化解决方案，最终找到问题的最优解。

在遗传算法解决 TSP 问题的过程中，首先需要定义个体的表示方式和适应度函数。通常，每个个体表示一个城市之间的路径，适应度函数用于评估路径的长度。然后，算法通过选择、交叉和变异等操作来生成新的个体，并使用适应度函数来评估它们的质量。最后，算法返回最优个体对应的解决方案。

遗传算法解决 TSP 问题的优点是简单、易于实现，并且可以处理大规模问题。然而，遗传算法也存在一些缺点，如可能会陷入局部最优解、计算复杂度高等。为了提高算法的性能，可以采用多种策略，如使用精英策略、使用启发式函数、采用多种交叉和变异操作等。

总之，遗传算法是一种有效的优化算法，可以用于解决 TSP 问题。通过不断改进算法，可以提高算法的性能，从而更好地解决实际问题。