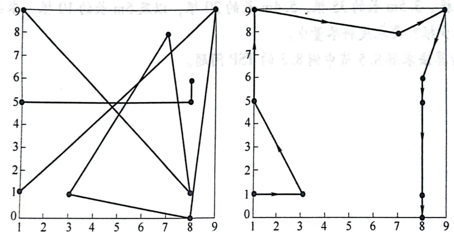
21374389 牛鹏军 210711

理论分析

【例 8.3】 旅行商问题（TSP）：10 个城市坐标依次为 {(1, 1), (9, 9), (8, 0), (3, 1), (7, 8), (8, 1), (1, 9), (1, 5), (8, 5), (8, 6)}。利用程序示例 8.3 的代码进行优化，优化旅行路径为 {0, 3, 7, 6, 4, 1, 9, 8, 5, 2}，{} 内的数字为城市索引，索引 0 表示第一个城市。原始数据和优化结果如图 8.11 所示。



a) 10个城市位置和原始顺序　　b) 模拟退火法优化后的结果

图 8.11　TSP 实例测试结果

用遗传算法解决 TSP 问题，编码选择城市序号即可。

算法设计

编码方式：

每个城市对应一个序号 i　$0 \le i \le n-1$，每个染色体为 n 个城市的一个序号串

适应度：

$$f(s) = \frac{1}{\sum_{i=1}^{n} d(c_i, c_{i+1})}$$

$d(c_i, c_{i+1})$ 为 c_i 到 c_{i+1} 距离.

选择算子：

轮盘赌.　$$f_i = \frac{F_i}{\sum_{j=1}^{N} F_j}$$

交叉算子：

对父1和父2，产生随机数，子代得到父1几个位置基因.

再从父2中不重复基因按序给子代。

变异算子：

随机产生2个变异位，换位.

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
```

```
Generation 40000: Best Fitness = 0.03454
Generation 41000: Best Fitness = 0.03454
Generation 42000: Best Fitness = 0.03454
Generation 43000: Best Fitness = 0.03454
Generation 44000: Best Fitness = 0.03454
Generation 45000: Best Fitness = 0.03454
Generation 46000: Best Fitness = 0.03454
Generation 47000: Best Fitness = 0.03454
Generation 48000: Best Fitness = 0.03454
Generation 49000: Best Fitness = 0.03454
Best Path Length: 28.95
Best Path Sequence: 0 3 7 6 4 1 9 8 2 5
```

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。