北 京 林 业 大 学

2019 学年—2020 学年第 2 学期计算机算法设计与实践实验报告书

专业：计算机科学与技术(创新实验班)　班级：计创 18

姓　　名：连月菡　　学　　号：181002222

实验地点：家　　任课教师：王春玲

实验题目：实验 6　TSP 问题的近似算法

实验环境：Visual Studio 2019 Community

一、 实验目的

（1）了解近似算法的局限性；
（2）掌握求解NP完全问题的一种方法：只对问题的特殊实例求解。

二、 实验内容

求解TSP问题的近似算法。

三、 实验步骤

1. **算法思路**

选择一个求解 TSP 的近似算法，有很多，在这次实验中考虑使用一种选择 Christofides' Algorithm，对于这种解法，运用多个测试用例进行测试，检验近似算法的稳定性,查看与最优解的差异。

对于 TSP 问题，应当定义一个结构体,存储一座城市的坐标信息。

1. 寻找最小生成树 T
2. 寻找在 T 中的奇度数顶点和最小匹配 M
3. 将 M 加入到 T 中
4. 寻找欧拉回路
5. 剪去多余的顶点

2. **C++代码**

定义一个 TSP 类

```
3.  class TSP
4.  {
5.  public:
6.      typedef struct City {
```

```
        int x;
        int y;
    };
    string in,out;//输入输出文件名
    vector<int>odds;//奇数结点
    int** cost;
    int n,pathLength;//城市数量  最短距离
    int** graph;
    int** path_vals;//路径
    vector<int> circuit;//欧拉路径
    vector<City> cities;
    vector<int>* adjlist;
    TSP(string in);//构造函数
    ~TSP();//析构函数
    int get_num() { return n; };
    int dis(struct City c1, struct City c2) { return floor((float)(sqrt((
c1.x - c2.x) * (c1.x - c2.x) + (c1.y - c2.y) * (c1.y - c2.y)) + .5)); }//
四舍五入
    void Match();//完美匹配
    void Euler(int start, vector<int>& path);//寻找欧拉回路
    void Hamilton(vector<int>& path, int& pathCost);//寻找哈密顿路径
    void Tree();//Prim 算法
};
```

TSP 类中的构造和析构函数

```
TSP::TSP(string input) {//构造
    ifstream in;
    in.open(input.c_str(), ios::in);
    int c,count = 0;//序号, 计数
    double x, y;
    while (!in.eof()) {
        in >> c >> x >> y;
        count++;
        City temp; temp.x = x; temp.y = y;
        cities.push_back(temp);
    }in.close();
    count--;n = count;
    graph = new int* [n];//初始化四个变量
    cost = new int* [n];
    path_vals = new int* [n];
    adjlist = new vector<int>[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new int[n];
        for (int j = 0; j < n; j++) graph[i][j] = 0;
```

```
    }
    for (int i = 0; i < n; i++) cost[i] = new int[n];
    for (int i = 0; i < n; i++) path_vals[i] = new int[n];
    for (int i = 0; i < cities.size(); i++) struct City cur = cities[i];
}

TSP::~TSP() {//析构
    for (int i = 0; i < n; i++) {
        delete[] graph[i];
        delete[] cost[i];
        delete[] path_vals[i];
    }
    delete[] path_vals;
    delete[] graph;
    delete[] cost;
    delete[] adjlist;
}
```

主函数

```
int main() {
    TSP tsp("C://Users/yuehan lian/Desktop/data.txt");//输入数据
    int city_num = tsp.get_num();
    for (int i = 0; i < tsp.n; i++)//构造距离矩阵
        for (int j = 0; j < tsp.n; j++)
            tsp.graph[i][j] = tsp.graph[j][i] = tsp.dis(tsp.cities[i], tsp.cities[j]);
    tsp.Tree();//最小生成树
    tsp.Match();//寻找与奇数顶点的最小权值匹配点 M
    int mindis = INT_MAX;//存储当前最短距离
    int mincity=0;//存储当前最短距离所代表的城市
    for (long t = 0; t < city_num; t++) {
        vector<int> path;
        tsp.Euler(t, path);
        int length;
        tsp.Hamilton(path, length);
        tsp.path_vals[t][0] = t; //设置起点
        tsp.path_vals[t][1] = length;//设置终点
        if (tsp.path_vals[t][1] < mindis) {
            mincity = tsp.path_vals[t][0];
            mindis = tsp.path_vals[t][1];
        }
    }
    tsp.Euler(mincity, tsp.circuit);//最小欧拉回路
    tsp.Hamilton(tsp.circuit, tsp.pathLength);
```

```
    cout << endl;
    cout << "路径为:" << endl << tsp.circuit.front() << "\t" << "距离
" << endl;
    for (vector<int>::iterator it = tsp.circuit.begin(); it != tsp.circuit.e
nd() - 1; ++it) {
        cout << *(it + 1) << "\t" << tsp.graph[*it][*(it + 1)] << endl;
    }
    cout << "\n 路径长度: " << tsp.pathLength << endl << endl;
}
```

01.寻找最小生成树 T

```
void TSP::Tree() {//利用 Prim 算法来确定最小生成树
    int* key = new int[n];
    bool* included = new bool[n];
    int* parent = new int[n];
    for (int i = 0; i < n; i++) {
        key[i] = 0x7fffffff;//初始化每个结点距离赋一个较大值
        included[i] = false;//最小生成树中是否有该点
    }
    key[0] = 0;//最小生成树的根的距离为 0
    parent[0] = -1;//没有父节点
    for (int i = 0; i < n - 1; i++) {
        //寻找未被包括在最小生成树的最近顶点
        int min = 0x7ffffff;
        int min_index;
        for (int i = 0; i < n; i++) {//遍历每个顶点
            if (included[i] == false && key[i] < min) {//如果这个顶点已经没有
未访问过的更近键值顶点
                min = key[i];//赋值给 min min_index
                min_index = i;
            }
        }
        included[min_index] = true;//找到了, 则设置为被包括在最小生成树里
        for (int j = 0; j < n; j++) {//检查最新加入的结点的没有被检测到的更近顶
点
            if (graph[min_index][j] && included[j] == false && graph[min_ind
ex][j] < key[j]) {
                parent[j] = min_index;//更新父节点
                key[j] = graph[min_index][j];//更新键值
            }
        }
    }
    for (int i = 0; i < n; i++) {//构建邻边
        int j = parent[i];
```

```
        if (j != -1) {
            adjlist[i].push_back(j);
            adjlist[j].push_back(i);
        }
    }
}
```

02.寻找在 T 中的奇度数顶点和最小匹配 M

```
void TSP::Match() {//用贪心法找到 M 的完美匹配
    int closest, length;
    std::vector<int>::iterator tmp, first;
    for (int i = 0; i < n; i++)
        if ((adjlist[i].size() % 2) != 0)//如果这个顶点度数是奇数
            odds.push_back(i);//放入奇数表中//寻找奇度数的顶点
    while (!odds.empty()) {
        first = odds.begin();
        vector<int>::iterator it = odds.begin() + 1;
        vector<int>::iterator end = odds.end();
        length = std::numeric_limits<int>::max();
        for (; it != end; ++it) {
            if (graph[*first][*it] < length) {//如果这个点比当前的更近,更新最近的长度和下标
                length = graph[*first][*it];
                closest = *it;
                tmp = it;
            }//两个点被匹配, 一直到奇度数顶点列表为空
        }
        adjlist[*first].push_back(closest);
        adjlist[closest].push_back(*first);
        odds.erase(tmp);
        odds.erase(first);
    }
}
```

04. 寻找欧拉回路

```
void TSP::Euler(int start, vector<int>& path) {//寻找欧拉回路
    vector<int>* tempList = new vector<int>[n];
    for (int i = 0; i < n; i++) {
        tempList[i].resize(adjlist[i].size());//复制近邻列表
        tempList[i] = adjlist[i];
    }
    stack<int> stack;
    int pos = start;
```

```
    path.push_back(start);
    while (!stack.empty() || tempList[pos].size() > 0) {
        if (tempList[pos].empty()) {//当前结点没有邻居
            path.push_back(pos);//加到路径中
            pos = stack.top();//移除上一个顶点
            stack.pop();
        }
        else {//当前结点有邻居
            stack.push(pos);//把顶点加入到栈里
            int neighbor = tempList[pos].back();
            tempList[pos].pop_back();//移除当前顶点和邻居之间的边
            for (int i = 0; i < tempList[neighbor].size(); i++) {
                if (tempList[neighbor][i] == pos) {
                    tempList[neighbor].erase(tempList[neighbor].begin() + i)
;
                }
            }
            pos = neighbor;//把邻居设置为当前结点
        }
    }
    path.push_back(pos);
}
```

05. 剪去多余的顶点

```
void TSP::Hamilton(vector<int>& path, int& pathCost) {//剪去多余的顶点
    bool* visited = new bool[n];//移除欧拉回路中被访问过的结点
    for (int i = 0; i < n; i++)
        visited[i] = 0;
    pathCost = 0;
    int root = path.front();
    vector<int>::iterator cur = path.begin();
    vector<int>::iterator iter = path.begin() + 1;
    visited[root] = 1;
    bool addMore = true;//遍历回路
    while (iter != path.end()) {
        if (!visited[*iter]) {
            pathCost += graph[*cur][*iter];//计算代价
            cur = iter;
            visited[*cur] = 1;
            iter = cur + 1;
        }
        else
            iter = path.erase(iter);
    }
```

```
    if (iter != path.end()) {
        pathCost += graph[*cur][*iter];
    }
}
```

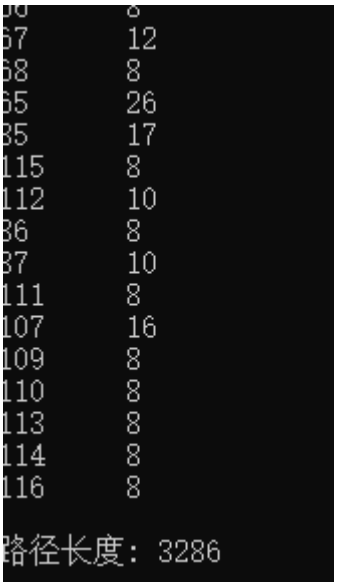
时间复杂度为多项式的范围内。
[TSP 的 Christofides 算法近似比为:1.5]

4.测试结果

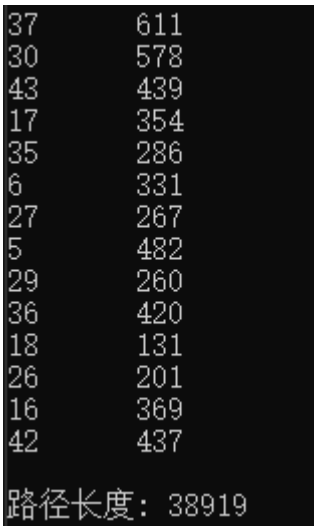
测试用例下载

测试结果标准

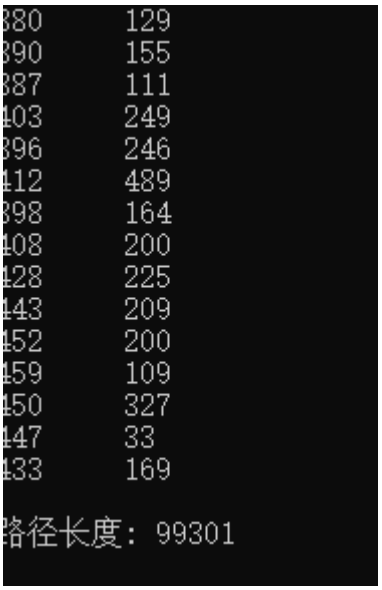
a280.tsp 最优解:2579



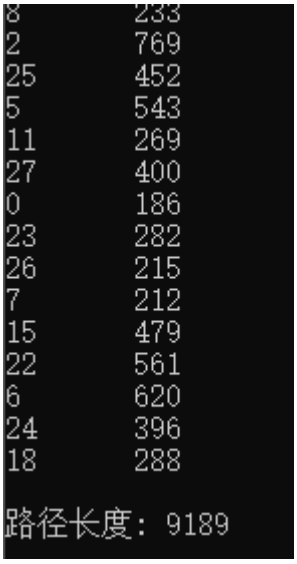
att48.tsp 最优解 10628



att532.tsp 最优解 27686



bayg29.tsp 最优解:1610



bays29.tsp 最优解:2020

```
20        228
4         272
8         233
2         769
25        452
5         543
11        269
27        400
0         186
23        282
26        215
7         212
15        479
22        561
6         620
24        396
18        288

路径长度: 9189
```

berlin52.tsp 最优解:7542

```
45        454
47        125
14        72
42        242
32        365
36        669
48        166
31        50
44        151
18        75
40        92
7         64
9         180
8         83

路径长度: 7972
```

bier127.tsp 最优解:118282

```
51        625
123       328
54        928
65        820
46        2026
48        478
52        232
117       259
47        259
45        706
93        991
111       1986
110       957
106       4190

路径长度: 127572
```

d1655.tsp 最优解:62128

```
1142    25
1155    33
1181    51
1193    64
1212    26
1234    46
1233    26
1232    25
1231    26
1235    102
1220    40
1219    25
1218    26
1245    57

路径长度: 79281
```

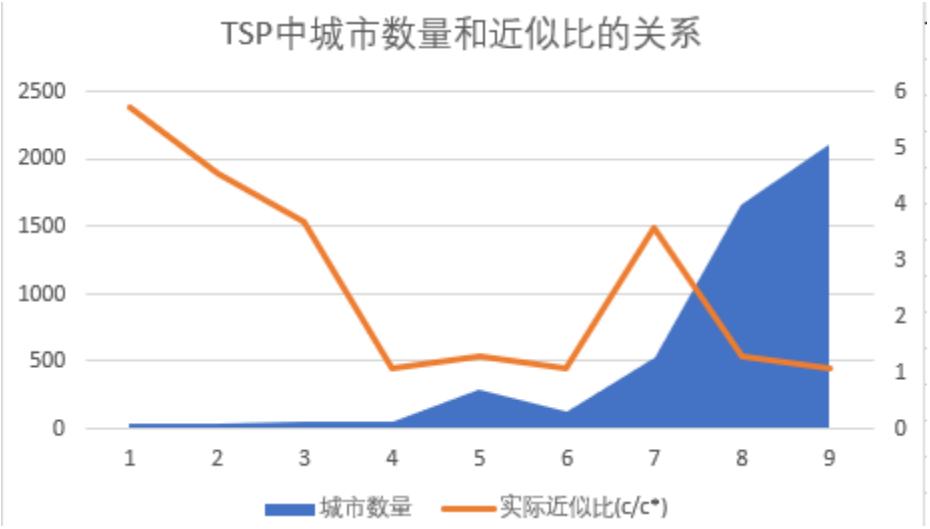
d2103.tsp 最优解:80450

```
1260    26
1259    25
1245    25
1246    26
1247    25
1257    25
1258    25
1248    56
1256    25
1249    36
1250    25
1251    26
1252    25
1239    64
1240    25
1310    92

路径长度: 86462
```

由上，得到下表:

城市数量	29	29	48	52	280	127	532	1655	2103
最优解c*	1610	2020	10628	7542	2579	118282	27686	62128	80450
近似最优值c	9189	9189	38919	7972	3286	127572	99301	79281	86462
实际近似比(c/c*)	5.70745	4.5490099	3.6619307	1.057014	1.27414	1.07854	3.5867	1.276091	1.07473



可以观察得到，在城市数量较大的时候，近似比趋于稳定，得到的结果较接近最优解。

四、实验心得

通过这次实验设计近似算法求解满足三角不等式的 TSP 问题，并求出近似比，理解了满足三角不等式的 TSP 问题的特点及应用实例。

并将求解满足三角不等式的 TSP 问题的近似算法应用于一般的 TSP 问题，考察它的近似性能，因为 NP 问题就是指其解的正确性可以在多项式时间内被检查的一类问题，从而理解了为什么说 TSP 问题是 NP 问题中最难的一个问题,因为在多项式的时间内,暂时还没有一个算法可以准确地找到 TSP 问题的最优解。