华东师范大学计算机科学技术系上机实践报告

课程名称：人工智能	**年级：2018级**	**上机实践成绩：**
指导教师：周爱民	**姓名： 汪子凡**	**创新实践成绩：**
上机实践名称：导航问题	**学号：10185102153**	**上机实践日期：2020/3/27**
上机实践编号：No.1	**组号：**	**上机实践时间：**

一、 问题介绍

知道1000个结点的二维坐标(从0开始), 并且某些结点间存在双向路径，长度为这两个结点的平面距离，求结点123到结点321的最短路径(长度已经经过的结点)。

二、 程序设计与算法分析

算法1： 最短路算法(Dijkstra求最短路)

用经典的Dijkstra算法求无向边的最短路，每次找到离开始结点最近的节点进行扩展，并更新与它关联点的距离。用朴素的遍历扩展更新需要是平方的复杂度，经过最小堆优化可以达到$O(VlogE)$。

其实可以看成启发函数h全为0的A^* 算法。

其中用数组d来记录某个结点与123起始结点的最短路，每次扩展即选d数组中值最小的一个进行扩展，并且扩展完这个结点后再用这个结点来更新d。最后的d[321]即为到结点123的最短路。在上述过程中，用pre数组来记录某个结点的前一个结点，这样就能把最短路的路径回溯出来。(具体实现在附件4代码中)

算法2： A*算法

由于两点间存在路径的话长度是两点的平面距离，若取离终点的距离为启发函数h，则h的关系符合三角不等式，是单调的。

由于h是单调的，每次排序可以按照g从小到大排序(可以用优先队列模拟open表), 从open表中取出的结点肯定是最短的，即已经在close表中的结点无需再扩展。并且用一个数组去模拟Close表(0表示在，1表示不在)，按照经典的A*图搜索算法即可，具体的实现和变量解释在附件4的代码中展现。

算法3： 局部搜索算法

可以用爬山法，将山顶作为离目标结点近的点，将山谷作为离目标结点远的点，每次构建当前点的邻域，然后随机挑出一个点，若这个点情况比当前节点好(即离目标结点近)，则向这个节点前进，构建新的邻域进行循环，直到最后到达321。

由于局部搜索方法未必是全局最优解，因此多进行了几组实验，当实验次数越多时，答案越接近前两种方法的答案。

三、 实验结果

用前两种算法得出一样的结果：

length: 760.1691

path: 123 -> 721 -> 607 -> 20 -> 739 -> 939 -> 321

由于这两种算法的过程并非完全一样，因此在这样较大的数据上能保持答案一致可以互相支持正确性。在自己构造了其他几组数据后，确定了这两种算法正确性。

用第三种算法答案和循环的次数有关：

1次：length: 1615.5867
path: 123->109->837->847->405->885->411->235->185->155->946->663->357->168->652->169->688->145->110->717->321

5次：length: 898.0018
path: 123->336->890->729->572->923->409->433->852->453->321

20次：length: 886.3907
path: 123->890->486->688->53->145->24->717->321

100次：length: 813.9104
path: 123->336->24->717->453->321

1000次：length: 793.4277
path: 123->797->486->222->610->39->150->321

四、 附件

算法1代码：

```
const double INF = 1e9;
const int maxn = 1e3 + 3;
typedef pair<double, int> P;

struct edge                                  //此结构体来记录边
{
    int to;                                 //这条边通往哪个节点
    double cost;                            //代价
    edge(int k1, double k2): to{k1}, cost{k2} {}
};

int pre[maxn], ans[maxn], V, E;        //pre记录一条路径，state记录条数
double d[maxn], coord[maxn][2];
vector<edge> G[maxn];
priority_queue< P, vector<P>, greater<P> > que;

void Dijkstra(int s)
{
    for(int i = 0; i < V; i++)
        d[i] = INF;                 //首先s到其他节点的距离初始化成INF
    d[s] = 0;
    que.push(P(0, s));

    while(!que.empty())                     //每次扩展离s最近的结点
    {
        P p = que.top(); que.pop();
```

```
        int v = p.second;
        if(d[v] < p.first) continue;
        for(unsigned int i = 0; i < G[v].size(); i++)            //更新d
        {
            edge e = G[v][i];
            if(d[e.to] > d[v] + e.cost)     //小于情况
            {
                pre[e.to] = v;                  //更新前驱结点
                d[e.to] = d[v] + e.cost;
                que.push(P(d[e.to], e.to));
            }
        }
    }
}

int main(int argv, char** args)
{
    int x, y, tmp;
    V = 1000;
    FILE *p1 = fopen(args[1], "r");                 //读取节点坐标
    for(int i = 0; i < V; i++)
    {
        fscanf(p1, "%d %lf %lf", &tmp, &coord[i][0], &coord[i][1]);
    }
    fclose(p1);

    p1 = fopen(args[2], "r");
    while(~fscanf(p1, "%d %d", &x, &y))             //读取边
    {
        E++;
        double c = sqrt((coord[x][0] - coord[y][0])*(coord[x][0] - coord[y][0])
+ (coord[x][1] - coord[y][1])*(coord[x][1] - coord[y][1]));
        G[x].push_back(edge(y, c));
        G[y].push_back(edge(x, c));
    }
    fclose(p1);

    Dijkstra(123);
    printf("length: %.4f\n\n", d[321]);

    int cnt = 0;                            //回溯记录路径
    for(int i = 321; i != 123; i = pre[i])
        ans[cnt++] = i;
    printf("path: 123");
    for(int i = cnt - 1; i >= 0; i--)
        printf(" -> %d", ans[i]);
    putchar('\n');
}
```

算法2代码：

```
/*A*算法，用优先队列来模拟open表(按照f值从小到大排序)，每次取出f最小的
用一个数组去模拟Close表(0表示在，1表示不在)
*/
```

```cpp
#include <bits/stdc++.h>

using namespace std;
typedef pair<double, int> P;                        //first为节点的f值，second为标号
const int maxn = 1e3 + 3;
const double INF = 1e9;

struct node
{
    int to;
    double c;
    node(int k1, double k2): to{k1}, c{k2} {}
};
vector<node> G[maxn];
priority_queue< P, vector<P>, greater<P> > Open;
double coord[maxn][2], h[maxn], g[maxn];
int Close[maxn], pre[maxn], V, s, t, ans[maxn];

double Astar()
{
    for(int i = 0; i < V; i++)
        g[i] = INF;
    g[s] = 0;                                   //开始时 g[s] = 0
    Open.push(P(g[s] + h[s], s));                       //将结点s放入open表
    while(!Open.empty())
    {
        P p = Open.top();  Open.pop();                      //取出open表的第一个结点
        int v = p.second;                                   //fist是第一个结点的f值，
second是这个结点的标号
        if(Close[v]) continue;                              //若是这个结点已经被放入close
表，则不需要再次遍历
        Close[v] = 1;
        if(v == t) return p.first;                          //如果为所求，Exit(Success)

        for(unsigned i = 0; i < G[v].size(); i++)       //Expand
        {
            node e = G[v][i];
            if(Close[e.to] || g[e.to] < g[v] + e.c) continue;      //如果这个结点在
Close表中，或者在open表中但比现在情况更优，则不需要再次扩展
            pre[e.to] = v;                                  //记录前驱结点
            Open.push(P(g[v] + e.c + h[e.to], e.to));           //f = g(pre) + c
+ h
            g[e.to] = g[v] + e.c;
        }
    }
    return -1;                  //当open表为空，Exit(fail)
}

int main(int argv, char** args)
{
    int x, y, tmp;
    s = 123, t = 321;
    V = 1000;
    FILE *p1 = fopen(args[1], "r");
    for(int i = 0; i < V; i++)                          //读取节点坐标
    {
        fscanf(p1, "%d %lf %lf", &tmp, &coord[i][0], &coord[i][1]);
    }
```

```
    fclose(p1);

    p1 = fopen(args[2], "r");
    while(~fscanf(p1, "%d %d", &x, &y))                    //读取边
    {
        double c = sqrt((coord[x][0] - coord[y][0])*(coord[x][0] - coord[y][0])
+ (coord[x][1] - coord[y][1])*(coord[x][1] - coord[y][1]));
        G[x].push_back(node(y, c));
        G[y].push_back(node(x, c));
    }
    fclose(p1);

    for(int i = 0; i < V; i++)                             //求得h值
        h[i] = sqrt((coord[i][0] - coord[t][0])*(coord[i][0] - coord[t][0]) +
(coord[i][1] - coord[t][1])*(coord[i][1] - coord[t][1]));

    double res = Astar();
    if(res == -1) printf("No Path\n");          //若是不存在路径
    else
    {
        printf("length: %.4f\n\n", res);

        int cnt = 0;
        for(int i = t; i != s; i = pre[i])                 //回溯
            ans[cnt++] = i;
        printf("path: 123");
        for(int i = cnt - 1; i >= 0; i--)
            printf(" -> %d", ans[i]);
        putchar('\n');
    }
}
```

算法3代码：

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int maxn = 1e3 + 3;
const int INF = 2e9 + 10;

vector<int> G[maxn];

double coord[maxn][2], d[maxn], ans;
int trace[maxn], ansTrace[maxn], len, surround[maxn];   //ansTrace记录循环中最好的
情况

double cal(int x, int y)                                //计算两个点的距离
{
    return  sqrt((coord[x][0] - coord[y][0])*(coord[x][0] - coord[y][0]) +
(coord[x][1] - coord[y][1])*(coord[x][1] - coord[y][1]));
}

int main(int argv, char** args)
{
```

```
    int s = 123, t = 321, V = 1000, k1, x, y;
    FILE *p1 = fopen(args[1], "r");
    for(int i = 0; i < V; i++)                          //读取节点坐标
    {
        fscanf(p1, "%d %lf %lf", &k1, &coord[i][0], &coord[i][1]);
    }
    fclose(p1);

    p1 = fopen(args[2], "r");
    while(~fscanf(p1, "%d %d", &x, &y))                 //读取边
    {
        G[x].push_back(y);
        G[y].push_back(x);
    }
    fclose(p1);

    for(int i = 0; i < V; i++)                          //计算每个点与点t的距离
        d[i] = cal(i, t);

    ans = INF;
    for(int cas = 1; cas <= 10; cas++)                  //进行十次局部搜索来优化答案
    {
        int cur = s, cnt = 0, num, p;                   //cur表示当前搜索点，num来表示邻域的大小，p表示搜索到邻域的第几个点
        double tmpAns = 0;                              //tmpAns表示当前的路径值
        trace[cnt++] = cur;                             //cnt来记录路径
        while(cur != t)                                 //当到达t时跳出
        {
            num = G[cur].size();
            p = 0;
            for(int i = 0; i < num; i++)
                surround[i] = G[cur][i];
            //for(int i = 0; i <num; i++)
              //  printf("%d ", surround[i]);
           // putchar('\n');
            random_shuffle(surround, surround + num);   //将邻域surround的元素随机化
            while(p < num)
            {
                int tmp = surround[p];                  //挑选一个元素tmp
                if(d[cur] > d[tmp])                     //如果tmp比cur离目标更近，重新构造邻域
                {
                    trace[cnt++] = tmp;
                    tmpAns += cal(cur, tmp);
                    cur = tmp;
                    break;
                }
                p++;
            }
            if(p == num) break;                         //若是cur已是该邻域的最大值
        }
        if(cur == t && tmpAns < ans)                    //如果成功搜到了t
        {
            ans = tmpAns;
            len = cnt;
```

```
            for(int j = 0; j < cnt; j++)
                ansTrace[j] = trace[j];
        }
    }

    printf("length: %.4f\npath: ", ans);
    for(int i = 0; i < len - 1; i++)
        printf("%d->", ansTrace[i]);
    printf("%d\n\n", 321);
}
```