湖南大學

人工智能
实验报告

姓名：杨杰
学号：201908010705
班级：计科 1907

实验一：搜索算法求解问题

一、 实验名称

实验一：搜索算法求解问题

二、 实验目的

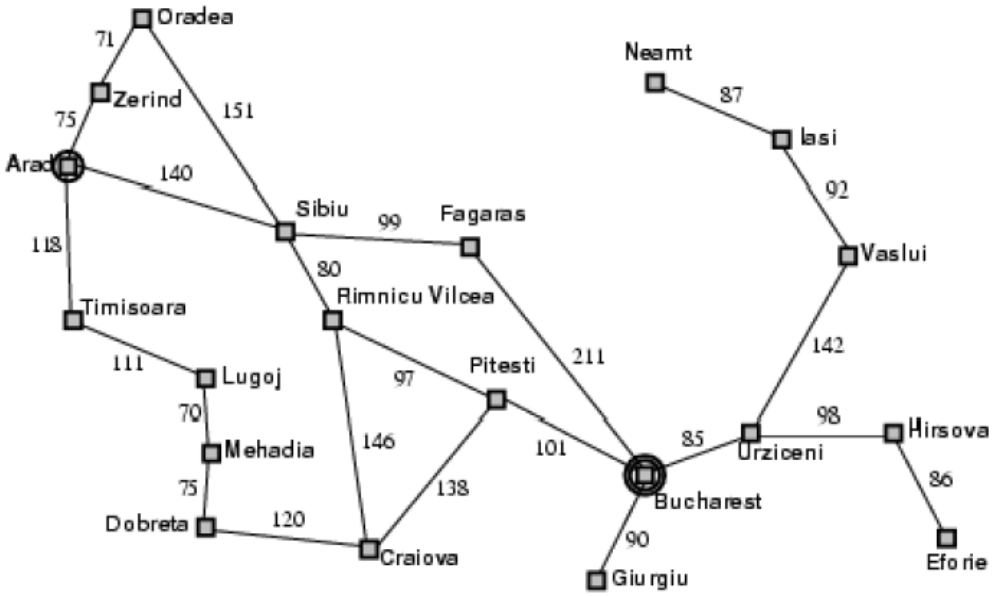
1、 掌握有信息搜索策略的算法思想；

2、 能够编程实现搜索算法；

3、 应用 A*搜索算法求解罗马尼亚问题。

三、 实验内容及步骤

实训内容：2-1 第三章 通过搜索进行问题求解



1、创建搜索树；

2、实现 A*搜索算法；

3、使用编写的搜索算法代码求解罗马尼亚问题；

4、分析算法的时间复杂度。

四、 算法原理及步骤

算法原理：

A*算法的原理是设计一个代价估计函数：其中评估函数 F(n)是从起始节点通过节点 n 到达目标节点的最小代价路径的估计值，函数 G(n)是从起始节点到 n 节点的已走过路径的实际代价，函数 H(n)是从 n 节点到目标节点可能的最优路径的估计代价 。

函数 H(n)表明了算法使用的启发信息，它来源于人们对路径规划问题的认识，依赖某种经验估计。根据 F(n)可以计算出当前节点的代价，并可以对下一次能够到达的节点进行评估。

采用每次搜索都找到代价值最小的点再继续往外搜索的过程，一步一步找到最优路径。

算法步骤：

路径搜索过程如下：

1. 首先创建两个集合，一个存储待访问的节点（openList），一个存储已经访问过的节点(closeList).

2. 添加起点到 openList 列表，并且计算该点的预估值 f(n) = g(n) + h(n).

3. 查找 openList 里预估值最小的节点，作为当前访问的节点，并且从 openList 中删除该节点，把该节点添加到 closeList 中，代表已经访问过了。

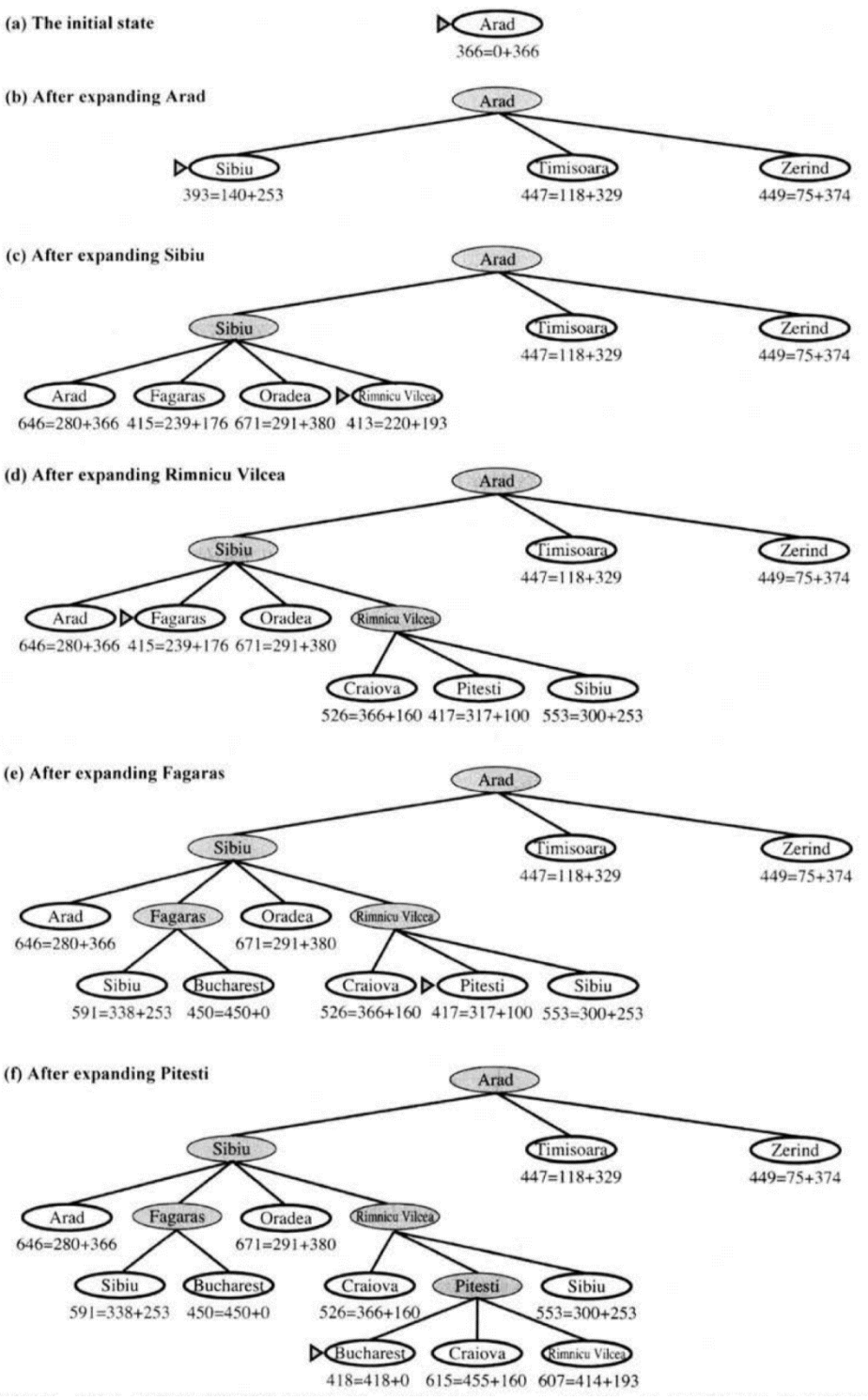
4. 获取当前节点的相邻节点，计算出他们的预估值，并且添加到 openList 列表中，若已经在 openList 中，则判断是否更新其预估值。

除了计算预估值，我们还需要把当前节点作为每个邻居节点的父节点，以便后面确定最终的路线。

5. 重复以上步骤 3 - 4 ，直到找到目标节点位置为止。

6. 循环输出目标节点的父节点，得到我们需要的路径。

搜索树：



五、 实验代码

```
while (!openList.empty())
    {
        /*取代价最小的节点*/
        node cur = openList[0]; //取首节点，代价最小
        if (cur.name == goal)   //目标
            return;
```

```
        openList.erase(openList.begin()); //从openList序列中删除这个节点
        list[cur.name] = false;           //将当前节点标记为不在openList
中
        closeList[cur.name] = true;       //将当前节点加入closeList

        /*遍历子节点*/
        for (int i = 0; i < 20; i++)
        {
            if (graph.getEdge(cur.name, i) != -1 && !closeList[i]) //节
点相邻并且不在close中，可访问
            {
                if (list[i]) //如果在list序列中，说明属于扩展节点集
                {
                    int j = 0;
                    for (int j = 0; j < openList.size(); j++)
                    { //遍历，找到当前节点的位置
                        if (openList[j].name == i)
                            break;
                    }
                    /*刷新节点*/
                    if (cur.g + graph.getEdge(cur.name, i) <
openList[j].g)
                    {
                        openList[j].g = cur.g + graph.getEdge(cur.name,
i); //更新g
                        openList[j].f = openList[j].g +
openList[j].h;      //更新f
                        sort(openList.begin(),
openList.end());           //排序
                        parent[i] =
cur.name;                                 //更新parent
                    }
                }
                else //节点不在openList，则创建一个新点，加入openList扩展
集
                {
                    node newNode(i, cur.g + graph.getEdge(cur.name, i),
h[i]);
                    parent[i] = cur.name;
                    openList.push_back(newNode);
                    sort(openList.begin(), openList.end()); //排序
                    list[i] = true;
                }
            }
```

```
        }
    }
}
```

时间复杂度分析：

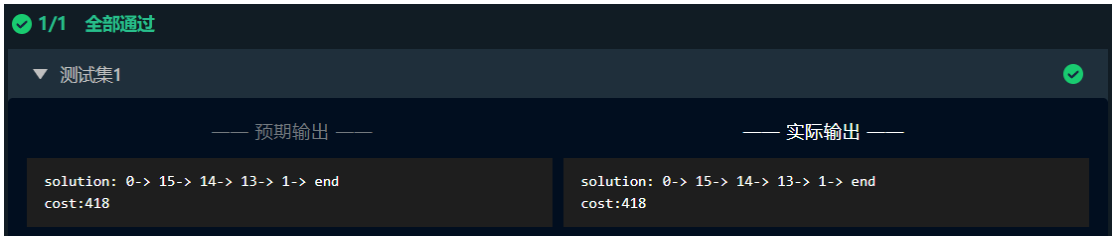
外循环：每次从 openList 中取出点，共取 n 次，

内循环：遍历它的邻接点 n(E)，并将这些邻接点放入 openList 中，若已存在则遍历 openList 找到邻接点，对 openList 进行排序，openList 表大小是 O(n)量级的，若用快排就是 O(nlogn)，所以内循环的时间复杂度为

O(n+n*logn) = O(n*logn)

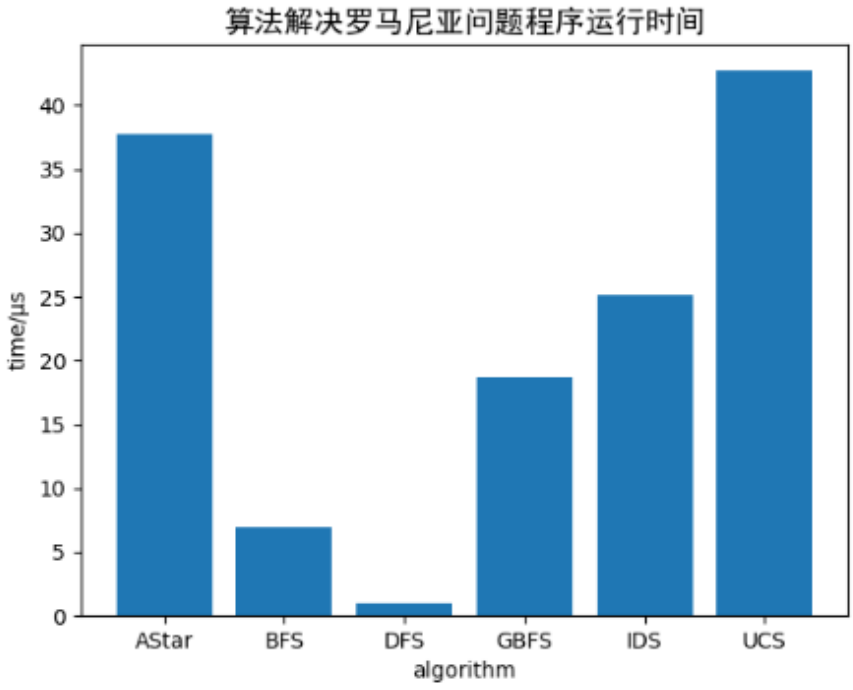
总复杂度为 O(n* n*logn) = O(n^2*logn)

运行结果：



六、 实验结果与分析

实验结果：



AStar:

```
Arad Bucharest
solution: Arad-> Sibiu-> Rimnicu-> Pitesti-> Bucharest-> end
cost:418
time: 37.8渭s
```

BFS:

```
Arad Bucharest
solution: Arad-> Sibiu-> Fagaras-> Bucharest-> end
cost:450
time: 6.9渭s
```

DFS:

```
Arad Bucharest
solution: Arad-> Sibiu-> Fagaras-> Bucharest-> end
cost:450
time: 1渭s
```

GBFS:

```
Arad Bucharest
solution: Arad-> Sibiu-> Fagaras-> Bucharest-> end
cost:450
time: 18.7渭s
```

IDS:

```
Arad Bucharest
min_depth:3
solution: Arad-> Sibiu-> Fagaras-> Bucharest-> end
cost:450
time: 25.1渭s
```

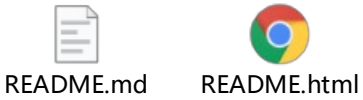
UCS:

```
Arad Bucharest
solution: Arad-> Sibiu-> Rimnicu-> Pitesti-> Bucharest-> end
cost:418
time: 42.7渭s
```

（注：编码问题，此处`渭 s`应为`μs`）

分析：

通过上述实验数据，我们不难看出 DFS 用时最少，但是 BFS、DFS、GBFS、IDS 都陷入了局部最优解，而 AStar 和 UCS 可以得到全局最优解，其中 AStar 耗时最少。



说明文档：

（请使用 markdown 编辑器打开 README.md[在本PDF附件中]，或者使用浏览器打开README.html）

七、 思考题

1、宽度优先搜索，深度优先搜索，一致代价搜索，迭代加深的深度优先搜索算法哪种方法最优？

根据我的程序跑出来的结果，在时间上明显是深度优先搜索耗时最短。但是，深度优先搜索和迭代加深的深度优先搜索都不具有完备性和最优性，宽度优先搜索最优性具有条件，即路径花费必须关于节点深度呈非递减函数，这三种搜索算法都会陷入局部最优解。只有一致代价搜索可以得到全局最优解。一致代价搜索扩展的是当前路径代价最小的结点，对于一般性的步骤代价而言算法是最优的。

2、贪婪最佳优先搜索和 A*搜索哪种方法最优？

贪婪最佳优先搜索耗时少，但会陷入局部最优解。考虑到实际问题的搜索求解时，A* 搜索算法是最优的。如果我们想要找到最小代价的解，首先扩展评估函数值最小的结点是合理的。可以发现这个策略不仅仅合理：假设启发式函数满足特定的条件，A* 搜索既是完备的也是最优的。

其中，A* 搜索算法的最优性取决于：如果启发式函数是可采纳的，那么 A* 的树搜索版本是最优的，如果启发式函数是一致的，那么图搜索的 A* 算法是最优的。

3、分析比较无信息搜索策略和有信息搜索策略。

无信息搜索指的是除了问题定义中提供的状态信息外没有任何附加信息，只是简单计算到达各个结点所需要的消耗值并比较。有信息搜索又称为启发式搜索，它是利用问题拥有的启发信息来引导搜索，达到减少搜索范围、降低问题复杂度的目的，它能比无信息搜索策略更有效的进行问题求解。

八、 实验总结

实验收获：

通过本实验我了解到了 4 种无信息搜索策略(BFS、DFS、IDS、UCS)和 2 种有信息搜索策略(GBFS、AStar)的算法思想以及它们的复杂度。

（1）BFS（宽度优先搜索）

宽度优先搜索是简单搜索策略，先扩展根结点，接着扩展根结点的所有后继，然后再扩展它们的后继，依此类推。一般地，在下一层的任何结点扩展之前，搜索树上本层深度的所有结点都应该已经扩展过。

（2）DFS（深度优先搜索）

深度优先搜索总是扩展搜索树的当前边缘结点集中最深的结点。搜索很快推进到搜索树的最深层，那里的结点没有后继。当那些结点扩展完之后，就从边缘结点集中去掉，然后搜索算法回溯到下一个还有未扩展后继的深度稍浅的结点。

（3）IDS（迭代加深的深度优先搜索）

从 0 开始递增到一个深度最大值，不断执行深度受限搜索，直到找到解为止。此算法需要先用到深度受限搜索，深度受限搜索原理：对深度优先搜索设置界限 I。深度为 l 的结点被当作没有后继对待。这种方法称为深度受限搜索。

（4）UCS（一致代价搜索）

总是从未扩展的节点中选择到起始节点耗费最小的节点来扩展，然后不断地更新其邻接点到起始点的路径耗散。

（5）GBFS（贪婪最佳优先算法）

贪婪最佳优先搜索，试图扩展离目标最近的结点，理由是这样可能可以很快找到解。因此，它只用启发式信息，即 f(n) = h(n)。

（6）A*搜索

它对结点的评估结合了 g(n)，即到达此结点已经花费的代价，和 h(n)，从该结点到目标结点所花代价: f(n) = g(n) + h(n).每次选择评估值 f(n)最小的节点扩展。

clock()函数、GetTickCount()函数、timeGetTime()函数、Boost 库中的 timer、高精度时控函数 QueryPerformanceFrequency()和 QueryPerformanceCounter()，以上函数都可以记录程序运行时间，但由于数据量较小时，程序运行时间很短，前三个函数的精度不足以计时，这时应该选用高精度时控函数 QueryPerformanceFrequency()和 QueryPerformanceCounter()。

难点重点讨论：

图搜索算法的基本流程如下：

1.创建一个容器，一般称为 openlist，用来存储将要访问的节点

2.将起点加入容器

3.开始循环：

- 弹出：从容器中取出一个节点
- 扩展：获取该节点周围的节点，将这些节点放入容器

深度优先，顾名思义即深度越大的节点越会被优先扩展。在 DFS 中，使用栈(Stack)数据结构来实现上述特性。DFS 能够快速地找到一条路径，是一种以时间换空间的方法。DFS 具有“不撞南墙不回头”的特性。

与之相比，BFS 在搜索时呈波状推进形式，一路稳扎稳打，它是一种以时间换空间的方法，能够保证搜索到的路径是最优的。为了实现波状推进搜索特性，BFS 采用队列(Queue)作为 openlist 的数据结构。

BFS 和 DFS 的区别主要在于节点的弹出策略，根据弹出策略的区别，分别使用了队列和栈两种数据结构，而队列和栈作为两种相当基本的容器，只将节点进入容器的顺序作为弹出节点的依据，并未考虑目标位置等因素，这就使搜索过程变得漫无目的，导致效率低下。

启发式搜索算法就是用来解决搜索效率问题的。如 GBFS、Dijkstra、A*.

GBFS 也是图搜索算法的一种，它的算法流程和 BFS、DFS 并没有本质的不同，区别仍在于 openlist 采用的数据结构，GBFS 使用的是优先队列(Priority Queue)，普通队列是一种先进先出的数据结构，而在优先队列中元素被赋予了优先级，最高优先级元素优先删除，也就是 first in, last out。但是在实际的地图中，常常会有很多障碍物，GBFS 很容易陷入局部最优的陷阱。GBFS 虽然搜索速度够快，但是找不到最优路径。

Dijkstra 算法是从一个顶点到其余各顶点的最短路径算法，其流程仍然与上述算法基本一致，它也是用优先队列作为 openlist 的数据结构，它和 GBFS 的区别在于代价函数的定义，Dijkstra 算法的 f(n)定义为：f(n) = g(n). Dijkstra 算法能够得到最优路径，但是它的速度和 BFS 是一样的，采取的都是稳扎稳打、波状前进的方式，导致速度较慢。

对比 GBFS 和 Dijkstra 算法，两者都采用优先队列作为 openlist，而代价函数的不同导致两者具有不同的优点：GBFS 用节点到目标点的距离作为代价函数，

将搜索方向引向目标点，搜索效率高；而 Dijkstra 算法采用起点到当前扩展节点的移动代价作为代价函数，能够确保路径最优。

A*算法将两者的代价函数进行融合，从而在保证路径最优的同时提高搜索效率。A*算法也是一种启发式算法，它的代价函数表示为：$f(n)=g(n)+h(n)$。根据这个式子，可以得到 A*算法的几个特点：

- 如果令 h(n)=0，A*算法就退化为 Dijkstra 算法；如果令 g(n)=0，A*算法就退化为 GBFS 算法。
- 能否找到最优路径的关键在于启发函数 h(n)的选取，如果 h(n)在大部分情况下比从当前节点到目标点的移动代价小，则能找到最优路径。
- 由于 A* 算法的启发函数是位置上的距离，因此在不带位置信息的图数据中不适用。