

# 人工智能-实验1

计科210X 甘晴void 202108010XXX

## 一、实验目的

- 掌握有信息搜索策略的算法思想；
- 能够编程实现搜索算法；
- 应用A\*搜索算法求解罗马尼亚问题。

## 二、实验平台

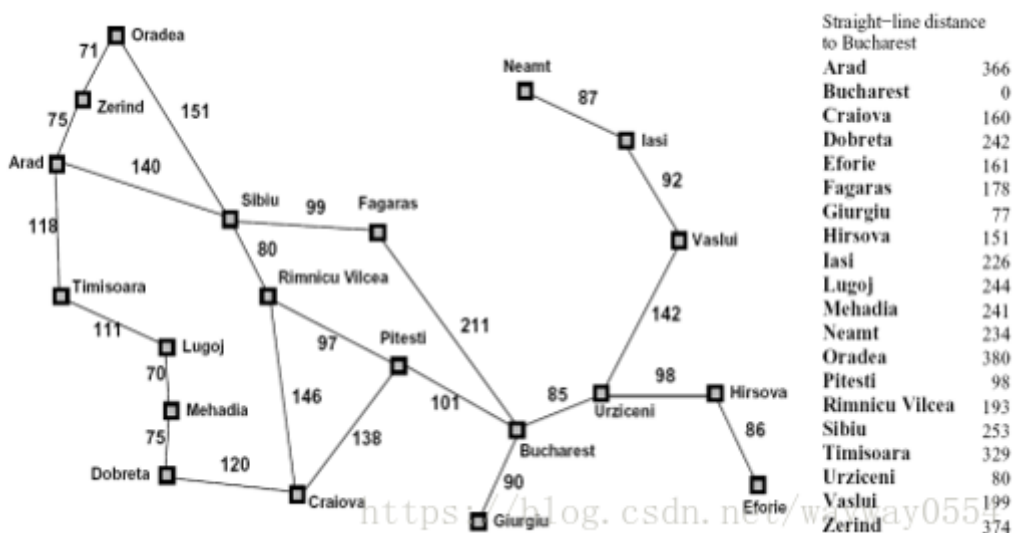
- 课程实训平台<https://www.educoder.net/shixuns/vgmzcukh/challenges>

## 三、实验内容

### 3.0 题目要求

罗马尼亚问题：agent 在罗马尼亚度假，目前位于 Arad 城市。agent 明天有航班从 Bucharest 起飞，不能改签退票。

现在你需要寻找到 Bucharest 的最短路径，在右侧编辑器补充 `void A_star(int goal,node &src,Graph &graph)` 函数，使用编写的搜索算法代码求解罗马尼亚问题：



### 3.1 A\*算法原理

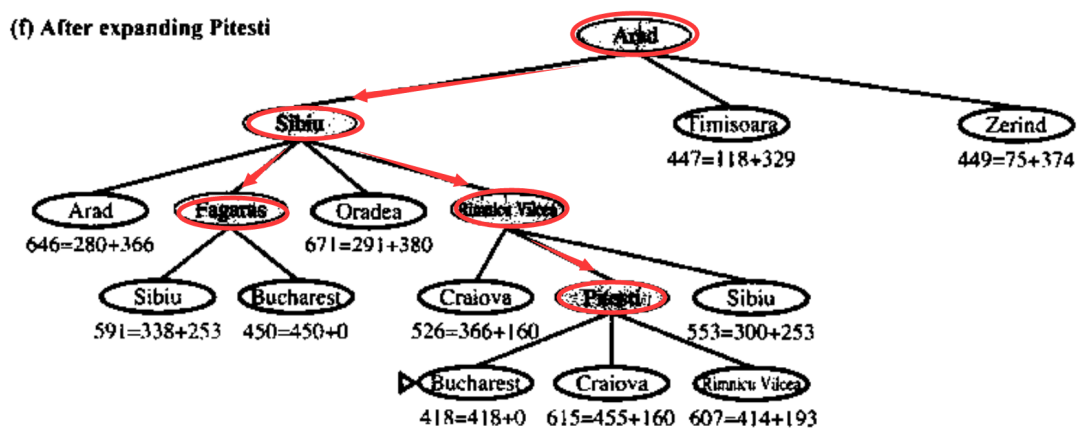
A\*算法的原理是设计一个代价估计函数：其中 评估函数  $F(n)$  是从起始节点通过节点  $n$  的到达目标节点的最小代价路径的估计值，函数  $G(n)$  是从起始节点到  $n$  节点的已走过路径的实际代价，函数  $H(n)$  是从  $n$  节点到目标节点可能的最优路径的估计代价。

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。

### 3.2 算法实现

根据题目要求，实现A\*算法，如下图。



初始给定出发节点，放入openList队列，然后进行扩展操作

- 扩展操作：对于所有与该节点相连的节点，计算它们的 $g()$ 值，然后加上 $h()$ 值，得到 $f()$ 值，并加入openList队列。
- openList队列：该队列本质是一个优先队列，以队列中各元素节点的 $f()$ 值为键进行排序。给定的程序中使用sort+优先队列来实现，实际上这里直接使用优先队列来进行维护会效率更高，这是可以优化的一个点。
- 选定操作：在扩展，优先化（也就是题中的sort）之后，选定一个 $f()$ 值最大的作为下一个访问的节点。
- 访问节点：任何节点的访问操作只能进行一次，但是扩展操作可以进行多次（这其实也可以理解，再次访问该节点显然比原来访问该节点的代价高）
- 终止条件：按照上述循环重复执行，直至访问到（不是扩展到）目标终点为止。

基本思路可以用下述伪代码展示

```
1 void A_star(int goal, node *src, Graph &graph)
```

```

2  {
3      openList.push_back(src);
4      sort(openList.begin(), openList.end(), cmp);
5
6      while (!openList.empty())
7      {
8          node *now = new node;
9          now = openList.front();
10         openList.erase(openList.begin());
11         // 选定操作（从优先的openList中获取第一个元素）
12         if (now->name == goal)
13         {
14             // 终止条件：到达终点，保存退出
15             return;
16         }
17         for (int i = 0; i < 20; i++)
18         {
19             if (graph.getEdge(now->name, i) != -1 &&
20             !visited[i])// 有边且未被访问过
21             {
22                 node *expand = new node(i, now->g +
23                 graph.getEdge(now->name, i), h[i], now);
24                 openList.push_back(expand);
25                 // 执行扩展操作
26             }
27             sort(openList.begin(), openList.end(), cmp);
28             // 保证按照键优先化
29         }
30     }

```

### 3.3 源码&分析

```

1  #include <algorithm>
2  #include <iostream>
3  #include <memory.h>
4  #include <stack>
5  #include <vector>
6  #define A 0
7  #define B 1
8  #define C 2

```

```

9  #define D 3
10 #define E 4
11 #define F 5
12 #define G 6
13 #define H 7
14 #define I 8
15 #define L 9
16 #define M 10
17 #define N 11
18 #define O 12
19 #define P 13
20 #define R 14
21 #define S 15
22 #define T 16
23 #define U 17
24 #define V 18
25 #define Z 19
26
27 using namespace std;
28
29 int h[20] =
30     {366, 0, 160, 242, 161,
31      178, 77, 151, 226, 244,
32      241, 234, 380, 98, 193,
33      253, 329, 80, 199, 374};
34
35 struct node
36 {
37     int g;
38     int h;
39     int f;
40     int name;
41     node(int name, int g, int h)
42     {
43         this->name = name;
44         this->g = g;
45         this->h = h;
46         this->f = g + h;
47     };
48     bool operator<(const node &a) const
49     {
50         return f < a.f;

```

```

51     }
52 };
53
54 class Graph
55 {
56 public:
57     Graph()
58     {
59         memset(graph, -1, sizeof(graph));
60     }
61     int getEdge(int from, int to)
62     {
63         return graph[from][to];
64     }
65     void addEdge(int from, int to, int cost)
66     {
67         if (from >= 20 || from < 0 || to >= 20 || to < 0)
68             return;
69         graph[from][to] = cost;
70     }
71
72     void init()
73     {
74         addEdge(O, Z, 71);
75         addEdge(Z, O, 71);
76
77         addEdge(O, S, 151);
78         addEdge(S, O, 151);
79
80         addEdge(Z, A, 75);
81         addEdge(A, Z, 75);
82
83         addEdge(A, S, 140);
84         addEdge(S, A, 140);
85
86         addEdge(A, T, 118);
87         addEdge(T, A, 118);
88
89         addEdge(T, L, 111);
90         addEdge(L, T, 111);
91
92         addEdge(L, M, 70);

```

```
93      addEdge(M, L, 70);
94
95      addEdge(M, D, 75);
96      addEdge(D, M, 75);
97
98      addEdge(D, C, 120);
99      addEdge(C, D, 120);
100
101      addEdge(C, R, 146);
102      addEdge(R, C, 146);
103
104      addEdge(S, R, 80);
105      addEdge(R, S, 80);
106
107      addEdge(S, F, 99);
108      addEdge(F, S, 99);
109
110      addEdge(F, B, 211);
111      addEdge(B, F, 211);
112
113      addEdge(P, C, 138);
114      addEdge(C, P, 138);
115
116      addEdge(R, P, 97);
117      addEdge(P, R, 97);
118
119      addEdge(P, B, 101);
120      addEdge(B, P, 101);
121
122      addEdge(B, G, 90);
123      addEdge(G, B, 90);
124
125      addEdge(B, U, 85);
126      addEdge(U, B, 85);
127
128      addEdge(U, H, 98);
129      addEdge(H, U, 98);
130
131      addEdge(H, E, 86);
132      addEdge(E, H, 86);
133
134      addEdge(U, V, 142);
```

```

135         addEdge(v, u, 142);
136
137         addEdge(I, v, 92);
138         addEdge(v, I, 92);
139
140         addEdge(I, N, 87);
141         addEdge(N, I, 87);
142     }
143
144 private:
145     int graph[20][20];
146 };
147
148 bool list[20];
149 vector<node> openList;
150 bool closeList[20];
151 stack<int> road;
152 int parent[20];
153
154 void A_star(int goal, node &src, Graph &graph)
155 {
156     openList.push_back(src);
157     sort(openList.begin(), openList.end());
158
159     while (!openList.empty())
160     {
161         /***** Begin *****/
162         node now = openList.front();
163         if (now.name == goal)
164             return;
165         openList.erase(openList.begin());
166         closeList[now.name] = 1;
167         for (int i = 0; i < 20; i++)
168         {
169             if (graph.getEdge(now.name, i) != -1 &&
170                 !closeList[i])
171             {
172                 node expand(i, now.g + graph.getEdge(now.name,
173                     i), h[i]);
174                 openList.push_back(expand);
175                 int flag = true;

```

```

174         for (unsigned int j = 0; j < openList.size();
j++)
175             {
176                 if (openList[j].name == expand.name &&
openList[j].g < expand.g)
177                     {
178                         flag = false;
179                     }
180             }
181             if (flag == true)
182                 parent[i] = now.name;
183         }
184     }
185     sort(openList.begin(), openList.end());
186     /***** End *****/
187 }
188 }
189
190 void print_result(Graph &graph)
191 {
192     int p = openList[0].name;
193     int lastNodeNum;
194     road.push(p);
195     while (parent[p] != -1)
196     {
197         road.push(parent[p]);
198         p = parent[p];
199     }
200     lastNodeNum = road.top();
201     int cost = 0;
202     cout << "solution: ";
203     while (!road.empty())
204     {
205         cout << road.top() << "-> ";
206         if (road.top() != lastNodeNum)
207         {
208             cost += graph.getEdge(lastNodeNum, road.top());
209             lastNodeNum = road.top();
210         }
211         road.pop();
212     }
213     cout << "end" << endl;

```



```

214     cout << "cost:" << cost;
215 }
216
217 int main()
218 {
219     Graph graph;
220     graph.init();
221     for (int i = 0; i < 20; i++)
222         parent[i] = -1;
223     node src(0, 0, h[0]);
224     A_star(1, src, graph);
225     print_result(graph);
226 }
227

```

具体分析如下：

- 结构体定义：结构体 `node` 定义了节点的属性，包括节点名称 `name`，从起点到该节点的路径长度 `g`，该节点到目标节点的估计距离 `h`，以及综合路径长度和估计距离的总代价 `f`。重载了小于操作符，以便在优先队列中进行排序。
- 图的表示：使用二维数组 `graph[20][20]` 表示图，数组大小为 20x20，即有 20 个节点。数组中存储了节点之间的边的权值。
- 初始化图：在 `Graph` 类的 `init()` 方法中，添加了节点之间的边及对应的权值。
- A\*搜索算法：`A_star` 函数实现了A\*搜索算法。它通过优先队列 `openList` 来管理待扩展的节点，并且利用数组 `closeList` 来记录已经访问过的节点。`parent` 数组用于记录每个节点的父节点，方便后续回溯路径。算法首先将起始节点加入到 `openList` 中，并进行排序。然后，循环进行以下步骤：
  - 取出 `openList` 中的首节点 `now`，如果该节点是目标节点，则搜索结束。
  - 将 `now` 加入到 `closeList` 中，表示已经访问过。
  - 遍历与当前节点相邻的节点，如果相邻节点未被访问过，则将其加入到 `openList` 中，并更新其父节点为当前节点，并根据当前节点到该相邻节点的路径长度以及该节点到目标节点的估计距离计算总代价 `f`。
  - 最后对 `openList` 进行排序，以保证优先扩展代价较小的节点。
- 打印结果：`print_result` 函数用于打印搜索结果，即输出找到的最短路径以及路径的总代价。
  - 将起始节点压入栈中。

- 从目标节点开始，通过 `parent` 数组逐步向上回溯，将经过的节点依次压入栈中，直到回溯到起始节点。
- 在压入栈的过程中，同时计算路径的总代价。因为 A\* 算法是一种启发式搜索算法，搜索到的路径并不一定是最优的，但它会在每一步中选择一个启发性最好的节点进行扩展，因此得到的路径一般是较优的。
- 最后，从栈中依次弹出节点，打印出完整的路径，并输出路径的总代价。
- 主函数：在主函数中，首先初始化图，然后调用 `A_star` 函数进行搜索，并最终打印搜索结果。【注意】主函数是在线平台中没有的，在线平台应该指定了程序入口并做了变量初始化的工作。因此我们用主函数来实现这个工作。

综上所述，这段代码实现了使用A\*算法在给定图中寻找从起点到目标点的最短路径，并输出了路径以及路径的总代价。

### 3.4 基于题目的代码：算法分析

时间复杂度可能趋近于 $O(n^3)$ ，主要原因与维护`parent`数组的最新性有关，这个后面在讲想法的时候会具体说明。

空间复杂度应该有 $O(n^2)$ ，因为使用邻接矩阵来存储图。

### 3.5 基于题目的代码：困惑思考

基于题目做这道题的时候，我感觉很困惑。

A\*算法不是一个非常复杂的算法，但题目中的一些操作让我感觉有点迷。

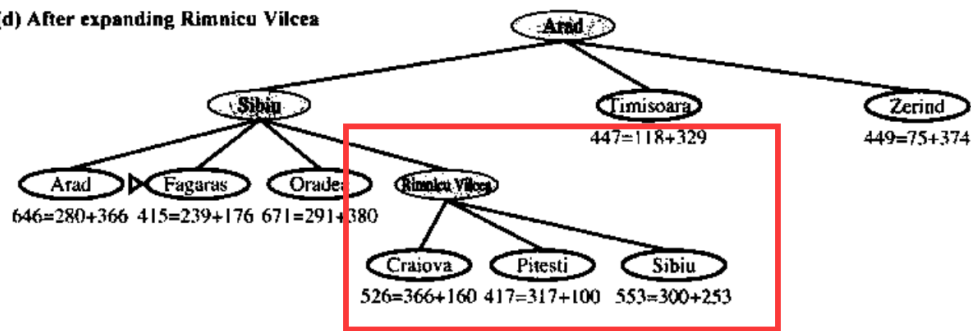
#### ①`parent`数组问题

题目使用`openList`中保存被扩展的节点，然后用`closeList`来标记被访问的节点，用`parent`来存储每个节点的父亲这种方式。

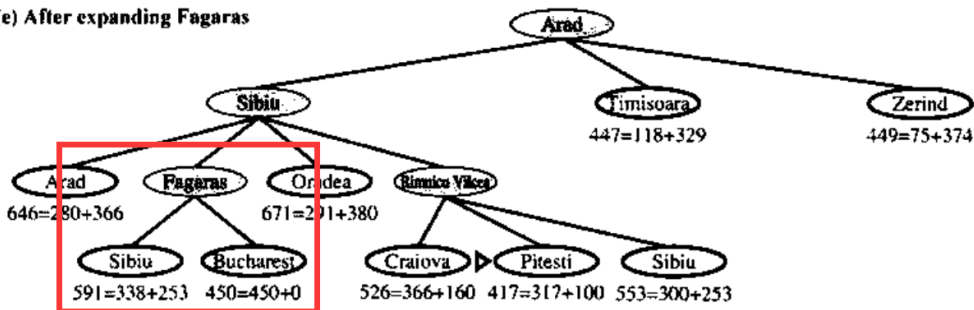
这会带来一个挑战：在一个节点被扩展之后，它不一定被立即访问。

如下面这张图，`Sibiu`节点的四个子节点中，最右子节点`Rimnicu Vilcea`先被访问，但后来`Sibiu`的左起第二个子节点`Fagaras`又被访问到了。此时若该两个子节点都有相同的另一个子节点`M`，则`M`可能同时具有`Rimnicu Vilcea`和`Fagaras`两个父节点。这样用一个`parent`数组显然是没办法表示的（注意这里不能是覆盖关系，因为这两个子节点都是“被扩展”的状态而不是“已被访问”的状态，真正被访问的节点有可能从它们之一产生）

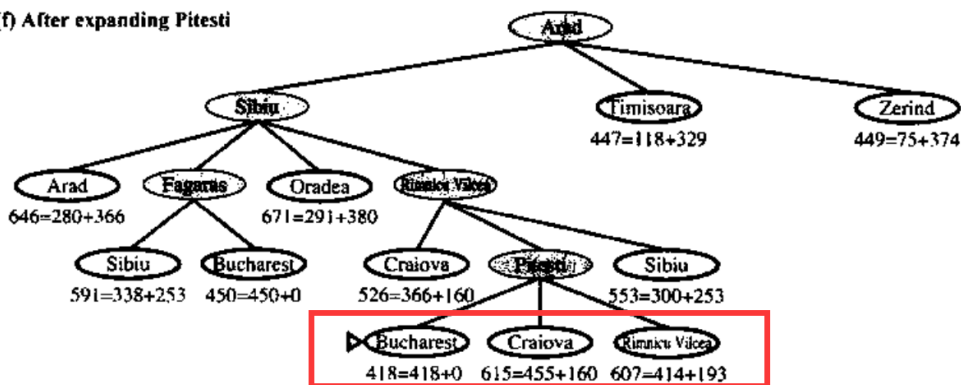
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras

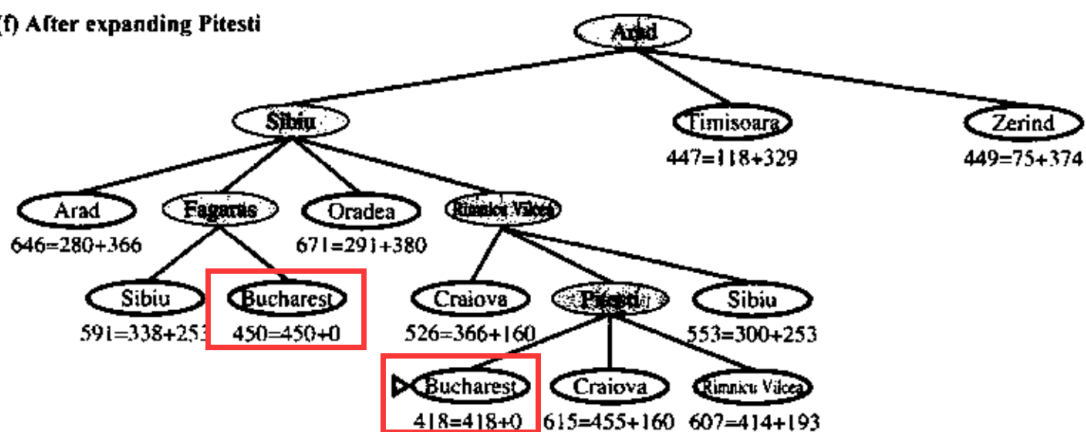


(f) After expanding Pitesti



比如，出现下图所示情况。若Bucharest不是最终节点，则它同时又两个parent，这显然无法用一个parent数组存下。

(f) After expanding Pitesti



因此理想的方法是将parent作为一个属性写入该节点的node结构体中去。但这样还没解决一个问题，输出结果会有点烦。

或者使用后面改进的方法，直接用指针来作为属性。这样只需要指针走一遍，就可以把顺序给呈现出来了。

但是，原题中我也想办法解决了，其实观察或者是从题目中可以发现，一个节点如果发现有比它 $g()$ 值更小的节点，实际上 $g()$ 值较大的那个显然就没有用了，即使予以保留，最终也会在较低优先级而不会被调用（这个实际上看的是 $f()$ 值，事实上是一样的，因为 $f()=g()+h()$ ， $h()$ 显然只与节点有关系）。故我直接略去 $g()$ 值较大的节点，默认它们直接被淘汰掉了，`parent`数组只保存 $g()$ 值最小的那个所对应的。

## ②vector+sort替代priority\_queue

题目中使用了vector来保存访问节点的结构体，再加上sort来保证优先级，其实可以直接用priority\_queue来实现，效率更高。

## 3.6 改进代码-结构体

只展示核心代码，略去重复的宏定义部分和graph类。

使用结构体和指针绕开了parent数组的问题

```
1  #include <algorithm>
2  #include <iostream>
3  #include <memory.h>
4  #include <stack>
5  #include <vector>
6  #define A-Z (略)
7
8  using namespace std;
9
10 int h[20] =
11     {366, 0, 160, 242, 161,
12     178, 77, 151, 226, 244,
13     241, 234, 380, 98, 193,
14     253, 329, 80, 199, 374};
15
16 struct node
17 {
18     int g;
19     int h;
20     int f;
21     int name;
22     node *parent;
23     node() {}
24     node(int name, int g, int h, node *parent)
```

```

25     {
26         this->name = name;
27         this->g = g;
28         this->h = h;
29         this->f = g + h;
30         this->parent = parent;
31     };
32     bool operator<(const node a) const
33     {
34         return f < a.f;
35     }
36 };
37
38 class Graph // (略)
39
40 vector<node *> openList;
41 node *des;
42 bool visited[20];
43
44 bool cmp(node *a, node *b) { return a->f < b->f; }
45 void A_star(int goal, node *src, Graph &graph)
46 {
47     openList.push_back(src);
48     sort(openList.begin(), openList.end(), cmp);
49
50     while (!openList.empty())
51     {
52         node *now = new node;
53         now = openList.front();
54         openList.erase(openList.begin());
55         visited[now->name] = 1;
56         // cout << now->name << endl;
57         // system("pause");
58         if (now->name == goal)
59         {
60             des = now;
61             return;
62         }
63         for (int i = 0; i < 20; i++)
64         {
65             if (graph.getEdge(now->name, i) != -1 &&
!visited[i])

```

```

66         {
67             node *expand = new node(i, now->g +
graph.getEdge(now->name, i), h[i], now);
68             openList.push_back(expand);
69             // cout << "expand: " << expand->name << endl;
70         }
71     }
72     sort(openList.begin(), openList.end(), cmp);
73 }
74 }
75
76 void print_result(Graph &graph)
77 {
78     cout << "solution: ";
79     stack<int> ans;
80     node *now = des;
81     while (now != NULL)
82     {
83         ans.push(now->name);
84         // cout << now->name << endl;
85         now = now->parent;
86     }
87     while (!ans.empty())
88     {
89         cout << ans.top() << "-> ";
90         ans.pop();
91     }
92     cout << "end" << endl;
93     cout << "cost:" << des->g << endl;
94 }
95
96 int main()
97 {
98     Graph graph;
99     graph.init();
100     memset(visited, 0, sizeof(visited));
101     node *src = new node(0, 0, h[0], NULL);
102     A_star(1, src, graph);
103     print_result(graph);
104 }
105

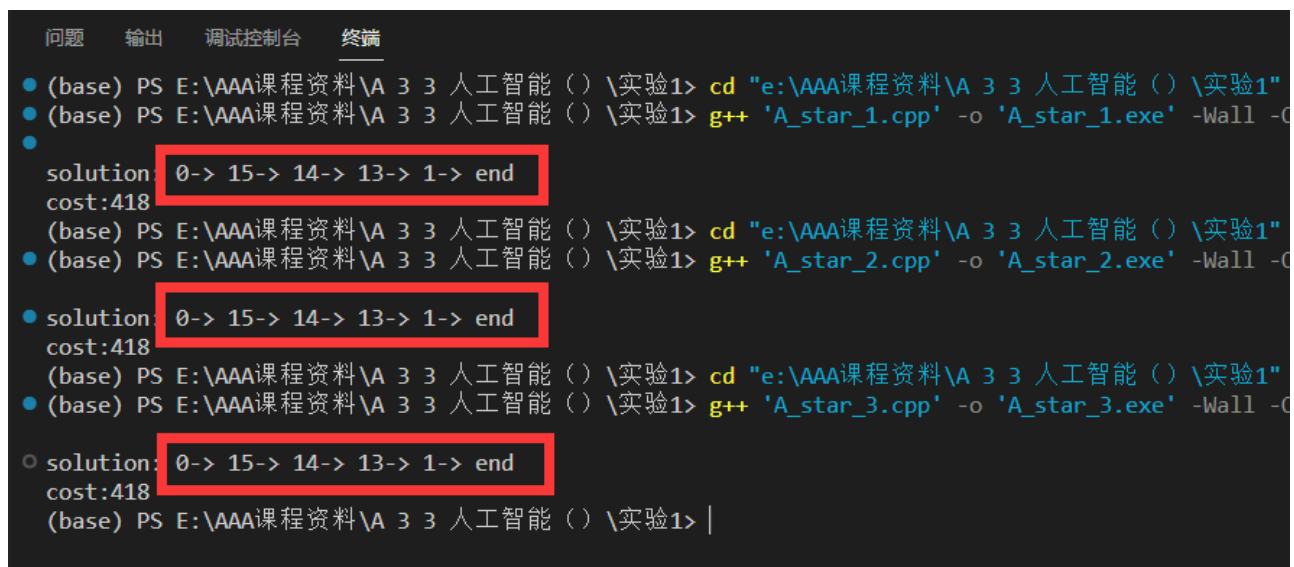
```

### 3.7 改进代码2-<priority\_queue>

只需要在上面代码的基础上更改部分即可，但考虑到这里使用的其实是结构体的指针，故重载运算符其实不太好操作。我采取使用比较函数的方法来完成大小的判断。

```
1  bool cmp(node *a, node *b) { return a->f > b->f; }
2  priority_queue<node *, vector<node *>, decltype(&cmp)>
   openList(&cmp);
3
4  void A_star(int goal, node *src, Graph &graph)
5  {
6      openList.push(src);
7
8      while (!openList.empty())
9      {
10         node *now = new node;
11         now = openList.top();
12         openList.pop();
13         visited[now->name] = 1;
14         // cout << now->name << endl;
15         // system("pause");
16         if (now->name == goal)
17         {
18             des = now;
19             return;
20         }
21         for (int i = 0; i < 20; i++)
22         {
23             if (graph.getEdge(now->name, i) != -1 &&
!visited[i])
24             {
25                 node *expand = new node(i, now->g +
graph.getEdge(now->name, i), h[i], now);
26                 openList.push(expand);
27                 // cout << "expand: " << expand->name << endl;
28             }
29         }
30     }
31 }
32
```

## 3.8 运行截图



```
问题 输出 调试控制台 终端
• (base) PS E:\AAA课程资料\A 3 3 人工智能 () \实验1> cd "e:\AAA课程资料\A 3 3 人工智能 () \实验1"
• (base) PS E:\AAA课程资料\A 3 3 人工智能 () \实验1> g++ 'A_star_1.cpp' -o 'A_star_1.exe' -Wall -C
•
solution 0-> 15-> 14-> 13-> 1-> end
cost:418
(base) PS E:\AAA课程资料\A 3 3 人工智能 () \实验1> cd "e:\AAA课程资料\A 3 3 人工智能 () \实验1"
• (base) PS E:\AAA课程资料\A 3 3 人工智能 () \实验1> g++ 'A_star_2.cpp' -o 'A_star_2.exe' -Wall -C
•
solution 0-> 15-> 14-> 13-> 1-> end
cost:418
(base) PS E:\AAA课程资料\A 3 3 人工智能 () \实验1> cd "e:\AAA课程资料\A 3 3 人工智能 () \实验1"
• (base) PS E:\AAA课程资料\A 3 3 人工智能 () \实验1> g++ 'A_star_3.cpp' -o 'A_star_3.exe' -Wall -C
•
solution: 0-> 15-> 14-> 13-> 1-> end
cost:418
(base) PS E:\AAA课程资料\A 3 3 人工智能 () \实验1> |
```

## 四、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？

首先分析这四种算法，

- 宽度优先搜索（BFS）：通常在最短路径问题上表现优异，但是空间复杂度很高，因为需要保存所有已经访问的节点。
- 深度优先搜索（DFS）：解空间较大，在解相对较浅的问题上可能更有效率，但是可能会陷入无限深度的分支。
- 迭代加深深度优先搜索（IDDFS）：结合了DFS和BFS的优点，在不断增加的深度限制上调用深度受限搜索。对于深度搜索问题而言，是一种比较有效的方法。
- 一致代价搜索（UCS）：保证在图中搜索的每一步都是最小代价的算法，通常在无启发式的情况下用于解决最短路径问题。

对于一般的问题而言，一致代价搜索是更优的。但对于不同问题要具体问题具体分析，如问题的时间或空间限制等。

2: 贪婪最佳优先搜索和A\*搜索哪种方法最优？

首先分析这两种算法，

- 贪婪最佳优先搜索：根据启发式函数 $h()$ 所提供的信息，每次选择看起来最有希望的节点进行扩展，但是它不能保证找到最优解，因为它没有考虑到节点到目标的真实代价。



- **A\*搜索算法**：通过综合考虑节点的实际代价 $g()$ 和启发式函数 $h()$ 的估计值，保证了在每一步都能选择到最优的节点进行扩展，从而保证找到最优解。

A\*搜索算法通常在需要找到最优解的问题上更为优秀，因为它考虑了实际代价。

### 3：分析比较无信息搜索策略和有信息搜索策略。

无信息搜索策略和有信息搜索策略是指搜索算法是否利用额外的信息来指导搜索方向：

- 无信息搜索策略，如深度优先搜索（DFS）、宽度优先搜索（BFS）和一致代价搜索（UCS），只利用当前节点的信息进行搜索，不考虑节点到目标的距离或代价，因此可能需要更多的搜索步骤来找到解。
- 有信息搜索策略，如A\*搜索算法和贪婪最佳优先搜索，利用启发式函数提供的额外信息（如节点到目标的估计距离）来指导搜索方向，从而更快地找到解。有信息搜索策略通常能更快地找到最优解，但是需要在空间和时间上付出更多的代价来计算和存储启发式函数的值。