

# 人工智能-作业1

---

计科210X 甘晴void 202108010XXX

## 第1题

考虑一个实时的在线电话翻译系统，该系统实现英语与日语之间的实时在线翻译，讨论该系统的性能度量，环境，执行器，感知器，并对该环境的属性进行分析。  
(10分)

答：

### 1. 性能度量：

- 实时性：系统需要能够在用户说话之后的几秒内提供翻译结果，延迟应尽可能地小。
- 准确性：翻译结果应尽可能地准确，尽可能贴近在另一种语言的原本含义，避免语义失真或误译。
- 稳定性：系统应该在长时间运行过程中保持稳定，避免崩溃或意外中断，维持好的用户体验。
- 并行性：系统应该能够处理多个用户同时进行的翻译请求，而不降低性能。
- 安全性：由于涉及到用户的语音数据和个人信息，系统需要具有高度的安全性，确保数据不被泄露或被未经授权的人访问。

### 2. 环境：

- 用户动作：用户是否输入语音，以及输入日文/英文语音
- 网络环境：系统需要可靠的网络连接，以确保语音数据能够及时传输。
- 语音环境：用户可能在各种环境中使用系统，包括有噪音的环境或网络质量较差的地方，系统需要能够处理这些情况。

### 3. 执行器：

- （以英语翻译日语为例，日语翻译英语同理）
- 语音识别模块：将用户说的英语语音转换为英语文本。
- 翻译引擎：将英语文本翻译成日语文本。
- 语音合成模块：将日语文本转换为日语语音。

### 4. 感知器：

- 语音输入：接收用户说的英语语音。
- 文本输入：接收语音识别模块输出的英语文本。
- 文本输出：接收翻译引擎输出的日语文本。

- 语音输出：接收语音合成模块输出的日语语音。
- （以上为包括内部模块的，如果只考虑将这个系统作为一个黑盒，则只有对外的一个麦克风，用于接收用户输入）

#### 5. 环境属性分析：

- 完全可观察：假设传感器运作都正常，则环境（用户输入的语音、网络环境等）是完全可观察的。
- 单**Agent**：显然只有一个智能体，即翻译器。
- 随机的：环境的下一状态不取决于**Agent**执行的动作。
- 静态的：**Agent**计算时环境不会变化，这里指已经读入的语音信息不会随着翻译的进行而随时变化，在翻译时是静态的。但若是同声传译则有可能随读入而发生变化，这个暂不考虑。
- 连续的：读入的语音是连续的，但在解析时会转化为离散的向量（这里书上也标注过不好区分）。
- 已知的：假定处理翻译的模型是一个预训练好的模型，那么处理翻译的规则是给定的，不会随读入而发生变化。

## 第2题

考虑一个医疗诊断系统的`agent`，讨论该`agent`最合适的种类(简单`agent`, 基于模型的`agent`, 基于目标的`agent`和基于效用的`agent`)并解释你的结论。（10分）

答：

#### 1. 简单反射**Agent**：

简单反射**Agent**是一种基本的反应式**Agent**，它只根据当前的输入执行特定的操作，而不考虑过去或未来的状态。简单**Agent**适用于一些简单的诊断任务，基于单一指标或症状的诊断，例如测量体温并根据特定的阈值判断是否发烧。但对于复杂的疾病诊断，简单**Agent**不够灵活和智能。

#### 2. 基于模型的反射**Agent**：

基于模型的反射**Agent**通过对环境的建模来进行决策，它能够考虑到环境中的动态变化以及行为的长期影响。基于模型的**Agent**可以利用医学知识库和患者历史数据来建立模型，以更准确地诊断疾病。它还可以学习医学知识和历史病例来不断改进自己的模型，从而提高诊断的准确性和效率。

#### 3. 基于目标的**Agent**：

基于目标的**Agent**会考虑到目标的重要性和实现目标的各种可能方法，并选择最佳的行动方案以达到预期的目标。我们假设目标是尽快准确地诊断疾病，以便采取适当的治疗措施。这种**Agent**会根据病情的严重程度和紧急性来优先考虑诊断某些疾病，从而提高治疗的及时性和有效性。

#### 4. 基于效用的Agent:

基于效用的Agent会评估不同行动的效用，并选择对整体效用最大化的行动。这里效用可能包括诊断准确性、治疗成功率、患者生存率等因素。这种Agent可以帮助医生在面临多种诊断选择时做出理性的决策，以最大程度地提高整体的医疗效果。

我认为基于效用的Agent会更好。因为不论治疗措施如何，治疗方式如何，对于病人来说，最终的治疗效果才是检验治疗的唯一标准。简单反射显然不满足医疗这种复杂情况，基于目标和可能无法很好定义目标函数从而降低效果，故我认为可能基于效用的Agent会更好。

### 第3题

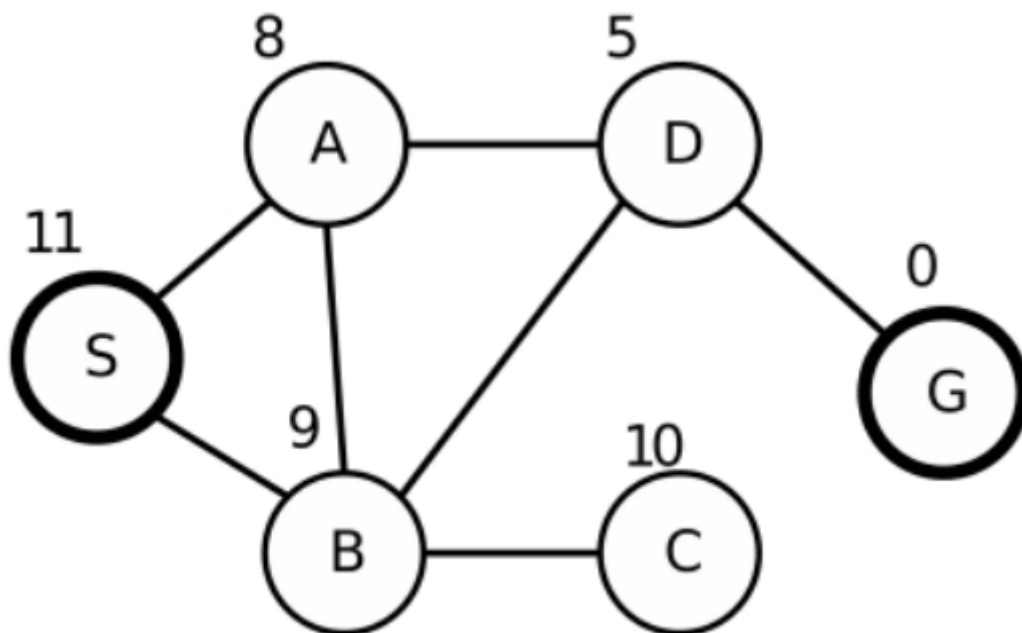
先建立一个完整的搜索树，起点是S,终点是G,如下图,节点旁的数字表示到达目标状态的距离，然后用以下方法表示如何进行搜索，并分析这几种算法的完备性、最优性、以及时间复杂度和空间复杂度。（40分）

(a).深度优先;

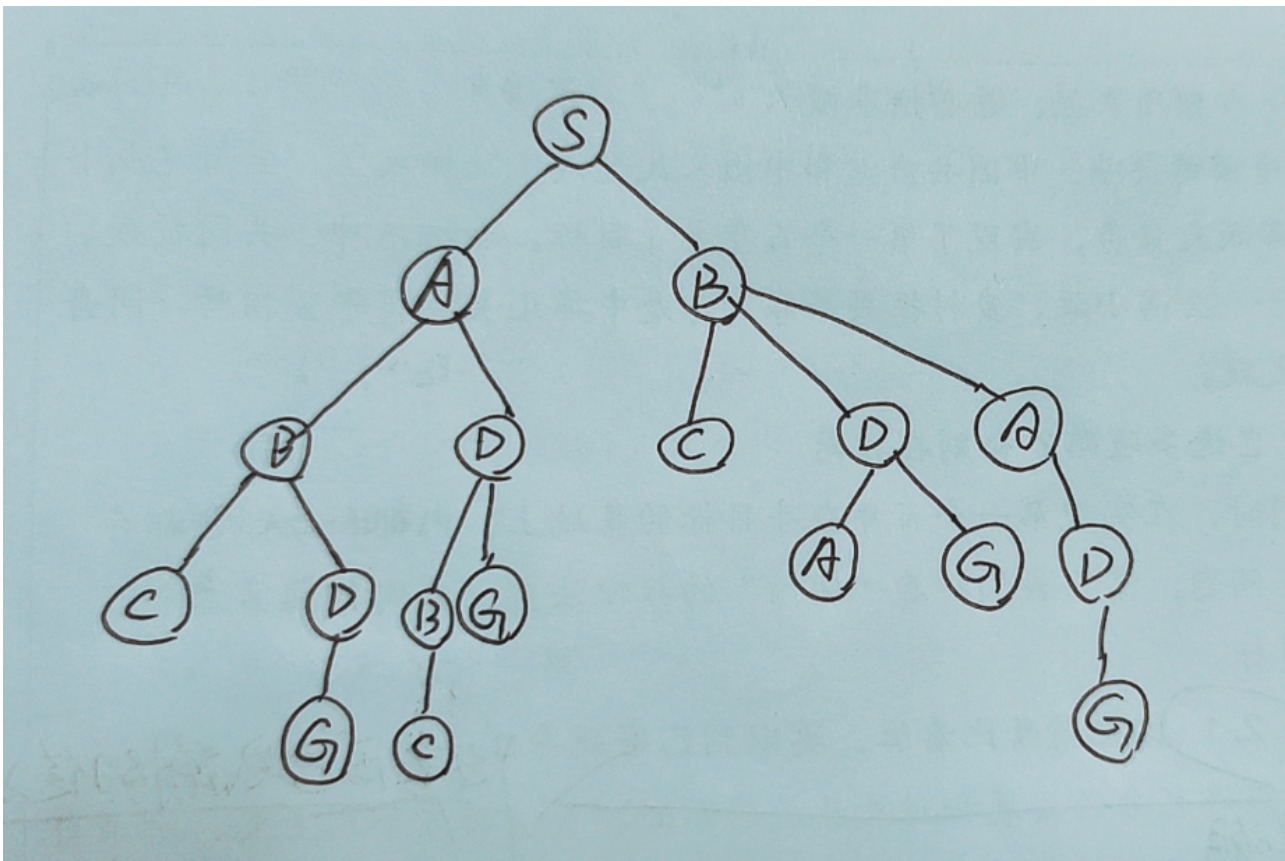
(b).宽度优先;

(c).爬山法;

(d).贪婪最佳优先。



解：先给出搜索树，再分析问题，再给出几种算法的完备性、最优性、以及时间复杂度和空间复杂度。



### (1) 深度优先

假设在扩展节点时按照序号升序选择，若无可扩展节点就回溯。

S -> A -> B -> C(无可扩展节点，回溯回B) -> D -> G

核心代码如下：

```

1  bool dfs_visited[N] = {0};
2  void dfs(int goal, node &src, Graph &graph)
3  {
4      if (src.name == goal)
5      {
6          cout << src.cname << endl;
7          return;
8      }
9      dfs_visited[src.name] = 1;
10     cout << src.cname << " -> ";
11     for (int i = 0; i < N; i++)
12     {
13         if (graph.getEdge(src.name, i) == 1 && !dfs_visited[i])
14         {
15             node des(i);
16             dfs(goal, des, graph);

```

```
17     }
18 }
19 }
```

## (2) 宽度优先

假设在扩展节点时按照序号升序选择。

S -> A/B, A -> D, B -> C, C无可扩展节点, D -> G

访问顺序应该是S -> A -> B -> D -> C -> G

核心代码如下:

```
1 void bfs(int goal, node &src, Graph &graph)
2 {
3     bool bfs_visited[N] = {0};
4     queue<node> q;
5     q.push(src);
6     while (!q.empty())
7     {
8         node src = q.front();
9         q.pop();
10        bfs_visited[src.name] = 1;
11        if (src.name == goal)
12        {
13            cout << src.cname << endl;
14            break;
15        }
16        cout << src.cname << " -> ";
17        for (int i = 0; i < N; i++)
18        {
19            if (graph.getEdge(src.name, i) == 1 &&
20                !bfs_visited[i])
21            {
22                node des(i);
23                bfs_visited[i] = 1;
24                q.push(des);
25                // cout << "extend" << i << endl;
26            }
27        }
28    }
29 }
```

```

27     }
28     return;
29 }

```

### (3) 爬山法

爬山法总是选择邻居状态中最好的（该问题下是值最小的）节点

访问顺序S -> A -> D -> G

核心代码如下：

```

1  bool cm_visited[N] = {0};
2  const int MAXNUM = 99999;
3  void climb_mountain(int goal, node &src, Graph &graph)
4  {
5      if (src.name == goal)
6      {
7          cout << src.cname << endl;
8          return;
9      }
10     cm_visited[src.name] = 1;
11     cout << src.cname << " -> ";
12     int h_best = MAXNUM;
13     int next_visit;
14     for (int i = 0; i < N; i++)
15     {
16         if (graph.getEdge(src.name, i) == 1 && !cm_visited[i])
17         {
18             if (h[i] < h_best)
19             {
20                 h_best = h[i];
21                 next_visit = i;
22             }
23         }
24     }
25     node des(next_visit, h_best);
26     climb_mountain(goal, des, graph);
27 }

```

### (4) 贪婪最佳优先

贪婪最佳优先算法总是选择 $h(n)$ 最小的节点作为扩展节点，本题与爬山法类似。

访问顺序S -> A -> D -> G

核心代码如下：

```
1  bool greedy_visited[N] = {0};
2  void greedy_best_first(int goal, node &src, Graph &graph)
3  {
4      if (src.name == goal)
5      {
6          cout << src.cname << endl;
7          return;
8      }
9      greedy_visited[src.name] = 1;
10     cout << src.cname << " -> ";
11     int h_best = MAXNUM;
12     int next_visit = 0;
13     for (int i = 0; i < N; i++)
14     {
15         if (graph.getEdge(src.name, i) == 1 &&
16             !greedy_visited[i])
17         {
18             if (h[i] < h_best)
19             {
20                 h_best = h[i];
21                 next_visit = i;
22             }
23         }
24     }
25     node des(next_visit, h_best);
26     greedy_best_first(goal, des, graph);
27 }
```

(5) 代码验证



```
C++ 作业1-dfs_bfs_climb_greedy.cpp X
E: > AAA课程资料 > A 3 3 人工智能 ( ) > 作业代码 > C++ 作业1-dfs_bfs_climb_greedy.cpp > C

1  #include <algorithm>
2  #include <iostream>
3  #include <memory.h>
4  #include <queue>
5  #include <stack>
6  #include <vector>
7  #define N 6
8  #define S 0
9  #define A 1
10 #define B 2
11 #define C 3
12 #define D 4
13 #define G 5
14

问题 3 输出 调试控制台 终端

• (base) PS C:\Users\y> cd "e:\AAA课程资料\A 3 3 人工智能 ( ) \作业代码"
• (base) PS E:\AAA课程资料\A 3 3 人工智能 ( ) \作业代码> g++ '作业1-dfs_bfs_climb_greedy.cpp' -o
  tatic-libgcc -fexec-charset=GBK ; if ($?) { &'./作业1-dfs_bfs_climb_greedy.exe' }
• dfs: S -> A -> B -> C -> D -> G
  bfs: S -> A -> B -> D -> C -> G
  climb mountain: S -> A -> D -> G
  greedy best first: S -> A -> D -> G
(base) PS E:\AAA课程资料\A 3 3 人工智能 ( ) \作业代码> |
```

## (6) 完备性、最优性、以及时间复杂度和空间复杂度

### ①深度优先:

- 完备性: 有限深度的深度优先搜索有完备性, 无限深度的无完备性
- 最优性: 不具有最优性, 因为在选择分支中可能错过最优解
- 时间复杂度:  $O(b^n)$ ,  $b$ 为分支因子,  $n$ 为最大深度
- 空间复杂度:  $O(bn)$ ,  $b$ 为分支因子,  $n$ 为最大深度

### ②宽度优先:

- 完备性: 有完备性
- 最优性: 在单位代价情况下具有最优性
- 时间复杂度:  $O(b^n)$ ,  $b$ 为分支因子,  $n$ 为最大深度
- 空间复杂度:  $O(b^n)$ ,  $b$ 为分支因子,  $n$ 为最大深度

### ③爬山法:



- 完备性：无完备性
- 最优性：不具备最优性。因为可能陷入局部最优解，而不是全局的
- 时间复杂度： $O(b^n)$ ， $b$ 为分支因子， $n$ 为最大深度
- 空间复杂度： $O(bn)$ ， $b$ 为分支因子， $n$ 为最大深度

#### ④贪婪最佳优先。

- 完备性：不具有完备性，有可能在评估函数值较小的节点终止（死胡同）
- 最优性：不具有最优性，有可能因为不断追求较小的评估值反而走了更远的路
- 时间复杂度： $O(b^n)$ ， $b$ 为分支因子， $n$ 为最大深度
- 空间复杂度： $O(b^n)$ ， $b$ 为分支因子， $n$ 为最大深度

#### (7) 完整代码

```

1  #include <algorithm>
2  #include <iostream>
3  #include <memory.h>
4  #include <queue>
5  #include <stack>
6  #include <vector>
7  #define N 6
8  #define S 0
9  #define A 1
10 #define B 2
11 #define C 3
12 #define D 4
13 #define G 5
14
15 using namespace std;
16
17 int h[20] = {11, 8, 9, 10, 5, 0};
18
19 struct node
20 {
21     int name;
22     char cname;
23     int h;
24     node(int name, int h)
25     {
26         this->name = name;
27         this->h = h;
28         switch (name)

```

```

29         {
30             case 0:
31             {
32                 cname = 'S';
33                 break;
34             }
35             case 1:
36             {
37                 cname = 'A';
38                 break;
39             }
40             case 2:
41             {
42                 cname = 'B';
43                 break;
44             }
45             case 3:
46             {
47                 cname = 'C';
48                 break;
49             }
50             case 4:
51             {
52                 cname = 'D';
53                 break;
54             }
55             case 5:
56             {
57                 cname = 'G';
58                 break;
59             }
60         }
61     };
62 };
63
64 class Graph
65 {
66 public:
67     Graph()
68     {
69         memset(graph, -1, sizeof(graph));
70     }

```

```

71     int getEdge(int from, int to)
72     {
73         return graph[from][to];
74     }
75     void addEdge(int from, int to, int cost)
76     {
77         if (from >= N || from < 0 || to >= N || to < 0)
78             return;
79         graph[from][to] = cost;
80     }
81
82     void init()
83     {
84         addEdge(S, A, 1);
85         addEdge(A, S, 1);
86         addEdge(S, B, 1);
87         addEdge(B, S, 1);
88         addEdge(A, B, 1);
89         addEdge(B, A, 1);
90         addEdge(A, D, 1);
91         addEdge(D, A, 1);
92         addEdge(B, D, 1);
93         addEdge(D, B, 1);
94         addEdge(D, G, 1);
95         addEdge(G, D, 1);
96         addEdge(B, C, 1);
97         addEdge(C, B, 1);
98     }
99
100 private:
101     int graph[N][N];
102 };
103
104 bool dfs_visited[N] = {0};
105 void dfs(int goal, node &src, Graph &graph)
106 {
107     if (src.name == goal)
108     {
109         cout << src.cname << endl;
110         return;
111     }
112     dfs_visited[src.name] = 1;

```

```

113     cout << src.cname << " -> ";
114     for (int i = 0; i < N; i++)
115     {
116         if (graph.getEdge(src.name, i) == 1 && !dfs_visited[i])
117         {
118             node des(i, 0);
119             dfs(goal, des, graph);
120         }
121     }
122 }
123
124 void bfs(int goal, node &src, Graph &graph)
125 {
126     bool bfs_visited[N] = {0};
127     queue<node> q;
128     q.push(src);
129     while (!q.empty())
130     {
131         node src = q.front();
132         q.pop();
133         bfs_visited[src.name] = 1;
134         if (src.name == goal)
135         {
136             cout << src.cname << endl;
137             break;
138         }
139         cout << src.cname << " -> ";
140         for (int i = 0; i < N; i++)
141         {
142             if (graph.getEdge(src.name, i) == 1 &&
143             !bfs_visited[i])
144             {
145                 node des(i, 0);
146                 bfs_visited[i] = 1;
147                 q.push(des);
148                 // cout << "extend" << i << endl;
149             }
150         }
151     }
152     return;
153 }

```

```

154 bool cm_visited[N] = {0};
155 const int MAXNUM = 99999;
156 void climb_mountain(int goal, node &src, Graph &graph)
157 {
158     if (src.name == goal)
159     {
160         cout << src.cname << endl;
161         return;
162     }
163     cm_visited[src.name] = 1;
164     cout << src.cname << " -> ";
165     int h_best = MAXNUM;
166     int next_visit = 0;
167     for (int i = 0; i < N; i++)
168     {
169         if (graph.getEdge(src.name, i) == 1 && !cm_visited[i])
170         {
171             if (h[i] < h_best)
172             {
173                 h_best = h[i];
174                 next_visit = i;
175             }
176         }
177     }
178     node des(next_visit, h_best);
179     climb_mountain(goal, des, graph);
180 }
181
182 bool greedy_visited[N] = {0};
183 void greedy_best_first(int goal, node &src, Graph &graph)
184 {
185     if (src.name == goal)
186     {
187         cout << src.cname << endl;
188         return;
189     }
190     greedy_visited[src.name] = 1;
191     cout << src.cname << " -> ";
192     int h_best = MAXNUM;
193     int next_visit = 0;
194     for (int i = 0; i < N; i++)
195     {

```

```

196         if (graph.getEdge(src.name, i) == 1 &&
!greedy_visited[i])
197         {
198             if (h[i] < h_best)
199             {
200                 h_best = h[i];
201                 next_visit = i;
202             }
203         }
204     }
205     node des(next_visit, h_best);
206     greedy_best_first(goal, des, graph);
207 }
208
209 int main()
210 {
211     Graph graph;
212     graph.init();
213     node src(S, h[S]);
214     cout << "dfs: ";
215     dfs(G, src, graph);
216     cout << "bfs: ";
217     bfs(G, src, graph);
218     cout << "climb mountain: ";
219     climb_mountain(G, src, graph);
220     cout << "greedy_best_first: ";
221     greedy_best_first(G, src, graph);
222 }
223

```

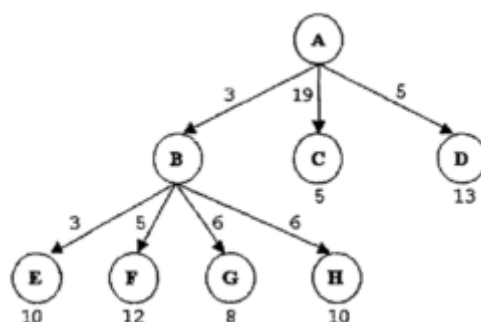
## 第4题

图二是一棵部分展开的搜索树，其中树的边记录了对应的单步代价，叶子节点标注了到达目标结点的启发式函数的代价值，假定当前状态位于结点A。用下列的搜索方法来计算下一步需要展开的叶子节点。注意必须要有完整的计算过程，同时必须对扩展该叶子节点之前的节点顺序进行记录：（20分）

- （1）贪婪最佳优先搜索
- （2）一致代价搜索

### (3) A\*树搜索

讨论以上三种算法的完备性和最优性。



解：先分析问题，再给出完备性和最优性界定。

#### (1) 贪婪最佳优先搜索

贪婪最佳优先搜索总是选择 $h()$ 值最小的节点进行扩展。

由于图中信息不足，故需要分类讨论，对于B处的 $h(B)$ 值进行讨论。

- ①若 $h(B) > 5$ ，则A  $\rightarrow$  C，到达叶子节点，搜索结束。答案为A  $\rightarrow$  C
- ②若 $h(B) \leq 5$ ，则A  $\rightarrow$  B。此时 $h(G)$ 最小，故选择G节点作为扩展节点。答案为A  $\rightarrow$  B  $\rightarrow$  G

#### (2) 一致代价搜索

一致代价搜索选择使得 $g()$ 最小的节点进行扩展。

由于 $g(B)=3$ 最小，故A  $\rightarrow$  B。又 $g(E)=6$ 最小，故B  $\rightarrow$  E，到达叶子节点，搜索结束。答案为A  $\rightarrow$  B  $\rightarrow$  E。

### (3) A\*树搜索

A\*树搜索结合贪婪最佳优先搜索和一致代价搜索的特点，考虑 $f(X)=g(X)+h(X)$ ，选择使得评估函数 $f(X)$ 最小的节点进行扩展。

由于图中信息不足，故需要分类讨论，仍然需要对于B处的 $h(B)$ 值进行讨论。

- $f(B)=3+h(B)$
- $f(C)=19+5=24$
- $f(D)=5+13=18$

①若 $h(B) \leq 15$ ，则 $f(B) \leq f(D)$ ，此时选择B作为扩展节点。



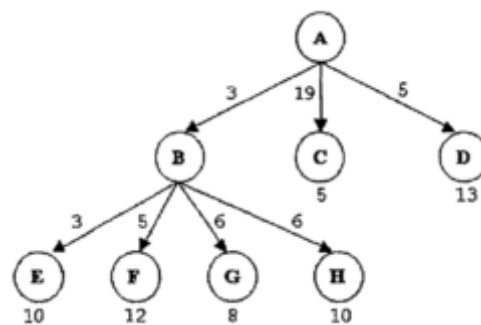
- $f(E)=6+10=16$
- $f(F)=8+12=20$
- $f(G)=9+8=17$
- $f(H)=9+10=19$

此时 $f(E)$ 最小，故选择E。

最终答案为A -> B -> E。

②若 $h(B)>15$ ，则 $f(B)>f(D)$ ，此时选择D作为扩展节点。

最终答案为A -> D。



#### (4) 完备性与最优性

- 贪婪最佳优先搜索：不具有完备性，不具有最优性。
- 一致代价搜索：有完备性，有最优性
- A\*树搜索：有完备性，有最优性

## 第5题

给定一个启发式函数满足 $h(G)=0$ ,其中G是目标状态，证明如果 $h$ 是一致的，那么它是可采纳的。（20分）

解：

可采纳性与一致性定义：

- 启发函数 $h(n)$ 是可采纳的条件：对于任意结点 $n$ ,  $h(n) \leq h^*(n)$ ，其中 $h^*(n)$ 是到达目标结点的真实代价
- 启发函数 $h(n)$ 是一致的条件：对于任意结点 $n$ ，以及 $n$ 的行为 $a$ 产生的后继结点 $n'$ ,满足如下公式： $h(n) \leq c(n,a,n') + h(n')$

证明如下：

假设 $n$ 为任意状态,  $G$ 为某目标状态, 且从状态 $n$ 到状态 $G$ 的一条最优路径为 $n, n_1, n_2, \dots, n_m$

根据一致性条件,  $h(n) \leq c(n, a_1, n_1) + h(n_1)$

$\leq c(n, a_1, n_1) + c(n_1, a_2, n_2) + h(n_2)$

$\leq c(n, a_1, n_1) + c(n_1, a_2, n_2) + \dots + c(n_m, a_{m+1}, G) + h(G)$

又  $h(G) = 0$ , 故  $h(n) \leq c(n, a_1, n_1) + c(n_1, a_2, n_2) + \dots + c(n_m, a_{m+1}, G)$

根据实际意义,  $h^*(n) = c(n, a_1, n_1) + c(n_1, a_2, n_2) + \dots + c(n_m, a_{m+1}, G)$ , 因为这是从 $n$ 到 $G$ 的实际距离。

综上,  $h(n) \leq h^*(n)$