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。

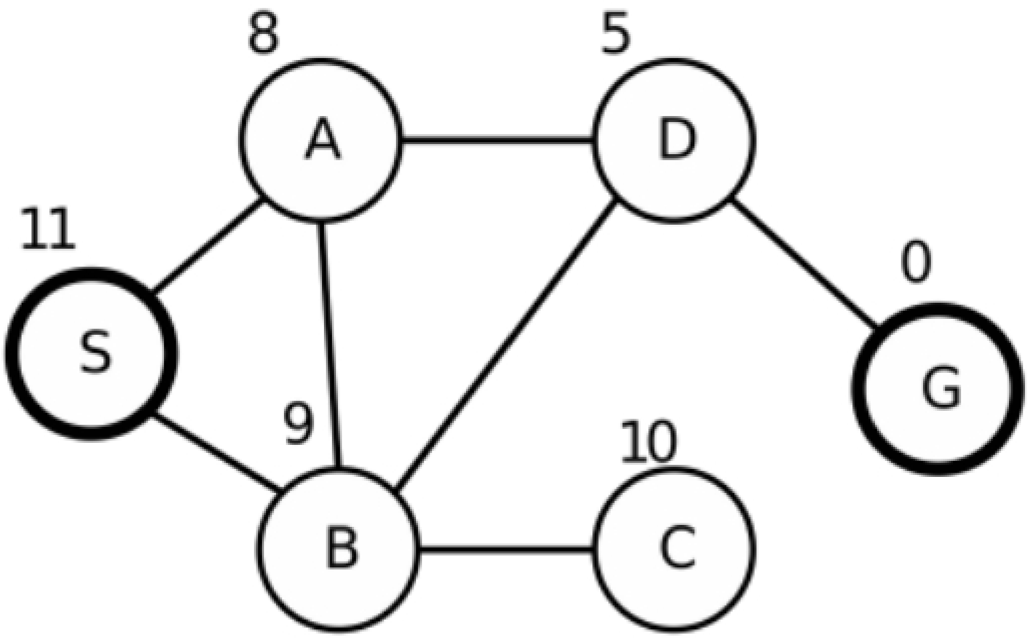
4. 基于效用的**Agent**：

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。

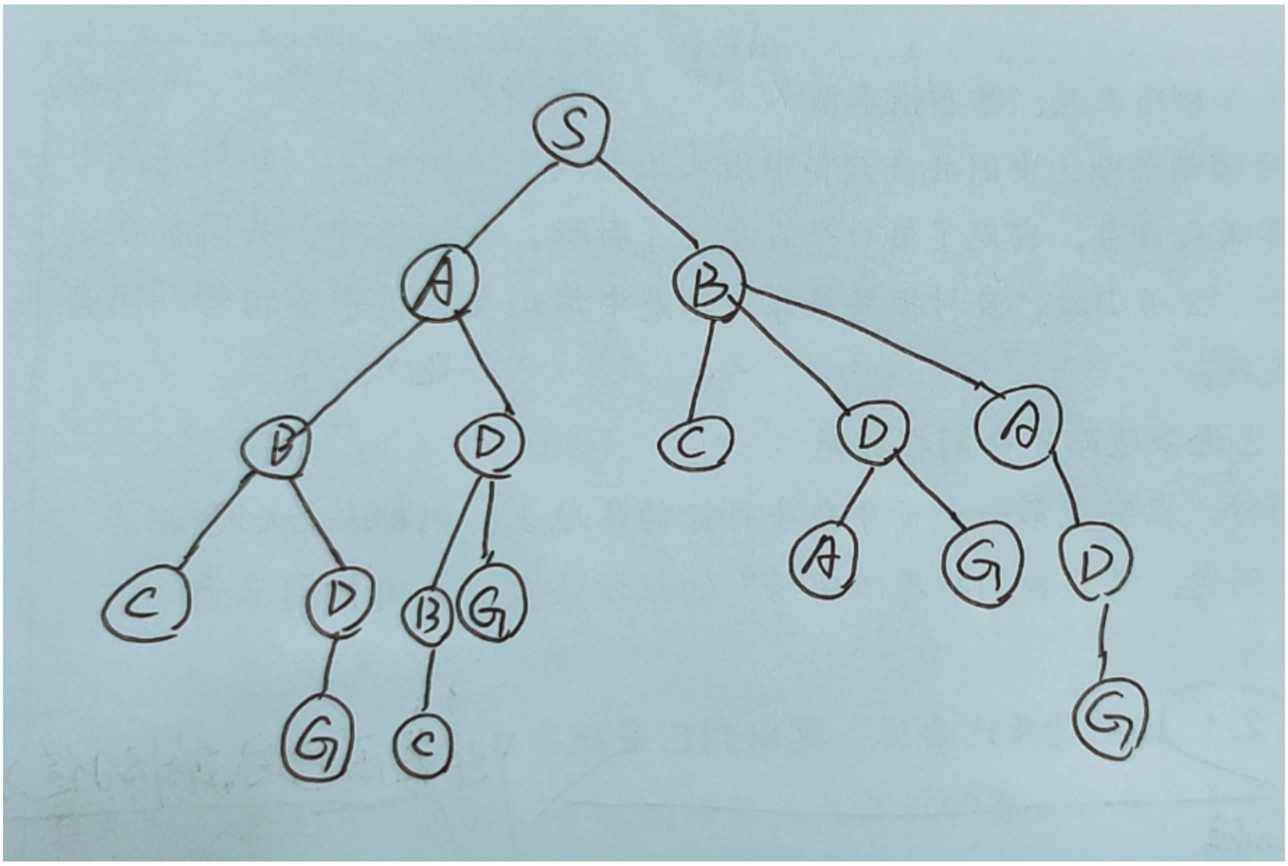
我认为基于效用的Agent会更好。因为不论治疗措施如何，治疗方式如何，对于病人来说，最终的治疗效果才是检验治疗的唯一标准。简单反射显然不满足医疗这种复杂情况，基于目标和可能无法很好定义目标函数从而降低效果，故我认为可能基于效用的Agent会更好。

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：

```
bool dfs_visited[N] = {0};
void dfs(int goal, node &src, Graph &graph)
{
    if (src.name == goal)
    {
        cout << src.cname << endl;
        return;
    }
    dfs_visited[src.name] = 1;
    cout << src.cname << " -> ";
    for (int i = 0; i < N; i++)
    {
        if (graph.getEdge(src.name, i) == 1 && !dfs_visited[i])
        {
            node des(i);
            dfs(goal, des, graph);
```

```
        }
    }
}
```

（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：

```
void bfs(int goal, node &src, Graph &graph)
{
    bool bfs_visited[N] = {0};
    queue<node> q;
    q.push(src);
    while (!q.empty())
    {
        node src = q.front();
        q.pop();
        bfs_visited[src.name] = 1;
        if (src.name == goal)
        {
            cout << src.cname << endl;
            break;
        }
        cout << src.cname << " -> ";
        for (int i = 0; i < N; i++)
        {
            if (graph.getEdge(src.name, i) == 1 &&
!bfs_visited[i])
            {
                node des(i);
                bfs_visited[i] = 1;
                q.push(des);
                // cout << "extend" << i << endl;
            }
        }
```

```
    }
    return;
}
```

（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：

```
bool cm_visited[N] = {0};
const int MAXNUM = 99999;
void climb_mountain(int goal, node &src, Graph &graph)
{
    if (src.name == goal)
    {
        cout << src.cname << endl;
        return;
    }
    cm_visited[src.name] = 1;
    cout << src.cname << " -> ";
    int h_best = MAXNUM;
    int next_visit;
    for (int i = 0; i < N; i++)
    {
        if (graph.getEdge(src.name, i) == 1 && !cm_visited[i])
        {
            if (h[i] < h_best)
            {
                h_best = h[i];
                next_visit = i;
            }
        }
    }
    node des(next_visit, h_best);
    climb_mountain(goal, des, graph);
}
```

（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：

```
bool greedy_visited[N] = {0};
void greedy_best_first(int goal, node &src, Graph &graph)
{
    if (src.name == goal)
    {
        cout << src.cname << endl;
        return;
    }
    greedy_visited[src.name] = 1;
    cout << src.cname << " -> ";
    int h_best = MAXNUM;
    int next_visit = 0;
    for (int i = 0; i < N; i++)
    {
        if (graph.getEdge(src.name, i) == 1 && !greedy_visited[i])
        {
            if (h[i] < h_best)
            {
                h_best = h[i];
                next_visit = i;
            }
        }
    }
    node des(next_visit, h_best);
    greedy_best_first(goal, des, graph);
}
```

（5）代码验证

作业1-dfs_bfs_climb_greedy.cpp ×

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

```cpp
#include <algorithm>
#include <iostream>
#include <memory.h>
#include <queue>
#include <stack>
#include <vector>
#define N 6
#define S 0
#define A 1
#define B 2
#define C 3
#define D 4
#define G 5

```

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

```
#include <algorithm>
#include <iostream>
#include <memory.h>
#include <queue>
#include <stack>
#include <vector>
#define N 6
#define S 0
#define A 1
#define B 2
#define C 3
#define D 4
#define G 5

using namespace std;

int h[20] = {11, 8, 9, 10, 5, 0};

struct node
{
    int name;
    char cname;
    int h;
    node(int name, int h)
    {
        this->name = name;
        this->h = h;
        switch (name)
```

```
        {
        case 0:
        {
            cname = 'S';
            break;
        }
        case 1:
        {
            cname = 'A';
            break;
        }
        case 2:
        {
            cname = 'B';
            break;
        }
        case 3:
        {
            cname = 'C';
            break;
        }
        case 4:
        {
            cname = 'D';
            break;
        }
        case 5:
        {
            cname = 'G';
            break;
        }
        }
    };
};

class Graph
{
public:
    Graph()
    {
        memset(graph, -1, sizeof(graph));
    }
```

```
    int getEdge(int from, int to)
    {
        return graph[from][to];
    }
    void addEdge(int from, int to, int cost)
    {
        if (from >= N || from < 0 || to >= N || to < 0)
            return;
        graph[from][to] = cost;
    }

    void init()
    {
        addEdge(S, A, 1);
        addEdge(A, S, 1);
        addEdge(S, B, 1);
        addEdge(B, S, 1);
        addEdge(A, B, 1);
        addEdge(B, A, 1);
        addEdge(A, D, 1);
        addEdge(D, A, 1);
        addEdge(B, D, 1);
        addEdge(D, B, 1);
        addEdge(D, G, 1);
        addEdge(G, D, 1);
        addEdge(B, C, 1);
        addEdge(C, B, 1);
    }

private:
    int graph[N][N];
};

bool dfs_visited[N] = {0};
void dfs(int goal, node &src, Graph &graph)
{
    if (src.name == goal)
    {
        cout << src.cname << endl;
        return;
    }
    dfs_visited[src.name] = 1;
```

```
    cout << src.cname << " -> ";
    for (int i = 0; i < N; i++)
    {
        if (graph.getEdge(src.name, i) == 1 && !dfs_visited[i])
        {
            node des(i, 0);
            dfs(goal, des, graph);
        }
    }
}

void bfs(int goal, node &src, Graph &graph)
{
    bool bfs_visited[N] = {0};
    queue<node> q;
    q.push(src);
    while (!q.empty())
    {
        node src = q.front();
        q.pop();
        bfs_visited[src.name] = 1;
        if (src.name == goal)
        {
            cout << src.cname << endl;
            break;
        }
        cout << src.cname << " -> ";
        for (int i = 0; i < N; i++)
        {
            if (graph.getEdge(src.name, i) == 1 &&
!bfs_visited[i])
            {
                node des(i, 0);
                bfs_visited[i] = 1;
                q.push(des);
                // cout << "extend" << i << endl;
            }
        }
    }
    return;
}

```

```
bool cm_visited[N] = {0};
const int MAXNUM = 99999;
void climb_mountain(int goal, node &src, Graph &graph)
{
    if (src.name == goal)
    {
        cout << src.cname << endl;
        return;
    }
    cm_visited[src.name] = 1;
    cout << src.cname << " -> ";
    int h_best = MAXNUM;
    int next_visit = 0;
    for (int i = 0; i < N; i++)
    {
        if (graph.getEdge(src.name, i) == 1 && !cm_visited[i])
        {
            if (h[i] < h_best)
            {
                h_best = h[i];
                next_visit = i;
            }
        }
    }
    node des(next_visit, h_best);
    climb_mountain(goal, des, graph);
}

bool greedy_visited[N] = {0};
void greedy_best_first(int goal, node &src, Graph &graph)
{
    if (src.name == goal)
    {
        cout << src.cname << endl;
        return;
    }
    greedy_visited[src.name] = 1;
    cout << src.cname << " -> ";
    int h_best = MAXNUM;
    int next_visit = 0;
    for (int i = 0; i < N; i++)
    {
```

```
        if (graph.getEdge(src.name, i) == 1 && !greedy_visited[i])
        {
            if (h[i] < h_best)
            {
                h_best = h[i];
                next_visit = i;
            }
        }
    }
    node des(next_visit, h_best);
    greedy_best_first(goal, des, graph);
}

int main()
{
    Graph graph;
    graph.init();
    node src(S, h[S]);
    cout << "dfs:  ";
    dfs(G, src, graph);
    cout << "bfs:  ";
    bfs(G, src, graph);
    cout << "climb mountain:  ";
    climb_mountain(G, src, graph);
    cout << "greedy_best_first:  ";
    greedy_best_first(G, src, graph);
}

```

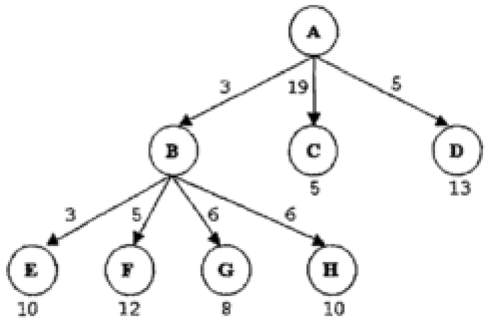
第4题

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 -> C，到达叶子节点，搜索结束。答案为A -> C
- ②若h(B)<=5，则A -> B。此时h(G)最小，故选择G节点作为扩展节点。答案为A -> B -> 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 -> B。又g(E)=6最小，故B -> E，到达叶子节点，搜索结束。答案为A -> B -> 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)<=15，则f(B)<=f(D)，此时选择B作为扩展节点。

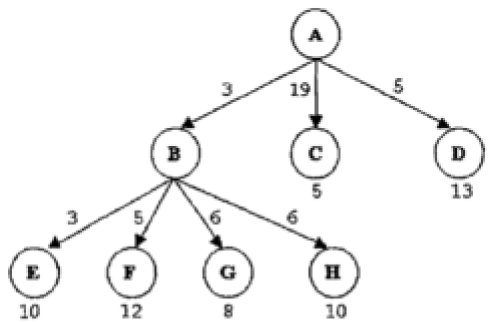
- f(E)=6+10=16
- f(F)=8+12=20
- f(G)=9+8=17
- f(H)=9+10=19

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) < = 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, n1, n2, ……, nm

根据一致性条件，$h(n) <= c(n, a1, n1) + h(n1)$

$<= c(n, a1, n1) + c(n1, a2, n2) + h(n2)$

$<= c(n, a1, n1) + c(n1, a2, n2) + \ldots\ldots + c(nm, am+1, G) + h(G)$

又 $h(G) == 0$，故 $h(n) <= c(n, a1, n1) + c(n1, a2, n2) + \ldots\ldots + c(nm, am+1, G)$

根据实际意义，$h^*(n) = c(n, a1, n1) + c(n1, a2, n2) + \ldots\ldots + c(nm, am+1, G)$，因为这是从n到G的实际距离。

综上，$h(n) <= h^*(n)$