华东师范大学数据科学与工程学院实验报告

课程名称：当代人工智能	**年级：**2021 级	**上机实践成绩：**
指导教师：李翔	**姓名：**李睿恩	**学号：**10215501434
上机实践名称：A*算法		**上机实践日期：**2023.10.31
上机实践编号：2	**组号：**无	**上机实践时间：**18:45

一、 实验目的

- 完成两道算法题，只能用 A*算法实现。

二、 实验任务

- 完成“冰雪魔方的冰霜之道”与“杰克的金字塔探险”的算法题。
- 在不限制编程语言的条件下，编程实现 A*算法。
- 每道题有 5 个测试输入，利用编写的算法预测对应输出

三、 实验环境

由于 C/C++在算法实现中可以有更好的运行效率，故本次实验基于 JetBrains 的 CLion 集成开发环境，使用 C++编程语言实现。

四、 实验过程

4.1 A*算法概述

A*算法是一种在有多个节点的路径的图形平面上，求出最低通过成本的算法，是启发式算法中的一种。传统的 BFS 与 DFS 总可以找到最优解，但他们的本质思想是遍历大量节点，因此运行效率较低。Dijkstra 算法会找到最短路径，但它会探索所有可能的路径，这仍然不够效率。Best-First Search 运行速度较快，但它不能保证找到最优解。而 A*算法结合了 Best-First Search 与 Dijkstra 算法的优点，在进行启发式搜索提高算法效率的同时，也能保证找到一条最短路径。

在这个算法中，我们会考量三个函数，分别为$f(n)$、$g(n)$与$h(n)$。其中，$g(n)$指的是从起点到任意节点 n 的实际距离；$h(n)$是任意节点 n 到目标顶点的估算距离，它也被称为启发式函数；$f(n)$被称为节点的估算函数，它是$g(n)$与$h(n)$二者之和，即

$$f(n)=g(n)+h(n).$$

在我们设计 A*算法时，最为重要的就是寻找到合适的启发式函数，只有在启发式函数设定合理的情况下，我们才能避免遍历过多不必要的节点。记$h^*(n)$为节点 n 到终点的实际距离，则 A*算法有如表 1 的特性。

因此，如果我们想要利用 A*算法尽可能快速地寻找到最优路径，我们应当确保自己选择的启发式函数可以确保每个节点的启发式函数值比实际的距离值要小，并且尽可能地让每一

个点的启发式函数值接近于该节点到达终点的实际最短距离。

表 1 启发式函数对 A*算法运行的影响

$h(n)$的大小	算法运行效果
$h(n)=0$	退化为 Dijkstra 算法，一定可以找到最优解，运行速度慢
$0<h(n)<h^*(n)$	一定可以找到最优解，运行速度更快了
$h(n)=h^*(n)$	一定可以找到最优解，运行速度非常快
$h(n)>h^*(n)$	不能保证可以找到最优解，但运行速度极其快

4.2 A*算法流程

创建两个数据结构，分别为 Open List 与 Close List。基于此，A*算法的流程如下。

1. 将起点加入 Open List；
2. 寻找 Open List 中当前 *f* 值最小的节点，把它作为当前要处理的节点 *p*；
3. 将该节点从 Open List 中删除，并放入 Close List 中；
4. 遍历节点 *p* 的每一个相邻节点 *s*：

4.1 如果节点 *s* 在 Close List 中，那么就忽略对节点 *s* 的判定；

4.2 如果 *s* 既不在 Open List 里也不在 Close List 里，则把节点 *s* 加入 Open List 中，并且把节点 *p* 设置为节点 *s* 的父节点。计算节点的 *f*, *g* 和 *h* 值并保存。

4.3 如果节点 *s* 已经在 Open List 中，依据 *g* 值检查这条路径是否是到达节点 *s* 的更好的路径。如果经过节点 *p* 到达节点 *s* 的路径更优，那么把 *s* 的父亲设置为 *p*，并重新计算保存 *s* 的 *g* 和 *f* 值，而如果原来的节点更优，则没有需要修改的值。

5. 如果 Open List 为空集，则没有找到目标最短路径，查找宣告失败。
6. 如果 Open List 不为空，则回到第二步。如果此时 *f* 值最小的节点为目标节点，则宣告找到了最短路径，否则继续执行第二步开始后的步骤。
7. 输出目标节点的 *g* 值，即为最终结果。

根据上述流程，我们有如图 1 的通用流程图可参考。

4.3 冰雪魔方的冰霜之道

4.3.1 题目描述

在遥远的冰雪王国中，存在一个由 9 个神秘冰块组成的魔法魔方。在这个魔法魔方中，有 8 块冰雪魔块，每块都雕刻有 1-8 中的一个数字（每个数字都是独特的）。魔方上还有一个没有雕刻数字的空冰块，用 0 表示。你可以滑动与空冰块相邻的冰块来改变魔块的位置。传说，当冰雪魔块按照一个特定的顺序排列（设定为 1 3 5 7 0 2 6 8 4）时，魔法魔方将会显露出通往一个隐秘冰宫的冰霜之道。现在，你站在这个神秘的魔方前，试图通过最少的滑动步骤将冰雪魔块排列成目标顺序。为了揭开这一神秘，你决定设计一个程序来从初始状态成功找到冰霜之道。

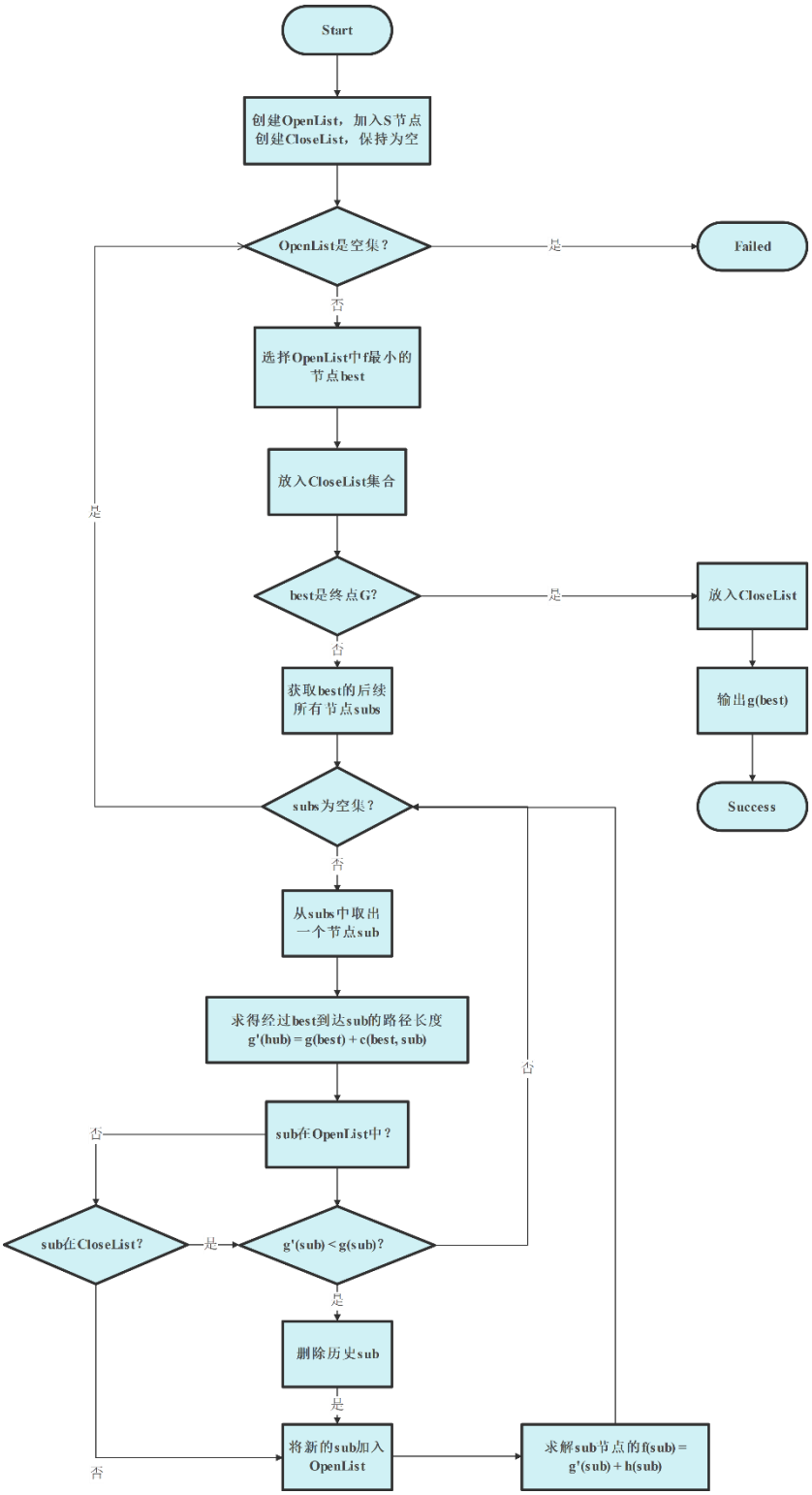


图 1 A*算法通用流程图

4. 3. 2 题目分析与求解

这道题目实际上是非常经典的八数码问题，题意即求从初始状态转变为目标状态的最少的滑动次数。

我们先定义一个 struct 结构体，将其命名为 IceState，该结构体中含有魔方目前的状态

（即数字的排列顺序），到达该状态所需的$f(n)$，$g(n)$，此时空冰块所在位置的横坐标 x 与纵坐标 y，以及到达此状态所滑动的次数。代码如图 2.

```
// Defines a state
struct IceState
{
    vector<vector<int>> Ice;
    // fn = gn + hn, where gn stands for the depth, hn stands for the prediction
    // and fn stands for the total
    int fn;
    int gn;
    // The position of empty
    int x;
    int y;
    int number;
};
```

图 2 问题一中描述魔方状态的结构体

在后续的代码执行过程中，我们需要保证魔方的状态是合法的。为此，我们编写 isSafe 的布尔函数，确认一个魔方的状态的合法性。代码如图 3.

```
// Make sure the state is okay
bool isSafe(int x, int y)
{
    return (x >= 0 && y >= 0 && x <= 2 && y <= 2);
}
```

图 3 问题一中确保魔方状态合法性的布尔函数

在 A*算法中，最重要的设计一个合理的启发式函数。在这里，我们采用**曼哈顿距离**作为启发式函数。在一个矩阵中，将两个点的横坐标之差的绝对值与纵坐标之差绝对值相加，得到的即为曼哈顿距离，即

$$d(i,j)=|x_1-x_2|+|y_1-y_2|$$

其中 i,j 为矩阵上的两个点。

计算魔方上的每一个冰块与其应当处于的位置的曼哈顿距离并求和，得到的即为该状态的启发式函数。该启发式函数值是一定会小于实际成本的，因为我们只是考虑了一个冰块直接移动到目标位置的代价，而没有考虑每次移动都必须发生在与空冰块相邻的冰块上，因此该值必然小于等于最优移动次数。

我们编写 Manhattan 函数计算目前状态与目标状态的曼哈顿距离，代码如图 4.

我们为每次移动定义一个函数 move，它会实现冰方块的移动，并返回新的状态。代码如图 5.

```cpp
// Calculate the Manhattan Distance of all nodes
int Manhattan(vector<vector<int>>& current, vector<vector<int>>& target)
{
    int distance = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            for (int k = 0; k < 3; k++)
            {
                for (int l = 0; l < 3; l++)
                {
                    if (current[i][j] == target[k][l] && current[i][j] != 0)
                    {
                        distance = distance + abs( X: k - i) + abs( X: l - j);
                    }
                }
            }
        }
    }
    return distance;
}
```

图 4 问题一中计算曼哈顿距离的函数

```cpp
// Change the state
IceState move(vector<vector<int>>& Ice, int x, int y, int new_x, int new_y, int gn, int number)
{
    vector<vector<int>> newIce = Ice;
    swap( &: newIce[x][y],  &: newIce[new_x][new_y]);

    IceState newIceState = { .Ice: newIce,  .fn: Manhattan( &: newIce,  &: Ice) + gn,  .gn: gn + 1,  .x: new_x,  .y: new_y,  .number: number + 1};
    return newIceState;
}
```

图 5 问题一中移动空冰块的函数

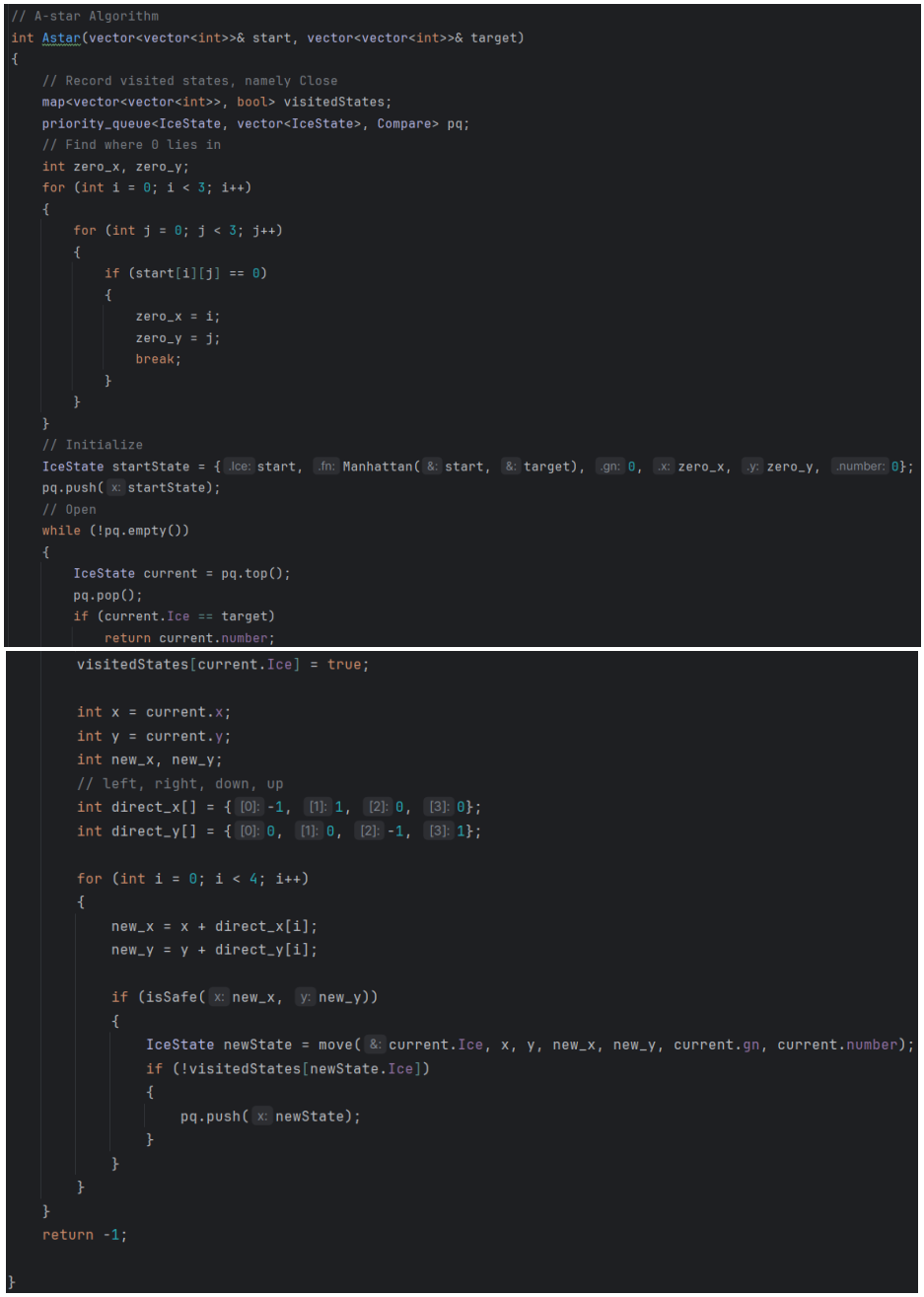
接下来是 Astar 函数，是 A*最为重要的算法实现函数。在这里，为了可以更加快捷地找到 OpenList 中的最小值，我们使用了优先队列(Priority Queue)的数据结构以存储 IceState 的结构体，每次从中取出最小值只需要 $O(1)$的时间复杂度。在这里，优先队列排序的依据是其中每一个 IceState 的 fn 的大小。

```cpp
// Will be used in A-star's priority queue
struct Compare
{
    bool operator()(const IceState& s1, const IceState& s2)
    {
        return s1.fn > s2.fn;
    }
};
```

图 6 问题一中优先队列的排序方式

同时，我们设置了一个映射(Map)数据结构，将一个动态数组(Vector)中的每个元素与一个布尔值建立映射关系，作为 CloseList，它会记录一个节点是否被访问过。

接下来的工作和我们在 4.2 中谈论到的一致，不额外赘述。具体实现代码如图 7.



```
// A-star Algorithm
int Astar(vector<vector<int>>& start, vector<vector<int>>& target)
{
    // Record visited states, namely Close
    map<vector<vector<int>>, bool> visitedStates;
    priority_queue<IceState, vector<IceState>, Compare> pq;
    // Find where 0 lies in
    int zero_x, zero_y;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (start[i][j] == 0)
            {
                zero_x = i;
                zero_y = j;
                break;
            }
        }
    }
    // Initialize
    IceState startState = { .Ice: start, .fn: Manhattan( &: start, &: target), .gn: 0, .x: zero_x, .y: zero_y, .number: 0};
    pq.push( x: startState);
    // Open
    while (!pq.empty())
    {
        IceState current = pq.top();
        pq.pop();
        if (current.Ice == target)
            return current.number;
        visitedStates[current.Ice] = true;

        int x = current.x;
        int y = current.y;
        int new_x, new_y;
        // left, right, down, up
        int direct_x[] = { [0]: -1,  [1]: 1,  [2]: 0,  [3]: 0};
        int direct_y[] = { [0]: 0,  [1]: 0,  [2]: -1,  [3]: 1};

        for (int i = 0; i < 4; i++)
        {
            new_x = x + direct_x[i];
            new_y = y + direct_y[i];

            if (isSafe( x: new_x,  y: new_y))
            {
                IceState newState = move( &: current.Ice, x, y, new_x, new_y, current.gn, current.number);
                if (!visitedStates[newState.Ice])
                {
                    pq.push( x: newState);
                }
            }
        }
    }
    return -1;

}
```

图 7 问题一中 A*算法的实现函数

4.4 杰克的金字塔探险

4.4.1 题目描述

在一个神秘的王国里，有一个名叫杰克的冒险家，他对宝藏情有独钟。传说在那片广袤的土地上，有一座名叫金字塔的奇迹，隐藏着无尽的财富。杰克决定寻找这座金字塔，挖掘隐藏在其中的宝藏。金字塔共有 N 个神秘的房间，其中 1 号房间位于塔顶，N 号房间位于塔底。在这些房间之间，有先知们预先设计好的 M 条秘密通道。这些房间按照它们所在的楼层顺序进行了编号。杰克想从塔顶房间一直探险到塔底，带走尽可能多的宝藏。然而，杰克对寻宝路线有着特别的要求：

(1) 他希望走尽可能短的路径，但为了让探险更有趣和挑战性，他想尝试 K 条不同的较短路径。

(2) 他希望在探险过程中尽量节省体力，所以在选择通道时，他总是从所在楼层的高处到低处。

现在问题来了，给你一份金字塔内房间之间通道的列表，每条通道用(X_i, Y_i, D_i)表示，表示房间 X_i 和房间 Y_i 之间有一条长度为 D_i 的下行通道。你需要计算出杰克可以选择的 K 条最短路径的长度，以便了解他在探险过程中的消耗程度。

4.4.2 题目分析与求解

该问题可以化归为一个有权有向图求最短 K 条路径的问题。

为此，我们先定义一个 struct 结构体，命名为 path，它存储了题目中有效的路径。这里的有效路径指的是我们输入的路径，且我们还需要额外手动排除掉从底层到高层的路径。代码如图 8.

```
struct path {
    int target;
    int cost;
};
```

图 8 问题二中记录有效路径的结构体

我们还需要创建另一个 struct 结构体，命名为 Node，它存储了一个节点的$f(n)$，$g(n)$，$h(n)$以及目前所处的房间。代码如图 9.

```
// fn stands for the sum of gn and hn,
// while gn stands for the real cost,
// hn stands for the heuristic
struct Node
{
    int fn;
    int gn;
    int hn;
    int start_point;
};
```

图 9 问题二中存储目前状态的结构体

我们还需要让我们输入的路线形成一张有向图。在这里，我们使用动态数组 Vector 实现一个邻接矩阵，其中的第 i 行第 j 列的值 k 说明从房间 i 到房间 j 需要付出 k 的代价，实现代码见图 10。

接下来我们直接关注 Astar 函数。如同我们在 4.2 中的分析，我们设立一个优先级队列作为 OpenList，存储每次遍历到的 Node。

```
vector<int> findResult(int N, int M, vector<vector<int>>& connections)
{
    map<int, vector<path>> graph;

    for (auto & connection :vector<int> & : connections)
    {
        int x = connection[0];
        int y = connection[1];
        int d = connection[2];
        graph[x].push_back({ .target: y, .cost: d});
    }

    vector<int> result = astar( &: graph, start: 0, end: N-1);

    return result;
}
```

图 10 问题二生成邻接矩阵的函数

该问题与上一道问题有一个很大的区别。上一道问题只需要求出一个最优解，因此我们可以使用 CloseList 来判断一个节点是否被遍历过。但这道问题要求我们求出前 K 个最优解，使用 CloseList 很可能会让我们少解。为此，我们不再设置 CloseList，而是将每次遍历的节点都纳入到 OpenList 中。即使有两个节点所处的房间相同，他们的 fn, gn 与 hn 属性也会使得他们有所区分。当 OpenList 为空集时，我们就获得了从小到大排列的所有最优解。

在这里，我们需要关注其中使用的启发式函数。一种流行的说法是执行一次 Dijkstra 算法，从而得到每个点更接近真实代价的启发式函数值。但我们已经在使用 A*算法来避免 Dijkstra 算法本身的缺点，因此我们计划使用更加简单的启发式函数。当我们到达一个房间时，我们将该房间的出路中成本最小的路径长度作为启发式函数值。容易知道，这样取得的启发式函数值必然是小于等于从该节点出发到达终点的最优解的，因为我们只计算了一条路径的长度，且这条路径还是该房间的所有出路中最短的一条。

基于此，我们有了 A*算法的代码。具体代码见图 11.

```
// Astar algorithm
vector<int> astar(map<int, vector<path>>& graph, int start, int end) {
    vector<int> result;
    priority_queue<Node *, std::vector<Node *>, CompareNode> pq;
    // The shortest path to the next is the heuristic
    int first_fn = 256;
    for (auto &i :path & : graph[0])
    {
        int cost = i.cost;
        if (cost < first_fn)
            first_fn = cost;
    }
    Node node_start = { .fn: first_fn, .gn: 0, .hn: first_fn, .start_point: 0};
    pq.push( x: &node_start);

    while (!pq.empty()) {
        Node *currentNode = pq.top();
        pq.pop();
        int fn_cur = currentNode->fn;
        int gn_cur = currentNode->gn;
        int hn_cur = currentNode->hn;
        int room_cur = currentNode->start_point;
        if (room_cur == end) {
            result.push_back(gn_cur);
            continue;
        }
```

```
        for (auto &i : path& : graph[room_cur])
        {
            int neighbour = i.target;
            int heuristic = 256;
            // The shortest path to the next is the heuristic
            for (auto &j : path& : graph[neighbour])
            {
                int path_cost = j.cost;
                if (path_cost < heuristic)
                    heuristic = path_cost;
            }
            int cost = i.cost;
            int new_cost = gn_cur + cost;
            int new_hn = heuristic;
            Node* node_next = new Node{ .fn: new_cost + new_hn, .gn: new_cost, .hn: new_hn, .start_point: neighbour};
            pq.push( x: node_next);
        }

    }
    return result;
}
```

图 11 问题二的 A*算法实现

4.5 运行测试代码

4.5.1 问题一的测试

我们要执行如图 12 的测试样例。

```
135720684
105732684
015732684
135782604
715032684
```

图 12 问题一的测试数据

将这 5 条测试数据依次在问题一的程序中输入，可以依次得到结果：1, 1, 2, 1, 3.

```
Please print a nine-digit number:135720684
1
Please print a nine-digit number:105732684
1
Please print a nine-digit number:015732684
2
Please print a nine-digit number:135782604
1
Please print a nine-digit number:715032684
3
```

图 13 问题一的测试数据结果

4.5.2 问题二的测试

我们要执行如图 14 的测试样例。

```
5 6 4
1 2 1
1 3 1
2 4 2
2 5 2
3 4 2
3 5 2
```

```
6 9 4
1 2 1
1 3 3
2 4 2
2 5 3
3 6 1
4 6 3
5 6 3
1 6 8
2 6 4
```

```
7 12 6
1 2 1
1 3 3
2 4 2
2 5 3
3 6 1
4 7 3
5 7 1
6 7 2
1 7 10
2 6 4
3 4 2
4 5 1
```

```
5 8 7
1 2 1
1 3 3
2 4 1
2 5 3
3 4 2
3 5 2
1 4 3
1 5 4
```

```
6 10 8
1 2 1
1 3 2
2 4 2
2 5 3
3 6 3
4 6 3
5 6 1
1 6 8
2 6 5
3 4 1
```

图 14 问题二的测试数据

将这 5 条测试数据依次在问题一的程序中输入，可以依次得到结果：

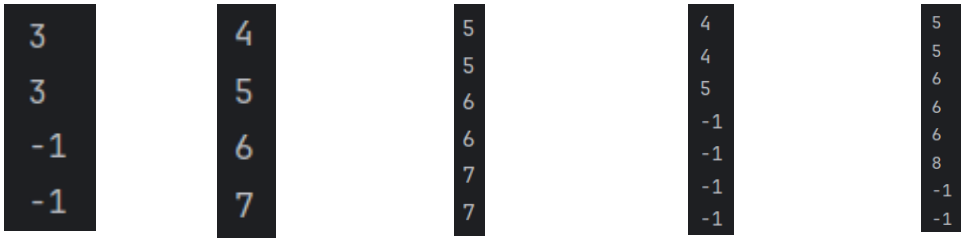


图 15 问题二的测试数据结果

五、 总结与分析

在理论课程的学习中我们了解到，传统的 BFS 与 DFS 总可以找到最优解，但他们的本质思想是遍历大量节点，因此运行效率非常低，而尽管 Dijkstra 算法会找到最短路径，但它会探索所有可能的路径，这仍然不够效率。Best-First Search 运行速度较快，但它却不能保证找到最优解。A*算法是启发式算法之一，它结合了 Best-First Search 与 Dijkstra 算法的优点，在进行启发式搜索提高算法效率的同时，也能保证找到一条最短路径。

A*算法的重点在于寻找启发式函数。如果一个节点的启发式函数值小于等于它之后的实际代价，那么该算法必然可以找到最优解，且运行效率相对较高。启发式函数值越接近之后的实际代价，该算法的运行效率就越高。

A*算法也有自己的缺点。一方面，如果一个节点的启发式函数值大于了它之后的实际代价，那么尽管该算法的运行效率会很快，却无法保证自己可以找到最优解。另一方面，寻找一个合适的，效率高的启发式函数并不是一件简单的事，如果设计地过于简单，代码的效率就不会有显著提升，而如果希望启发式函数值尽可能接近实际代价，则又可能设计地过于复杂。

但综合来看，A*算法作为一个启发式算法，对比于其他算法仍然有着非常大的优势。在未来类似的问题中，我们可以尝试使用 A*算法来提升运行效率。

六、 参考技术博客

[1] A*算法（A-star Algorithm）详解：理论与实例
URL: https://zhuanlan.zhihu.com/p/590786438/
[2] K 短路算法——A*算法
URL: https://zhuanlan.zhihu.com/p/32371294/