人工智能实验

无信息搜索和启发式搜索

第 8 次实验

姓名：陈家豪

班级：2018 级计科 1 班

学号：18308013

目录

1 算法原理

1.1 搜索问题

在人工智能中，搜索问题是指已知智能体的初始状态和目标状态，求解一个动作序列使得智能体可以从初始状态转移到目标状态。如果所求序列可以使得总耗散最低，则问题称为最优搜索问题。

解决搜索问题的算法分为两大类，一类是无信息搜索，一类是启发式搜索。

1.2 算法性能评价

在评价一个搜索算法的性能时，我们需要从四个方面进行分析：

(1) 完备性：当问题有解时，这个算法能否保证找到解；

(2) 最优性：搜索策略能否找到最优解；

(3) 时间复杂度：找到解所需要的时间，也叫搜索代价；

(4) 空间复杂度：执行搜索过程中需要多少内存空间；

1.3 无信息搜索

无信息搜索又称为盲目搜索。这类搜索策略都采用固定的规则来选择下一需要被扩展的状态。这些规则不会随着要搜索解决的问题的变化而变化。此外，这些策略不考虑任何与要解决的问题领域相关的信息。一般常见的无信息搜索有宽度优先搜索、深度优先搜索、一致代价搜索、深度受限搜索、迭代加深搜索和双向搜索等。

1.3.1 宽度优先搜索和深度优先搜索

宽度优先搜索和深度优先搜索是最基础的两种搜索算法，后续的搜索算法大多数以这两种策略为基础。

宽度优先搜索（BFS）是一般图搜索算法的一个实例。其节点扩展顺序与目标节点的位置无关，每次总是扩展深度最浅的节点，通过将边缘组织成 FIFO 队列来实现。其具有完备性，可以遍历整个图得到解；但其不一定有最优性。只有当路径代价是基于节点深度的非递减函数时，BFS 才具有最优性。

假设 b 为问题中一个状态最大的后继状态个数，d 为最短解的动作个数，那么 BFS 的时间复杂度为 $O(b^{d+1})$，空间复杂度为 $O(b^{d+1})$。

深度优先搜索（DFS）总是扩展搜索树的当前边缘节点集中最深的节点。如果最深层节点扩展完了，就回溯到下一个还有未扩展节点的深度稍浅的节点。因此，其一般使用 LIFO 队列实现。只有在状态空间有限、且对重复路径进行剪枝的情况下，其具有完备性。此外，其没有最优性。DFS 的时间复杂度为 $O(b^m)$，其中 m 是遍历过程中最长路径的长度，而空间复杂度是线性的，为 $O(bm)$。

1.3.2 一致代价搜索

一致代价搜索（UFS）可以看成是一种特殊的宽度优先搜索，其不是优先扩展深度最浅的节点，而是优先扩展代价最低的节点。每个节点都有一个代价，指的是从初始节点到当前节点需要的成本。在选择

下一个扩展节点时，我们总是选择下一步能到达的代价最低的未扩展节点。显然，当代价为深度时，一致代价搜索相当于宽度优先搜索。

当代价不小于 0 时，一致代价搜索具有完备性和最优性。一方面，所有成本较低的路径都会在成本高的路径之前被扩展；另一方面，在给定成本时该成本的路径数量是有限的。因此，在有解的情况下我们可以找到最短的路径。而对于时间复杂度和空间复杂度而言，当最优解成本为 C^*、每次动作的代价至少为 e 时，复杂度为 $O(b^{1+[C^*/e]})$。而当每步代价都相等时，复杂度将变成宽度优先搜索的 $O(b^d)$。

1.3.3 深度受限搜索和迭代加深搜索

深度受限搜索是对深度优先搜索的一种改进。在使用深度优先搜索时，可能会出现运行时间长、在无限状态空间下无限运行等问题。因此，我们可以提前设置搜索深度 L。如果当前搜索的节点深度大于 L 时，我们就把其当做最深层节点来对待，不再继续往下搜索。显然，这个算法既不具有完备性也不具有最优性，因为若目标节点的深度大于 L，我们将无法找到解。此外，由于限制了最大深度为 L，故其时间复杂度为 $O(b^L)$，空间复杂度为 $O(bL)$。

迭代加深搜索是对深度受限搜索的多次应用，其同时结合了宽度优先搜索和深度优先搜索的优点。简单来说，其一开始设置深度限制为 $L = 0$，然后迭代增加深度限制，并对每个深度限制都进行一次深度受限搜索，直到找到解或者深度限制不能再提高为止。显然，迭代加深搜索具有完备性，而当每个动作成本一致时也具有最优性。此外，其时间复杂度为 $O(b^d)$，空间复杂度为 $O(bd)$。

1.3.4 双向搜索

双向搜索是一种特殊的搜索算法，其同时从初始节点向前搜索和从目标节点向后搜索。当搜索边缘存在交集时就找到了解。双向搜索的完备性和最优性依赖于两个搜索所采用的算法。当搜索采用宽度优先搜索，且每个动作成本一致时，具有完备性和最优性。由于是同时从两端开始搜素，因此时间和空间复杂度为 $O(b^{d/2})$。

1.4 启发式搜索

启发式搜索又称为有信息搜索。相比于无信息搜索，其使用问题所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。常见的启发式搜索有 A* 搜索、IDA* 搜索等。

1.4.1 启发式函数

对于一个具体的问题，启发式搜索构造一个专用于该领域的启发式函数 $h(n)$, 用于估计从节点 n 到达目标节点的成本。启发式搜索的完备性和最优性一般依赖于 $h(n)$ 的设计。一般而言，$h(n)$ 需要满足以下两个条件：

(1) 可采纳性：若从节点 n 到达目标节点的预估成本小于真实成本，即 $h(n) < h^*(n)$，那么 $h(n)$ 具有可采纳性，算法必然可以找到从初始节点到目的节点的最短路径；

(2) 一致性（单调性）：若对任意节点 n_1 和 n_2，有 $h(n_1) \leq c(n_1 \to n_2) + h(n_2)$，则 $h(n)$ 具有一致性。满足一致性的启发式函数也一定满足可采纳性；

一般而言，一个正确的启发式函数一定会满足可采纳性，而一致性可以保证启发式搜索在环检测下也能正确执行。

在本次迷宫问题中，我们可以选取曼哈顿距离和切比雪夫距离作为启发式函数。这两种启发式函数都满足可采纳性和一致性。

1.4.2 A* 搜索

A* 搜索可以看成是一种特殊的一致代价搜索，区别在于扩展节点时优先选择评价值最低的节点。我们定义一个节点的评价函数如下：

$$f(n) = g(n) + h(n) \tag{1}$$

其中 $g(n)$ 是从初始节点到节点 n 的路径成本，而 $h(n)$ 为启发式估计值。从初始节点开始，每次扩展我们都会选择可到达的 $f(n)$ 值最小的未扩展节点进行搜索。当启发式函数满足可采纳性和一致性时，A* 搜索也具有完备性和最优性。若启发式函数只有可采纳性，使用环路检测时 A* 搜索不能保证最优性。

当 $h(n) = 0$ 时，我们可以发现 A* 搜索相当于一致代价搜索。因此一致代价搜索的时间和空间复杂度下界与 A* 搜索的相同。

1.4.3 IDA* 搜索

IDA* 搜索可以看成一种特殊的迭代加深搜索，区别在于划定界限的不是深度，而是评价值 f。其一开始设置评价值限制为 0，然后迭代提高限制，并对每个限制都进行一次依据评价值进行选择的深度优先搜索。同样，当启发式函数满足可采纳性和一致性时，IDA* 具有完备性和最优性，其时间和空间复杂度下界与迭代加深搜索相同。相比于 A* 搜索，IDA* 搜索空间复杂度更小。

2 伪代码/流程图

本次实验，对于无信息搜索，我分别实现了一致代价搜索、迭代加深搜索和双向搜索。对于启发式搜索，我实现了 A* 搜索和 IDA* 搜索。

2.1 一致代价搜索

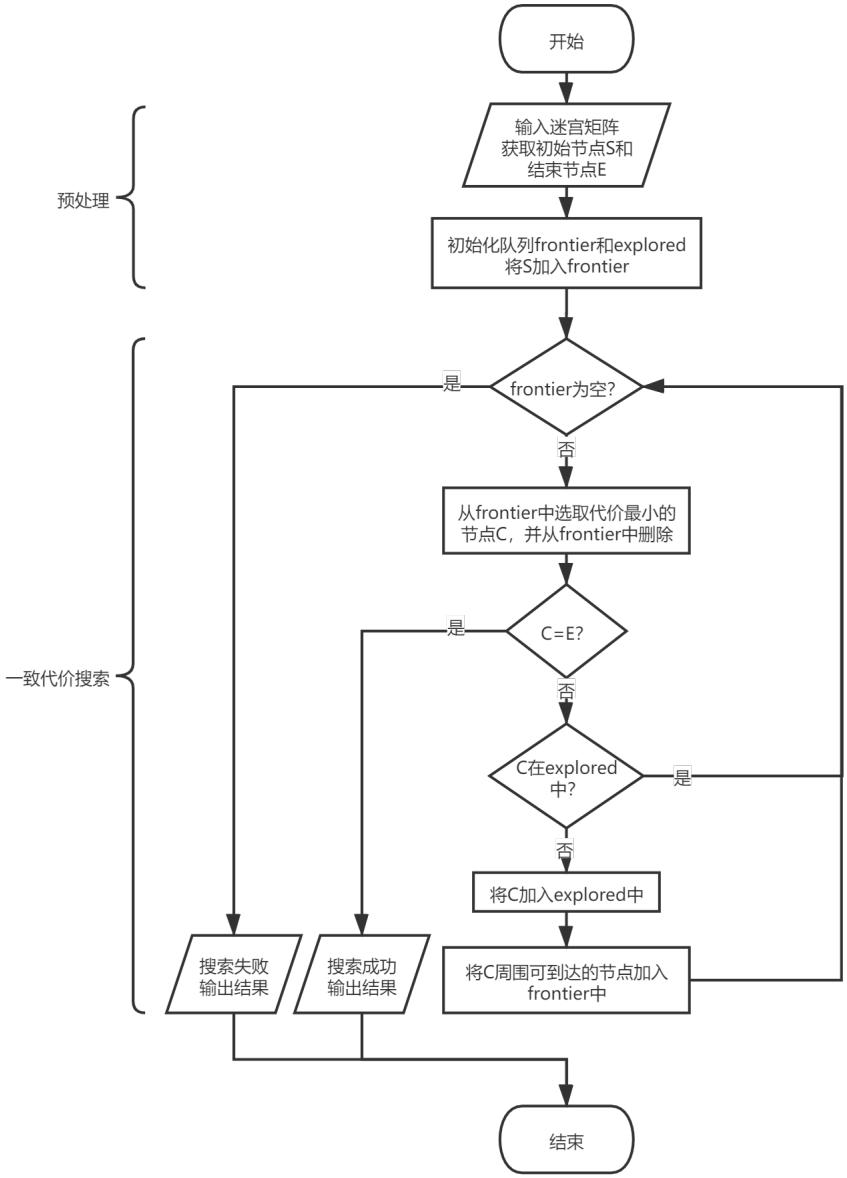

图 1: 一致代价搜索流程图

2.2 迭代加深搜索

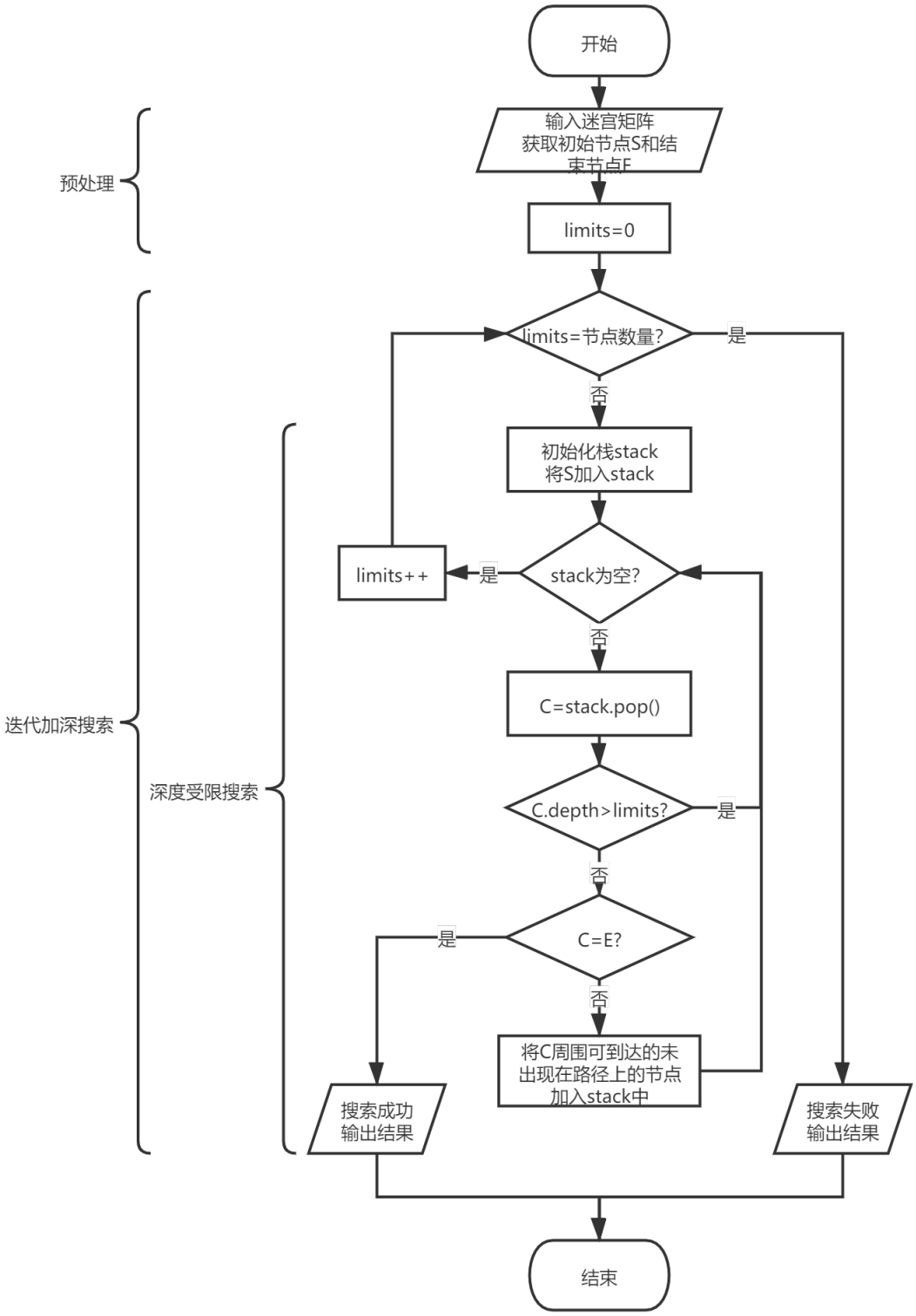

图 2: 迭代加深搜索流程图

因为若有解存在，那么一定有非环路的解存在，故我们假设最大深度即为节点数量。

2.3 双向搜索

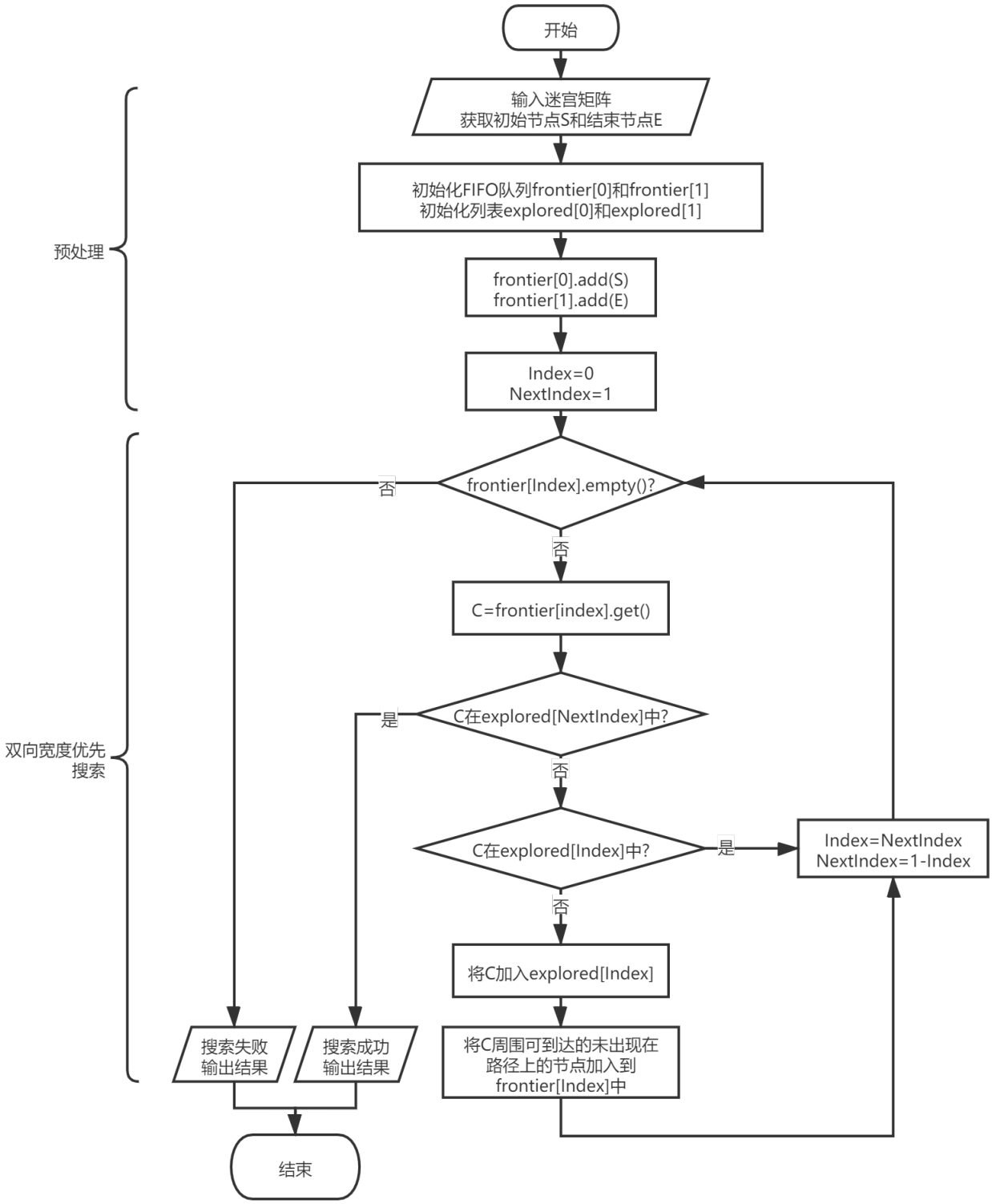

图 3: 双向搜索流程图

2.4 A* 搜索

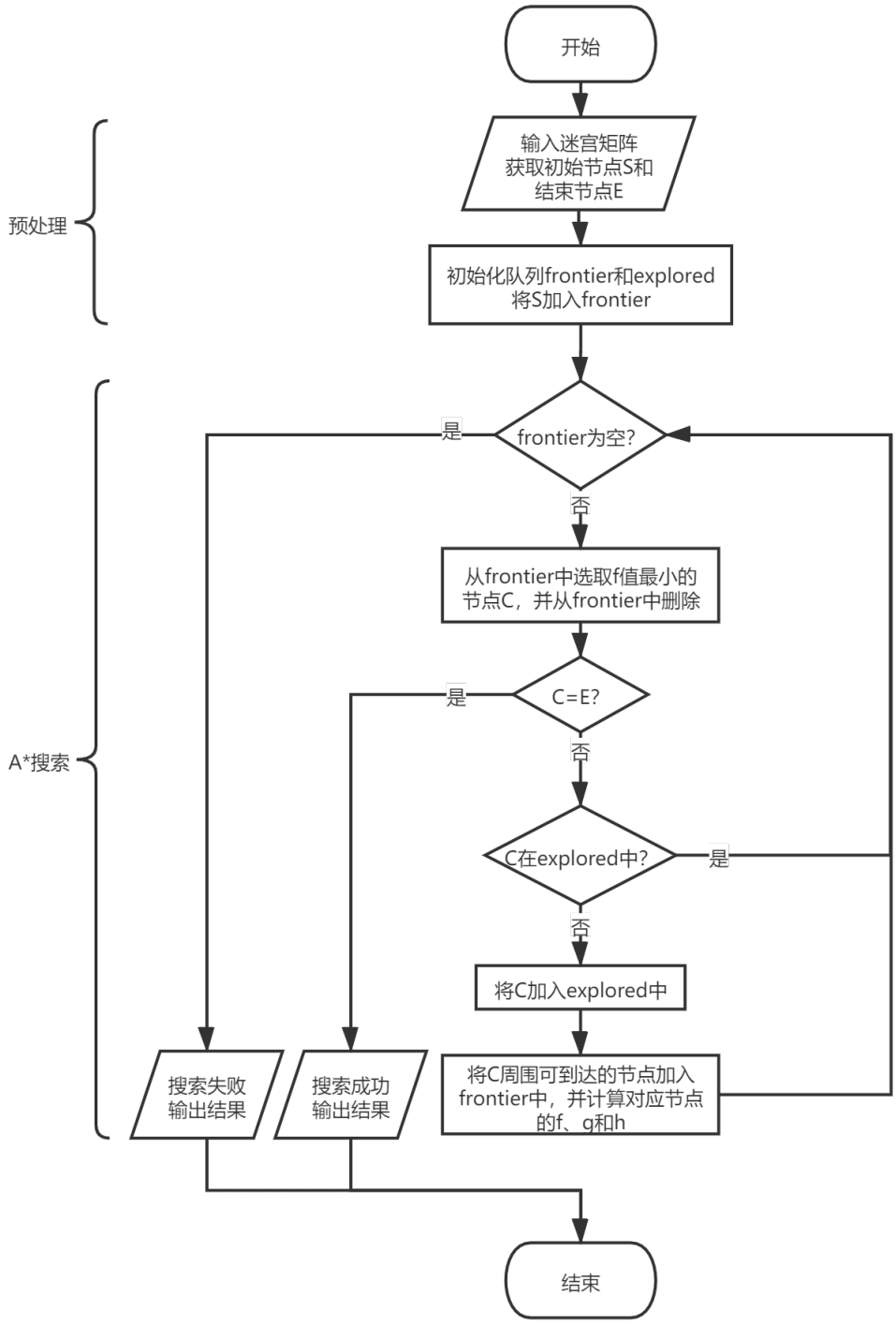

图 4: A* 搜索流程图

2.5 IDA* 搜索

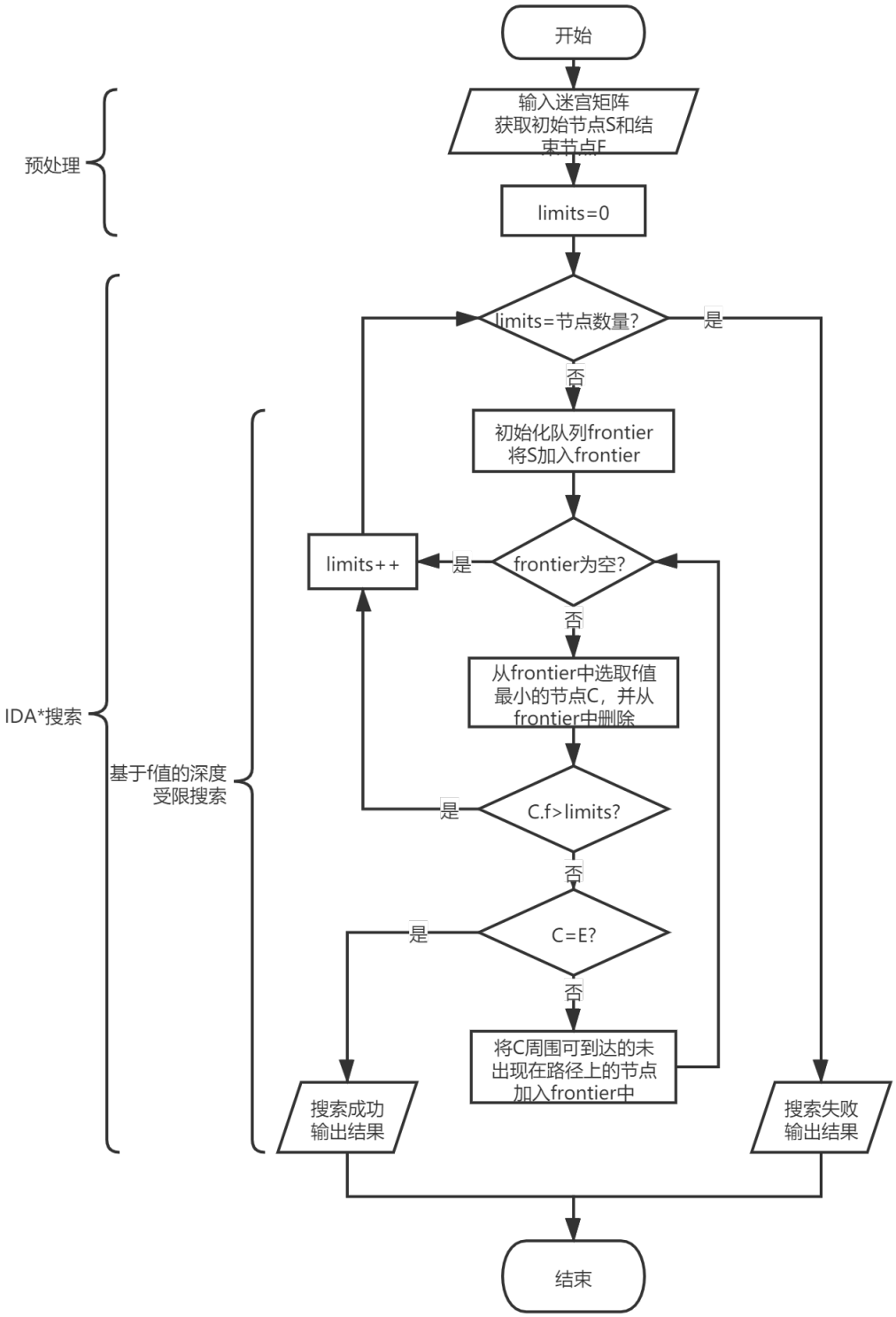

图 5: IDA* 搜索流程图

同样，若有解存在，那么一定有非环路的解存在。在迷宫问题中，一个节点的评价值不会大于节点数量，故我们假设最大 f 即为节点数量。

3 关键代码

3.1 一致代价搜索

一致代价搜索的关键代码如下所示：

```
class node():
    def __init__(self,X,Y,Cost):
        self.x=X                # 坐标x
        self.y=Y                # 坐标y
        self.cost=Cost          # 从初始节点到当前节点的代价
        self.path=[(X,Y)]       # 从初始节点到当前节点的路径
    def __lt__(self,other): # 重载<，用于优先队列
        return self.cost<other.cost
    def get_address(self):      # 返回当前坐标的元组
        return (self.x, self.y)
    def update_path(self,last):
        self.path=list(last.path)
        self.path.append(self.get_address())

def UCS(graph, StartNode, EndNode):
    frontier=PriorityQueue()# 等待探索的节点优先队列
    explored=set()          # 已经探索过的节点集合
    lenX=len(graph)
    lenY=len(graph[0])
    frontier.put(StartNode)
    while True:
        if frontier.empty():# 没有可探索的节点，搜索失败
            return False
        else:
            CurrentNode=frontier.get()# 获得队列中cost最小的节点
            if CurrentNode.get_address()==EndNode.get_address():# 是目标节点，搜索成功
                return True
            if CurrentNode.get_address() in explored:# 已经探索过，跳过
                continue
            else:
                explored.add(CurrentNode.get_address())
            # 将周围可到达的节点加入队列
            if CurrentNode.y>0 and graph[CurrentNode.x][CurrentNode.y-1]!='1':
                NextNode=node(CurrentNode.x,CurrentNode.y-1,CurrentNode.cost+1)
                NextNode.update_path(CurrentNode)
                frontier.put(NextNode)
            ..... # 共有四个节点需要判断
```

我们首先定义了 node 类和对应的一系列方法。正如图1所示，我们使用优先队列实现从 frontier 中选取代价最小的节点。当搜索到目标节点时，我们输出路径长度和相应的路径，返回 True。若没有找到解，则返回 False。

3.2 迭代加深搜索

迭代搜索的关键代码如下所示：

```
def Depth_limited_Search(graph, StartNode, EndNode, limited_Depth):
    # 深度受限搜索
    stack=list()
    lenX=len(graph)
    lenY=len(graph[0])
    stack.append(StartNode)
    while len(stack)!=0:
        CurrentNode=stack.pop()
        if CurrentNode.depth>limited_Depth:# 超过最大深度，跳过
            continue
        else:
            if CurrentNode.get_address()==EndNode.get_address():# 搜索到目标节点，成功
                return CurrentNode
            else:
                # 将周围可到达的不在路径上的节点压栈
                if CurrentNode.y>0 and graph[CurrentNode.x][CurrentNode.y-1]!='1':
                    NextNode=node(CurrentNode.x,CurrentNode.y-1,CurrentNode.depth+1)
                    if NextNode.get_address() not in CurrentNode.path:
                        NextNode.update_path(CurrentNode)
                        stack.append(NextNode)
                ..... # 共有四个节点需要判断
    return None # 不存在解，返回空值

def Ierative_Deepening_Search(graph, StartNode, EndNode):
    # 迭代加深搜索
    lenX=len(graph)
    lenY=len(graph[0])
    for i in range(lenX*lenY):
        finish=Depth_limited_Search(graph,StartNode,EndNode,i)
        if finish is None:# 该深度没有解
            continue
        else:             # 该深度有解
            return True
    return False
```

在迭代加深搜索中，node 类与一致代价搜索的 node 类基本相同。我们构造了深度受限函数 Depth_limited_Search()，如果其能搜索到目标节点就返回对应的 node 类，否则返回空值。而迭代加深搜索使用不同的搜索深度调用深度受限搜索函数，如果有节点返回则说明有解，若始终返回空值则说明失败。

因为若有解存在，那么一定有非环路的解存在，故我们假设最大深度上限即为节点数量。

3.3 双向搜索

双向宽度优先搜索的关键代码如下所示：

```python
def BiBFS(graph, StartNode, EndNode):
    # 双向宽度优先搜索
    frontier=[]
    frontier.append(Queue())# 正向搜索待搜索队列
    frontier.append(Queue())# 反向搜索待搜索队列
    explored=[]
    explored.append([])# 正向搜索已搜索列表
    explored.append([])# 反向搜索已搜索列表
    lenX=len(graph)
    lenY=len(graph[0])
    frontier[0].put(StartNode)
    frontier[1].put(EndNode)
    Index=0
    NextIndex=(Index+1)%2
    while True:
        if frontier[Index].empty():# 当前队列为空，搜索失败
            return False
        else:
            CurrentNode=frontier[Index].get()
            # 判断当前节点是否已被另一搜索搜过
            for OtherNode in explored[NextIndex]:
                if CurrentNode.get_address()==OtherNode.get_address():
                    # 已被另一搜索搜过，搜索成功
                    return True
            # 判断当前节点是否已被当前搜索搜过
            flag=False
            for OtherNode in explored[Index]:
                if CurrentNode.get_address()==OtherNode.get_address():
                    flag=True
                    break
            if flag:# 已被搜过，跳过
                continue
            else:   # 未被搜过，加入已搜索列表
                explored[Index].append(CurrentNode)
            # 将周围可到达的节点加入队列
            if CurrentNode.y>0 and graph[CurrentNode.x][CurrentNode.y-1]!='1':
                NextNode=node(CurrentNode.x,CurrentNode.y-1,CurrentNode.depth+1)
                NextNode.update_path(CurrentNode)
                frontier[Index].put(NextNode)
            ..... # 共有四个节点需要判断
            # 切换至另一个搜索
            Index=(Index+1)%2
            NextIndex=(Index+1)%2
```

由于在单线程的情况下无法做到同时搜索，故我采用循环的方式依次进行正向搜索和反向搜索。每个搜索都需要一个队列用于存储待探索的节点，和一个列表用于存储已探索的节点。

3.4 A* 搜索

A* 搜索的关键代码如下所示：

```
class node():
    def __init__(self,X,Y,G):
        self.x=X
        self.y=Y
        self.g=G
        self.h=0
    def __lt__(self,other):      # 重载<，根据f值排序
        return self.g+self.h<other.g+other.h
    def get_address(self):
        ......
    def update_path(self,last): # 更新从初始节点到当前节点的路径
        ......
    def update_h(self,end,D):    # 使用曼哈顿距离
        self.h=D*abs(self.x-end.x)+abs(self.y-end.y)

def A_star_search(graph,StartNode,EndNode):
    # A*搜索
    ......
```

我们为 node 类添加了当前代价值 g 和启发式函数值 h，update_h() 函数用于获取当前节点的 h。A* 搜索的代码与3.1的一致代价搜索代码基本一致，区别在于每个节点在加入优先队列前需要更新 h 值，以及通过重载 node 类的 < 使得优先队列按照 f 值排序而不是代价排序。故不再重复粘贴代码。

3.5 IDA* 搜索

IDA* 搜索的关键代码如下所示：

```
def Search(graph, StartNode, EndNode, limits):
    # f值受限搜索
    frontier=PriorityQueue()
    lenX=len(graph)
    lenY=len(graph[0])
    frontier.put(StartNode)
    while not frontier.empty():
        CurrentNode=frontier.get()
        if CurrentNode.get_f()>limits:  # 当前节点f值超过limits，直接退出
            return None
        else:
            if CurrentNode.get_address()==EndNode.get_address():# 搜到目标节点，返回该节点
                return CurrentNode
            else:
                #  将周围可到达的不在路径上的节点加入队列
                if CurrentNode.y>0 and graph[CurrentNode.x][CurrentNode.y-1]!='1':
                    NextNode=node(CurrentNode.x,CurrentNode.y-1,CurrentNode.g+1)
                    if NextNode.get_address() not in CurrentNode.path:
```

```
                            NextNode.update_path(CurrentNode)
                            NextNode.update_h(EndNode,1)
                            frontier.put(NextNode)
                    ..... # 共有四个节点需要判断
    return None # 队列为空，无解，返回空值

def IDA_Star_Search(graph, StartNode, EndNode):
    # IDA*搜索
    lenX=len(graph)
    lenY=len(graph[0])
    for i in range(lenX*lenY):
        finish=Search(graph,StartNode,EndNode,i)
        if finish is None:
            continue
        else:
            return True
    return False
```

IDA* 搜索的 node 类与 A* 搜索的 node 类相同。与3.2的迭代加深搜索相似，我们根据不同的 f 值限制调用受限搜索函数。由于在迷宫问题中，f 值不会超过所有节点数量，故我们设置最大 f 值上限为节点数量。

4 实验过程与结果分析

4.1 一致代价搜索结果

一致代价搜索结果如图6所示，可见程序找到了从初始节点到目标节点的最短路径，长度为 68。

```
(base) PS E:\code\Windows\Python\AIlab8> python UCS.py
Success! The length is 68
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1, 26), (1, 25), (2, 25), (3, 25
), (3, 26), (3, 27), (4, 27), (5, 27), (6, 27), (6, 26), (6, 25), (6, 24), (5, 24), (5, 23), (5, 22), (5,
21), (5, 20), (6, 20), (7, 20), (8, 20), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26), (8, 27), (9
, 27), (10, 27), (11, 27), (12, 27), (13, 27), (14, 27), (15, 27), (15, 26), (15, 25), (15, 24), (15, 23),
 (15, 22), (15, 21), (15, 20), (15, 19), (15, 18), (15, 17), (15, 16), (15, 15), (15, 14), (15, 13), (15,
12), (15, 11), (15, 10), (16, 10), (16, 9), (16, 8), (16, 7), (16, 6), (16, 5), (16, 4), (16, 3), (16, 2),
 (16, 1)]
```

图 6: 一致代价搜索结果

将路径可视化如图7所示。

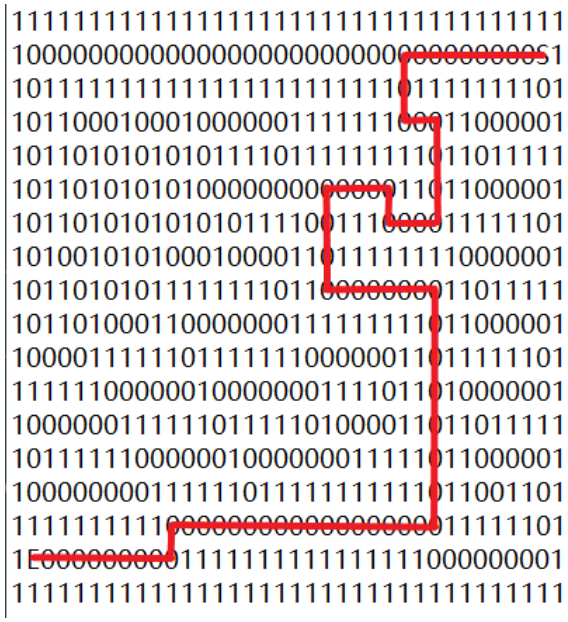

图 7: 结果可视化

4.2 迭代加深搜索结果

一致代价搜索结果如图8所示，可见程序找到了从初始节点到目标节点的最短路径，长度为 68。

```
(base) PS E:\code\Windows\Python\AIlab8> python IDS.py
Success! The length is 68
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1, 26), (1, 25), (2, 25), (3
, 25), (3, 26), (3, 27), (4, 27), (5, 27), (6, 27), (6, 26), (6, 25), (6, 24), (5, 24), (5, 23), (5, 2
2), (5, 21), (5, 20), (6, 20), (7, 20), (8, 20), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26),
 (8, 27), (9, 27), (10, 27), (11, 27), (12, 27), (13, 27), (14, 27), (15, 27), (15, 26), (15, 25), (15
, 24), (15, 23), (15, 22), (15, 21), (15, 20), (15, 19), (15, 18), (15, 17), (15, 16), (15, 15), (15,
14), (15, 13), (15, 12), (15, 11), (15, 10), (16, 10), (16, 9), (16, 8), (16, 7), (16, 6), (16, 5), (1
6, 4), (16, 3), (16, 2), (16, 1)]
```

图 8: 迭代加深搜索结果

结果可视化与图7相同。

4.3 双向搜索结果

双向广度优先搜索结果如图9所示，可见程序找到了从初始节点到目标节点的最短路径，长度为 68。程序输出了两条搜索路径，分别是正向搜索路径和反向搜索路径。

```
(base) PS E:\code\Windows\Python\AIlab8> python BiBFS.py
Success! The length is 68
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1, 26), (1, 25), (2, 25), (3
, 25), (3, 26), (3, 27), (4, 27), (5, 27), (6, 27), (6, 26), (6, 25), (6, 24), (5, 24), (5, 23), (5, 2
2), (5, 21), (5, 20), (6, 20), (7, 20), (8, 20), (8, 21), (8, 22)]
[(8, 22), (8, 23), (8, 24), (8, 25), (8, 26), (8, 27), (9, 27), (10, 27), (11, 27), (12, 27), (13, 27)
, (14, 27), (15, 27), (15, 26), (15, 25), (15, 24), (15, 23), (15, 22), (15, 21), (15, 20), (15, 19),
(15, 18), (15, 17), (15, 16), (15, 15), (15, 14), (15, 13), (15, 12), (15, 11), (15, 10), (16, 10), (1
6, 9), (16, 8), (16, 7), (16, 6), (16, 5), (16, 4), (16, 3), (16, 2), (16, 1)]
```

图 9: 双向广度优先搜索结果

将路径可视化如图10所示。其中红色为正向搜索的路径，蓝色的为反向搜索的路径。

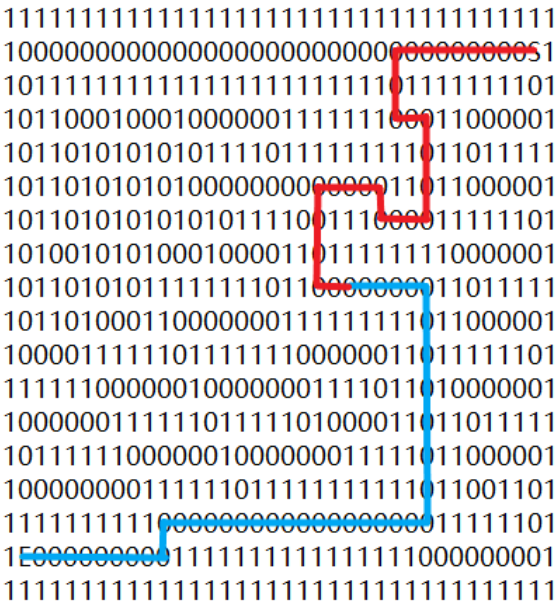

图 10: 结果可视化

4.4 A* 搜索结果

A* 搜索结果如图11所示，我们采用曼哈顿距离作为启发式函数，$D = 1$。可见程序找到了从初始节点到目标节点的最短路径，长度为 68。

```
(base) PS E:\code\Windows\Python\AIlab8> python A_star.py
Success! The length is 68
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1, 26), (1, 25), (2, 25), (3
, 25), (3, 26), (3, 27), (4, 27), (5, 27), (6, 27), (6, 26), (6, 25), (6, 24), (5, 24), (5, 23), (5, 2
2), (5, 21), (5, 20), (6, 20), (7, 20), (8, 20), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26),
 (8, 27), (9, 27), (10, 27), (11, 27), (12, 27), (13, 27), (14, 27), (15, 27), (15, 26), (15, 25), (15
, 24), (15, 23), (15, 22), (15, 21), (15, 20), (15, 19), (15, 18), (15, 17), (15, 16), (15, 15), (15,
14), (15, 13), (15, 12), (15, 11), (15, 10), (16, 10), (16, 9), (16, 8), (16, 7), (16, 6), (16, 5), (1
6, 4), (16, 3), (16, 2), (16, 1)]
```

图 11: A* 搜索结果

结果可视化与图7相同。

4.5 IDA* 搜索结果

IDA* 搜索结果如图12所示，我们采用曼哈顿距离作为启发式函数，$D = 1$。可见程序找到了从初始节点到目标节点的最短路径，长度为 68。

```
(base) PS E:\code\Windows\Python\AIlab8> python IDA_star.py
Success! The length is 68
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1, 26), (1, 25), (2, 25), (3
, 25), (3, 26), (3, 27), (4, 27), (5, 27), (6, 27), (6, 26), (6, 25), (6, 24), (5, 24), (5, 23), (5, 2
2), (5, 21), (5, 20), (6, 20), (7, 20), (8, 20), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26),
 (8, 27), (9, 27), (10, 27), (11, 27), (12, 27), (13, 27), (14, 27), (15, 27), (15, 26), (15, 25), (15
, 24), (15, 23), (15, 22), (15, 21), (15, 20), (15, 19), (15, 18), (15, 17), (15, 16), (15, 15), (15,
14), (15, 13), (15, 12), (15, 11), (15, 10), (16, 10), (16, 9), (16, 8), (16, 7), (16, 6), (16, 5), (1
6, 4), (16, 3), (16, 2), (16, 1)]
```

图 12: IDA* 搜索结果

结果可视化与图7相同。

4.6 算法性能对比与分析

4.6.1 完备性与最优性

由1.3和1.4可知，在理论上而言，一致代价搜索在路径成本不小于 0 时具有完备性和最优性；迭代加深搜索和双向广度优先搜索都具有完备性和最优性；当使用曼哈顿距离作为启发式函数时，该函数具有可采纳性和一致性，此时 A* 搜索和 IDA* 搜索具有完备性与最优性。

实验结果也印证了理论的正确性，五种算法都找到了迷宫的最短路径。

4.6.2 时间复杂度和空间复杂度

对于时间复杂度，我们将五种算法的输出全部移除，使用 python 的 timeit 模块测量五种算法单纯搜索的所需时间。为了进一步确认搜索时间，我们还在另外的程序中统计五种算法探索的节点数量。最终结果如表1所示：

	UCS	**IDS**	**BiBFS**	**A***	**IDA***
理论时间复杂度	$O(b^{1+[C^*/\epsilon]})$	$O(b^d)$	$O(b^{d/2})$	$O(b^{1+[C^*/\epsilon]})$	$O(b^d)$
执行 500 次时间 (sec)	4.1457	84.7733	10.6738	3.9116	42.2533
探索节点数	547	17093	344	414	3595

表 1: 算法时间性能分析

可以看到，无论是探索节点的数量还是执行时间，都与理论时间复杂度相近。五种算法中，IDS 耗时最长、探索节点数最多，而 IDA* 次之，这是因为两种算法都是基于不断迭代实现的，会有大量节点在不同限度的搜索下被重复搜索。从探索节点数上看，双向广度优先搜索数量最少，但执行时间并不占优势，我猜测这是因为两个搜索队列的不断切换，以及只能通过遍历的方式去判断节点是否被已被探索导致的。执行时间最短的是 UCS 和 A* 搜索，相差并不大，而它们的理论时间复杂度下界也一致。

对于空间复杂度，我们使用 python 中的 memory-profiler 测试程序的占用内存。同时，我们在另外的程序中统计各个算法在存储待探索节点时达到的最高数量。结果如表2所示。

	UCS	**IDS**	**BiBFS**	**A***	**IDA***
理论空间复杂度	$O(b^{1+[C^*/\epsilon]})$	$O(bd)$	$O(b^{d/2})$	$O(b^{1+[C^*/\epsilon]})$	$O(bd)$
内存测量占用 (MiB)	0.000	0.000	0.000	0.000	0.000
最多节点存储数	18	9	19	55	17

表 2: 算法空间性能分析

可能是我电脑有问题、程序运行不当，也有可能是数据量太小，memory-profiler 无法准确测出各个程序的内存增减情况，在程序运行期间显示每行代码的内存增量为 0.000 MiB。因此我们重点分析各个算法在存储待探索节点时达到的最高数量。可见，IDS 和 IDA* 由于采用了迭代的方式，存储节点数量远远小于其他算法。而 A* 搜索最多存储节点数远远高于其他算法，与理论空间复杂度不太符合。个人猜测可能是其搜索路径和 UCS 不太一样，真实原因还有待研究。

5 思考题

5.1 BFS/DFS 和双向搜索的优缺点和适用场景

对于广度优先搜索，优点有：

(1) 实现思路简单；

(2) 具有完备性和最优性，在特定情况下可以找到最优解；

缺点有：

(1) 运行效率低，会占用大量内存空间；

(2) 具有完备性，在特定情况下可以找到最优解；

对于深度优先搜索，优点有：

(1) 实现思路简单；

(2) 占用内存空间比 BFS 低；

缺点有：

(1) 不具有最优性，在特定情况下没有完备性；

(2) 在深度很大的情况下效率不高；

因此，当场景要求一定要找到解时可以考虑宽度优先搜索，而深度优先搜索可以应用在运行空间有限、不强调结果的场景中。

对于双向搜索，其相比普通 BFS 和 DFS 所耗时间更少、在使用 BFS 实现双向搜索时同样有完备性和最优性，但实现起来相对复杂、占用较多内存。因此在强调时间时可以采用双向搜索。

5.2 UCS 的优缺点和适用场景

一致代价搜索可以视为一种特殊的广度优先搜索。当所有的路径成本一样时，UCS 相当于 BFS。UCS 优点与 BFS 相一致，都具有完备性和最优性。但当路径成本存在为负的情况时，UCS 可能会有错误出现。此外，UCS 同样有占用内存空间过大的问题。因此，在内存要求不高、各个路径的成本有所不同且非负的情况下，可以使用一致代价搜索。

5.3 DLS/IDS 的优缺点和适用场景

深度受限搜索是一种特殊的深度优先搜索，其提前设置了搜索深度上限。优点有：

(1) 可以防止因为深度过大导致搜索时间过长；

缺点有：

(1) 不具有完备性和最优性，找到解与最优解的概率可能比 DFS 更小；

深度受限搜索适用于那些需要使用深度优先搜索、但强调运行时间不能过长的场景中。

迭代加深搜索通过多次调用深度受限搜索，解决深度优先搜索和广度优先搜索的问题。其优点有：

(1) 具有完备性和最优性，可以找到最优解；

(2) 占用内存空间低；

缺点有：

(1) 由于通过不断迭代搜索实现，一些节点会被不断重复搜索，搜索时间可能会比单纯地使用 BFS/DFS 长；

因此，迭代加深搜索适用于内存空间较小且需要找到最优解的场景中。

5.4 A* 搜索和 IDA* 搜索的优缺点和适用场景

A* 搜索使用了启发式函数。优点有：

(1) 当启发式函数具有可采纳性和一致性时，算法具有完备性和可采纳性；

(2) 在选择节点时通过估计的方式引入全局信息，理想情况下搜索用时更短；

缺点有：

(1) 算法性能与效果依赖于启发式函数的选择；

(2) 可能会占用大量内存空间；

因此，当我们有较好的启发式函数设计时，不妨考虑用 A* 搜索替换 BFS 和 UCS。

IDA* 搜索是将迭代加深搜索和 A* 搜索结合起来的方式。其优点是采用了启发式函数、会比无信息搜索更快；采用迭代的方式减少了内存的占用。同样，由于存在重复搜索同一节点、“时间换空间” 的情况，搜索时间可能比 A* 搜索长。故当内存较少且不强调时间时，可以考虑用 IDA* 搜索。

6 实验总结与感想

本次实验要求我们使用无信息搜索和启发式搜索解决迷宫问题。相比之前的实验，本次实验难度不算很大。在如何测量算法性能时我遇到了一些困难，最后查阅了网上的资料解决了问题。

希望我能顺利完成下一次实验。