无信息搜索&启发式搜索实验报告

18340040 冯大纬

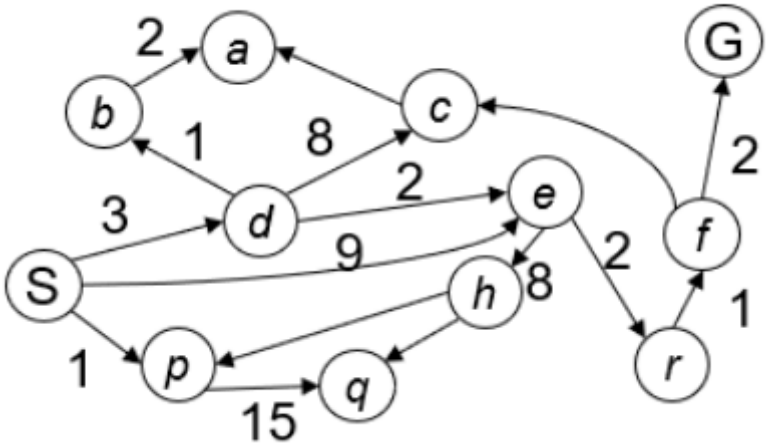
无信息搜索

一致代价搜索（UCS）

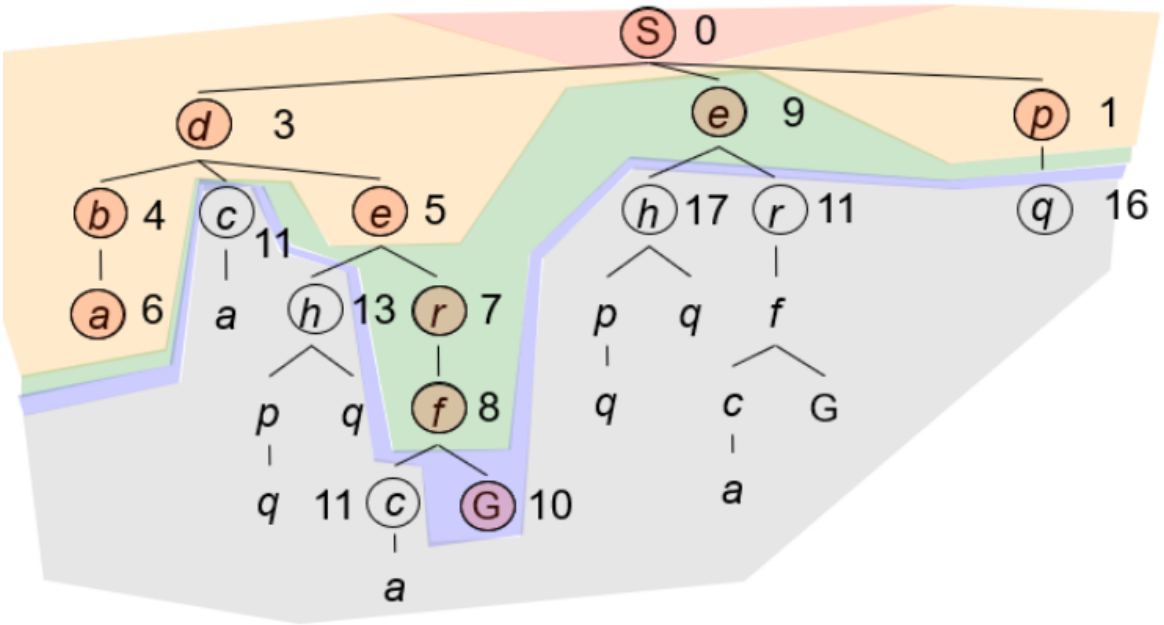
算法原理

一致代价搜索是在广度优先搜索上进行扩展的，基本原理是：一致代价搜索总是扩展路径消耗最小的节点N。N点的路径消耗等于前一节点N-1的路径消耗加上N-1到N节点的路径消耗。

比如在下面这张图中



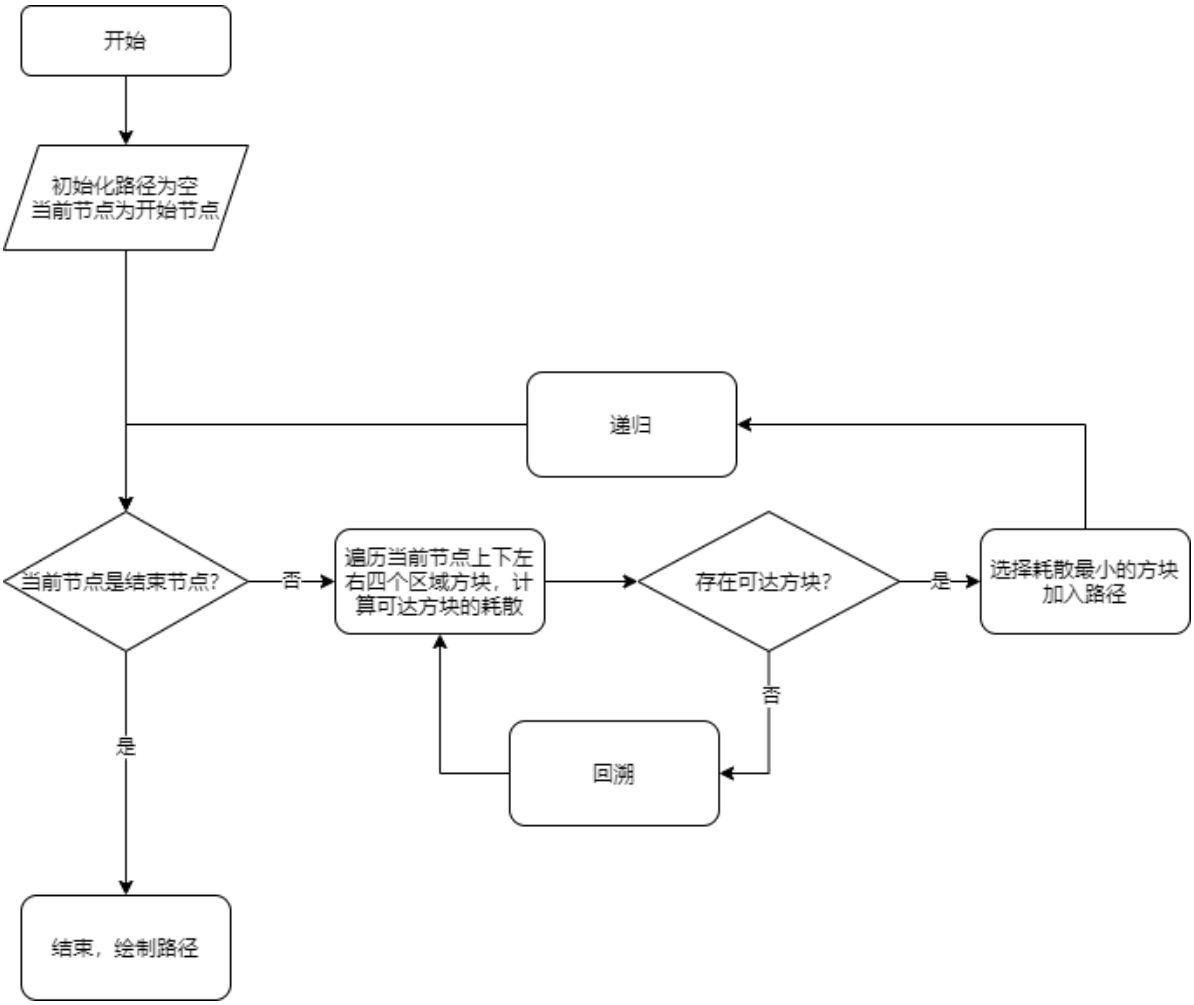
如果使用一致代价搜索，搜索过程的结构如下



可见，一致代价搜索的扩展过程应当是**S->p(1)->d(3)->b(4)->e(5)->a(6)->r(7)->f(8)->e(9)->G(10)**

伪代码与流程图

流程图



伪代码

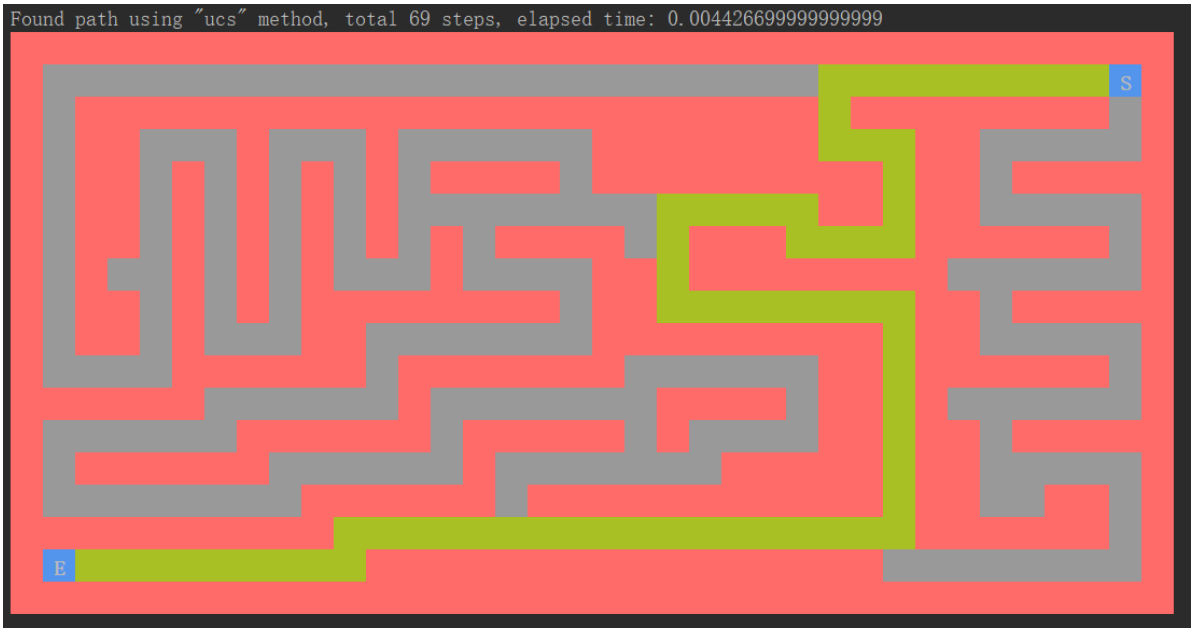
```
开始
    如果没有可达方块
        返回
    遍历可达方块
    计算方块耗散值
    选择具有最小耗散值的方块递归
    返回路径
结束
```

关键代码

```
def ucs(self, x, y, pre_c=None, p_queue=None, path=None, need_init=False):
    if need_init:
        path, p_queue, pre_c = [], [], 0  # 初始化路径
    if self.__is_success(x, y):
        path.append((x, y))  # 加入结束点
        return True, [(x, y)]
    path.append((x, y))  # 加入当前点
    for i in [-1, 1]:  # 遍历上下左右
        nextX, nextY = x + i, y
        if self.__is_legal(nextX, nextY) and (nextX, nextY) not in path:
```

```
            p_queue.append((self.__get_dist(nextX, nextY, x, y) + pre_c, (nextX,
nextY), (x, y)))  # 计算代价，加入队列
        nextX, nextY = x, y + i
        if self.__is_legal(nextX, nextY) and (nextX, nextY) not in path:
            p_queue.append((self.__get_dist(nextX, nextY, x, y) + pre_c, (nextX,
nextY), (x, y)))
    p_queue.sort(key=lambda k: k[0])  # 按代价排序
    suc, res = self.ucs(p_queue[0][1][0], p_queue[0][1][1], p_queue[0][0],
p_queue[1:], path[:])  # 递归
    if suc:
        for item in p_queue:
            if item[1] == res[-1]:
                res.append(item[2])
        return suc, res
    return False, path
```

实验结果



迭代加深搜索（IDS）

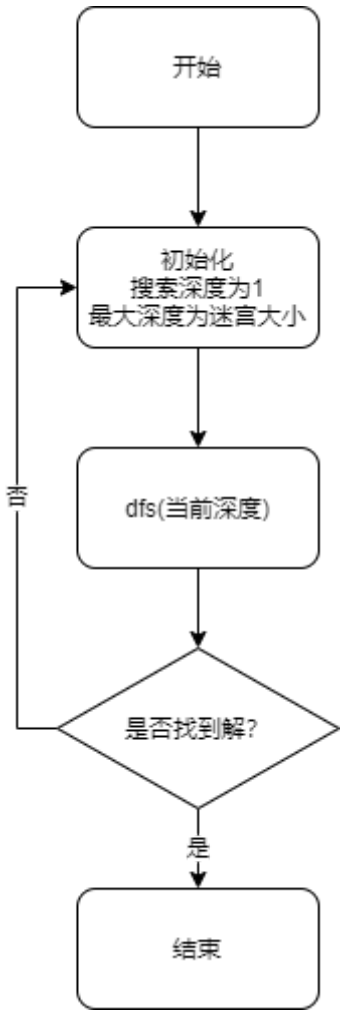
算法原理0

我对这个搜索的理解就是以BFS的思想写DFS。

具体来说就是，首先深度优先搜索k层，若没有找到可行解，再深度优先搜索k+1层，直到找到可行解为止。由于深度是从小到大逐渐增大的，所以当搜索到结果时可以保证搜索深度是最小的。这也是迭代加深搜索在一部分情况下可以代替广度优先搜索的原因。

伪代码与流程图

流程图



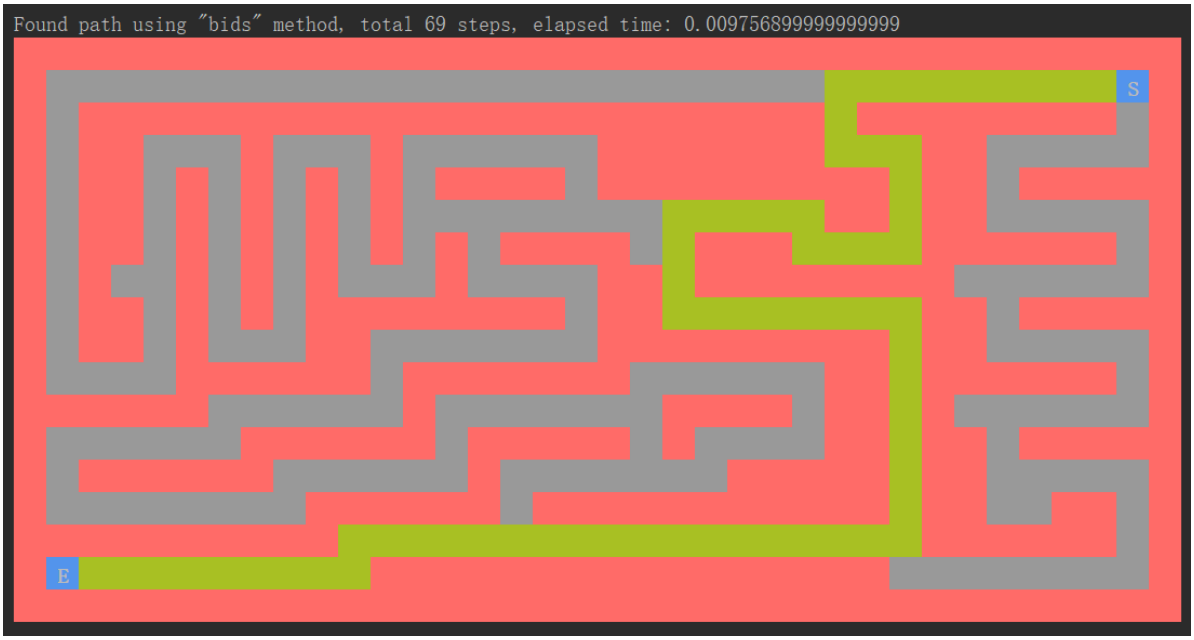
伪代码

```
开始
    初始化，当前深度 k，最大深度 max = width * height
    for i:=0 to max:
        suc=dfs(k,start,end)
        if suc:
            return 1
        else:
            continue
     return 0
结束
```

关键代码

```
def ids(self, x, y, path=None, depth=None, max_depth=None, need_init=False):
    if need_init:
        path, depth, max_depth = [], 0, self.maxX * self.maxY  # 初始化做大深度为长乘宽，因为最大步数不会大于这个值
    if self.__is_success(x, y):
        path.append((x, y))  # 如果到达终点，将终点加入路径
        return True, path
    for i in range(max_depth):
        suc, res = self.__dls(x, y, path[:], depth, i)  # 迭代加深搜索
        if suc:
            return suc, res
    return False, path
```

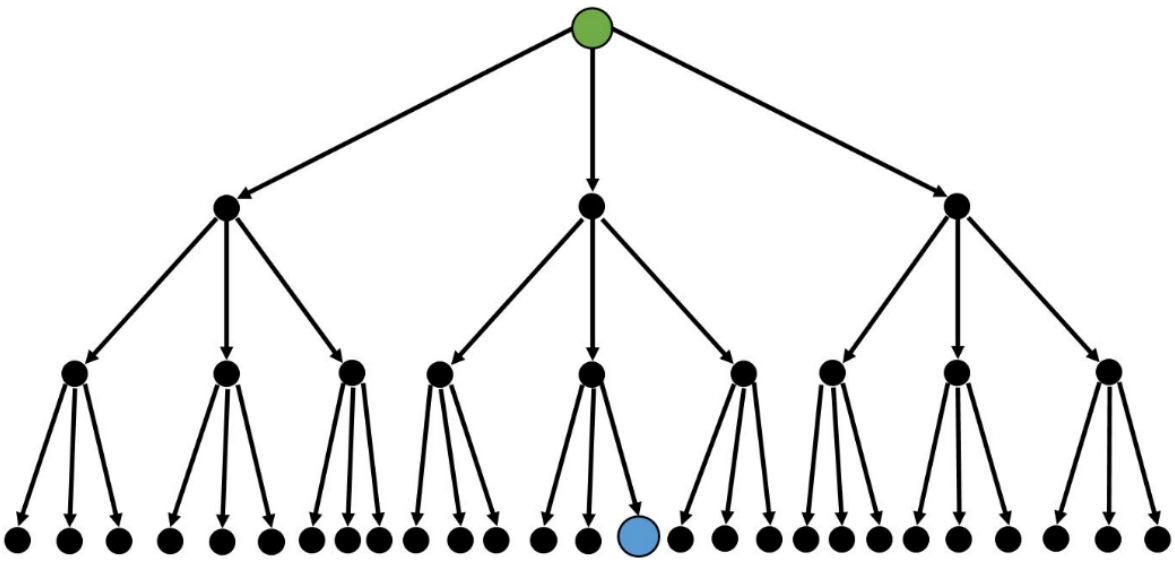
实验结果



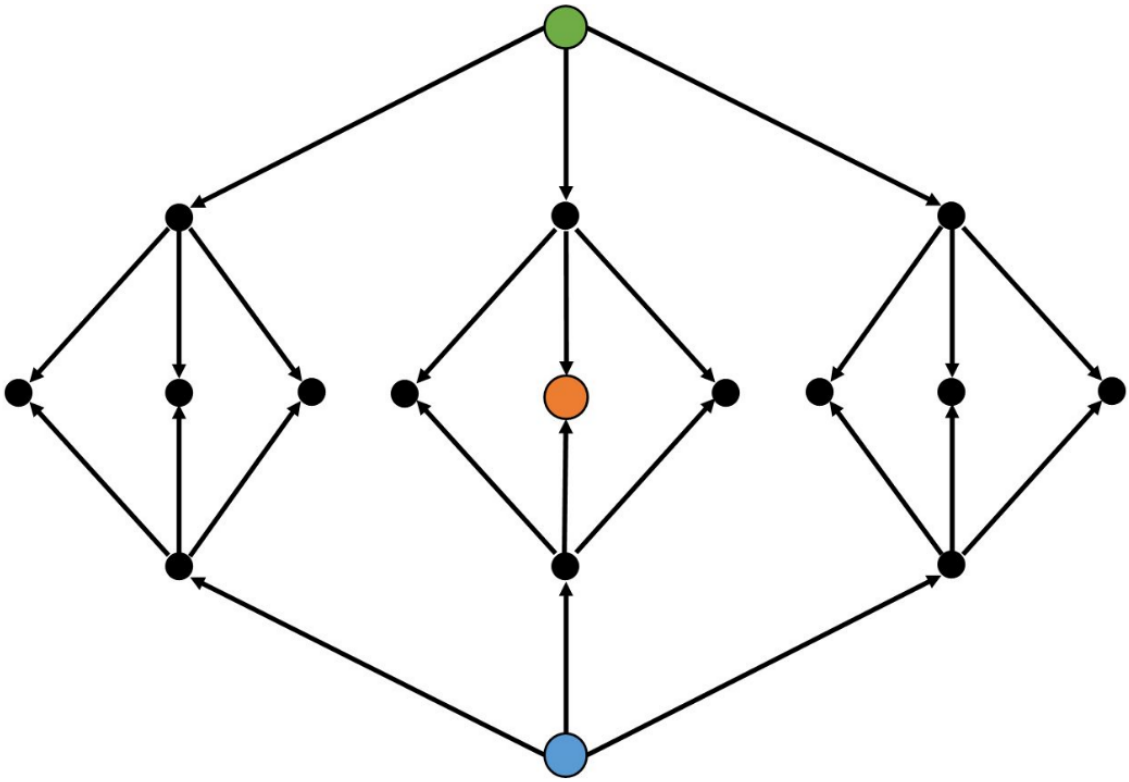
双向迭代加深搜索（BIDS）

算法原理

对于有些迷宫情景，如果我们用常规的搜索方法，从起点开始往下搜，那得到的解答树可能非常庞大，这样漫无目的的搜索就像大海捞针。比如下面这张图



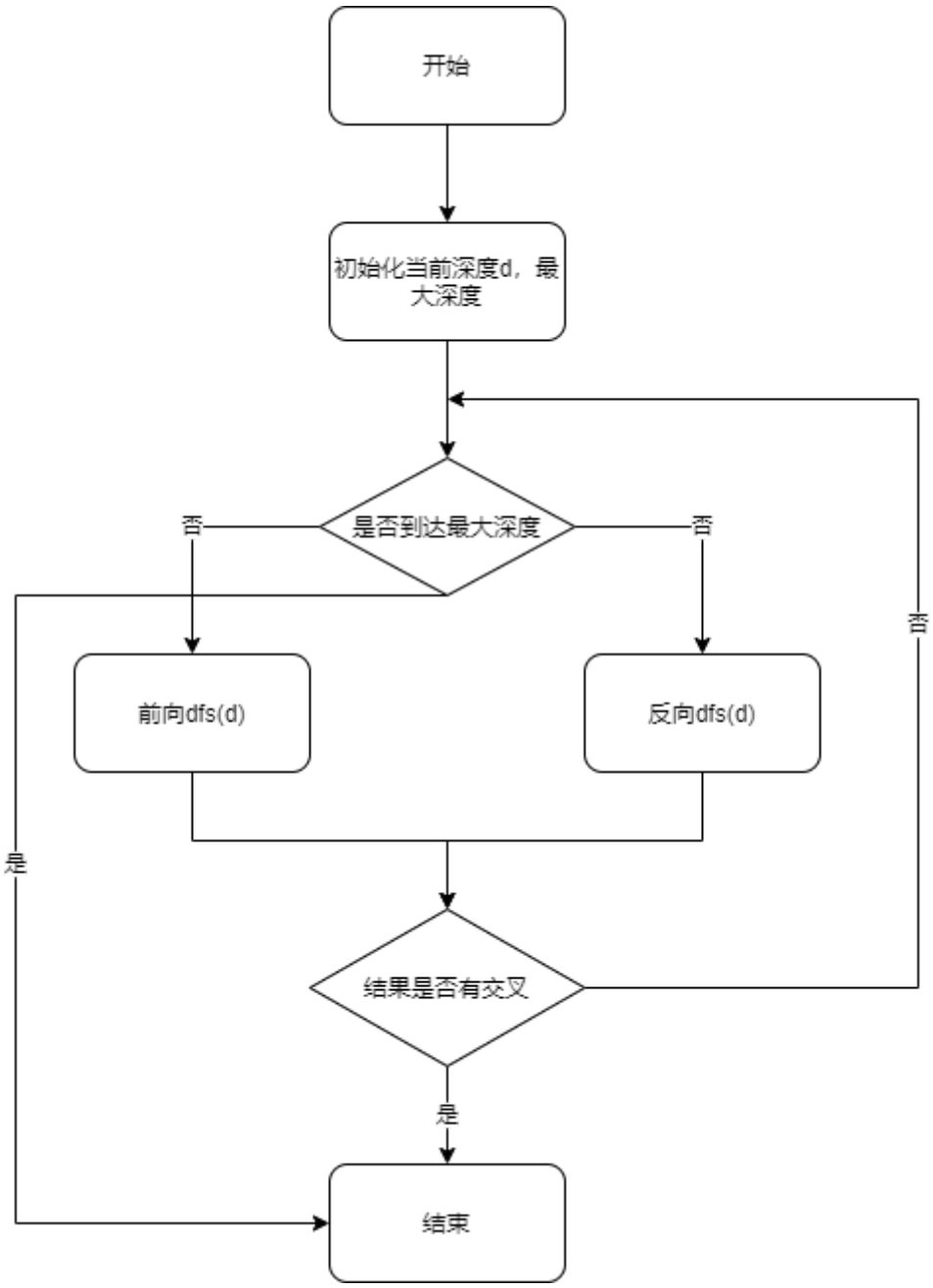
那么我们可以尝试从两边开始搜索



可见很快就能搜索到目标结果，如果原本的解答树规模是 a^n ，使用双向搜索后，规模立刻缩小到了约 $2a^{n/2}$，当 较n大时优化非常可观。

伪代码与流程图

流程图



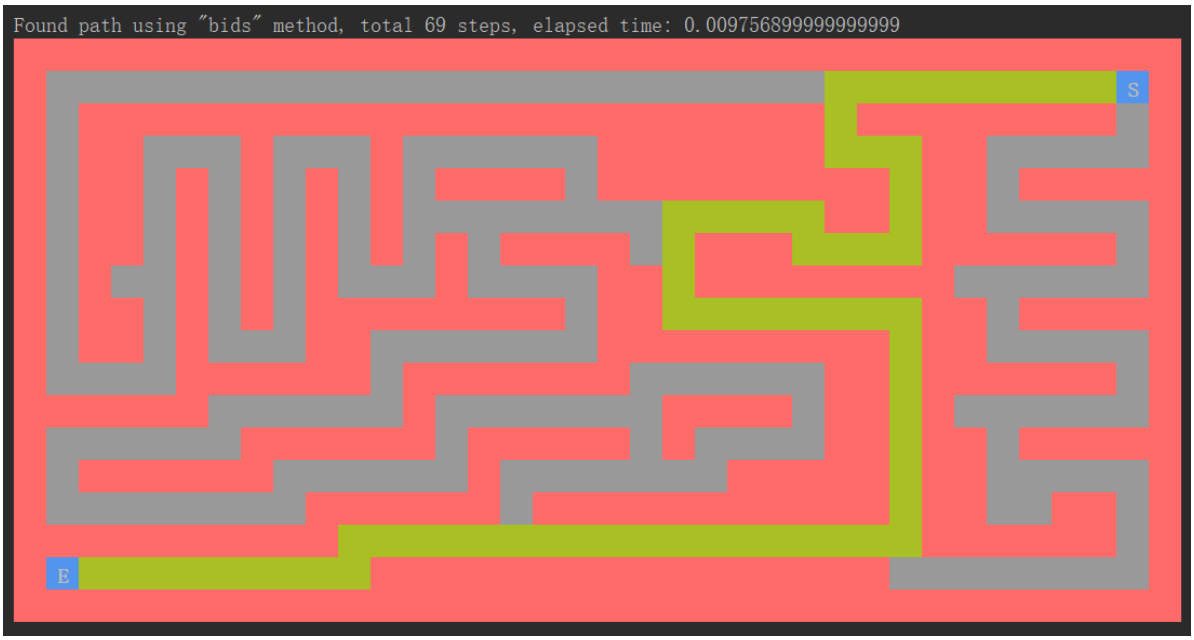
伪代码

```
初始化D，found
while (D <= MAXD / 2) // MAXD为题中要求的最大深度
{
    dfs(st, 0, 1); // st为起始状态
    if (found)
        return
    dfs(ed, 0, 2); // ed为终止状态
    if (found)
        return
    D++;
}
```

关键代码

```
def bids(self, x, y, path=None, depth=None, max_depth=None, need_init=False):
    if need_init:
        path, depth, max_depth = [], 0, self.maxX * self.maxY  # 初始化最大深度
    if self.__is_success(x, y):
        path.append((x, y))  # 成功，返回路径
        return True, path
    for i in range(max_depth):
        f_res = self.__f_dls(self.__startX, self.__startY, path[:], depth, i)  # 前向搜索
        self.__swap()  # 交换起点终点
        b_res = self.__b_dls(self.__startX, self.__startY, path[:], depth, i)  # 反向搜索
        self.__swap()
        for pt in f_res:
            if pt in b_res:  # 如果存在信息交叉
                path = f_res[pt].copy()
                path.append(pt)
                path.extend(list(reversed(b_res[pt].copy())))
                return True, path
    return False, 0
```

实验结果



启发式搜索

IDA*搜索

算法原理

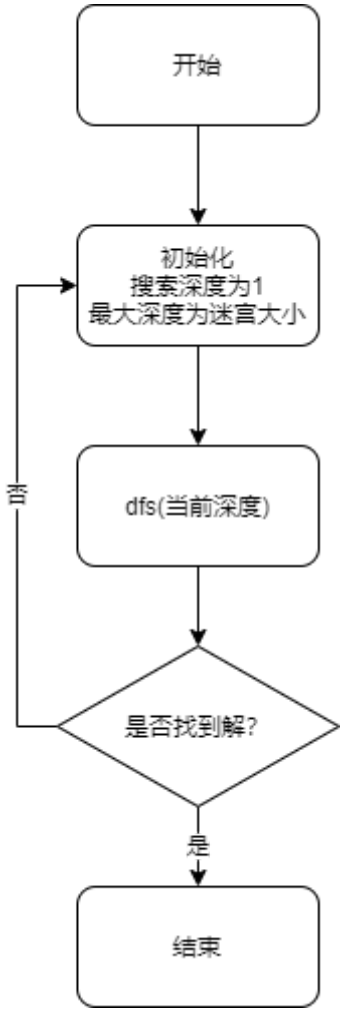
IDA* 算法是A* 算法和迭代加深算法的结合。

迭代加深算法是在dfs搜索算法的基础上逐步加深搜索的深度，它避免了广度优先搜索占用搜索空间太大的缺点，也减少了深度优先搜索的盲目性。它主要是在递归搜索函数的开头判断当前搜索的深度是否大于预定义的最大搜索深度，如果大于，就退出这一层的搜索，如果不大于，就继续进行搜索。这样最终获得的解必然是最优解。

IDA*的基本思路是：首先将初始状态结点的H值设为阈值maxH，然后进行深度优先搜索，搜索过程中忽略所有H值大于maxH的结点；如果没有找到解，则加大阈值maxH，再重复上述搜索，直到找到一个解。

伪代码与流程图

流程图



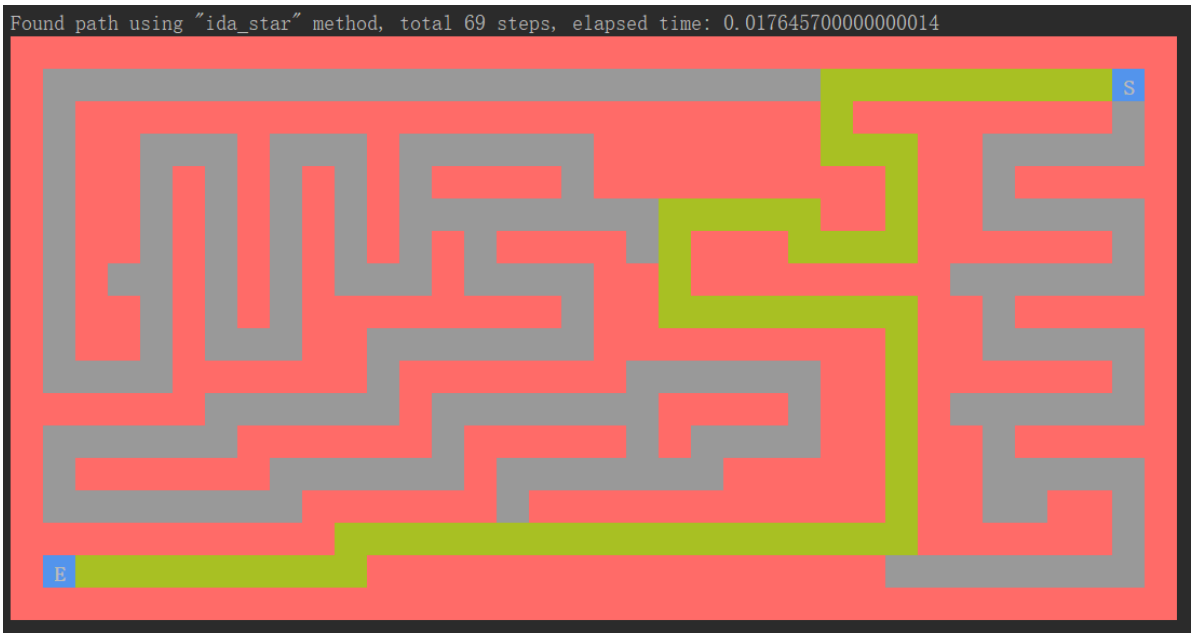
伪代码

```
初始化maxH
while True:
    启发式搜索(bound=maxH)
    if 找到解
        return
    else
        bound+=1
```

关键代码

```
def ida_star(self, x, y, need_init=False):
    cur_bound = int(self.__get_dist(self.__startX, self.__startY, self.__endX,
self.__endY))
    max_bound = cur_bound * 2
    for i in range(cur_bound, max_bound):
        suc, res = self.__ids_s(x, y, 0, i, need_init=True)
        if suc:
            return suc, res
    return False, 0
```

实验结果



结果分析

对每一种搜索策略运行100次，取平均值比较运行速度

方法	时间	是否找到最优解
UCS（一致代价搜索）	0.0030 s	是
IDS（迭代加深搜索）	0.0842 s	是
BIDS（双向迭代加深搜索）	0.0110 s	是
IDA*	0.0212 s	是

双向IDS在这个迷宫中可以达到普通IDS 8倍的速度

思考题

这些策略的优缺点是什么？它们分别适用于怎样的场景？

- UCS

  优点：保证最优性和完备性，对于无负代价的搜索等价于动态规划，保证局部最优性。
  缺点：需要维护较复杂的数据结构，且空间复杂度较大 ，为完整状态空间。

- IDS

  优点：保证最优性和完备性，空间复杂度优秀。
  缺点：不加优化会造成搜索冗余。

- BIDS

  优点：保证最优性和完备性，对于稠密图可以大量减少搜索的状态数 （状态数可以开根号）空间时间复杂度都很优秀 。
  缺点：不具有普适性，对于有些问题目标状态不唯一，无法进行双向搜索。

- IDA*

  优点：保证最优性和完备性，空间复杂度优秀，引入估价函数提升效率减少冗余搜索。
  缺点：只能进行 dfs ，如果进行 bfs 则和 A* 搜索没有差别。使用 DFS 有时不方便对状态记忆化提高时间效率。