中山大學
SUN YAT-SEN UNIVERSITY

# 人工智能实验

# Lab-9

姓名：刘斯宇
班级：计算机科学与技术 4 班
学号：17341110

## Contents

**1 一致代价搜索** **3**
1.1 算法原理 . . . . . 3
1.2 算法伪代码 . . . . . 3
1.3 实验代码 . . . . . 3
1.3.1 一致代价实现 . . . . . 3
1.4 实验结果 . . . . . 4

**2 双向搜索** **5**
2.1 算法原理 . . . . . 5
2.2 算法伪代码 . . . . . 5
2.3 实验代码 . . . . . 5
2.3.1 双向搜索的实现 . . . . . 5
2.4 实验结果 . . . . . 7

**3 迭代加深搜索** **8**
3.1 算法原理 . . . . . 8
3.2 算法伪代码 . . . . . 8
3.3 实验代码 . . . . . 8

3.3.1 迭代加深实现 . . . . . 8
3.4 实验结果 . . . . . 9

**4 $A^*$ 搜索算法** **10**
4.1 算法原理 . . . . . 10
4.2 算法伪代码 . . . . . 10
4.3 实验代码 . . . . . 11
4.3.1 $A^*$ 算法的实现 . . . . . 11
4.4 实验结果 . . . . . 12

**5 $IDA^*$ 搜索算法** **12**
5.1 算法原理 . . . . . 12
5.2 算法伪代码 . . . . . 12
5.3 实验代码 . . . . . 13
5.3.1 $IDA^*$ 算法的实现 . . . . . 13
5.4 实验结果 . . . . . 14

**6 各种算法比较** **15**
6.1 一致代价 . . . . . 15
6.1.1 完备性和最优性 . . . . . 15
6.1.2 时间复杂度 . . . . . 15
6.1.3 空间复杂度 . . . . . 16
6.1.4 优缺点 . . . . . 16
6.2 双向搜索 . . . . . 16
6.2.1 完备性和最优性 . . . . . 16
6.2.2 时间复杂度 . . . . . 16
6.2.3 空间复杂度 . . . . . 16
6.2.4 优缺点 . . . . . 16
6.3 迭代加深 . . . . . 17
6.3.1 完备性和最优性 . . . . . 17
6.3.2 时间复杂度 . . . . . 17
6.3.3 空间复杂度 . . . . . 17
6.3.4 优缺点 . . . . . 17
6.4 $A^*$ 算法 . . . . . 17
6.4.1 完备性和最优性 . . . . . 17
6.4.2 时间复杂度 . . . . . 17
6.4.3 空间复杂度 . . . . . 17
6.4.4 优缺点 . . . . . 17
6.5 $IDA^*$ 算法 . . . . . 18
6.5.1 完备性和最优性 . . . . . 18
6.5.2 时间复杂度 . . . . . 18
6.5.3 空间复杂度 . . . . . 18
6.5.4 优缺点 . . . . . 18

# 1 一致代价搜索

## 1.1 算法原理

一致代价搜索方式制定了一个路径消耗函数 g(n), 该函数描述根节点到 n 的路径消耗 (假设每个边上有相关的消耗值), 每次都扩展当前 g(n) 最小的边缘节点。

## 1.2 算法伪代码

```
function Uniform-cost-search():return a solution or a failure
  node <- a node with state = problem.initial.state ,path-cost = 0
  frontier <- a priority queue ordered by path-cost with node as element
  explored <- an empty set
  loop do
    if EMPTY(frontier) return failure
    node <- POP(frontier)
    if problem.SATISFY(node.state)
      return SOLUTION(node.state)
    add node.state to explored
    for each action in problem.ACTION(node.state) do
      child <- CHILD-STATE(node.state, action)
      if child.state nor in explored or frontier
        frontier <- INSERT(child.state)
```

## 1.3 实验代码

### 1.3.1 一致代价实现

实验思路: 每次选择从根结点到该点的代价最小的点进行 bfs。

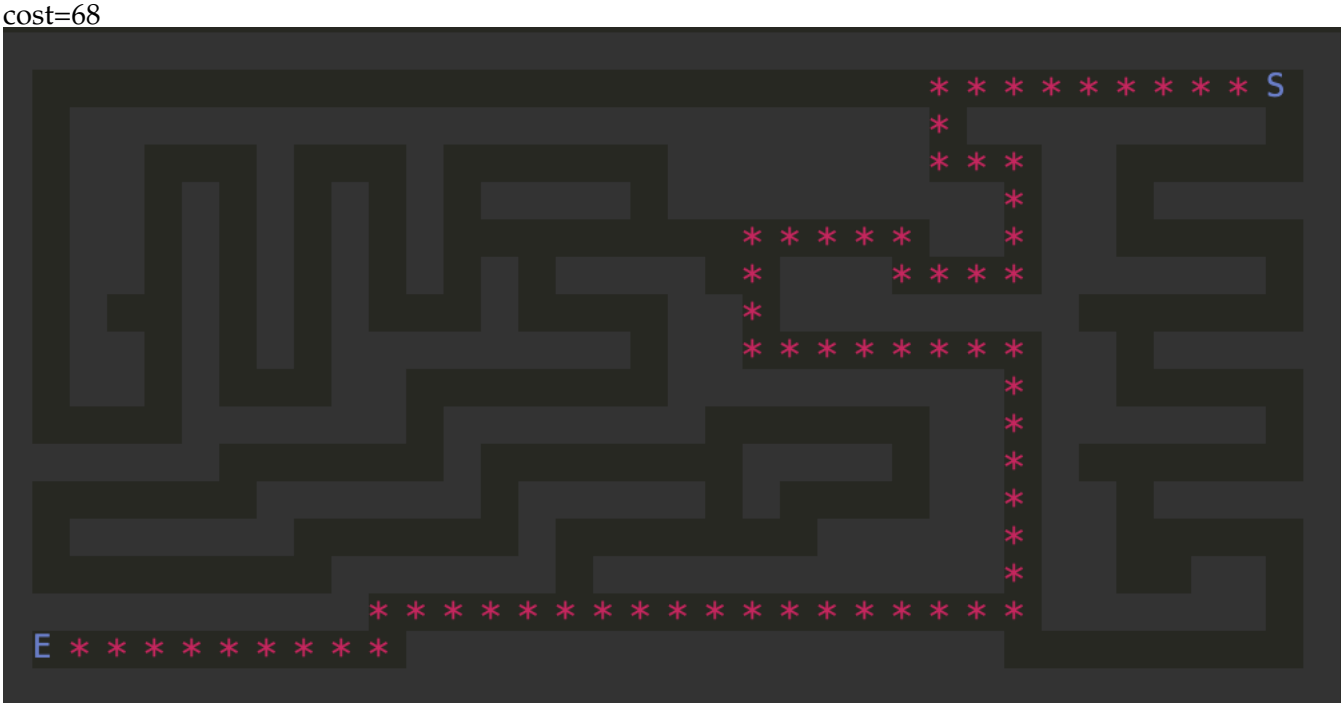
```
def Uniform_cost(Maze, end_point, distance, visited, before):
"""
参数意义:
Maze: list, 表示迷宫, 1表示障碍, 0表示可走的路线
end_point: tuple, 终点
distance: dic, 从根结点到该节点的代价
visited: list, 表示已经遍历的点
before: dic, 表示某个点的前继
"""
    now_point = list(distance.keys())[0]
    dis = list(distance.values())[0]
    if now_point == end_point:
        return dis
    distance.pop(now_point)
```

```
    if now_point in visited:
        return Uniform_cost(Maze, end_point, distance, visited, before)
    visited.append(now_point)
    for i in range(4):
        temp_line = now_point[0] + change[i][0]
        temp_col = now_point[1] + change[i][1]
        if temp_line < 0 or temp_line >= Maze.shape[
               0] or temp_col < 0 or temp_col >= Maze.shape[1] or Maze[
                   temp_line][temp_col] == '1' or (temp_line,
                                                   temp_col) in visited:
            continue
        distance[(temp_line, temp_col)] = dis + 1
        before[(temp_line, temp_col)] = now_point
    distance = dict(sorted(distance.items(), key=lambda x: x[1],
                           reverse=False))
    return Uniform_cost(Maze, end_point, distance, visited, before)
```

### 1.4 实验结果

其中红色的 * 表示的是路线

cost=68



# 2 双向搜索

## 2.1 算法原理

双向广度优先搜索算法是对广度优先算法的一种扩展。广度优先算法从起始节点以广度优先的顺序不断扩展，直到遇到目的节点；而双向广度优先算法从两个方向以广度优先的顺序同时扩展，一个是从起始节点开始扩展，另一个是从目的节点扩展，直到一个扩展队列中出现另外一个队列中已经扩展的节点，也就相当于两个扩展方向出现了交点，那么可以认为我们找到了一条路径。

## 2.2 算法伪代码

```
初始状态 和 目标状态 都知道，求初始状态到目标状态的最短距离；
利用两个队列，初始化时初始状态在1号队列里，目标状态在2号队列里，
并且记录这两个状态的层次都为0，然后分别执行如下操作：

a.若1号队列已空，则结束搜索，否则从1号队列逐个弹出层次为K(K >= 0)的状态；
    i. 如果该状态在2号队列扩展状态时已经扩展到过，
        那么最短距离为两个队列扩展状态的层次加和，结束搜索；
    ii. 否则和BFS一样扩展状态，放入1号队列，
        直到队列首元素的层次为K+1时执行b；
b.若2号队列已空，则结束搜索，否则从2号队列逐个弹出层次为K(K >= 0)的状态；
    i. 如果该状态在1号队列扩展状态时已经扩展到过，
        那么最短距离为两个队列扩展状态的层次加和，结束搜索；
    ii. 否则和BFS一样扩展状态，放入2号队列，
        直到队列首元素的层次为K+1时执行a；
```

## 2.3 实验代码

### 2.3.1 双向搜索的实现

```
def bi_search(Maze, now_start_point, now_end_point, start_visited,
    end_visited,start_before, end_before, turn, start_distance, end_distance):
"""
    参数的含义：
    Maze：list，迷宫
    now_start_point：tuple，从起点出发的当前的节点
    now_end_point：tuple，从终点出发的当前的节点
    start_visited：list，从起点出发所经过的节点
    end_visited：list，从终点出发所经过的节点
    start_before：dic，从起点出发的每个点的前继
    end_before：dic，从终点出发的每个点的前继
    turn：bool，表示该从起点那端出发还是终点那端出发
```

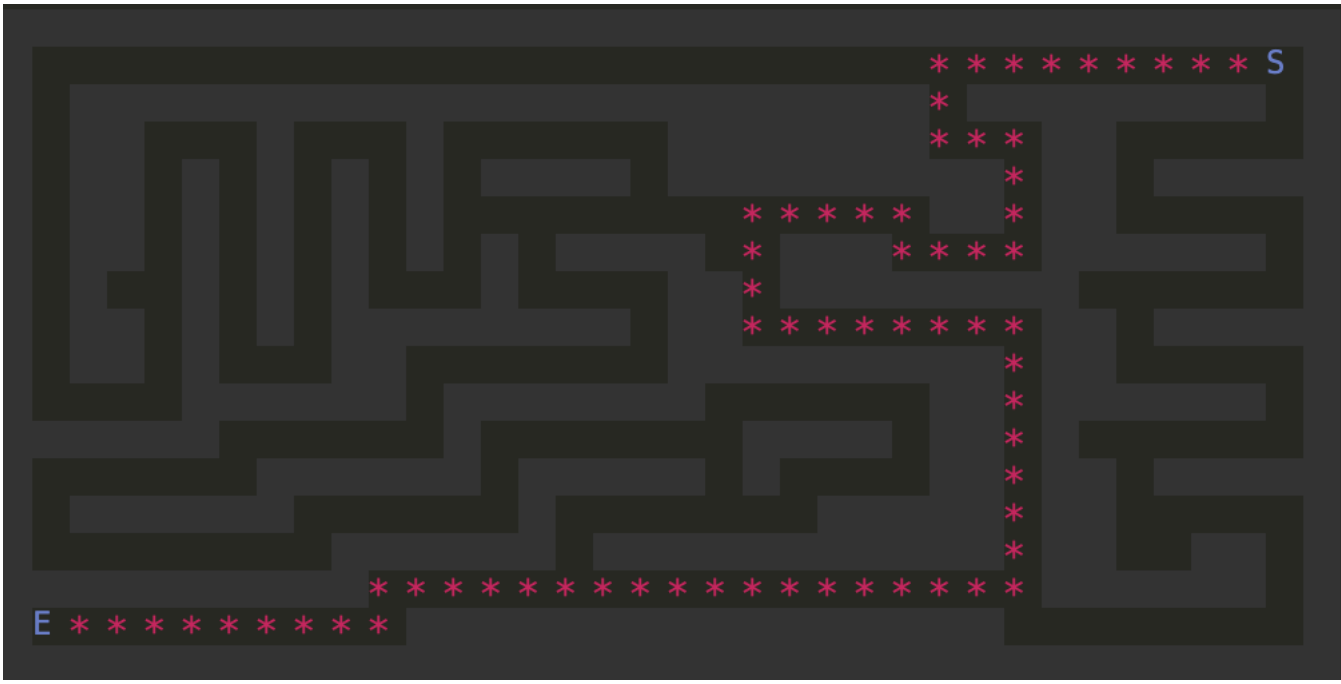
```
    start_distance: dic, 代价
    end_distance: dic, 代价
"""
if now_start_point == now_end_point or now_start_point in end_visited or
    now_end_point in start_visited:
if now_start_point == now_end_point:
    return now_start_point
elif now_start_point in end_visited:
    return now_start_point
elif now_end_point in start_visited:
    return now_end_point
if turn is True:
if len(start_distance) > 0:
    now_point = list(start_distance.keys())[0]
    dis = list(start_distance.values())[0]
    f.write(str(now_point) + '\n')
    start_distance.pop(now_point)
    start_visited.append(now_point)
    for i in range(4):
        temp_line = now_point[0] + change[i][0]
        temp_col = now_point[1] + change[i][1]
        if temp_line < 0 or temp_line >= Maze.shape[
               0] or temp_col < 0 or temp_col >= Maze.shape[
                   1] or Maze[temp_line][temp_col] == '1' or (
                       temp_line, temp_col) in start_visited:
            continue
        start_distance[(temp_line, temp_col)] = dis + 1
        start_before[(temp_line, temp_col)] = now_point
    start_distance = dict(
        sorted(start_distance.items(),
               key=lambda x: x[1],
               reverse=False))
    return bi_search(Maze, now_point, now_end_point, start_visited,
                     end_visited, start_before, end_before, not turn,
                     start_distance, end_distance)
else:
    return bi_search(Maze, now_start_point, now_end_point,
                     start_visited, end_visited, start_before,
                     end_before, not turn, start_distance,
                     end_distance)
else:
if len(end_distance) > 0:
    now_point = list(end_distance.keys())[0]
```

```
    dis = list(end_distance.values())[0]
    f.write(str(now_point) + '\n')
    end_distance.pop(now_point)
    end_visited.append(now_point)
    for i in range(4):
        temp_line = now_point[0] + change[i][0]
        temp_col = now_point[1] + change[i][1]
        if temp_line < 0 or temp_line >= Maze.shape[
               0] or temp_col < 0 or temp_col >= Maze.shape[
                   1] or Maze[temp_line][temp_col] == '1' or (
                       temp_line, temp_col) in end_visited:
            continue
        end_distance[(temp_line, temp_col)] = dis + 1
        end_before[(temp_line, temp_col)] = now_point
    end_distance = dict(
        sorted(end_distance.items(), key=lambda x: x[1],
              reverse=False))
    return bi_search(Maze, now_start_point, now_point, start_visited,
                     end_visited, start_before, end_before, not turn,
                     start_distance, end_distance)
else:
    return bi_search(Maze, now_start_point, now_end_point,
                     start_visited, end_visited, start_before,
                     end_before, not turn, start_distance,
                     end_distance)
```

## 2.4 实验结果

其中红色的 * 表示的是路线

cost=68




# 3 迭代加深搜索

## 3.1 算法原理

## 3.2 算法伪代码

```
def IDA_Star(STATE startState):
    maxDepth = 0
    while true:
        if( DFS(startState, 0, maxDepth) ):
            return
        maxDepth = maxDepth + 1
```

## 3.3 实验代码

### 3.3.1 迭代加深实现

实验思路: 每次选择从根结点到该点的代价最小的点进行 bfs。

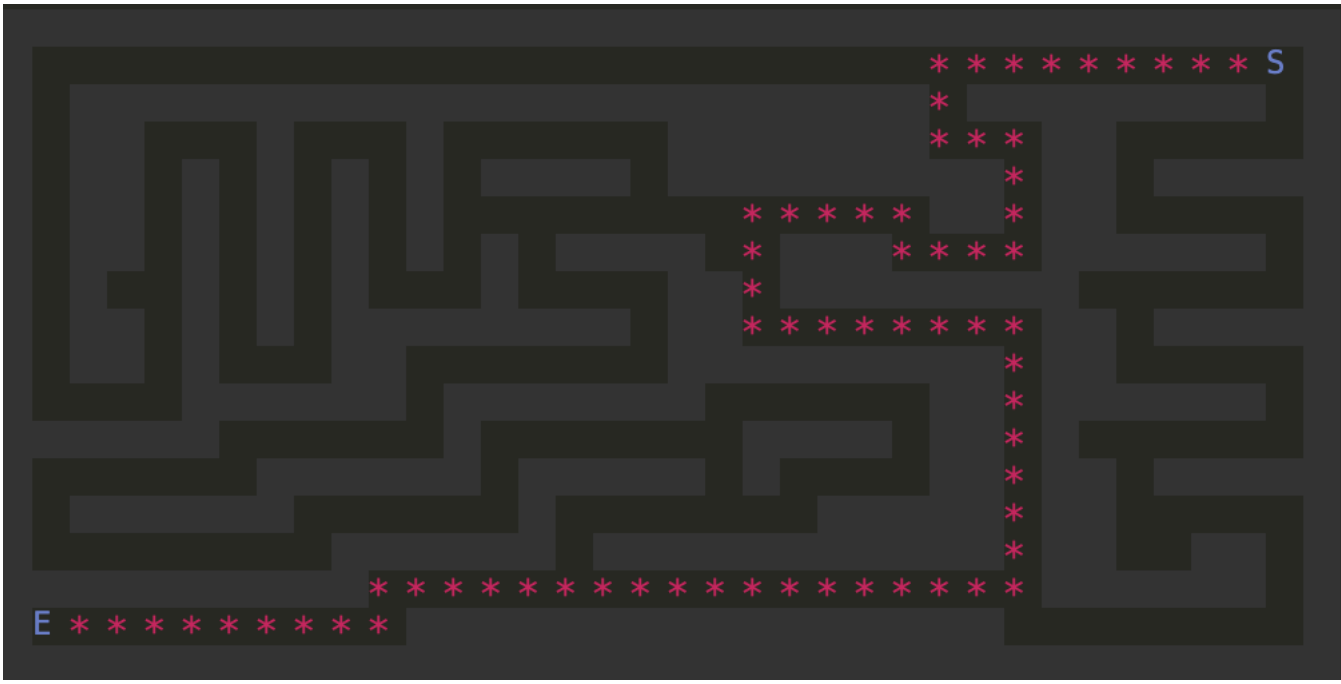
```
def IDDFS(Maze, end_point, visited, before, num, now_point):
"""
参数的含义:
Maze: list, 迷宫
end_point: tuple, 终点
visited: list, 表示已经遍历的点
before: dic, 表示某个点的前继
num:剩下遍历的深度
```

```
now_point: tuple,当前正在遍历的点
"""
  global flag
  global res_before
  if flag:
      return
  x = now_point[0]
  y = now_point[1]
  if x < 0 or x >= Maze.shape[0] or y < 0 or y >= Maze.shape[1] or num < 0:
      #see_path(Maze, visited)
      return
  if now_point == end_point:
      flag = True
      res_before = before
      return
  for i in range(4):
      temp_line = now_point[0] + change[i][0]
      temp_col = now_point[1] + change[i][1]
      if temp_line < 0 or temp_line >= Maze.shape[
             0] or temp_col < 0 or temp_col >= Maze.shape[1] or Maze[
                 temp_line][temp_col] == '1' or (temp_line,
                                                 temp_col) in visited:
          continue
      point = (temp_line, temp_col)
      before[point] = now_point
      visited.append(point)
      visited_copy = visited.copy()
      before_copy = before.copy()
      IDDFS(Maze, end_point, visited_copy, before_copy, num-1, point)
      visited.remove(point)
      before.pop(point)
```

### 3.4 实验结果

其中红色的 * 表示的是路线

cost=68




# 4 $A^*$ 搜索算法

## 4.1 算法原理

A* 算法通过下面这个函数来计算每个节点的优先级：

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

- f(n) 是节点 n 的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。
- g(n) 是节点 n 距离起点的代价。
- h(n) 是节点 n 距离终点的预计代价，这也就是 A* 算法的启发函数。关于启发函数我们在下面详细讲解。

A* 算法在运算过程中，每次从优先队列中选取 f(n) 值最小（优先级最高）的节点作为下一个待遍历的节点。另外，A* 算法使用两个集合来表示待遍历的节点，与已经遍历过的节点，这通常称之为 open_set 和 close_set。

## 4.2 算法伪代码

```
* 初始化open_set和close_set;
* 将起点加入open_set中，并设置优先级为0（优先级最高）；
* 如果open_set不为空，则从open_set中选取优先级最高的节点n:
    * 如果节点n为终点，则:
        * 从终点开始逐步追踪parent节点，一直达到起点;
        * 返回找到的结果路径，算法结束;
    * 如果节点n不是终点，则:
```

```
        * 将节点n从open_set中删除，并加入close_set中；
        * 遍历节点n所有的邻近节点：
            * 如果邻近节点m在close_set中，则：
                * 如果此时的g(n)比之前的小，更新g(n)
                * 否则，跳过
            * 如果邻近节点m也不在open_set中，则：
                * 设置节点m的parent为节点n
                * 计算节点m的优先级
                * 将节点m加入open_set中
```

## 4.3 实验代码

### 4.3.1 $A^*$ 算法的实现

实验思路: 每次选择从根结点到该点的代价最小的点进行 bfs。

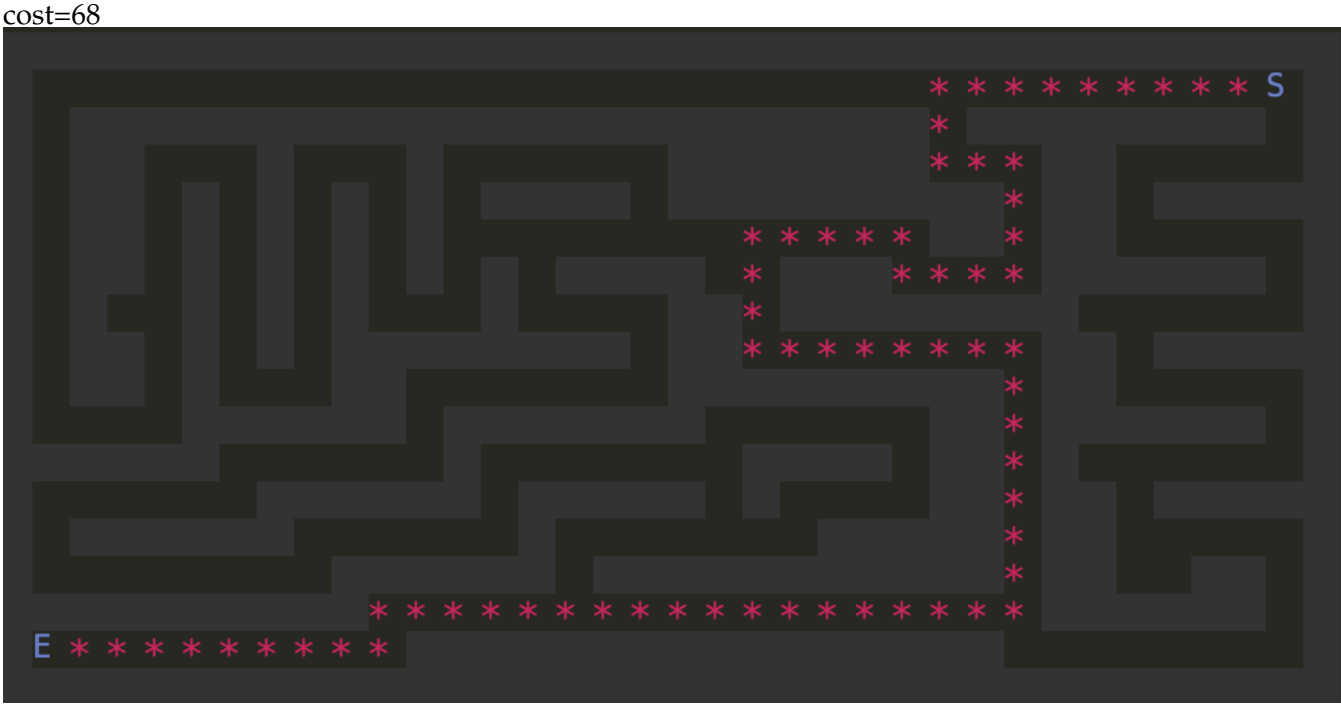
```
  def Astar(Maze, end_point, distance, visited, real_distance, before):
"""
参数的含义:
Maze: list, 迷宫
end_point: tuple, 终点
distance: dic, 从根结点到该节点再到终点的预计代价
visited: list, 表示已经遍历的点
real_distance:从根节点到当前节点的代价
before: dic, 表示某个点的前继
"""
  now_point = list(distance.keys())[0]
  dis = real_distance[now_point]
  if now_point == end_point:
      return real_distance[now_point]
  distance.pop(now_point)
  if now_point in visited:
      return Astar(Maze, end_point, distance, visited, real_distance, before)
  visited.append(now_point)
  for i in range(4):
      temp_line = now_point[0] + change[i][0]
      temp_col = now_point[1] + change[i][1]
      if temp_line < 0 or temp_line >= Maze.shape[
             0] or temp_col < 0 or temp_col >= Maze.shape[1] or Maze[
                 temp_line][temp_col] == '1' or (temp_line,
                                                 temp_col) in visited:
          continue
      before[(temp_line, temp_col)] = now_point
```

```
28      distance[(temp_line,
29               temp_col)] = dis + 1 + abs(temp_line -
30                                          end_point[0]) + abs(temp_col -
31                                                              end_point[1])
32      real_distance[(temp_line, temp_col)] = dis + 1
33  distance = dict(sorted(distance.items(), key=lambda x: x[1],
34                         reverse=False))
35  return Astar(Maze, end_point, distance, visited, real_distance, before)
```

### 4.4 实验结果

其中红色的 * 表示的是路线

cost=68




## 5 $IDA^*$ 搜索算法

### 5.1 算法原理

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。带启发式的有限制的深度优先搜索，本质是在启发式限制下以不同的深度进行 dfs。在稀疏的有向图中深度优先效果往往优于广度优先，所以会好于 A* 算法，然而如果是棋盘类稠密问题，应该是 A* 更占优。由于不再采用动态规划的方法，内存占用少。

### 5.2 算法伪代码

```
1  function IDA_search(node, g, bound):
```

```
  // node当前节点
  // g当前节点的路径消耗值
  // bound当前搜索的一个界限值
  f = g + h(node)
  if f > bound:
      return f
  if node == B:
      return FOUND
  min = r
  for succ in successors(node) do:
      t = search(succ, g + cost(node, succ), bound)
  if t == FOUND:
      return FOUND
  if t < min:
      min = t
  return min

bound = h(A)
while(True):
  t = IDA_search(root, 0, bound)
  if t == FOUND:
      return bound
  if t = :
      return NOT_FOUND
  bound := t
```

## 5.3 实验代码

### 5.3.1 $IDA^*$ 算法的实现

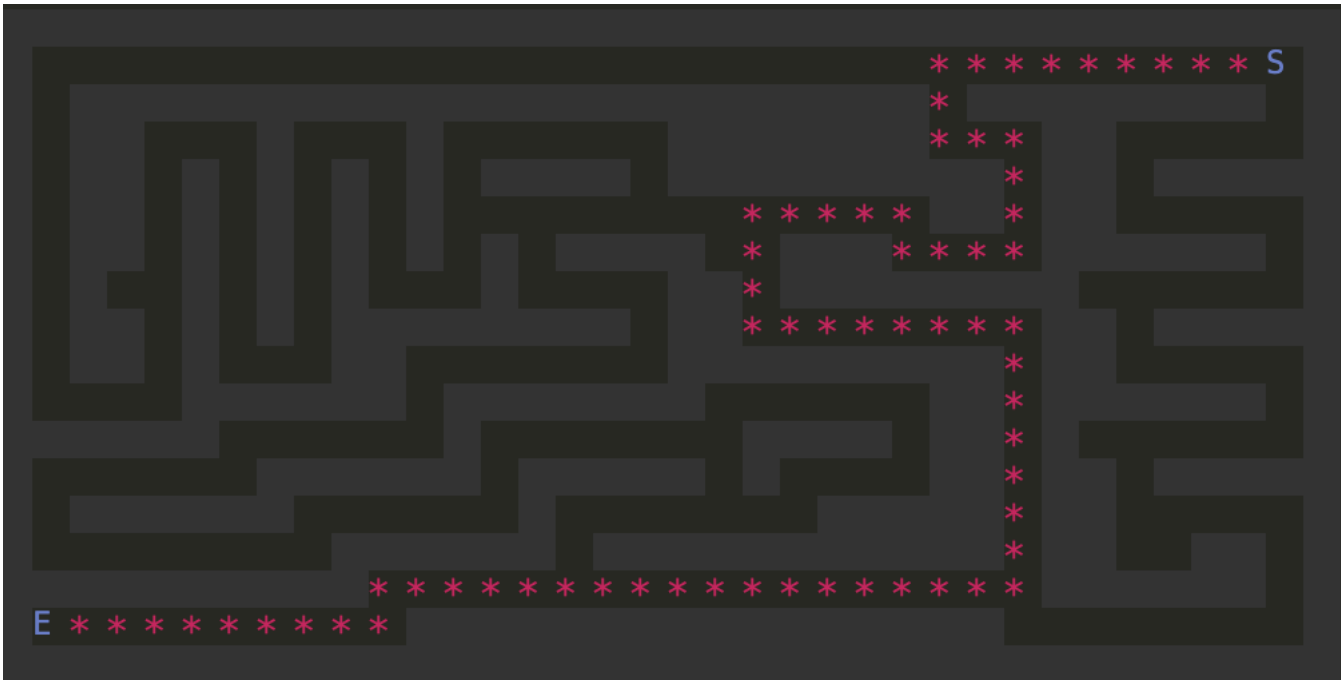
实验思路: 每次选择从根结点到该点的代价最小的点进行 bfs。

```
  def IDAstar(Maze, end_point, before, MAX_H, now_point, nowcost,
      estimatecost):
    global flag
    global res_before
    if flag:
        return
    #see_path(Maze, list(before.keys())+list(before.values()))
    #print(nowcost, now_point, end_point)
    x = now_point[0]
    y = now_point[1]
    if estimatecost > MAX_H or x < 0 or x >= Maze.shape[
```

```
            0] or y < 0 or y >= Maze.shape[1]:
        return
    if now_point == end_point:
        flag = True
        res_before = before
        return
    estimate = {}
    for i in range(4):
        temp_line = now_point[0] + change[i][0]
        temp_col = now_point[1] + change[i][1]
        if temp_line < 0 or temp_line >= Maze.shape[
                0] or temp_col < 0 or temp_col >= Maze.shape[1] or Maze[
                    temp_line][temp_col] == '1' or (temp_line,
                                                    temp_col) in before.keys():
            continue
        estimate[(temp_line, temp_col)] = nowcost + 1 + abs(
            temp_line - end_point[0]) + abs(temp_col - end_point[1])
    if len(estimate) == 0:
        return
    estimate = dict(sorted(estimate.items(), key=lambda item: item[1]))
    for item in estimate:
        point = item
        before[point] = now_point
        before_copy = before.copy()
        IDAstar(Maze, end_point, before_copy, MAX_H, point, nowcost + 1,
                estimate[item])
        before.pop(point)
```

### 5.4 实验结果

其中红色的 * 表示的是路线




# 6 各种算法比较

我们所有的算法都假定根结点算第 0 层。最大的子节点数目为 b。

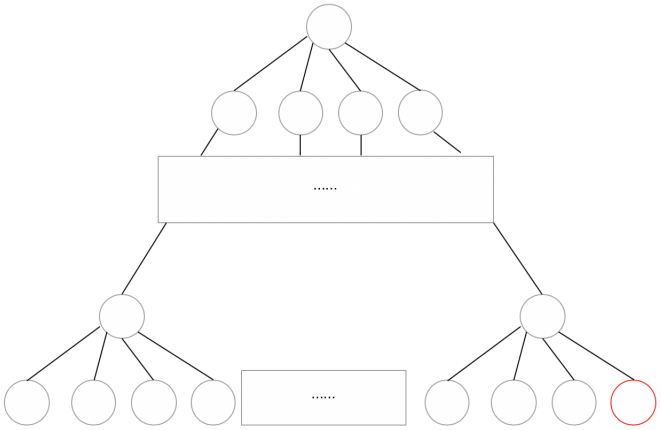
## 6.1 一致代价

设一致代价搜索中每一步的代价的最小值为 a，从根结点到目标节点的总代价为 C。

### 6.1.1 完备性和最优性

如果分支是有限并且代价都是非负的话，那么是具备完备性的，因为一致代价是 BFS 的优化版，同时也是具有最优性的，因为一致代价每次都是找的最小的节点。

### 6.1.2 时间复杂度



如图，我们假定红色的节点是目标节点，那么时间复杂度为因为要把红色节点的之前的所有的节点都便利才能遍历到红色节点，那么层数最大为 $[\frac{C}{a}]$, 其中 [] 为取整符号。那么时间复杂度为 $O(b^0 + b^1 + \cdots + b^{[\frac{C}{a}]}) = O(b^{[\frac{C}{a}]+1})$

#### 6.1.3 空间复杂度

同样的，还是上面的图，因为一致代价搜索是先存然后再遍历，那么一致代价搜索的空间复杂度为 $O(b(b^{[\frac{C}{a}]} - 1)) = O(b^{[\frac{C}{a}]+1})$

#### 6.1.4 优缺点

- 优点
  比 BFS 改进了，肯定能得到最优解
- 缺点
  时间复杂度和空间复杂度比较高
- 适用情况
  问题的规模比较小，对时间和空间不是很在意的情况下。

### 6.2 双向搜索

#### 6.2.1 完备性和最优性

如果子节点是有限的，并且双边都使用宽度优先，那么是一定能够找到最优解的，所以是具有完备性的，如果每一步的代价都是一样的，并且两边都使用 BFS，那么是一定能找到最优解的，所以具有最优性。

#### 6.2.2 时间复杂度

我们假设，单向 BFS 需要搜索 d 层才能到达终点，在每个层需要进行的判断量, 那么，单 BFS 的运算量为：$b^d$。如果换成双 BFS，那么最好的情况下前后各搜索 $\frac{d}{2}$ 层，那么总的运算量为：$2 * (b^{\frac{d}{2}+1})$。显然当 b 比较大时，在运算量上不仅仅不仅仅是减半那么简单。所以双向 BFS 能很好的降低复杂度。

#### 6.2.3 空间复杂度

同理复杂度，也是一样的算法，为 2 个空间之和，为 $2 * (b^{\frac{d}{2}+1})$

#### 6.2.4 优缺点

- 优点
  时间和空间复杂度比较低
- 缺点
  最优解的限制比较高，并且需要知道目标节点。
- 适用情况
  已知目标节点，单步代价相同且双向都使用宽度优先搜索。

## 6.3 迭代加深

### 6.3.1 完备性和最优性

如果子节点是有限的，那么一定能够找到解，所以是具有完备性的，如果每一步的代价都是一样的，那么是具有最优性的，因为是按照 BFS 的思路来写 DFS。

### 6.3.2 时间复杂度

还是一样，按照上面的图，找到红色的节点需要遍历的所有的节点数目是 $O(b^d)$

### 6.3.3 空间复杂度

因为需要存兄弟节点，所以我们假定最长的路径为 $l$，那么空间复杂度为 $bl$

### 6.3.4 优缺点

- 优点
  空间复杂度比较低
- 缺点
  时间复杂度比较高
- 适用情况
  对空间要求比较高的情况下。

## 6.4 $A^*$ 算法

### 6.4.1 完备性和最优性

如果子节点是有限的，那么一定能够找到解，所以是具有完备性的和最优性。

### 6.4.2 时间复杂度

比一致代价搜索的时间复杂度更低，有时候能达到线性。

### 6.4.3 空间复杂度

具有指数级别的空间复杂度，因为其要存的内容跟一致代价搜索的很像。

### 6.4.4 优缺点

- 优点
  时间复杂度比较低
- 缺点
  空间复杂度比较高，该算法一般要使用大量的空间用于存储已搜索过的中间状态，防止重复搜索。需要找到启发式函数
- 适用情况
  能够找到启发式的函数

## 6.5 $IDA^*$ 算法

### 6.5.1 完备性和最优性

如果子节点是有限的，那么一定能够找到解，所以是具有完备性的和最优性。

### 6.5.2 时间复杂度

时间复杂度比较高，因为是 DFS 所以会存在重复搜索。

### 6.5.3 空间复杂度

空间复杂度低，因为不需要维护 open_set 和 close_set.

### 6.5.4 优缺点

- 优点
  空间复杂度比较低，IDA* 要比 A* 方便，因为不需要保存结点，不需要判重复，也不需要根据 H 值对结点排序，占用空间小。
- 缺点
  时间复杂度高，该算法一般要使用大量的空间用于存储已搜索过的中间状态，防止重复搜索。需要找到启发式函数
- 适用情况
  能够找到启发式的函数，对空间复杂度要求比较高的情况下。