中山大學 SUN YAT-SEN UNIVERSITY **实验报告**

学院：数据科学与计算机学院　　专业：计算机科学与技术　　科目：人工智能
学号：17341213　　姓名：郑康泽

无信息搜索 & 启发式搜索

一. 算法原理

1. 一致代价搜索：

1) 原理：

一致代价搜索 UCS 是宽度优先 BFS 的加强版，每次迭代，先将队列中的代价最小的节点取出，检查是否为目标节点，如果是，停止迭代，否则将相邻的未被扩展过的节点加入到队列中，继续迭代。

2) 完备性：

只要目标节点是可达的，那么它就有一定的成本，那么给定了成本，并且在一致代价搜索中，所有成本较低的路径都会在成本高的路径之前被扩展，所以路径数量就是有限的，那么就可以在有限的步骤中找到该成本的所有路径，其中就包括到达目标节点的路径。所以一致代价搜索是完备的。

3) 最优性：

在一致代价搜索中，成本较低的路径会在成本高的路径之前被扩展，又由于有完备性，所以代价最小的路径一定能被找到。

4) 时间复杂度：

对于 BFS，并且目标检测应用于节点被扩展时，假设 d 为最优解的路径长度，b 为一个节点最多的后继节点个数，那么时间复杂度为：

$$1 + b + b^2 + \cdots + b(b^d - 1) = O(b^{d+1})$$

同理，对于 UCS，假设最优成本是 C^*，每个节点被扩展的最低成本是 s>0，那么相当于至多需要 C^*/s 步就能找到最优解，所以 USC 的时间复杂度为：

$$O(b^{C^*/s+1})$$

5) 空间复杂度：

同上，BFS 的空间复杂度为：

$$b(b^d - 1) = O(b^{d+1})$$

所以 USC 的空间复杂度为：

$$O(b^{C^*/s+1})$$

2. IDA*搜索：

1) 原理：

引入预估函数$f(n) = g(n) + h(n)$，实际代价$g(n)$表示从起点到n的代价，启发式函数$h(n)$表示从n到终点的估计代价。以下说的$h(n)$都是单调的，即$h(n) \leq cost(n, n') + h(n')$，这样能保证最优性。

IDA*是迭代加深搜索的加强版，迭代加深搜索限制的是探索的深度，而 IDA*限制的预估函数$f(n)$。每次给定一个阈值 bound，然后开始 DFS，当 DFS 返回信息表示受到阈值限制无法继续 DFS，我们就将阈值调大，增加的值为上次 DFS 时超过阈值 bound 的最小值，然后继续 DFS，直至找到目标节点或者所有的路径都找过但仍然

无法到达目标节点。

2) 完备性：

对于可达的目标节点，它的$f(n)$是有限的，那么只要当$bound \geq f(n)$时，进行 DFS 时，就能找到$f(n) \leq bound$的所有路径，其中就包括到达目标节点的路径。

3) 最优性：

在$f(n) \leq bound$的并且能到达目标节点的所有路径，都会在 DFS 中搜索到，只需要在每次到达目标节点的时候更新当前的最优路径即可。不需要担心$f(n) \geq bound$会不会存在一条更短的路径，因为$h(n)$的单调性保证了先遍历的节点即预估函数$f(n)$小的实际代价$g(n)$也小。

4) 时间复杂度：

对于 DFS，假设 d 为搜索的最长路径的长度，b 为一个节点最多的后继节点个数，那么时间复杂度为：

$$1 + b + b^2 + \cdots + b^d = O(b^d)$$

同理，对于 IDA*，当找到最优路径时，设置的阈值为$bound$，每个待扩展节点的最小$f(n)$为$s > 0$，那么搜索的最长路径的长度为$bound/s$，所以 IDA*的时间复杂度为：

$$O(b^{bound/s})$$

5) 空间复杂度：

同上，DFS 的空间复杂度为：

$$O(bd)$$

所以 IDA*的空间复杂度为：

$$O(b * bound/s)$$

二. 伪代码

1. 一致代价搜索：

```
Procedure USC(start, end)
Input: start denotes the begin node, end denotes the end node.
Output: 1 represents successfully find, 0 represents failure.

explored ← an empty set
queue ← an empty priority queue ordered by cost, which means that the node with the
smallest cost are always placed on the head of the queue.
insert start into queue
loop do:
    if queue is empty:
        return 0
    node ← the head of queue
    if node is in explored:
        continue
    if node == end:
        return 1
    for the neighbors of node:
        if the neighbor is not in explored:
```

insert the *neighbor* into *queue*

2. **IDA*搜索：**

Procedure IDA* (*start*, *end*)
Input: *start* denotes the begin node, *end* denotes the end node.
Output: *1* represents successfully find, *0* represents failure.

loop do:
    bound ← a certain value
    res, *val* ← DFS(*start*, *end*, *bound*)
    // *val* is the minimum of excess of *bound*
    if *res* represents successfully find the path:
        return *1*
    else if *res* represents no node can be explored:
        return *0*
    else:
        bound ← *bound* + *val*

三. 代码解释

1. 一致代价搜索：

1) 定义节点的类：记录节点的坐标、父节点的坐标以及达到该节点的代价。

```
class Point:
    def __init__(self, pos, front_p, cost=1):
        self.pos = pos                          # 节点坐标
        self.front_p = front_p                  # 父节点坐标
        self.cost = cost                        # 到达的代价

    def get_pos(self):                          # 返回节点坐标
        return self.pos

    def get_front(self):                        # 返回父节点坐标
        return self.front_p

    def get_cost(self):                         # 返回到达的代价
        return self.cost
```

2) 定义拥有优先队列功能的类：该类永远返回代价最小的节点。

```python
class Priority_Queue:
    def __init__(self):
        self.queue = []                          # 储存队列
        self.len = 0                             # 队列长度

    def empty(self):
        return self.len == 0                     # 返回队列是否为空，空返回True，非空返回False

    def top(self):                               # 如果队列不空，返回队头的节点
        if not self.empty():
            return self.queue[0]
        return None
```

```python
    def insert(self, point):                     # 将节点插入队列
        index = 0                                # 插入的位置
        for i in range(self.len):
            if self.queue[i].get_cost() > point.get_cost():
                index = i
                break
            else:
                index = i + 1
        self.queue.insert(index, point)
        self.len += 1

    def pop(self):                               # 删除队头的节点
        if not self.empty():
            del(self.queue[0])
            self.len -= 1
```

3) 读取文件中的迷宫进内存，找出起点及终点，并将两个点的标签标为'0'。

```python
def read_maze(path):
    '''
    :param path: the path of file which records the information of maze
    :return: the start point (x, y), the end point (x, y), the maze m x n matrix
    '''
    maze = []
    with open(path, 'r') as f:
        lines = f.readlines()
        for i in range(len(lines)):
            row = []
            line = lines[i].split()[0]
            for j in range(len(line)):
                if line[j] == 'S':               # 如果是起点，记录下来，并将该位置的标签置为'0'
                    start = (i, j)
                    row.append('0')
                elif line[j] == 'E':             # 如果是终点，记录下来，并将该位置的标签置为'0'
                    end = (i, j)
                    row.append('0')
                else:
                    row.append(line[j])
            maze.append(row)
    return start, end, maze
```

4) 一致代价搜索：先将起点加入到队列，然后开始循环，每次从队列中取出代价最小的节点，检查是不是目标节点，是则已经找到，否则，将该节点从队列中删去，并将它四邻域中满足以下要求的邻点加入到队列中：

A) 没有越界：$0 \leq x < the\ number\ of\ rows, 0 \leq y < the\ number\ of\ columns$

B) 没有探索过；

C) 没有障碍：

```
def UCS(start, end, maze):
    '''
    :param start: (x, y) represents the start point
    :param end: (x, y) represents the end point
    :param maze: m x n matrix, 0 denotes no obstacle, 1 denotes obstacle
    :return: whether have a path from start point to end point
    '''
    m = len(maze)           # 迷宫的行数
    n = len(maze[0])        # 迷宫的列数

    # record_mat[x][y]记录(x, y)的父节点，如果为(-1, -1)，说明(x, y)还没有被扩展
    record_mat = [[(-1, -1) for j in range(n)] for i in range(m)]
    q = Priority_Queue()                # 创建一个优先队列的对象
    p = Point(start, start, 0)          # 创建一个点的对象，该对象记录了起点的信息
    q.insert(p)
```

```
    while True:
        if q.empty():
            # 队列中没有可以扩展的节点，则找不到一条从起点到终点的路径，通过文件写，画出走过的点
            with open('fail.txt', 'w') as f:
                for i in range(m):
                    for j in range(n):
                        if record_mat[i][j] != (-1, -1):
                            f.write('2')
                        else:
                            f.write(str(maze[i][j]))
                    f.write('\n')
            return False

        t = q.top()         # 取出队列中代价最小的节点
        q.pop()             # 删除该节点

        x, y = t.get_pos()  # 得到该节点的坐标
        if record_mat[x][y] != (-1, -1):    # 该节点已被扩展
            continue
        record_mat[x][y] = t.get_front()    # 记录该节点的父节点坐标

        if (x, y) == end:                   # 到达终点
            break
```

```
        # 将该节点的四邻域的符合条件的节点加入队列，条件为：
        # 1. 没有越界
        # 2. 没有障碍
        # 3. 没有被扩展过
        if x > 0 and maze[x-1][y] != '1' and record_mat[x-1][y] == (-1, -1):
            q.insert(Point((x-1, y), (x, y), t.get_cost()+1))
        if y > 0 and maze[x][y-1] != '1' and record_mat[x][y-1] == (-1, -1):
            q.insert(Point((x, y-1), (x, y), t.get_cost()+1))
        if x < m-1 and maze[x+1][y] != '1' and record_mat[x+1][y] == (-1, -1):
            q.insert(Point((x+1, y), (x, y), t.get_cost()+1))
        if y < n-1 and maze[x][y+1] != '1' and record_mat[x][y+1] == (-1, -1):
            q.insert(Point((x, y+1), (x, y), t.get_cost()+1))
```

```
    # 通过record_mat倒着找路径
    x, y = end
    while True:
        f_x, f_y = record_mat[x][y]
        if (f_x, f_y) == start:
            break
        maze[f_x][f_y] = '2'            # 标记
        x, y = f_x, f_y
    maze[start[0]][start[1]] = 'S'
    maze[end[0]][end[1]] = 'E'

    # 写文件
    with open('answer_1.txt', 'w') as f:
        for row in maze:
            f.write(''.join(row) + '\n')

    return True
```

2. IDA*搜索：

1) 定义节点的类：记录节点的坐标，父节点的坐标以及到达该节点的实际代价和到目标节点的预估代价。

```
class Point:
    def __init__(self, pos, front_p, g, f):
        self.pos = pos                          # 节点的坐标
        self.front_p = front_p                  # 父节点的坐标
        self.g = g                              # 到达该节点的实际代价
        self.f = f                              # 到达目标节点的估计代价

    def get_pos(self):                          # 返回节点的坐标
        return self.pos

    def get_front(self):                        # 返回父节点的坐标
        return self.front_p

    def get_g(self):                            # 返回到达该节点的实际代价
        return self.g

    def get_f(self):                            # 返回到达目标节点的估计代价
        return self.f
```

2) 读取文件中的迷宫进内存的函数同上。

3) IDA*搜索：限制深搜的 f 值（从起点经过该节点并到达终点的预估代价），然后开始深搜，每次深搜有三种情况，返回 1 代表已找到一条最优路径；返回 0 代表受到 f 值限制，无法继续深搜下去，此时应该适当增加 f 值，再进行深搜，所以深搜应当在返回 0 的时候，一起返回超过阈值的最小值，然后在下次深搜前，将该值加到限制深搜的 f 值；返回-1 代表所有可能的路径都扩展过了，但还是找不到一条可行的路径。也就是说，只有深搜返回 1 或者-1，才能结束迭代。最后，如果能找到最优路径，就将路径输出到文件，记录路径的方法还是记录每个节点的父节点，最后从终点的父节点倒回去，直到起点。

```python
def IDA(start, end, maze):
    '''
    :param start: (x, y) represents the start point
    :param end: (x, y) represents the end point
    :param maze: m x n matrix, 0 denotes no obstacle, 1 denotes obstacle
    :return: whether have a path from start point to end point
    '''
    m = len(maze)                # 迷宫的行数
    n = len(maze[0])             # 迷宫的列数

    bound = 10                   # 深搜的阈值
    while True:
        # record_mat[x][y]记录(x, y)的父节点，如果为(-1, -1)，说明(x, y)还没有被扩展
        record_mat = [[(-1, -1) for j in range(n)] for i in range(m)]
        point = Point(start, start, 0, 0)       # 创建包含起点信息的点对象
        best_path = [1000, []]                  # best_path[0]表示最短距离，best_path[1]储存最短路径下的record_mat
        res = DFS(point, end, maze, record_mat, bound, best_path)   # 给定一定的阈值深搜
        if res == 1:                            # 找到最优路径
            break
        elif res == -1:                         # 无法到达终点
            return 0
        else:                                   # 可能阈值太小，无法到达终点
            bound += 10
```

```python
    bound = 10                  # 深搜的阈值
    while True:
        # record_mat[x][y]记录(x, y)的父节点，如果为(-1, -1)，说明(x, y)还没有被扩展
        record_mat = [[(-1, -1) for j in range(n)] for i in range(m)]
        point = Point(start, start, 0, 0)       # 创建包含起点信息的点对象
        best_path = [1000, []]                  # best_path[0]表示最短距离，best_path[1]储存最短路径下的record_mat
        res = DFS(point, end, maze, record_mat, bound, best_path)   # 给定一定的阈值深搜
        if res[0] == 1:                             # 找到最优路径
            break
        elif res[0] == -1:                          # 无法到达终点
            return 0
        else:                                       # 可能阈值太小，无法到达终点
            # print(res[1])
            bound += res[1]
```

```python
    # 通过best_path中的record_mat倒着找路径
    x, y = end
    while True:
        f_x, f_y = best_path[1][x][y]
        if (f_x, f_y) == start:
            break
        maze[f_x][f_y] = '2'                    # 标记路径
        x, y = f_x, f_y
    maze[start[0]][start[1]] = 'S'
    maze[end[0]][end[1]] = 'E'

    # 写文件
    with open('answer_2.txt', 'w') as f:
        for row in maze:
            f.write(''.join(row) + '\n')

    return True
```

4) 深搜：遍历当前节点的四周可行的子节点，继续深搜下去，只要有一个子节点深搜的结果为 1（找到可行的路径），那么当前的节点的结果也是 1；如果所有的子节点深搜的结果都是-1，说明当前的节点是不可行的；如果过有一个子节点深搜的结果是 0，而其他子节点深搜的结果是-1 或者 0，那么当前的节点可能有可行路径，只是被 bound 限制了，所以当前 return 的是 max(子节点深搜结果)以及需要增加的 bound 值（当返回值为 0 时有用）。并且每次找到可行路径，要更新当前可行路径中最优路径以及长度。

```
def DFS(point, end, maze, record_mat, bound, best_path):
    '''
    :param point: current node, object of class Point
    :param end: position of object node (x, y)
    :param maze: m x n matrix, 0 denotes no obstacle, 1 denotes obstacle
    :param record_mat: m x n matrix, record_mat[x][y] = (-1, -1) represents (x, y) hasn't been explored, otherwise
                       represents the position of father node of node (x, y)
    :param bound: limitation of depth-first search
    :param best_path: best_path[0] is the length of shortest path, best_path[1] is the record_mat of shortest path
    :return:
    '''
    if point.get_f() > bound:              # 当前节点的h值超过限制的h值，返回0
        return 0, point.get_f()-bound
```

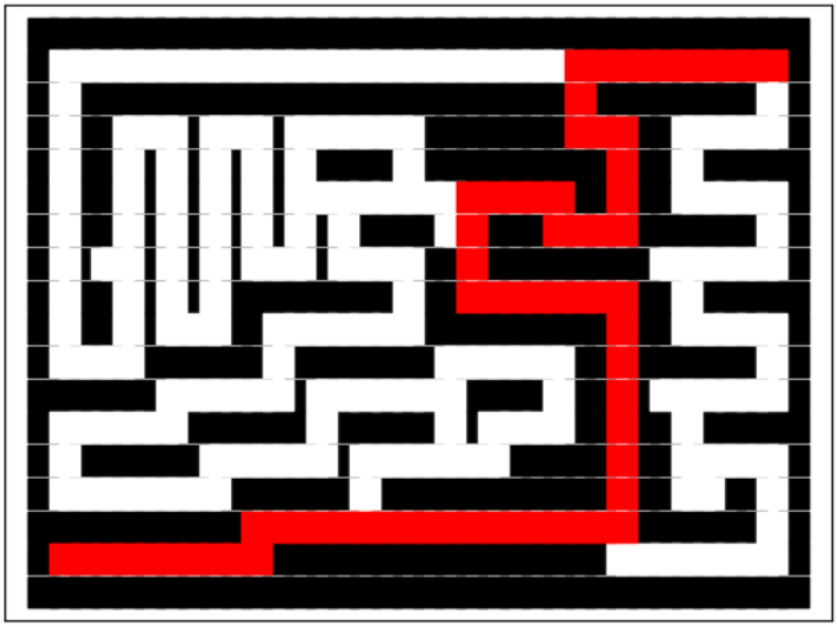
```
    x, y = point.get_pos()                  # 获得当前节点的坐标
    record_mat[x][y] = point.get_front()    # 记录当前节点的父节点的坐标
    if (x, y) == end:                       # 找到目标节点
        if point.get_g() < best_path[0]:    # 如果代价小于之前找到的
            best_path[0] = point.get_g()    # 更新最优路径的代价
            best_path[1] = copy.deepcopy(record_mat)    # 更新最优路径的record_mat
        return 1, None

    m = len(maze)                           # 迷宫的行数
    n = len(maze[0])                        # 迷宫的列数
    next_point = []                         # 符合限制的四邻域的节点
    if x > 0 and maze[x-1][y] == '0' and record_mat[x-1][y] == (-1, -1):
        f = point.get_g() + 1 + abs(x-1-end[0]) + abs(y-end[1])
        next_point.append(Point((x-1, y), (x, y), point.get_g()+1, f))
    if y > 0 and maze[x][y-1] == '0' and record_mat[x][y-1] == (-1, -1):
        f = point.get_g() + 1 + abs(x-end[0]) + abs(y-1-end[1])
        next_point.append(Point((x, y-1), (x, y), point.get_g() + 1, f))
    if x < m-1 and maze[x+1][y] == '0' and record_mat[x+1][y] == (-1, -1):
        f = point.get_g() + 1 + abs(x+1-end[0]) + abs(y-end[1])
        next_point.append(Point((x+1, y), (x, y), point.get_g() + 1, f))
    if y < n-1 and maze[x][y+1] == '0' and record_mat[x][y+1] == (-1, -1):
        f = point.get_g() + 1 + abs(x-end[0]) + abs(y+1-end[1])
        next_point.append(Point((x, y+1), (x, y), point.get_g() + 1, f))
```

```
    if len(next_point) == 0:            # 没有可扩展的节点，说明这条路径不行
        return -1, None
    # cmpfun = operator.attrgetter('h')
    # next_point.sort(key=cmpfun)
    res = -1
    value = 100
    for i in range(len(next_point)):    # 遍历所有四周可能的节点
        r, v = DFS(next_point[i], end, maze, record_mat, bound, best_path)
        res = max(res, r)
        if res == 0 and r == 0:         # 更新下次应当增加的Δbound
            value = min(value, v)
        x, y = next_point[i].get_pos()
        record_mat[x][y] = (-1, -1)     # 恢复
    return res, value
```

四． 实验结果以及分析

1. 一致代价搜索：

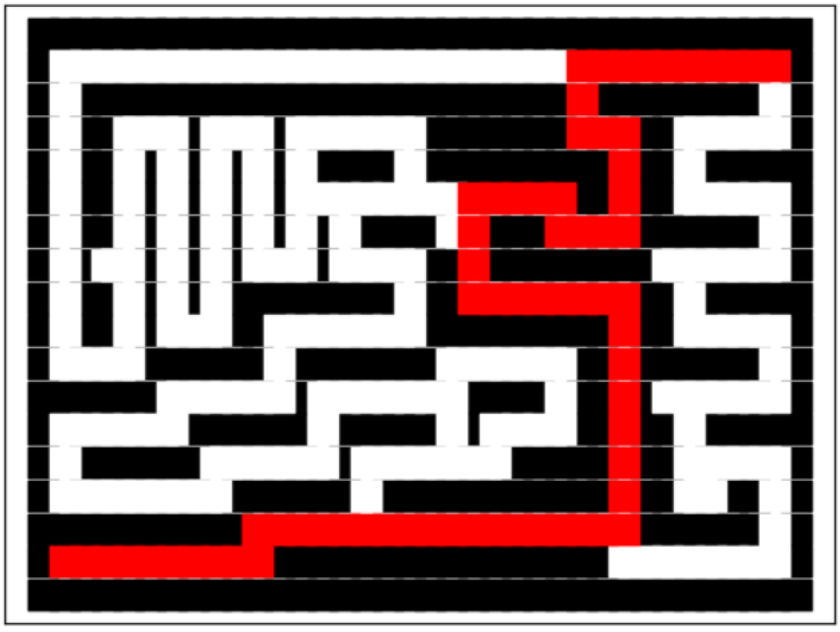
1) 结果：



2） 花费时间：2ms

2. IDA*搜索：

1） 结果：



2） 花费时间：10ms

3. 对比与分析：

一致代价搜索和 IDA*的完备性和最优性在第一部分都有提到，它们都是可以找到最优解的，只要目标节点的代价是有限的，所以这两个算法得到答案都是一样的，因为最优解只有一个。至于在时间上花费，一致代价搜索花费的时间比 IDA*少的多，这也很正常，因为 IDA*在 bound 值没有到达目标节点的预估函数值 f 时，所做的 DFS 都是没有意义

的，所以这就浪费了许多时间。我一开始设的 bound 值为 10，之后根据 DFS 返回值，依次加了 38、2、2、2、2、2、2、2、2、2、2，然后才找到了最优路径，也就是做了 12 次没有用的 DFS，所以 IDA*所花费的时间比一致代价搜索多。最后是空间复杂度，正如第一部分所说，一致代价搜索的空间复杂度是指数级的，而 IDA*的空间复杂度是线性的，因为一致代价搜索要维护一个队列，而 IDA*中的 DFS 不需要。

五． 思考题

1. 这些策略的优缺点是什么？它们分别适用于怎样的场景？

搜索策略	优点	缺点	适用场景
深度优先搜索	空间复杂度是线性的，即与搜索的最长路径长度成正比	如果状态空间是无限的情况下，深度优先搜索是不具有完备性的	状态空间是有限的
宽度优先搜索	在状态空间是无限情况下，宽度优先搜索的完备性还是满足的	空间复杂度是指数级的，即当路径长度太大时，会用到很大的内存	最优解的路径长度不会很长
一致代价搜索	同宽度优先搜索	同宽度优先搜索	相邻节点之间的成本不为 1
深度受限搜索	克服了深度优先搜索的缺点，即在状态空间无限的情况下，只要目标节点的成本有限，就可以同意设定合适的深度找到解	合适的深度需要一定试验才能找到	状态空间是无限的且最优解的路径长度较长
迭代加深搜索	解决了 DFS 和深度受限搜索的缺点	在找到合适的深度之前浪费了许多时间来做深度受限搜索	状态空间是无限的且最优解的路径长度较长
双向搜索（BFS）	时空复杂度较 BFS 缓解了 1/2 次方	空间复杂度还是指数级	起点与终点之间没有环
A*	加入了对先验知识的考虑，使得搜索的效率得到提高	启发式函数至少得是可采纳的；空间复杂度是指数级的	最优解的路径长度不会很长
IDA*	加入了对先验知识的考虑，使得搜索的效率得到提高；空间复杂度是线性的	启发式函数至少得是可采纳的；在找到合适的阈值之前浪费了许多时间做阈值受限搜索	状态空间是无限的且最优解的路径长度较长