中山大学计算机学院本科生实验报告

(2020 秋季学期)

课程名称：人工智能 **任课老师：**饶洋辉

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.11.24

实验八 无信息搜索与启发式搜索

1 实验原理

1.1 无信息搜索

在实验中，无信息搜索选择的是一致代价搜索。一致代价搜索的本质为一个广度优先搜索。通过将边缘节点组织成按路径消耗 $g(n)$ 排序的队列，选择扩展的是 $g(n)$ 最小的节点 n。而在这次实验中，由于每一步的代价都是相同的（为 1），一致代价搜索在解决迷宫问题上退化成了广度优先搜索。

在实验中，采用队列扩展结点，并在扩展队列时将相邻可扩展的节点加入队列，直到队列为空或达到终点。一致代价搜索的伪代码如下

Algorithm 1 一致代价搜索

Input: 起点 $Start\_Point$; 终点 End_Point; 图 Map;
输出: 起点到终点的路径或不存在路径

```
function UNIFORM-COST-SEARCH(Start_Point, End_Point, Map)
    初始化队列 frontier，初始化空集 explored
    起点入队 frontier.append(Start_Point)
    loop
        if EMPTY?(frontier) then return failure
        end if
        根据路径代价获得新扩展节点 Point = frontier.pop()
        if Goal-Test(Point) then return PathToGetPoint(Point)
        end if
        标志该节点为已扩展 add Point to explore
        for Next_Point in Adj(Point) do
            if Next_Point not in explored then frontier.pop(Next_Point)
```

13: **else if** $Next\_Point$ is in $frontier$ with higher PATH-COST **then** 替换 $frontier$ 中的 $Next\_Point$
14: **end if**
15: **end for**
16: **end loop**
17: **end function**

1.2 启发式搜索

启发式搜索又叫有信息的搜索，它利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。而启发式搜索算法提出了估值函数来评估一个节点的重要性。估价函数为：

$$f(x) = h(x) + g(x),$$

其中 $g(x)$ 是从初始节点到节点 x 付出的实际代价；$h(x)$ 是从节点 x 到目标节点的最优路径的估计代价。$h(x)$ 建模了启发式搜索问题中的启发信息。启发式搜索的关键就是如何设计启发式函数。

启发式函数必须包含两个特性，一个是可采纳性：

$$h(n) \leq h^*(n)$$

这个特性保证了估计值要小于真实值，可采纳性也就意味着最优性。如果启发式函数是可采纳的，但在进行了环检测之后就不一定能保持最优性。因此还需要保证单调性：

$$h(n_1) \leq c(n_1 \to n_2) + h(n_2)$$

可以理解就是 $h(n_i)$ 随着 i 的增加要和 $h^*(n_i)$ 越来越接近。只要启发式函数具有一致性，就能在进行环检测之后仍然保持最优性。

在实验中，启发式搜索选择的是 A* 搜索算法。A* 算法可以看作是 UCS 算法的升级版，在原有的 UCS 算法的基础上加入了启发式信息。从初始节点开始，选择估值函数 $f(x)$ 最小的节点进行扩展，直到找到目标节点。A* 搜索算法的伪代码如下

Algorithm 2 A* 搜索

Input: 起点 $Start\_Point$; 终点 End_Point; 图 Map; 估值函数 $f(x) = h(x) + g(x)$
输出: 起点到终点的路径或不存在路径

1: **function** A-Star($Start\_Point, End\_Point, Map, f(x)$)
2: 初始化队列 $frontier$，初始化空集 $explored$
3: 起点入队 $frontier$.append($Start\_Point$)
4: **loop**
5: **if** EMPTY?($frontier$) **then** return failure

end if
根据估值函数 $f(x) = h(x) + g(x)$ 排序队列 $frontier$.sort()
获得估值函数最小的新扩展节点 $Point = frontier$.pop()
if Goal-Test($Point$) **then** return PathToGetPoint($Point$)
end if
标志该节点为已扩展 add $Point$ to explore
for $Next\_Point$ in Adj($Point$) **do**
if $Next\_Point$ not in $explored$ **then** $frontier$.pop($Next\_Point$)
else if $Next\_Point$ is in $frontier$ with higher PATH-COST **then** 替换 $frontier$ 中的 $Next\_Point$
end if
end for
end loop
end function

2 实验过程

2.1 无信息搜索

本次实验使用一致代价搜索实现无信息搜索。

2.1.1 关键代码

本次实验采用 python 实现一致代价搜索，在迷宫问题中，由于扩展的代价都为 1，则简单地实现广度优先搜索。

实验中定义了 Search 类用以封装需要的方法，保存了搜索用到的变量，类的初始化函数如下

```
    def __init__(self, search_policy):
        # 起点 终点 图 图的长与宽（在本次实验中为 18*36）
        self.start, self.end, self.map, self.map_length, self.map_width =
        ↪  self._load_map()
        # 采用的搜索策略
        self.search_policy = search_policy
        # 在节点的行动到下标的映射
        self.action_to_ix = {'Up': 0, 'Down': 1, 'Left': 2, 'Right': 3}
        # 搜索到的最终路径图
```

```
        self.result_map = self.map
        # 每个节点的实际代价 g(x)
        self.point_g = {}
```

类采用 _load_map 函数从 MazeData.txt 文件中读取图，并获取起点与终点。返回的 map 为字典型，表示位置（i, j）处的字符表示，字符对应含义与 MazeData.txt 文件中的含义相同，_load_map 函数实现代码如下

```
    def _load_map(self):
        # 采用字典保存读取的图
        map = {}
        with open('MazeData.txt', 'r') as fp:
            map_list = fp.readlines()
        for i in range(len(map_list)):
            for j in range(len(map_list[i]) - 1):
                # 对应（i, j）位置的字符保存
                map[(i, j)] = map_list[i][j]
                # 保存起点
                if map[(i, j)] == 'S':
                    start = (i, j)
                # 保存终点
                if map[(i, j)] == 'E':
                    end = (i, j)
        return start, end, map, len(map_list), len(map_list[0]) - 1
```

为方便判断，定义 _can_reach 函数与 _gold_test 函数分别判断从当前节点是否能进行移动（上下左右）与是否达到终点，实现代码如下

```
    def _can_reach(self, state, action):
        # Up
        if action == 0:
            return (state[0] - 1, state[1]), self.map[(state[0] - 1,
            ↪  state[1])] != '1'
        # Down
        elif action == 1:
```

```
            return (state[0] + 1, state[1]), self.map[(state[0] + 1,
            ↪  state[1])] != '1'
        # Left
        elif action == 2:
            return (state[0], state[1] - 1), self.map[(state[0], state[1] -
            ↪  1)] != '1'
        # Right
        else:
            return (state[0], state[1] + 1), self.map[(state[0], state[1] +
            ↪  1)] != '1'

    def _gold_test(self, point):
        return point == self.end
```

根据以上得到的图与判断函数，可进行一致代价搜索。由于退化成广度优先搜索，采用简单的队列即可实现搜索。根据算法实现的代码如下

```
    def _UCS(self):
        count = 0                   # 搜索次数
        father_point = {}           # 记录父节点
        queue_length = []
        search_queue = []       # 搜索队列
        search_queue.append(self.start)
        self.point_g[self.start] = 0        # 初始化起点的真实代价
        visted = []                 # 保存已搜索过的节点
        flag = True
        while flag and search_queue:
            queue_length.append(len(search_queue))
            point = min(search_queue, key=self._get_g)    # 返回最小代价
            search_queue.remove(point)
            count += 1
            # 判断是否到达终点
            if self._gold_test(point):
                flag = False
                break
            # 相邻的可到达节点入队
```

```
            for action in range(4):
                next, reachable = self._can_reach(point, action)
                if next not in visted and reachable:
                    search_queue.append(next)
                    self.point_g[next] = self.point_g[point] + 1
                    # 保存父节点
                    if next not in father_point:
                        father_point[next] = point
            visted.append(point)
        # 输出结果
        if not flag:
            print('Using', self.search_policy, 'needs', count, 'steps')
            print('Using', self.search_policy, 'needs', max(queue_length),
            ↪  'queue length')
            self._get_result_map(father_point)
            self._print_map(self.result_map)
        else:
            print('No result')
```

2.1.2 搜索结果

采用一致代价搜索得到的搜索结果如图1, 一致代价搜索一共搜索了 276 步，最终得到的路径长度为 68，队列长度最大为 9。

```
Using UCS needs 276 steps
Using UCS needs 9 queue length
The path length found is 68
11111111111111111111111111111111
10000000000000000000000·········S1
10111111111111111111111·1111111101
10110001000100000011111···11000001
10110101010101110111111111·11011111
10110101010100000000·····11·11000001
10110101010101011110·111····11111101
10100101010001000011·111111110000001
10110101011111111011·········11011111
10110100011000000011111111·11000001
10000111111011111110000011·11111101
11111100000010000000111011·10000001
10000001111101111101000011·11011111
10111111000000100000011111·11000001
10000000011111101111111111·11001101
1111111111···················11111101
1E·········11111111111111000000001
11111111111111111111111111111111111
```

图 1: 一致代价搜索

2.2 启发式搜索

本次实验使用 A* 搜索实现启发式搜索。

2.2.1 关键代码

本次实验采用 python 实现 A* 搜索，其主要的函数代码实现与无信息搜索相同，区别在于启发式函数的设计与优先队列选取的依据。其中启发式函数采用了目标节点与当前节点的曼哈顿距离，更符合本次迷宫中四领域的移动，实验中定义了 _heuristic_plus 函数计算估值函数 $f(x) = h(x) + g(x)$，实现代码如下

```
    def _heuristic_plus(self, point):
        return abs(point[0] - self.end[0]) + abs(point[1] - self.end[1]) +
        ↪  self.point_g[point]
```

A* 搜索时，队列中的每一节点根据估值函数进行排序，选取估值函数最小的节点作为下一扩展节点。A* 搜索的具体实现代码如下

```
    def _Astar(self):
        count = 0                # 搜索次数
        father_point = {}        # 记录父节点
        search_queue = []        # 搜索队列
        search_queue.append(self.start)
        self.point_g[self.start] = 0        # 初始化起点的真实代价
        visted = []              # 保存已搜索过的节点
        flag = True
        while flag and search_queue:
            # 选择最小估值函数的扩展节点
            point = min(search_queue, key=self._heuristic_plus)
            search_queue.remove(point)
            if self._gold_test(point):
                flag = False
                break
            for action in range(4):
                next, reachable = self._can_reach(point, action)
                # 判断是否更新真实代价
                if next in search_queue and self.point_g[point] + 1 <
                ↪  self.point_g[next]:
```

```
                    self.point_g[next] = self.point_g[point] + 1
                    continue
                # 相邻节点入队
                if next not in visted and reachable:
                    search_queue.append(next)
                    # 保存真实代价
                    self.point_g[next] = self.point_g[point] + 1
                    if next not in father_point:
                        father_point[next] = point
            visted.append(point)
            count += 1
        # 打印结果
        if not flag:
            print('Using', self.search_policy, 'needs', count, 'steps')
            self._get_result_map(father_point)
            self._print_map(self.result_map)
        else:
            print('No result')
```

2.2.2 搜索结果

采用 A* 搜索得到的搜索结果如图2, A* 搜索一共搜索了 223 步，最终得到的路径长度为 68，队列长度最大为 9。

```
Using Astar needs 223 steps
Using Astar needs 8 queue length
The path length found is 68
1111111111111111111111111111111111
1000000000000000000000000·········S1
1011111111111111111111111·1111111101
1011000100010000001111111···11000001
1011010101010111101111111·11011111
1011010101010000000·····11·11000001
1011010101010101111·111····11111101
1010010101000100001·111111110000001
1011010101111111011·········11011111
1011010001100000001111111·11000001
1000011111101111111100000011·11111101
1111110000001000000011110111·10000001
1000000111111011111010000011·11011111
1011111100000010000000111111·11000001
1000000001111110111111111111·11001101
1111111111···················11111101
1E·········1111111111111111000000001
1111111111111111111111111111111111
```

图 2: A* 搜索

3 实验分析

3.1 算法性能比较

3.1.1 完备性

一致代价搜索对解路径的步数并不关心，只关心路径总代价，因此如果存在零代价的行动就可能陷入死循环。在实验中，由于每一步的代价都大于 0，一致代价搜索会将图中相连的路径节点都加入队列。如果存在路径到达目标点，它一定能搜索出一条路径到达。因此在这种情况下，一致代价搜索具有完备性。

A* 搜索的完备性与启发式函数的设计有关，当 $h(n) = 0$ 时，对于任何的节点，启发式函数都是单调的，此时 $A*$ 搜索就变成了一致代价搜索。当启发式函数的设计满足一致性，那么 A* 搜索就具有完备性。

在实验中，一致代价搜索与启发式搜索都找到了解，都具有完备性。

3.1.2 最优性

一致代价搜索使用的优先队列保证了每一个出队扩展的节点都是最优的，能保证第一次探索到的节点时到达该节点的路径是最优的，因此一致代价搜索具有最优性。

A* 搜索的最优性与启发式函数的设计有关，当 $h(n) = 0$ 时，对于任何的节点，启发式函数都是单调的，此时 $A*$ 搜索就变成了一致代价搜索。当启发式函数的设计满足可采纳性，那么 A* 搜索就具有最优性。

在实验中，一致代价搜索与启发式搜索都找到了最优解，都具有最优性。

3.1.3 时间复杂度

一致代价搜索中，搜索的时间复杂度与路径代价有关，假设 C 是起点到目标点最优情况下总代价，而 e 是整个图中所有路径最小的代价。则从起点到终点至多要走 $[C/e]+1$ 次，设 b 为每个点的分叉数量。每个点最多只能向四个方向拓展，则分支数最多为 4，则时间复杂度为 $O(b([C/e]+1))$。

A* 搜索中，搜索的时间复杂度与启发式函数的设计有关。在实验中以扩展的节点数作为时间复杂度的衡量标准，尝试使用不同的启发式函数，搜索的时间复杂度如表1。

表 1: 搜索策略相应的扩展节点数

搜索策略	**一致代价搜索**	**A* 搜索**（曼哈顿距离）	**A* 搜索**（2 倍曼哈顿距离）	**A* 搜索**（对角线距离）
扩展的节点数	276	223	220	230

在表中可知，A* 搜索的时间复杂度与启发式函数的设置有关，采用正确启发式函数的 A* 搜索的时间复杂度低于启发式函数。由于在实验中根据曼哈顿距离与对角线距离设置的启发式函数都满足一致性，而曼哈顿距离更接近于当前节点与终点的实际距离，在实验中的效果较好。而 2 倍的曼哈顿距离提高了估值函数中启发式函数的占比，在实验中个别点的选取上起到了帮助。

3.1.4 空间复杂度

一致代价搜索中，搜索的空间使用在于保存边界扩展节点的信息，每当扩展一个新的节点都会增加相应节点的相邻节点信息，因此其空间复杂度与时间复杂度类似。

A* 搜索中，搜索的空间复杂度与启发式函数的设计有关。在实验中以队列长度的最大值作为空间复杂度的衡量标准，尝试使用不同的启发式函数，搜索的空间复杂度如表2。

表 2: 搜索策略相应的队列长度最大值

搜索策略	一致代价搜索	A* 搜索（曼哈顿距离）	A* 搜索（2 倍曼哈顿距离）	A* 搜索（对角线距离）
队列长度最大值	9	8	8	9

在表中可知，A* 搜索的空间复杂度与启发式函数的设置有关。而在本次实验中，由于路径较少，因此两种实现方法的区别并不明显，空间复杂度大致与时间复杂度的比较相似。

4 实验思考

- 这些策略的的优缺点是什么？它们分别适用于怎样的情景？

  (1) 一致代价搜索：一致代价搜索每一次扩展节点能保证该节点是沿着最优的路径搜索到的，确保了最优性。但它和 BFS 相同，在边界队列需要保存指数级的节点数量，在搜索深度较大时会消耗大量空间，一般适用于解决最短路径的搜索问题。

  (2) A* 搜索：A* 搜索加入了启发式函数对边界节点进行排序，搜索的性能很大程度取决于设置的启发式函数。启发式函数设置不好，会耽误搜索方向，导致搜索效率降低。在对终点位置有已知的情况下，可设置较合适的启发式函数，从而加快搜索进程。A* 搜索的适用场景与一致代价搜索类似，适合解决最短路径的问题。

  (3) 迭代加深搜索：迭代加深搜索本质上是一个深度优先搜索，DFS 适用于对空间复杂度要求较高的环境，而当深度过大的时候，搜索效率低。迭代加深搜索是对 DFS 这个缺点的一个改进，通过限制搜索深度来防止过深的搜索，并在这个过程中进行剪枝。但由于搜索深度难以设置，需要在实验过程中一步步加深，会导致节点

的重复访问。适用于解决需要进行广度优先搜索，但却没有足够的空间进行搜索的问题。

(4) IDA* 搜索：IDA* 搜索与迭代加深搜索限制搜索深度的方法相同，并且融入启发式函数获得终点的信息，因此需要设置适当的估值函数，要尽量接近实际深度且不超过实际深度，将信息用在搜索上以加快搜索进程。其适用场景与迭代加深搜索类似。

(5) 双向搜索：双向搜索从起点和终点两个方向同时开始搜索。因为搜索深度的减半，双向搜索可以提高搜索效率，而由于双向搜索需要维护两个边界队列，空间复杂度随着搜索的深度的提高而增加。双向搜索的适用于起点终点信息都已知的情景。