M3 应用层大作业报告

通信 1904 儒曼 19211214

摘要

本文为作者 M3 组应用层总结报告，总结了在独立完成本次大作业中所做的工作。本文第一部分介绍了自主学习 python 的过程、内容以及参考资料；第二部分介绍了搜索部分的课堂示例代码 maze.py，详细分析了代码的原理并实现了除原先代码已有方案外的其它 3 种搜索方案；第三部分介绍哈佛 CS50 搜索的项目 1—degrees，介绍了其内容、原理和实现方法，并通过实验比较了不同实现方案的优劣；第四部分介绍哈佛 CS50 搜索的项目 2—tictactoe, 介绍了其内容、原理、实现方法以及代码优化。第五部分介绍作者通过完成大作业获得的收获和学习经验。

1 Python 学习

为完成本次大作业，本人用 2 天的时间通过不同渠道简单学习了与 python 语言有关的相关知识，具体包括：

1. Python 基础语法
2. Python 变量的定义和使用
3. Python 运算符
4. Python 条件语句
5. Python 循环语句
6. Python 列表的定义与操作
7. Python 元组的定义与操作
8. Python 字典的定义与操作
9. Python 函数的定义与使用
10. Python 文件读入和读出
11. Python 面向对象的编程方法。

具体的参考资料包括：

1. 王一行同学的《Python 基础指南》
2. 网站 runoob.com Python 基础教程
3. 《Python 从入门到精通》
4. https://www.bilibili.com/video/BV1wD4y1o7AS?spm_id_from=333.337.search-card.all.click

本人之前接触过 C 语言和 java 语言，这些学习基础对学习 Python 有所助益。通过短期学习我了解了 Python 语言的基本知识，不仅为顺利完成大作业打下基础，还让我掌握了一门简单实用的编程语言。

2 课堂实例代码 mave.py

本次大作业我选择的题目是搜索（0-search），包括老师课堂演示的解迷宫实例代码和两个哈佛 CS50 项目。此处首先介绍解迷宫实例代码。

2.1 主要代码原理分析

主函数：

1. 读入 maze.txt 并构建迷宫。
2. 绘制迷宫图案。
3. Solve 迷宫，记录搜索状态数。
4. 打印迷宫与搜索路径结果。
5. 读出迷宫 maze.png

Node 类定义 3 个属性：state 为状态，以元组（i,j）表示，i 为迷宫的行数，j 为迷宫的列数。读入迷宫时为左上到右下，左上序号小右下序号大；parent 为父节点；action 为搜索方向，包括上下左右 4 种。

```
class Node():
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
```

堆栈数据结构，包括 5 个方法：def __init__(self) 初始化，定义其为列表格式；def add(self, node) 从栈顶添加成员；def contains_state(self, state) 判断堆栈内是否存在某个状态；def empty(self) 判断堆栈是否为空；def remove(self) 出栈操作，从栈顶即列表最后一个元素删除并返回成员。

```
class StackFrontier():
    def __init__(self): # 初始化堆栈，列表格式
        self.frontier = []
    def add(self, node): # 堆栈添加成员
        self.frontier.append(node)
    def contains_state(self, state): # 判断某一状态 state 在堆栈内是否存在
        return any(node.state == state for node in self.frontier)
    def empty(self): # 判断堆栈是否为空
        return len(self.frontier) == 0
    def remove(self): # 移除操作，从队尾移除，返回移除的 node
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node
```

队列数据结构，重写堆栈类中的 remove(self)即可，将从列表末尾删除成员改为从列表开头删除成员。

```
class QueueFrontier(StackFrontier):
  def remove(self):  # 队列与堆栈唯一不同的一点是，移除成员时从队首移除
    if self.empty():
      raise Exception("empty frontier")
    else:
      node = self.frontier[0]
      self.frontier = self.frontier[1:]
      return node
```

迷宫类，其初始化函数功能是通过扫描将 maze.txt 转化为迷宫，格式为布尔表。

```
    class Maze():  # 迷宫类
      def __init__(self, filename):
        # 读取迷宫文件
        with open(filename) as f:
          contents = f.read()
        # 确认迷宫的合法性，迷宫至少要有一个起点和一个终点
        if contents.count("A") != 1:
          raise Exception("maze must have exactly one start point")
        if contents.count("B") != 1:
          raise Exception("maze must have exactly one goal")
        # 得到迷宫的高度和宽度
        contents = contents.splitlines()
        self.height = len(contents)  # 迷宫高度
        self.width = max(len(line) for line in contents)  # 迷宫宽度
        # 扫描迷宫
        self.walls = []
        for i in range(self.height):
          row = []
          for j in range(self.width):  # 遍历迷宫的每一格
            try:
              if contents[i][j] == "A":  # 找到迷宫的起点
```

```
                    self.start = (i, j)
                    row.append(False)
                elif contents[i][j] == "B": # 找到迷宫的终点
                    self.goal = (i, j)
                    row.append(False)
                elif contents[i][j] == " ": # 迷宫的空位置
                    row.append(False)
                else:
                    row.append(True) # 迷宫的墙壁
            except IndexError:
                row.append(False)
        self.walls.append(row) # 将迷宫转化为布尔表
    self.solution = None

```

neighbors 函数用于返回某一状态下的所有可行邻居，返回列表。

```
def neighbors(self, state): # 找到当前状态下所有的可行邻居
    row, col = state
    candidates = [
        ("up", (row - 1, col)),
        ("down", (row + 1, col)),
        ("left", (row, col - 1)),
        ("right", (row, col + 1))
    ]
    result = []
    for action, (r, c) in candidates:
        if 0 <= r < self.height and 0 <= c < self.width and not self.walls[r][c]:
            result.append((action, (r, c)))
    return result

```

由于篇幅原因，打印迷宫函数 print 与绘制迷宫函数 output_image 不再赘述。

原先课堂代码中迷宫解法为深度优先搜索，其原理是是尽可能“深”地搜索树。它的基本思想是：为了求得问题的解，先选择某一种可能情况向前（子结点）探索，在探索过程中，一旦发现原来的选择不符合要求，就回溯至父结点重新选择另一结点，继续向前探索，如此反复进行，直至求

得最优解。实现的方法为堆栈，即后入先出。实现函数为 solve_DFS，具体代码如下。其核心部分为每次选择未被探索的可行邻居入栈，取栈顶成员继续探索；堆栈为空表明无解；找到目标后逐一寻找父节点得到迷宫解法。

```
    def solve_DFS(self): # 深度优先搜索
        """Finds a solution to maze, if one exists."""
        # Keep track of number of states explored
        self.num_explored = 0 # 已搜索的状态数目
        # Initialize frontier to just the starting position
        start = Node(state=self.start, parent=None, action=None) # 设置初始节点，状态为{start,nome,none}
        frontier = StackFrontier() # 初始化堆栈
        frontier.add(start) # 将初始节点加入堆栈
        # Initialize an empty explored set
        self.explored = set() # 已搜索的节点
        # Keep looping until solution found
        while True:
            # If nothing left in frontier, then no path
            if frontier.empty(): # 堆栈中无 node 表示无解
                raise Exception("no solution")
            # Choose a node from the frontier
            node = frontier.remove() # 移出堆栈顶 node
            self.num_explored += 1 # 探索的节点数+1
            # If node is the goal, then we have a solution
            if node.state == self.goal: # 如果 node 为目标表明找到解
                actions = []
                cells = []
                while node.parent is not None: # 逐一寻找 parent
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse() # 对列表逆序排列
                cells.reverse() # 对列表逆序排列
                self.solution = (actions, cells)
```

```
                return
            # Mark node as explored
            self.explored.add(node.state) # 该节点已被探索
            # Add neighbors to frontier
            for action, state in self.neighbors(node.state): # 遍历当前状态的所有邻居
                if not frontier.contains_state(state) and state not in self.explored: # 堆栈中不包含该状态且未被探索
                    child = Node(state=state, parent=node, action=action)
                    frontier.add(child)
```

宽度优先搜索与深度有限思路不同，其思想是尽量“广”地进行搜索。先发现的节点最先被探索，符合队列先入先出的特点。宽度优先搜索与深度优先搜索仅仅是在探索节点的顺序上有所不同，因此只需将深度优先搜索中的堆栈改为队列即可，即 frontier = StackFrontier()改为 frontier = QueueFrontier()。

贪婪算法是深度优先搜索的变种，其思想与深度优先搜索相同。唯一不同的是深度优先搜索在相邻可行状态入栈时是无序的，但贪婪优先搜索是以该状态与目标之间的曼哈顿距离（成本）作衡量，曼哈顿距离大的先入栈，曼哈顿距离小的后入栈。这样 Agent 首先探索的就是曼哈顿距离小的节点，体现了“贪婪”的思想。

```
        def solve_Greedy(self): # 贪婪搜索
            """Finds a solution to maze, if one exists."""
            # Keep track of number of states explored
            self.num_explored = 0 # 已搜索的状态数目
            # Initialize frontier to just the starting position
            start = Node(state=self.start, parent=None, action=None) # 设置初始节点，状态为{start,nome,none}
            frontier = StackFrontier() # 初始化堆栈
            frontier.add(start) # 将初始节点加入堆栈
            # Initialize an empty explored set
            self.explored = set() # 已搜索的节点
            # Keep looping until solution found
            while True:
                # If nothing left in frontier, then no path
                if frontier.empty(): # 堆栈中无 node 表示无解
                    raise Exception("no solution")
```

```
203
204             # Choose a node from the frontier
205             node = frontier.remove()  # 移出堆栈顶 node
206             self.num_explored += 1  # 探索的节点数+1
207             # If node is the goal, then we have a solution
208             if node.state == self.goal:  # 如果 node 为目标表明找到解
209                 actions = []
210                 cells = []
211                 while node.parent is not None:  # 逐一寻找 parent
212                     actions.append(node.action)
213                     cells.append(node.state)
214                     node = node.parent
215                 actions.reverse()  # 对列表逆序排列
216                 cells.reverse()  # 对列表逆序排列
217                 self.solution = (actions, cells)
218                 return
219             # Mark node as explored
220             self.explored.add(node.state)  # 该节点已被探索
221             # Add neighbors to frontier
222             choices = []  # 待入栈邻居列表
223             for action, state in self.neighbors(node.state):  # 遍历当前状态的所有邻居
224                 if not frontier.contains_state(state) and state not in self.explored:  # 堆栈中不包含该状态且未
225 被探索
226                     child = Node(state=state, parent=node, action=action)
227                     choices.append(child)
228             # 将邻居根据曼哈顿距离由大到小排序
229             for i in range(len(choices) - 1):
230                 for j in range(i + 1, len(choices)):
231                     if abs(self.goal[0] - choices[i].state[0]) + abs(self.goal[1] - choices[i].state[1]) < abs(
232                             self.goal[0] - choices[j].state[0]) + abs(self.goal[1] - choices[j].state[1]):
233                         temp = choices[i]
234                         choices[i] = choices[j]
235                         choices[j] = temp
```

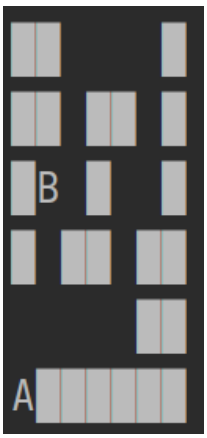
```
            # 将排序好的邻居顺序入栈
            for i in range(len(choices)):
                frontier.add(choices[i]) # 将搜索到的邻居加入堆栈
```

A*搜索与贪婪算法不同，其不仅考虑从当前状态到达目标的成本，还考虑从起始位置到达当前位置的花费成本。该算法跟踪（到现在为止的路径成本+目标的估计成本），一旦它超过某个先前选项的估计成本，算法将放弃当前路径并返回到先前的选项，从而防止自己沿着一条长而低效的路径搜索。具体做法是每次选取节点时从列表中选取启发式函数值最小的节点进行探索，启发式函数如下，启发式函数值为从起点到当前状态的曼哈顿距离+从当前状态到达目标的曼哈顿距离。

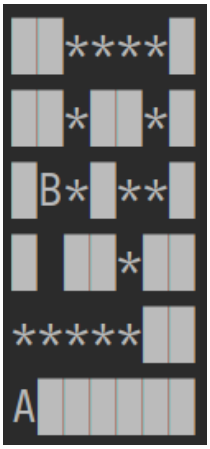
```
        def findMinNode(self, frontier): # 用于寻找启发函数最小的节点，启发式为：与起点的曼哈顿距离+与终点的曼哈顿距离
            NodeM = frontier[-1]
            for i in range(len(frontier)):
                node = frontier[i]
                if (abs(self.start[0] - node.state[0]) + abs(self.start[1] - node.state[1]) + abs(
                        self.goal[0] - node.state[0]) + abs(self.goal[1] - node.state[1]) < abs(
                    self.start[0] - NodeM.state[0]) + abs(self.start[1] - NodeM.state[1]) + abs(
                    self.goal[0] - NodeM.state[0]) + abs(self.goal[1] - NodeM.state[1])):
                    NodeM = node
            return NodeM
```

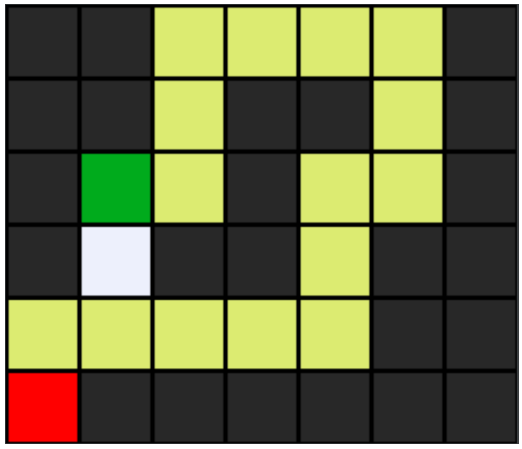
2.2 运行结果

首先是深度优先，解如下迷宫（“A”为起点，“B”为终点）：



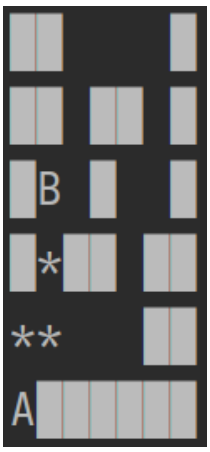
运行结果如下：

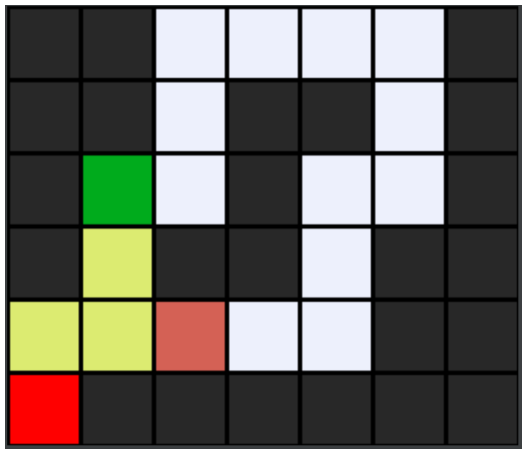




显然，由于深度优先搜索的“一条路走到黑”的局限性，Agent 未能搜索到一条最短路径。

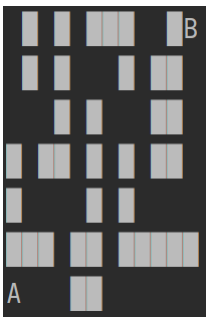
改用宽度优先搜索解同一迷宫，运行结果如下：



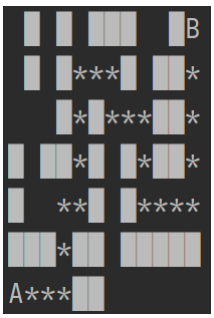


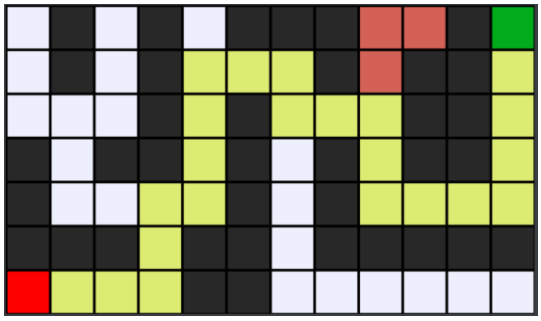
此时所得路径即为最优，得益于宽度优先搜索搜索范围“广”的优势。

运用贪婪算法求解如下迷宫：

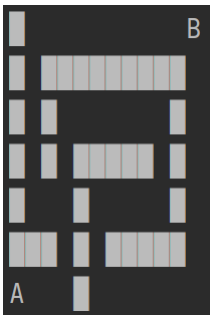


运行结果如下：

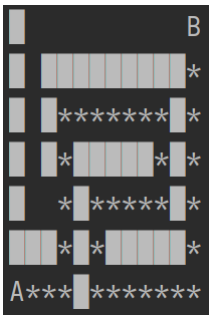


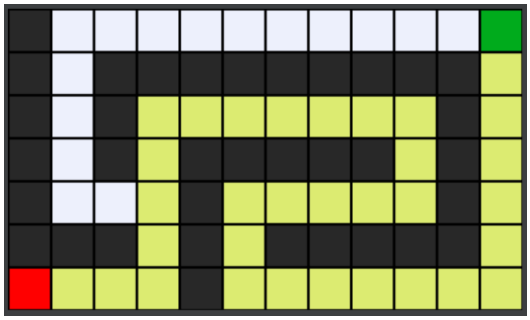


可以看到，虽然中途曾陷入歧途，但贪婪算法确实选择了一条最短的求解路径。但当迷宫中存在误导性路径时，贪婪算法的效率就会显著下降。如下图迷宫所示：

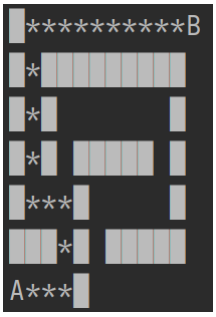


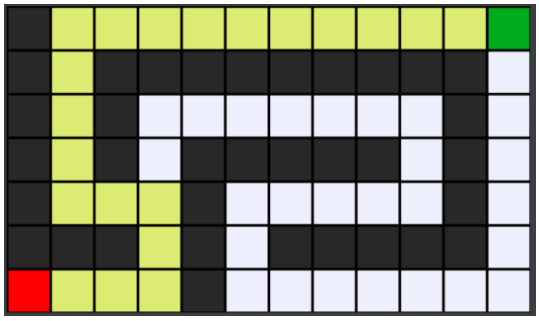
贪婪算法运行结果如下：





可以看到，由于贪婪算法在分叉路口选择了错误的方向，同时又“一路走到黑”，Agent 最终给出了一条低效路径。此时可以选用 A*算法，一旦启发式函数值超过某个先前选项的估计成本，则放弃当前路径并返回到先前的选项，从而防止 Agent 沿着一条长而低效的路径搜索。运用 A*搜索解同一迷宫，运行结果如下：





此时 Agent 选择了最短路径，没有被误导选择低效路径。

3 Degrees

就像游戏 Six Degrees of Kevin Bacon 中一样，任何一个好莱坞影星都可以在 6 步内与 Kevin Bacon 产生联系。如果两个影星共同参演同一电影，则两人之间就能产生联系。例如 Jennifer Lawrence 和 Kevin Bacon 共同参演“X-Men: First Class”，Kevin Bacon 和 Tom Hanks 共同参演“Apollo 13”，则 Jennifer Lawrence 和 Tom Hank 之间就产生了关系，步数为 2。项目的要求是给定两个影星，求解出他们之间的最短关系

3.1 项目分析

实际上这也是一个搜索问题。如果将它与迷宫问题类比，则 states 就是影星，action 就是电影，不同的影星通过电影联系了起来。但与迷宫问题不同的是，这个问题很难归纳并设计出一个启发式函数，因此贪婪算法与 A*算法不再适用，理论上只能运用深度优先搜索/宽度优先搜索进行求解。

项目给定了两个文件夹，分别命名为 small 和 large，small 中的数据量小，为十的数量级；large 中的数据量很大，为百万的数量级。后文会分析，深度优先搜索和宽度优先搜索在这两种场合的适用度是不同的。

文件夹下共有 3 个文件，分别是 movies.csv、people.csv 和 stars.csv。movies.csv 给出了不同电影的序号、名称和年代；people.csv 中给出了不同影星的序号、姓名和出生年代；stars.csv 中给出了影星与电影的联系，用二者的序号表示。

影星与电影之间的关系用元组表示，最终求解得到的路径用列表表示。例如[(1,2),(3,4)]表示初始影星与影星 2 通过电影 1 产生联系，影星 2 与目标影星 4 通过电影 3 产生联系。堆栈和队列的定义与 2 中一致，求解的关键函数如下：

```
    def shortest_path(source, target):
        """
        Returns the shortest list of (movie_id, person_id) pairs
        that connect the source to the target.

        If no possible path, returns None.
        """
        global time
        start = Node(state=source, parent=None, action=None)
        frontier = QueueFrontier()
        frontier.add(start)

        num_explored = 0
        explored = set()

        while True:
            time = time + 1
            if frontier.empty():
                raise Exception("no solution")

            node = frontier.remove()
            num_explored += 1

            if node.state == target:
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
```

```
            cells.reverse()
            solution = list(zip(actions, cells))
            return solution

        neighbors = neighbors_for_person(node.state)
        explored.add(node.state)
        for action, state in neighbors:
            if not frontier.contains_state(state) and state not in explored:
                child = Node(state=state, parent=node, action=action)
                frontier.add(child)
```

显然这与 2 中的宽度优先搜索完全一致，只不过 states 就是影星，action 就是电影而已。

3.2 运行结果与方案比较

Small 中运用宽度优先搜索实现查找 Chris Sarandon 和 Bill Paxton 之间的最小距离如下：

```
Loading data...
Data loaded.
Name: Chris Sarandon
Name: Bill Paxton
3 degrees of separation.
1: Chris Sarandon and Robin Wright starred in The Princess Bride
2: Robin Wright and Gary Sinise starred in Forrest Gump
3: Gary Sinise and Bill Paxton starred in Apollo 13
8
```

其含义是 Chris Sarandon 和 Bill Paxton 之间的距离为 3，Chris Sarandon 和 Robin Wright 共同演过“Princess Bride”，Robin Wright 和 Gary Sinise 共同演过“Forest Gump”，Gary Sinise 和 Bill Paxton 共同演过“Apollo 13”。查找演员和电影的对应关系可验证其正确性。搜索步骤数为 8。

使用 small 文件夹下的数据时，也可以使用深度优先搜索。将原先宽度优先搜索代码中的 rontier = QueueFrontier()改为frontier = StackFrontier()，运行结果如下：

```
Data loaded.
Name: Chris Sarandon
Name: Bill Paxton
3 degrees of separation.
1: Chris Sarandon and Robin Wright starred in The Princess Bride
2: Robin Wright and Tom Hanks starred in Forrest Gump
3: Tom Hanks and Bill Paxton starred in Apollo 13
7
```

可以看到虽然演员名不同，但最终结果也是找到了最短路径，而且深度优先搜索比宽度优先搜索查找的步骤还要短。分别对不同的 source 和 target 查找 10 次，比较深度优先搜索和宽度优先搜索的查找步数：

source	Kevin Bacon	Cary Elwes	Mandy Patinkin	Chris Sarandon	Jack Nicholson	Sally Field	Gerald R. Molen	Cary Elwes	Dustin Hoffman	Chris Sarandon
target	Tom Cruise	Tom Hanks	Dustin Hoffman	Demi Moore	Bill Paxton	Valeria Golino	Gary Sinise	Mandy Patinkin	Demi Moore	Jack Nicholson
steps	1	1	5	4	2	4	3	1	2	4
BFS	5	5	14	10	9	15	8	3	7	12
DFS	10	5	8	12	15	8	14	2	5	7

BFS 所用平均查找步数为 8.8，DFS 所用平均查找步数为 8.6，二者相差无几。因此在数据量小时两种方法并无明显优劣之分。

但当数据量很大时，BFS 和 DFS 的查找效率就有明显差异。使用项目所给定的 large 文件夹下的数据用 BFS 进行搜索，搜索 Ingmar Bergman 和 Marlon Brando 的最短距离，结果如下：

```
Loading data...
Data loaded.
Name: Ingmar Bergman
Name: Marlon Brando
2 degrees of separation.
1: Ingmar Bergman and Gene Hackman starred in A Look at Liv
2: Gene Hackman and Marlon Brando starred in Superman
898
```

用 DFS 搜索结果如下：

```
Loading data...
Data loaded.
Name: Ingmar Bergman
Name: Marlon Brando
```

在有限的运算资源条件下，BFS 能运行得到结果，而 DFS 未能在有限时间内运算得到结果。原因是演员间联系的路径数目要远多于演员之间联系的深度。DFS 是“一条路走到黑”，没有探索完一个路径就不会切换到另一可能的路径上。BFS 在广度上搜索，能迅速排除大量无用信息。因此在数据量很大的情况下，对于这一问题而言只能使用宽度优先搜索。

4 Tictactoe

项目 tictactoe 要求我们设计一个能够和玩家玩井字棋的 Agent，井字棋的规则是一方为“X”一方为“O”，“X”先手下棋，棋盘为 3*3 的九宫格，哪一方先连成 3 个棋子哪一方就获胜。项目要求我们完成部分函数的编写以完成项目，并能运行程序使 Agent 与玩家对弈，玩家永远不可能胜过 Agent。

4.1 辅助函数编写

项目要求我们编写的辅助函数有 6 个，分别是 player(board), actions(board), result(board,action), winner(board), terminal(board), utility(board)。下面依次介绍。

棋盘 Board 类型为列表，共有 3 个列表元素分别对应棋盘的 3 行，每个列表元素分别有 3 个元素。空状态为 Empty=None，有棋子时分别对应 X=“X”，O=“O”。初始状态时 9 个位置均为 Empty。

player(board)根据棋盘状态返回下一个落子的一方，X 或 O。实现方法是统计棋盘上的 X 和 O 棋子个数，相等则为 X，不等则为 O。

```
def player(board):
    """
    Returns player who has the next turn on a board.
    """
    countX = 0
    countO = 0
    for i in range(3):
        for j in range(3):
```

```
            if board[i][j] == 'X':
                countX = countX + 1
            if board[i][j] == 'O':
                countO = countO + 1
    if countX == countO:
        return X
    else:
        return O

actions(board)根据棋盘状态返回所有空位置（列表）。
def actions(board):
    """
    Returns set of all possible actions (i, j) available on the board.
    """
    actionSet = []
    for i in range(3):
        for j in range(3):
            if board[i][j] == EMPTY:
                actionSet.append((i, j))
    return actionSet

result(board,action)根据棋盘状态和行为（元组（i,j）表示，表明在 i 行 j 列落子）返回落子后的棋盘。需要注意落子位置必须是合法位置，即没有落子过的位置，否则报错；且不能更改原有棋盘状态，而是新建一个同样的棋盘进行操作。
def result(board, action):
    """
    Returns the board that results from making move (i, j) on the board.
    """
    newBoard = [[EMPTY, EMPTY, EMPTY],
                [EMPTY, EMPTY, EMPTY],
                [EMPTY, EMPTY, EMPTY]]
    for i in range(3):
        for j in range(3):
            newBoard[i][j] = board[i][j]

    if newBoard[action[0]][action[1]] != EMPTY:
        raise NotImplementedError
    else:
        if player(board) == X:
            newBoard[action[0]][action[1]] = X
        else:
            newBoard[action[0]][action[1]] = O
        return newBoard

winner(board)根据棋盘状态给出胜利者，胜者为 X 则返回 X，胜者为 O 则返回 O，没有胜者则返回 None。
def winner(board):
    """
    Returns the winner of the game, if there is one.
    """
    if ((board[0][0] == X) & (board[0][1] == X) & (board[0][2] == X)) | (
            (board[1][0] == X) & (board[1][1] == X) & (board[1][2] == X)) | (
            (board[2][0] == X) & (board[2][1] == X) & (board[2][2] == X)) | (
            (board[0][0] == X) & (board[1][1] == X) & (board[2][2] == X)) | (
```

```
        (board[0][2] == X) & (board[1][1] == X) & (board[2][0] == X)) | (
        (board[0][0] == X) & (board[1][0] == X) & (board[2][0] == X)) | (
        (board[0][1] == X) & (board[1][1] == X) & (board[2][1] == X)) | (
        (board[0][2] == X) & (board[1][2] == X) & (board[2][2] == X)):
      return X
    if ((board[0][0] == O) & (board[0][1] == O) & (board[0][2] == O)) | (
        (board[1][0] == O) & (board[1][1] == O) & (board[1][2] == O)) | (
        (board[2][0] == O) & (board[2][1] == O) & (board[2][2] == O)) | (
        (board[0][0] == O) & (board[1][1] == O) & (board[2][2] == O)) | (
        (board[0][2] == O) & (board[1][1] == O) & (board[2][0] == O)) | (
        (board[0][0] == O) & (board[1][0] == O) & (board[2][0] == O)) | (
        (board[0][1] == O) & (board[1][1] == O) & (board[2][1] == O)) | (
        (board[0][2] == O) & (board[1][2] == O) & (board[2][2] == O)):
      return O
    return None
```

terminal(board)根据棋盘状态判断是否棋局结束，是则返回 True，否则返回 False。实现方法是遍历棋盘看是否存在 Empty。

```
def terminal(board):
    """
    Returns True if game is over, False otherwise.
    """
    win = winner(board)
    if (win == X) | (win == O):
        return True
    else:
        count = 0
        for i in range(3):
            for j in range(3):
                if board[i][j] == EMPTY:
                    count = count + 1
        if count != 0:
            return False
        else:
            return True
```

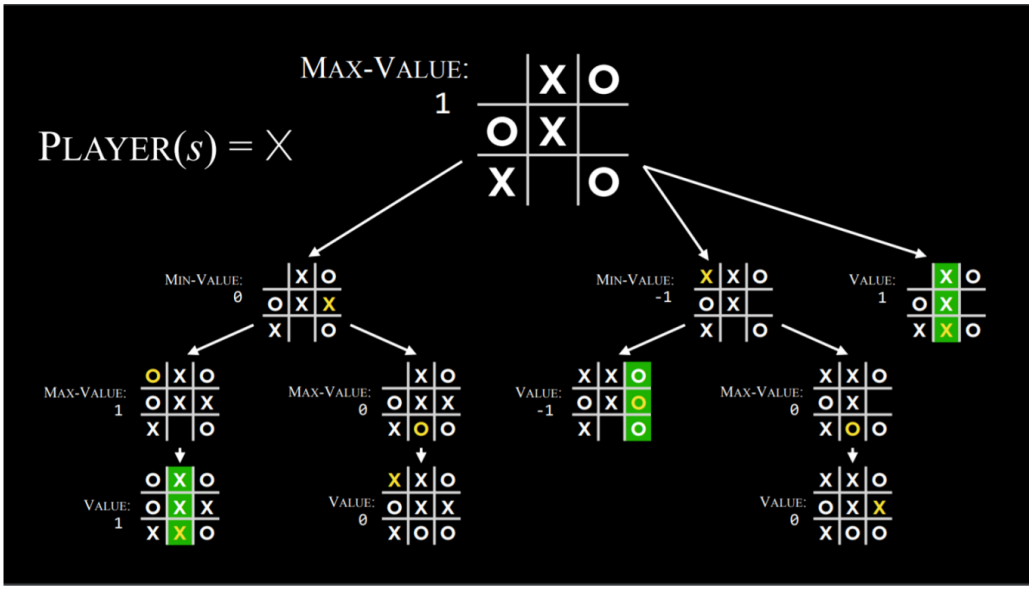
utility(board)若 X 赢则返回 1，O 赢则返回-1。

```
def utility(board):
    """
    Returns 1 if X has won the game, -1 if O has won, 0 otherwise.
    """
    Result = winner(board)
    if Result == X:
        return 1
    elif Result == O:
        return -1
    else:
        return 0
```

4.2 关键函数编写

本项目的关键函数为 minimax(board)，它根据棋盘状态给出当前落子的一方的最优落子位置，这是个典型的对抗性搜索问题，可以利用极小化极大算法求解。Minimax 将获胜条件表示为一侧的(-1)和另一侧的(+1)。进一步的行动将由这些条件驱动，最小化一方试图获得最低分数，而最大化一方

试图获得最高分数。递归地，该算法模拟所有可能发生的游戏，这些游戏可以从当前状态开始，直到达到最终状态。 每个终端状态的值是 (-1)、0 或 (+1)。根据轮到谁的状态，该算法可以知道当前玩家在最佳游戏时是否会选择导致具有较低或较高值的状态的动作。这样，算法在最小化和最大化之间交替，为每个可能的动作产生的状态创建值。最大化玩家在每一个回合都会问：“如果我采取这个行动，就会产生一个新的状态。如果最小化玩家发挥最佳，该玩家可以采取什么行动来达到最低价值？”然而，为了回答这个问题，最大化玩家必须问：“要知道最小化玩家会做什么，我需要在最小化玩家的头脑中模拟相同的过程：最小化玩家会尝试问：’如果我采取这个行动，最大化的玩家可以采取什么行动来达到最高价值？’”最终，通过这个递归推理过程，最大化玩家为每个状态生成值，这些值可能由当前状态下所有可能的动作产生。在获得这些值之后，最大化玩家选择最高的一个。



上述递归过程使用的两个关键函数伪代码如下，两个函数相互调用：

最大化最小值函数：

```
function MAX-VALUE(state):
    if TERMINAL(state):
        return UTILITY(state)
    v = -∞
    for action in ACTIONS(state):
        v = MAX(v, MIN-VALUE(RESULT(state, action)))
    return v
```

最小化最大值函数：

```
function MIN-VALUE(state):
    if TERMINAL(state):
        return UTILITY(state)
    v = ∞
    for action in ACTIONS(state):
        v = MIN(v, MAX-VALUE(RESULT(state, action)))
    return v
```

最终实现的代码如下：

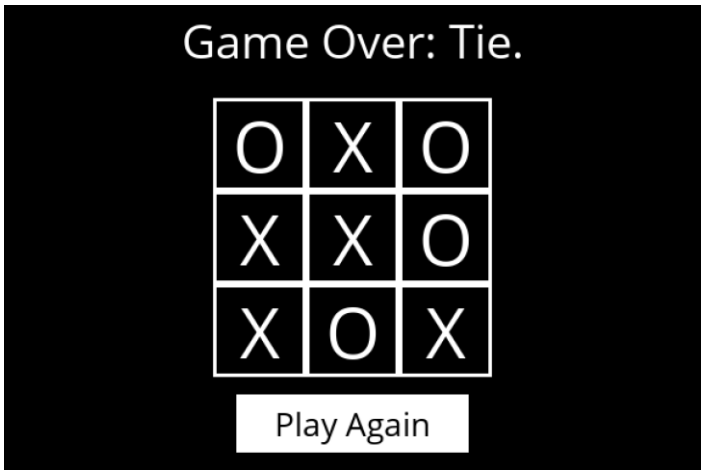
```
def minimax(board):
    """
    Returns the optimal action for the current player on the board.
    """
```

```
    turn = player(board)
    Actions = actions(board)
    Action = Actions[0]
    if turn == X:
        for action in Actions:
            if min_value(result(board, Action)) < min_value(result(board, action)):
                Action = action
        return Action
    else:
        for action in Actions:
            if max_value(result(board, Action)) > max_value(result(board, action)):
                Action = action
        return Action

def max_value(board):
    if terminal(board):
        return utility(board)
    v = -10000
    for action in actions(board):
        v1 = min_value(result(board, action))
        if v1 > v:
            v = v1
    return v

def min_value(board):
    if terminal(board):
        return utility(board)
    v = 10000
    for action in actions(board):
        v1 = max_value(result(board, action))
        if v1 < v:
            v = v1
    return v
```

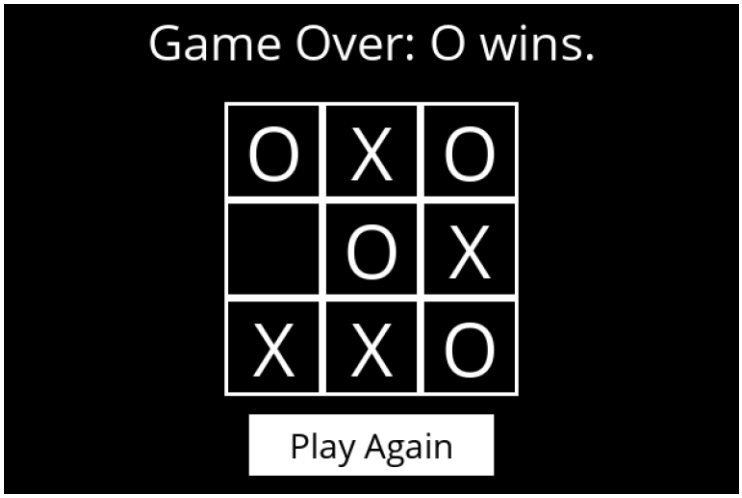
4.3 项目运行结果

玩家先手有两种可能：与 Agent 打平（tie）或者 Agent 赢（“O win”）：

Tie:

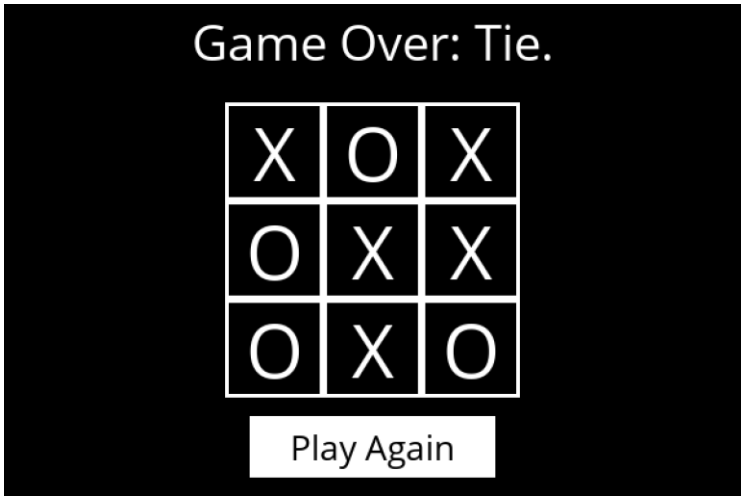


O win:

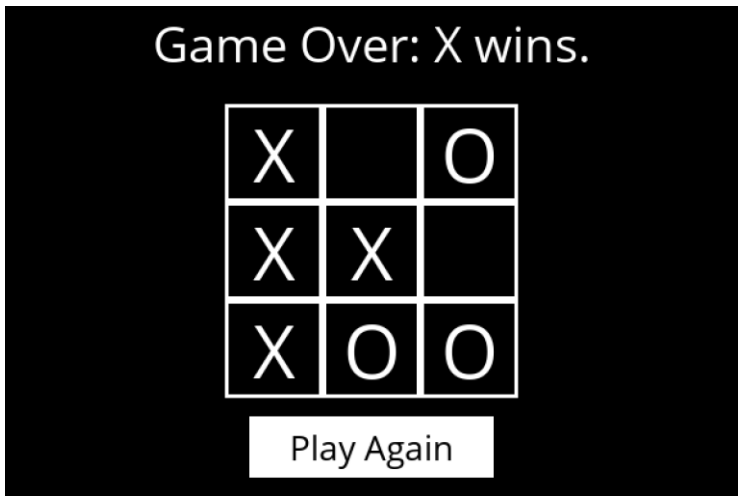


玩家后手同样有两种可能：与 Agent 打平（tie）或者 Agent 赢（“O win”）：

Tie:



O win:



玩家永远不可能胜过 Agent，项目完成。

4.4 项目优化

本人在实际测试的时候发现，玩家后手时 Agent 思考第一步下法时所需时间很长，原因是情况较多计算量很大导致的。对于这个项目来讲，可以设定玩家后手时 Agent 必定在中心落子，原因是在中心落子一定对 Agent 最有利。优化代码如下：

```
def minimax(board):
```

```
    """
    Returns the optimal action for the current player on the board.
    """
    turn = player(board)
    Actions = actions(board)
    Action = Actions[0]
    if turn == X:
        if (UUser == O) & empty(board):
            return (1, 1)
        for action in Actions:
            if min_value(result(board, Action)) < min_value(result(board, action)):
                Action = action
        return Action
    else:
        for action in Actions:
            if max_value(result(board, Action)) > max_value(result(board, action)):
                Action = action
        return Action

def empty(board):
    count = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != EMPTY:
                count = count + 1
    if count == 0:
        return True
    else:
        return False
```

实测发现这样做可以大幅提升运算效率，提升用户体验。

在本项目中可以不作优化，直接使用对抗性搜索就能给用户带来很好的体验，原因是井字棋相对简单。但在一些相对复杂的诸如围棋、象棋等问题中，直接使用对抗性搜索会使需要搜索的情况过多，超过计算资源和计算时间能够容忍的范围。这时可以采用 Alpha-Beta 修剪和深度受限的对抗性搜索。Alpha-Beta 修剪是指与普通的对抗性搜索相比跳过了一些绝对不利的递归计算。在确定一个动作的价值后，如果有初步证据表明接下来的动作可以使对手获得比已经确定的动作更好的分数，则 无需进一步调查该动作，因为它肯定不如之前建立的一个。深度受限的对抗性搜索是指在搜索停止之前只考虑预先定义的移动次数，而不会到达最终状态。但是，这不允许为每个动作获得精确的值，因为尚未达到假设游戏的结束。为了解决这个问题，深度受限的对抗性搜索依赖于一个评估函数从给定状态估计游戏的预期效用。例如，在国际象棋游戏中，效用函数会将棋盘的当前配置作为输入，尝试评估其预期效用，然后返回正数或一个负值，表示棋盘对一个玩家相对于另一个玩家的有利程度。这些值可以用来决定正确的动作，并且评估函数越好，依赖它的 Minimax 算法就越好。这就好比一个优秀的棋手能够预想到后几步棋的下法，但无法预想到整个棋局的变化一样。

5 收获与经验

我通过查阅资料，观看视频自主学习了 python，学到了一门实用的编程语言。本人在编程上的基础是 C 语言和 java 语言，并未学习过 python。但通过这次大作业我了解并基本掌握了 python 的使用方法，通过理解老师的实例代码和项目代码，改动和调试代码使我在短期内熟悉了这门编程语言。我感觉到 python 比以往我学习的任何一种编程语言还要通俗简单，极大提升了我在编写代码时的效率。我相信我在以后的学习、工作中还将不断使用 python 语言解决不同的实际问题。

在听完老师在搜索这方面通俗易懂的讲座后，我选择了搜索这个知识点完成大作业。一个原因是我对这个知识点理解最为深入，另一方面是我对这个方面最感兴趣，很想自己动手写写代码实现。理解了搜索部分的课堂示例代码 maze.py 并自己编写了其它三种搜索方案让我更加深刻地理解了不同的搜索方法之间的差异和优劣。完成哈佛 CS50 搜索的项目 1—degrees 并比较了 BFS 和 DFS 的运行结果让我理解了 BFS 和 DFS 各自的适用条件；完成哈佛 CS50 搜索的项目 2—tictactoe, 我理解并掌握了对抗性搜索的实现方法与相应的优化算法。总而言之，老师所讲的知识点通过我理解并编写代码的过程一步步印证。通过完成本次大作业我理解了很多，收获了很多。

参考文献

王一行同学的《Python 基础指南》

网站 runoob.com Python 基础教程

《Python 从入门到精通》

https://www.bilibili.com/video/BV1wD4y1o7AS?spm_id_from=333.337.search-card.all.click

第 0 讲 – 搜索 ppt

Degrees - CS50's Introduction to Artificial Intelligence with Python

Tic-Tac-Toe - CS50's Introduction to Artificial Intelligence with Python