人工智能基础 第一次大作业 小老鼠运鸡蛋

张泽楷 2019011856

一、简介：

本题是一个搜索问题，需要求解老鼠在迷宫中走动，将鸡蛋推入指定位置的最佳**（推动鸡蛋次数最少）**路径。我结合了**前向推理**（死锁剪枝）与**启发函数**，使用A^*算法求解。(有效分支因子$b^* \approx 1.1$)

在求解结果方面，可以通过计算曼哈顿距离，用可接受的启发函数精确求解，也可以通过惩罚不在目标位置的鸡蛋数量，用非可接受的启发函数极其快速地求解。而鸡蛋和箱子存在对应关系时，求解的困难度会急剧增加。

在关卡设计方面，我改编了GitHub上一系列经典推箱子地图，内置七大关卡（第零关是题目示例）。在游玩时，可以选择不同关卡以及鸡蛋和目标是否一一对应。（这些地图中已经具有多个鸡蛋，多种地形。同时比较困难，可以证明算法的有效性。鸡蛋和目标对应的情况不一定有解。）

游玩方式：

```
conda create -n AIproject1 python=3.9
conda activate AIproject1
pip install -r requirements.txt
python game.py
```

具体请参见readme.txt，以及demo.mp4。

二、算法设计(problem.py, search.py)

1.搜索问题定义：

（1）基本类的定义

对于搜索问题的定义，我使用第一次编程的模板，定义了**Node**,**Problem**以及本次大作业中需要用到的**Maze**类。

```
class Node(object):                 # Represents a node in a search tree
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.state = state
        self.parent = parent
        self.path_cost = path_cost
        self.heuristic_score = 0    # heuristic function of each node
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1
        ...
class Problem(object):
    def __init__(self, init_state=None, goal_state=None):
        self.init_state = Node(init_state)
        self.goal_state = Node(goal_state)
    def g(self, cost, from_state, pushed, to_state):
```

```
        return cost + pushed
        ...
class Maze(Problem):
    def __init__(self,
                 init_state=None,
                 goal_state=None,
                 map=None,
                 ordered=False):
        super().__init__(init_state, goal_state)
        self.map = map
        self.ordered = ordered
        self.dead_squares = self.simple_deadlock()
```

对于老鼠推鸡蛋问题，**Node**的**depth**代表它是第几步，**path_cost**代表了推动鸡蛋的次数。

而**Maze**类对象的**state**是一个词典，存储了两个list，其一是存储各个鸡蛋坐标(x,y)的'egg_pos'，其二是存储老鼠位置的'mouse_pos'：如下所示：（目标状态中由于不限制老鼠位置，'mouse_pos'=[]）。地图是固定的，用**map**这个**numpy array**来存储。

```
e.g. {'egg_pos': [(2, 2), (3, 4), (3, 7), (4, 6)], 'mouse_pos': [3, 1]}
```

按照规则，老鼠只能推动一个鸡蛋（鸡蛋不能重叠），也不能穿墙。每次检查老鼠的位置以及老鼠的四个方向**'UP'(-1,0),'DOWN'(1,0),'RIGHT'(0,1),'LEFT'(0,-1)**，如果该方向有一面墙，或者一个鸡蛋挨着另一个鸡蛋或一面墙，那么不能往对应方向移动。（当然若对应方向是空地，老鼠可以移动，只是不推动鸡蛋）。可以定义**Maze**类的**action()**如下：

```
def actions(self, state):
    possible_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
    pos = state['mouse_pos']
    # detect precomputed dead_squares and freeze deadlocks
    if any(self.dead_squares[pos[0]][pos[1]] for pos in state['egg_pos']):
        return []
    elif self.freeze_deadlock(state):
        return []

    if self.map[pos[0] - 1][pos[1]] == WALL:    # a wall on certain direction
        possible_actions.remove('UP')
    elif (pos[0]-1, pos[1]) in state['egg_pos'] # an egg on certain direction
        and ((pos[0]-2, pos[1]) in state['egg_pos'] or self.map[pos[0]-2]
[pos[1]] == -9):
            possible_actions.remove('UP')
    if self.map[pos[0] + 1][pos[1]] == WALL:
        possible_actions.remove('DOWN')
    elif (pos[0] + 1, pos[1]) in state['egg_pos']
    and ((pos[0] + 2, pos[1]) in state['egg_pos'] or self.map[pos[0] + 2]
[pos[1]] == -9):
    ...
    return possible_actions
```

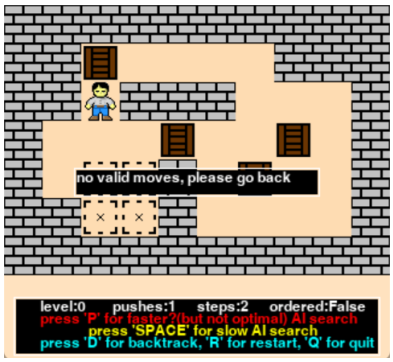
进而可以定义**Maze**类的**move()**，在移动时检测移动方向上是否有鸡蛋。若有，则推动鸡蛋，**push=True**（用于计算新节点的推动次数）；同时由于鸡蛋可能有序，必须找到被推动鸡蛋在list中的index，替换对应index处的坐标。如果鸡蛋无序，则使用**sorted()**对存储鸡蛋坐标list按第一个坐标排序。

```
def move(self, state, action):
    new_state = deepcopy(state)
    mouse = state['mouse_pos']

    pushed = False
    if action == 'RIGHT':
        if (mouse[0], mouse[1] + 1) in new_state['egg_pos']:
            pushed = True
            idx = state['egg_pos'].index((mouse[0], mouse[1] + 1))
            new_state['egg_pos'][idx] = (mouse[0], mouse[1] + 2)
    new_state['mouse_pos'] = [mouse[0], mouse[1] + 1]
...
    if not self.ordered:
        new_state['egg_pos'] = sorted(new_state['egg_pos'])

    return new_state, pushed
```

（2）前向推理剪枝

同时我参考https://github.com/BrunosBastos/SokobanAI.git，在**action()**中额外添加了两种剪枝方法：（这两种剪枝是重要的，后续加以说明）

其一是简单死锁剪枝，通过拉鸡蛋的方法粗略判断从目标位置开始，哪些位置是不可能达到的，或者说**鸡蛋被墙壁和墙角锁死**。以题目示例为例，下图就是一种简单死锁：



其二是相互死锁剪枝：由于老鼠一次只能推一个鸡蛋，如果多个鸡蛋相互靠近，它们可能都不能移动。下图是一种相互死锁：

由于代码比较繁琐，下面只给出简单死锁剪枝的部分代码。思路是如果某个位置(x,y)的右边至少有两个空位（不考虑鸡蛋），那么老鼠可能到达(x,y+1)，并且移动到(x,y+2)同时将鸡蛋拉到(x,y+1)：

```
if x in range(hor_size) and y + 2 in range(ver_size) and not (self.map[x][y + 2]
== -9):
    open_nodes.append((x, y + 1))
```

此外其实还有很多针对推箱子问题的特殊剪枝方法，如PI-corral剪枝（按区域剪枝），详见：Sokoban solver

2.搜索算法的设计：

（1）启发函数

认为最佳路径是**推动鸡蛋次数最少**（同时一般的Sokoban问题中也是推箱子次数越少越好），因此：

若鸡蛋和目标一一对应，启发函数可设计为每个鸡蛋与它的目标位置的曼哈顿距离之和。易于证明它是可接受的。

若鸡蛋和目标没有对应关系，使用贪婪算法求启发函数，对每个鸡蛋找**离它最近的目标**，计算它们的曼哈顿距离。如下列代码所示：（同样易于证明它是最优的，因为移动一步，被推动的鸡蛋对应曼哈顿距离最多减一）

```
def heuristic_function(self, state, fast=False):
    heuristic = 0
    if self.ordered:    # if ordered, calc pairwise manhat-dis
        heuristic = np.sum(np.abs(np.array(state['egg_pos']) -
np.array(self.goal_state.state['egg_pos'])))
    else:               # if not ordered, calc manhat-dis greedily
        for egg in state['egg_pos']:
            heuristic += np.min(np.sum(np.abs(np.array(egg) -
np.array(self.goal_state.state['egg_pos'])),axis=1))

    return heuristic
```

最好的方法应当是使用匈牙利算法来计算启发函数，找出n个鸡蛋与n个目标的n!种对应中,曼哈顿距离之和的最大的一个。但是使用中发现它计算非常缓慢，此处不再展开。同时，为了加快A*搜索速度，可以对启发函数进行如下所示的改进：对于每个不在目标位置的鸡蛋进行惩罚，启发函数加9999:

```
if fast:            # if use fast, penalize each egg not in goal
    for i, pos in enumerate(state['egg_pos']):
        if self.ordered:
            if pos not in self.goal_state.state['egg_pos']:
                heuristic += 9999
            else:
                if pos != self.goal_state.state['egg_pos'][i]:
                    heuristic += 9999
```

它在一般情况下非常快（因此建议游玩时尽量使用它），但是可惜的是它并不是可接受的。不保证得到最优解。如test.py中实验结果所示（在某个特定初始状态、特定目标状态下，两者结果不同）：

```
--------A* search---------
fast(but not optimal) solution
1)temporal complexity: 11021 nodes generated
2)spatial complexity: 480 simultaneous nodes
search time(FAST): 1.2293405532836914
pushes= 20 , steps= 84
--------A* search---------
slow(but optimal) solution
1)temporal complexity: 228915 nodes generated
2)spatial complexity: 1886 simultaneous nodes
search time(SLOW): 39.56592106819153
pushes= 18 , steps= 90
error!!!
```

但是在很多情况下，它是很好的估计。这是因为它相当于对节点进行分层，在不同层之间生成了"通道"，促进老鼠尽快将鸡蛋推进目标，使老鼠的运动更具有方向性；而对于同一层节点之间的细微关系，它造成的影响不大。

(2) A*搜索

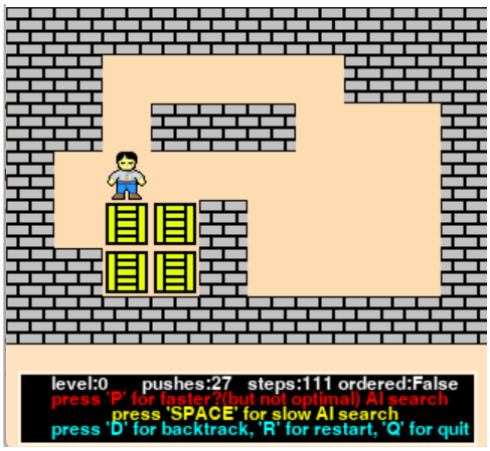
相比于启发函数的设计，A*搜索是简单的。我加入了一个变量**fast**，便于玩家选择是使用上述快速（但非最优）的启发函数，还是较慢（但严格最优）的启发函数。部分代码如下：

```
def A_star(problem, fast=False):
    nodes = 0
    open = PriorityQueue(problem.init_state)
    closed = set()
    result = -1
    while not open.empty():
        node = open.pop()
        all_pos = deepcopy(node.state['egg_pos'])
        all_pos += (node.state['mouse_pos'])
        closed.add(tuple(all_pos))
        for child in problem.expand(node):
            nodes += 1
            child.heuristic_score = problem.heuristic_function(child.state,fast)
            all_pos = deepcopy(child.state['egg_pos'])
            all_pos+=(child.state['mouse_pos'])
            if not open.find(child) and tuple(all_pos) not in closed:
            ...
    return result
```

三、GUI界面与实验结果(game.py)

1.实验结果

对于题目所给示例，如果不考虑鸡蛋与目标位置的对应，求解结果如下（27次推动，111次移动）




```
--------A* search---------
fast(but not optimal) solution
1)temporal complexity: 9210 nodes generated
2)spatial complexity: 431 simultaneous nodes
search time: 1.7299304008483887
cost= 27 , steps= 111
--------A* search---------
slow(but optimal) solution
1)temporal complexity: 874279 nodes generated
2)spatial complexity: 2325 simultaneous nodes
search time: 170.2259407043457
pushes= 27 , steps= 115
```

发现这种情况下，使用不严格的启发函数（$1.730s$）确实远快于严格的启发函数（$170.23\ s$），而且也可以得到最优解。同时不严格的启发函数有效分支因子$b^* = 9210^{\frac{1}{111}} = 1.086$，严格启发函数的有效分支因子为$b^* = 874279^{\frac{1}{115}} = 1.126$

此外通过ablation test发现，在不进行死锁剪枝只使用启发函数时，对于题目中给的问题，在有限时间无法解决。因此剪枝很重要，对于剪枝和启发函数各自的作用，也可以参考如下链接：Sokoban solver，或如下图所示：

My program uses precalculated deadlocks to handle simple deadlocks like this one, but the corral-pruning is a valuable contribution to the program, and experiments shows that the corral-pruning almost can hold its candle to the precalculated deadlocks, if it is asked to do so:

Statistics - SokEvo/107 - Forwards search - no packingorder - AMD 2600+

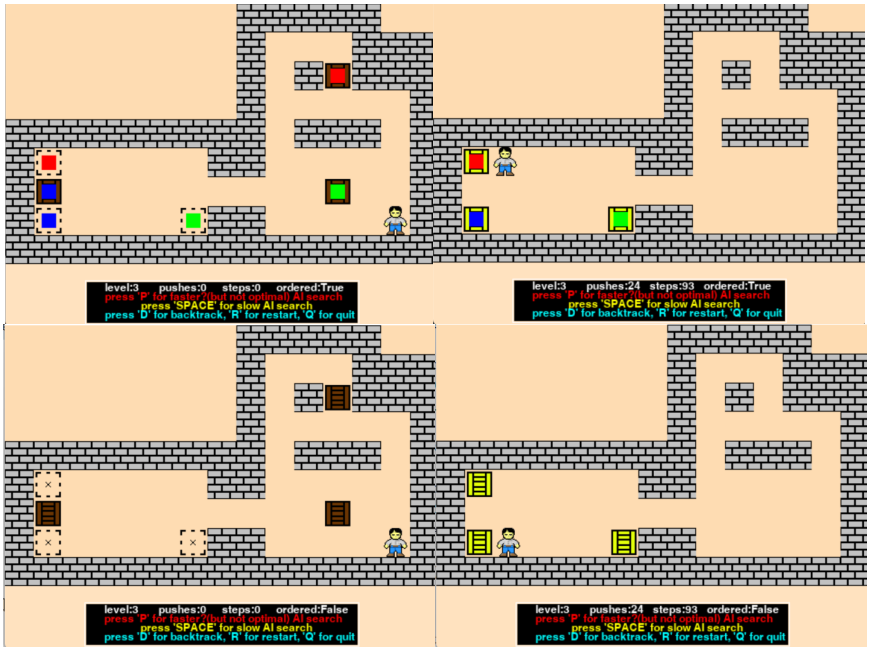
A. A* search + transposition table: 11974023 positions 51938064 pushes 65 seconds

B. A* search + transposition table + corrals: 4632203 positions 11918161 pushes 21 seconds

C. A* search + transposition table + precalculated deadlocks: 2030736 positions 7011203 pushes 11 seconds

D. A* search + transposition table + precalculated deadlocks + corrals: 1231376 positions 3299797 pushes 7 seconds

但是如果鸡蛋和目标位置有对应，搜索大概率无解（因此游玩时请尽量不选择**"ordered"**，即尽量使鸡蛋无序）。此时不用题目所给示例，用自创的一关显示鸡蛋和目标对应情况下的搜索：



发现在有解的情况下，有对应的搜索反而比无对应快。这并不奇怪，因为有对应时**启发函数更接近实际代价**。

```
--------A* search---------      # ordered
fast(but not optimal) solution
1)temporal complexity: 31932 nodes generated
2)spatial complexity: 1592 simultaneous nodes
search time: 2.8135478496551514
cost= 24 , steps= 93
--------A* search---------      # not ordered
fast(but not optimal) solution
1)temporal complexity: 308670 nodes generated
2)spatial complexity: 12138 simultaneous nodes
search time: 46.11959624290466
cost= 24 , steps= 93
```

2.游玩方式

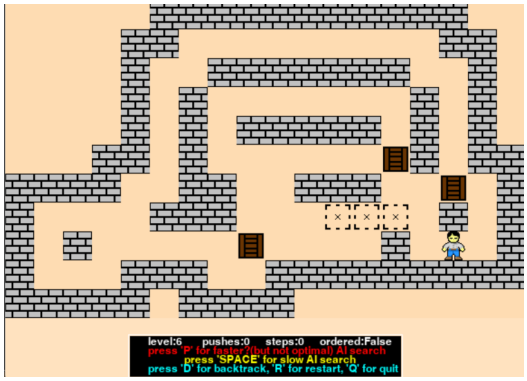
可视化主要改编自：https://github.com/morenod/sokoban.git，代码不在报告中具体给出。

游玩方式为：在终端输入

```
python game.py
```

并依次输入关卡序号，鸡蛋和目标是否一一对应，开始游玩。

游玩时，按上下左右键移动，按'D'回上一步，按'R'重启关卡，按'P'快速求解当前关，按'空格'严格求解当前关：求解后按顺序显示解的各步，并在终端输出相关结果。



值得一提的有两点：

（1）**Pygame**中**screen**的第一维沿沿水平方向，第二维沿竖直方向。因此把矩阵中(row,col)的元素显示到屏幕上，实际上使用的是(col,row)。如下所示：

```
if (row, col) in state['egg_pos']:
        screen.blit(egg_docked, (IMG_SIZE * col, IMG_SIZE * row))
```

（2）求解后要按顺序播放解的各步，不要在**Pygame**的主循环中再生成**for**循环播放。因为它是按照一定间隔刷新，如果播放时间太长，会卡住。因此借助主循环，通过计数器实现一步一帧播放。

```
if result:
    print_game(game, result[result_show_step].state, screen)
    display_state(screen, result[result_show_step])
    pygame.display.update()
    pygame.time.wait(100)
    if result_show_step == len(result) - 1:
        pygame.time.wait(900)
        result = []
        result_show_step = -1
    result_show_step += 1
```

同时项目文件夹中的demo视频也演示了游玩方式。

四、总结与反思

1.对我而言，最重要的一点是学会分摊算法的复杂度。在此处的推箱子问题中，如果在**action()**中只引入最简单的判断而不剪枝，只靠启发函数剪枝，效率极低。此外学会了**Pygame**，很有趣。

2.问题定义方面，主流的Sokoban求解算法将整个地图作为状态，同时定义的**action()**是"对箱子的一步推动"（用BFS检验能否推动）。很明显这样做效率更高。因为我们只关心鸡蛋被推动的次数，不关心老鼠是如何到达一个鸡蛋从而推动它的。我为了兼顾玩家的使用，将老鼠的一步作为**action**。

3.求解速度还是稍慢。但是实际上剪枝几乎已经到了极致（题目示例情况下$b^* \approx 1.1$）。真正拖慢速度的原因是Python本身太慢。参考前述图D.中作者生成了百万数量级的节点，只需$7\ s$。而我的实现中最复杂的情况也是生成百万数量级的节点，但是需要$180\ s$左右。