作业 1：Bait 游戏 实验报告

郭子良（221240056、221240056@smail.nju.edu.cn）

1 顶层设计

如果只依赖原始的 python 框架，很难便捷地完成这次实验的任务。下面的两个设计可以明显降低设计搜索算法时候的难度和思考量，我们将在之后的每个任务中都采用这些设计：

1.1 搜索树节点 Node

仿照 MCTS 中的设计，对每个算法，我们在考虑这个算法具体如何实现之前，先考虑这个算法对应的搜索树的节点上应该保存哪些信息，并把这些信息都作为成员变量打包成一个 Node，从而把原来的游戏问题方便地抽象回图上的搜索。

一般地，Node 要保存以下信息：

1. 直到抵达这个节点为止，全体 action 构成的集合。方便在最后通过一个特定的节点（如 goal）还原整个路径。
2. 到达当前节点时 env 的拷贝，方便评估当前 env 或者判断当前状态是否已经探索过。
3. 针对特定算法的辅助信息，比如 LDFS-Node 应当保存深度信息。

具体的 Node 涉及将在后面介绍。

1.2 避免搜索回路

在最简单的搜索问题中，我们往往只需要通过图上节点的 index 就得出当前点 A 是否和已经搜索点集中的某个点 A’ 是同一个点，而在本次实验中，我们首先要解决判断两个 Node 是否相等的问题。

阅读源代码不难发现，env 有一个成员变量 grid（二维数组）保存了地图上每一个格子的信息，比如这个格子有没有坑洞或者箱子，或者站在这个格子上的玩家（如果有）是否持有钥匙。容易看出，如果两个 Node 中 env.grid 是一致的，那么这两个 Node 在这个游戏中就应该被视为同一个 Node。另外 env.grid 可以通过 env 的成员函数 getobs 来获取。

env.py ↗ …

```
  6  class BaitEnv:
128      def step(self, action):
153
154          # Get the entities in the new cell
155          new_cell = self.grid[new_y][new_x]
156          current_cell = self.grid[y][x]
157          reward = 0
158          info = {}
159
160          # Get the current avatar type
161          if 'avatar_nokey' in current_cell:
162              avatar_type = 'avatar_nokey'
163          elif 'avatar_withkey' in current_cell:
164              avatar_type = 'avatar_withkey'
165          else:
166              # Should not happen
167              raise Exception('Avatar not found in current cell')
```

判断 Node 是否相等的函数如下所示（以 getobs 的返回值作为输入）：

```
def state_equals(self, state1, state2):
    for row in range(len(state1)):
        for col in range(len(state1[0])):
            if state1[row][col] != state2[row][col]:
                return False
    return True
```

在此基础上可以进行进一步的操作，比如判断回路，是需要把当前 Node 和 Visited 集合中的每一个 Node 依次比较就行。（注：这个方案其实有改进的空间，比如利用玩家的当前 x, y 坐标可以排除很多没有比较意义的 Node，笔者在写报告的时候才意识到。）

判断 Node 相等以及判断 Node 是否处于 Visited(DFS), Open, Closed(A-star) 集合中的方法下面将不再具体介绍。

接下来介绍每个任务的具体实现，最后会统一展示和讨论算法的运行结果。

2 任务 1：DFS(level-0)

2.1 Node 设计

参考之前的叙述，只需要保存最基本的信息就行。

```
class DFSNode:
    def __init__(self, env, parent=None, action=None):
        self.env = env
        self.parent = parent
        self.action = action
        if parent:
            self.actionlist = parent.actionlist + [action]
            self.depth = parent.depth + 1
        else:
            self.actionlist = []
            self.depth = 0
```

2.2 算法设计

原问题已经转化成了图上的一个搜索算法，其中每个点和当前 env 经过四种 action 产生的四个新节点（如果动作不合法则不存在对应的新 Node）相邻。从 reset 之后的状态构建一个 root 节点，然后开始 DFS，过程中用 Visited 集合保证每个节点都只访问一次。最后搜索到 Goal-Node，返回保存在这个 Node 中的 action-list 即可。

需要注意的是，如果新探索的节点代表了玩家死亡的局面（体现为 step 的返回值为 done 但是新 env 中 goal 仍然存在），我们就需要跳过这个节点，后面将不再解释这一点。

具体实现比较简单，这里略过。

算法运行结果在后面展示。

3 任务 2：LDFS (level-0)

3.1 Node 设计

和 DFS 相比，LDFS 首先需要保存当前的搜索深度；同时为了方便评估当前 Node 是否是“优秀的”（用 Score 来衡量），还需要记录当前的角色位置，以及角色是否拥有 key （没拿到 key 时候的优化目标和拿到了 key 之后的目标应该是不同的，对应的评分方案也应该不同，同时拿到了 key 的状态应当比没拿到 kye 的状态更优）。

```
class LDFSNode:
    def __init__(self, env, parent=None, action=None):
        self.env = env
        self.parent = parent
        self.action = action
        self.score = float(1e9)
        self.x, self.y = env.avatar_pos
        if 'avatar_withkey' in env.grid[self.y][self.x]:
            self.haskey = True
        else:
            self.haskey = False
        if parent:
            self.actionlist = parent.actionlist + [action]
            self.depth = parent.depth + 1
        else:
            self.actionlist = []
            self.depth = 0
```

3.2 算法设计

首先需要一个评估函数，以 Node 作为输入，如果已经获取了 key，则评分为到 goal 的曼哈顿距离，否则为到 key 的曼哈顿距离再加上 key 到目标的曼哈顿距离（其实就是从当前开始完成游戏的理论最小步数）。

```
def get_node_score(self, node):
    nodepos = (node.x, node.y)
    if node.haskey:
        return abs(nodepos[0] - self.goalpos[0]) + abs(nodepos[1] - self.goalpos[1])
    else:
        return abs(nodepos[0] - self.keypos[0]) + abs(nodepos[1] - self.keypos[1]) \
            + abs(self.goalpos[0] - self.keypos[0]) + abs(self.goalpos[1] - self.keypos[1])
```

在这个基础上从当前状态为 Root-Node 进行 LDFS，当搜索的深度超过限制的时候提前终止，如果搜索到了目标就返回目标 Node，如果最后也没有搜索到目标（因为只完成了对整个图的部分搜索）就返回搜索过程中发现的最优秀（score 最小）的 Node，并指导 Agent 按照 Node.actionlist 中的第一个 action 行动。

算法有三个细节：**第一是搜索多步但是只有第一步被执行，这样可以避免陷入局部最优解（6.2.2）**；第二是每次搜索之前清空 Visited，否则之前探索过但是没有被走过的 Node 会干扰搜索；最后，每个 Node 在入栈的时候都应该及时计算并更新 Score。

```
while len(sstack) > 0:
    node = sstack.pop()
    # print("now node depth", node.depth)
    if self.tick > self.tick_max:
        break
    if node.depth > self.maxdepth:
        continue
    if node.depth == self.maxdepth:
        continue
```

```
    for action in D_ACTIONS:
        newenv = deepcopy(node.env)
        obs_, reward, done, _ = newenv.step(action)
        self.tick += 1
        if done:
            if not newenv.goal_exists:
                return node.actionlist + [action]
            else:
                continue
        if self.state_has_visited(obs_):
            continue
        self.visited.append(obs_)
        newnode = LDFSNode(newenv, node, action)
        newnode.score = self.get_node_score(newnode)
        if newnode.score < best_score:
            best_score = newnode.score
            best_node = newnode
        sstack.append(newnode)
assert best_node is not None
return best_node.actionlist
```

4 任务 3：A-star (level0, 1, 2, 3)

和上一个算法相比， A-star 算法同样是找出搜索的有限范围之内最优秀的 Node 并返回，只是探索的先后顺序上不同：A-star 算法总是探索当前待探索的节点中最优的节点。

4.1 Node 设计

成员同 LDFS，但是定义了 Node 之间的比较函数来使得我们后面可以方便地使用优先队列。

```
def __lt__(self, other):
    return self.score < other.score
```

4.2 算法设计

原始的 A-star 算法中，Score 应当由 f 和 g 两个部分组成，其中 f 一般是已经付出的代价，但是在本任务中我们只需要考虑如何取得游戏胜利就行，reward 和步数都不用考虑，所以为了简化不妨 f=0，此时 score 可以直接用 LDFS 中设计的启发式函数，我们只需要考虑如何搜索就行。

4.2.1 A-star：准备工作

在每次搜索开始之前，应当初始化 Open-list 为空，而 Close-list 如果按照 LDFS 的思路，也可以初始化成一个空集，但是这里有可以改进的空间——虽然被探索却没有被走过的 Node 应该保留被探索的机会，但是所有被探索且最后被走过的 Node 不需要再一次被探索，所以我们可以将已经走过的 Node 保存在一个集合 have-reached 中，并在搜索开始前把 Close-list 初始化成前者（注意这里应当使用 deepcopy 而不是直接用一个等号进行赋值，后者实际上是引用）。

4.2.2 A-star：探索新的 Node

从 Open-list 中寻找当前最优的节点 A 并考虑它的子节点 B，如果 B 是已经探索过的节点（检查 Close-list），或者死亡节点（见 2.2）就跳过，如果已经在 Open-list 中（同样地，检查 Open-list），按照原本的 A-star 算法应该比较 old-B 和 new-B 的 score 并更新，但其实在本次任务中 Node 的 score 是既定的（只由对应的游戏当前局面决定），更新不更新没区别。最后，如果 B 是一个新的未探索节点，加入 Open-list。所有 B 探索完之后把 A 加入 Close-list。

```
def astar(self):
    self.openlist = []
    # use deepcopy, not use =, to avoid reference(to see in note)
    self.closedlist = deepcopy(self.have_reached)
    rootnode = AstarNode(self.env)
    rootnode.score = self.get_node_score(rootnode)
    heapq.heappush(self.openlist, rootnode)
    bestnode = rootnode
    bestscore = rootnode.score
    while len(self.openlist) > 0:
        current = heapq.heappop(self.openlist)
        # print("Current depth and score:", current.depth, current.score)
        if current.depth > self.maxdepth:
            continue
        self.closedlist.append(current.env._get_observation())

    for action in D_ACTIONS:
        newenv = deepcopy(current.env)
        _obs, reward, done, _ = newenv.step(action)
        if done:
            if not newenv.goal_exists:
                return current.actionlist + [action]
            else:
                continue
        if self.state_is_closed(_obs):
            continue
        newnode = AstarNode(newenv, current, action)
        newnode.score = self.get_node_score(newnode)
        # print("New node score:", newnode.score)
        if newnode.score < bestscore:
            bestnode = newnode
            bestscore = newnode.score
        openindex = self._open_node_search(newnode)
        if openindex == -1:
            heapq.heappush(self.openlist, newnode)
        else:
            if newnode.score < self.openlist[openindex].score:
                self.openlist[openindex] = newnode
assert bestnode.actionlist is not None
return bestnode.actionlist
```

5 任务 4：MCTS

5.1 顶层设计

由 MCTS-Agent 指导的 act 函数如下：

1. 初始化一个蒙特卡洛搜索树(MCTS)。
2. 调用 MCTS 的 run 方法完成树的构建。
3. MCTS 构建完成之后，找出根节点的子节点中被访问最多的那个，其对应的 move 就是当前所求的最优 action。

```
def act(self, env):

    self.mcts = MCTS(self.tick_max, copy.deepcopy(env))
    self.mcts.run()

    node = self.mcts.root

    node = max(node.children, key=lambda node: node.visits)

    return node.move
```

5.2 MCTS-Node

在讨论 MCTS 的构建细节之前，我们先讨论蒙特卡洛搜索树中 Node 的结构。

首先是成员变量。和前面的各种算法相比，有一些不同：首先是用一个 list 记录的当前节点的所有 child ；其次用了名为 state 的数组来存放 action-list，这和之前的命名不同。

```
class Node:
    def __init__(self, move='', env='', parent=None):
        self.move = move
        self.parent = parent
        self.children = []
        self.visits = 0
        self.score = 0
        self.env = env
        if parent is None:
            self.state = []
        else:
            self.state = self.parent.state + [move]
```

其次是成员函数。Expand 函数用来探索当前节点的随机一个未探索子节点：

```
def is_fully_expanded(self):
    return len(self.children) == len(ACTIONS)

def expand(self):
    moves_tried = [child.move for child in self.children]
    untried_moves = [move for move in ACTIONS if move not in moves_tried]
    move = random.choice(untried_moves)
    env_child = copy.deepcopy(self.env)
    env_child.step(move)
    child_node = Node(move=move, env=env_child, parent=self)
    self.children.append(child_node)
    return child_node
```

Simulate 函数用来模拟当前 Node 随机行动一段时间之后的局面；backpropagate 函数用来沿着父节点一路回溯来更新搜索的结果：

```
def simulate(self):
    if self.env.done:
        return (0, 1) if self.env.goal_exists else (5, 1)
    env = copy.deepcopy(self.env)
    score = 0
    done = False
    used_ticks = 0
    while not done:
        action = random.choice(ACTIONS)
        _, reward, done, _ = env.step(action)
        used_ticks += 1
        score += reward
        if done or len(self.state) + used_ticks > 20:
            break
    return score, used_ticks

def backpropagate(self, result):
    self.visits += 1
    self.score += result
    if self.parent:
        self.parent.backpropagate(result)
```

5.3 MCTS 的构建

搞清楚了 Node 的含义，就能很方便地理解 MCTS 的构建过程了。**我们先指出 MCTS 的主要思想：根据一个策略进行行动来探索整个状态空间，并根据探索的结果来更新当前策略。**

```
def ucb1(self, node):
    Q = node.score / (node.visits + 1e-5)
    N = node.parent.visits
    n = node.visits
    c = math.sqrt(2)
    value = Q + c * math.sqrt(math.log(N + 1) / (n + 1e-5))
    return value + random.random()

def run(self):
    used_ticks_total = 0
    while used_ticks_total < self.tick_max:
        leaf = self.select()
        if not leaf.is_fully_expanded():
            leaf = leaf.expand()
        score, used_ticks = leaf.simulate()
        leaf.backpropagate(score)
        used_ticks_total += used_ticks
```

我们结合多臂老虎机的模型来具体解释这个思想：

对一个给定的 Node A，考虑 A.env 对应的所有 (env, action, new-env) 元组，在当前 env 下的每一种可能的动作在执行之后都会最终带来一些回报(就像是一个完全未知的老虎机，每一个摇臂都会返回某种奖励)，不同的 action “最终带来高回报”的能力肯定不是相同的，但我们对这样的“能力”是完全不了解的。

从概率上讲，每种 action 最终带来的回报值是一个随机变量，有一个期望 E ，但是这个 E 也是未知的；在实践中，我们用一个函数 ucb1 来近似表述这个“能力”(ucb1 的计算方式见上面的 ucb1 函数，可以看出他兼顾了已经探索过的所有最终回报的均值（经验部分）和 action 的潜力（exploration 部分))。在当前策略的指导下，Agent 会始终选择可选 action 中使得 ucb1 最大的分支，并不断沿着已经探索的树的部分往下，直到发现一个 “leaf”，这就是 select 函数。

结合上面的叙述，我们来具体解释 select 函数：如果当前节点有未被探索过的 children，那对应的 ucb1 直接就是正无穷（潜力很大），选择这个 children 作为 leaf（一个完全没被探索过的新 state）并返回。否则就按照 ucb1 最大的贪心策略继续运行；如果当前节点已经是整个状态空间的叶子（env.done），那也没关系，我们还是找一个他的 children 作为 leaf，因为这个新节点和原来的节点是一样的。**总之， select 会返回一个全新的 state 或者终止的 state，现有策略对这样的 state 已经失去了意义。**

然后，就可以用这个 state 来更新现有策略，通过 simulate 来评估这个节点，并通过把这个新节点的 score 一步步 backpropagate 来更新对应的父节点对于其选择的 action 的估计，从而完成整个策略的更新。

当最终的 MCT 构建完成，策略也最终确定了，在这个过程中被最多次选择的根节点的子节点自然可以被认为是代表了一个比较明智的选择，这就是 MCTS 算法。

6 算法的运行和分析

6.1 DFS, LDFS(level-0)

没啥好分析的，DFS 能过 level0，跑的够久可以过 level1，几乎过不了 level2。

LDFS 如果 max-depth 只开到 5 基本上只能过 level0，当然这个参数是可以调的，但是考虑到我几乎把 Astar 写的和 LDFS 一模一样，我们放在下一节一起讨论。

6.2 A-star(level-1, 2, 3)

为什么 LDFS 如果 max-depth 只开到 5 基本上只能过 level0 ？问题的关键是，要顺利通过 level1 以及以后的关卡需要一些复杂的行为，简单的曼哈顿距离贪心没有办法建模这些行为，比如在 level2 中你需要先远离 key 来解开一个箱子的限制，并把蘑菇吃掉才能最终获胜，如果一直贪心就会把一个箱子推进死胡同（这里并不赘述）。总之如果 max-depth 比较小，用现有的启发式函数（这个游戏要设计一个好的启发式函数感觉挺难的）。

6.2.1 Max depth 怎么样比较合适？

如果不考虑剪枝，max depth 每增加 1 最终的探索时间都会乘上 4 。

但是 max depth 至少要能保证能够搜索到“这局游戏中最核心的决策对应的一系列动作”，为此可能需要你先玩一把这一关。

6.2.2 为什么搜索若干步只走一步？(3.2)

我们以一个 Astar 算法运行过程中的一些 log 来解释：

```
Action list: [3, 3, 3, 3, 2, 2, 2, 2, 3, 3]
Step: 5, Action taken: 3, Reward: 1, Done: False, Info: {'message': 'Box fell into hole.'}
Action list: [3, 3, 3, 2, 2, 2, 2, 3, 3]
Step: 6, Action taken: 3, Reward: 0, Done: False, Info: {}
Action list: [3, 3, 2, 2, 2, 2, 3, 3]
Step: 7, Action taken: 3, Reward: 0, Done: False, Info: {}
Action list: [3, 2, 2, 2, 2, 3, 3]
Step: 8, Action taken: 3, Reward: 0, Done: False, Info: {}
Action list: [2, 2, 2, 2, 3, 3]
Step: 9, Action taken: 2, Reward: 0, Done: False, Info: {}
Action list: [2, 2, 2, 3, 3]
Step: 10, Action taken: 2, Reward: 0, Done: False, Info: {}
Action list: [2, 2, 3, 4, 2, 4, 4, 1, 3, 3, 3, 3, 3]
Step: 11, Action taken: 2, Reward: 0, Done: False, Info: {}
Action list: [2, 3, 4, 1, 1, 3, 2, 4, 2, 3, 3, 3]
```

一开始，因为搜索深度的限制，Agent 搜索到的最优 Node（Action list 表示从当前 Node 采取一系列动作可以到达目标 Node）是一个局部最优解，但是他和最优解的前几步是相同的，所以走到 step10 之后，它就可以在有限的时间内搜索到一个比局部最优解更高明的路线，避免了陷入之前的局部最优陷阱。

如果没有这个技巧，max depth 就要开的很大了。

6.2.3 启发式函数的弊端

尽管有 6.2.2 的技巧，max depth 往往还是要开的很大，而且其实理论上超过 18 (complexity: 2^36) 问题就完全不可解了。

而且尽管曼哈顿函数是这种游戏最常见的启发式函数，但是在这个需要一点“策略”的游戏里面真的很鸡肋，这里不作赘述。