Bait 实验报告

宋磊 181220049 974534426@qq.com

（南京大学 人工智能学院）

1 DepthFirst

1.1 思路

在当前节点通过枚举尝试每个动作，找到解路径，之后记录下来，之后的每次 act 都按照顺序选择动作并执行，最终完成任务。

如图是对相同状态的判重，避免形成环：

```
if(isVisited(stCopy))
    continue;
```

```
public boolean isVisited(StateObservation stateObs)
{
    for(StateObservation state: flag)
        if(state.equalPosition(stateObs))
            return true;
    return false;
}
```

如图维护了一个 ArrayList 类型的 ans 解路径，如果从当前节点可以找到路径，则直接返回，否则 remove 该节点的路径，最后只需要按照 ans 的顺序执行 Action。

```
if(DFS(stCopy))
    return true;
else
    ans.remove( index: ans.size()-1);
```

1.2 代码结果

代码可以完成前三关，但由于是在使用深度优先搜索找到路径后直接返回，不是对树的完全搜索，找到的不一定是最优路径，例如：在第一关拿到钥匙返回目标时会绕远路，在第二关中，由于箱子位置改变，也会导致 state 的变化，使得 state.equalPosition()方法的到 false，所以会走很多在人看起来重复的路，第三关较为良好。

2 limitDepthfirst

2.1 代码解释

2.1.1 整体思路

在第一问的基础上，增加深度的参数，每次递归层数增加时，对应增加深度，达到深度时停止扩展并通过启发式函数判断是否优于当前最优解，并进行更新。与第一问的不同是在每步都进行一次深度受限搜索，得到当前最优 Action 后执行，下一步重新搜索。

2.1.2 启发式函数

对启发式函数在下面的 Astar 算法 3.2 启发式函数设计中进行详细讨论。这里只进行简单分析。通过官网相关资料，发现可以通过 state.getAvatarType()判断精灵是否拿到钥匙，简单通过这个函数进行判断，分为三种情况，已经拿到钥匙，对应第一个分支，启发式函数变为当前位置到目标的 l1 距离。未拿到钥匙但已经找到了通往钥匙的道路，对应第二个分支，则直接向钥匙前进，启发式函数为现在位置到钥匙的距离，未拿到钥匙切未找到通往钥匙的路，对应最后一部分，此时需要判断不会让箱子压到钥匙。

```
if(is_get_key == 1) //得到钥匙
{
    return l1_norm(now, goalpos);
}
else
{
    if(state.getAvatarType() == 4)
    {
        best_action = now_action;
        best_f = (int) (Math.abs(now.x - keypos.x) + Math.abs(now.y - keypos.y));
        return l1_norm(now, keypos);
    }

    ArrayList<Observation>[] movingPositions = state.getMovablePositions();
    box1 = movingPositions[1].get(0).position;
    box2 = movingPositions[1].get(1).position;
    if(box1.equals(keypos) || box2.equals(keypos))
        return Integer.MAX_VALUE;

    return  l1_norm(now, keypos);
}
```

2.1.3 最优动作的选择

对当前最优动作的记录，由于是树状搜索，所以只需要在深度为 1 时记录 now_action，之后如果在深度较深时发现更优解，一定是由 now_action 扩展得到，则将 best_action 更新为 now_action，在搜索完成之后执行 best_action。

```
if(depth == 1)//记录第一个动作
{
    now_action = action;
    flag.clear();
    flag.add(stateObs);
}
```

如图是相比于第一问增加的对深度的判断，并更新当前最优解。

```
if(depth > DEPTH)
{
    if(h(stateObs) < best_f)
    {
        best_action = now_action;
        best_f = h(stateObs);
    }
    return;
}
```

2.2 代码结果

在 depth 适当比如 3 时，通过第一关，但在第二关中需要加深深度，分析认为，在深度为 3 时，做出的 Action 不会对局面造成影响（这里说的是向钥匙的靠近，因为没有考虑箱子和洞的位置关系），所以需要对启发式函数进行改良，具体改良见 Astar 中的启发式函数。

3 Astar

3.1 思路

维护 openList 和 closedList 的优先队列，在循环寻找路径时，将 openList 中优先级最高的元素取出，移入 closedList 中，表示该点已经访问过，对该点可以执行的动作进行执行，得到新的状态，如果新的状态没有访问（即不在 closedList 中）并且不在 openList 中，或是在 openList 中但代价 g 值更小，则将代价更小的状态加入 openList 中，并通过 java 的 priority_queue 自动维护堆性质。

通过不断访问当前最优的状态，得到最优路径。代码的前部为取出最优状态并进行扩展，如图

```
while(!openList.isEmpty() && elapsedTimer.remainingTimeMillis()>remai
{
    Node node = openList.poll();
    closedList.add(node);

    for(int i = 0; i < actions.size(); i++)
    {
        StateObservation stCopy = node.stateObs.copy();
        action = actions.get(i);
        stCopy.advance(action);
```

之后对新状态进行合法性检验，如果位置没有发生变化，或是精灵已经死亡，或该状态已经访问过，则直接退出该次动作，不将其加入 openList 中。如果是一个合法的状态，并且不在 openList 中或比原本 openList 中的同一状态更优，则对 openList 中状态更新为更优解，如图：

```
Node now = new Node(stCopy);
now.g = node.g + 1;
now.parent = node;

if(stCopy.getGameWinner() == Types.WINNER.PLAYER_WINS)
{
    best_state = now;
    return;
}
else if(stCopy.isGameOver() && stCopy.getGameWinner() == Types.WINNER.PLAYER_LOSES)
{
    continue;
}

Node pre = isInOpen(now);

if((pre != null && now.g+get_h(now.stateObs) < pre.f) || pre == null)
{
    now.h = get_h(now.stateObs);
    now.update_f();
    openList.add(now);
}
if(now.f < best_f)
    best_state = now;
```

通过后一个动作找到最开始执行的动作的代码对网上进行了参考，与之前同样的思路，动作的执行扩展为一棵四叉树，通过记录其父节点 parent，进行回溯，找到最开始执行的动作，并执行。代码中是找到了执行完第一个动作后的状态，通过 getAvatarLastAction()函数找到上一个动作，并返回（所以在循环判断条件中要保证 p.parent.parent 不为空）。

```
public Types.ACTIONS getAction()
{
    Node p = best_state;
    while(p!= null && p.parent != null && p.parent.parent != null)
    {
        p = p.parent;
    }
    if(p != null)
        return p.stateObs.getAvatarLastAction();
    return null;
}
```

3.2 启发式函数设计

3.2.1 第一关

最初设计的启发式函数时最简单的，先分情况讨论是否拿到钥匙，以及箱子不能压到钥匙的简单判断条件，再用 l1 范数计算距离，但之后发现这样简单的函数只能通过第一关，随后进行了改良。

```
if(is_get_key == 1) //得到钥匙
{
    return l1_norm(now, goalpos);
}
else
{
    if(state.getAvatarType() == 4)
    {
        best_action = now_action;
        best_f = (int) (Math.abs(now.x - goalpos.x) + Math.abs(now.y - goalpos.y));
        return l1_norm(now, goalpos);
    }

    ArrayList<Observation>[] movingPositions = state.getMovablePositions();
    box1 = movingPositions[1].get(0).position;
    box2 = movingPositions[1].get(1).position;
    if(box1.equals(keypos) || box2.equals(keypos))
        return Integer.MAX_VALUE;

    return  l1_norm(now, keypos);
}
```

3.2.2 第二关

第二关过不去的主要原因认为是在之前的函数中，只计算了自身和钥匙的距离，所以精灵只会向钥匙靠近，而不知道将箱子推到洞上，这样肯定是不会通关的，所以加入了钥匙和箱子的相对位置的 l1 范数，将箱子向洞靠近，也可以得到更小的 h，从而让精灵知道用箱子填洞。

代码前半段是通过 fixedPosition 和 movingPosition 获得箱子和洞的位置，后面通过二重循环计算每一个箱子和洞的 l1 范数，记录为 h，最后的返回结果中加 h，从而使得降低箱子和洞的距离也可以获得更好地评分。

```
ArrayList<Observation>[] fixedPositions = state.getImmovablePositions();
ArrayList<Observation>[] movingPositions = state.getMovablePositions();

ArrayList<Observation> holes = null, boxes = null;
if (fixedPositions.length > 2)
    holes = fixedPositions[fixedPositions.length - 2];
if (movingPositions != null)
    boxes = movingPositions[movingPositions.length - 1];

int h = 0;
//计算箱子和洞的距离
if (boxes != null && holes != null && boxes.size() > 0 && holes.size() > 0)
{
    for(Observation box: boxes)
    {
        for(Observation hole: holes)
        {
            h += l1_norm(box.position, hole.position);
        }
    }
}

return h+l1_norm(nowpos, keypos);
```

3.2.3 第三关

但这样的代码依然是有问题的，在第二关中表现良好，路径也是最优，但第三关中，精灵总是希望把所有箱子都推到洞上，但其实只需要将左边的推上就可以吃到钥匙和蘑菇，进而完成任务。

第三问遇到的问题是精灵填完左边的洞之后，又走到右边希望填右边的洞，但找不到合适路径，会出现在右边震荡的情况。所以希望限制得分，在得到一定分数时降低 h 值，进而使其在得到一定分数时会去

找钥匙而不是通过填洞获得更低的 h，所以在返回值中加入了 getGameScore()*λ，其中λ为比例系数，为分数项在总 h 中占的比重，经过预测在 0-200 之间，并通过二分法尝试发现λ取为 50 时能够比较稳定通过第三关。之所以是比较稳定，是因为在 Astar 算法中，我会在当前决策实践不足时即下图中后半部分不满足时就跳出，这个会给算法带来一定的随机性，在极少数情况下，会寻路失败。

```
while(!openList.isEmpty() && elapsedTimer.remainingTimeMillis()>remainingLimit)
```

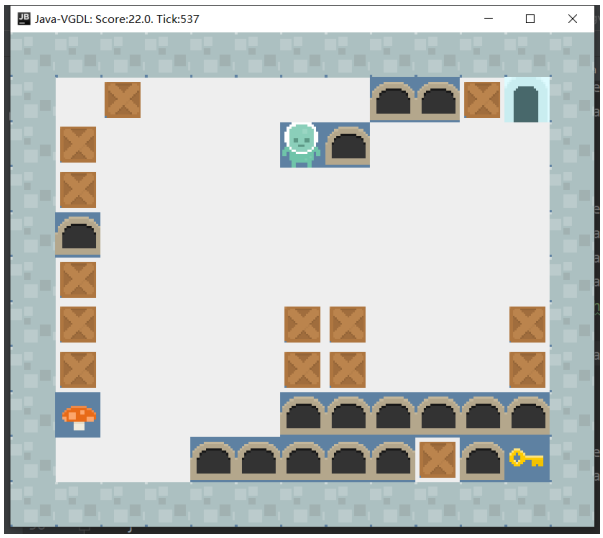
随后得到了第三关的表达式，如图：

```
return h+l1_norm(nowpos, keypos)-(int)(state.getGameScore())*50;
```

3.2.3 第四关及第五关

这里最终只完成了第五关，第四关只能得到较高的分数，但无法完成。

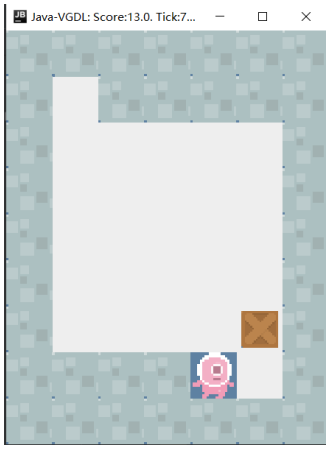
在第四关中，由于之前的启发式函数中计算了每一个箱子和每一个洞的距离，第四关中，精灵会疯狂填洞，如图所示：



分析认为是第三关中添加的`)-(int)(state.getGameScore())*50;`出现了问题，在第四关精灵可以得到特别高的分数，得分较高时，该项可能为负，则一定是最好结果，所以精灵会疯狂填洞，在尝试改变比例系数后发现没有明显变化，所以希望在这里给精灵一个惩罚，在分数较高时，让 h 变高，即让精灵避免只填箱子的情况，所以将减号改为加号，即给了精灵一个类似惩罚项的东西，避免得到较高分数，最终表达式如图：

```
return h+l1_norm(nowpos, keypos)+(int)(state.getGameScore())*50;
```

但发现效果不是很明显，反而是原本无法通过的第五关，在改变为该式并延长搜索时间后，可以通关第五关，并且是完美通关，十分神奇的拍脑袋启发式函数，如图：



3.3 代码结果

最终通过同一个启发式函数稳定完成了前三关，在对一个正负号改变并延长搜索时间后可以稳定完成第五关，第四关只可以得到较高的分数。但由于在 Astar 中加入了对时间的判断，防止超时，所有代码有了一些随机性，有时可以最优路径完成，有时会卡在一个地方两三次，有时甚至在走出一步之后直接停止，但整体看结果还是十分稳定，通关情况也十分好。

4 sampleMCTS

4.1 变量分析

epsilon 是一个很小的常数，使得在计算为访问过节点的值时，分母不会为 0，而是 epsilon，使得得到的 Value 极大，所以会对没有访问过的节点优先进行尝试。

egreedyEpsilon 是在 egreedy()中使用的常量，大于这个值时会选择当前最好的 Action，小于这个值时会随机选择 Action，采用的是一种贪婪的方法，但这个函数没有被调用，所以不做详细讨论。

State，parent，children 是自身的状态，以及树的相关变量。

Totalvalue 和 nvisits 是这个节点的当前的值和访问次数，用于计算后面的 uctValue。

m_rnd 是随机数发生器，在计算 uctValue 时用于添加噪声，以及扩展节点时会使用。

m_depth 为当前节点的深度，在 Agent 文件中有一个 ROLLOUT_DEPTH，是最大深度，防止树无限向下延伸。

4.2 代码分析

4.2.1 主要思路分析

mctsSearch()是执行的核心，第一步是 Selection，对应代码中的 treePolicy()函数，就是在树中找到一个最好的节点进行探索，优先选择未探索过的节点，如果都探索过，则选择值最大的节点。第二步是 Expansion，对应 treePolicy()中调用的 expand()函数，一般是随机执行一个未与之前子节点重复的操作，在代码中通过对每个节点生成一个随机数，选择随机数最大的节点进行执行来达到随机选择的目的。第三步是 Simulation，对应 Rollout()函数，就是在前面新 Expansion 出来的节点开始进行游戏，直到到达游戏结束状态，这样可以得到这个 expansion 出来的节点的得分是多少，Rollout()过程中采取的 Action 是随机的，从而实现快速达到游戏结束的目的。第四步是 Backpropagation，对应 backUp()函数，将新扩展的节点的得分进行回溯，即对所有祖先节点的得分和访问次数进行累加，更新 totalvalue 和 nvisits，方便后面计算 uctValue。执行完成后通过 mostVisitedAction()函数执行 act 进行游戏。

重复上述操作，通过不断的模拟游戏得到大部分节点的 uctValue，然后下次模拟的时候根据 uctValue 值对 exploration 和 exploitation 进行平衡，选择最好的节点，该算法适用于搜索空间巨大，不能通过简单的搜索枚举找到最优解的情况，MCTS 在这种情况下能更大概率找到更好的 Action。

4.2.2 函数分析

treePolicy()是选择 Action 的函数，循环的判断条件使得搜索的深度不会超过 ROLLOUT_DEPTH，不会

在书中无限向下搜索，之后如果当前节点没有完全扩展，则进行扩展，否则沿树向下搜索直到找到一个未完全扩展的节点。

Expand()函数是对节点的扩展，对为扩展出的每一个节点生成一个随机数，挑取最大的一个作为下一个扩展的节点，达到随机选择扩展节点的目的，如图。

```
for (int i = 0; i < children.length; i++) {
    double x = m_rnd.nextDouble();
    if (x > bestValue && children[i] == null) {
        bestAction = i;
        bestValue = x;
    }
}
```

Uct()函数是计算 uct 值，核心代码如下图，公式如下

$$childValue(\text{平均估值}) = normalize(\frac{childTotalValue}{childVisitTimes+\epsilon})$$

$$uctValue(\text{节点分数}) = childValue + \sqrt{\frac{2\ln parentVisitTimes + 1}{childVisitTimes + \epsilon}} + \xi$$

通过这个公式计算自身的值，childValue 为之前模拟中的平均估值，后面一项使得节点在被访问次数较少时会优先访问，最后的ξ为噪声。

```
double childValue =  hvVal / (child.nVisits + this.epsilon);
```

```
double uctValue = childValue +
        Agent.K * Math.sqrt(Math.log(this.nVisits + 1) / (child.nVisits + this.epsilon));

// small sampleRandom numbers: break ties in unexpanded nodes
uctValue = Utils.noise(uctValue, this.epsilon, this.m_rnd.nextDouble());     //break ties randomly
```

Rollout()函数在随机选择 Action 模拟游戏，在很短的时间内结束游戏，得到预测值，并通过 backUp()函数进行更新。如图为 Rollout 随机选择行动的函数。

```
while (!finishRollout(rollerState,thisDepth)) {

    int action = m_rnd.nextInt(Agent.NUM_ACTIONS);
    rollerState.advance(Agent.actions[action]);
    thisDepth++;
}
```