Assignment1：Bait 游戏作业报告

李佳鑫（211220006、211220006@smail.nju.edu.cn）

(南京大学 计算机科学与技术系，南京 210093)

摘 要: 关于 AI_Intro 的 Assignment1 的报告，利用 DepthFisrtSerarch 以及 A*算法对 GVGAI 框架下的 Bait 的通关路径进行搜索，对 SampleMCTS 进行学习。

1.任务一

针对第一个关卡，实现深度优先搜索 controllers.DepthFirst.java，在游戏一开始就使用深度优先搜索找到成功的路径通关，记录下路径，并在之后每一步按照路径执行动作。

(1)成员变量

```
public static Types.ACTIONS[] actions;//当前状态可获取的行动
public  ArrayList<StateObservation> stateHadExList=new ArrayList<StateObservation>();/**搜索中已经出现过的状态列表
                                                                                        防止形成环路**/
public  ArrayList<Types.ACTIONS> todoact=new ArrayList<Types.ACTIONS>();//最终路径
int step=0;//执行最终路径的序号
boolean found=false;//是否找到最终路径
```

(2)成员函数

主要函数为 DepthfirstSearch

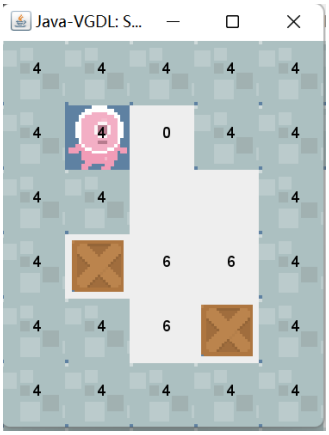
```
public void dfs(StateObservation so){
    for(int i=0;i<stateHadExList.size();i++){
        if(so.equalPosition(stateHadExList.get(i))){
            return;
        }
    }//对于状态so如果它已经在状态成员列表中存在，那么就直接结束当前状态搜索
    stateHadExList.add(so);//加入状态成员记录列表
    if(so.isGameOver()&&so.getGameWinner()==Types.WINNER.PLAYER_WINS){
        found=true;
        return;
    }//如果已经能结束游戏将最终路径存在设置为true
    else if(so.isGameOver()&&so.getGameWinner()==Types.WINNER.PLAYER_LOSES){
        return;
    }//如果这一步会导致游戏输掉就结束这一步的搜索
    ArrayList<Types.ACTIONS> act = so.getAvailableActions();//获取当前状态能进行的行动
    Types.ACTIONS[] toactions = new Types.ACTIONS[act.size()];
    for(int i = 0; i < toactions.length; ++i)//对能进行的行动进行遍历
    {
        if(found)break;//上一个行动能找到完成路径则直接停止遍历
        toactions[i] = act.get(i);
        todoact.add(toactions[i]);//将本行动加入最终行动列表
        StateObservation stCopy=so.copy();
        stCopy.advance(toactions[i]);//进行该行动，状态转变
        dfs(stCopy);//递归搜索
        if(!found){//这个行动不能完成即回溯
            todoact.remove(todoact.size()-1);//将本行动从最终行动列表中退出
        }
     }
    return;
}
```

因为在游戏开始前就要完成搜索，那么在构造函数中调用该函数即可，在最终的 act 函数中直接遍历取得 todocat[]中的行动就可以直接完成行动。

同时可以将每个 StateObservation 看成一个结点，每个 AvailableAction 看做边，这样就不用特地建树就可以有现成的路径以供搜索。

```
public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer)
{
    grid = so.getObservationGrid();
    block_size = so.getBlockSize();
    ArrayList<Types.ACTIONS> act = so.getAvailableActions();
    actions = new Types.ACTIONS[act.size()];
    for(int i = 0; i < actions.length; ++i)
    {
        actions[i] = act.get(i);
    }
    dfs(so);
}
```

（3） 任务一完成截图



2.任务二

在任务 1 的基础上，实现深度受限的深度优先搜索 LimitedDepthFirst

由于有任务 1 的基础，那么只要增加一个变量就可以限制搜索深度，在达到深度后直接同游戏输了一样直接 return 就行了

(1) 成员变量

```
public  ArrayList<StateObservation> stateHadExList=new ArrayList<StateObservation>();
public  ArrayList<Types.ACTIONS> todoact=new ArrayList<Types.ACTIONS>();
int step=0;
int steplimits=100;//深度限制
boolean found=false;
```

(2) 成员函数

同样是 dfs 唯一的不同只是在任务一的深度搜索上实现了深度限制

```java
public void dfs(StateObservation so,int depth){
    for(int i=0;i<stateHadExList.size();i++){
        if(so.equalPosition(stateHadExList.get(i))){
            return;
        }
    }//对于状态so如果它已经在状态成员列表中存在，那么就直接结束当前状态搜索
    if(depth>steplimits){
        return;
    }//如果这一步已经超过了深度限制则直接返回
    stateHadExList.add(so);
    if(so.isGameOver()&&so.getGameWinner()==Types.WINNER.PLAYER_WINS){
        found=true;
        return;
    }//如果已经能结束游戏将最终路径存在设置为true
    else if(so.isGameOver()&&so.getGameWinner()==Types.WINNER.PLAYER_LOSES){
        return;
    }//如果这一步会导致游戏输掉就结束这一步的搜索
    ArrayList<Types.ACTIONS> act = so.getAvailableActions();//获取当前状态能进行的行动
    Types.ACTIONS[] toactions = new Types.ACTIONS[act.size()];
    for(int i = 0; i < toactions.length; ++i)//对能进行的行动进行遍历
    {
        if(found)break;//上一个行动能找到完成路径则直接停止遍历
        toactions[i] = act.get(i);
        todoact.add(toactions[i]);//将本行动加入最终行动列表
        StateObservation stCopy=so.copy();
        stCopy.advance(toactions[i]);//进行该行动，状态转变
        dfs(stCopy,depth+1);//递归搜索，同时深度增加
        if(!found){//这个行动不能完成即回溯
            todoact.remove(todoact.size()-1);//将本行动从最终行动列表中退出
        }
    }
    return;
```

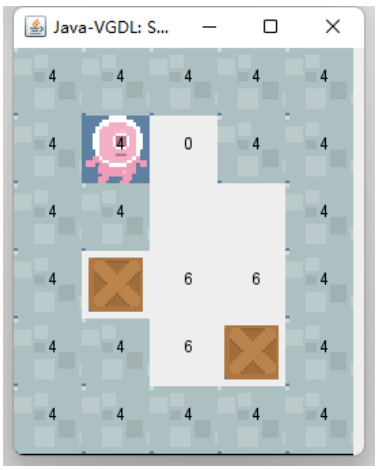
(3) 启发函数

本游戏有个很重要的特点是并不走直线路径，而是网格式的路径，那么距离的判断就不需要使用欧式距离，直接使用曼哈顿距离就可以了。那么启发函数的思路就很方便了，在拿到钥匙之前采用精灵与钥匙之间的距离，在拿到钥匙之后采用精灵与目标之间的距离。

因为要利用启发函数对路径选择进行排序，所以我选择了把启发函数放到了 StateObservation 便于重载 compareTo 函数

```java
public double hope(StateObservation stateObs){
    ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
    ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
    Vector2d avatarpos=stateObs.getAvatarPosition();//获取精灵坐标
    if(movingPositions!=null&&movingPositions.length!=0&&movingPositions[0].size()!=0){/**保证不会出现空指针和数组越界的情况
                                                                                        因为在获取到钥匙之后仍然调用钥匙位置会显示数组越界
                                                                                        如果在游戏结束时调用会显示空指针**/
    Vector2d keypos = movingPositions[0].get(index:0).position;//获取钥匙位置
    return Math.abs(avatarpos.x-keypos.x)+Math.abs(avatarpos.y-keypos.y);//返回精灵和目标的曼哈顿距离
    }
    if(!stateObs.isGameOver()){
    Vector2d goalpos = fixedPositions[1].get(index:0).position; //目标的坐标
    return Math.abs(goalpos.x-avatarpos.x)+Math.abs(goalpos.y-avatarpos.y);//返回和目标的曼哈顿距离
    }
    return 0;
}
public int compareTo(StateObservation o){//可以让StateObservation直接比较，便于排序
   return (int)hope(this)-(int)hope(o);
}
```

任务二完成截图



3.任务三

(1)成员函数

A*算法直接在 DFS 的基础上对路径算法进行排序 f(n)=g(n)+h(n)由于对于每一步的 g(n)都是相同的，那么 f(n)就退化成 h(n)单纯使用启发函数进行排序就可以了。启发函数在任务二就已经进行过描述，并对 StateObservation 的比较实现了重载以便于直接比较，那么在任务三直接使用即可

```java
public void dfs(StateObservation so,int depth){
    for(int i=0;i<stateHadExList.size();i++){
        if(so.equalPosition(stateHadExList.get(i))){
            return;
        }
    }//对于状态so如果它已经在状态成员列表中存在，那么就直接结束当前状态搜索
    stateHadExList.add(so);//加入状态成员记录列表
    if(depth>steplimits){
        return;
    }//如果这一步已经超过了深度限制则直接返回
    if(so.isGameOver()&&so.getGameWinner()==Types.WINNER.PLAYER_WINS){
        found=true;
        return;
    }//如果已经能结束游戏将最终路径存在设置为true
    else if(so.isGameOver()&&so.getGameWinner()==Types.WINNER.PLAYER_LOSES){
        return;
    }//如果这一步会导致游戏输掉就结束这一步的搜索
    ArrayList<Types.ACTIONS> act = so.getAvailableActions();//获取当前状态能进行的行动
    Types.ACTIONS[] toactions = new Types.ACTIONS[act.size()];
    HashMap<StateObservation,Integer>stob=new HashMap<StateObservation,Integer>();/**由于不能将action直接比较，
                                                                                    那么就需要action对应的StateObservation进行排序
                                                                                    利用HashMap实现对StateObservation的快速排序和查询
    for(int i = 0; i < toactions.length; ++i)//对能进行的行动进行遍历
    {
        if(found)break;
        toactions[i] = act.get(i);
        StateObservation stCopy=so.copy();
        stCopy.advance(toactions[i]);
        stob.put(stCopy,i);//将StateObservation和action序号存入map，将StateObservation作为key值以实现对作为序号的value值查询
    }
    ArrayList<StateObservation> sorted=new ArrayList<>(stob.keySet());
    Collections.sort(sorted);//对StateObservation用启发函数进行排序
    for(int i = 0; i < toactions.length; ++i)
    {
        if(found)break;
        todoact.add(toactions[stob.get(sorted.get(i))]);//利用启发函数排序后StateObservation的对应action加入action列表
        dfs(sorted.get(i),depth+1);//进行进一步的深度优先搜索
        if(!found){
            todoact.remove(todoact.size()-1);//如果没找到最终路径则需要进行回溯
        }
    }
    return;
}
```

剩余函数和成员变量同任务一和任务二

(2)任务三截图

4. 阅读 controllers.sampleMCTS.Agent.java 控制程序，并介绍其算法。

(1)初始化 Agent 和 mctsPlayer

```
public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer)
{
    //Get the actions in a static array.
    ArrayList<Types.ACTIONS> act = so.getAvailableActions();
    actions = new Types.ACTIONS[act.size()];
    for(int i = 0; i < actions.length; ++i)
    {
        actions[i] = act.get(i);
    }
    NUM_ACTIONS = actions.length;

    //Create the player.
    mctsPlayer = new SingleMCTSPlayer(new Random());
}
```

(2)执行调用 act，初始化 mctsPlayer，调用 run 函数返回执行序号，返回行动

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

    ArrayList<Observation> obs[] = stateObs.getFromAvatarSpritesPositions();
    ArrayList<Observation> grid[][] = stateObs.getObservationGrid();

    //Set the state observation object as the new root of the tree.
    mctsPlayer.init(stateObs);
    You, 20小时前 • first_assignment
    //Determine the action using MCTS...
    int action = mctsPlayer.run(elapsedTimer);

    //... and return it.
    return actions[action];
}
```

(3)init 函数

创建根结点，并将根节点状态设置为初始状态

```
public void init(StateObservation a_gameState)
{
    //Set the game observation to a newly root node.
    m_root = new SingleTreeNode(m_rnd);
    m_root.state = a_gameState;

}
```

(4)run 函数

传入 elapsedTimer，并执行 mctsSearch 在限定时间内进行搜索，并调用 mostVisitedAction 函数决定下一步的行动

```
public int run(ElapsedCpuTimer elapsedTimer)
{
    //Do the search within the available time.
    m_root.mctsSearch(elapsedTimer);

    //Determine the best action to take and return it.
    int action = m_root.mostVisitedAction();
    //int action = m_root.bestAction();
    return action;
}
```

(5)mctsSearch 函数

在时间允许范围内，循环调用三个函数分别为

a) treePolicy 挑选待探索的结点

b) rollOut 对该结点进行探索

c) backup 回溯

```
public void mctsSearch(ElapsedCpuTimer elapsedTimer) {      You, 21小时前 • first_assignment

    double avgTimeTaken = 0;
    double acumTimeTaken = 0;
    long remaining = elapsedTimer.remainingTimeMillis();
    int numIters = 0;

    int remainingLimit = 5;
    while(remaining > 2*avgTimeTaken && remaining > remainingLimit){
        ElapsedCpuTimer elapsedTimerIteration = new ElapsedCpuTimer();
        SingleTreeNode selected = treePolicy();
        double delta = selected.rollOut();
        backUp(selected, delta);

        numIters++;
        acumTimeTaken += (elapsedTimerIteration.elapsedMillis()) ;

        avgTimeTaken  = acumTimeTaken/numIters;
        remaining = elapsedTimer.remainingTimeMillis();
        //System.out.println(elapsedTimerIteration.elapsedMillis() + " --> " + acumTimeTaken + " (" + remaining + ")");
    }
    //System.out.println("-- " + numIters + " -- ( " + avgTimeTaken + ")");
}
```

(6)treePolicy 函数

循环执行如果当前结点的状态没有结束并且深度不超过深度限制，如果当前结点未完全探索即没有子节点，就继续探索调用 expand，如果已经找不到了则调用 uct 函数，对返回的结点继续探索

```
public SingleTreeNode treePolicy() {

    SingleTreeNode cur = this;

    while (!cur.state.isGameOver() && cur.m_depth < Agent.ROLLOUT_DEPTH)
    {
        if (cur.notFullyExpanded()) {
            return cur.expand();

        } else {
            SingleTreeNode next = cur.uct();
            //SingleTreeNode next = cur.egreedy();
            cur = next;
        }
    }

    return cur;
}
```

(7)expand 函数

产生一个随机数 0-1.0 并进行迭代，最终效果是随机挑选出该结点的一个行动，对该行动所产生的状态建立结点并返回。

```
public SingleTreeNode expand() {

    int bestAction = 0;
    double bestValue = -1;

    for (int i = 0; i < children.length; i++) {
        double x = m_rnd.nextDouble();      You, 21小时前 • first_assignment
        if (x > bestValue && children[i] == null) {
            bestAction = i;
            bestValue = x;
        }
    }

    StateObservation nextState = state.copy();
    nextState.advance(Agent.actions[bestAction]);

    SingleTreeNode tn = new SingleTreeNode(nextState, this, this.m_rnd);
    children[bestAction] = tn;
    return tn;

}
```

(8)uct 函数

对该结点的子节点进行挑选，挑选规则如下选用 uctValue 值最高的结点，从计算公式中看，决定因素主要是 totValue 和 nVisits，分值最高的同时访问次数最少的结点越容易选中。

```
public SingleTreeNode uct() {
    You, 21小时前 • first assignment
    SingleTreeNode selected = null;
    double bestValue = -Double.MAX_VALUE;
    for (SingleTreeNode child : this.children)
    {
        double hvVal = child.totValue;
        double childValue =  hvVal / (child.nVisits + this.epsilon);


        childValue = Utils.normalise(childValue, bounds[0], bounds[1]);

        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

        // small sampleRandom numbers: break ties in unexpanded nodes
        if (uctValue > bestValue) {
            selected = child;
            bestValue = uctValue;
        }
    }

    if (selected == null)
    {
        throw new RuntimeException("Warning! returning null: " + bestValue + " : " + this.children.length);
    }

    return selected;
```

(9)rollOut 函数

对当前结点随机选取一个行动，在游戏结束或者达到搜索深度限制之前不停搜索，最终调用 value 函数返回 detla，其中 value 函数的规则是如果最终找到游戏结束，如果游戏胜利则给这个结点给出极高的评分，否则就对该结点给出极低评分为负权，如果是达到搜索深度则给出该结点当前所得游戏分，（直觉上感觉就是 0 分，游戏结束给出的罚分太过离谱，大概是用于在同样能游戏胜利更期望获得更高评分的路径）

```
public double rollOut()
{
    StateObservation rollerState = state.copy();
    int thisDepth = this.m_depth;

    while (!finishRollout(rollerState,thisDepth)) {

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

    double delta = value(rollerState);

    if(delta < bounds[0])
        bounds[0] = delta;

    if(delta > bounds[1])
        bounds[1] = delta;

    return delta;
}
```

(10) mostVisitedAction 函数

在所有子节点中优先挑选出访问平均次数最多的节点并执行相对应的行动，如果所有的节点访问次数都一样多则调用 bestAction 函数挑选出下一步的行动

```
public int mostVisitedAction() {
    int selected = -1;
    double bestValue = -Double.MAX_VALUE;
    boolean allEqual = true;
    double first = -1;

    for (int i=0; i<children.length; i++) {

        if(children[i] != null)
        {
            if(first == -1)
                first = children[i].nVisits;
            else if(first != children[i].nVisits)
            {
                allEqual = false;
            }

            double childValue = children[i].nVisits;
            childValue = Utils.noise(childValue, this.epsilon, this.m_rnd.nextDouble());     //break ties randomly
            if (childValue > bestValue) {
                bestValue = childValue;
                selected = i;
            }
        }
    }

    if (selected == -1)
    {
        System.out.println(x:"Unexpected selection!");
        selected = 0;
    }else if(allEqual)
    {
        //If all are equal, we opt to choose for the one with the best Q.
        selected = bestAction();
    }
    return selected;
```

(11)bestAction 函数

类比 uct 函数 m 决定因素主要是 totValue 和 nVisits，分值最高的同时访问次数最少的结点越容易选中。

```
public int bestAction()
{
    int selected = -1;
    double bestValue = -Double.MAX_VALUE;

    for (int i=0; i<children.length; i++) {

        if(children[i] != null) {
            double childValue = children[i].totValue / (children[i].nVisits + this.epsilon);
            childValue = Utils.noise(childValue, this.epsilon, this.m_rnd.nextDouble());     //break ties randomly
            if (childValue > bestValue) {
                bestValue = childValue;
                selected = i;
            }
        }
    }

    if (selected == -1)
    {
        System.out.println(x:"Unexpected selection!");
        selected = 0;
    }

    return selected;
}
```

算法核心：通过不断的随机搜索判断，并对该节点能否最终完成游戏的进行加分和罚分，通过该节点访问次数和分数来选出最大可能完成游戏胜利的行动序列最终完成游戏。