作业 1 报告

201300069 邓嘉宏 1739413207@qq.com

摘要:

本文报告针对作业 1 的四项任务, 分别针对 Bait 游戏实现深度优先搜索算法、深度受限的深度优先搜索算法、A*算法,并详细介绍了以上算法的实现思想和具体代码, 最后介绍蒙特卡洛树搜索算法的思想.

关键词

深度优先搜索; 深度受限的深度优先搜索; A*; 蒙特卡洛树搜索

引言

搜索问题是人工智能领域的重要问题, 我们尝试利用多种搜索算法来自动操作 Bait 游戏.

任务 1

任务 1 要求实现**深度优先搜索算法**, 我们使用**树结构**搜索并使用**栈**存储空间存储树节点. 因此, 我们需要一个**树节点类**, 定义为 Node 类(具体实现在下方代码段中介绍).

深度优先搜索过程:

1. 首先, 生成根节点 root 并将其存入待遍历节点数组 fringe;
2. 接下来, 不断取出 fringe 数组的第一个元素节点, 并将其存入已遍历节点数组 closed,
3. 接着, 对该节点进行goal_test, 若已获胜, 则递归返回从根节点到该节点的路径, 搜索结束; 若还未获胜并且也没输, 则扩展该节点, 将扩展的子节点中未遍历过的节点依次加入 fringe 数组的头部(以此确保是深度优先搜索而不是宽度优先搜索).
4. 若 fringe 不为空,则回到 2. 继续执行.

以下为对代码的详细介绍:

```java
package controllers.depthfirst;

import core.game.Observation;
import core.game.StateObservation;
import ontology.Types;
import ontology.effects.unary.TurnAround;
import tools.ElapsedCpuTimer;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;

public class Agent extends controllers.sampleRandom.Agent{
    //searchedActionsList存储搜索到结果(成功的路径)
    protected  List<Types.ACTIONS> searchedActionsList = null;

    public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer){
        super(so,elapsedTimer);
    }
    //框架代码每次调用act函数获得一个动作
    public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

        ArrayList<Observation>[] npcPositions = stateObs.getNPCPositions();
        ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
        ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
        ArrayList<Observation>[] resourcesPositions = stateObs.getResourcesPositions();
        ArrayList<Observation>[] portalPositions = stateObs.getPortalsPositions();
        grid = stateObs.getObservationGrid();

        /*printDebug(npcPositions,"npc");
        printDebug(fixedPositions,"fix");
        printDebug(movingPositions,"mov");
        printDebug(resourcesPositions,"res");
        printDebug(portalPositions,"por");
        System.out.println();               */

        Types.ACTIONS action = null; //存储待返回的单个动作
        StateObservation stCopy = stateObs.copy(); // 拷贝当前状态用于搜索

        if(searchedActionsList==null){
            //若还未搜索过则进行搜索
            searchedActionsList=depthFirstSearch(stCopy,elapsedTimer);
        }

        if(searchedActionsList.size()>0)
            //已搜索过，返回搜索结果的单个动作
            action =searchedActionsList.remove(0);

        System.out.print(action);//打印动作，用于观察和debug
        System.out.print(" ");
```

```
    return action;
}
//深度优先搜索算法实现函数
private List<Types.ACTIONS> depthFirstSearch(StateObservation stCopy,ElapsedCpuTimer elapsed
    LinkedList<Types.ACTIONS> searchedActions = new LinkedList<Types.ACTIONS>();//存储搜索结果

    Node root = new Node(stCopy);//创建根节点

    LinkedList<Node> fringe = new LinkedList<Node>(); //待遍历节点数组
    LinkedList<Node> closed = new LinkedList<Node>(); //遍历过的节点的数组
    fringe.add(root);// 将根节点加入待遍历数组

    //以下几个变量用于计时
    double avgTimeTaken = 0;
    double acumTimeTaken = 0;
    long remaining = elapsedTimer.remainingTimeMillis();
    int numIters = 0;
    int remainingLimit = 5;

    //while循环不断取出fringe的节点直到成功
    while(fringe.size()>0&&remaining > 2*avgTimeTaken && remaining > remainingLimit){
        //每次取出一个节点前记录时间
        ElapsedCpuTimer elapsedTimerIteration = new ElapsedCpuTimer();
        Node node = fringe.remove();//取出第一个节点
        closed.add(node);//将该节点加入已遍历节点数组

        if(node.state.getGameWinner()==Types.WINNER.PLAYER_WINS){
            //若该节点时的状态已获胜,则获取路径并结束搜索
            searchedActions=node.getpath();
            return searchedActions;
        }

        if (!node.state.isGameOver()) {
            //若该节点游戏还未结束,则扩展该节点
            List<Node> nodes = node.expand();

            for (Node n : nodes) {
                //遍历扩展的节点，检查是否遍历过了
                boolean is_visited = false;
                for (int i = 0; i < closed.size(); i++) {
                    //遍历closed数组的节点
                    if (n.state.equalPosition(closed.get(i).state)) {
                        //使用StateObservation类的equalPosition方法判断是否遍历过
                        is_visited = true;
                        break;
                    }
                }
                if (is_visited == false) {
                    //若该节点还未遍历过,则将其加入 fringe
                    fringe.addFirst(n);
```

```
                }
            }
        }

        //以下记录时间,用于判断是否超时
        numIters++;
        acumTimeTaken += (elapsedTimerIteration.elapsedMillis()) ;
        //System.out.println(elapsedTimerIteration.elapsedMillis() + " --> " + acumTimeTaker
        avgTimeTaken  = acumTimeTaken/numIters;
        remaining = elapsedTimer.remainingTimeMillis();
    }
    //返回搜索结果
    return searchedActions;
}

//Node的实现
static class Node {

    public StateObservation state = null; //该节点的局面状态
    public List<Node> child = null; //子节点
    public Node parent = null; //父节点
    public int cost = 0; //从根节点到该节点的cost
    public Types.ACTIONS pre_action=null; //记录从父节点到该节点走的动作,用于获取成功路径走过的动


    public Node(StateObservation s) {   //初始化
        state = s;
    }

    public List<Node> expand() {    //扩展子节点
        ArrayList<Node> subnodes= new ArrayList<Node>(4); //用于存储子节点
        ArrayList<Types.ACTIONS> a=state.getAvailableActions(); //获取当前节点能走的动作

        //依次走各个动作，并将子节点加入 subnodes
        for (int i=0;i<a.size();i++) {
            StateObservation s = state.copy(); //复制当前节点的状态，用于生成子节点状态
            s.advance(a.get(i));
            Node n = new Node(s); //子节点
            n.parent = this; //当前节点为子节点的父节点
            n.pre_action = a.get(i); // 记录当前节点到子节点的动作
            subnodes.add(n);
        }
        return subnodes; //返回子节点组成的列表
    }

    // 获取根节点到该节点的路径
    public LinkedList<Types.ACTIONS> getpath(){
        LinkedList<Types.ACTIONS> ans=new LinkedList<Types.ACTIONS>(); //存储路径
        Node now=this; // 从当前节点开始获取
        while (now.parent.parent!=null){
            //若还能往前获取则往前获取
```

```
                ans.addFirst(now.pre_action);
                now=now.parent;
            }
            ans.addFirst(now.pre_action);
            return ans; //返回路径列表
        }


    }
}
```

任务 2

任务 2 要求在任务 1 的基础上，实现**深度受限的深度优先搜索**. 任务 1 的深度优先搜索没有限制, 直至搜索到成功的结果才返回; 深度受限的深度优先搜索与深度优先搜索整体过程无差异, 只是添加了一个**限制条件**(在任务 2 中限制条件为有限的时间), 在搜索的过程中, 当该条件无法满足时则结束搜索, 并不一定搜索到成功的结果.

如此一来, 既然没有成功的路径, 那搜索结果该返回什么呢? 应该返回搜索到的最好结果.

何为最好结果? 这与设计的**启发式函数**有关, 用于评估某个状态的好坏, 在 任务2 中, 我们用目标的位置和钥匙的位置构造启发式函数, 精灵、钥匙、目标之间的**曼哈顿距离**之和为评估结果, 该结果越小则当前状态越好.

与任务 1 的代码相比, 任务 2 的代码主要修改了以下几点:

- **添加**: 当因限制条件不满足而结束搜索时, 搜索算法的实现函数在返回结果前, 根据启发函数的**评估结果**从遍历过的节点中挑选最好的节点.
- **添加**: Node 类添加成员函数(**启发式函数**), 用于计算该节点的评估结果.
- **修改**: Node 类的成员 **cost** 用于记录评估函数的评估结果.

以下为对代码的详细介绍:

```java
package controllers.limitdepthfirst;

import core.game.Observation;
import core.game.StateObservation;
import ontology.Types;
import ontology.effects.unary.TurnAround;
import tools.ElapsedCpuTimer;
import tools.Vector2d;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;

public class Agent extends controllers.sampleRandom.Agent{
    //searchedActionsList存储搜索到结果
    protected  List<Types.ACTIONS> searchedActionsList = null;

    public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer){
        super(so,elapsedTimer);
    }
    //框架代码每次调用act函数获得一个动作
    public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

        ArrayList<Observation>[] npcPositions = stateObs.getNPCPositions();
        ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
        ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
        ArrayList<Observation>[] resourcesPositions = stateObs.getResourcesPositions();
        ArrayList<Observation>[] portalPositions = stateObs.getPortalsPositions();
        grid = stateObs.getObservationGrid();

        /*printDebug(npcPositions,"npc");
        printDebug(fixedPositions,"fix");
        printDebug(movingPositions,"mov");
        printDebug(resourcesPositions,"res");
        printDebug(portalPositions,"por");
        System.out.println();               */

        Types.ACTIONS action = null;//存储待返回的单个动作
        StateObservation stCopy = stateObs.copy();// 拷贝当前状态用于搜索

        if(searchedActionsList==null){
            //若还未搜索过则进行搜索
            searchedActionsList=limiteddepthFirstSearch(stCopy,elapsedTimer);
        }

        if(searchedActionsList.size()>0)
            //已搜索过，返回搜索结果的单个动作
            action =searchedActionsList.remove(0);
        System.out.print(action);//打印动作，用于观察和debug
        System.out.print(" ");
```

```
    return action;
}

//深度受限的深度优先搜索算法实现函数
private List<Types.ACTIONS> limiteddepthFirstSearch(StateObservation stCopy,ElapsedCpuTimer
    LinkedList<Types.ACTIONS> searchedActions = new LinkedList<Types.ACTIONS>();//存储搜索结果

    Node root = new Node(stCopy);//创建根节点

    LinkedList<Node> fringe = new LinkedList<Node>();//待遍历节点数组
    LinkedList<Node> closed = new LinkedList<Node>();//遍历过的节点的数组
    root.cost= root.informed();//根节点的评估结果
    fringe.add(root);// 将根节点加入待遍历数组

    //以下几个变量用于计时
    double avgTimeTaken = 0;
    double acumTimeTaken = 0;
    long remaining = elapsedTimer.remainingTimeMillis();
    int numIters = 0;
    int remainingLimit = 5;

    //while循环不断取出fringe的节点直到成功
    while(fringe.size()>0&&remaining > 2*avgTimeTaken && remaining > remainingLimit){
        //每次取出一个节点前记录时间
        ElapsedCpuTimer elapsedTimerIteration = new ElapsedCpuTimer();
        Node node = fringe.remove();//取出第一个节点
        closed.add(node);//将该节点加入已遍历节点数组

        if(node.state.getGameWinner()==Types.WINNER.PLAYER_WINS){
            //若该节点时的状态已获胜,则获取路径并结束搜索
            searchedActions=node.getpath();
            return searchedActions;
        }

        if (!node.state.isGameOver()) {
            //若该节点游戏还未结束,则扩展该节点
            List<Node> nodes = node.expand();
            for (Node n : nodes) {
                //遍历扩展的节点，检查是否遍历过了
                boolean is_visited = false;
                for (int i = 0; i < closed.size(); i++) {
                    //遍历closed数组的节点
                    if (n.state.equalPosition(closed.get(i).state)) {
                        //使用StateObservation类的equalPosition方法判断是否遍历过
                        is_visited = true;
                        break;
                    }
                }
                if (is_visited == false) {
                    //若该节点还未遍历过,则将其加入 fringe
                    fringe.addFirst(n);
```

```
                }
            }
        }
        //以下记录时间,用于判断是否超时
        numIters++;
        acumTimeTaken += (elapsedTimerIteration.elapsedMillis()) ;
        //System.out.println(elapsedTimerIteration.elapsedMillis() + " --> " + acumTimeTaker
        avgTimeTaken  = acumTimeTaken/numIters;
        remaining = elapsedTimer.remainingTimeMillis();
    }

    //因不满足限制条件而结束搜索时，寻找最优节点(即cost值最小的节点)
    Double min_num=closed.get(0).cost;
    int min_id=0;
    for(int i=1;i<closed.size();++i){
        if (closed.get(i).cost<min_num && closed.get(i).state.isGameOver()==false){
            min_num= closed.get(i).cost;
            min_id=i;
        }
    }
    //记录根节点到最优节点的路径
    searchedActions=closed.get(min_id).getpath();

    return searchedActions; //返回搜索结果
}



//Node的实现
static class Node {
    public StateObservation state = null;//该节点的局面状态
    public List<Node> child = null;//子节点
    public Node parent = null;//父节点
    public double cost = 0; //评估函数对该节点的评估结果
    public Types.ACTIONS pre_action=null;//记录从父节点到该节点走的动作,用于获取成功路径走过的动

    public Node(StateObservation s) {//初始化
        state = s;
    }

    //启发式函数
    public double informed(){
        ArrayList<Observation>[] fixedPositions=state.getImmovablePositions();
        ArrayList<Observation>[] movingPositions=state.getMovablePositions();
        Vector2d avapos=state.getAvatarPosition(); //记录精灵位置
        Vector2d goalpos=new Vector2d(); //记录目标位置
        Vector2d keypos=new Vector2d(); //记录钥匙位置
        boolean f_key=false; //标记是否找到钥匙

        //在fixedPositions中寻找目标位置(一定能找到)
```

```
    for (int i=0;i<fixedPositions.length;++i){
        if(fixedPositions[i].size()!=0&&fixedPositions[i].get(0).itype
        ==7){
            goalpos=fixedPositions[i].get(0).position;
            break;
        }
    }

    if (state.getAvatarType()==1){ //若精灵还没拿到钥匙，则能获取到钥匙位置
        for (int i=0;i<movingPositions.length;++i){
            if(movingPositions[i].size()!=0&&movingPositions[i].get(0).itype==6){
                keypos= movingPositions[i].get(0).position;
                f_key=true;
                break;
            }
        }
        if(f_key) {
            //精灵没拿到钥匙，返回精灵和钥匙距离与钥匙和目标距离之和
            return avapos.d(keypos) + keypos.d(goalpos);
        }
        else{
            System.out.print("not find key");
        }
    }

    //精灵已经拿到钥匙，返回精灵和目标的距离
    return avapos.d(goalpos);

}

public List<Node> expand() { //扩展子节点
    ArrayList<Node> subnodes= new ArrayList<Node>(4);//用于存储子节点
    ArrayList<Types.ACTIONS> a=state.getAvailableActions();//获取当前节点能走的动作

    //依次走各个动作，并将子节点加入 subnodes
    for (int i=0;i<a.size();i++) {
        StateObservation s = state.copy();//复制当前节点的状态，用于生成子节点状态
        s.advance(a.get(i));
        Node n = new Node(s); //子节点
        n.cost=n.informed(); //对子节点进行评估
        n.parent = this;  //当前节点为子节点的父节点
        n.pre_action = a.get(i); // 记录当前节点到子节点的动作
        subnodes.add(n);
    }
    return subnodes;//返回子节点组成的列表
}

// 获取根节点到该节点的路径
public LinkedList<Types.ACTIONS> getpath(){
    LinkedList<Types.ACTIONS> ans=new LinkedList<Types.ACTIONS>();//存储路径
    Node now=this; // 从当前节点开始获取
```

```
            while (now.parent.parent!=null){
                ans.addFirst(now.pre_action);
                //若还能往前获取则往前获取
                now=now.parent;
            }
            ans.addFirst(now.pre_action);
            return ans; //返回路径列表
        }


    }
}
```

任务 3

任务 3 要求在任务 2 的基础上, 将深度优先搜索换成**A*算法**, 两者的区别在于: A*算法每次从 fringe 拿出节点时, 总是拿出认为的**最优节点**.

何为最优节点? 经过当前节点到达目标的**总开销最小**的节点.

$f(n) = g(n) + h(n), f(n) =$ 总开销, $g(n) =$ 从根节点到当前节点的开销, $h(n) =$ 估计从当前节点到目标的开销.

因此, 任务 3 的关键是计算出f(n). g(n)是已走过路径的开销, 容易得到. h(n)是需要估计的开销,与设计的启发式函数有关. 因此, 任务 3 的关键是设计**启发式函数**估计当前节点到目标所需要的开销.

尝试 : 继续使用任务 2 中的启发式函数计算 h(n), 但并不能在较短时间内通过前三关.

失败原因: A*算法的关键是启发式函数, 也就是说, 人为地指导精灵下一步该怎么走. 任务 2 的启发式函数简单地以精灵、钥匙、目标之间的距离作为评估标准, 忽略了游戏的具体**规则**, 如: 精灵、箱子、洞、钥匙等之间的关系, 因此也就不能提供很好的"指导", 搜索较慢.

改良: 我们需要利用规则重新设计启发式函数. 我们不考虑游戏得分, 即不关心蘑菇, 我们"指导"精灵:**用最近的箱子填上最近的洞, 努力拿到钥匙并走到终点**. 因此, 在启发式函数中, 我们需要获得精灵位置, 各箱子位置, 各洞位置, 钥匙位置, 目标位置. 计算出精灵到最近箱子距离、精灵到钥匙或者目标的距离、每个箱子到最近洞的距离, 将它们配以一定系数相加, 结果作为启发式函数的评估结果.

与任务 2 的代码相比, 任务 3 的代码主要修改了以下几点:

- **修改:** 用**优先队列**实现 fringe ,每次从 fringe 取出节点时总是取出总开销最小的节点.
- **修改:** 启发式函数.
- **添加:** 节点在 expand 时, 计算每个子节点的总开销.

以下为对代码的详细介绍:

```java
package controllers.Astar;

import core.game.Observation;
import core.game.StateObservation;
import ontology.Types;
import ontology.effects.unary.TurnAround;
import tools.ElapsedCpuTimer;
import tools.Vector2d;

import java.util.*;

public class Agent extends controllers.sampleRandom.Agent{
    //searchedActionsList存储搜索到结果
    protected  List<Types.ACTIONS> searchedActionsList = null;

    public Agent(StateObservation so, ElapsedCpuTimer elapsedTimer){
        super(so,elapsedTimer);
    }
    //框架代码每次调用act函数获得一个动作
    public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

        ArrayList<Observation>[] npcPositions = stateObs.getNPCPositions();
        ArrayList<Observation>[] fixedPositions = stateObs.getImmovablePositions();
        ArrayList<Observation>[] movingPositions = stateObs.getMovablePositions();
        ArrayList<Observation>[] resourcesPositions = stateObs.getResourcesPositions();
        ArrayList<Observation>[] portalPositions = stateObs.getPortalsPositions();
        grid = stateObs.getObservationGrid();

        /*printDebug(npcPositions,"npc");
        printDebug(fixedPositions,"fix");
        printDebug(movingPositions,"mov");
        printDebug(resourcesPositions,"res");
        printDebug(portalPositions,"por");
        System.out.println();               */

        Types.ACTIONS action = null;//存储待返回的单个动作
        StateObservation stCopy = stateObs.copy();// 拷贝当前状态用于搜索

        if(searchedActionsList==null){
            //若还未搜索过则进行搜索
            searchedActionsList=Astar(stCopy,elapsedTimer);
        }

        if(searchedActionsList.size()>0)
            //已搜索过，返回搜索结果的单个动作
            action =searchedActionsList.remove(0);
        System.out.print(action);//打印动作，用于观察和debug
        System.out.print(" ");
        return action;
    }
```

```
//A*算法实现函数
private List<Types.ACTIONS> Astar(StateObservation stCopy,ElapsedCpuTimer elapsedTimer) {
    LinkedList<Types.ACTIONS> searchedActions = new LinkedList<Types.ACTIONS>();//存储搜索结果

    Node root = new Node(stCopy);//创建根节点

    //用优先队列存储待遍历节点，tot_cost最小的节点排在第一位
    PriorityQueue<Node> fringe = new PriorityQueue<Node>((x, y) -> Double.compare(x.tot_cost
    LinkedList<Node> closed = new LinkedList<Node>();//遍历过的节点的数组
    root.tot_cost=root.informed();//根节点的评估结果
    fringe.add(root);// 将根节点加入待遍历队列

    //以下几个变量用于计时
    double avgTimeTaken = 0;
    double acumTimeTaken = 0;
    long remaining = elapsedTimer.remainingTimeMillis();
    int numIters = 0;
    int remainingLimit = 5;

    //while循环不断取出fringe的节点直到成功
    while(fringe.size()>0&&remaining > 2*avgTimeTaken && remaining > remainingLimit){
        //每次取出一个节点前记录时间
        ElapsedCpuTimer elapsedTimerIteration = new ElapsedCpuTimer();
        Node node = fringe.poll();//取出第一个节点
        closed.add(node);//将该节点加入已遍历节点数组
        if(node.state.getGameWinner()==Types.WINNER.PLAYER_WINS){
            //若该节点时的状态已获胜,则获取路径并结束搜索
            searchedActions=node.getpath();
            return searchedActions;
        }
        if (!node.state.isGameOver()) {
            //若该节点游戏还未结束,则扩展该节点
            List<Node> nodes = node.expand();

            for (Node n : nodes) {
                //遍历扩展的节点，检查是否遍历过了
                boolean is_visited = false;
                for (int i = 0; i < closed.size(); i++) {
                    //遍历closed数组的节点
                    if (n.state.equalPosition(closed.get(i).state)) {
                        //使用StateObservation类的equalPosition方法判断是否遍历过
                        is_visited = true;
                        break;
                    }
                }
                if (is_visited == false) {
                    //若该节点还未遍历过并且游戏未输,则将其加入 fringe
                    if (!(n.state.getGameWinner()==Types.WINNER.PLAYER_LOSES)){
                        fringe.add(n);
                    }
```

```
                }
            }
        }
        //以下记录时间,用于判断是否超时
        numIters++;
        acumTimeTaken += (elapsedTimerIteration.elapsedMillis()) ;
        //System.out.println(elapsedTimerIteration.elapsedMillis() + " --> " + acumTimeTaker
        avgTimeTaken  = acumTimeTaken/numIters;
        remaining = elapsedTimer.remainingTimeMillis();
    }

    //因不满足限制条件而结束搜索时，寻找最优节点(即tot_cost值最小的节点)
    Double min_num=closed.get(0).tot_cost;
    int min_id=0;
    for(int i=1;i<closed.size();++i){
        if (closed.get(i).tot_cost<min_num && closed.get(i).state.isGameOver()==false){
            min_num= closed.get(i).tot_cost;
            min_id=i;
        }
    }
    //记录根节点到最优节点的路径
    searchedActions=closed.get(min_id).getpath();

    return searchedActions; //返回搜索结果
}


//Node的实现
static class Node {
    public StateObservation state = null;//该节点的局面状态
    public List<Node> child = null;//子节点
    public Node parent = null;//父节点
    public double pre_cost = 0; //从根节点到该节点的已有开销
    public double fut_cost=0; //启发式函数的评估结果
    public double tot_cost=0; //总开销
    public Types.ACTIONS pre_action=null;//记录从父节点到该节点走的动作,用于获取成功路径走过的动

    public Node(StateObservation s) {//初始化
        state = s;
    }

    //启发式函数
    public double informed(){
        ArrayList<Observation>[] fixedPositions=state.getImmovablePositions();
        ArrayList<Observation>[] movingPositions = state.getMovablePositions();

        ArrayList<Observation> boxes=new ArrayList<Observation>();//存储所有箱子
        ArrayList<Observation> holes=new ArrayList<Observation>();//存储所有洞

        //寻找并记录洞
```

```
for (int i=0;i<fixedPositions.length;i++){
    if(fixedPositions[i].size()!=0&&fixedPositions[i].get(0).itype==2){
        for(int m=0;m<fixedPositions[i].size();m++){
            Observation mm=fixedPositions[i].get(m);
            holes.add(mm);
        }
        break;
    }
}
Vector2d goalpos=new Vector2d();
//寻找目标的位置
for (int i=0;i<fixedPositions.length;i++){
    if(fixedPositions[i].size()!=0&&fixedPositions[i].get(0).itype ==7){
        goalpos=fixedPositions[i].get(0).position;
        break;
    }
}

Vector2d avapos=state.getAvatarPosition();//精灵位置
//寻找并存储钥匙和所有箱子位置
Vector2d keypos=new Vector2d();
if (state.getAvatarType()==1){

    for (int i=0;i<movingPositions.length;i++){
        if(movingPositions[i].size()!=0&&movingPositions[i].get(0).itype==6){
            keypos= movingPositions[i].get(0).position;
        }
        if(movingPositions[i].size()!=0&&movingPositions[i].get(0).itype==8){
            for(int m=0;m<movingPositions[i].size();m++){
                Observation mm=movingPositions[i].get(m);
                boxes.add(mm);
            }
        }
    }

}
else if(movingPositions!=null){
    for(int i=0;i<movingPositions.length;i++){
        if(movingPositions[i].size()!=0&&movingPositions[i].get(0).itype==8){
            for(int m=0;m<movingPositions[i].size();m++){
                Observation mm=movingPositions[i].get(m);
                boxes.add(mm);
            }
            break;
        }
    }
}
//计算精灵到钥匙或目标、精灵到最近的箱子、每个箱子到最近洞的距离
double ava_key=0,ava_box=0,boxes_holes=0;

if(state.getAvatarType()==1){
```

```
            ava_key=avapos.d(keypos);
        }
        else{
            ava_box=0;
            boxes_holes=0;
            ava_key=avapos.d(goalpos);
            double cost=ava_key+4*ava_box+32*boxes_holes;
            return cost;
        }

        if(boxes.size()!=0){
            double min_d=avapos.d(boxes.get(0).position);
            for (int i=1;i<boxes.size();i++){
                double n=avapos.d(boxes.get(i).position);
                if(n<min_d){
                    min_d=n;
                }
            }
            ava_box=min_d;
        }
        if(boxes.size()!=0&&holes.size()!=0){
            for (int i=0;i<boxes.size();i++){
                double  min_d=boxes.get(i).position.d(holes.get(0).position);
                for (int j =1;j<holes.size();j++){
                    double n=boxes.get(i).position.d(holes.get(j).position);
                    if(n<min_d){
                        min_d=n;
                    }
                }
                boxes_holes+=min_d;
            }
        }
        //计算评估结果
        double cost=ava_key+4*ava_box+32*boxes_holes;
        return cost;

    }

    public List<Node> expand() {//扩展子节点
        ArrayList<Node> subnodes= new ArrayList<Node>(4);//用于存储子节点
        ArrayList<Types.ACTIONS> a=state.getAvailableActions();//获取当前节点能走的动作

        //依次走各个动作，并将子节点加入 subnodes
        for (int i=0;i<a.size();i++) {
            StateObservation s = state.copy();//复制当前节点的状态，用于生成子节点状态
            s.advance(a.get(i));
            Node n = new Node(s);//子节点
            n.parent = this;//当前节点为子节点的父节点
            n.pre_cost=n.parent.pre_cost+50; //子节点的已有开销为父节点的已有开销+50(即走一步)
            n.pre_action = a.get(i);// 记录当前节点到子节点的动作
            //计算评估的开销
```

```
                if(!n.state.isGameOver()){
                    n.fut_cost=n.informed();
                }
                else{
                    n.fut_cost=1;
                }
                n.tot_cost=n.pre_cost+n.fut_cost;//总开销=已有开销+评估的开销
                subnodes.add(n);
            }
            //返回子节点组成的列表
            return subnodes;
        }

        // 获取根节点到该节点的路径
        public LinkedList<Types.ACTIONS> getpath(){
            LinkedList<Types.ACTIONS> ans=new LinkedList<Types.ACTIONS>();//存储路径
            Node now=this;// 从当前节点开始获取
            while (now.parent.parent!=null){
                ans.addFirst(now.pre_action);
                //若还能往前获取则往前获取
                now=now.parent;
            }
            ans.addFirst(now.pre_action);
            return ans; //返回路径列表
        }


    }
}
```

任务 4

蒙特卡洛树搜索算法:
首先以当前状态构造根节点, 接着在限制条件未达到前, 进行搜索, 不断重复进行以下步骤:

- 令当前节点为根节点, 若当前节点不是叶节点, 则令当前节点= **uct** 值最大的子节点, 直至当前节点为叶节点;
- 此时, 若当前节点已访问过, 则将当前节点扩展并添加到树中, 当前节点=第一个新节点, 再进行**rollOut**, 若当前节点未访问过, 则直接进行**rollOut**;
- 进行**backUp**: 若rollOut的结果是胜利, 则路径上的所有节点遍历次数+1, 值+HUGE_POSITIVE, 若失败, 则路径上的所有节点遍历次数+1, 值-HUGE_NEGATIVE.

停止搜索后, 选择根节点的 **uct** 值最大的子节点作为下一步精灵走的动作.

思想: 根据一定的评估标准(uct值), 进行大量的"试走", 以找到最好的下一步走法.

结束语

本文介绍了对多种算法的理解和实现, 在理解和实现这些算法的过程中, 我遇到了不少麻烦, 如: 对**框架代码**的理解和**接口的使用**有一些困难, 感谢为我耐心解答的朋友们.