

The University of Texas at Dallas

Game: PAC-MAN

Simulation of MiniMax and - ! runin" AI"orithm

P#\$%&CT G#\$UP ' (

)y San*et Chandor*ar



Fall 2012

Simulation of MiniMax and α - β pruning algorithm for game:



Project Group 12

By

Sanket Chandorkar

sxc127330@utdallas.edu

Table of Contents

1. Introduction to Pac-Man	4
1.1 What Kind of game is Pac-Man?	4
1.2 Problem Statement:	4
2. Goal Formalization:	5
3. Architecture design of Pac-Man game:	6
4. Utility Function:	6
5. MiniMax Algorithm:	7
5.1 Why MiniMax?	7
5.2 Algorithm.....	7
5.3 MiniMax Pseudo Code:	7
5.4 MiniMax Simulation:	8
6. Alpha-Beta Pruning Algorithm:	9
6.1 Alpha-Beta Pseudo Code:.....	9
6.2 Alpha Beta Simulation:	10
7. Result Comparison:	11
8. Implementation Details:.....	11
8.1 Some Important classes and their responsibilities.	11
8.2 Following are few of the responsibilities/Roles:	12
9. Future Scope:	12
10. Conclusion:	13
Appendix A: Game Snapshots	14
Appendix B: Sample Game code Snippets	16
References:.....	21

1. Introduction to Pac-Man

The game Pac-Man was developed by Namco in the 1980's. The game is immensely popular from its original release to the present day. Pac-Man is considered one of the classics of the medium, virtually synonymous with video games, and an icon of 1980s popular culture.

1.1 What kind of game is Pac-Man?

Intelligent Games can be classified broadly on the basis of Information content and state transition.

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	blind tictactoe	nuclear war

NOTE: Pac-Man is a **deterministic, multi-agent and a perfect information game**.

1.2 Problem Statement:

- This project involves the classic game of Pac-Man which has the agent as Pac-Man who tries to win the game by eating all dots and adversary as the ghosts who tries to stop Pac-Man.
- This project will simulate MiniMax and α - β pruning algorithm on the game of Pac-Man.
Note: This project will not implement the entire PAC-Man game but only implement part of it necessary for simulating the given algorithms.

(- Goal Formalization:

The game is a multi-agent adversarial game where the PAC man tries to win by eating all dots and the adversary "The ghosts" trying to stop PAC Man.

Let us formalize the problem by defining different terms.

State: A state is described by the position of the PAC Man and The Ghosts, and the dots in the game maze.

Game Maze: This describes the constrained environment / state space where the agent - PAC Man and the adversary The ghost moves.

Goal: PAC Man i. e the agent wins if he eats all the dots within the game maze. He loses is Ghost eats PAC MAN before the same.

Constraint: PAC man and Ghost have to move inside the Game maze. Their motion is bound within its walls.

Operators:

Moving: One step motion of ghost/PAC Man in UP, DOWN, LEFT or RIGHT direction.

There can be the following motions:

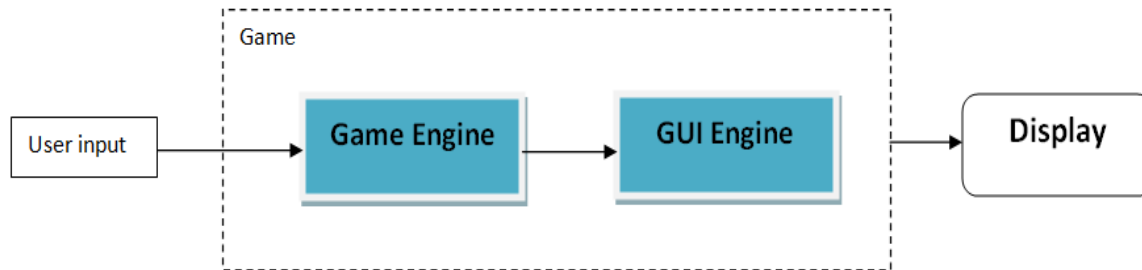
- **Simple_Move:** A simple move is any direction.
- **Eating_dot_move:** PAC Man eats the dot in adjacent space.
- **Killing_move:** Ghost kills PAC Man in adjacent space.

Initial state: This state is predefined in the game as part of maze design / level designing part of game development process.

Path cost for one action: One for each operation.

Total Path Cost: Sum of all path costs for the PAC Man to win the game.

+ - Ar/hite/ture desi"n of Pa/-Man "ame:



, - Utility Sun/tion:

- Utility Function is the one which scores the terminal nodes in the game-tree.
- It is the most important component of the game design.
- The Game Intelligence is decided by the strength of the utility Function.

The Utility Function used in our implementation is:

**Value = [Step taken by Pac-Man X (-1)] +
[Dots Eaten X (100)] +
[Win X 5000 / Lose X -5000] -
[Min actual distance between Pac-Man & Ghost]**

4- MiniMax Algorithm:

4-1 Why MiniMax

- Perfect play for deterministic environment with perfect information.
- Try to find next best move in a game with 2 player .
- The object of a search is to find a path from the starting position to a goal position
- It calculates all possible game states by examining all opposing moves .
- Determine the next move against best play[opponent] .

4-2 Algorithm

- Generate the game tree completely
- Determine utility of each terminal state
- Propagate the utility values upward in the tree by applying MIN and MAX operators on the nodes in the current level
- At the root node use minimax decision to select the move with the max (of the min) utility value

4-3 MiniMax Pseudo Code:

```

function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

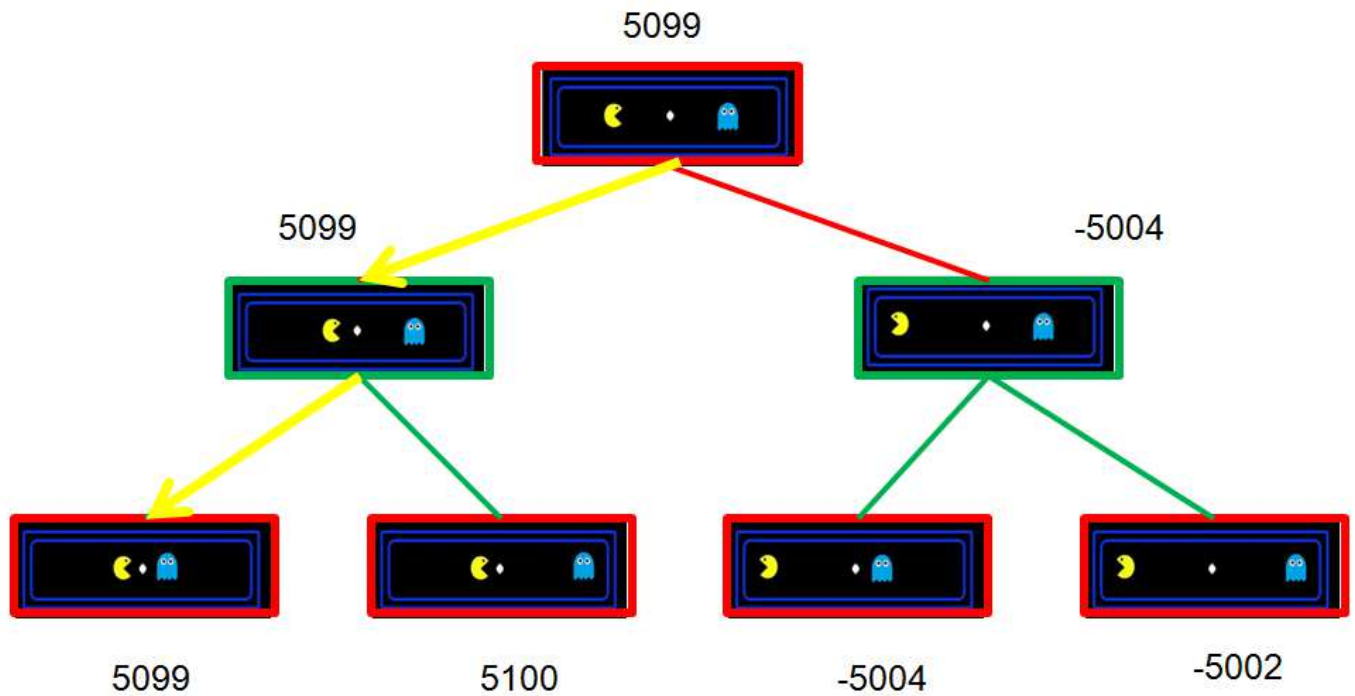
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for a, s in SUCCESSORS(state) do  $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$ 
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$ 
  for a, s in SUCCESSORS(state) do  $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$ 
  return v

```

4-, MiniMax Simulation:

MiniMax Simulation



Note: Sam! le simulation sho: n for "ame tree of de! th +-

7- Alpha-Beta Pruning Algorithm:

The MiniMax search tree can grow to be very large, and so it is common practice to use advanced techniques to limit the amount of time and resources that are required to do the MiniMax search. The easiest such technique is to simply limit the number of moves to look ahead (search depth). Another technique is called Alpha-Beta Pruning.

The Alpha-Beta Pruning scheme allows MiniMax to do all of the same analysis more efficiently, without losing any information. First we must always traverse the search tree in a predetermined order, say from left to right, top down, depth first; we will skip (prune) all the nodes that cannot influence the determination of the best value.

7-1 Alpha-Beta Pseudo Code:

```

function ALPHA-BETA-DECISION(state) returns an action
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))



---


function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
    inputs: state, current state in game
               $\alpha$ , the value of the best alternative for MAX along the path to state
               $\beta$ , the value of the best alternative for MIN along the path to state

    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow -\infty$ 
    for a, s in SUCCESSORS(state) do
         $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s, \alpha, \beta))$ 
        if  $v \geq \beta$  then return v
         $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
    return v

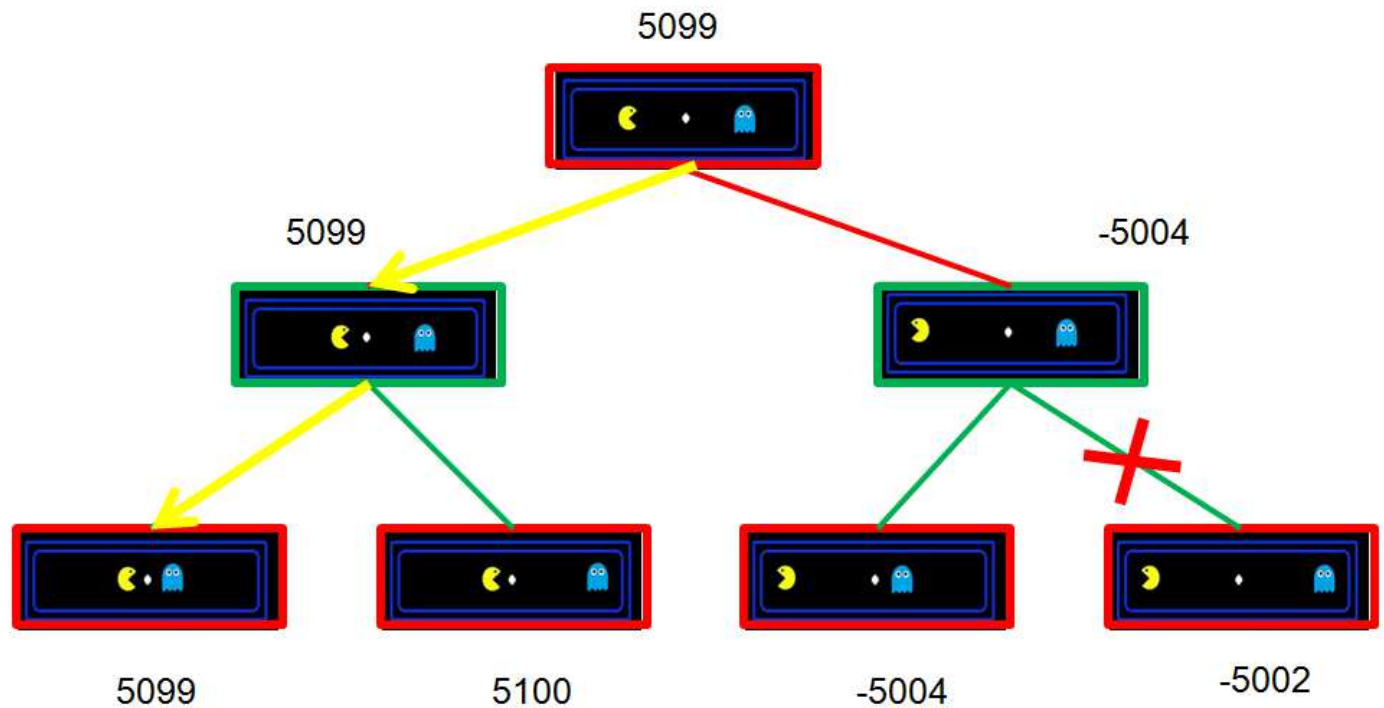


---





function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
    same as MAX-VALUE but with roles of  $\alpha$ ,  $\beta$  reversed
  
```

7-(AI! ha)eta Simulation:

Alpha-Beta Pruning Simulation



8- #esult Com! arison:

Map	Minimax Nodes analyzed	Alpha Beta Nodes analyzed	% nodes analysis reduction
	25	23	8%
	249	230	7.6%
	21074	10266	51.2%

9- .m! lementation Details:

9-' Some .m! ortant /lasses and their res! onsi3ilities-

1. Class Name: Map

Data :char map[][] - A two dimensional array containing the environment/map read.

Responsibility: Responsible for storing the map design of the game.

2. Class Name: Game Engine

Responsibility:

- i. Population game tree.

- ii. Solving minimax and alpha beta algorithm for the game tree and return a game plan.

3. Class Name: State

Data: Agent information, successor states, and score for the given state;

Responsibility: Store state level information.

4. Class Name: Agent

Data : Contains position information for a game agent/adversary. Also stores number of steps moved by the PAC Man and ghost.

9- (50%) of the responsibilities:

- a Problem definition/formulation.
- b Designing
- c Implementation
 - Algorithm for minimax and alpha beta pruning
 - Implementing and providing interfaces for User Interface/GUI.
- d Other project documentation including project report.

;- Future Scope:

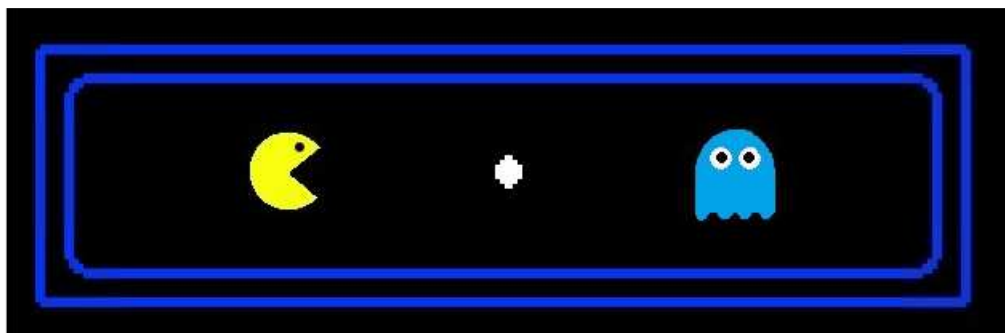
- Currently it is **not supporting big maze(Due to big game tree)** due to **limitations in processing capabilities**.
- This can be fixed by limiting the tree to a certain depth and then computing the next best move. i.e depth limited minimax/alpha-beta algorithm.
- Better evaluation function for increased Game- Intelligence.

Conclusion:

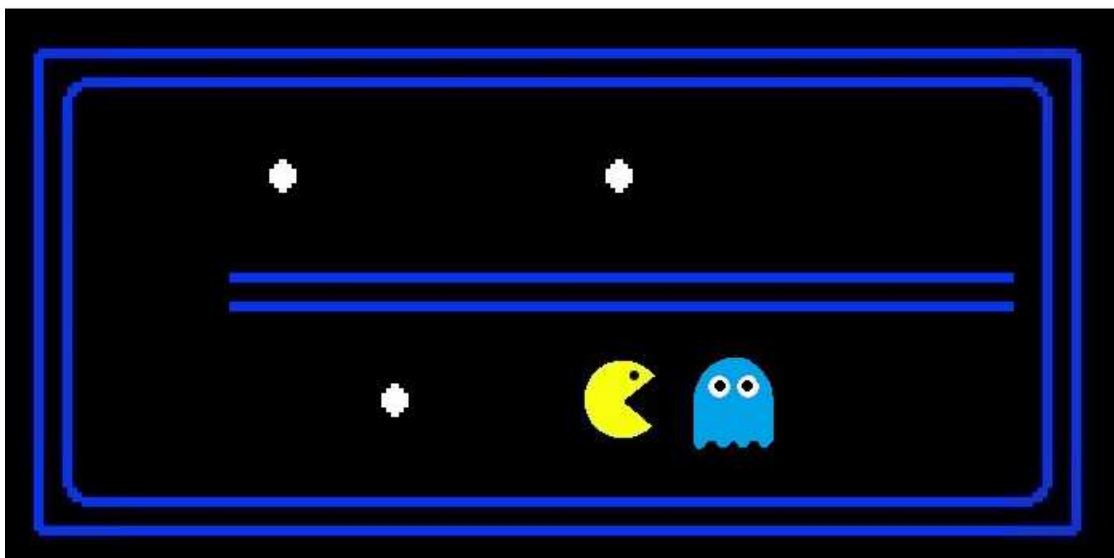
- Have successfully implemented MiniMax and alpha-beta pruning algorithm of game of Pac-Man.
- Alpha beta performs better than MiniMax. (Especially for bigger game trees)

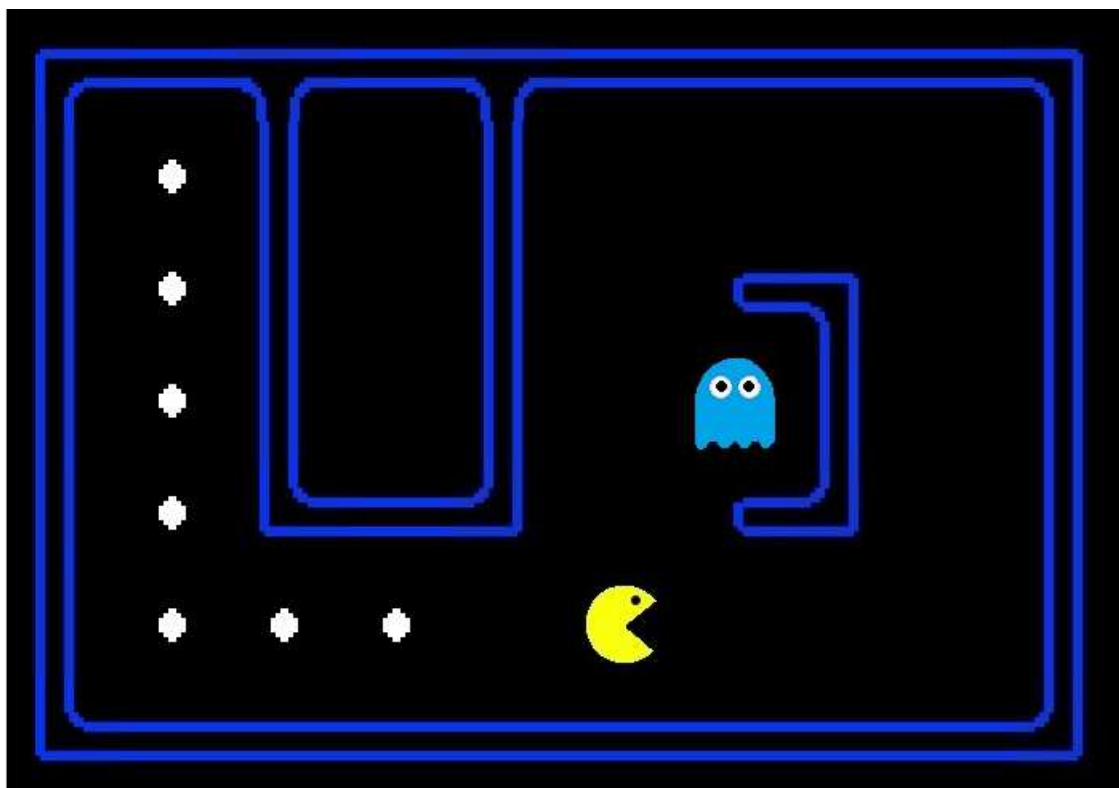
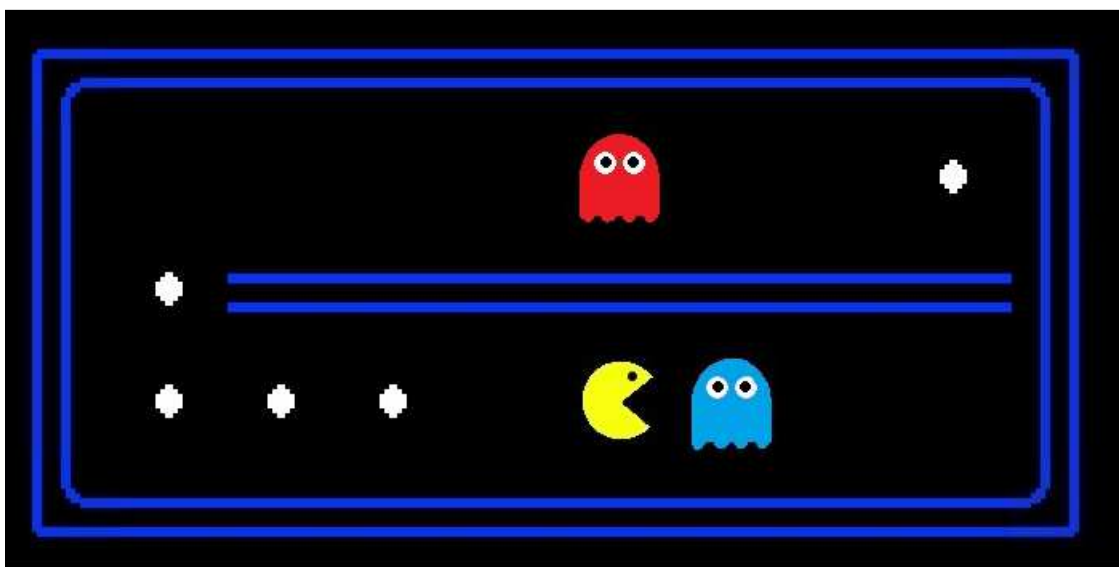
Appendix A: Game Snatches

Demo Map1:



Demo Map2:



Demo Map3:**Demo Map4:**

A!!endix): Sam! le Game /ode Sni!! ets

Function: populateGameTree - Populates the game tree

```
public void populateGameTree(){
    Queue<State> sQueue = new LinkedList<>();
    State currState = null;

    sQueue.add(root);

    while(sQueue.peek() != null){
        currState = sQueue.poll();

        // check for terminal state
        if(currState.isTerminalState()){
            currState.setSuccessorStates(null);
            continue;
        }

        // check for loop state
        if(isVisitedState(currState)){
            currState.setLoopState(true);
            currState.setSuccessorStates(null);
            continue;
        }

        // process state
        visitedStateSet.add(currState);

        // Update successor state list
        ArrayList<State> successorStateList =
            State.findSuccessorState(currState);
        currState.setSuccessorStates(successorStateList);

        // add successors in queue
        for(State s : successorStateList){
            sQueue.add(s);
        }
    }

    CommonAPIs.dumpGameTree(root);
}
```


Function: solveMinimax - Solves minimax of game tree

```
public ArrayList<State> solveMinimax(){
    ArrayList<State> list = maxValue(root);
    return list;
}

private ArrayList<State> maxValue(State currState){
    ArrayList<State> gamePlan,successorPlan = null;
    if(currState.isTerminalState()){
        gamePlan = new ArrayList<>();
        gamePlan.add(currState);
        return gamePlan;
    }

    ArrayList<State> maxSuccessorPlan = null;
    for(State s: currState.getSuccessorStates()){
        successorPlan = minValue(s);
        if(maxSuccessorPlan == null)
            maxSuccessorPlan = successorPlan;
        else
            maxSuccessorPlan =
                getMaxValueState(maxSuccessorPlan, successorPlan);
    }

    int index = maxSuccessorPlan.size() - 1;
    State sEnd = maxSuccessorPlan.get(index);
    currState.setStateValue(sEnd.getStateValue());
    maxSuccessorPlan.add(currState);
    return maxSuccessorPlan;
}

private ArrayList<State> minValue(State currState) {
    ArrayList<State> gamePlan,successorPlan = null;
    if(currState.isTerminalState()){
        gamePlan = new ArrayList<>();
        gamePlan.add(currState);
        return gamePlan;
    }

    ArrayList<State> minSuccessorPlan = null;
    for(State s: currState.getSuccessorStates()){
        successorPlan = maxValue(s);
        if(minSuccessorPlan == null)
            minSuccessorPlan = successorPlan;
    }
}
```

```

        else
            minSuccessorPlan =
                getMinValueState(minSuccessorPlan, successorPlan);
    }

    int index = minSuccessorPlan.size() - 1;
    State sEnd = minSuccessorPlan.get(index);
    currState.setStateValue(sEnd.getStateValue());
    minSuccessorPlan.add(currState);
    return minSuccessorPlan;
}

```

Function: solveAlphaBeta - Performs alpha beta pruning on the game tree

```

public ArrayList<core.State> solveAlphaBeta() {
    ArrayList<State> list =
        abMaxValue(root,Integer.MIN_VALUE,Integer.MAX_VALUE);
    return list;
}

private ArrayList<State> abMaxValue(State currState, int alpha, int beta){
    ArrayList<State> gamePlan,successorPlan = null;
    int value = Integer.MIN_VALUE;

    if(currState.isTerminalState()){
        gamePlan = new ArrayList<>();
        gamePlan.add(currState);
        return gamePlan;
    }

    ArrayList<State> maxSuccessorPlan = null;
    for(State s: currState.getSuccessorStates()){

        successorPlan = abMinValue(s,value,beta);
        State last = getLast(successorPlan);
        if(!last.isLoopState())
        {
            value = last.getStateValue();
            if(value >= beta){
                currState.setStateValue(value);
                successorPlan.add(currState);
            }
        }
    }
    return maxSuccessorPlan;
}

```

```

        System.out.println("***** pruned *****");
        CommonAPIs.printState(currState);
        return successorPlan;
    }
}

if(maxSuccessorPlan == null)
    maxSuccessorPlan = successorPlan;
else
    maxSuccessorPlan =
        getMaxValueState(maxSuccessorPlan, successorPlan);

// update value of alpha
alpha = maxOf(last.getStateValue(), alpha);
}

State sEnd = getLast(maxSuccessorPlan);
currState.setStateValue(sEnd.getStateValue());
maxSuccessorPlan.add(currState);
return maxSuccessorPlan;
}

private ArrayList<State> abMinValue(State currState, int alpha, int beta) {
    ArrayList<State> gamePlan, successorPlan = null;
    int value = Integer.MAX_VALUE;

    if(currState.isTerminalState()){
        gamePlan = new ArrayList<>();
        gamePlan.add(currState);
        return gamePlan;
    }

    ArrayList<State> minSuccessorPlan = null;
    for(State s: currState.getSuccessorStates()){

        successorPlan = abMaxValue(s, alpha, value);
        State last = getLast(successorPlan);
        if(!last.isLoopState())
        {
            value = last.getStateValue();
            if(value <= alpha){
                currState.setStateValue(value);
                successorPlan.add(currState);
                System.out.println("***** pruned *****");
                CommonAPIs.printState(currState);
            }
        }
    }
}

```

```
                return successorPlan;
            }
        }

        if(minSuccessorPlan == null)
            minSuccessorPlan = successorPlan;
        else
            minSuccessorPlan =
                getMinValueState(minSuccessorPlan, successorPlan);

        // update value of beta
        beta = minOf(last.getStateValue(), beta);
    }

    State sEnd = getLast(minSuccessorPlan);
    currState.setStateValue(sEnd.getStateValue());
    minSuccessorPlan.add(currState);
    return minSuccessorPlan;
}
```

#eferen/es:

- <http://en.wikipedia.org/wiki/Pac-Man>
- **Text Book: Artificial Intelligence: A Modern Approach.**
- **Images: Google**