Cairo University
Faculty of Computers & Artificial Intelligence
Department of Computer Science
Artificial Intelligence Course

Lab 6: Game Playing

Game playing in artificial intelligence involves writing a computer program that can play a certain game successfully. A game can be considered a search problem with states and moves; however, the traditional search techniques are not entirely suitable or even accurate to use with games. That's why specific search techniques/algorithms were developed for game playing.

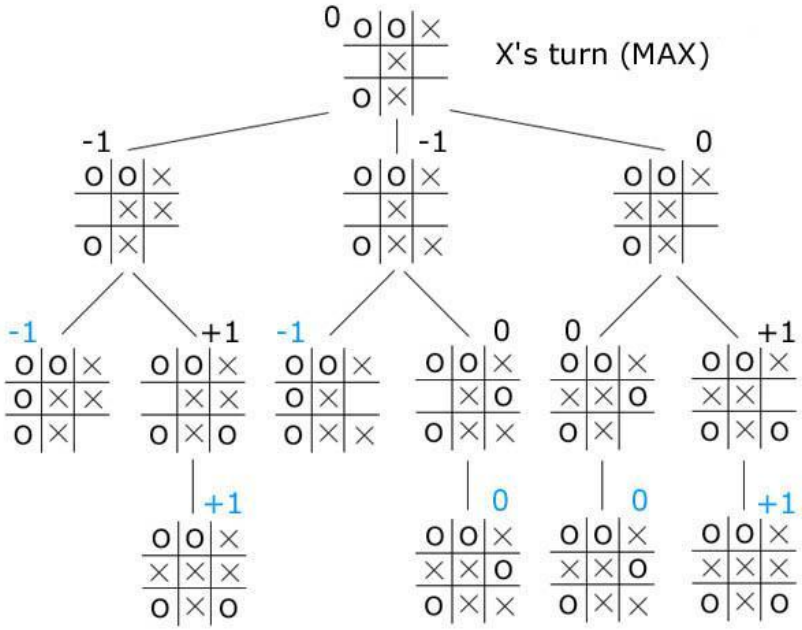
Minimax Algorithm:

Minimax is **a recursive algorithm that can find the optimal move** for the computer **by traversing the game tree (depth first) assuming the opponent plays optimally.**

In this algorithm, **two players play the game against each other;** the first is **MAX** who aims at maximizing the value of the state while the second is **MIN** who aims at minimizing the value. In other words, both players fight in alternating turns to get the maximum benefit and leave their opponent with the minimum benefit.

Example:

Suppose we have simple 2-player game like tic-tac-toe and it's the computer's turn to play from the current state. So, it will run "Minimax" from the current state and build the game tree as follows:



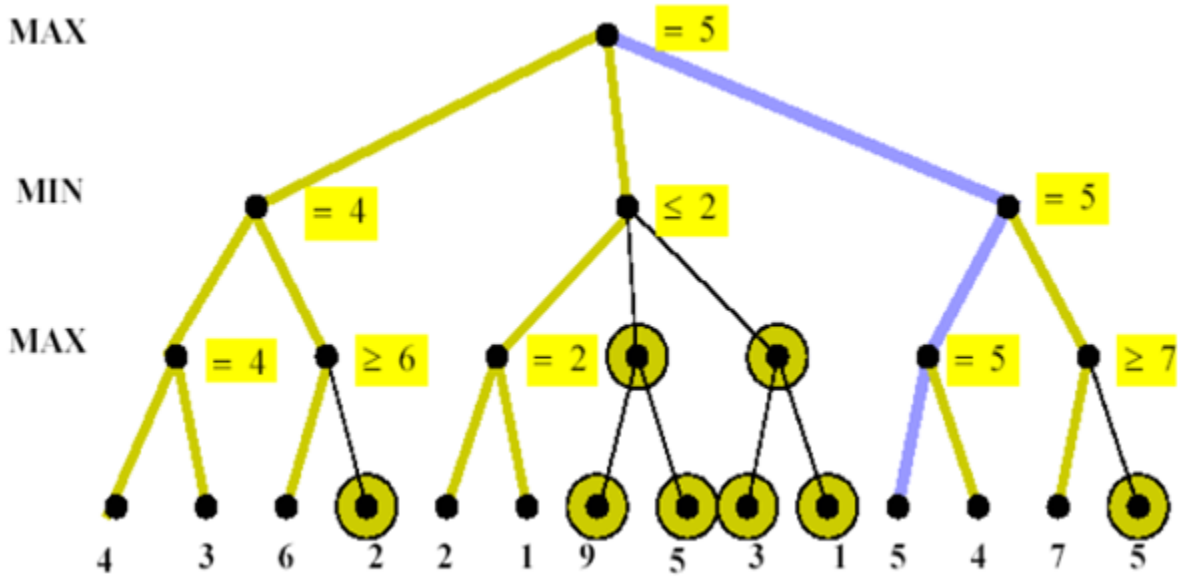
The algorithm traverses the tree using **depth first search.** A state (node) in the tree can either be a **terminal state for which a utility function can be applied** and returns a value, or a **non-terminal state which has child states** that need to be evaluated through minimax recursively. Once all the children of a state have been evaluated, the best value for this state/node is chosen depending on who was going to play at this state MAX or MIN.

Alpha-Beta Pruning:

Notice that minimax explores the entire game tree, even the branches that won't affect the final path. This can be computationally expensive, so **alpha-beta pruning seeks to solve this problem by decreasing the number of nodes evaluated by minimax.**

Basically, **alpha-beta pruning keeps track of two values (alpha and beta)** at each node. Alpha is updated if the current node is MAX and has a higher value than alpha while beta is updated if the current node is MIN and has a lower value than beta. If, **at any node, the value of beta becomes less than or equal to alpha, we can prune** the next child(ren) of the current node. Initially alpha is set to negative infinity while beta is set to infinity.

Example:



Tic-Tac-Toe & Minimax in Prolog:

We first start by writing the necessary predicates and rules for tic-tac-toe in "game.pl":

- We have 2 players: a human player and the computer.
- Both players take turns and play.
- When it's the human's turn, Prolog reads the user input.
- When it's the computer's turn, the computer runs the minimax algorithm.
- This process continues until a terminal state where a player won, or the board was full is reached.

```
startGame:-
    write("Do you want x or o?"), nl,
    read(Human),
    % Human player can be x or o but x is always MAX and o is MIN
    StartState = ['-','-','-','-','-','-','-','-','-'],
    play([Human, StartState], Human).

otherPlayer(x,o).
otherPlayer(o,x).

play([_, State], _):-
    isTerminal(State), !,
    nl, draw(State), nl.

play([Human, State], Human):-
    !, nl, draw(State), nl,
    write('Enter your move\'s index'),nl,
    read(HumanMove),
    % Check that the move is valid
      % then replace in the board(using select & insert)
    nth0(HumanMove, State, '-', TmpList),
    nth0(HumanMove, NextState, Human, TmpList),
    otherPlayer(Human, Computer),
    play([Computer, NextState], Human).

play([Computer, State], Human):-
    nl, draw(State), nl,
    computerTurn([Computer,State], NextState),
    play(NextState, Human).

computerTurn(State, Next):-
    minimax(State, Next, _).

isTerminal(State):-
    getWinner(State, Winner), write(Winner), write(' wins!'), nl, !.

isTerminal(State):-
    not(member('-', State)), write('It\'s a draw!'), nl.
```

```
getWinner(State, Winner):-
    ( (State = [Z,Z,Z,_,_,_,_,_,_], !);
      (State = [_,_,_,Z,Z,Z,_,_,_], !);
      (State = [_,_,_,_,_,_,Z,Z,Z], !);
      (State = [Z,_,_,Z,_,_,Z,_,_], !);
      (State = [_,Z,_,_,Z,_,_,Z,_], !);
      (State = [_,_,Z,_,_,Z,_,_,Z], !);
      (State = [Z,_,_,_,Z,_,_,_,Z], !);
      (State = [_,_,Z,_,Z,_,Z,_,_], !) ), Z \= '-', Winner = Z.

draw([]):-!.
draw([H|T]):-
    length(T, N),
    write(H),
    (0 is N mod 3 -> nl ; write('   ')),
    draw(T).
```

Secondly, we move on to the code of minimax in the "solver.pl" file:

- Try to get the children of the current state.
- If the state has children, choose the child with the best value, and return it. In order to choose the best child, we have to run minimax again from each child to get its value and compare it with the other children.
- If the state has no children, calculate its value using a utility function.

```
minimax(Pos, BestNextPos, Val):-
    bagof(NextPos, move(Pos, NextPos), NextPosList),
    best(NextPosList, BestNextPos, Val), !.

minimax(Pos, _, Val):-
    utility(Pos, Val).

best([Pos], Pos, Val):-
    minimax(Pos, _, Val), !.

best([Pos1 | Tail], BestPos, BestVal) :-
    minimax(Pos1, _, Val1),
    best(Tail, Pos2, Val2),
    betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal) :-
    isMinPlayer(Pos1), % next (Pos1) is MIN, so the parent is MAX
    (Val1 >= Val2 -> (BestPos = Pos1, BestVal is Val1, !)
                  ;  (BestPos = Pos2, BestVal is Val2)
    ), !.

betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal) :-
    (Val1 =< Val2 -> (BestPos = Pos1, BestVal is Val1, !)
                  ;  (BestPos = Pos2, BestVal is Val2)
    ), !.
```

Finally, we have 3 missing rules that are game-specific, yet minimax still needs them to run:

- **Move:** to get a new possible state from the current state.
- **Utility function:** to calculate the value of a terminal state.
- **Player checker:** to check whether the player at a node is MAX or MIN.

We can write these rules for tic-tac-toe as follows (and put them in the **"game .pl"** file):

```
move([Player, State], Next):-
     not(getWinner(State,_)),
     getMove([Player, State], Next).

getMove([Player, State], [NextPlayer, NextState]):-
    otherPlayer(Player, NextPlayer),
    State = ['-'|T],
    NextState = [Player|T].

getMove([Player, State], [NextPlayer, NextState]):-
    State = [H|T],
    NextState = [H|NextT],
    move([Player,T], [NextPlayer, NextT]).

utility([_,State], Val):-
    getWinner(State, Winner),!,
    ((Winner = x, Val = 1, !);
    (Winner = o, Val = -1, !)).

utility(_,0).

isMinPlayer([o,_]). % o is the next player
```

Tic-Tac-Toe & Alpha-Beta Pruning in Prolog:

To use alpha-beta pruning with our game, we only need to change the "solver.pl" file:

- Keep 2 new arguments in the alpha-beta rule (alpha and beta).
- Update alpha **or** beta at each node.
- Check if beta is less than or equal to alpha and if so, cut!

```
alphabeta(Pos, Alpha, Beta, BestNextPos, Val):-
    bagof(NextPos, move(Pos, NextPos), NextPosList),
    best(NextPosList, Alpha, Beta, BestNextPos, Val), !.

alphabeta(Pos, _, _,_, Val):-
    utility(Pos, Val).

best([Pos|_],Alpha, Beta, _, Val):-
    Beta =< Alpha, !,
    (isMinPlayer(Pos) -> Val is Alpha ; Val is Beta).
```

```
best([Pos], Alpha, Beta, Pos, Val):-
    alphabeta(Pos, Alpha, Beta, _, Val), !.

best([Pos1 | Tail], Alpha, Beta, BestPos, BestVal) :-
    alphabeta(Pos1, Alpha, Beta, _, Val1),
    updateValues(Pos1, Val1, Alpha, Beta, NewAlpha, NewBeta),
    best(Tail, NewAlpha, NewBeta, Pos2, Val2),
    betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal).

updateValues(Pos1, Value, Alpha, Beta, NewAlpha, Beta):-
    isMinPlayer(Pos1), !,
    (Value > Alpha -> (NewAlpha is Value, !)
                   ;   NewAlpha is Alpha
    ).

updateValues(_, Value, Alpha, Beta, Alpha, NewBeta):-
    (Value < Beta -> (NewBeta is Value, !)
                  ;   NewBeta is Beta
    ).

betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal) :-
    isMinPlayer(Pos1),
    (Val1 >= Val2 -> (BestPos = Pos1, BestVal is Val1, !)
                  ;  (BestPos = Pos2, BestVal is Val2)
    ), !.

betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal) :-
    (Val1 =< Val2 -> (BestPos = Pos1, BestVal is Val1, !)
                  ;  (BestPos = Pos2, BestVal is Val2)
    ), !.
```

Don't forget to change the "computerTurn" in the game file to make it use the "alphabeta" rule instead of minimax.

E.g. "**`alphabeta(State, -100, 100, Next, _).`**"