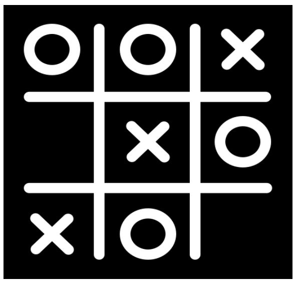
Lab-3

Minimax agent for Tic-Tac-Toe



We're assuming everyone knows what Tic-Tac-Toe is. It's this >
The first player to get 3 of his characters in a row/column/diagonal wins.
If both players play optimally, Tic-Tac-Toe always ends in a draw.
Your agent should be unbeatable.

Some numbers

The branching factor at turn i is (9-i). The maximum depth is 9.
A (rather loose) upper bound would suggest that the tree can have a max of 9! = 362880 nodes.
(In reality, there are only close to 6000 legal configurations of the board.)
This means we can actually reach the leaves of the game tree.
If your minimax implementation is decent, You can expect to get a move for even an empty board in under a minute and any further moves in a few seconds.

Today's problem

Write an unbeatable minimax agent for the Tic-Tac-Toe game.

Search the tree till the leaves.

State:

Use a list of 9 elements to represent the state of the board.

Use '–' for a blank, '*o*' and '*x*' for the players.

The configuration in the image corresponds to the list: *[o,o,x, -,x,o, x,o,-]*

Predicates:

The predicate declarations are based on our solution. If you wnat to, you can change it to suit your approach.

The underlined terms are those to be 'returned'. We've tried to stick as close to the slide code as possible.

The following are predicates you need to define

minimax_decision(State, Player,   <u>Action</u>, <u>Value</u>):-

- Given a *State*, Return the optimal *Action* that *Player* can take and the optimal *Value* that can be attained.
- *Player* is either o or x.
- *Action* is a list of the form [character, position].
    - Use get_action(State, SuccessorState) to get the action.

min-value(State, Player, <u>sv(MinSuccessorState, MinValue)</u>):-

- Find the minimum utility that the game can be taken to from a *State.*
- *sv(MinSuccessorState, MinValue)* is a structure. *MinSuccessorState* is the successor which takes the game to the minimum utility. *MinValue* is the minimum utility achieved.
- Here, *Player* is the symbol of the Agent. Its aim is always to maximize utility.
- In a min-value call, it is the turn of the adversary. Use the *other_player* predicate to get the symbol of the adversary.

max-value(State, Player, <u>sv(MaxSuccessorState, MaxValue)</u>):-

- Find the maximum utility that the game can be taken to from a *State.*
- *sv(MaxSuccessorState, MaxValue)* is a structure. *MaxSuccessorState* is the successor which takes the game to the minimum utility. *MaxValue* is the minimum utility achieved.
- In a max-value call, it is the turn of the Agent.

terminal_state(State, Player, <u>Value </u>):-

- True if *State* is a terminal state.
- *Value* is 1 if *Player* wins, 0 if draw, -1 if *Player* loses.

The following are predicates we've defined for you so you can just implement minimax.

successor(State, Player, <u>Successor </u>):-

- *Successor* is a state reached if *Player* makes a move from *State.*
- Use with findall to get all successors of *State*

other_player(Player, OtherPlayer).

- Returns the symbol of the other player.

get_action(FromState, ToState, <u>Action</u>):-

- Returns the *Action* that takes from *FromState* to *ToState*
- *Action* is a list of the form [character, position]. (Refer code)

find_max_sv(SVList, MaxSV):-

- MaxSV is the sv structure in SVList having the maximum value

find_min_sv(SVList, MinSV):-

- MinSV is the sv structure in SVList having the minimum value

The following are predicates you don't have to use directly.

winning_config(WinningStates, Player):-

- *WinningStates* is a list of states where *Player* wins.

list_replace(Find, Replace, In, Result):-

- Replaces an occurence of *Find* in *In* with *Replace* and the result is in *Result.*
- Used in the successor function as list_replace(-, Player, State, SuccessorState)

MINIMAX

MINIMAX Recurrence

Given a game tree, the optimal strategy can be determined from the minimax value of each state, which we write as **MINIMAX (s)**. The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following :

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

MINIMAX Algorithm

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations.

Minimax algorithm

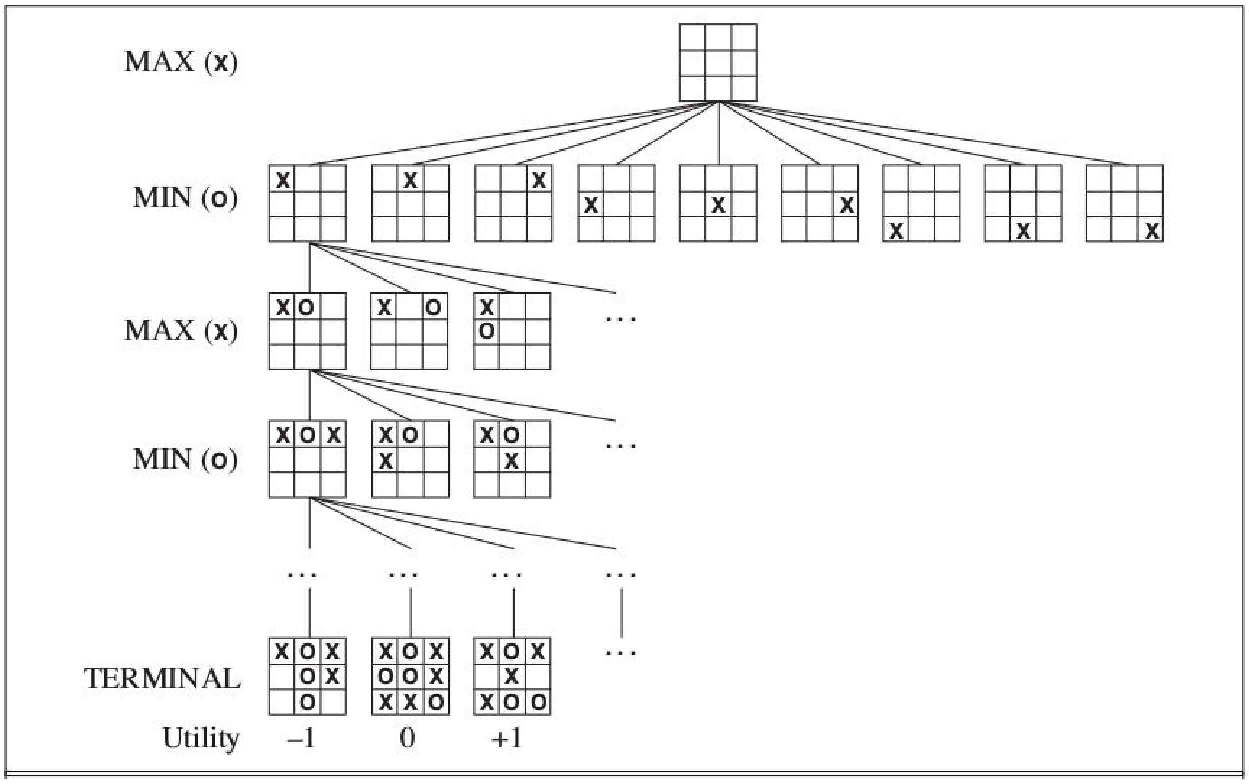
```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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 $v \leftarrow$ MAX(v, 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 $v \leftarrow$ MIN(v, MAX-VALUE(s))
    return v
```

MINIMAX game tree

A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX(X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.


MAX (x)
MIN (o)
MAX (x)
MIN (o)
TERMINAL
Utility
–1
0
+1