Game Playing

A (pure) strategy:

a complete set of advance instructions that specifies a definite choice for every conceivable situation in which the player may be required to act.

In a two-player game, a strategy allows the player to have a response to every move of the opponent.

Game-playing programs implement a strategy as a software mechanism that supplies the right move on request.

Two-Person Perfect Information Deterministic Game

- Two players take turns making moves
- Call one Min and the other Max
- Deterministic moves: Board state fully known,
- One player wins by defeating the other (or else there is a tie)
- Want a strategy to win, assuming the other person plays rationally

A logic-based approach to games

Find a winning strategy by ***proving*** that the game can be won -- use backward chaining.

A very simple game: nim.

- initially, there is one stack of chips;
- a move: select a stack and divide it in two unequal non-empty stacks;
- a player who cannot move loses the game.

(The player who moves first can win.)

Nim

Static evaluation

A **static evaluation function** returns the value of a move without trying to play (which would mean simulating the rest of the game but not playing it).

"Usually" a static evaluation function returns positive values for positions advantageous to **Player 1**, negative values for positions advantageous to **Player 2.**

If **Player 1** is rational, he will choose the maximal value of a leaf.
Then, **Player 2** will choose the minimal value.

Static evaluation

If we can have (guess or calculate) the value of an internal node **N**, we can treat it as if it were a leaf. This is the basis of the minimax procedure.

No tree would be necessary if we could evaluate the initial position statically. Normally we need a tree, and we need to look-ahead into it. Further positions can be evaluated more precisely, because there is more information, and a more focussed search.

Minimax Tree

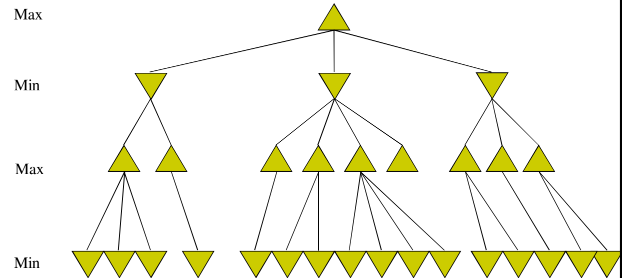
- Create a utility function
  - Evaluation of board/game state to determine how strong the position of each player.
  - Player 1 wants to maximize the utility function
  - Player 2 wants to minimize the utility function
- Minimax tree
  - Generate a new level for each move
  - Levels alternate between "max" (player 1 moves) and "min" (player 2 moves)

Minimax Tree Evaluation

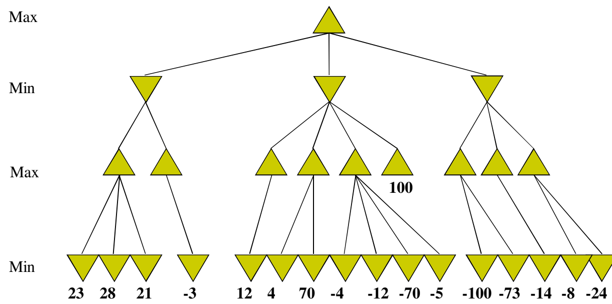
- Assign utility values to leaves
  - If leaf is a "final" state, assign the maximum or minimum possible utility value (depending on who would win)
  - If leaf is not a "final" state, must use some other heuristic, specific to the game, to evaluate how good/bad the state is at that point

```
minimax(player,board)
      If(game over in current board position) return winner
      children = all legal moves for player from this board
      if(max's turn)
            return maximal score of calling minimax on all the children
      else (min's turn)
             return minimal score of calling minimax on all the children
```

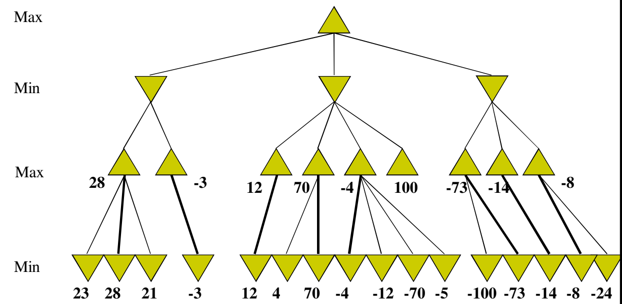
Minimax tree



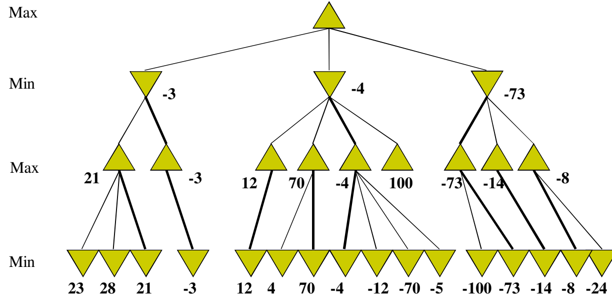
Minimax tree



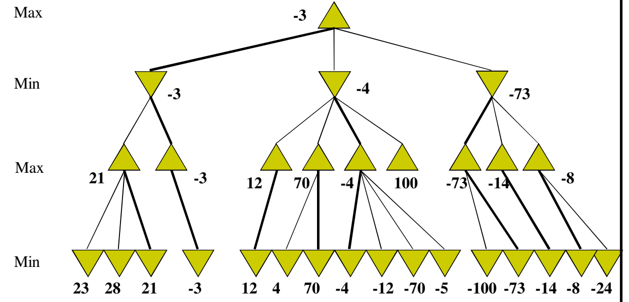
Minimax tree



Minimax tree



Minimax tree



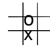
Tic-Tac-Toe

Let player **A** be x and let open(x), open(o) mean the number of lines open to x and o. There are 8 lines. An evaluation function for position P:

f(P) = -∞ if o wins

f(P) = +∞ if x wins, otherwise

f(P) = open(x) - open(o)

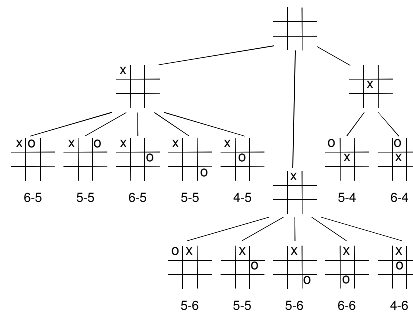


Example:

open(x) - open(o) = 4 - 6

Assumptions:

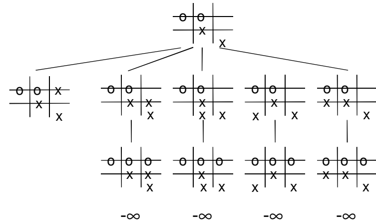
only one of symmetrical positions is generated;



Player **B** chooses the minimal backed-up value among level 1 nodes.

Player **A** chooses the maximal value, and makes the move.

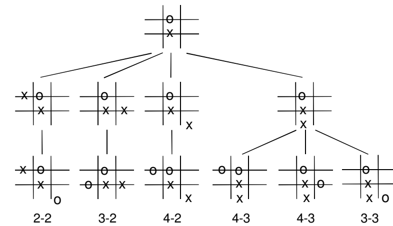
Player **B**, as a rational agent, selects the optimal response.

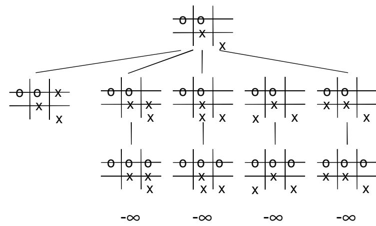


Building complete piles is usually not necessary. If we evaluate a position when it is generated, we may save a lot.

Assume that we are at a minimizing level. If the evaluation function returns -∞, we do not need to consider other positions:
-∞ will be the minimum.

The same applies to +∞ at a maximizing level.





Building complete piles is usually not necessary. If we evaluate a position when it is generated, we may save a lot.

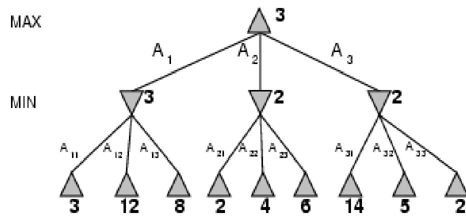
Assume that we are at a minimizing level. If the evaluation function returns -∞, we do not need to consider other positions:
-∞ will be the minimum.

The same applies to +∞ at a maximizing level.

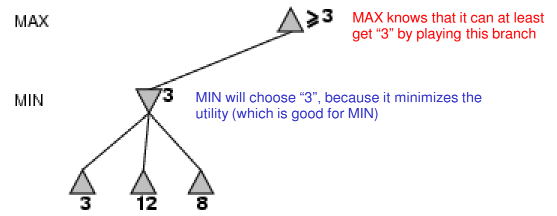
Pruning the Minimax Tree

- Minimax works best for large trees, but it can be useful even in mini-games such as tic-tac-toe.
- Since we have limited time available, we want to avoid unnecessary computation in the minimax tree.
- Pruning: ways of determining that certain branches will not be useful. Then cut of these branches

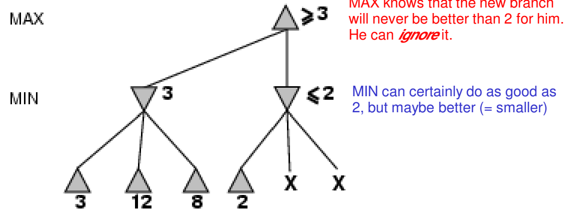
pruning



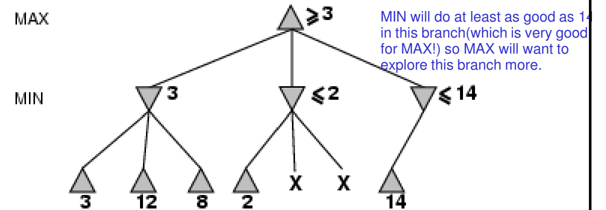
α pruning



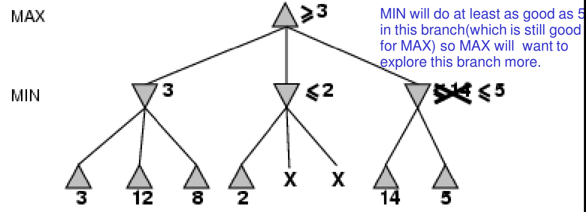
α pruning



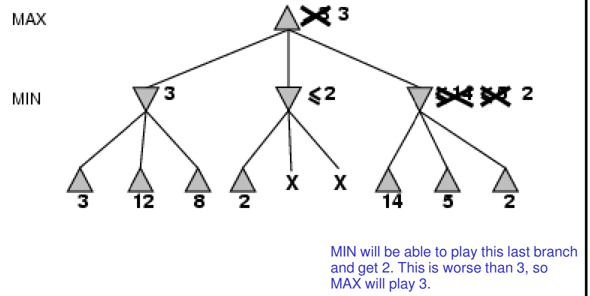
α pruning



α pruning



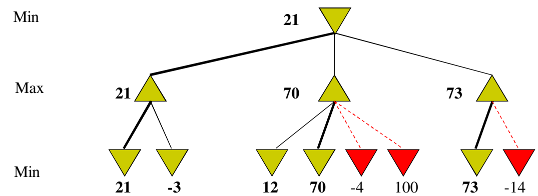
α pruning



β pruning

- Similar idea to α pruning, but the other way around
- If the current minimum is less than the successor’s max value, don’t look down that max tree any more

β pruning example



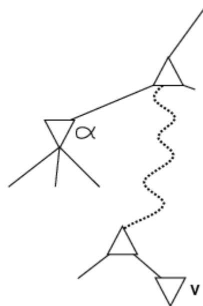
- Some subtrees at second level already have values > min from previous, so we can stop evaluating them.

Why is it called α-β?

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for ***max***
- If ***v*** is worse than α, ***max*** will avoid v

→ prune that branch

- Define β similarly for ***min***



α-β Pruning properties

- Pruning by these cuts does not affect final result
  - May allow you to go much deeper in tree
- Properties:
  - Evaluating "best" branch first yields better likelihood of pruning later branches
  - Perfect ordering reduces time to $b^{m/2}$

Properties of minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an rational opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)

- For chess, b ≈ 35, m ≈100 for "reasonable" games
  → exact solution completely infeasible