# Game Playing

## Search the action space of 2 players

Russell & Norvig Chapter 5

Bratko Chapter 24

University of Amsterdam

# Game Playing

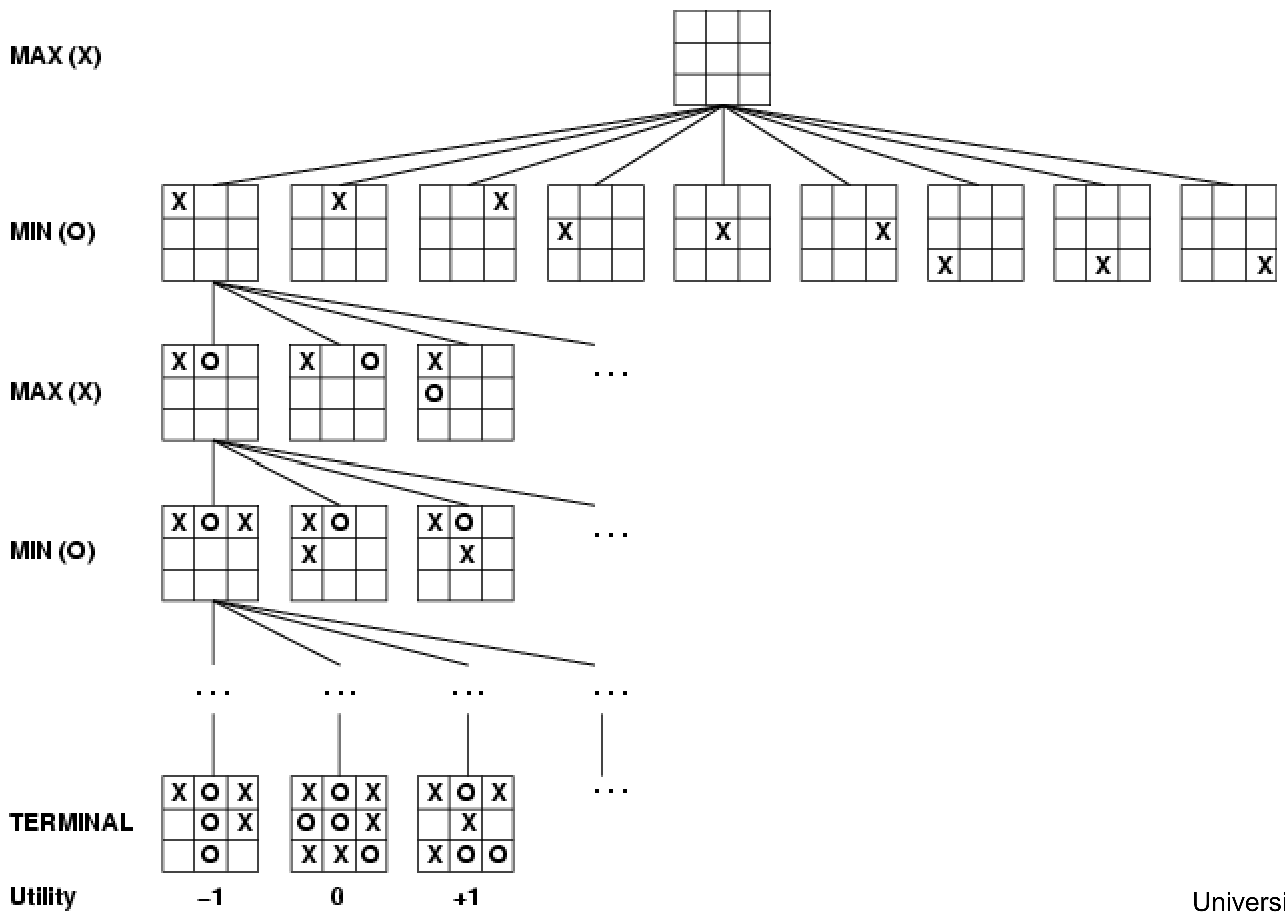
- ‘Games contribute to AI like Formula 1 racing contributes to automobile design.’
- ‘Games, like the real world, require the ability to make *some* decision, even when the *optimal* decision is infeasible.’
- ‘Games penalize inefficiency severely’.

# Games vs. search problems

- "Unpredictable" opponent → specifying a move for every possible opponent reply
- Time limits → unlikely to find *the* solution, must approximate *a* solution

University of Amsterdam

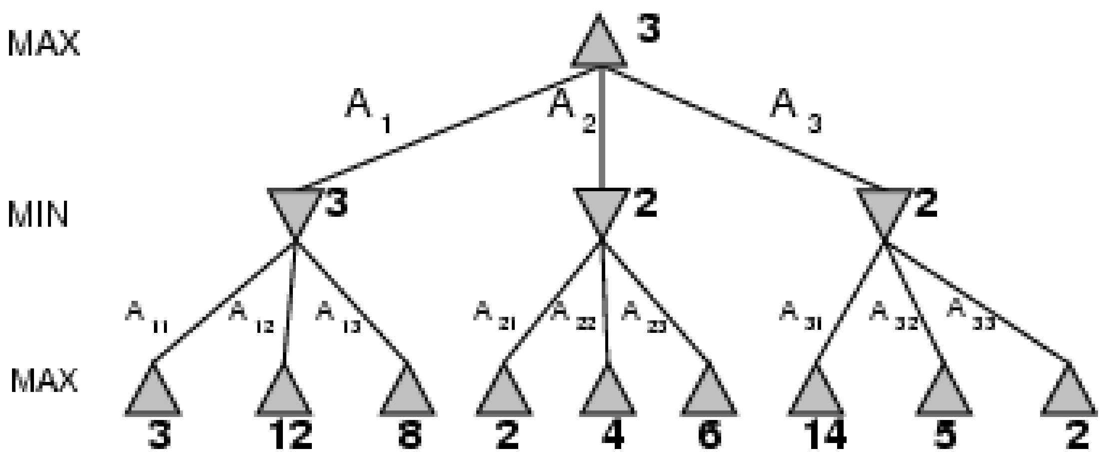
# Game tree of tic-tac-toe
## (2-player, deterministic, turn-taking, zero sum)




University of Amsterdam

# Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value = best achievable payoff against perfect playing opponent
- E.g., 2-ply game:



# Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

# Minimax prolog implementation

```
minimax( Pos, BestSucc, Val)  :-
  moves( Pos, PosList), !,                 % Legal moves in Pos
  best( PosList, BestSucc, Val)
   ;
  staticval( Pos, Val).                    % Terminal Pos has no successors

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

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

betterof( Pos0, Val0, Pos1, Val1, Pos0, Val0)  :-
  min_to_move( Pos0), Val0 > Val1, !       % MAX prefers the greater value
  ;
  max_to_move( Pos0), Val0 < Val1, !.      % MIN prefers the lesser value

betterof( Pos0, Val0, Pos1, Val1, Pos1, Val1).
% Otherwise Pos1 better than Pos0
```

# Minimax Python implementation

```
def minimax_decision(state, game):
    """Given a state in a game, calculate the best move by searching
    forward all the way to the terminal states. [Fig. 6.4]"""

    player = game.to_move(state)

    def max_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for (a, s) in game.successors(state):
            v = max(v, min_value(s))
        return v

    def min_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for (a, s) in game.successors(state):
            v = min(v, max_value(s))
        return v

    # Body of minimax_decision starts here:
    action, state = argmax(game.successors(state),
                           lambda ((a, s)): min_value(s))
    return action
```

This pseudo code is provided by [Russell & Norvig]

# Game interface

- Bratko's implementation: fig22_3.txt
- The tic-tac-toe game interface is based on 4 relations:

```
moves( Pos, PosList)        % Legal moves in Pos, fails when Pos is terminal
staticval( Pos, Val).       % value of a Terminal node (utility function)
min_to_move( Pos )          % the opponents turn
max_to_move( Pos )          % our turn
```

- Bratko's terminal position are win (+1) or loose (-1),

# Game interface

- Russell & Norovig implementation:
- The game interface is based on 4 functions:

```
game.successors(state)
game.utility(state, player)
game.to_move(state)
game.terminal_test(state)
```

# Properties of minimax

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

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

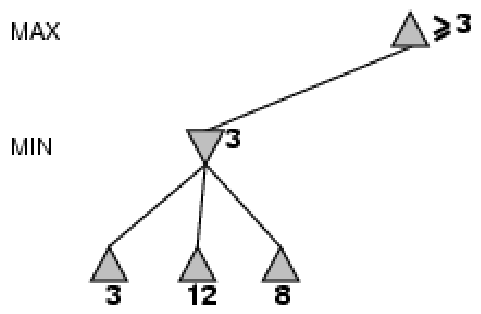
University of Amsterdam

# α-β pruning

- Efficient minimaxing
- Idea: once a move is clearly inferior to a previous move, it is not necessary to know *exactly* how much inferior.
- Introduce two bounds:

  Alpha = minimal value the MAX is guaranteed to achieve

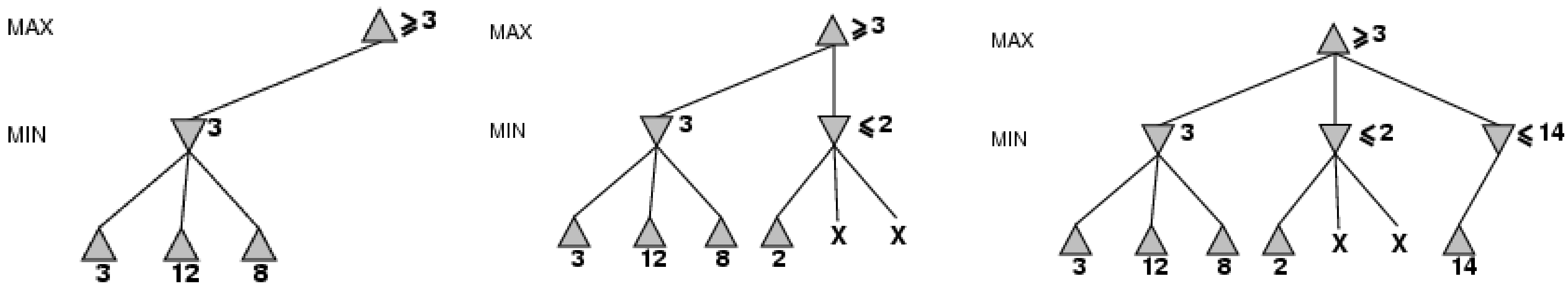
  Beta = maximal value the MAX can hope to achieve
- Example:




University of Amsterdam

# α-β pruning

- Example:




Alpha = 3

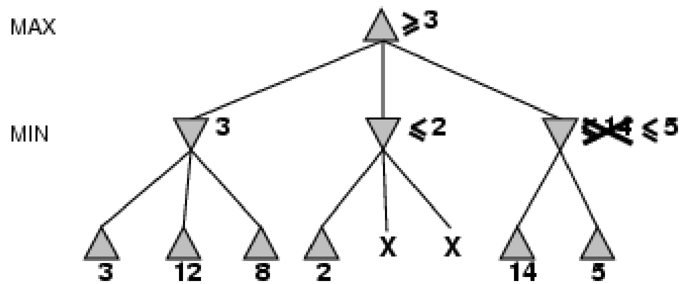
Val < Alpha,
!

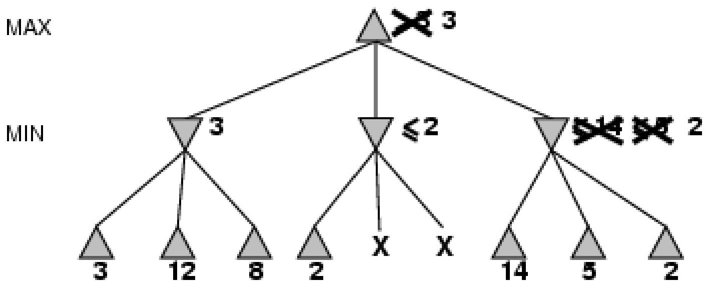
Val > Alpha
Newbound(β)

University of Amsterdam

# α-β pruning

- Example:




Val > α

Newbound(β)

Val < α

!

University of Amsterdam

# Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{m/2})$
  → doubles depth of search
- A simple example of the value of reasoning about which computations are relevant
  (a form of meta-reasoning)

University of Amsterdam

# AlphaBeta prolog implementation

```
alphabeta( Pos, Alpha, Beta, GoodPos, Val) :-
  moves( Pos, PosList), !,                    % Legal moves in Pos
  boundedbest( PosList, Alpha, Beta, GoodPos, Val)
   ;
  staticval( Pos, Val).                       % Terminal Pos has no successors

boundedbest( [Pos | PosList], Alpha, Beta, GoodPos, GoodVal) :-
  alphabeta( Pos, Alpha, Beta, _, Val),
  goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).
…
goodenough( _, Alpha, Beta, Pos, Val, Pos, Val) :-
  min_to_move( Pos), Val > Beta, !         % MAX prefers the greater value
  ;
  max_to_move( Pos), Val < Alpha, !.       % MIN prefers the lesser value

goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal)  :-
  newbounds( Alpha, Beta, Pos, Val, NewAlpha, NewBeta),     % Refine bounds
  boundedbest( PosList, NewAlpha, NewBeta, Pos1, Val1),
  betterof( Pos, Val, Pos1, Val1, GoodPos, GoodVal).
```

# AlphaBeta Python implementation

```
def alphabeta_full_search(state, game):
    """Search game to determine best action; use alpha-beta pruning.
    As in [Fig. 6.7], this version searches all the way to the leaves."""

    player = game.to_move(state)

    def max_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for (a, s) in game.successors(state):
            v = max(v, min_value(s, alpha, beta))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for (a, s) in game.successors(state):
            v = min(v, max_value(s, alpha, beta))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    # Body of alphabeta_search starts here:
    action, state = argmax(game.successors(state),
                           lambda ((a, s)): min_value(s, -infinity, infinity))
    return action
```

# Properties of α-β implementation

+ straightforward implementation

- It doesn't answer the solution tree

- With the depth-first strategy, it is difficult to control

University of Amsterdam

# Prolog assignment

- Download AlphaBeta implementation from Bratko: fig22_5.txt
- Replace in your solution minimax for AlphaBeta. Create test-routines to inspect the performance difference

```
alphabeta( Pos, Alpha, Beta, GoodPos, Val, MaxDepth)
```