CS W4701
Artificial Intelligence

Fall 2013

Chapter 5:

Adversarial Search

Jonathan Voris

(based on slides by Sal Stolfo)

Warm Up

- Let’s play some games!

Outline

- Optimal decisions
- Imperfect, real-time decisions
- α-β pruning

Games vs. Search Problems

- "Unpredictable" opponent
  - Specifying a move for every possible opponent reply
- Time limits
  - Unlikely to locate goal, must approximate

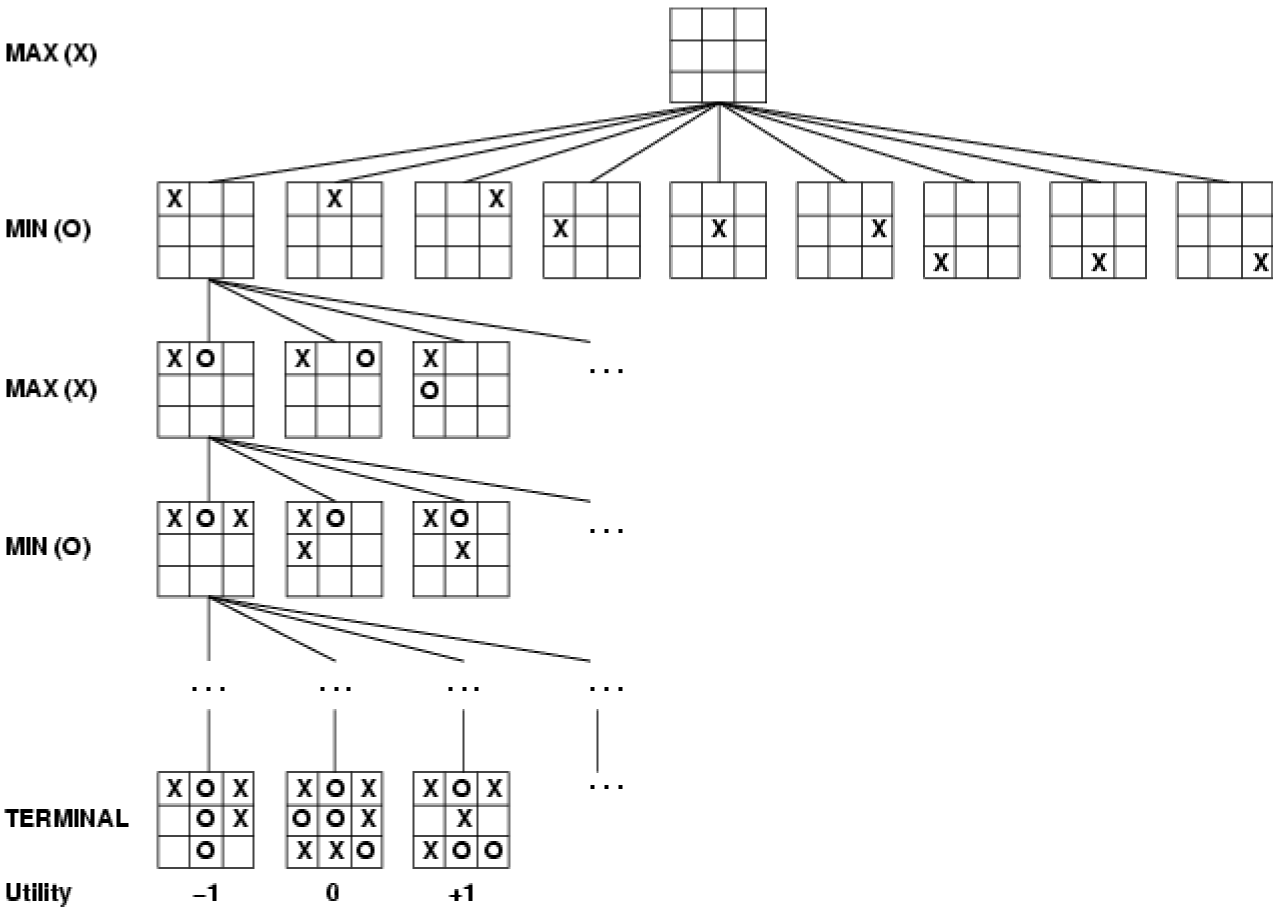
Game Tree Definitions

- s_0: start state
- player(s) – whose turn is it?
- action(s) – options?
- result(s,a) – outcome of action
- terminal-test(s) – game over?
- utility(s,p) – value of end state to player p

Minimax Search

- Core of many computer games
- Pertains primarily to:
  - Turn based games
  - Two players
  - Players with “perfect knowledge”
  - Zero-sum
    - At end of game, player utilities are “equal and opposite”

Game Tree
(2-player, Deterministic, Turns)



Game Tree

- Nodes are states
- Edges are decisions
- Levels are called “plys”

Naïve Approach

- Agent must develop a *strategy*
  - A move for each state
  - A ? agent uses a ? stratgey
- Given a game tree, what would be the most straightforward playing approach?

Evaluation Functions

- Assign a utility score to a state
  - Different for players?
- Usually a range of integers
  - [-1000,+1000]
- +infinity for win
- -infinity for loss

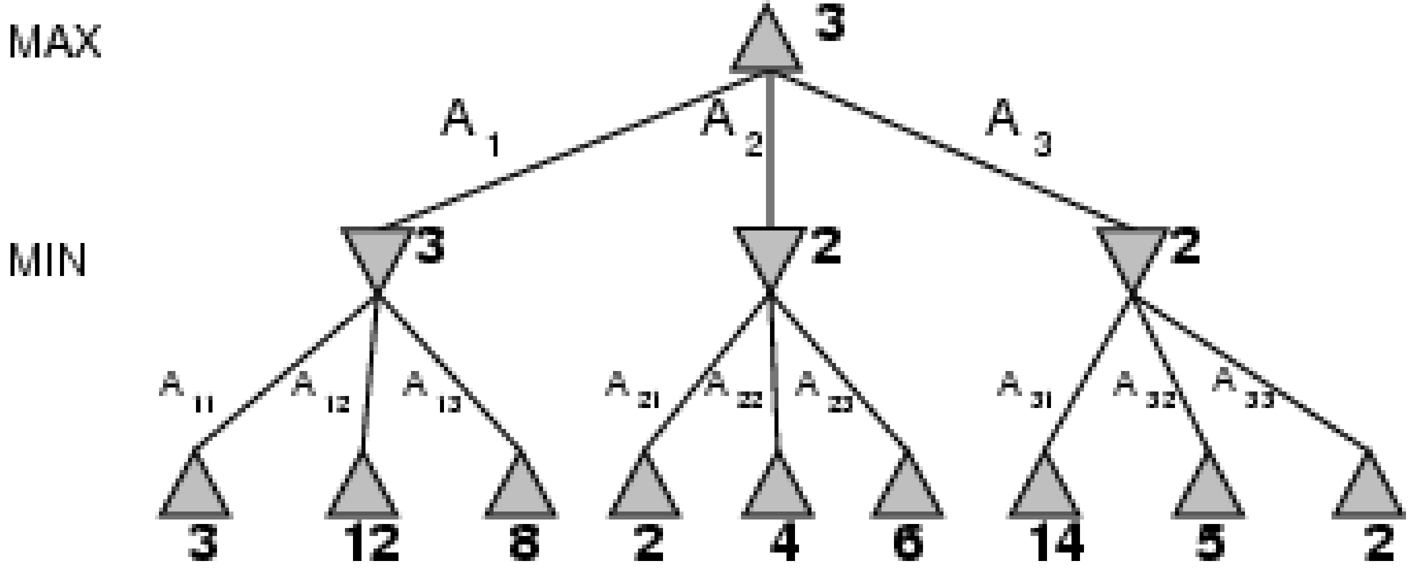
Minimax

- **Mini**mizing the **max**imum possible loss
- Choose move which results in best state
  - Select highest expected score for you
- Assume opponent is playing optimally too
  - Will choose lowest expected score for you

Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest **minimax value**

  = best achievable payoff against best play
- 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
```

Naïve Approach

- Any potential problems?

Properties of Minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
  - "Don't you know, there are some things that can beat smartness and foresight? Awkwardness and stupidity can. The best swordsman in the world doesn't need to fear the second best swordsman in the world; no, the person for him to be afraid of is some ignorant antagonist who has never had a sword in his hand before; he doesn't do the thing he ought to." - Mark Twain*, A Connecticut Yankee in King Arthur's Court*, (1889)
- 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

Resource Limits

Suppose we have 100 seconds, explore 10^4 nodes/sec
→ **10^6** nodes per move

Standard approach:

- **Cutoff test**:
  e.g., depth limit (perhaps add **quiescence search**)
- **Evaluation function**
  = estimated desirability of position

Cutting Off Search

- How to score a game before it ends?
  - You have to fudge it!
- Use a **heuristic** function to approximate state's utility

Cutting Off search

MinimaxCutoff is identical to *MinimaxValue* except

1. *Terminal?* is replaced by *Cutoff?*
2. *Utility* is replaced by *Eval*

Does it work in practice?

$b^m = 10^6$, b=35 → m=4

4-ply lookahead is a hopeless chess player!

- 4-ply ≈ human novice
- 8-ply ≈ typical PC, human master
- 12-ply ≈ Deep Blue, Kasparov

(A computer program which evaluates no further than its own legal moves plus the legal responses to those moves is searching to a depth of two-ply.)

Example Evaluation Function

- For chess, typically linear weighted sum of features

  $$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g., $w_1 = 9$ with

  $f_1(s)$ = (number of white queens) – (number of black queens), etc.

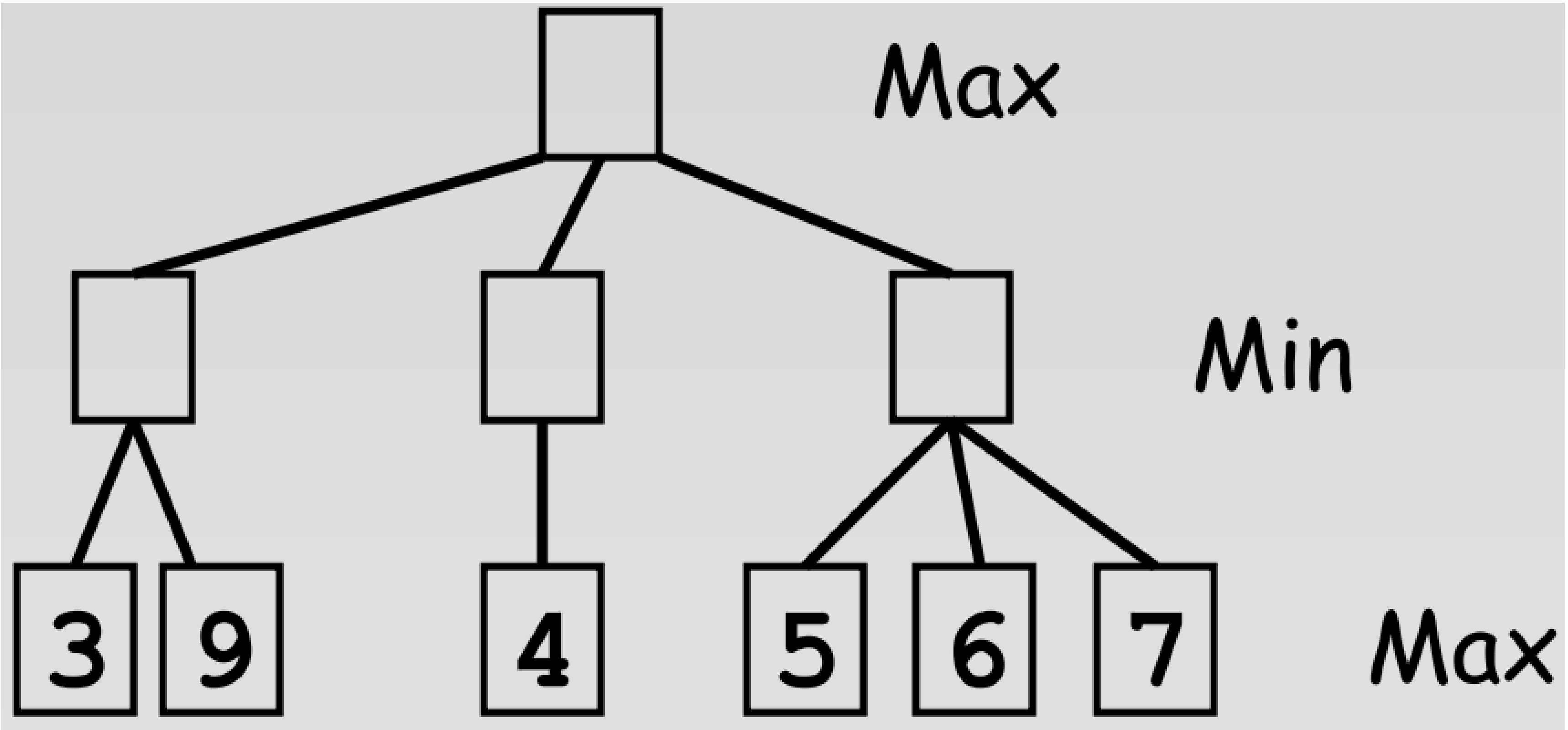
Evaluating States

- Assuming an ideal evaluation function, how would you make a move?
- Is this a good strategy with a bad function?

Look Ahead

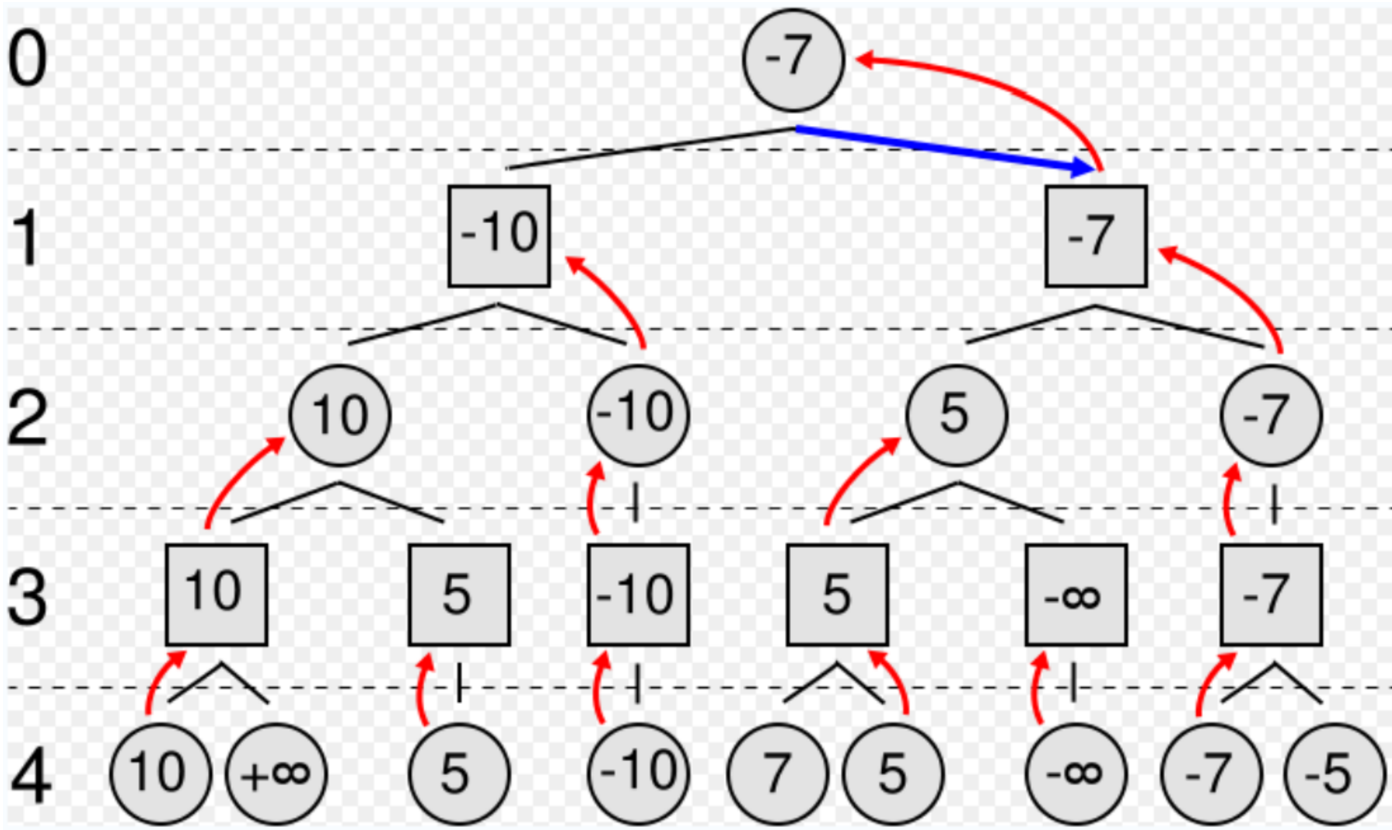
- Instead of only evaluating immediate future, look as far ahead as possible

Look Ahead



Bubbling Up

- Looking ahead allows utility values to "bubble up" to root of search tree



Minimax Algorithm

- BESTMOVE function
- Inputs:
  - Board state
  - Depth bound
- Explores search tree to specified depth
- Output:
  - Best move

Minimax Algorithm

```
BESTMOVE = proc (posn, depth)
  begin  (movelist, bestscore, bestm, try, tryscore) %local variables

         % posn: is the current BOARD CONFIGURATION FROM WHICH A MOVE
         %       MUST BE CHOSEN BY OUR INTELLIGENT AGENT COMPUTER

         % depth: MAXIMUM NUMBER OF PLIES TO LOOK AHEAD. THIS IS DETERMINED
         %        BY SPEED CONSTRAINTS AND COMPLEXITY OF THE GAME
         %        (i.e. branching factor b)

         % movelist: the list of all possible MOVES from the current posn
         % bestscore: the best "backed up" score found so far as we iterate
         %            thru the movelist
         % bestm:     the best move found so far that results in the bestscore
         % try:       just a tmp to hold returned values from recursive calls
         % tryscore:  just a tmp to hold returned scores from recursive calls
```

Minimax Algorithm

```
if depth = 0 then return list(EVALUATE(posn), nil) fi;
 %This ends the recursion at the bottom of the search space
 %Note a two element list of values is returned.

Movelist = POSSIBLE-MOVES(posn);
 %According to the rules of the game, we generate all possible
 %moves from the given board position.

Bestscore = -infinity;  %initializations
Bestm = nil;
```

Minimax Algorithm

```
%We are now ready to scan the movelist and select the
%best move. Note how we initialized Bestscore and Bestm.

while ( Movelist <> nil )
 repeat,
   try = BESTMOVE( NEWPOSITION(posn, first(Movelist)), depth-1);
         %Here is the main recursive call that expands and searches
         %the state space from the selected move.

   tryscore = - first(try); %recall BESTMOVE returns two values

   %Now we determine how well this current move did and whether
   %it should be selected as our best move found so far.

   if tryscore > Bestscore then
    do,
     Bestscore = tryscore;
     Bestm = first(Movelist);
    od;

   %Now we continue scanning down the list of moves to see
   %if we can find a better move then found so far.

   Movelist = rest(Movelist);
 taeper;
```

Minimax Algorithm

```
     %After scanning the entire Movelist, we have our best move so:
    return( list(Bestscore, Bestm) );
nigeb; %End of procedure min-max.
```

Minimax Algorithm

- Did you notice anything missing?

Minimax Algorithm

- Did you notice anything missing?
- Where were Max-Value and Min-Value?

Minimax Algorithm

- Did you notice anything missing?
- Where were Max-Value and Min-Value?
- What is going on here?

```
tryscore = - first(try); %recall BESTMOVE returns two values
```

Be Careful!

- Things to worry about?

Complexity

- What is the space complexity of depth-bounded Minimax?

Complexity

- What is the space complexity of depth-bounded Minimax?
  - Board size s
  - Depth d
  - Possible moves m

Complexity

- What is the space complexity of depth-bounded Minimax?
  – Board size s
  – Depth d
  – Possible moves m
- O(ds+m)
- Board positions can be released as bubble up

Minimax Algorithm

- Did I just do all your work for you?

Minimax Algorithm

- Did I just do all your work for you?
- No!

Minimax Algorithm

- Did I just do all your work for you?
- No!
- You need to create:
  - Evaluation function
  - Move generator
  - did_i_win? function

Recap

- What is a zero sum game?

Recap

- What is a zero sum game?
- What is a game tree?

Recap

- What is a zero sum game?
- What is a game tree?
- What is Minimax?

Recap

- What is a zero sum game?
- What is a game tree?
- What is Minimax?
  - Why is it called that?

Recap

- What is a zero sum game?
- What is a game tree?
- What is Minimax?
  - Why is it called that?
- What is its space complexity?

Recap

- What is a zero sum game?
- What is a game tree?
- What is Minimax?
  - Why is it called that?
- What is its space complexity?
- How can the Minimax algorithm be simplified?

Recap

- What is a zero sum game?
- What is a game tree?
- What is Minimax?
  - Why is it called that?
- What is its space complexity?
- How can the Minimax algorithm be simplified?
  - Will this work for all games?

Recap

- What is a zero sum game?
- What is a game tree?
- What is Minimax?
  - Why is it called that?
- What is its space complexity?
- How can the Minimax algorithm be simplified?
  - Will this work for all games?

Next Up

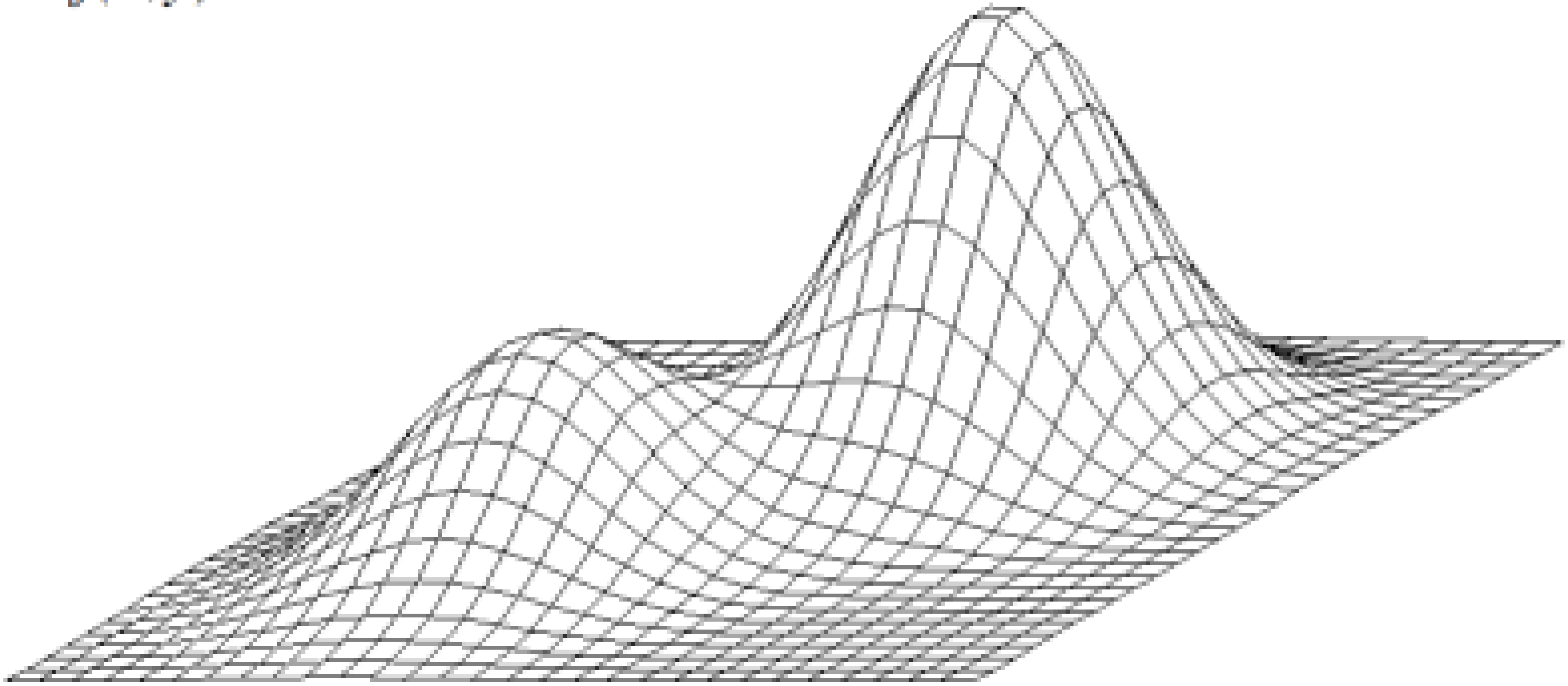
- Recall that minimax will produce optimal play against an optimal opponent if entire tree is searched
- Is the same true if a cutoff is used?

Horizon Effect

- Your algorithm searches to depth n
- What happens if:
  - Evaluation(s) at depth n is very positive
  - Evaluation(s) at depth n+1 is very negative
- Or:
  - Evaluation(s) at depth n is very negative
  - Evaluation(s) at depth n+1 is very positive
- Will this ever happen in practice?

Local Maxima Problem

$$f(x,y)=e^{-(x^2+y^2)} + 2e^{-((x-1.7)^2+(y-1.7)^2)}$$



Search Limitation Mitigation

- Sometimes it is useful to look deeper into game tree
- We could peak past the horizon…
- But how can you decide what nodes to explore?
  - Quiescence search

Quiescence Search

- Human players have some intuition about move quality
  - “Interesting vs “boring”
  - “Promising” vs “dead end”
  - “Noisy” vs “quiet”
- Expand horizon for potential high impact moves
- Quiescence search adds this to Minimax

Quiescence Search

- Additional search performed on leaf nodes
- if looks_interesting(leaf_node):

  extend_search_depth(leaf_node)

  else:

  normal_evaluation(leaf_node)

Quiescence Search

- What constitutes an “interesting” state?
  - Moves that substantially alter game state
  - Moves that cause large fluctuations in evaluation function output
- Chess example: capture moves
- Must be careful to prevent indefinite extension of search depth
  - Chess: checks vs captures

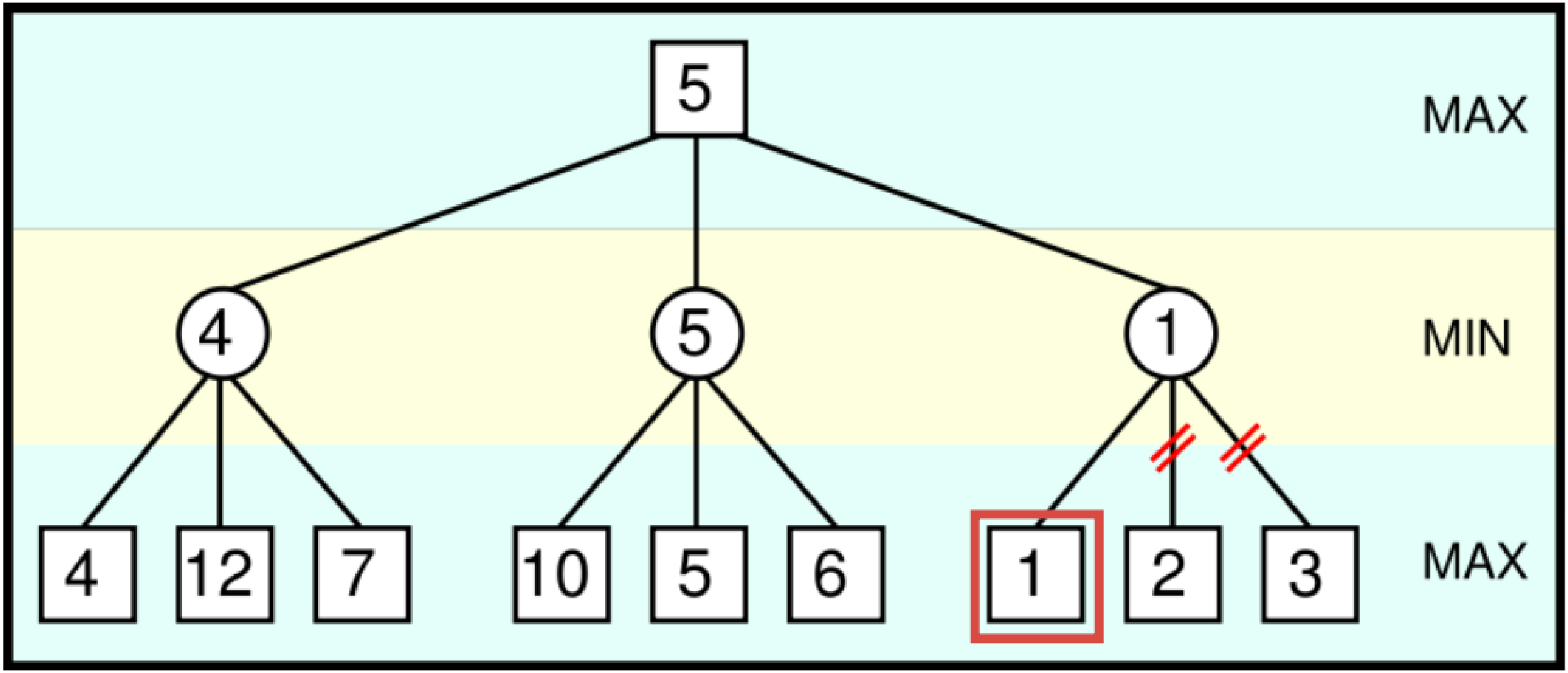
Search Limitation Mitigation

- Do you always need to search the entire tree?
  - No!
- Sometimes it is useful to look *less deeply* into tree
- But how can you decide what branches to ignore?
  - Tree pruning

Tree Pruning

- Moves chosen under assumption of optimal adversary
- You know the best move so far
- If you find a branch with a worse move, is there any point in looking further?
- Thought experiment: bag game

Pruning Example



Alpha-Beta Pruning

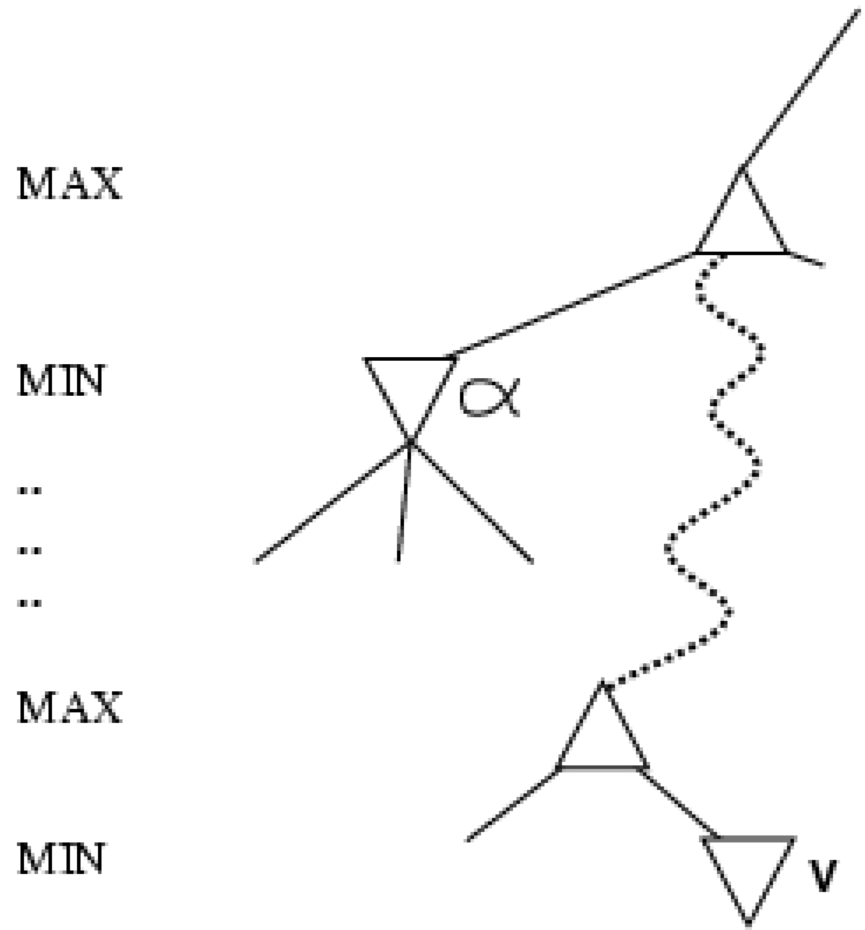
- During Minimax, keep track of two additional values
- Alpha
  - Your best score via any path
- Beta
  - Opponent’s best score via any path

Alpha-Beta Pruning

- Max player (you) will never make a move that could lead to a worse score for you
- Min player (opponent) will never make a move that could lead to a better score for you
- Stop evaluating a branch whenever:
  – A value greater than beta is found
  – A value less than alpha is found

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 it
  → prune that branch
- Define β similarly for *min*

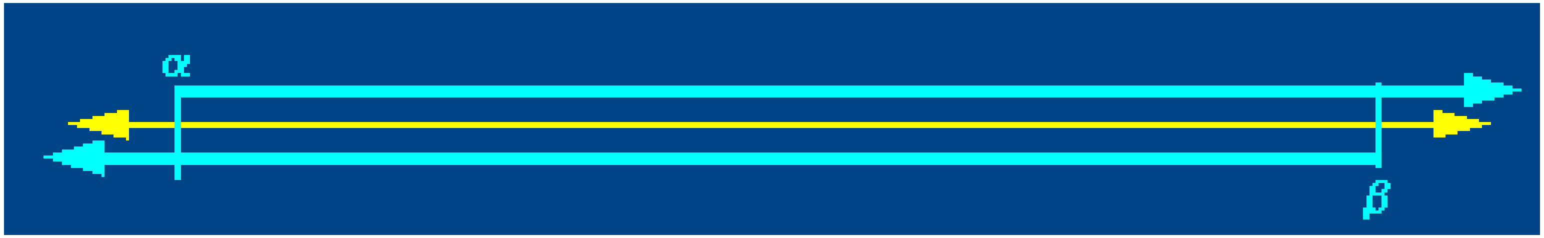


Alpha-Beta Pruning

- Based on observation that for all viable paths utility value n will be **α <= n <= β**

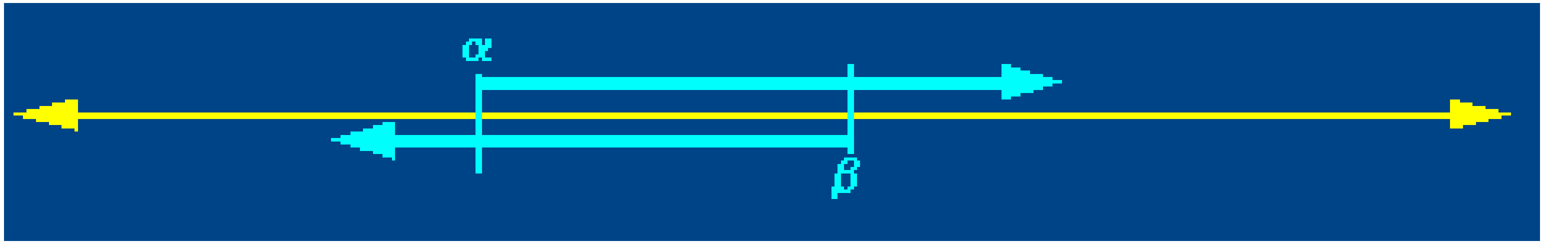
Alpha-Beta Pruning

- Initially, α = -infinity, β=infinity



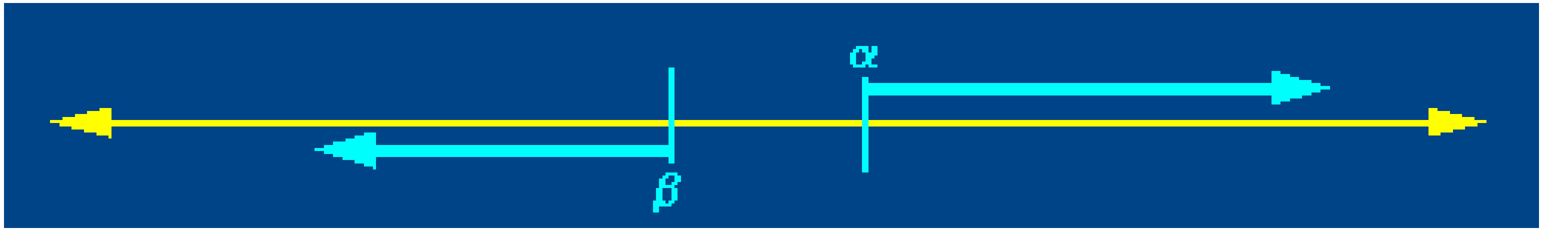
Alpha-Beta Pruning

- As the search tree is traversed, the possible utility value window shrinks as
  - Alpha increases
  - Beta decreases



Alpha-Beta Pruning

- Once there is no longer any overlap in the possible ranges of alpha and beta, it is safe to conclude that the current node is a dead end



Minimax Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*

    $v \leftarrow$ 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*)
    $v \leftarrow -\infty$
    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

The α-β Algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  inputs: *state*, current state in game

  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in SUCCESSORS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, 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**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

The α-β Algorithm

function MIN-VALUE(*state*, α, β) **returns** *a utility value*

inputs: *state*, current state in game

α, the value of the best alternative for MAX along the path to *state*

β, 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**

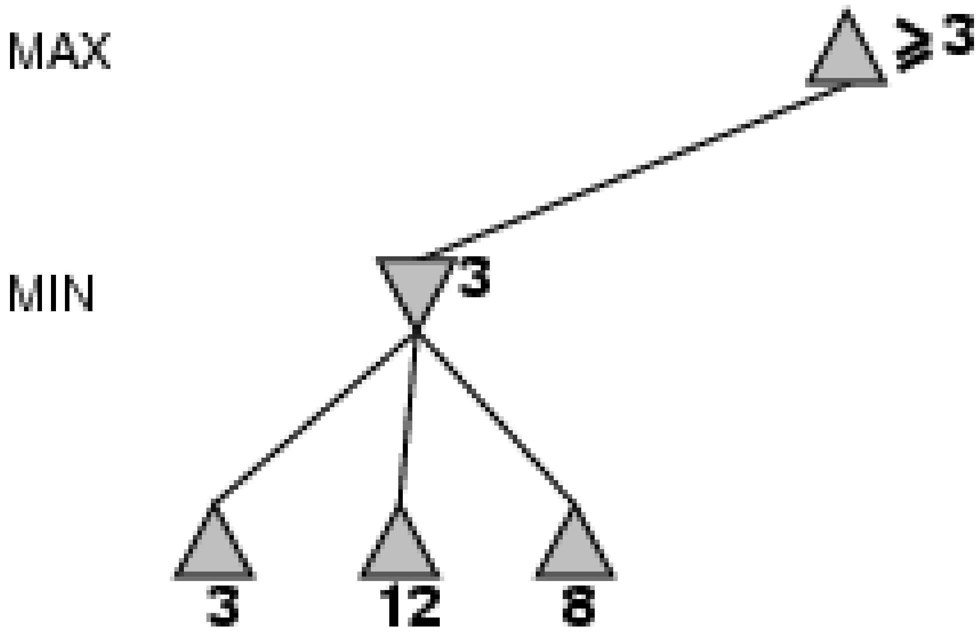
$v \leftarrow$ MIN(v, MAX-VALUE(s, α, β))

if $v \leq \alpha$ **then return** v

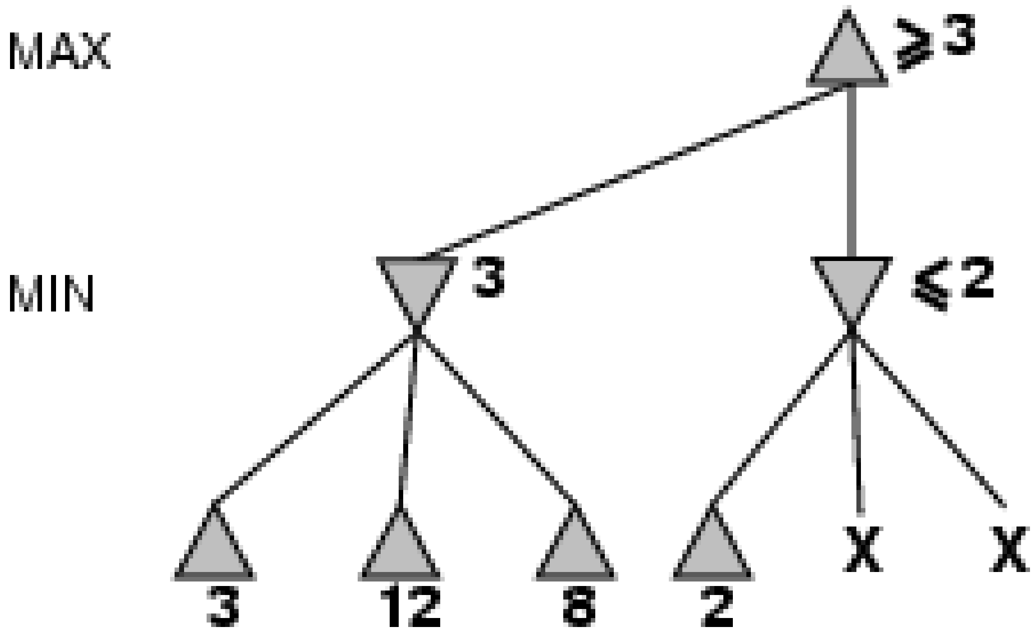
$\beta \leftarrow$ MIN(β, v)

return v

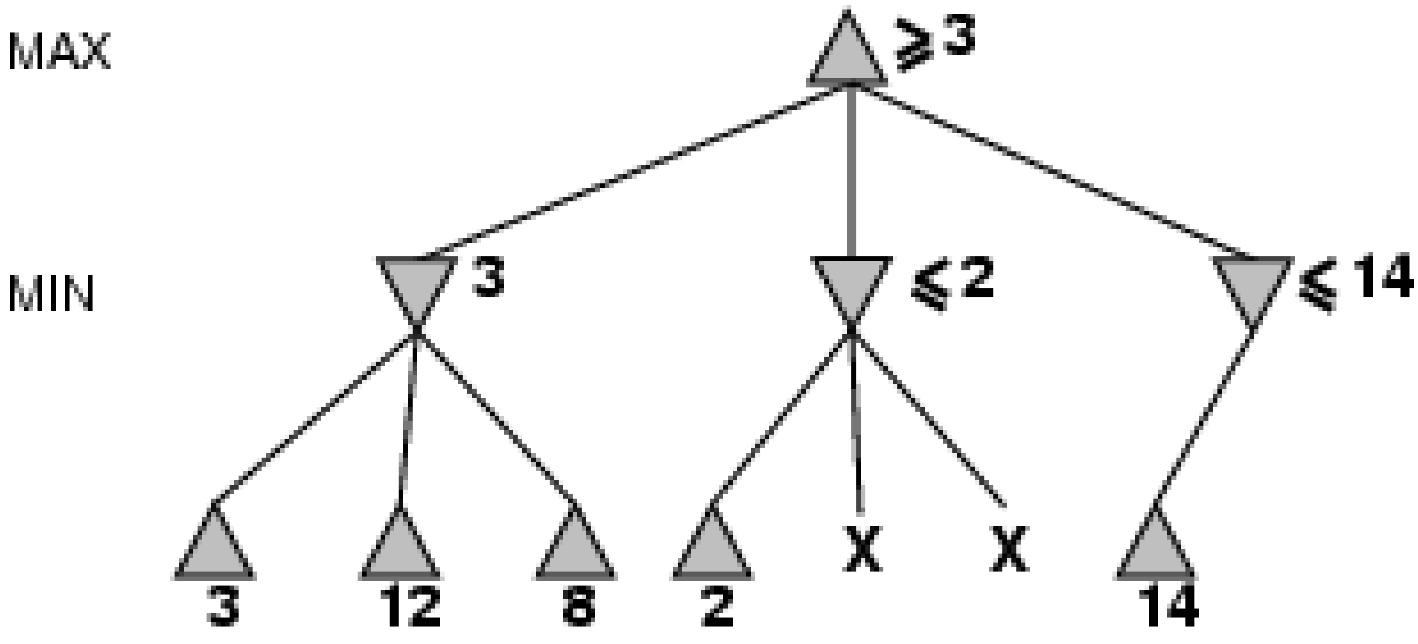
α-β Pruning Example



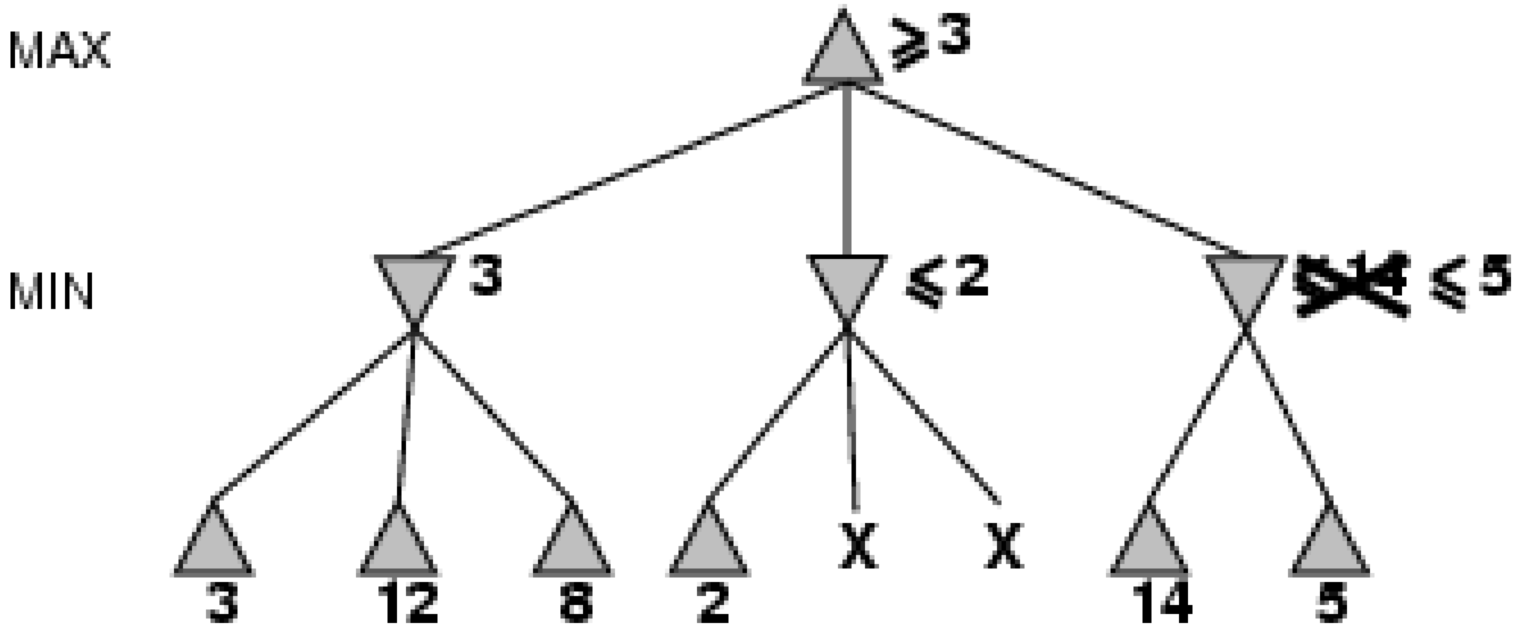
α-β Pruning Example



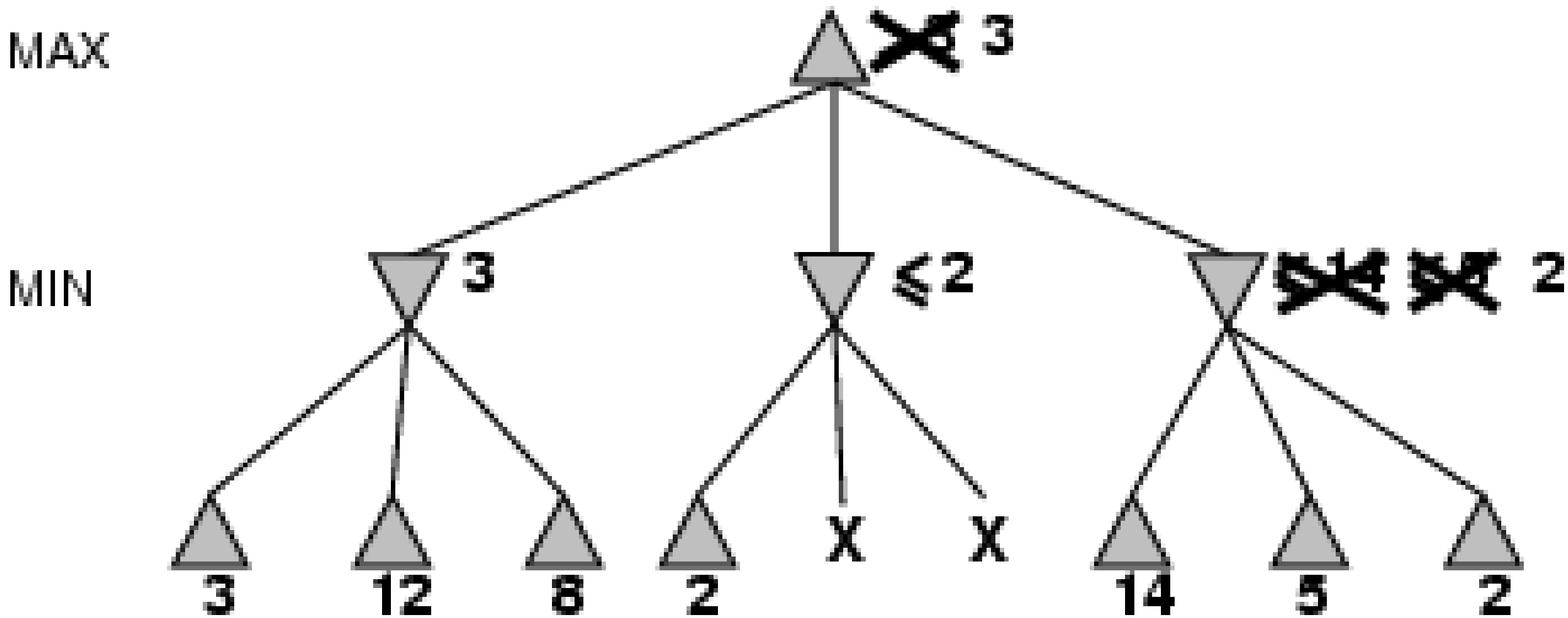
α-β Pruning Example



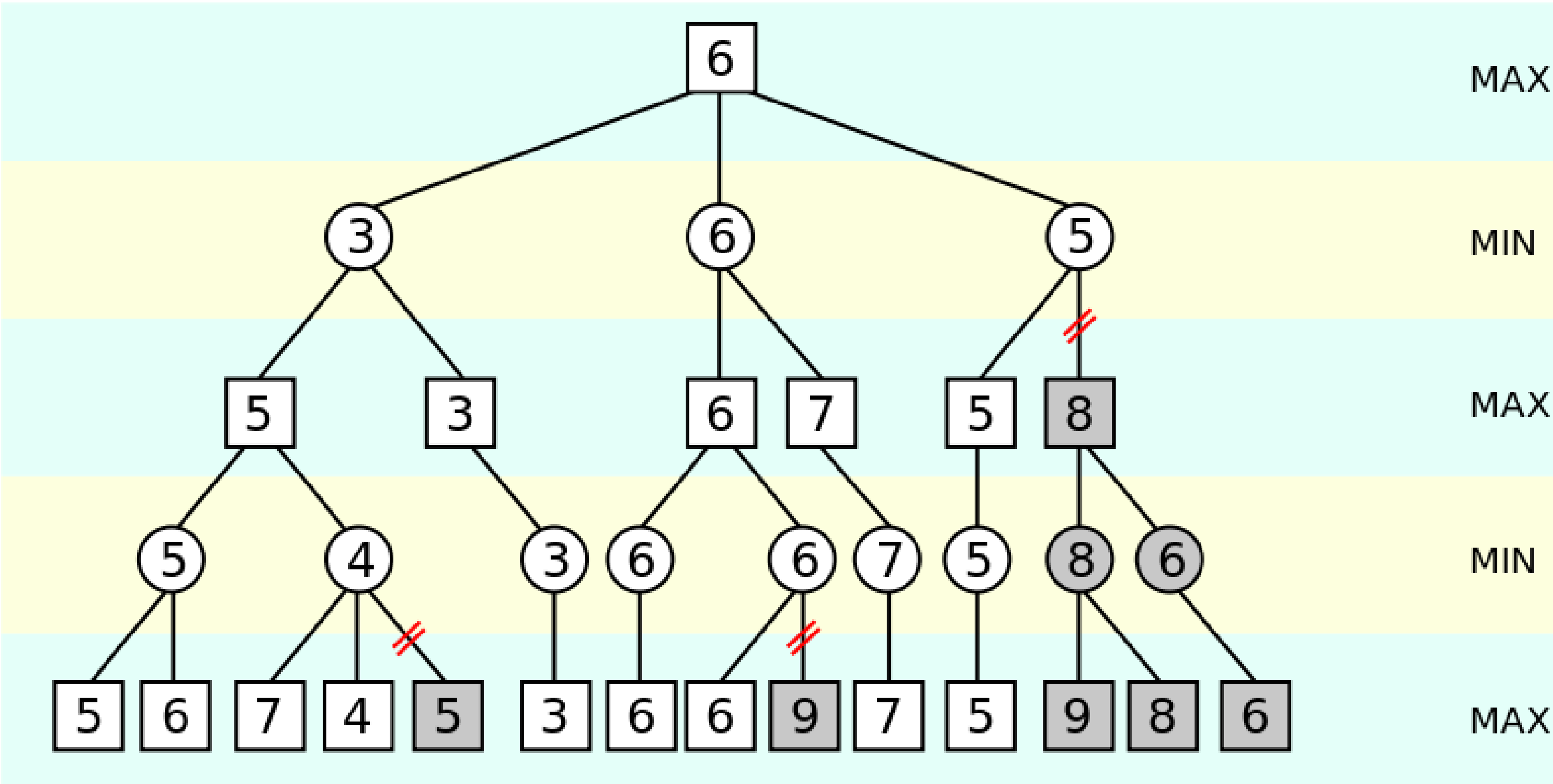
α-β Pruning Example



α-β Pruning Example



Another α-β Pruning Example



Minimax Psuedocode

```
BESTMOVE = proc (posn, depth)
  begin  (movelist, bestscore, bestm, try, tryscore) %local variables

         % posn: is the current BOARD CONFIGURATION FROM WHICH A MOVE
         %       MUST BE CHOSEN BY OUR INTELLIGENT AGENT COMPUTER

         % depth: MAXIMUM NUMBER OF PLIES TO LOOK AHEAD. THIS IS DETERMINED
         %       BY SPEED CONSTRAINTS AND COMPLEXITY OF THE GAME
         %       (i.e. branching factor b)

         % movelist: the list of all possible MOVES from the current posn
         % bestscore: the best "backed up" score found so far as we iterate
         %            thru the movelist
         % bestm:     the best move found so far that results in the bestscore
         % try:       just a tmp to hold returned values from recursive calls
         % tryscore:  just a tmp to hold returned scores from recursive calls
```

Alpha-Beta Psuedocode

```
BESTMOVE = proc (Posn, Depth, Mybest, Herbest)
         %Note we added two additional parameters. Mybest should be
         %initialized to -infinity, while herbest is initialized to
         %+infinity when calling this version of BESTMOVE.

  begin  (Movelist, Bestscore, Bestm, Try, Tryscore) %local variables
```

Minimax Psuedocode

```
if depth = 0 then return list(EVALUATE(posn), nil) fi;
 %This ends the recursion at the bottom of the search space
 %Note a two element list of values is returned.

Movelist = POSSIBLE-MOVES(posn);
 %According to the rules of the game, we generate all possible
 %moves from the given board position.

Bestscore = -infinity;  %initializations
Bestm = nil;
```

Alpha-Beta Psuedocode

```
if Depth = 0 then return list(EVALUATE(Posn), nil) fi;
 %This ends the recursion at the bottom of the search space
 %Note a two element list of values is returned.

Movelist = POSSIBLE-MOVES(Posn);
 %According to the rules of the game, we generate all possible
 %moves from the given board position.

Bestscore = Mybest  % note the change to initializations here
Bestm = nil;
```

Minimax Psuedocode

```
%We are now ready to scan the movelist and select the
%best move. Note how we initialized Bestscore and Bestm.

while ( Movelist <> nil )
 repeat,
   try = BESTMOVE( NEWPOSITION(posn, first(Movelist)), depth-1);
         %Here is the main recursive call that expands and searches
         %the state space from the selected move.

   tryscore = - first(try); %recall BESTMOVE returns two values

   %Now we determine how well this current move did and whether
   %it should be selected as our best move found so far.

   if tryscore > Bestscore then
    do,
     Bestscore = tryscore;
     Bestm = first(Movelist);
    od;

   %Now we continue scanning down the list of moves to see
   %if we can find a better move then found so far.

   Movelist = rest(Movelist);
 taeper;
```

Alpha-Beta Psuedocode

```
%We are now ready to scan the movelist and select the
%best move. Note how we initialized Bestscore and Bestm.

while ( Movelist <> nil )
 repeat,
   Try =
    BESTMOVE( NEWPOSITION(posn, first(Movelist)),
              Depth-1,
              -Herbest,
              -Bestscore );
         %Here is the main recursive call that expands and searches
         %the state space from the selected move. But notice we
         %added the two additional parameters here. This makes
         %sense when we view the following conditional expressions.

   Tryscore = - first(Try); %recall BESTMOVE returns two values

   %Now we determine how well this current move did and whether
   %it should be selected as our best move found so far.

   if Tryscore > Bestscore then
    do,
     Bestscore = Tryscore;
     Bestm = first(Movelist);
    od;
```

Minimax Algorithm

```
      %After scanning the entire Movelist, we have our best move so:
    return( list(Bestscore, Bestm) );
nigeb; %End of procedure min-max.
```

Alpha-Beta Psuedocode

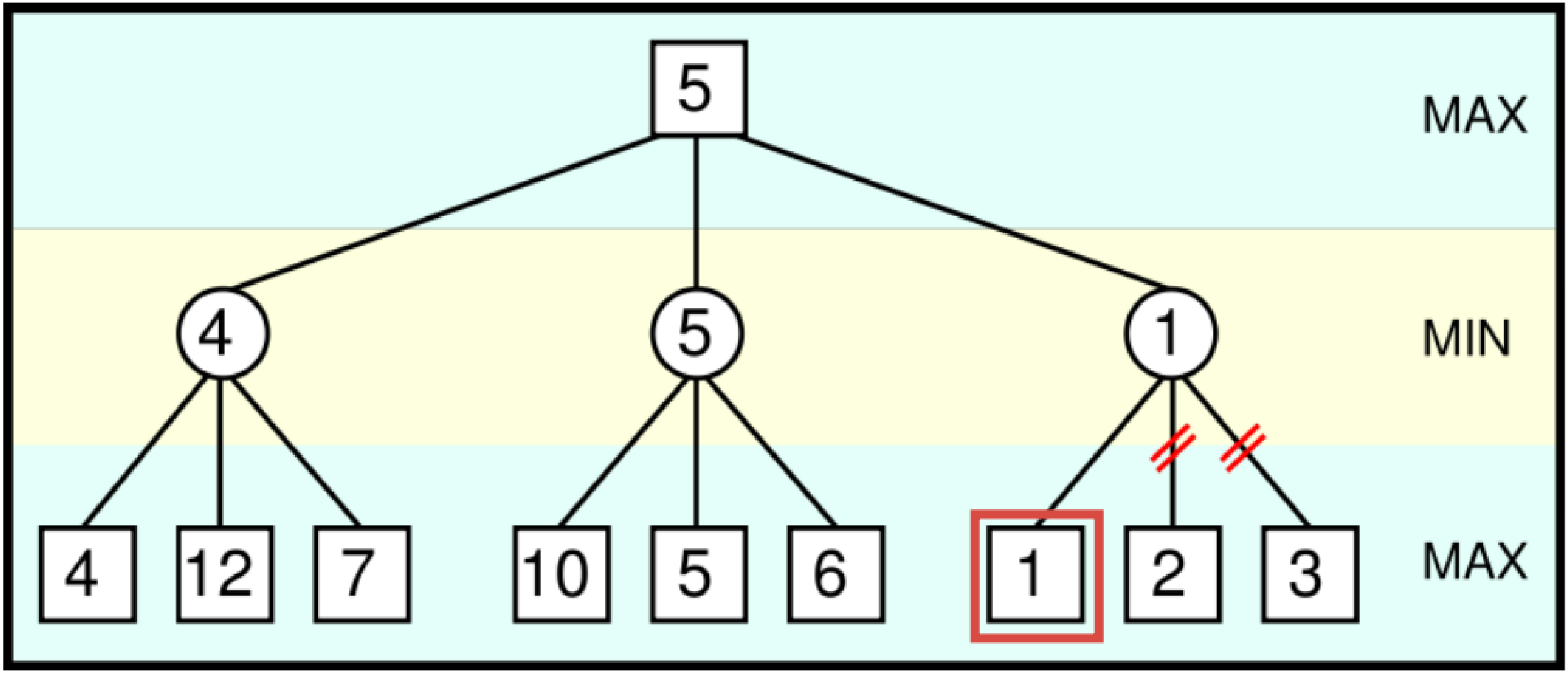
```
        if Bestscore > Herbest then return ( list(Bestscore, Bestm));

        %Now we continue scanning down the list of moves to see
        %if we can find a better move then found so far.

        Movelist = rest(Movelist);
      taeper;

     %After scanning the entire Movelist, we have our best move so:

    return( list(Bestscore, Bestm) );

nigeb; %End of procedure alpha-beta.
```

Tree Pruning vs Heuristics

- Search depth cut off may affect outcome of algorithm
- How about pruning?

Move Ordering

- Does the order in which moves are listed have any impact of alpha-beta?



Move Ordering

- Techniques for improving move ordering
- Apply evaluation function to nodes prior to expanding children
  - Search in descending order
  - But sacrifices search depth
- Cache results of previous algorithm

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 *metareasoning*)

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

- Othello: human champions refuse to compete against computers, who are too good.

Deterministic Games in Practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

- Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

- Othello: human champions refuse to compete against computers, who are too good.

- Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Summary

- Games are fun to work on!
- They illustrate several important points about AI
- Perfection is unattainable
  - Must approximate
- Good idea to think about what to think about