CMSC 471
Fall 2012

Class #8

Tue 9/25/12
Game Playing

Kevin Winner, winnerk1@umbc.edu

Game Playing

Chapter 5

Some material adopted from notes by Charles R. Dyer, University of Wisconsin-Madison

Today's class

- Homework 2
- Game playing
  - State of the art and resources
  - Framework
- Game trees
  - Minimax
  - Alpha-beta pruning
  - Stochastic games
- Homework 1
- Homework 2 design

Why study games?

- Offer an opportunity to study interesting environments which are still easy to understand
  - Multiagent, adversarial
  - Stochastic
  - Partially observable
- Clear criteria for success
- Games often define very large search spaces
  - Chinook (checkers) has a dictionary of 39 trillion end-game states
- Fun

State of the art

- How good are computer game players?
  - **Chess**:
    - Deep Blue beat Gary Kasparov in 1997
  - **Checkers**: Chinook (an AI program with a *very large* endgame database) has solved checkers
  - **Go**: Computer players have finally reached tournament-level play
  - **Backgammon**: TD-Gammon trained on itself to reach the ability to beat top-level players
  - **Poker**: After chess, probably the most studied game for AI players

Typical case

- 2-person game
- Players alternate moves
- **Zero-sum**: one player's loss is the other's gain
- **Perfect information**: both players have access to complete information about the state of the game. No information is hidden from either player.
- **Deterministic**: no chance (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- Not: Bridge, Solitaire, Backgammon, ...

How to play a game

- A way to play such a game is to:
  - Consider all the legal moves you can make
  - Compute the new position resulting from each move
  - Evaluate each resulting position and determine which is best
  - Make that move
  - Wait for your opponent to move and repeat
- Key problems are:
  - Representing the "board"
  - Generating all legal next boards
  - Evaluating a position

Evaluation function

- **Evaluation function** or **static evaluator** is used to evaluate the “goodness” of a game position.
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node
- The zero-sum assumption allows us to use a single evaluation function to describe the goodness of a board with respect to both players.
  - **f(n) >> 0**: position n good for me and bad for you
  - **f(n) << 0**: position n bad for me and good for you
  - **f(n) near 0**: position n is a neutral position
  - **f(n) = +infinity**: win for me
  - **f(n) = -infinity**: win for you

Evaluation function examples

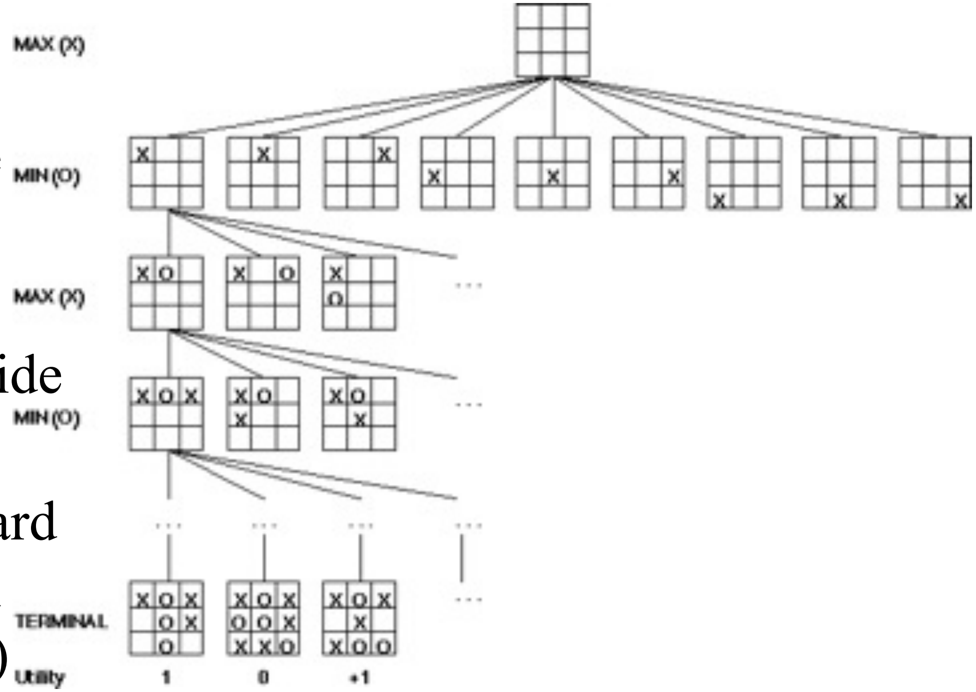
- Example of an evaluation function for Tic-Tac-Toe:

  f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you]

  where a 3-length is a complete row, column, or diagonal
- Alan Turing's function for chess
  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white's pieces and b(n) = sum of black's
- Most evaluation functions are specified as a weighted sum of position features:

  $f(n) = w_1 * feat_1(n) + w_2 * feat_2(n) + ... + w_n * feat_k(n)$
- Example features for chess are piece count, piece placement, squares controlled, etc.
- Deep Blue had over 8000 features in its evaluation function

Game trees

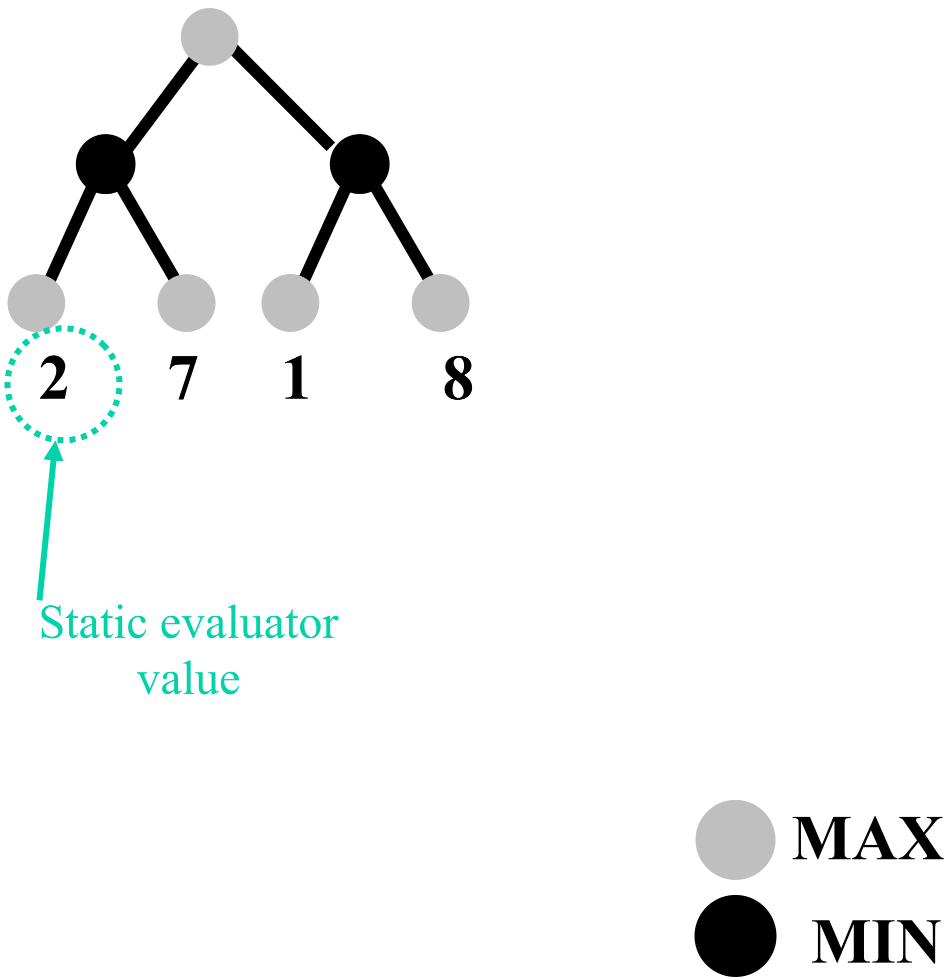


- Problem spaces for typical games are represented as trees
- Root node represents the current board configuration; player must decide the best single move to make next
- **Static evaluator function** rates a board position. f(board) = real number with f>0 “white” (me), f<0 for black (you)
- Arcs represent the possible legal moves for a player
- If it is **my turn** to move, then the root is labeled a "**MAX**" node; otherwise it is labeled a "**MIN**" node, indicating **my opponent's turn**.
- Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1

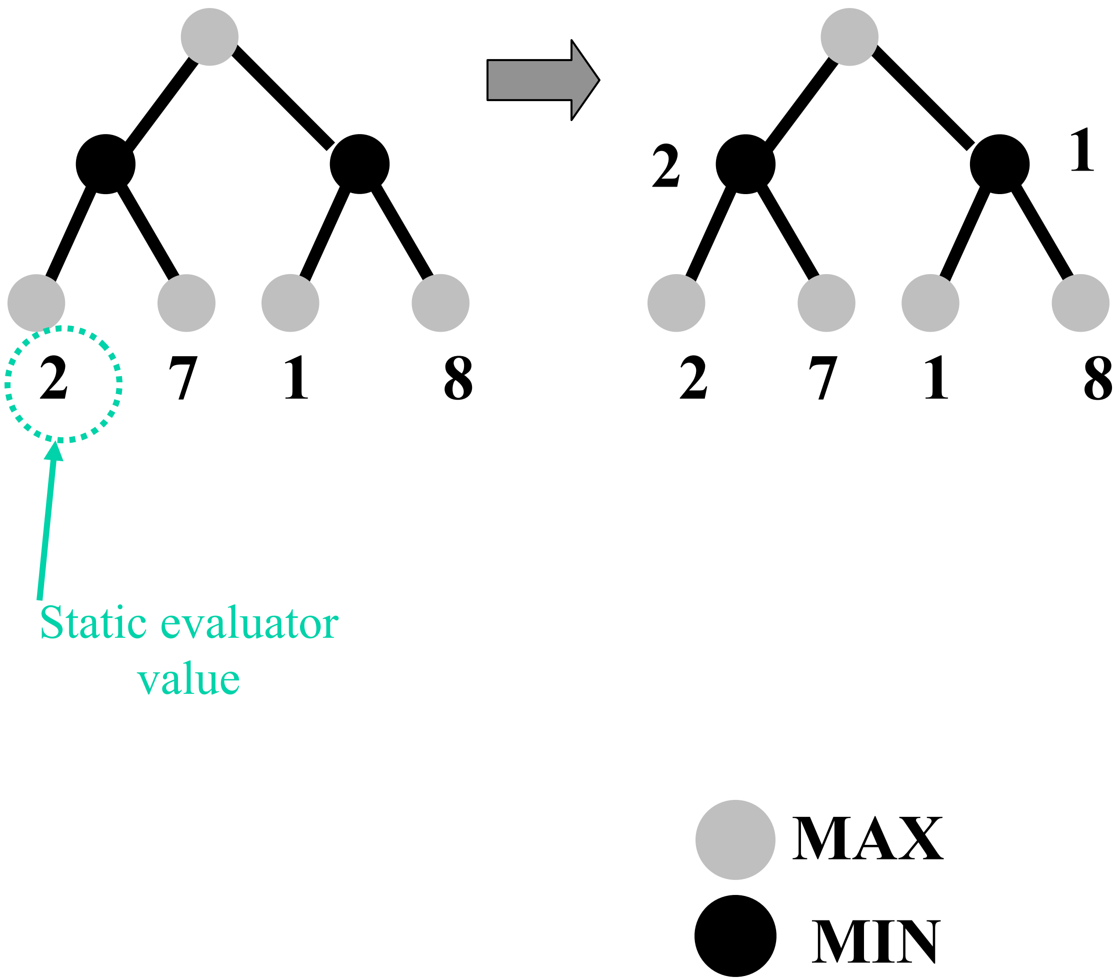
Minimax procedure

- Create start node as a MAX node with current board configuration
- Expand nodes down to some **depth** (a.k.a. **ply**) of lookahead in the game
- Apply the evaluation function at each of the leaf nodes
- "Back up" values for each of the non-leaf nodes until a value is computed for the root node
  - At MIN nodes, the backed-up value is the **minimum** of the values associated with its children.
  - At MAX nodes, the backed-up value is the **maximum** of the values associated with its children.
- Pick the operator associated with the child node whose backed-up value determined the value at the root

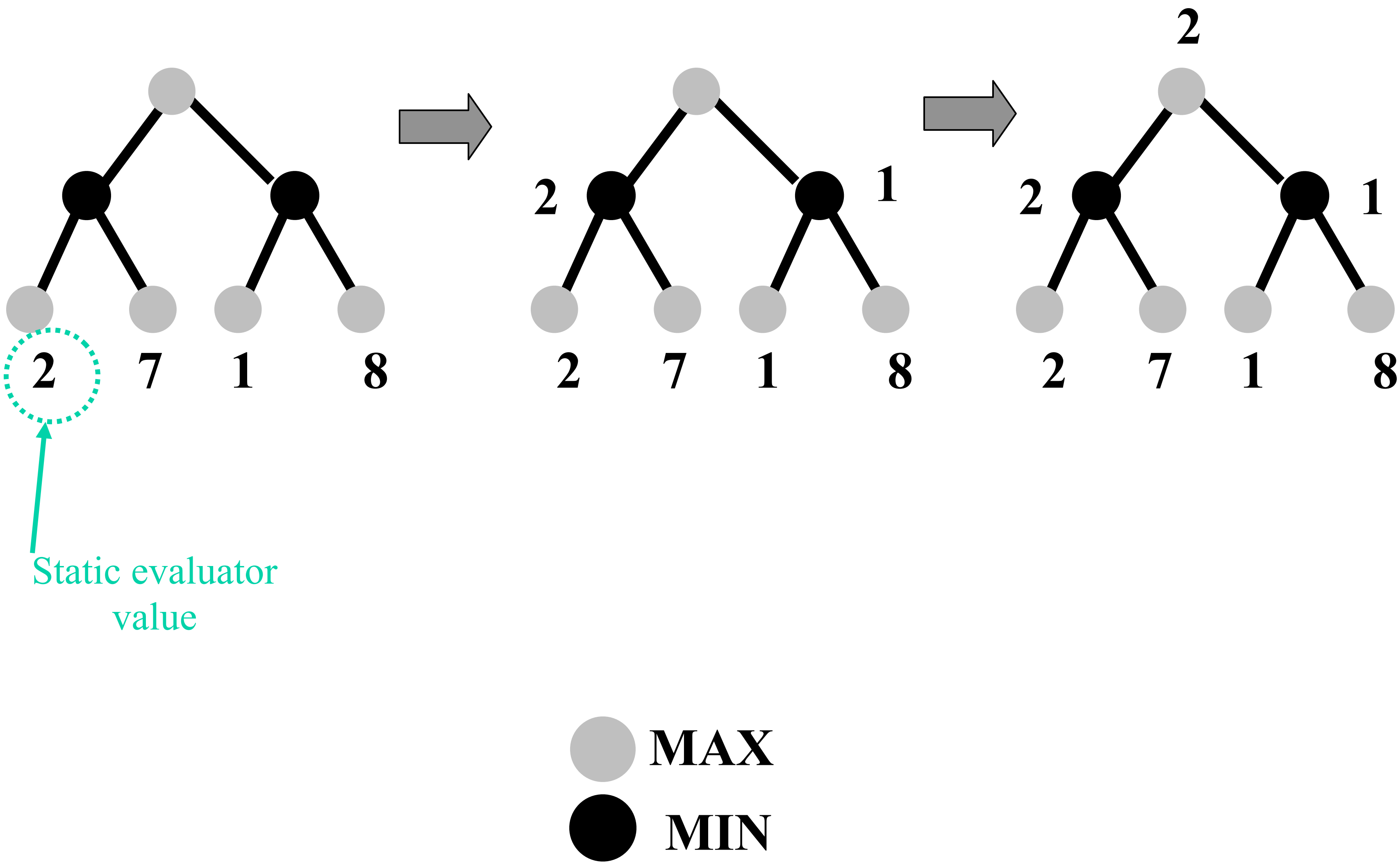
Minimax Algorithm



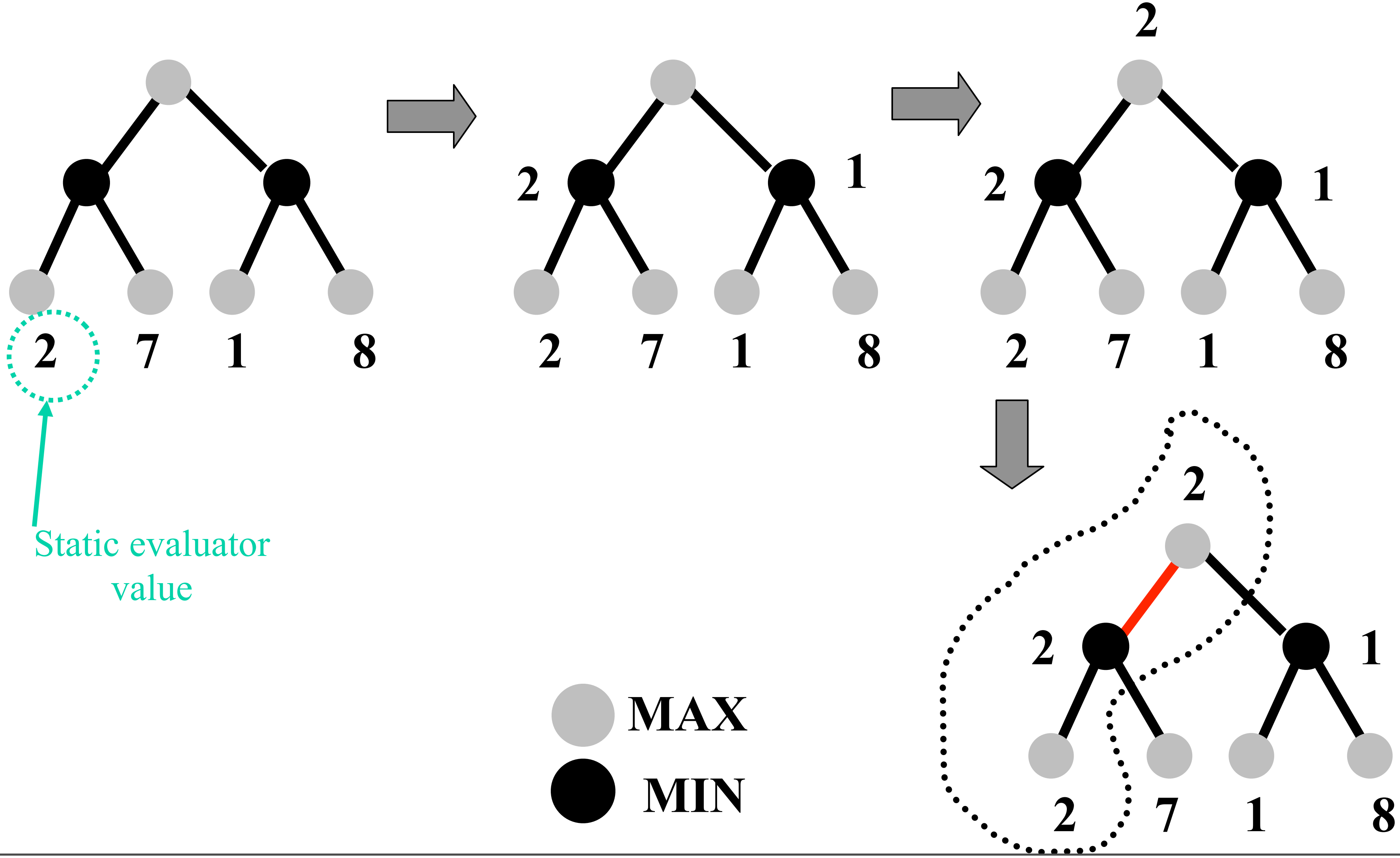
Minimax Algorithm



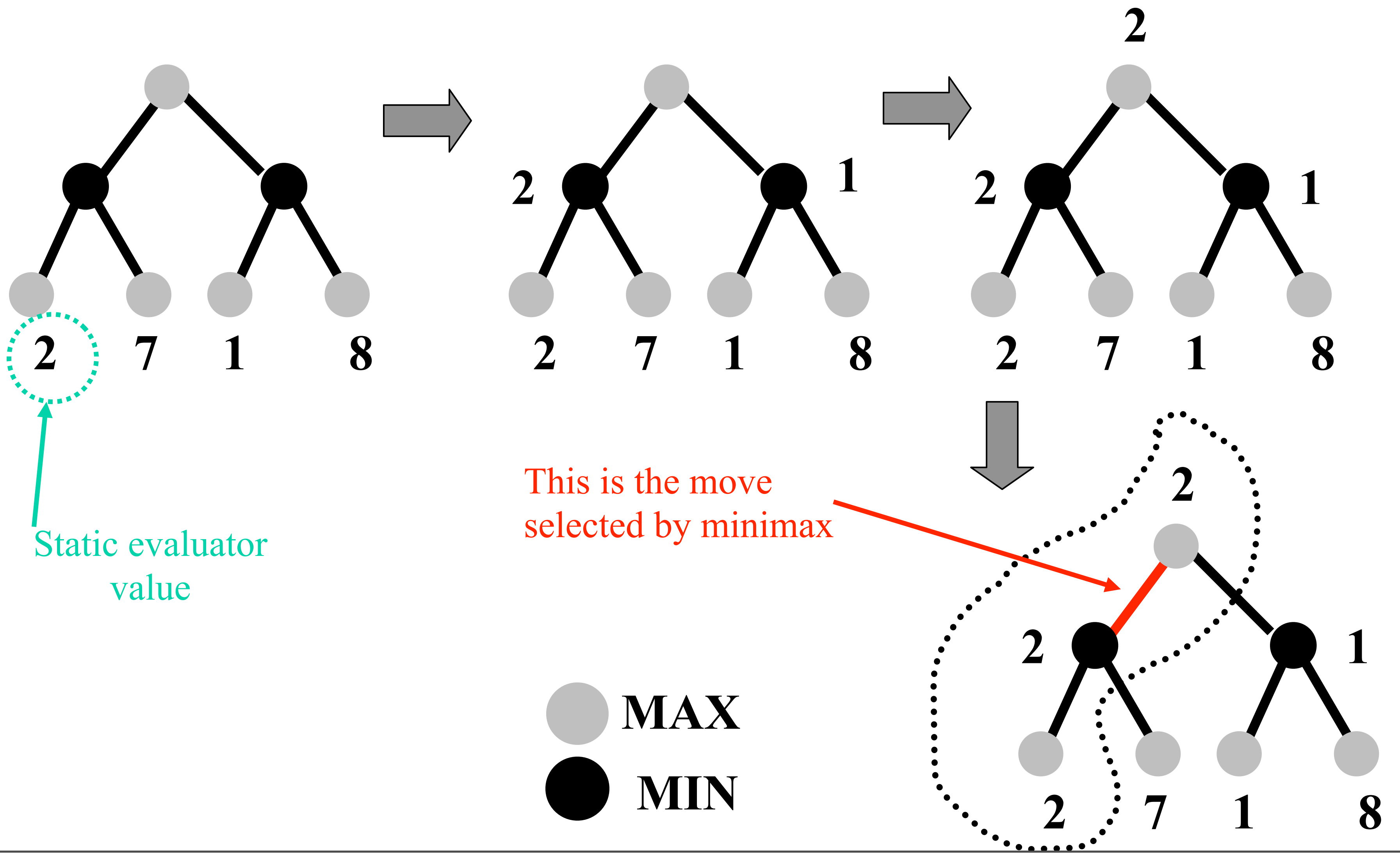
Minimax Algorithm



Minimax Algorithm

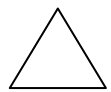


Minimax Algorithm



Minimax Tree

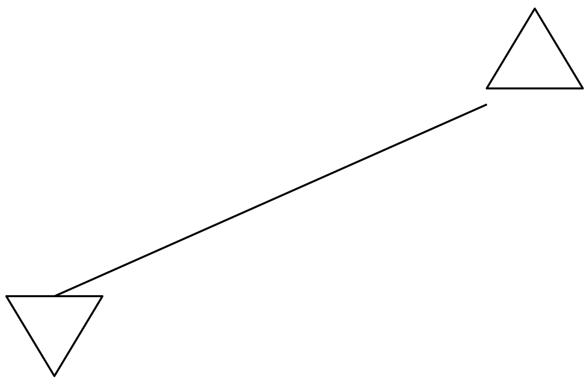
MAX



MIN

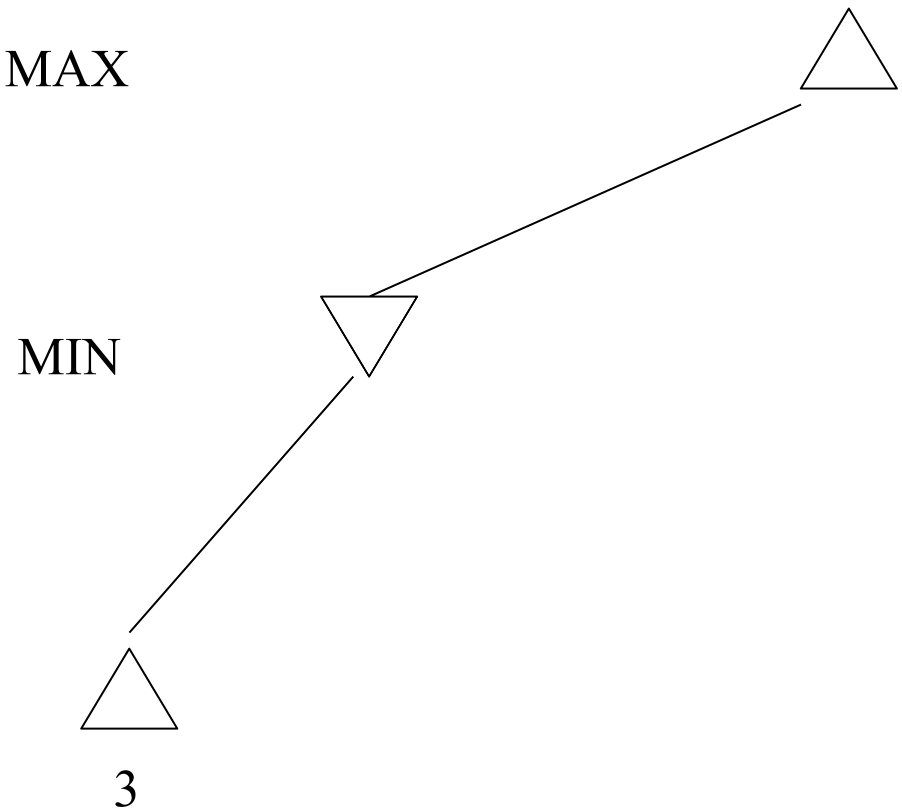
Minimax Tree

MAX

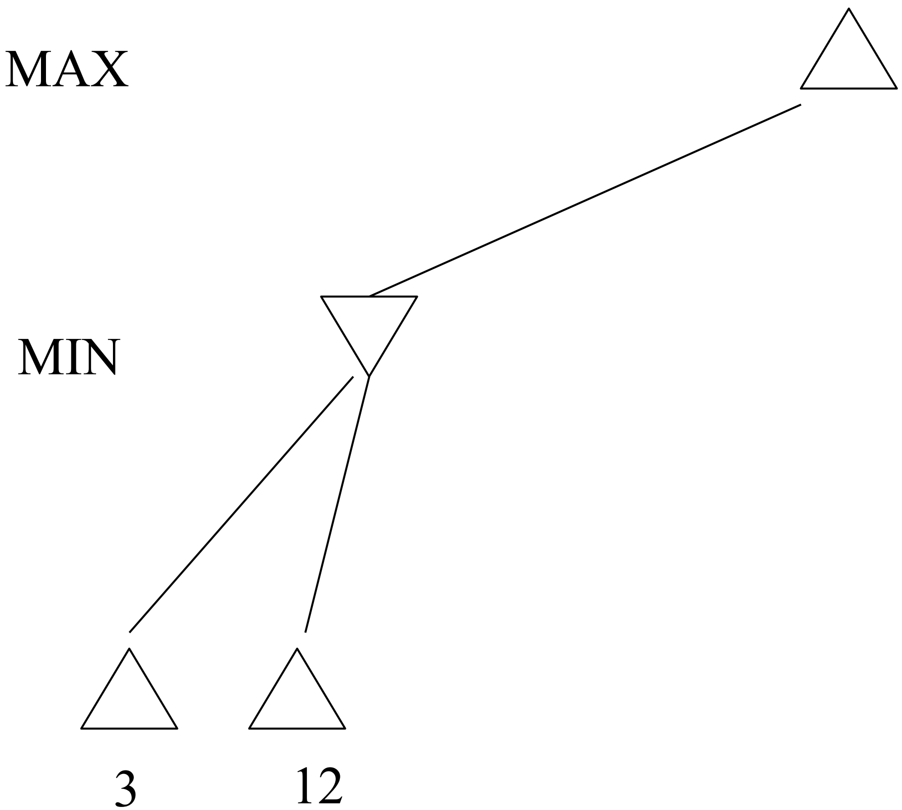


MIN

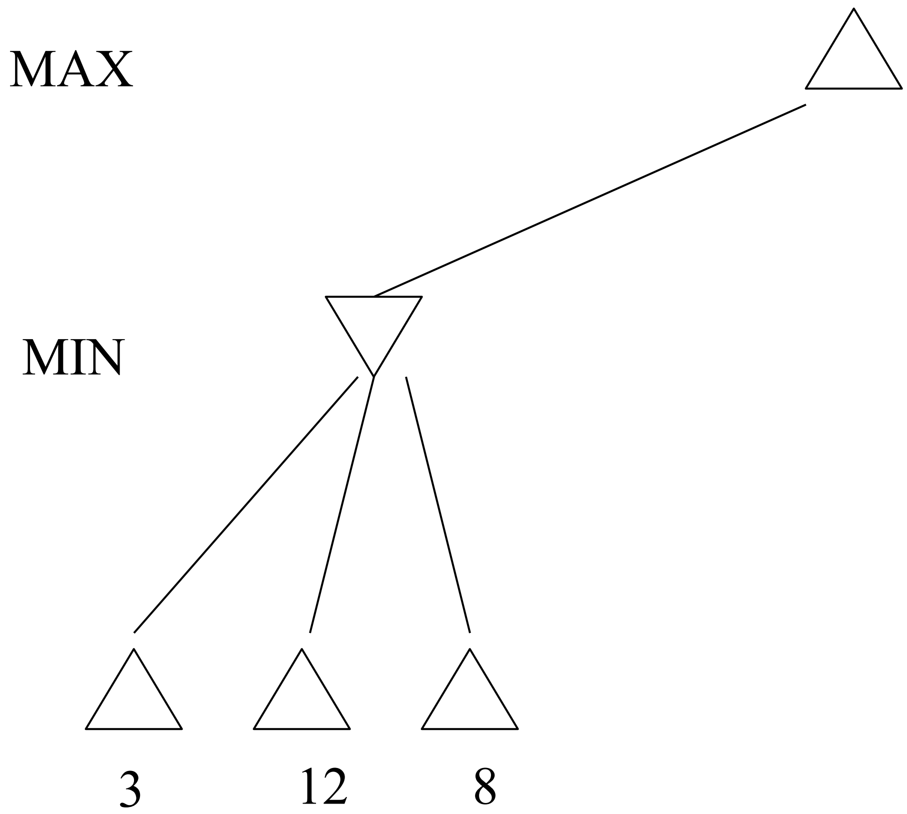
Minimax Tree



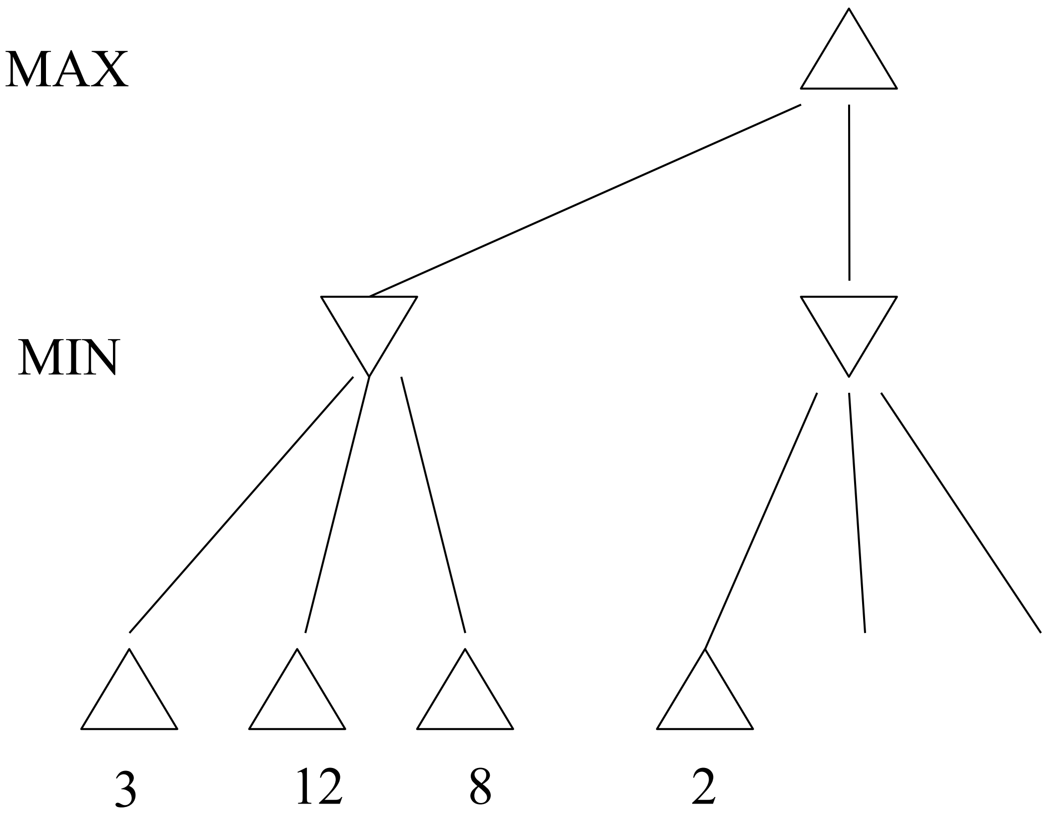
Minimax Tree



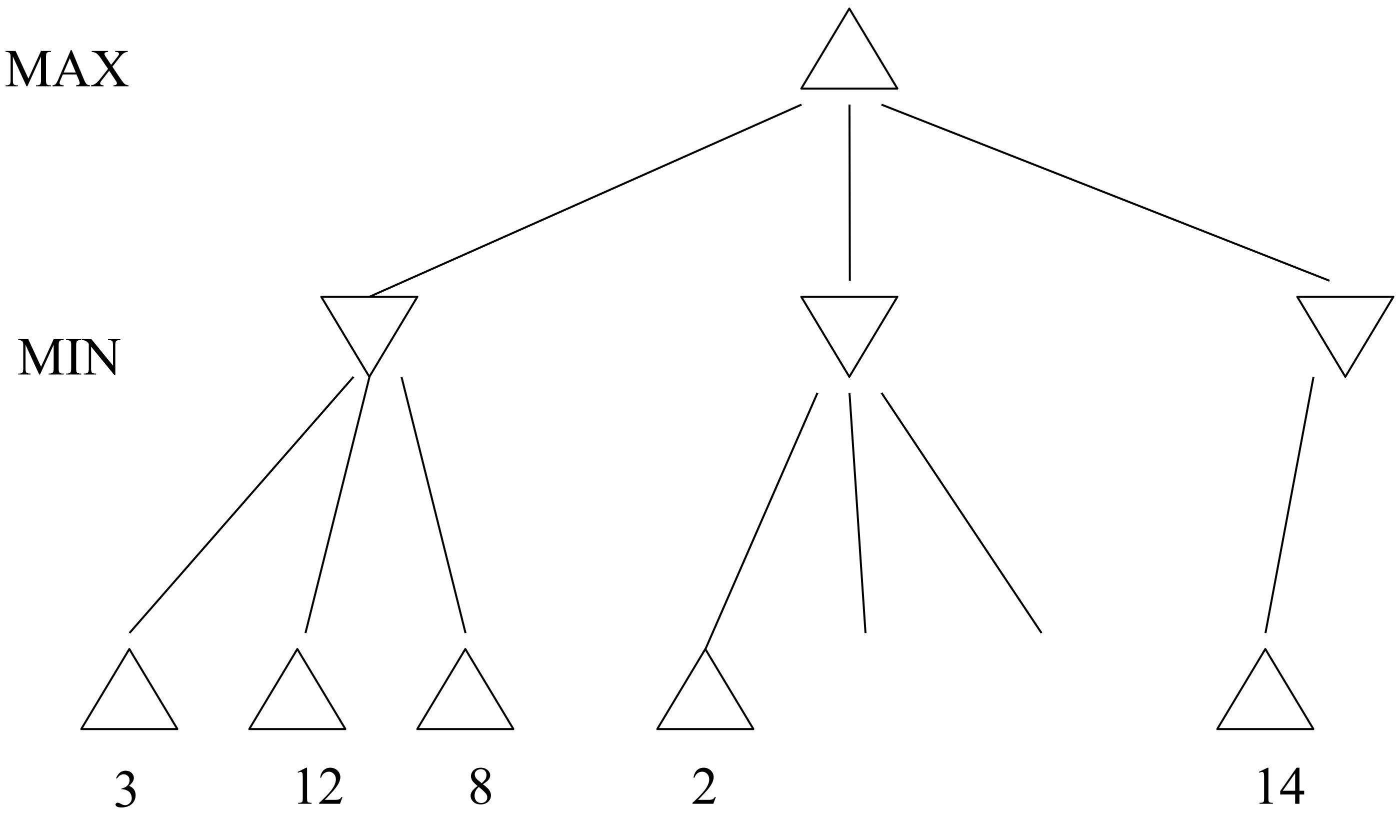
Minimax Tree



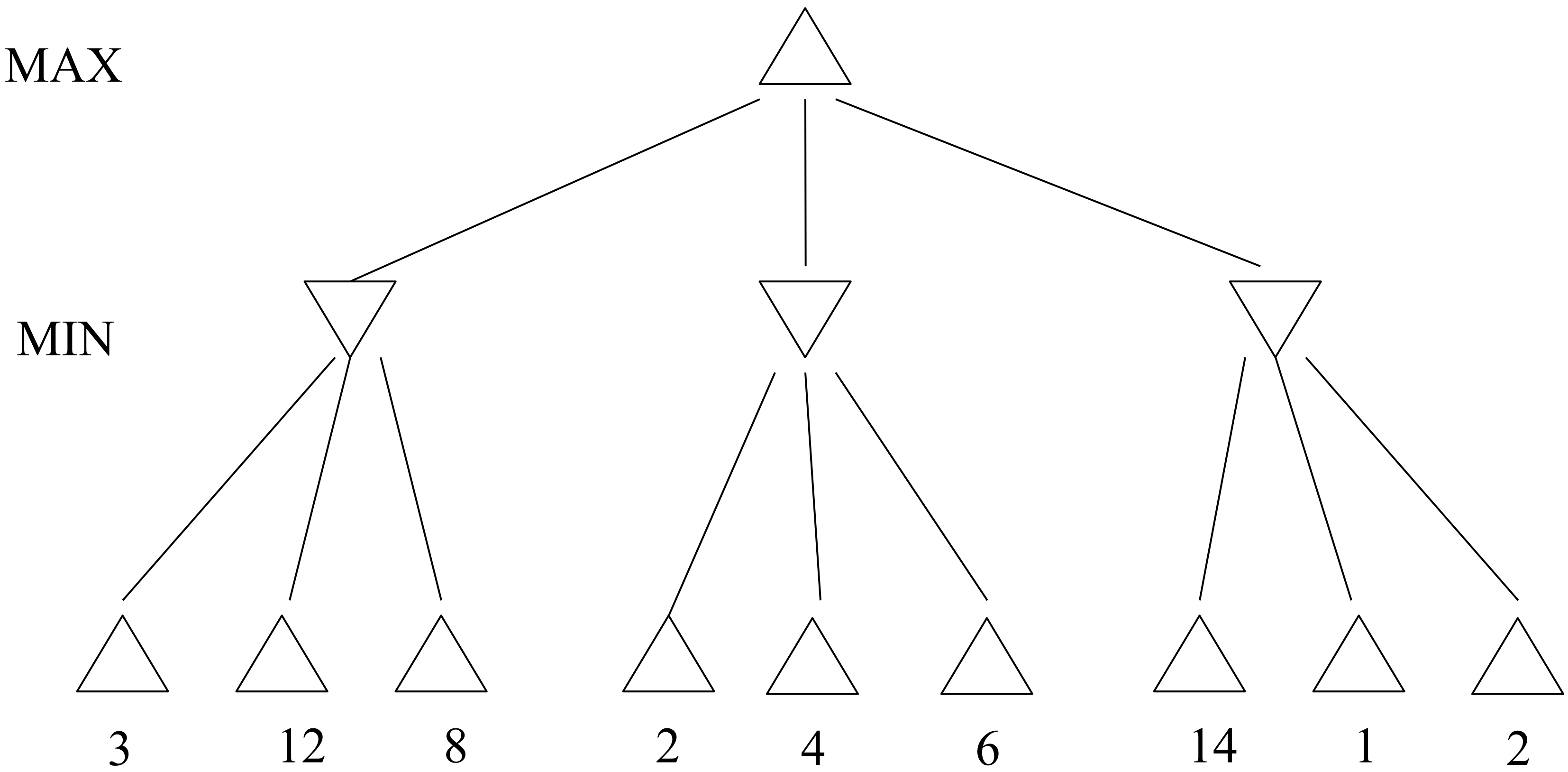
Minimax Tree



Minimax Tree

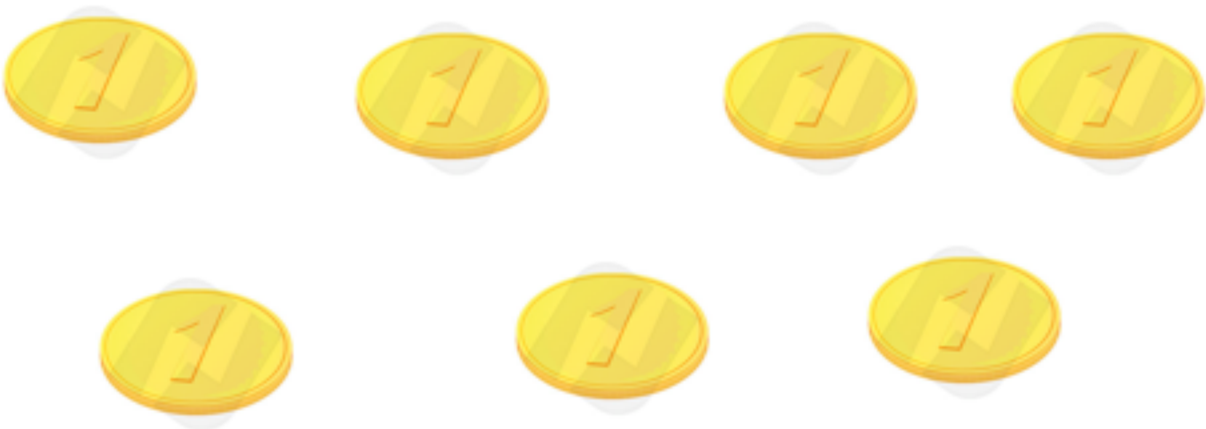


Minimax Tree



Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses



Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Example: Nim

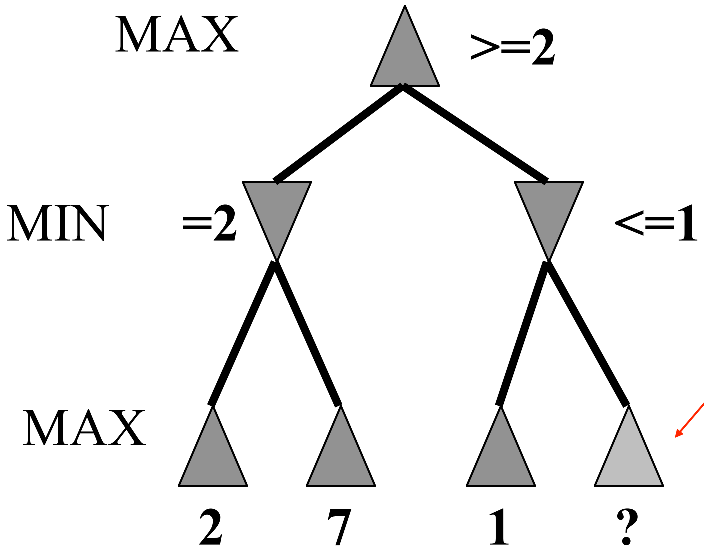
- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table -- we'll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Alpha-beta pruning

- We can improve on the performance of the minimax algorithm through **alpha-beta pruning**
- Basic idea: *"If you have an idea that is surely bad, don't take the time to see how truly awful it is."* -- Pat Winston



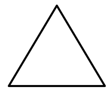
- We don't need to compute the value at this node.
- No matter what it is, it can't affect the value of the root node.

Alpha-beta pruning

- Traverse the search tree in depth-first order
- At each **MAX** node n, **alpha(n)** = maximum value found so far
- At each **MIN** node n, **beta(n)** = minimum value found so far
  – Note: The alpha values start at -infinity and only increase, while beta values start at +infinity and only decrease.
- **Beta cutoff**: Given a MAX node n, cut off the search below n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(i) for some MIN node ancestor i of n.
- **Alpha cutoff:** stop searching below MIN node n if beta(n) <= alpha(i) for some MAX node ancestor i of n.

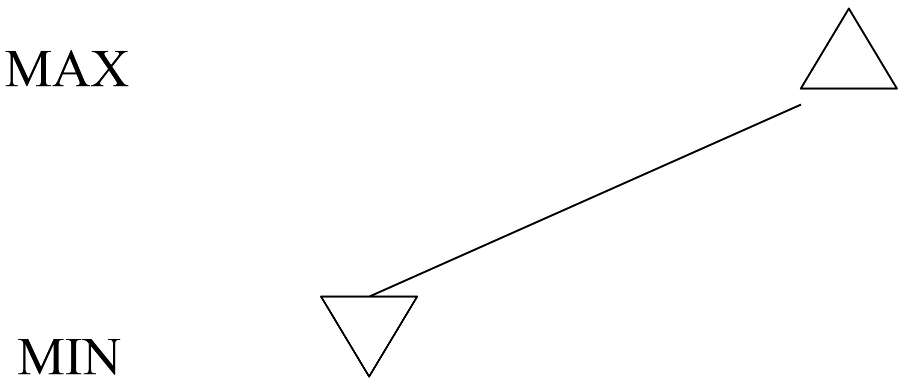
Alpha-beta example

MAX

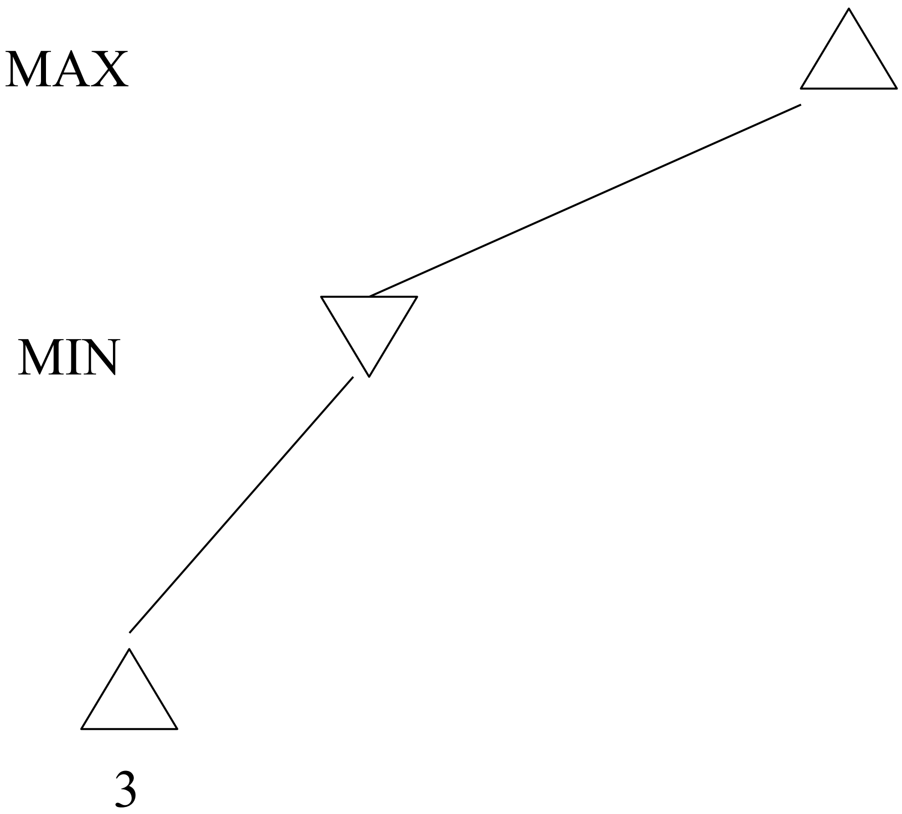


MIN

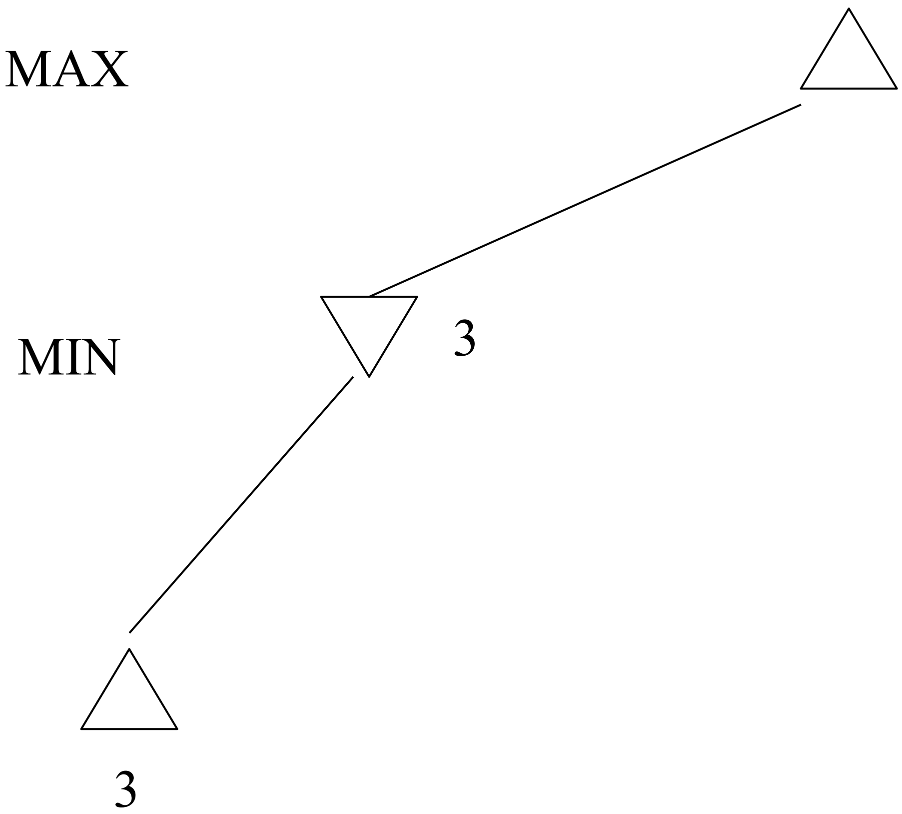
Alpha-beta example



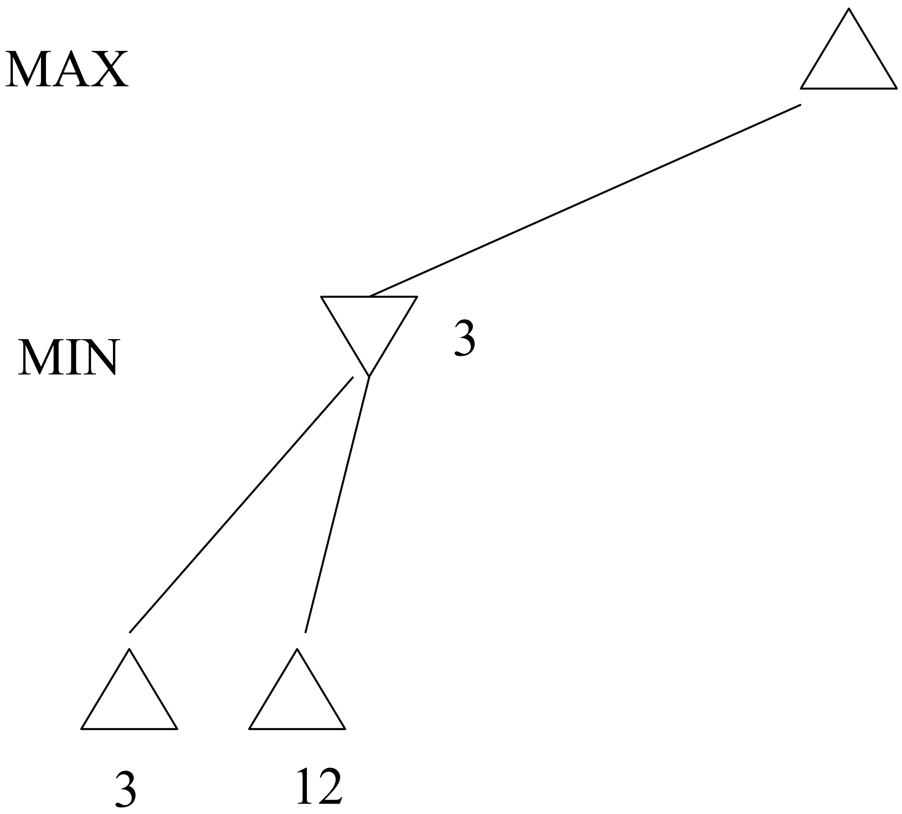
Alpha-beta example



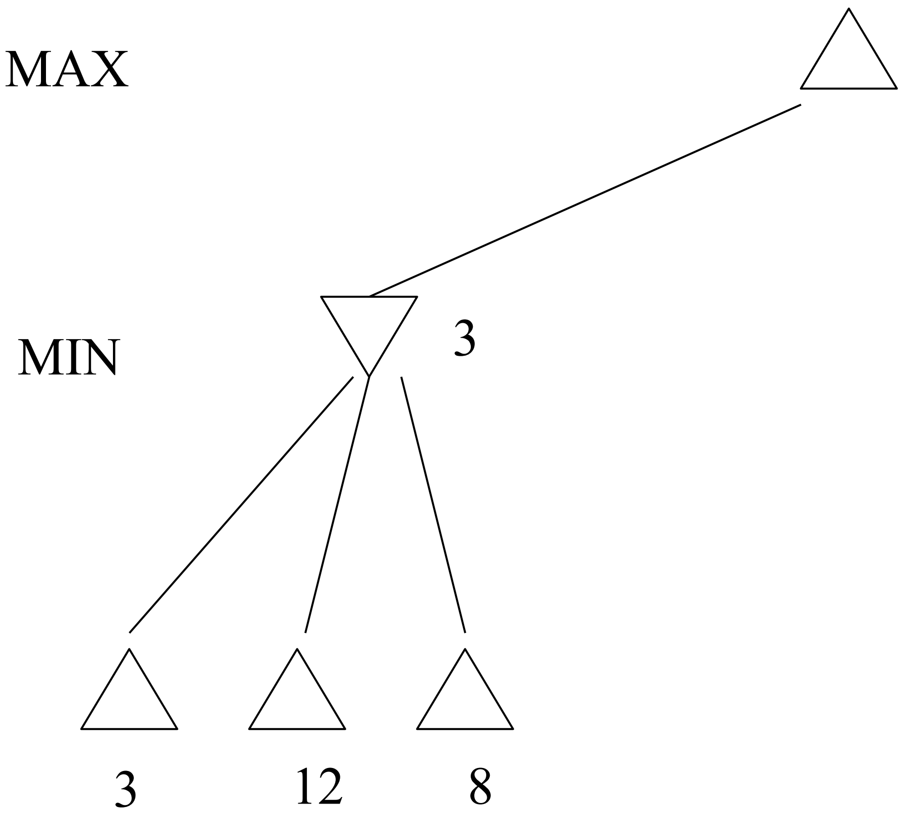
Alpha-beta example



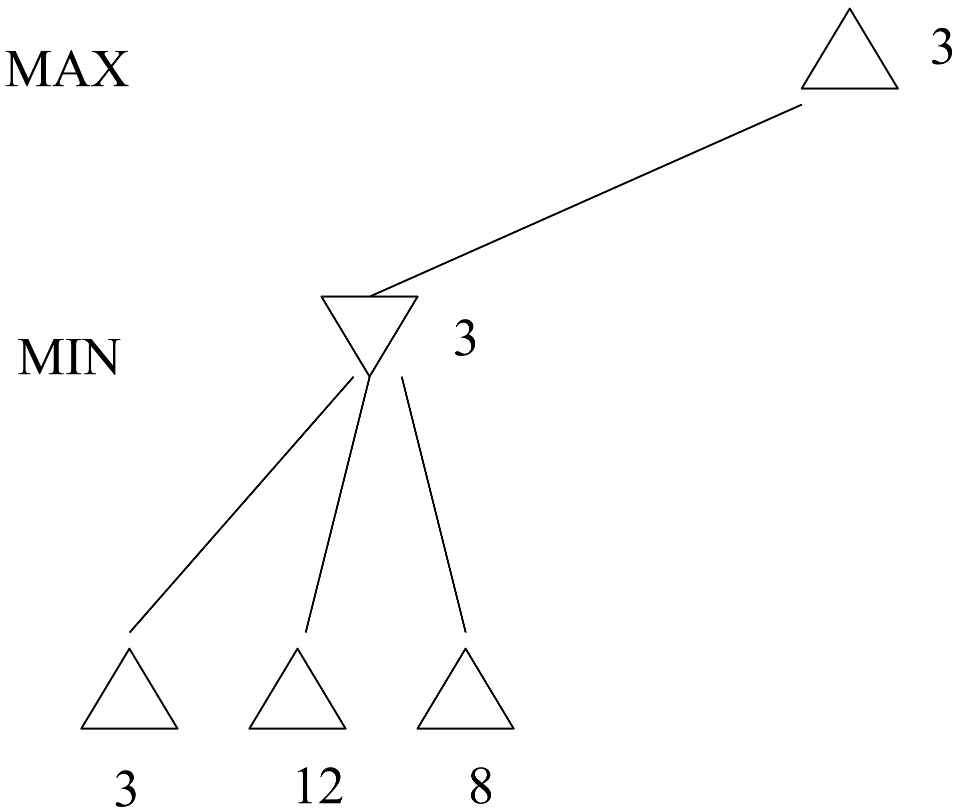
Alpha-beta example



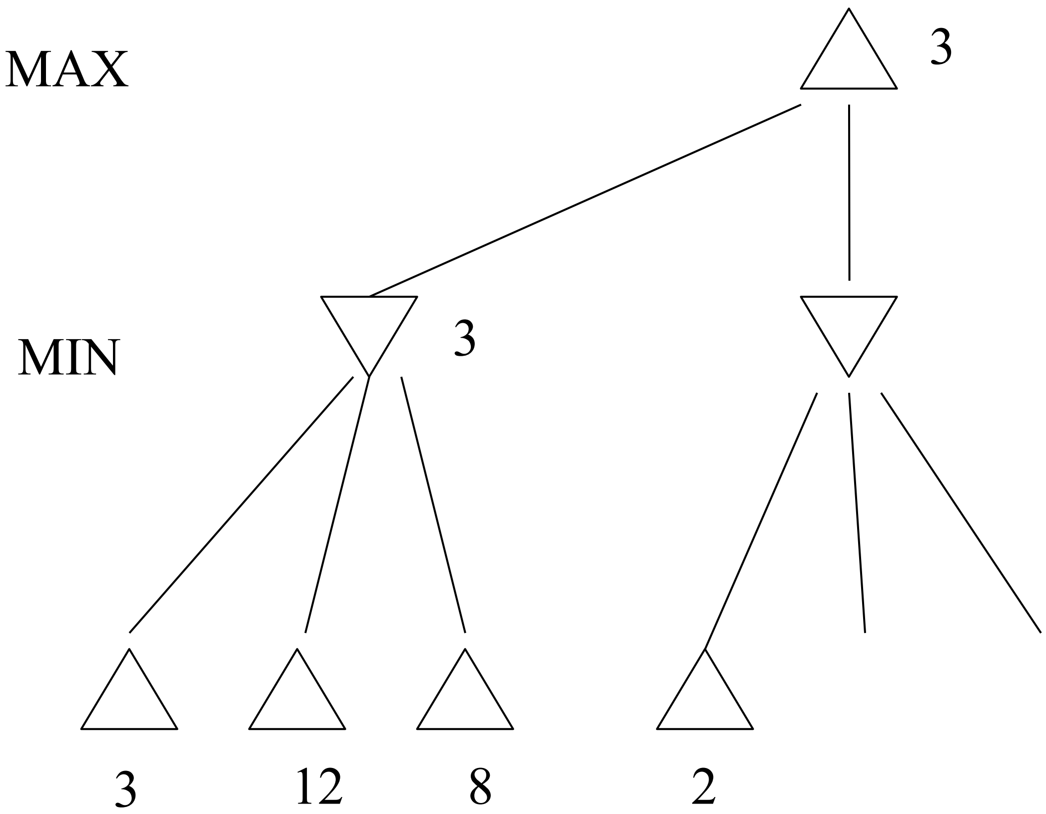
Alpha-beta example



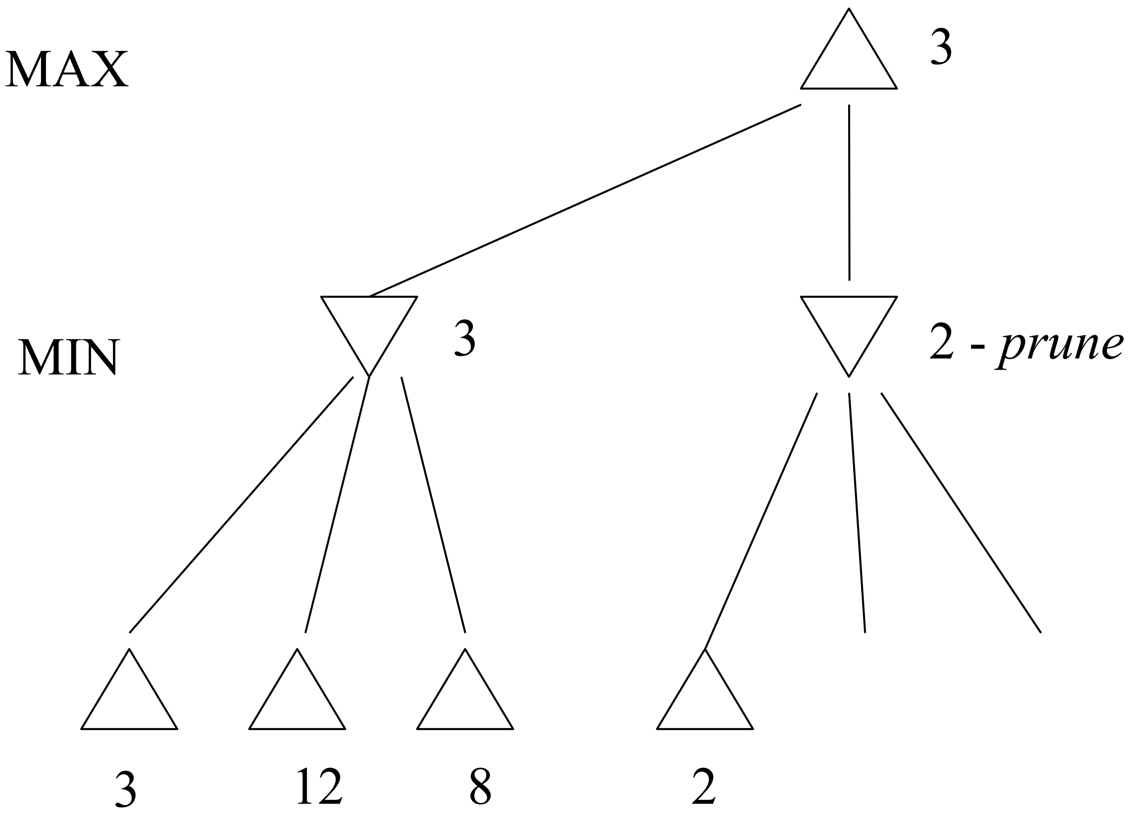
Alpha-beta example



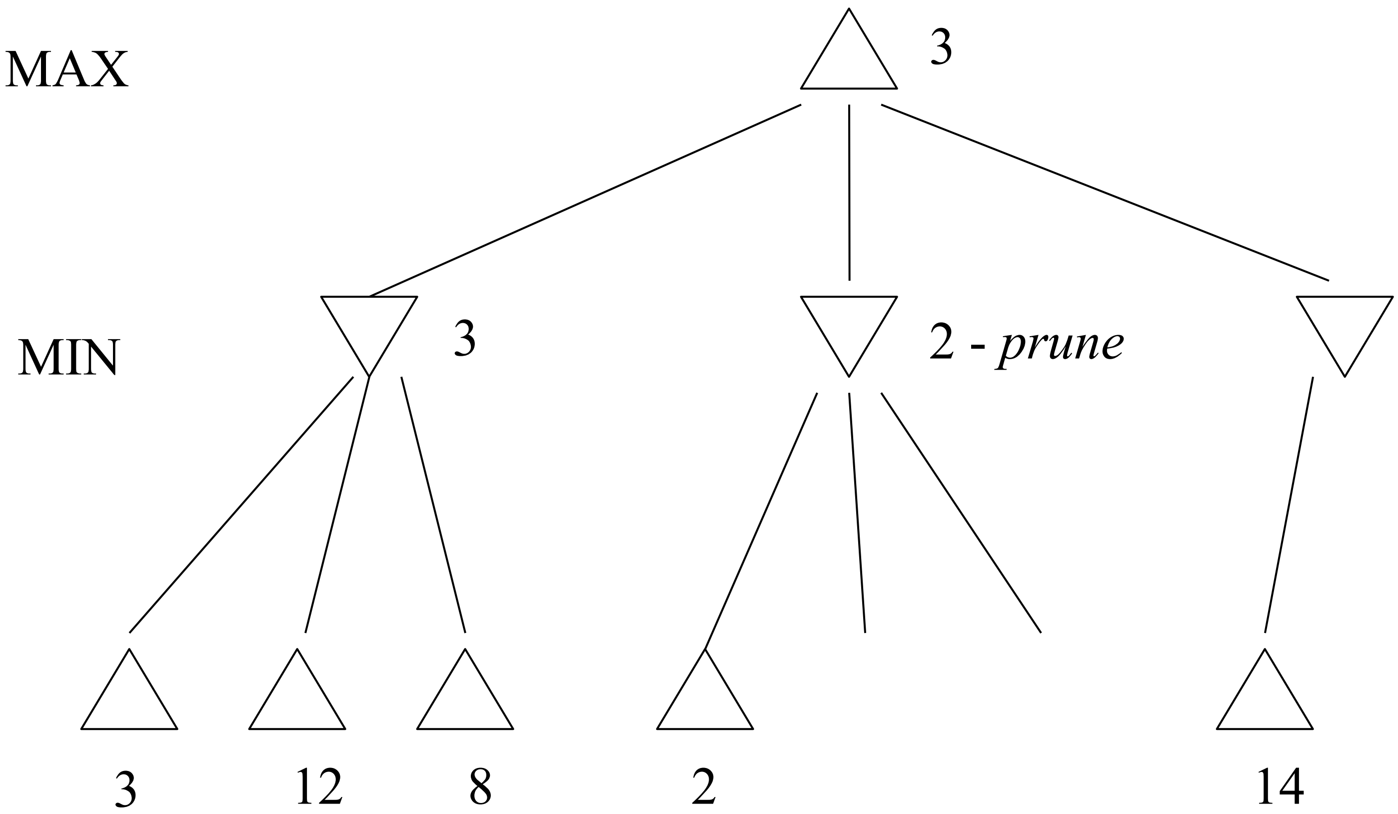
Alpha-beta example



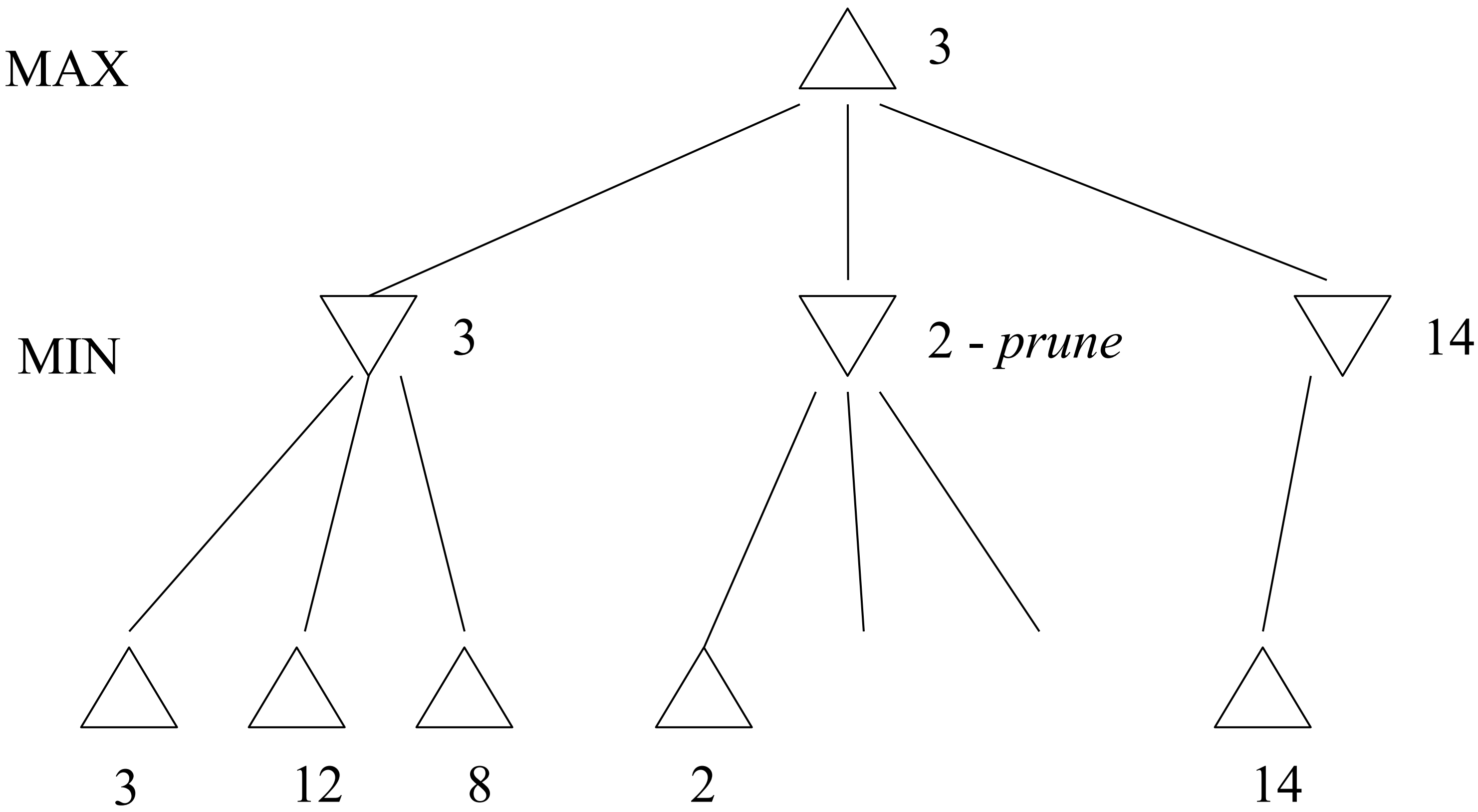
Alpha-beta example



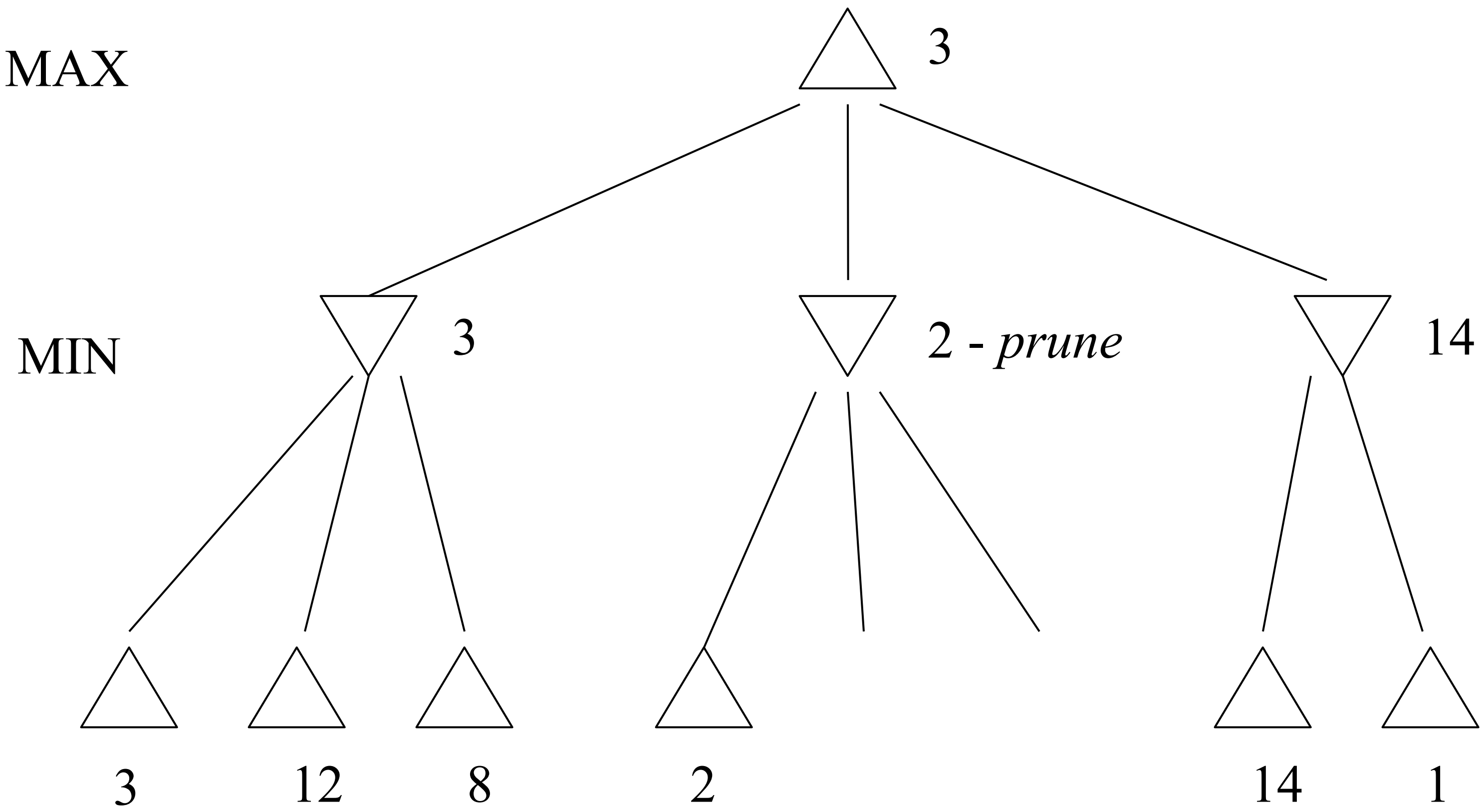
Alpha-beta example



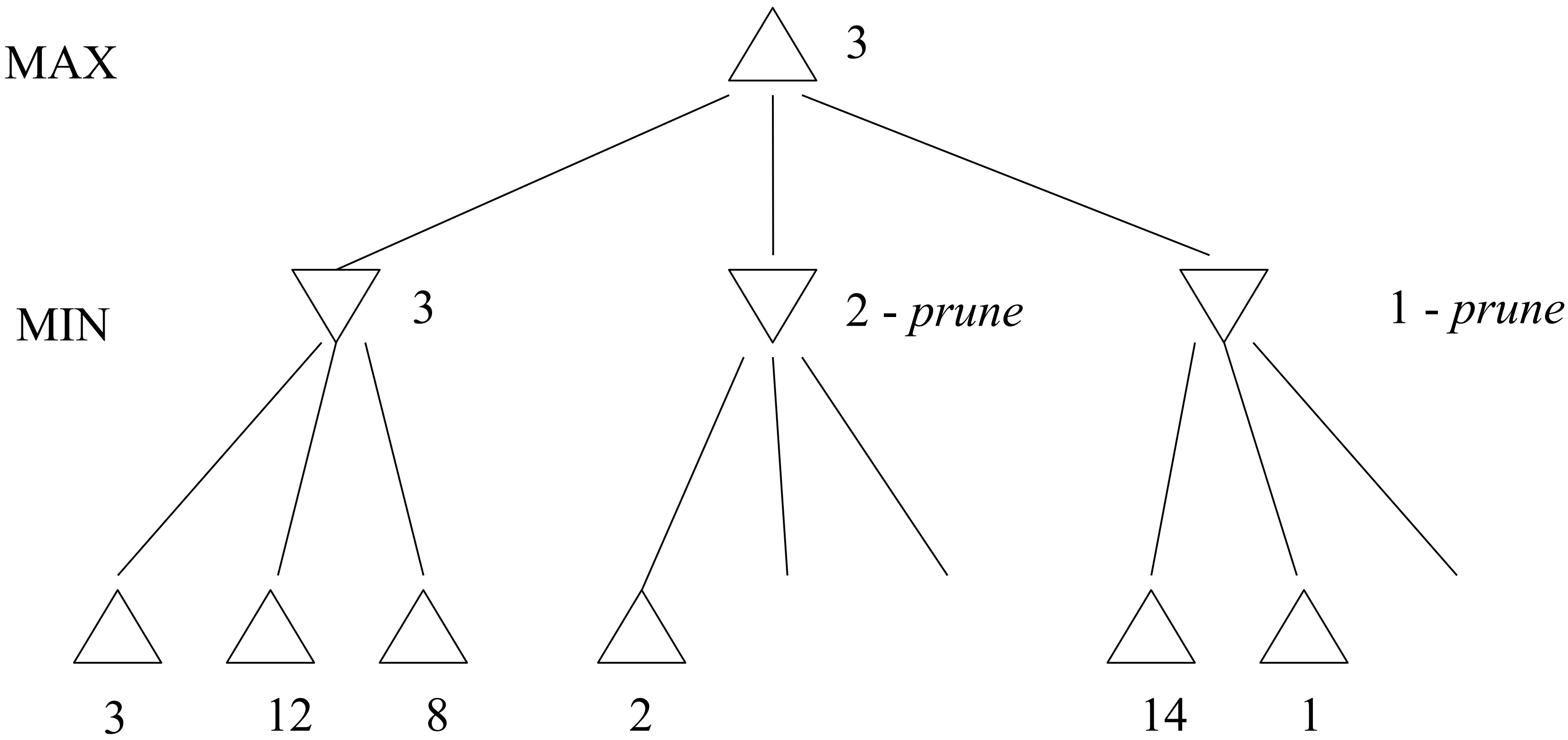
Alpha-beta example



Alpha-beta example



Alpha-beta example



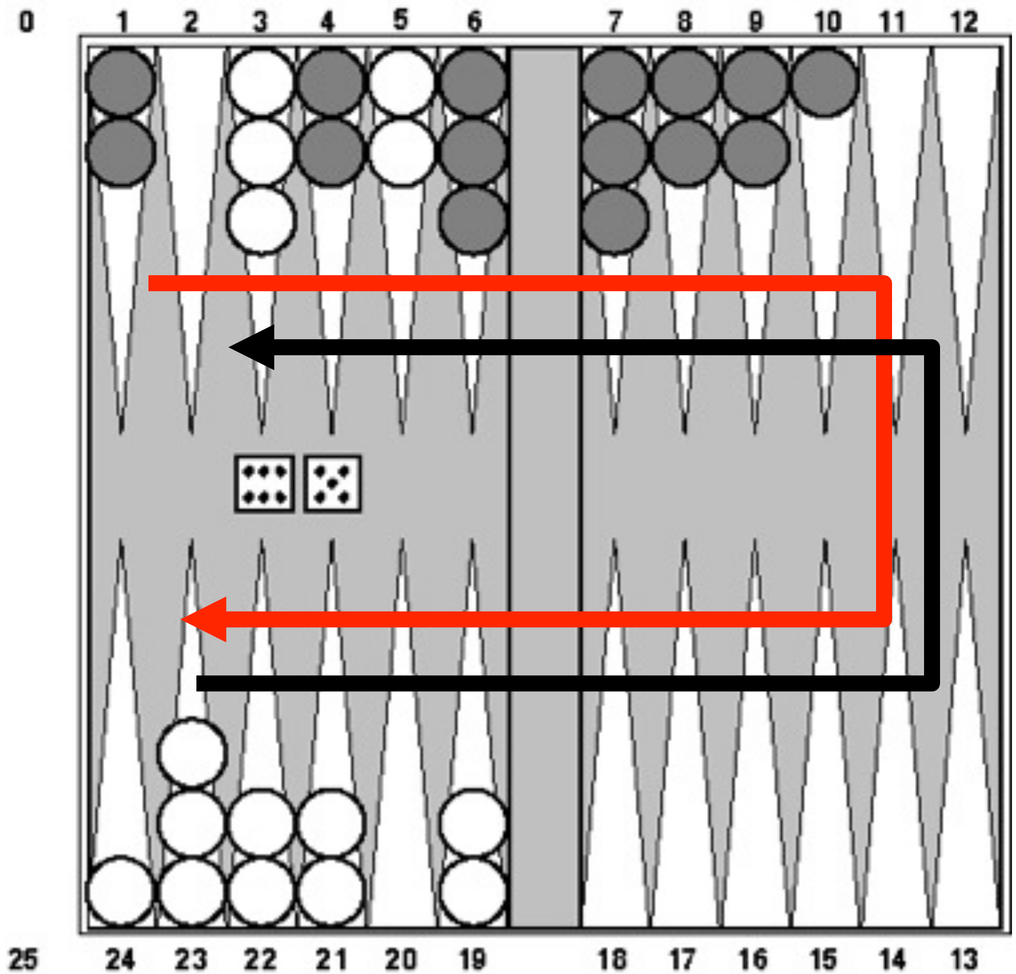
Alpha-beta algorithm

```
function MAX-VALUE (state, α, β)
    ;; α = best MAX so far; β = best MIN
if TERMINAL-TEST (state) then return UTILITY(state)
v := -∞
for each s in SUCCESSORS (state) do
    v := MAX (v, MIN-VALUE (s, α, β))
    if v >= β then return v
    α := MAX (α, v)
end
return v

function MIN-VALUE (state, α, β)
if TERMINAL-TEST (state) then return UTILITY(state)
v := ∞
for each s in SUCCESSORS (state) do
    v := MIN (v, MAX-VALUE (s, α, β))
    if v <= α then return v
    β := MIN (β, v)
end
return v
```

Games of chance

- Backgammon is a two-player game with **uncertainty**.
- Players roll dice to determine what moves to make.
- White has just rolled *5 and 6* and has four legal moves:
    - 5-10, 5-11
    - 5-11, 19-24
    - 5-10, 10-16
    - 5-11, 11-16
- Such games are good for exploring decision making in adversarial problems involving skill and luck.



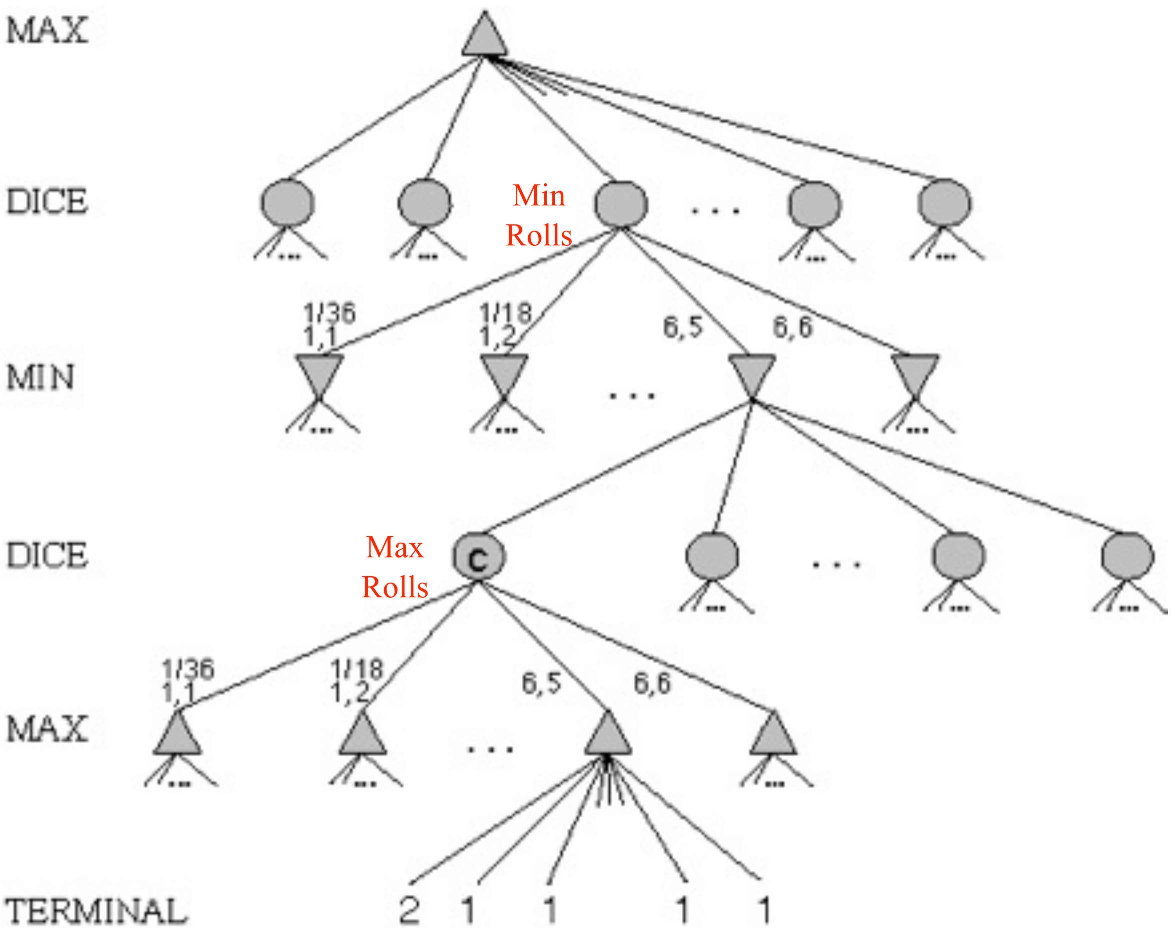
Game trees with chance nodes

- **Chance nodes** (shown as circles) represent random events
- For a random event with N outcomes, each chance node has N distinct children; a probability is associated with each
- (For 2 dice, there are 21 distinct outcomes)
- Use minimax to compute values for MAX and MIN nodes
- Use **expected values** for chance nodes
- For chance nodes over a max node, as in C:

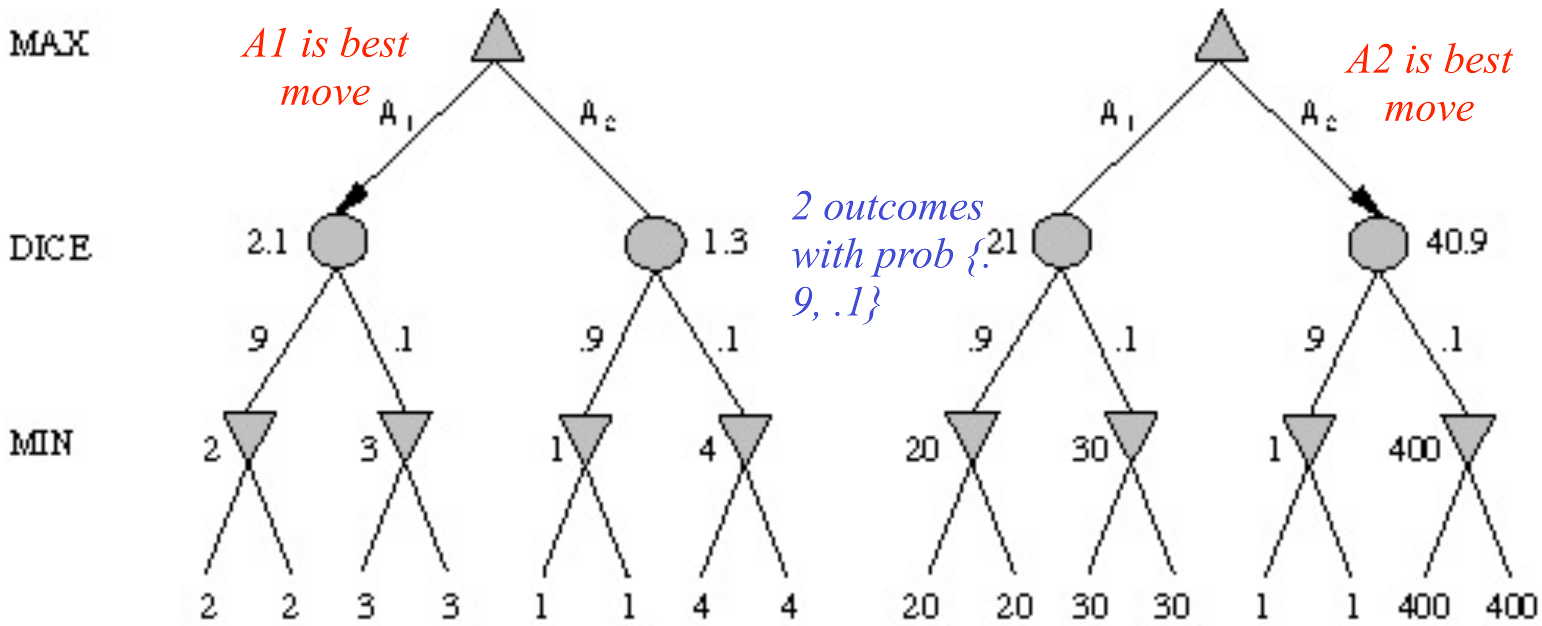
expectimax(C) = $\sum_i(P(d_i) * \text{maxvalue}(i))$

- For chance nodes over a min node:

expectimin(C) = $\sum_i(P(d_i) * \text{minvalue}(i))$



Meaning of the evaluation function



- Dealing with probabilities and expected values means we have to be careful about the “meaning” of values returned by the static evaluator.
- Note that a “relative-order preserving” change of the values would not change the decision of minimax, but could change the decision with chance nodes.
- Linear transformations are OK

Example: Oopsy-Nim

- Starts out like Nim
- Each player in turn has to pick up either one or two objects
- Sometimes (with probability 0.25), when you try to pick up two objects, you drop them both
- Picking up a single object always works
- Whoever picks up the last object loses

- Question: Why can't we draw the entire game tree?

Example: Oopsy-Nim

- Starts out like Nim
- Each player in turn has to pick up either one or two objects
- Sometimes (with probability 0.25), when you try to pick up two objects, you drop them both
- Picking up a single object always works
- Whoever picks up the last object loses

- Question: Why can't we draw the entire game tree?

Example: Oopsy-Nim

- Starts out like Nim
- Each player in turn has to pick up either one or two objects
- Sometimes (with probability 0.25), when you try to pick up two objects, you drop them both
- Picking up a single object always works
- Whoever picks up the last object loses

- Question: Why can't we draw the entire game tree?

Example: Oopsy-Nim

- Starts out like Nim
- Each player in turn has to pick up either one or two objects
- Sometimes (with probability 0.25), when you try to pick up two objects, you drop them both
- Picking up a single object always works
- Whoever picks up the last object loses

- Question: Why can't we draw the entire game tree?

Example: Oopsy-Nim

- Starts out like Nim
- Each player in turn has to pick up either one or two objects
- Sometimes (with probability 0.25), when you try to pick up two objects, you drop them both
- Picking up a single object always works
- Whoever picks up the last object loses

- Question: Why can’t we draw the entire game tree?