Adversarial Search and Games

Simon Dixon
Queen Mary University of London

Objectives

Queen Mary
University of London

- Learn about
  - Optimal decisions in adversarial situations
  - The minimax algorithm
  - α-β pruning
  - Imperfect, real-time decision making

Adversarial search and games

- We have assumed so far that the world we are searching doesn't change
  - This is often not the case
- Consider Games
  - Two or more players, usually in competition
  - There are many kinds of game, but we will only deal with *zero-sum games of perfect information*

2 players:	player one wins	→	+1
	player two loses	→	-1
	TOTAL		0

Why study games?

- Games are a form of multi-agent environment
  - What do other agents do and how do they affect our success?
  - Cooperative vs competitive multi-agent environments
  - Competitive multi-agent environments (games) give rise to *adversarial* search
- Why study games?
  - Interesting subject of AI study because they are hard for humans
  - Easy to represent; agents are restricted to a small number of actions
  - They are fun
- "Google AI algorithm masters ancient game of Go", *Nature News*, 27/1/16
  - "A computer has beaten a human professional for the first time at Go — an ancient board game that has long been viewed as one of the greatest challenges for artificial intelligence (AI)." See D. Silver et al., Nature, 529: 484–489 (2016).

Adversarial vs non-adversarial search

Queen Mary
University of London

- Search with no adversary
  - Solution is a set of actions for reaching goal
  - Heuristics and constraint satisfaction techniques can find optimal solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling, solitaire games
- Games – with adversary
  - Solution is a strategy (considering every possible opponent reply to a move)
  - Sometimes time limits force an approximate solution
  - Evaluation function: evaluate quality of game position
  - Examples: chess, checkers, Othello, war games, simulations of competition for limited resources

Types of game

Queen Mary
University of London

	deterministic	chance
perfect information	chess, draughts, go, othello	backgammon, monopoly
imperfect information		bridge, poker, scrabble, nuclear war

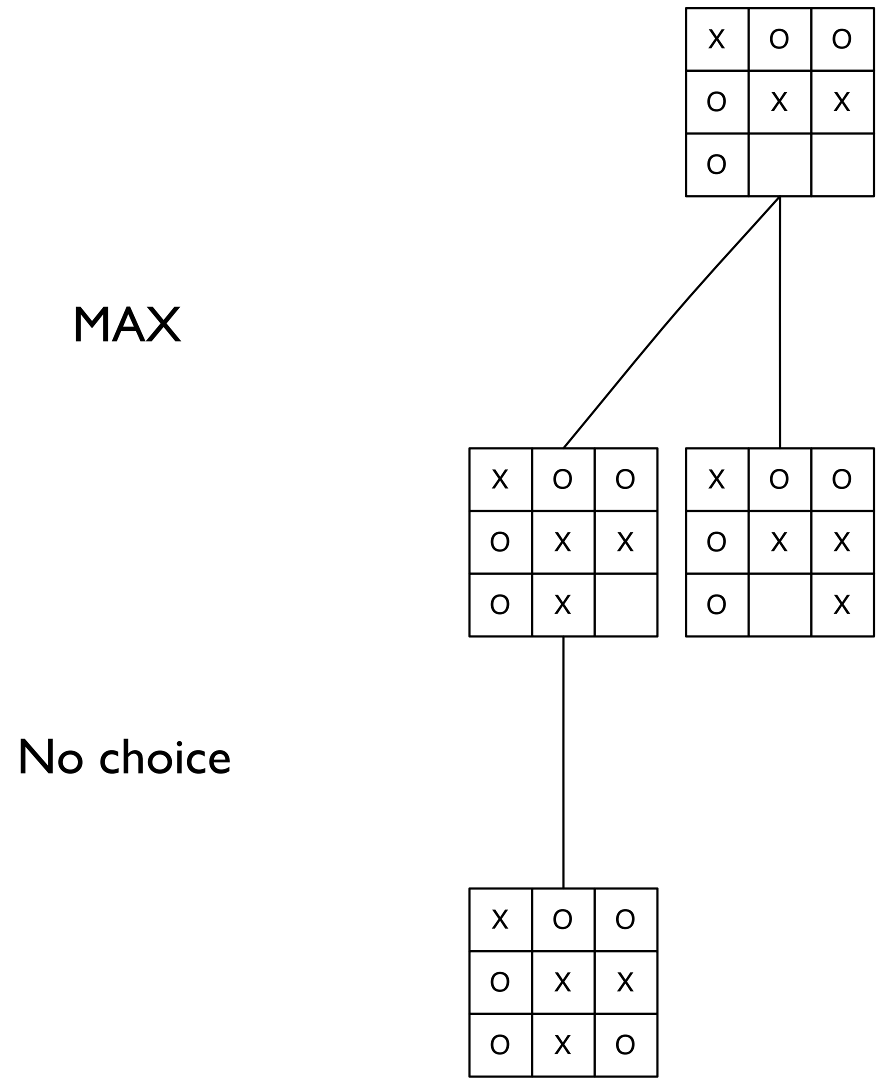
The MINIMAX method

- Game setup
  - Two players: MAX and MIN
  - MAX moves first (by convention) and they take turns until the game is over
  - Winner gets prize, loser gets penalty (same size as prize)
- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (legal move, resulting state) pairs from current state
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states
    - E.g. MAX wins (+1), loses (-1) or draws (0) in noughts and crosses (tic-tac-toe)
  - Each player uses a search tree to determine next move

Partial game tree for noughts and crosses

Queen Mary
University of London

- This example starts well into the game
- Suppose X is MAX



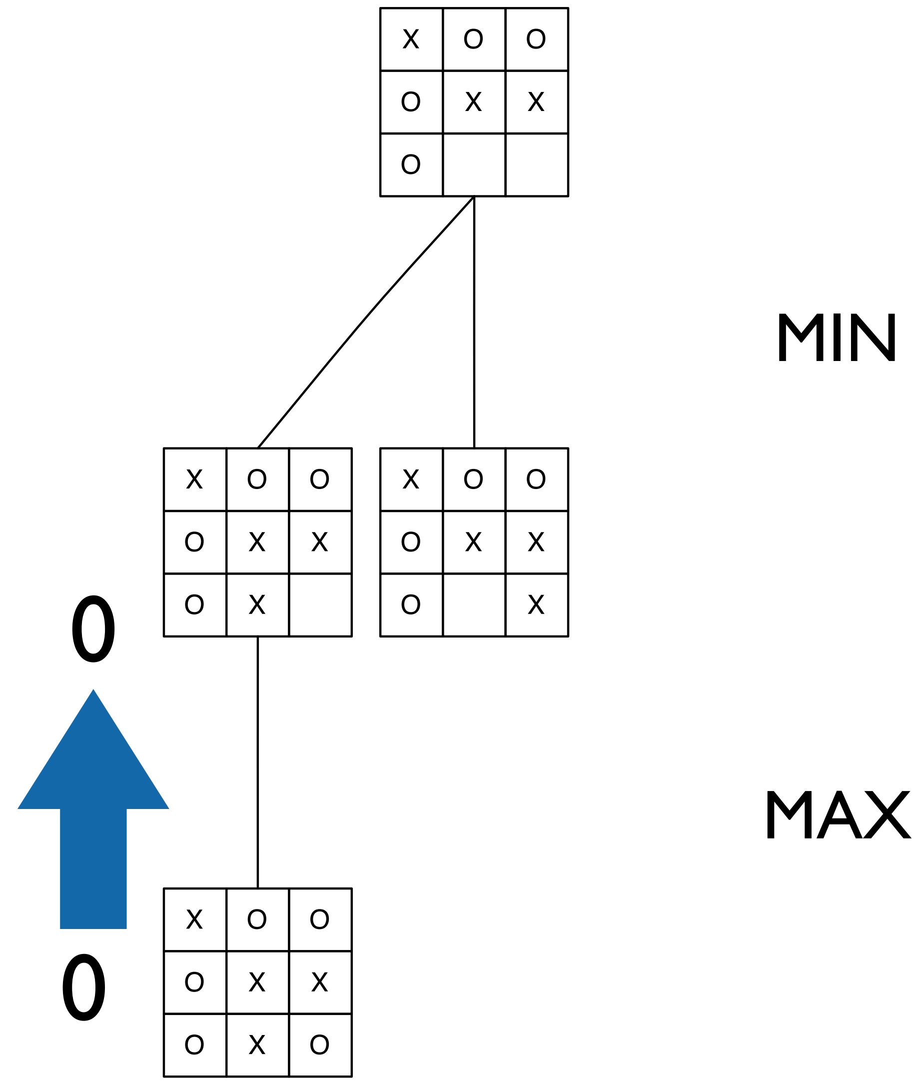
Optimal strategies

- Find the contingent strategy for MAX against opponent MIN
- Assumption: Both players play optimally
- Given a game tree, the optimal strategy can be determined by using the minimax value of each node, n:
  - if n is a terminal node, MINIMAX-VALUE(n) = UTILITY(n)
  - if n is a MAX node, MINIMAX-VALUE(n) = $\max_{s \in successors(n)}$ MINIMAX-VALUE(s)
  - if n is a MIN node, MINIMAX-VALUE(n) = $\min_{s \in successors(n)}$ MINIMAX-VALUE(s)

Partial game tree for noughts and crosses

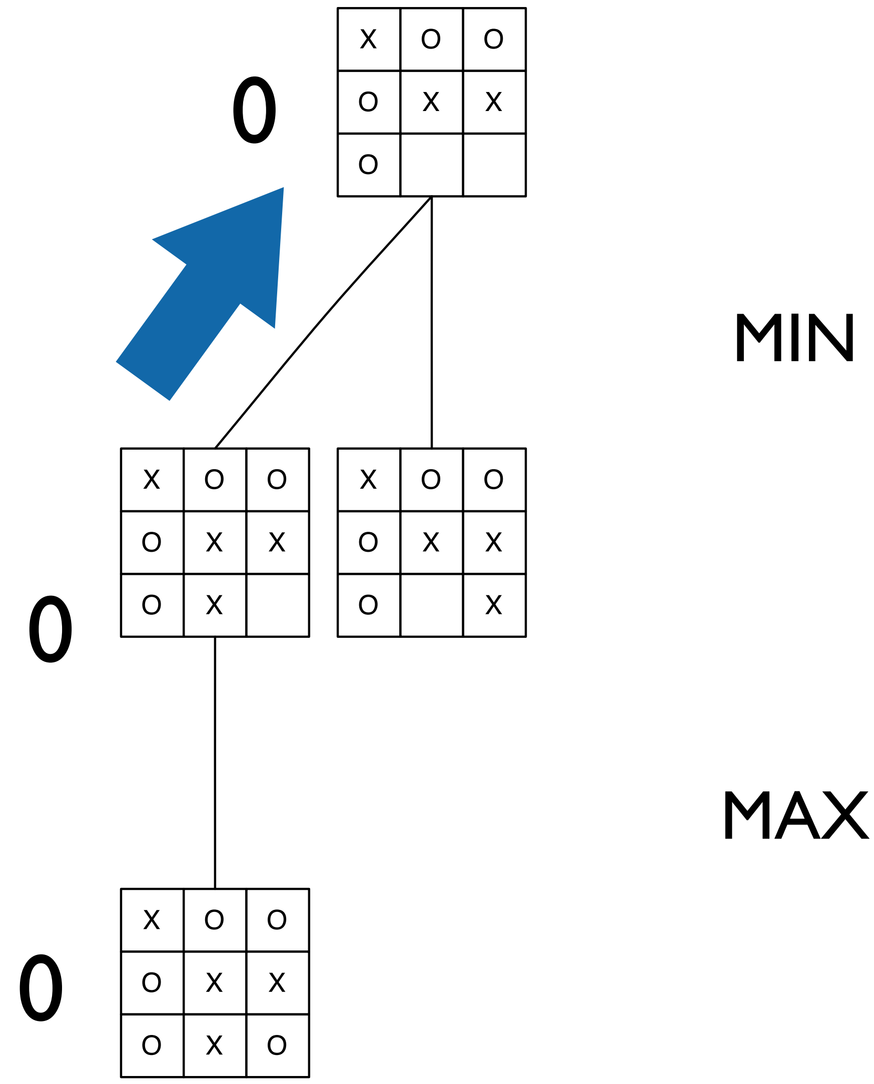
Queen Mary
University of London

- Perform DFS exhaustively to compute the topmost MINIMAX value
- O is MAX (played first), X is MIN
- Example utility fn:
  - 1 point for a win to MAX
  - 0 for a draw
  - -1 for a loss for MAX



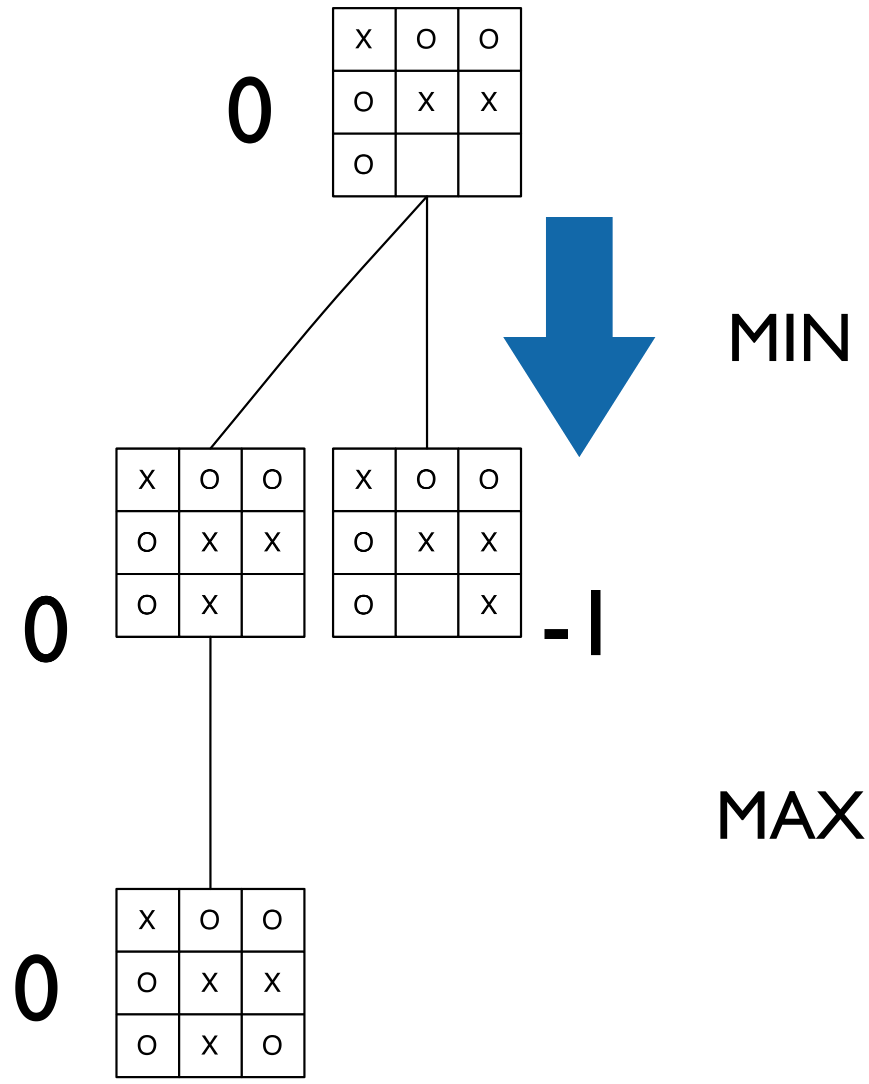
Partial game tree for noughts and crosses

- Perform DFS exhaustively to compute the topmost MINIMAX value
- O is MAX (played first), X is MIN
- Example utility fn:
  - 1 point for a win to MAX
  - 0 for a draw
  - -1 for a loss for MAX



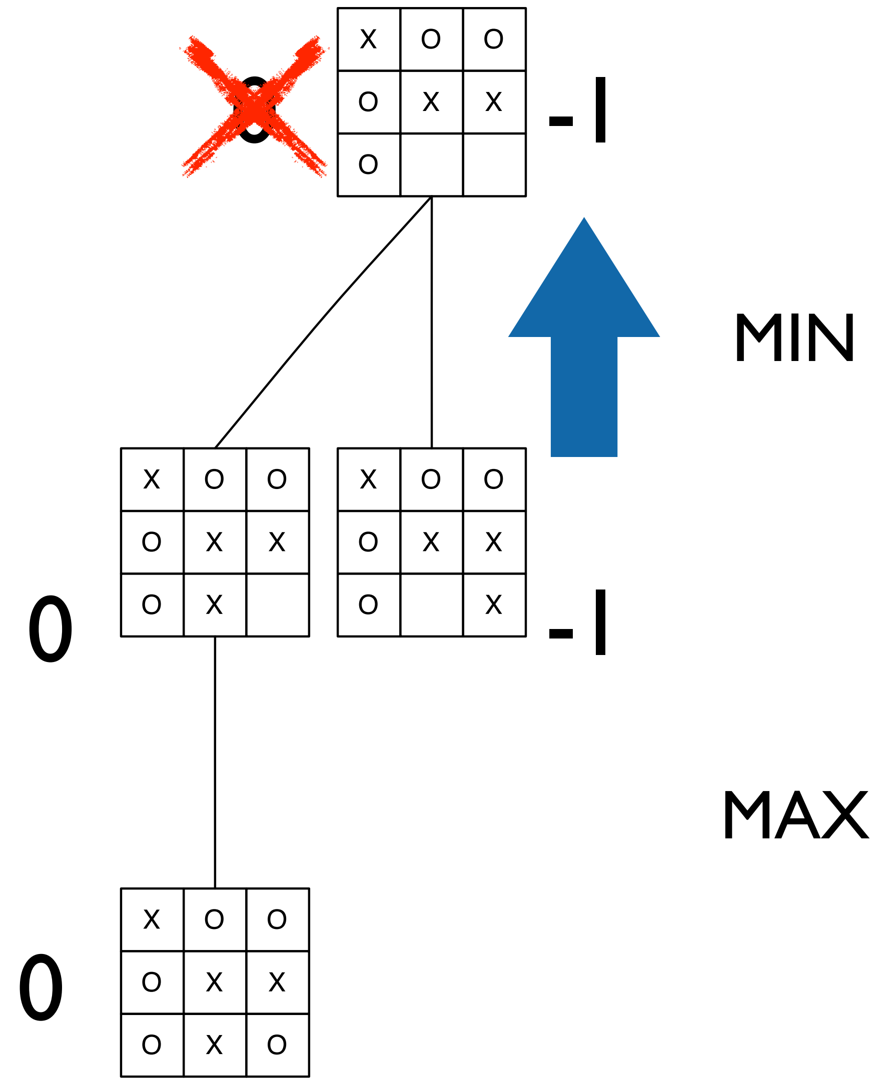
Partial game tree for noughts and crosses

- Perform DFS exhaustively to compute the topmost MINIMAX value
- O is MAX (played first), X is MIN
- Example utility fn:
  - 1 point for a win to MAX
  - 0 for a draw
  - -1 for a loss for MAX



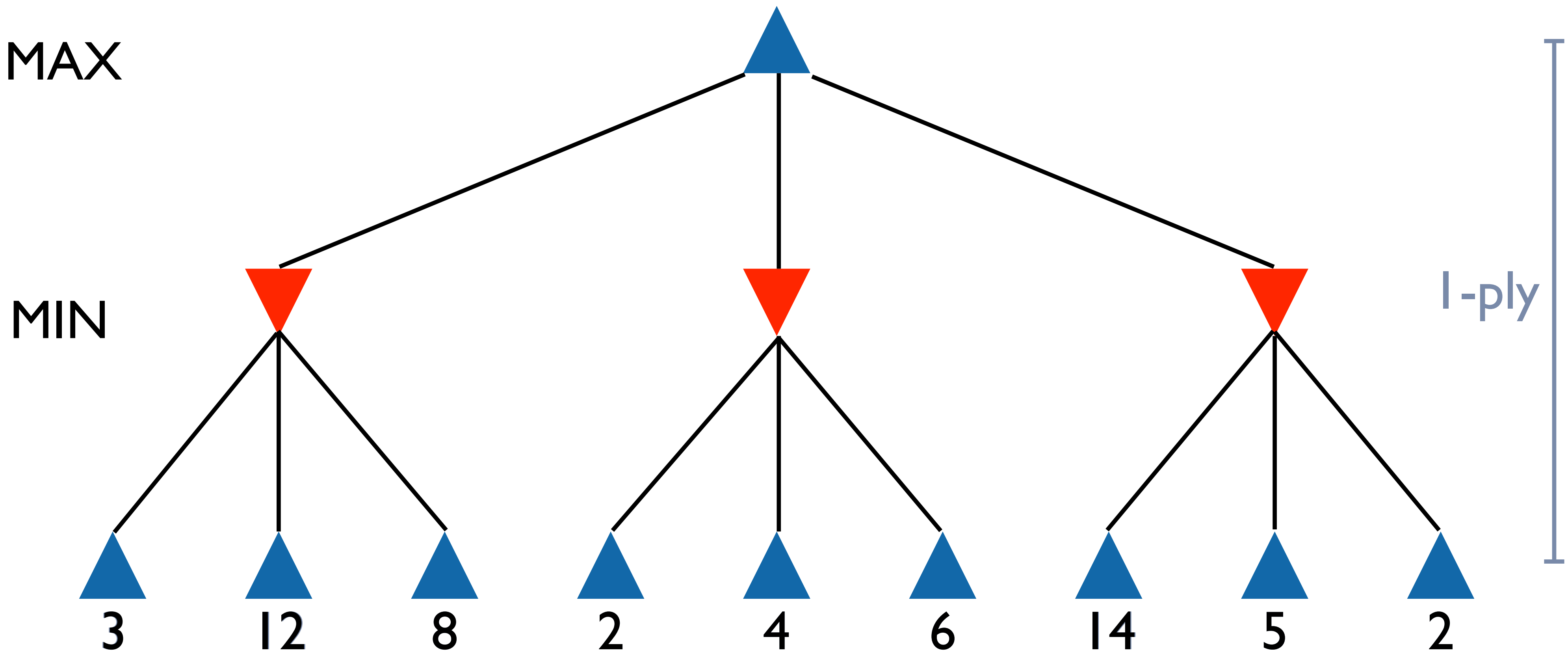
Partial game tree for noughts and crosses

- Perform DFS exhaustively to compute the topmost MINIMAX value
- O is MAX (played first), X is MIN
- Example utility fn:
  - 1 point for a win to MAX
  - 0 for a draw
  - -1 for a loss for MAX



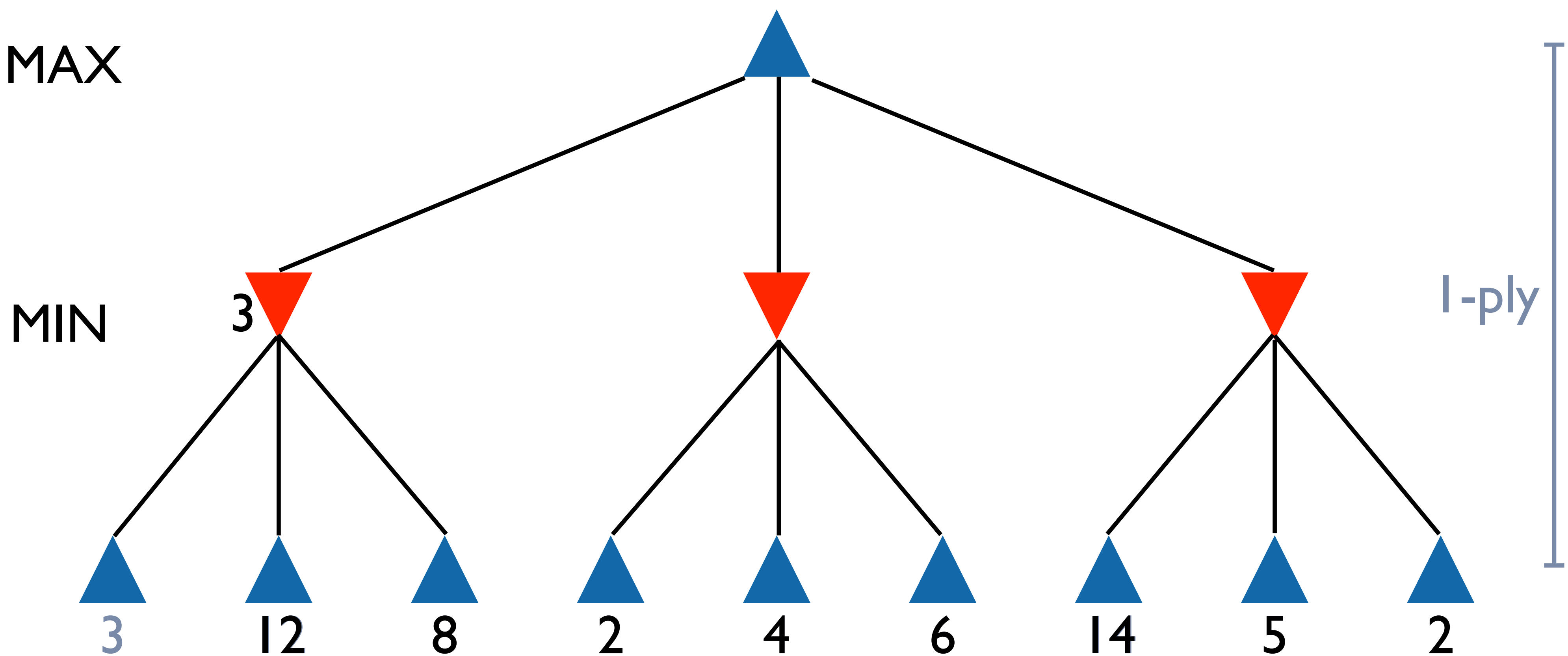
General 2-ply game tree

- Imaginary game, made-up values for example here



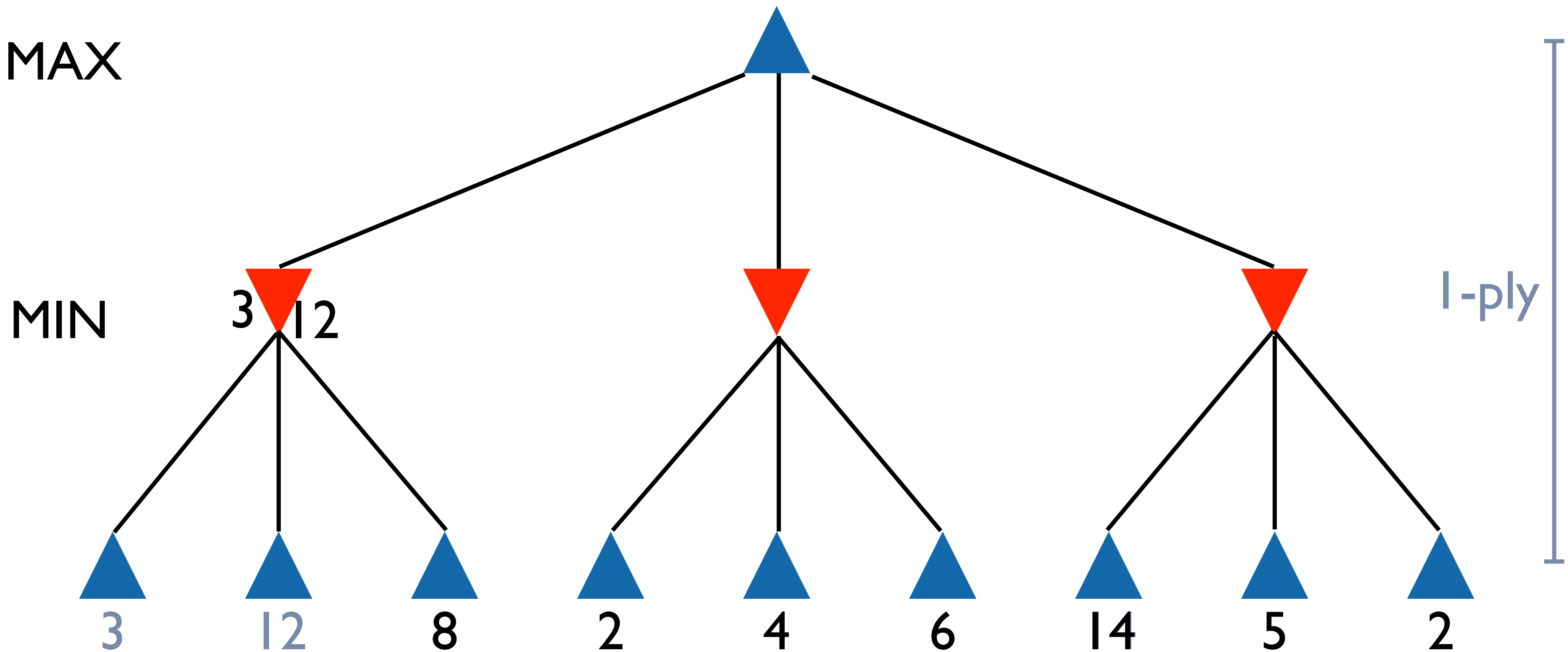
General 2-ply game tree

- Imaginary game, made-up values for example here



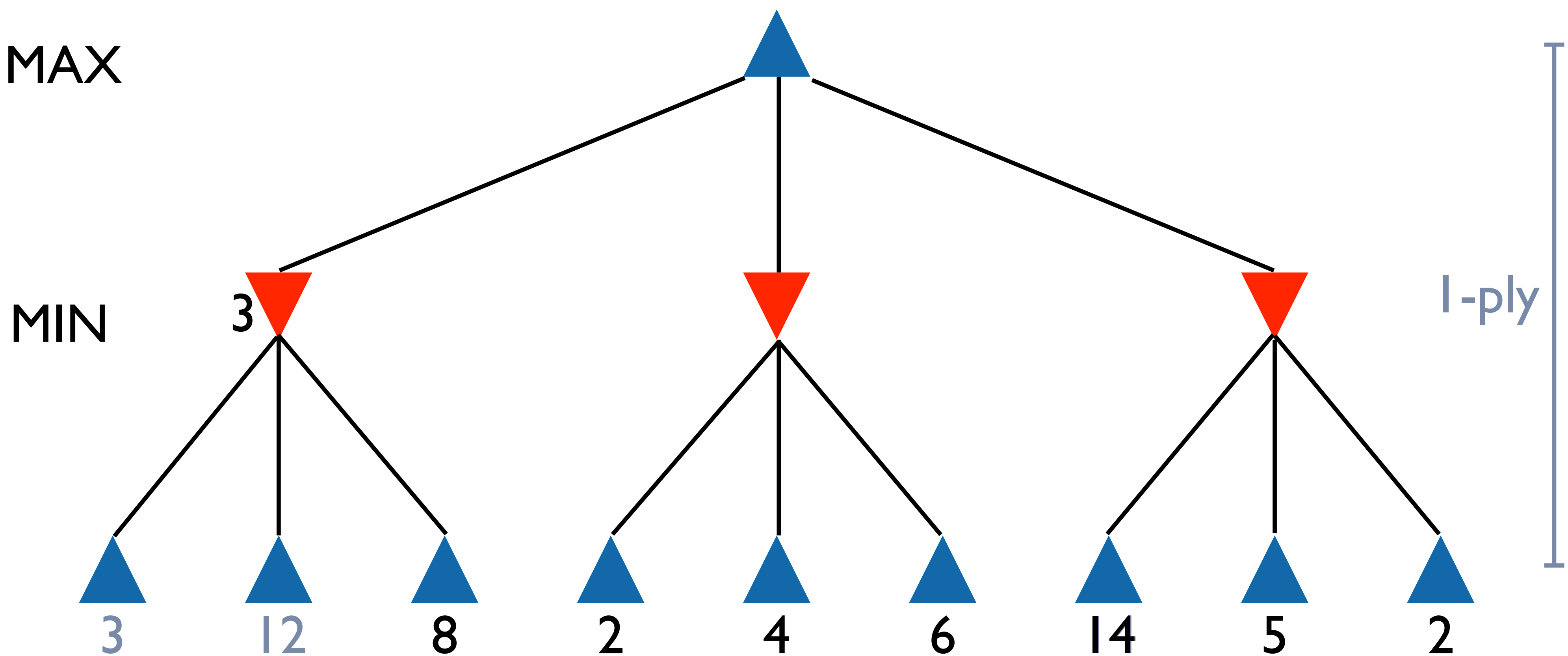
General 2-ply game tree

- Imaginary game, made-up values for example here



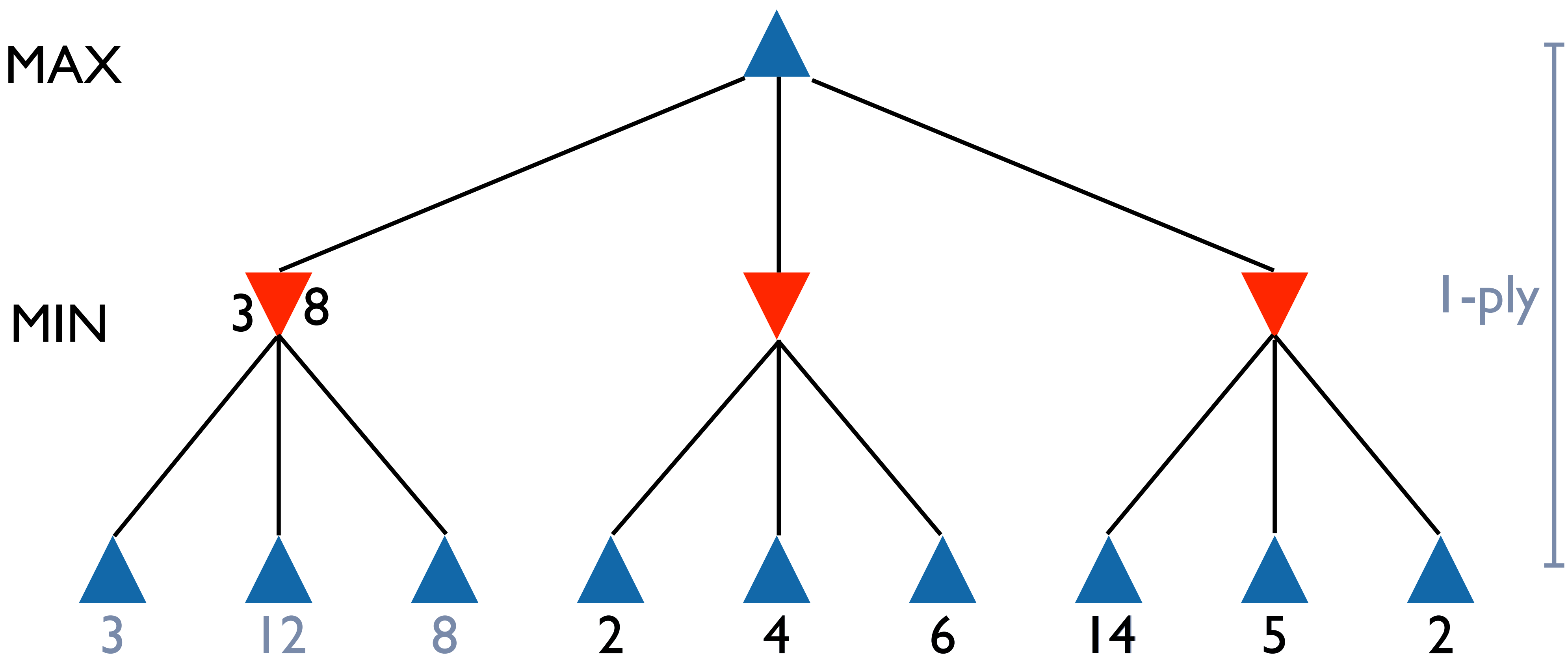
General 2-ply game tree

- Imaginary game, made-up values for example here



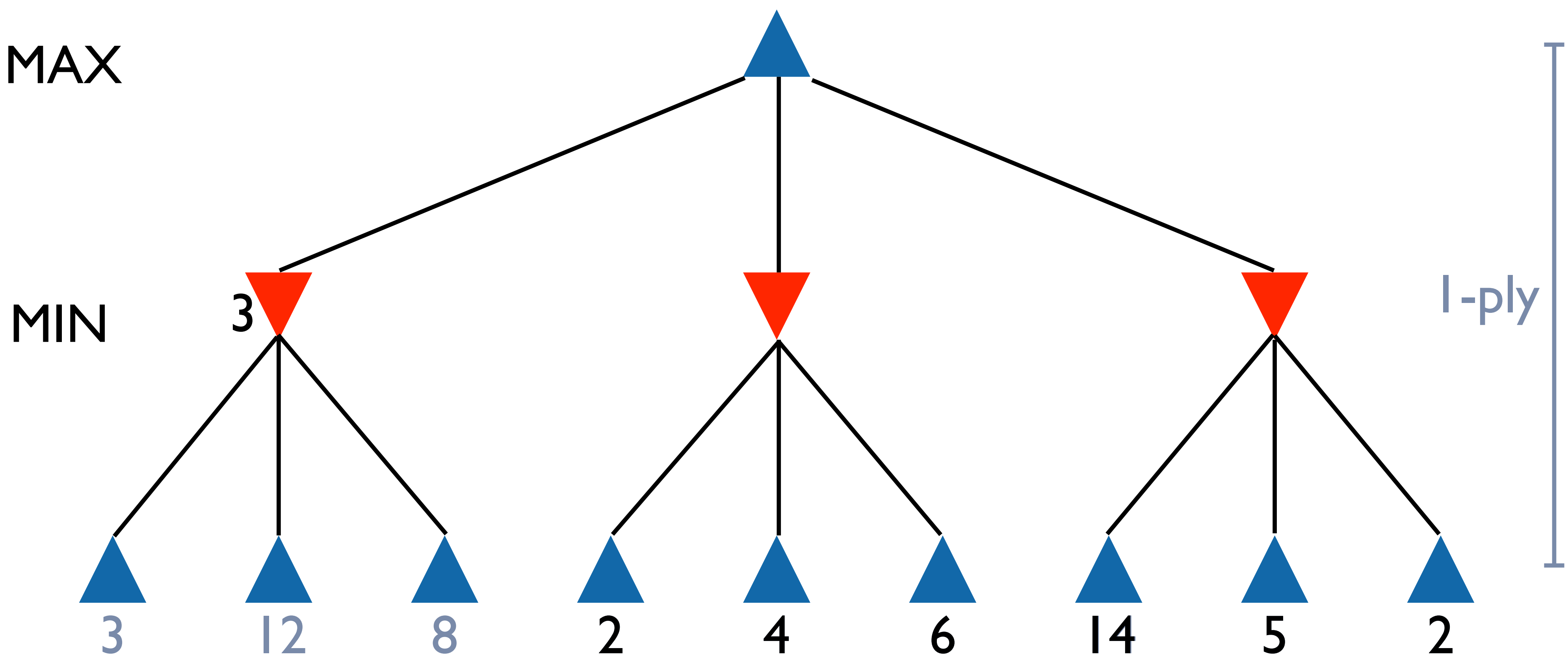
General 2-ply game tree

- Imaginary game, made-up values for example here



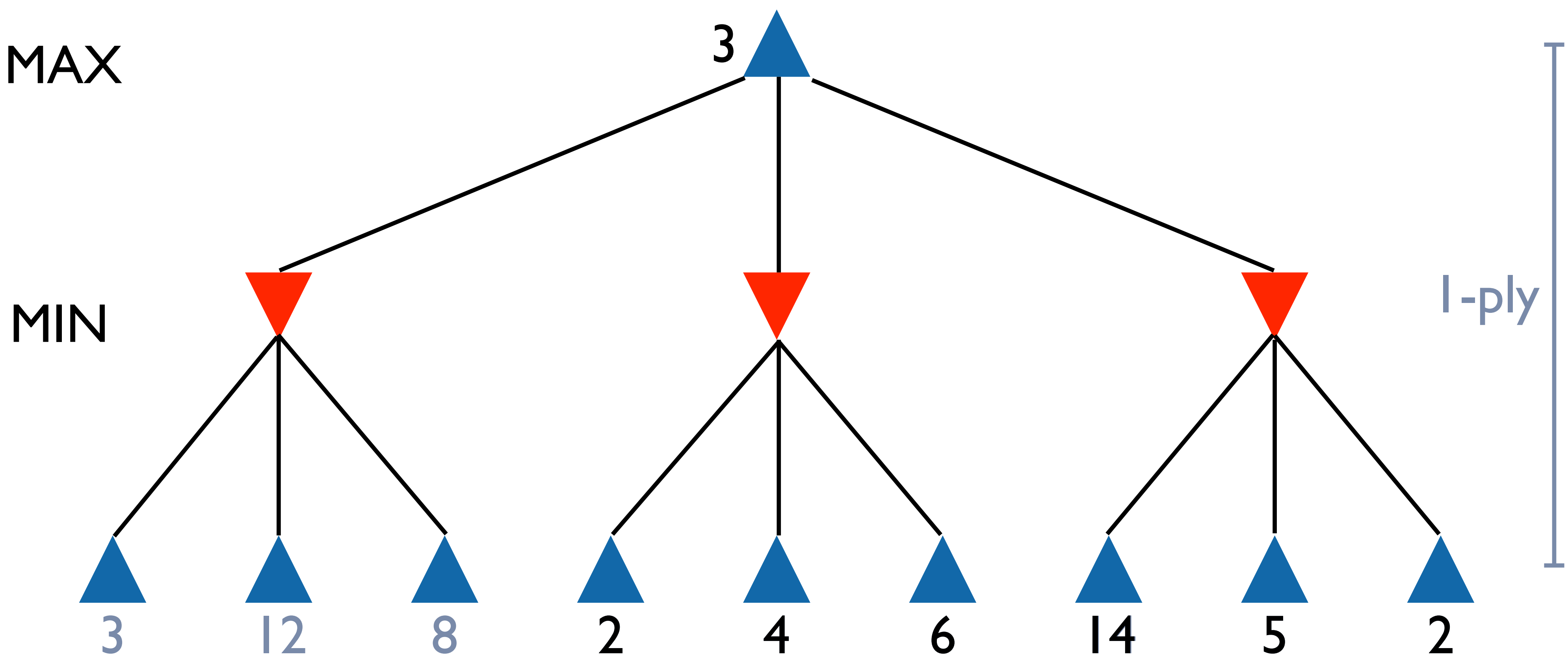
General 2-ply game tree

- Imaginary game, made-up values for example here



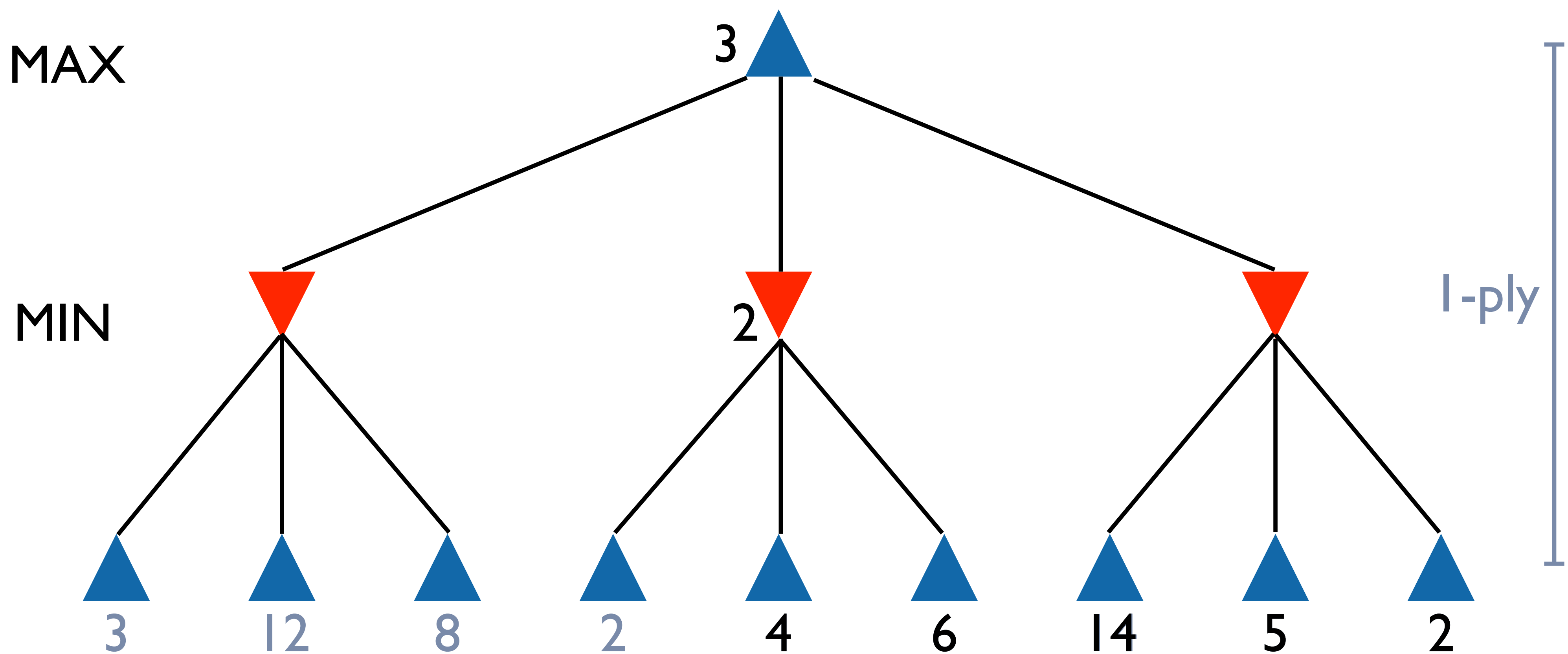
General 2-ply game tree

- Imaginary game, made-up values for example here



General 2-ply game tree

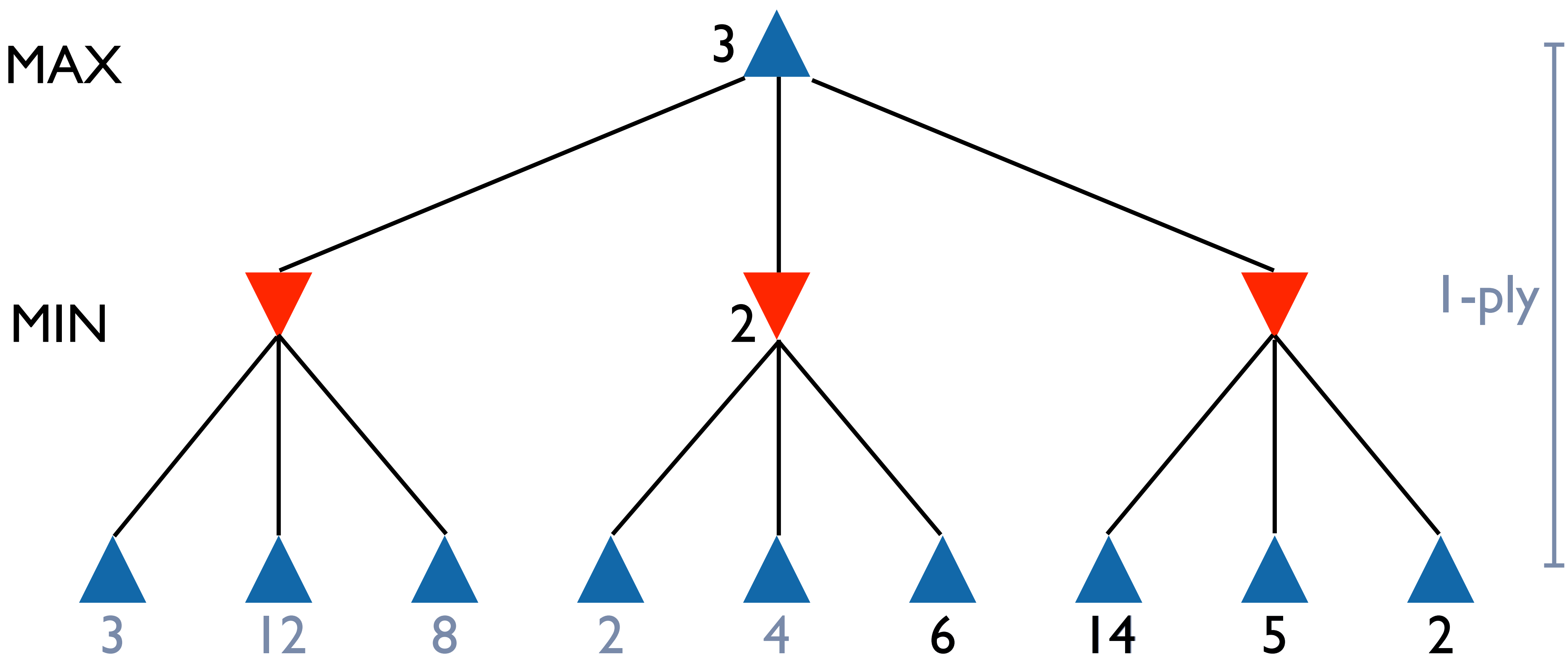
- Imaginary game, made-up values for example here



General 2-ply game tree

Queen Mary
University of London

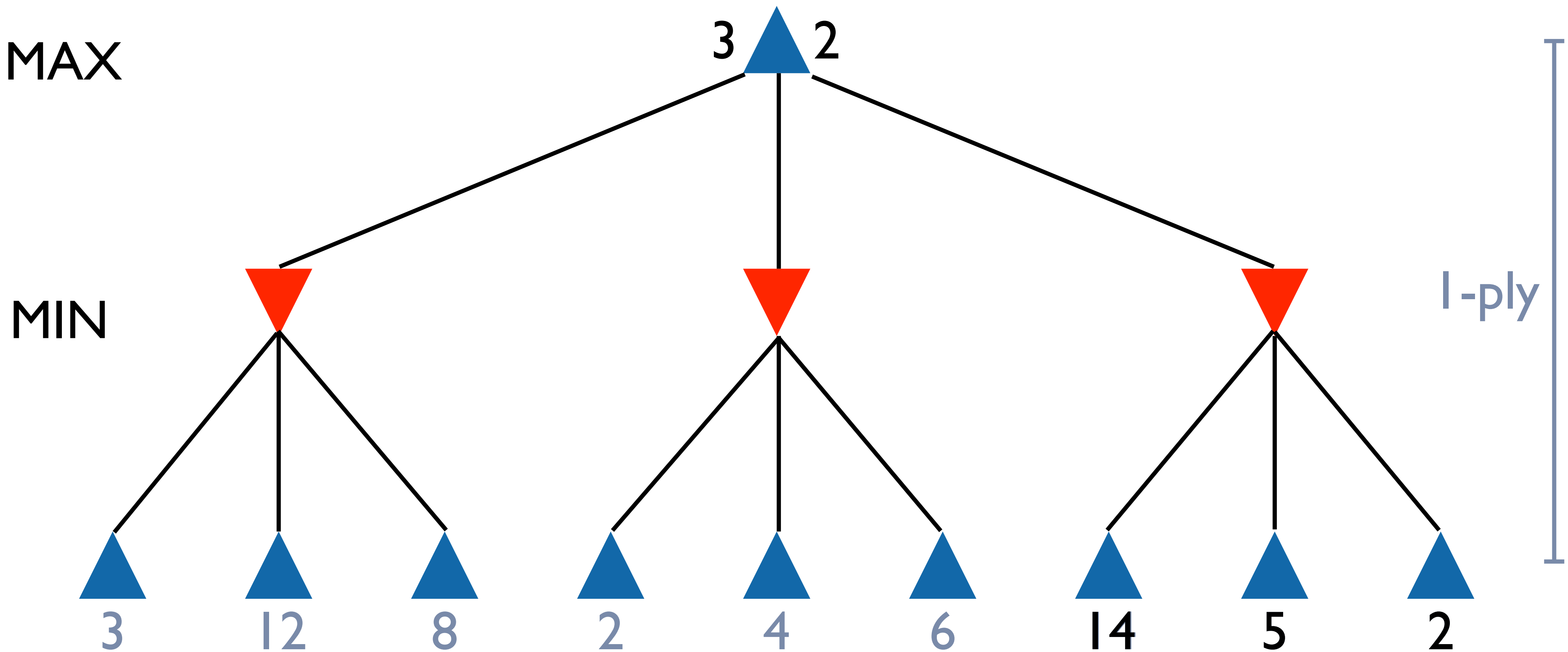
- Imaginary game, made-up values for example here



General 2-ply game tree

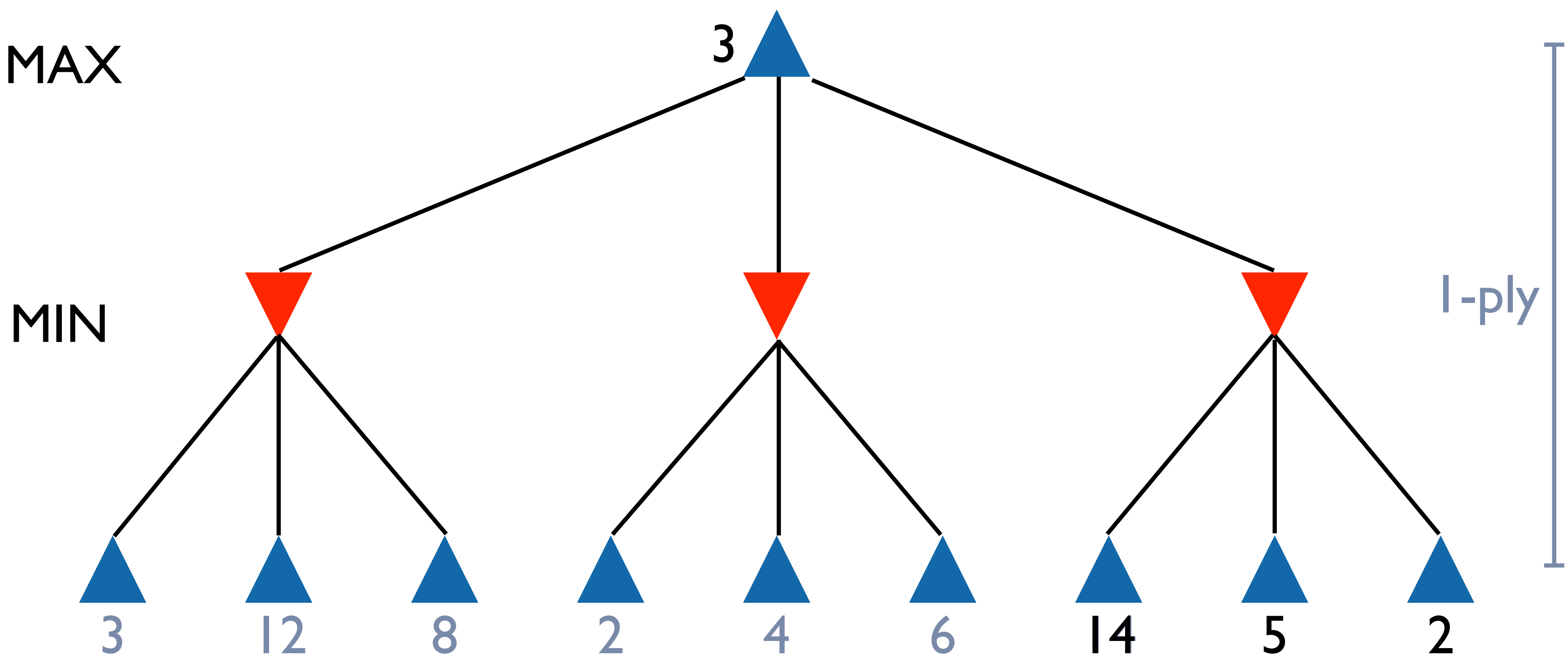
Queen Mary University of London

- Imaginary game, made-up values for example here



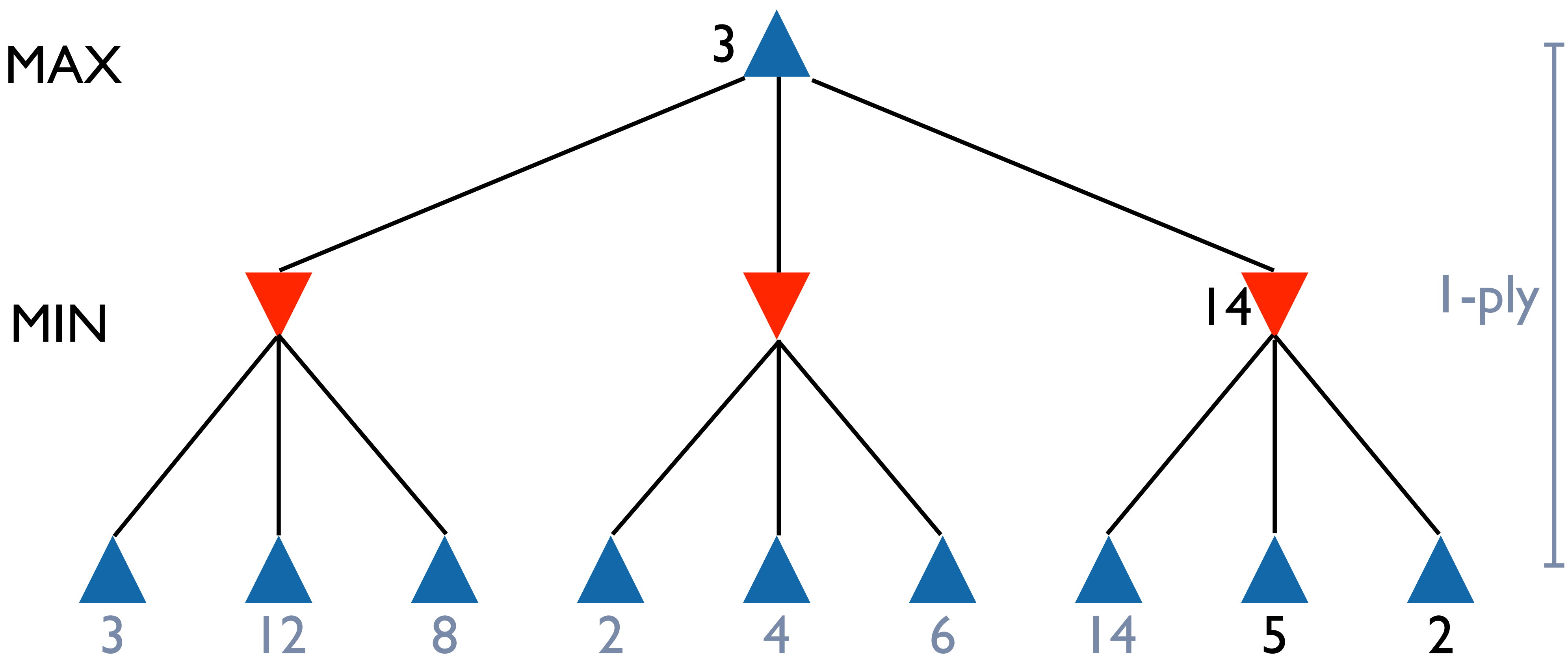
General 2-ply game tree

- Imaginary game, made-up values for example here



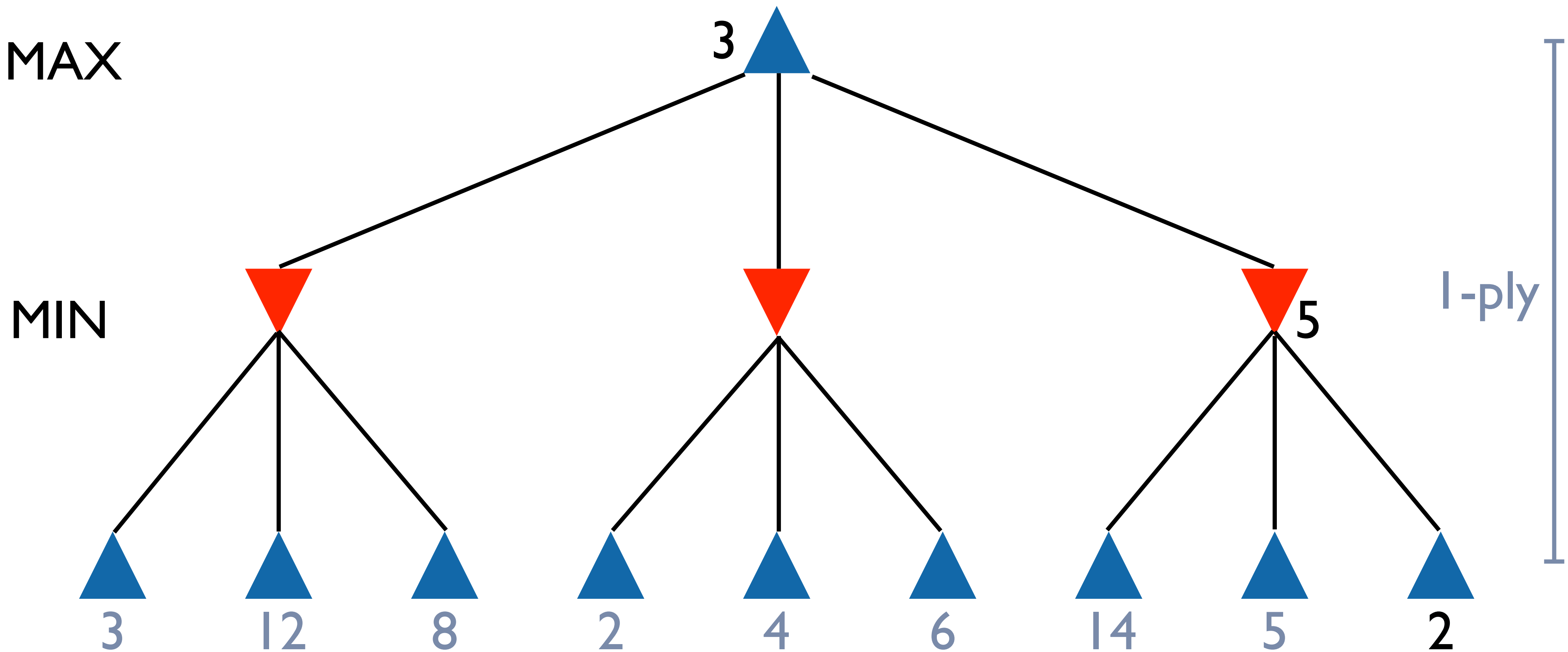
General 2-ply game tree

- Imaginary game, made-up values for example here



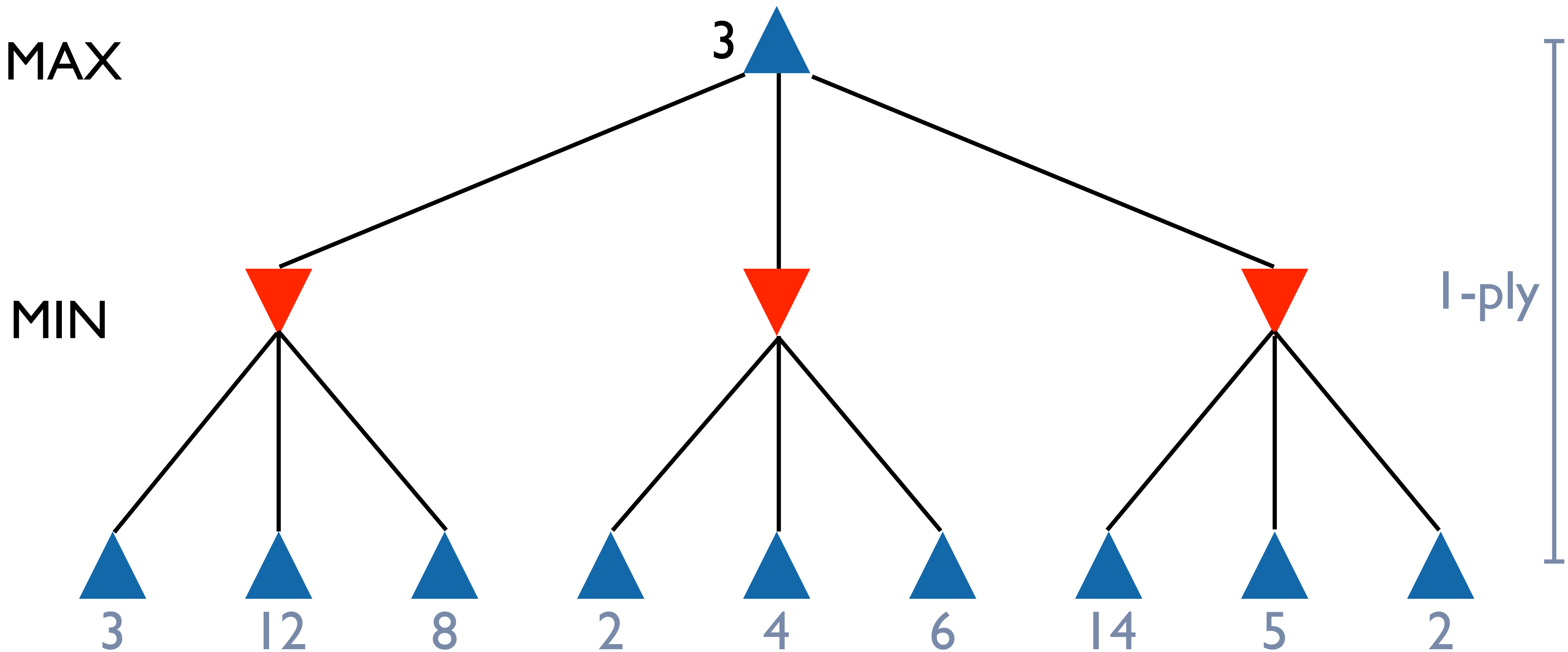
General 2-ply game tree

- Imaginary game, made-up values for example here



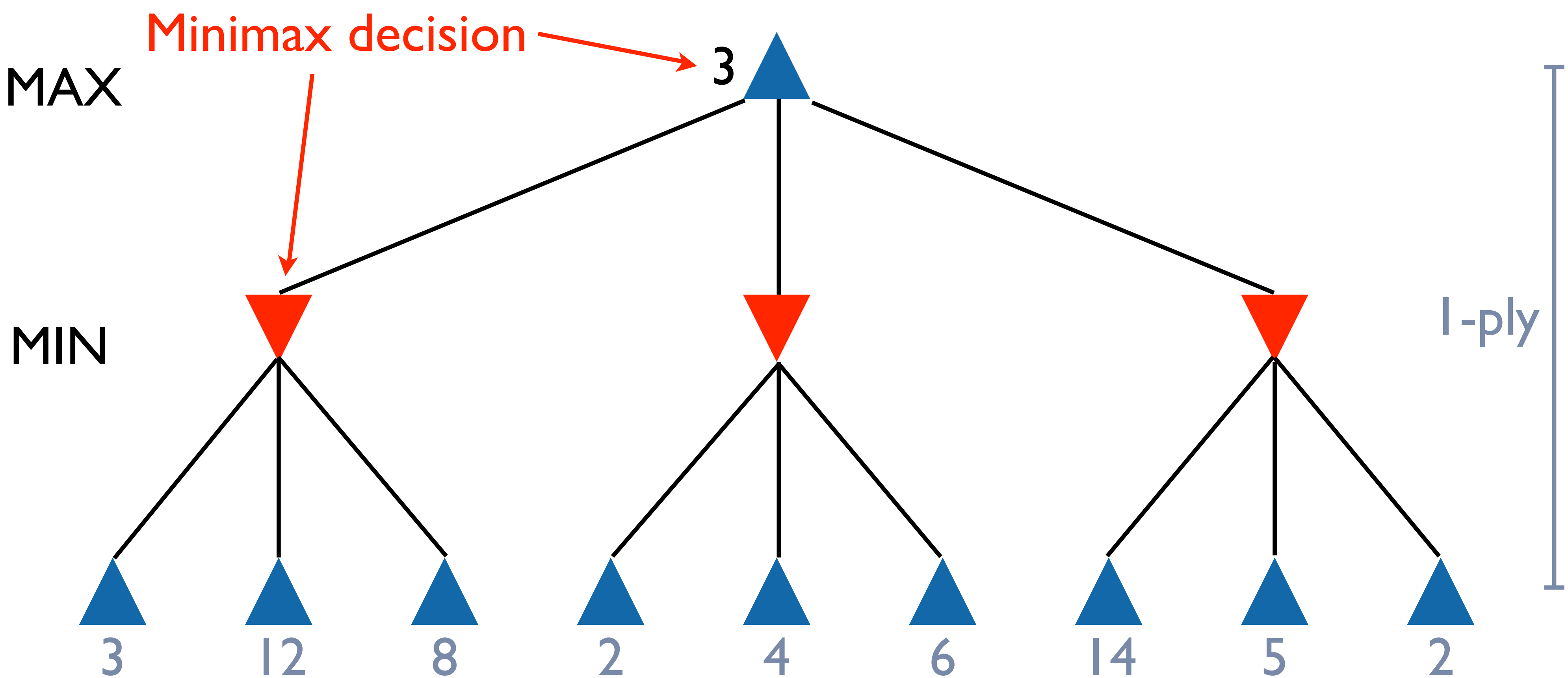
General 2-ply game tree

- Imaginary game, made-up values for example here



General 2-ply game tree

- Imaginary game, made-up values for example here
- *Algorithm maximises the worst-case outcome for MAX*



MINIMAX algorithm

Queen Mary
University of London

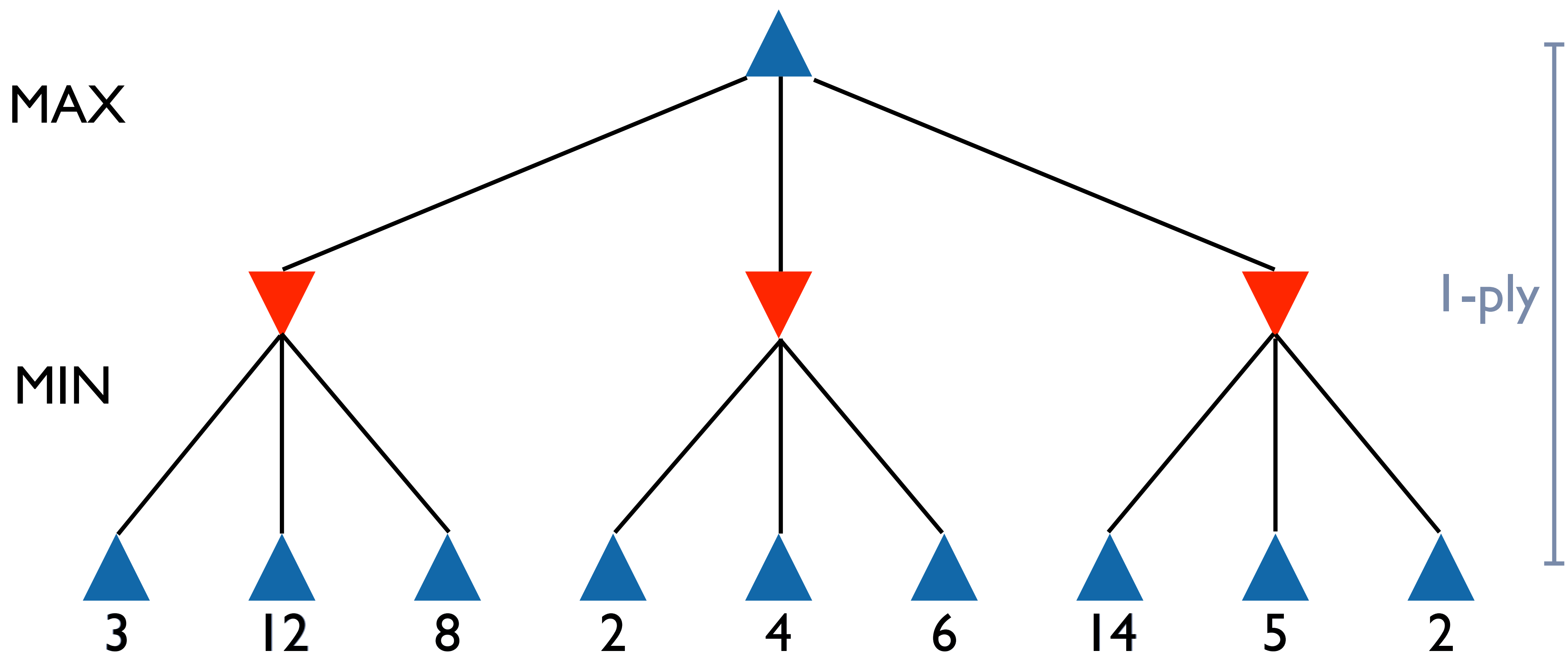
```
function MINIMAX-MaxPlayer(state) returns an (action, utility) pair
    input: state - current state of game
if terminal-Test(state) then return (null, utility(state))
best := (null, -∞)
for a in actions(state) do        // actions() returns all legal moves from a state
    value := MINIMAX-MinPlayer(makeMove(state, a)).utility
    if value > best.utility then  best := (a, value)
return best
```

```
function MINIMAX-MinPlayer(state) returns an (action, utility) pair
    input: state - current state of game
if terminal-Test(state) then return (null, utility(state))
best := (null, +∞)
for a in actions(state) do
    value := MINIMAX-MaxPlayer(makeMove(state, a)).utility
    if value < best.utility then  best := (a, value)
return best
```

More about MINIMAX

- Definition of optimal play for MAX assumes MIN plays optimally
  - maximises worst-case outcome for MAX
- But what happens if MIN plays worse than optimally?
  - MAX will do even better
- Complexity:
  - Time: $O(b^m)$ - exponential in depth of tree
  - Space: $O(m)$ - linear in depth of tree
- Can improve effective complexity by Alpha-Beta (α-β) Pruning

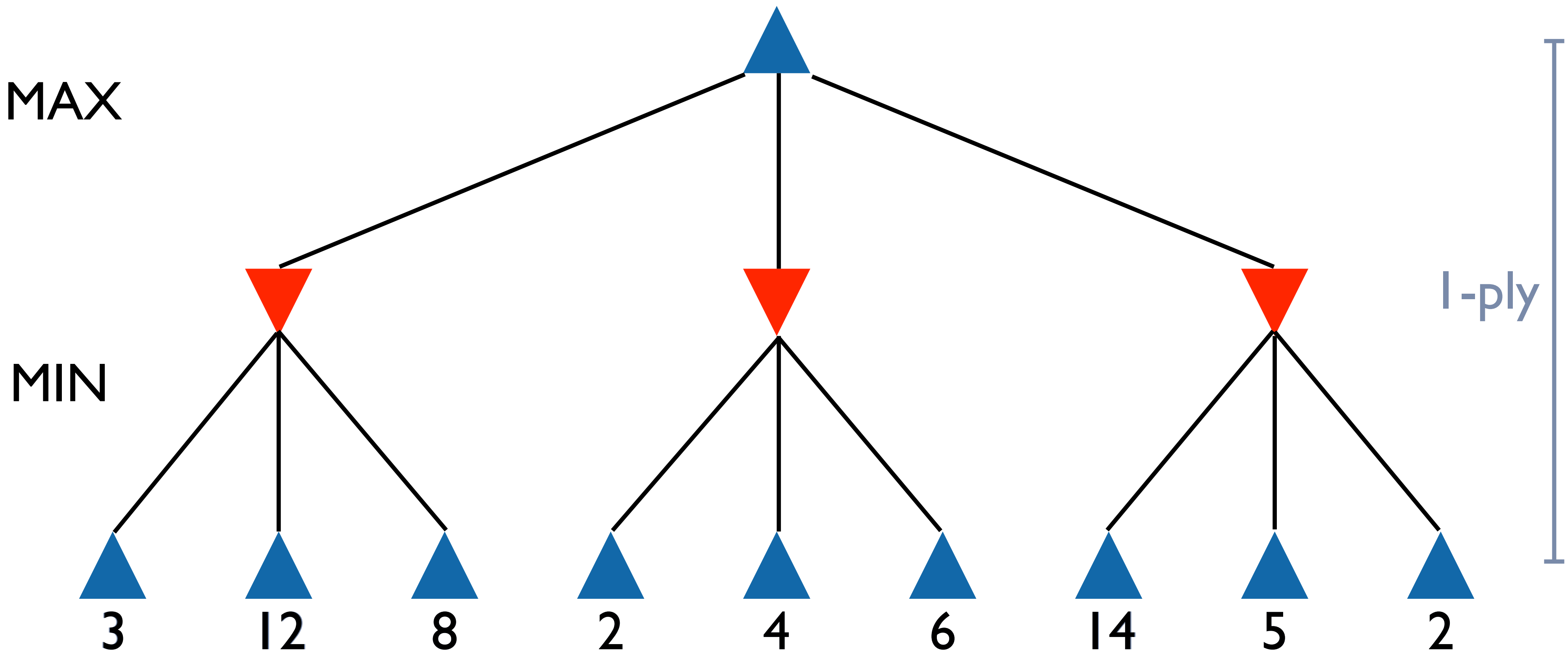
Alpha-beta pruning in MINIMAX



Alpha-beta pruning in MINIMAX

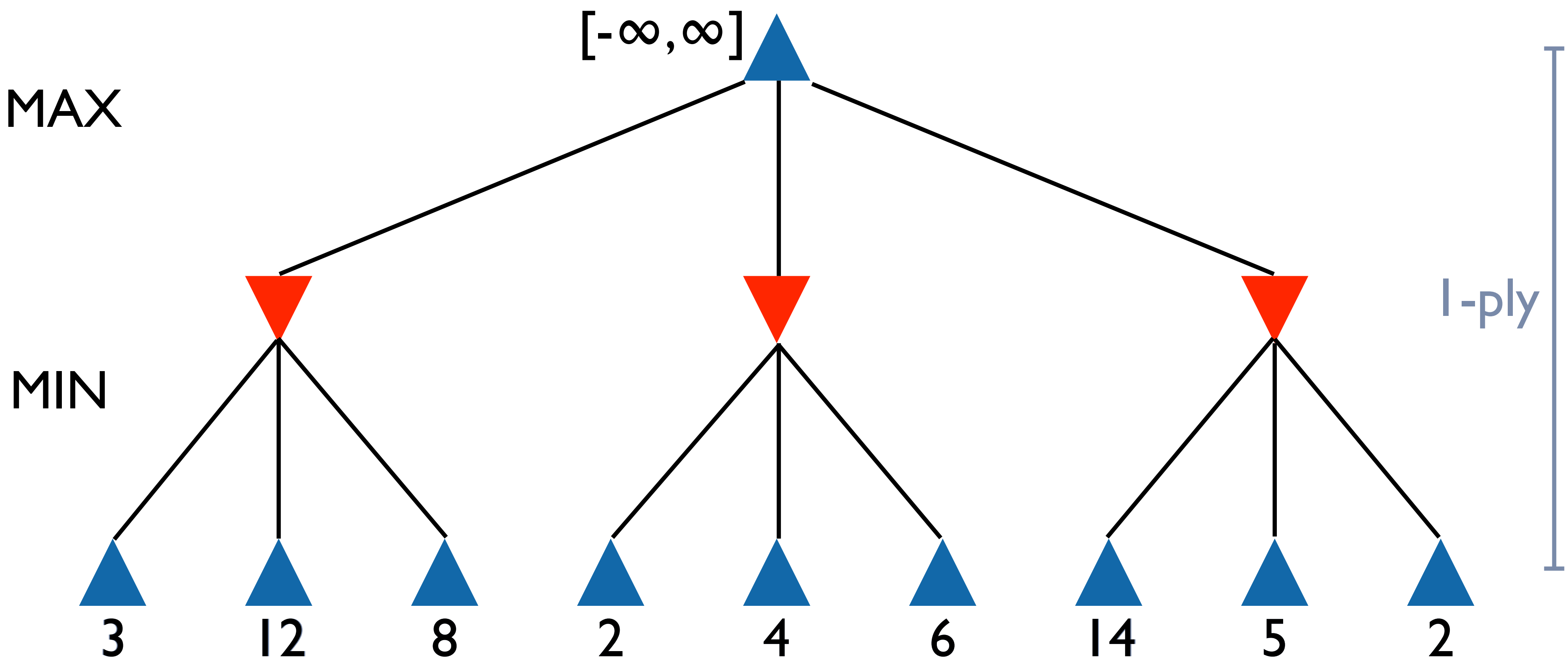
Queen Mary
University of London

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



Alpha-beta pruning in MINIMAX

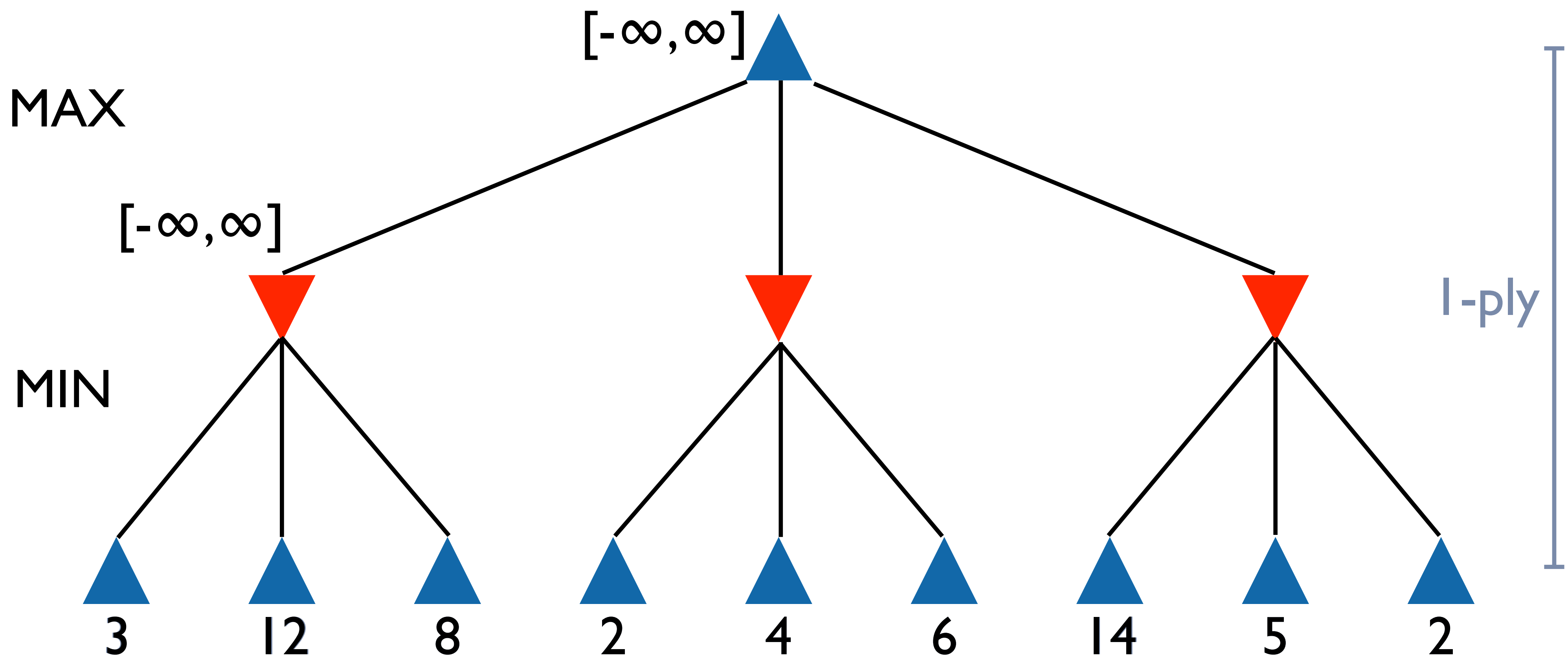
- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



Alpha-beta pruning in MINIMAX

Queen Mary University of London

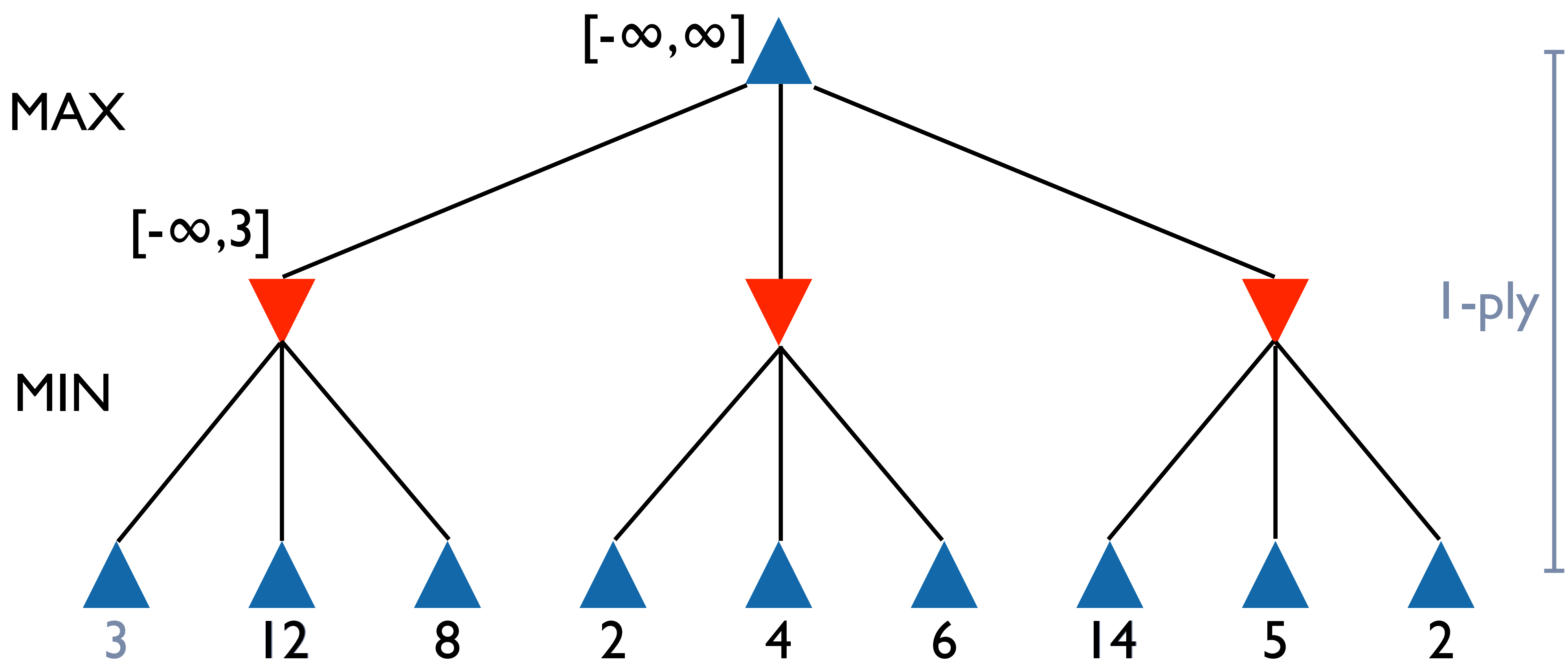
- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α,β**]



Alpha-beta pruning in MINIMAX

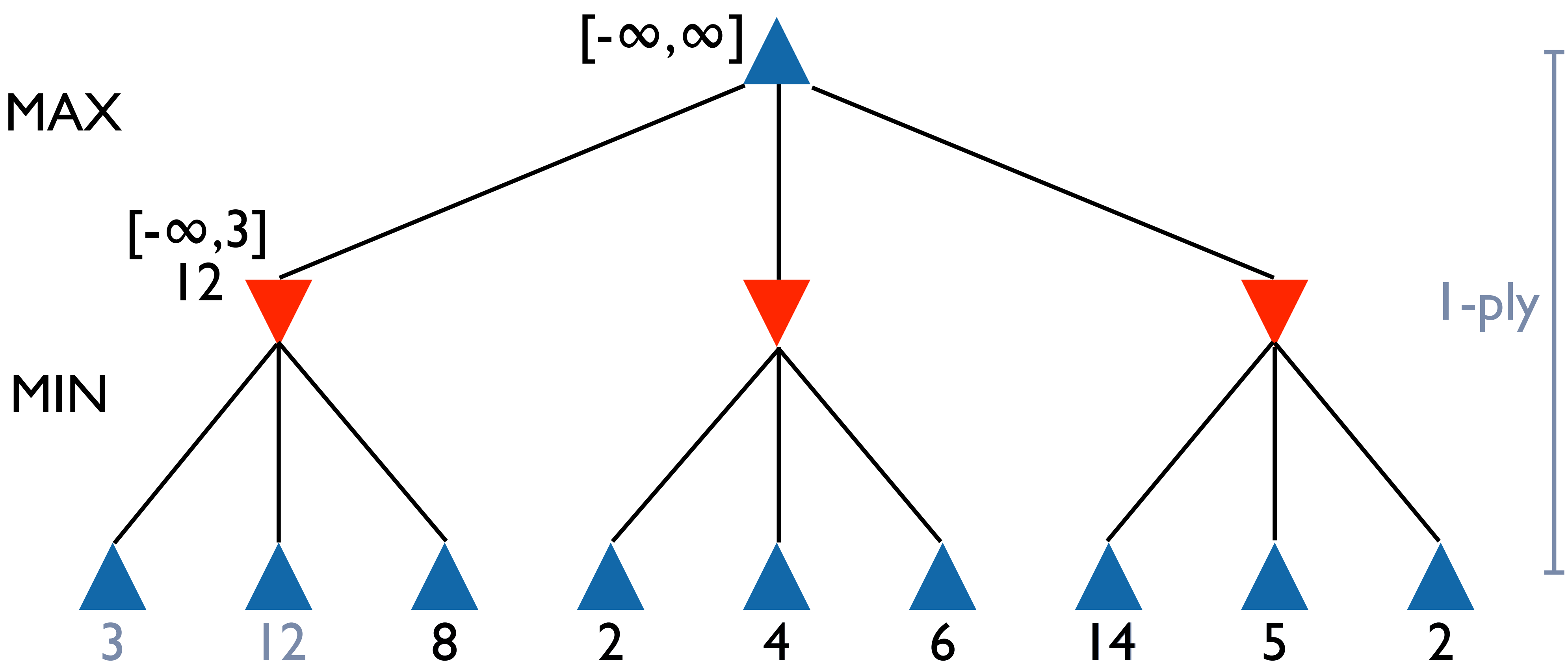
Queen Mary
University of London

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



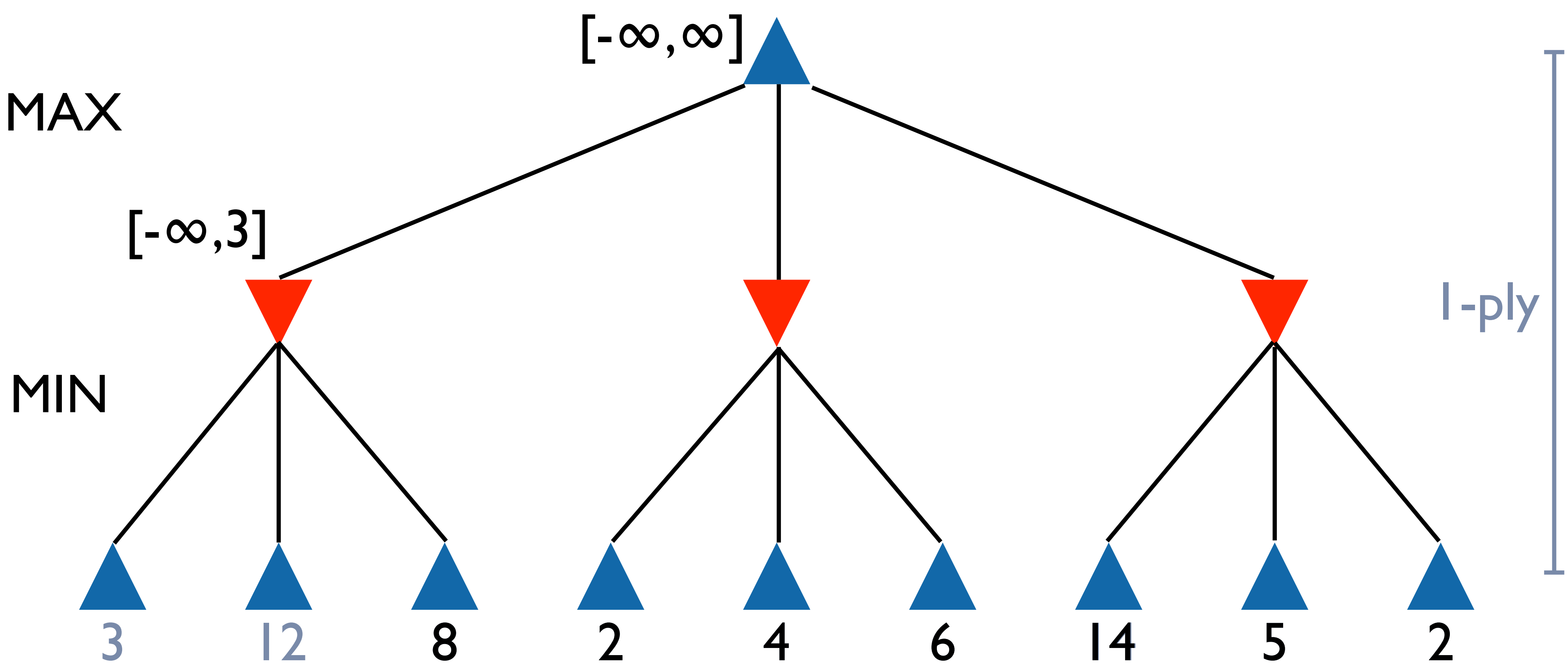
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α,β**]



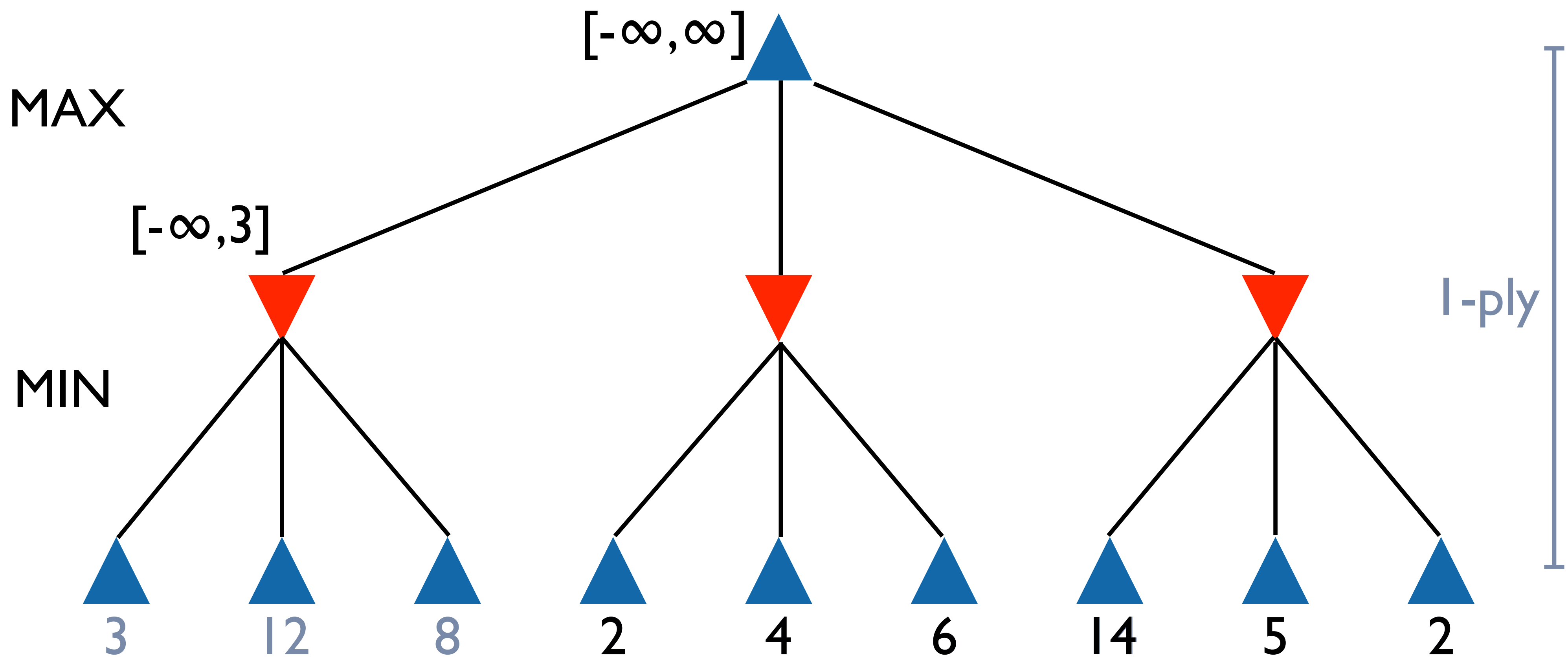
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



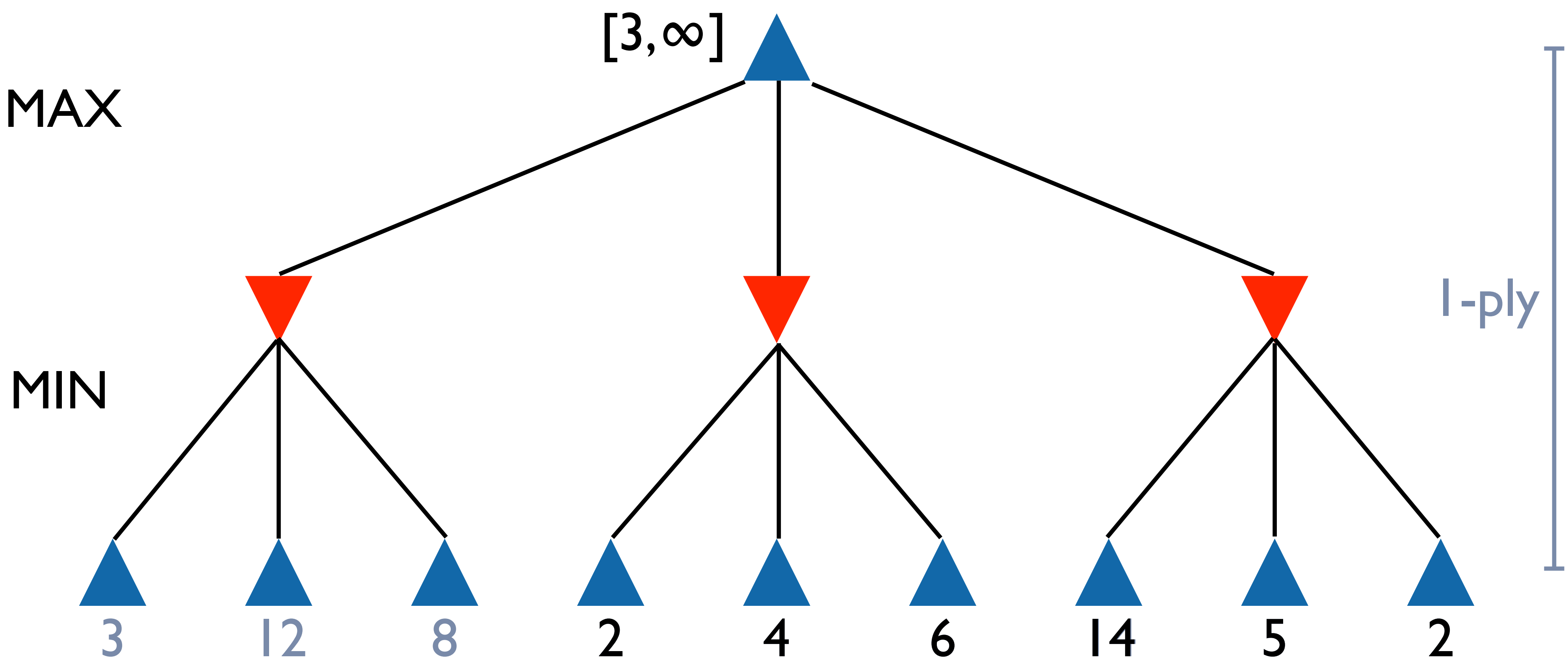
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



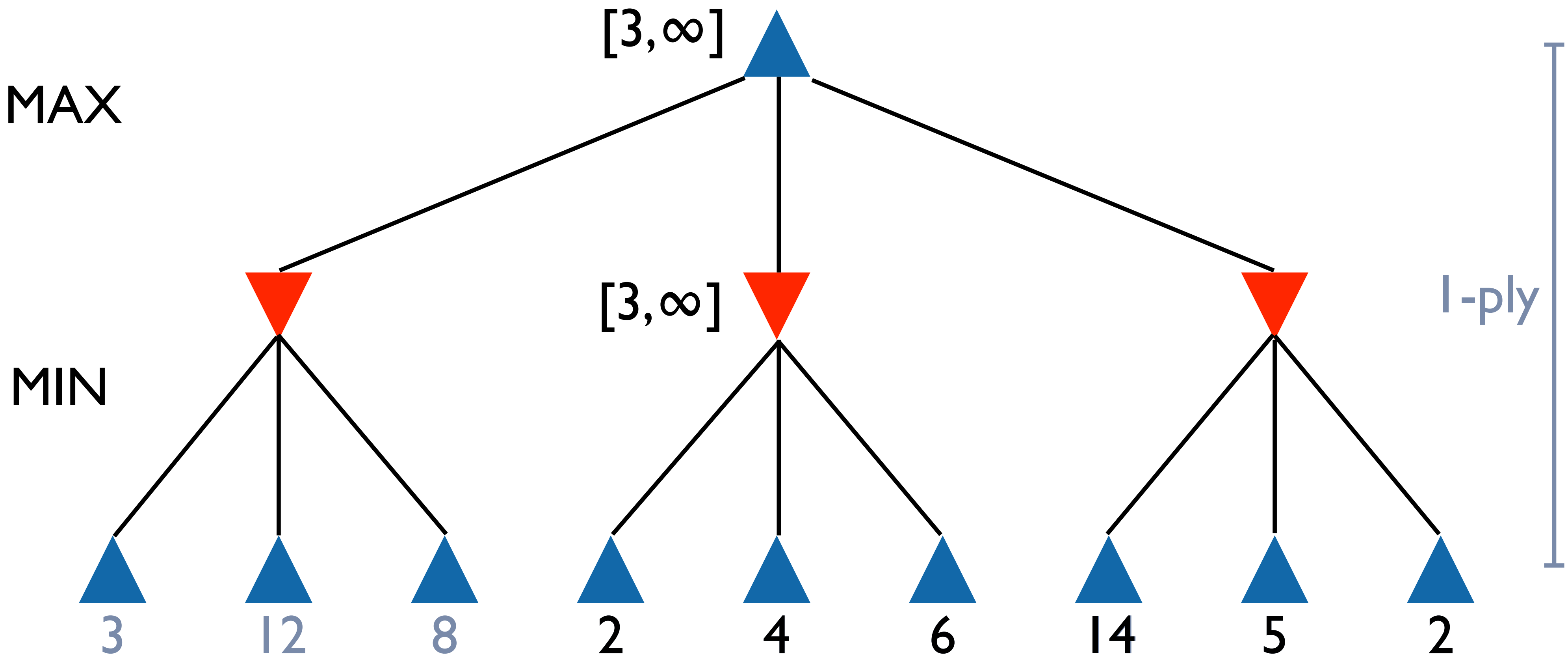
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



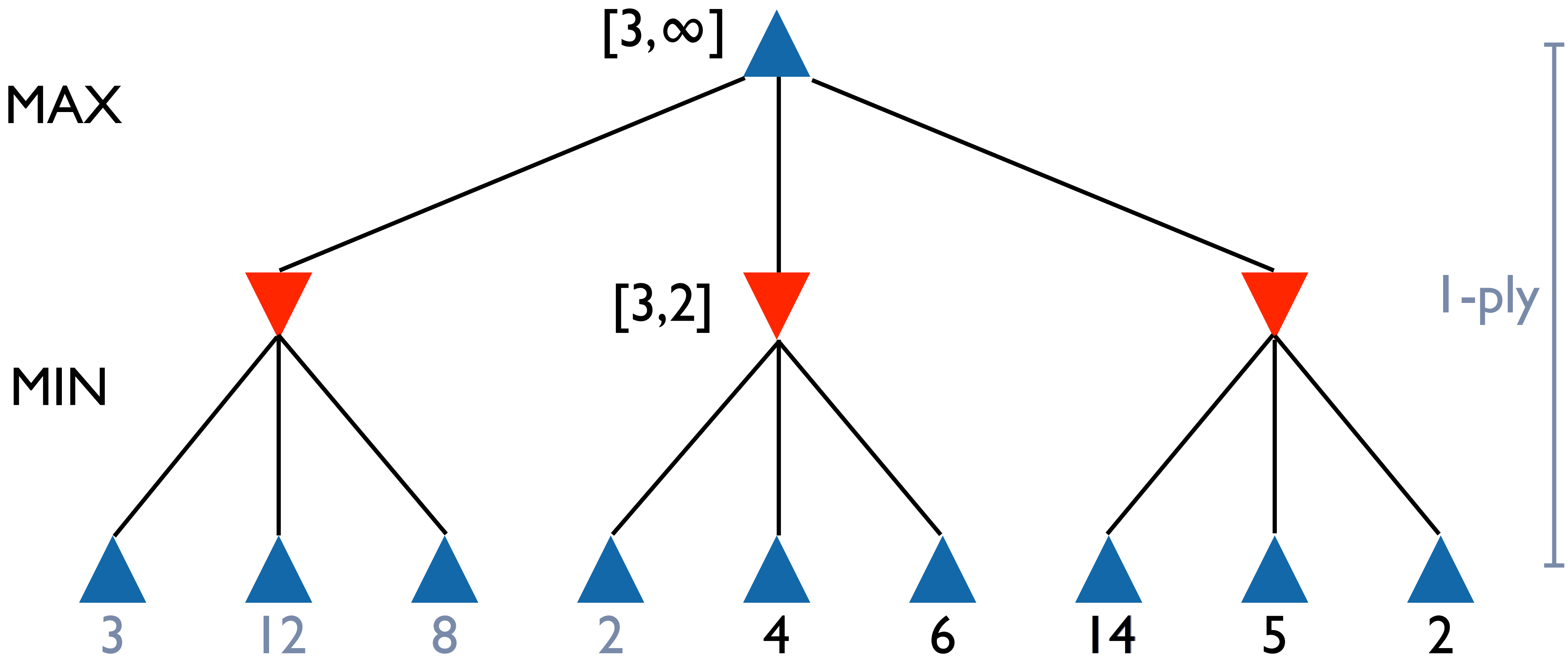
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or $[\alpha, \beta]$



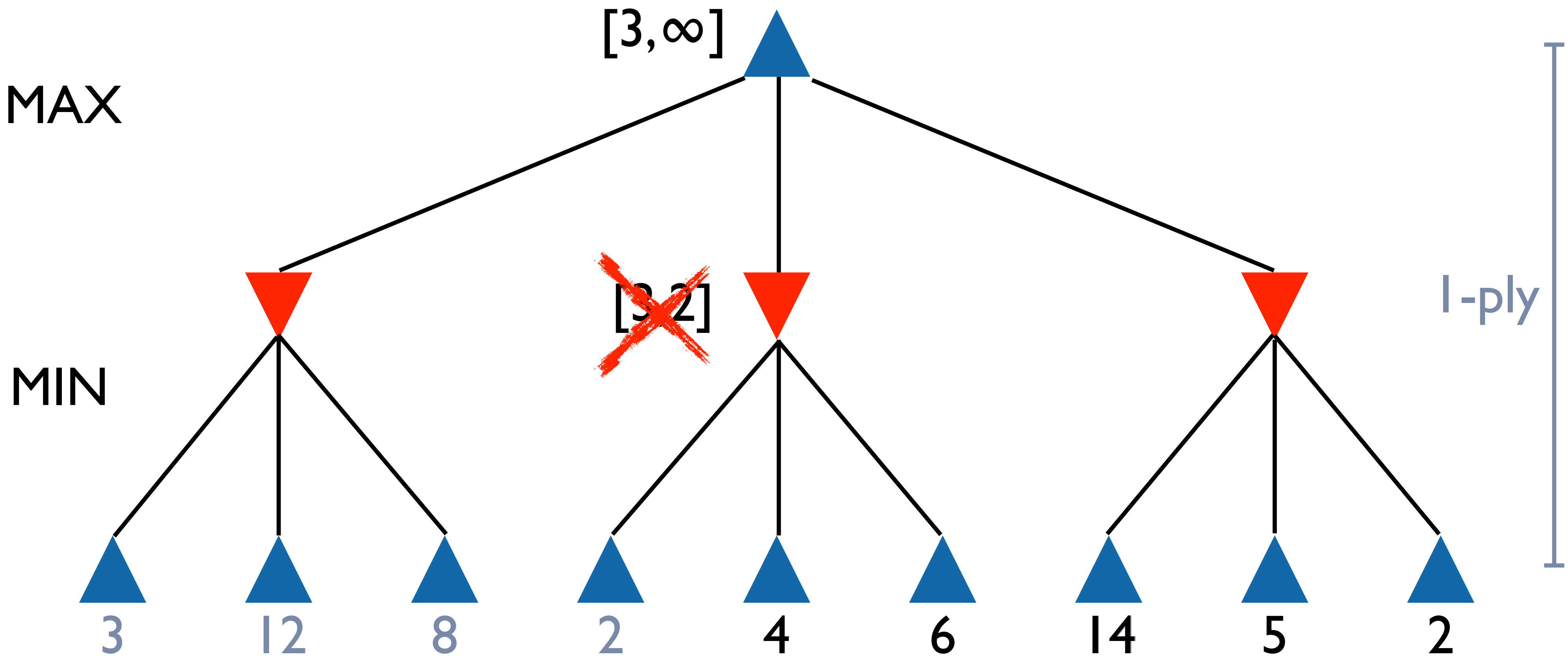
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



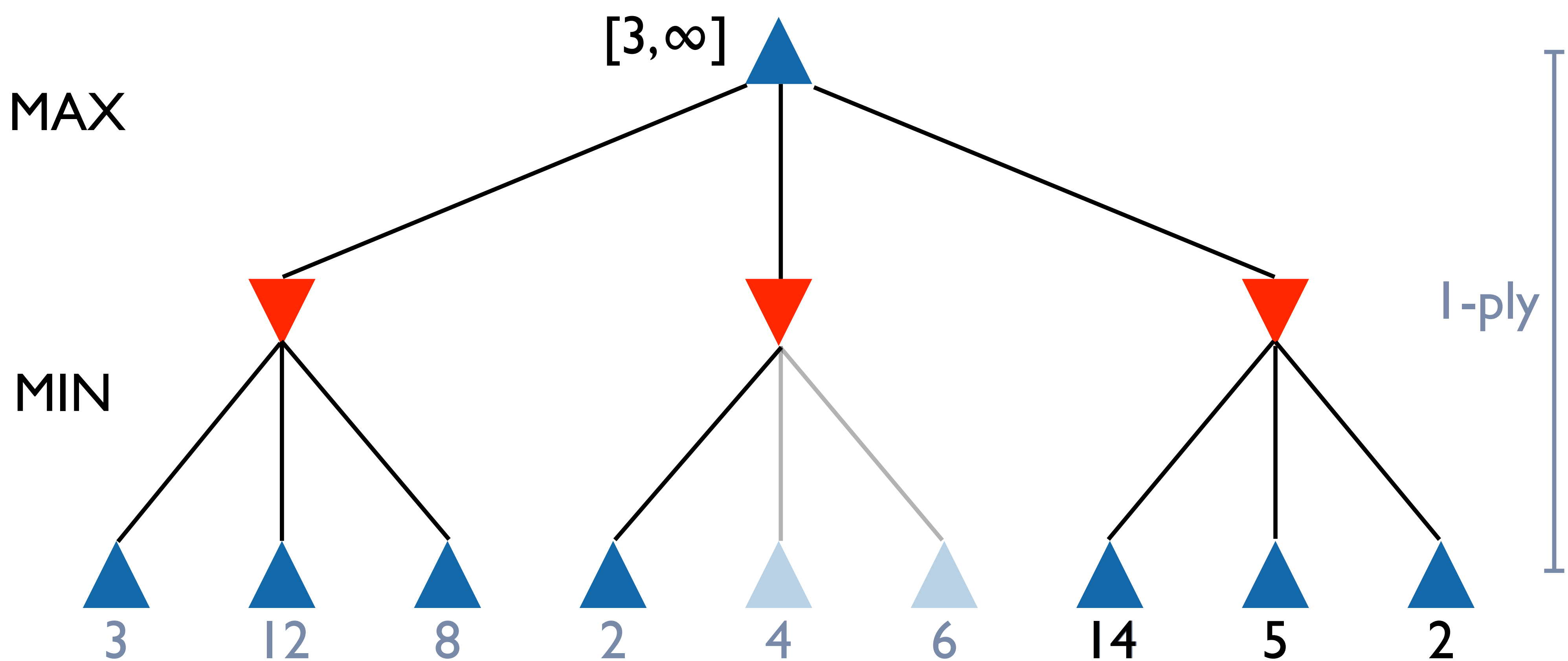
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



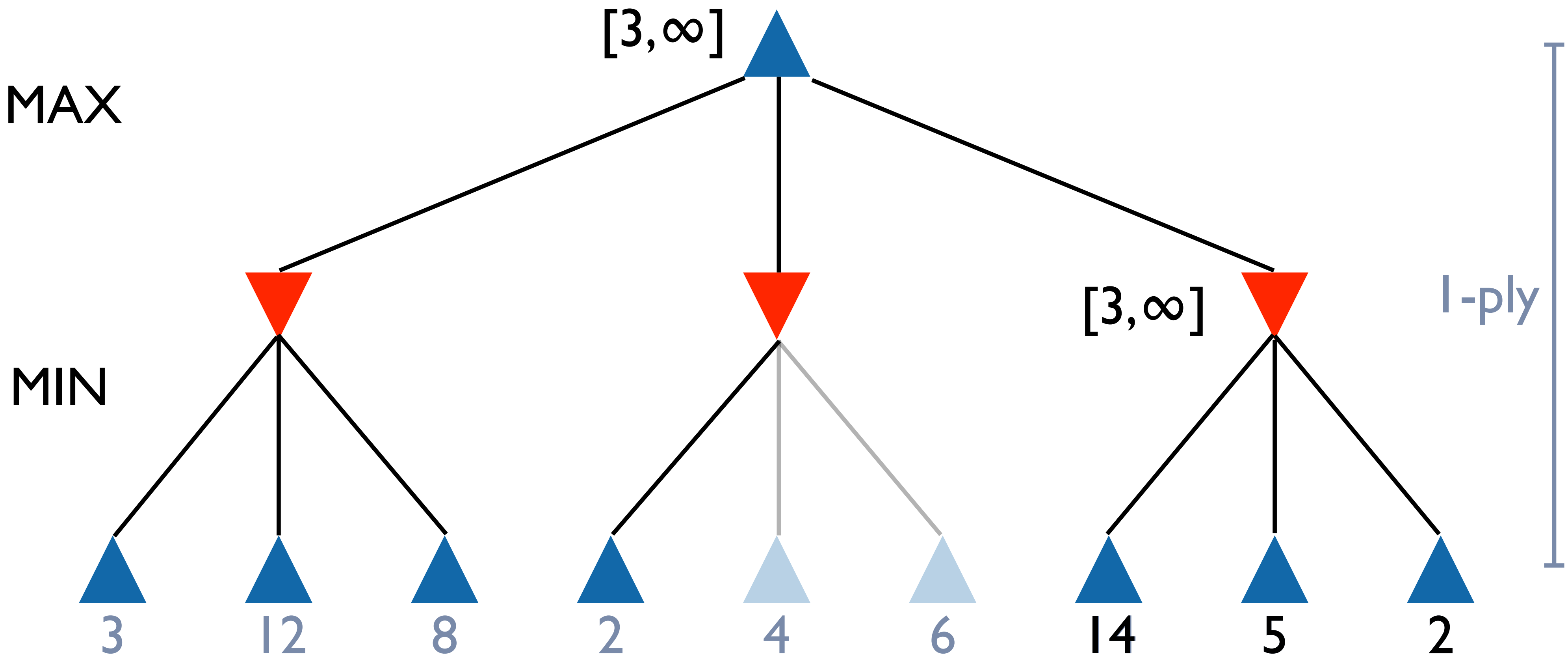
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



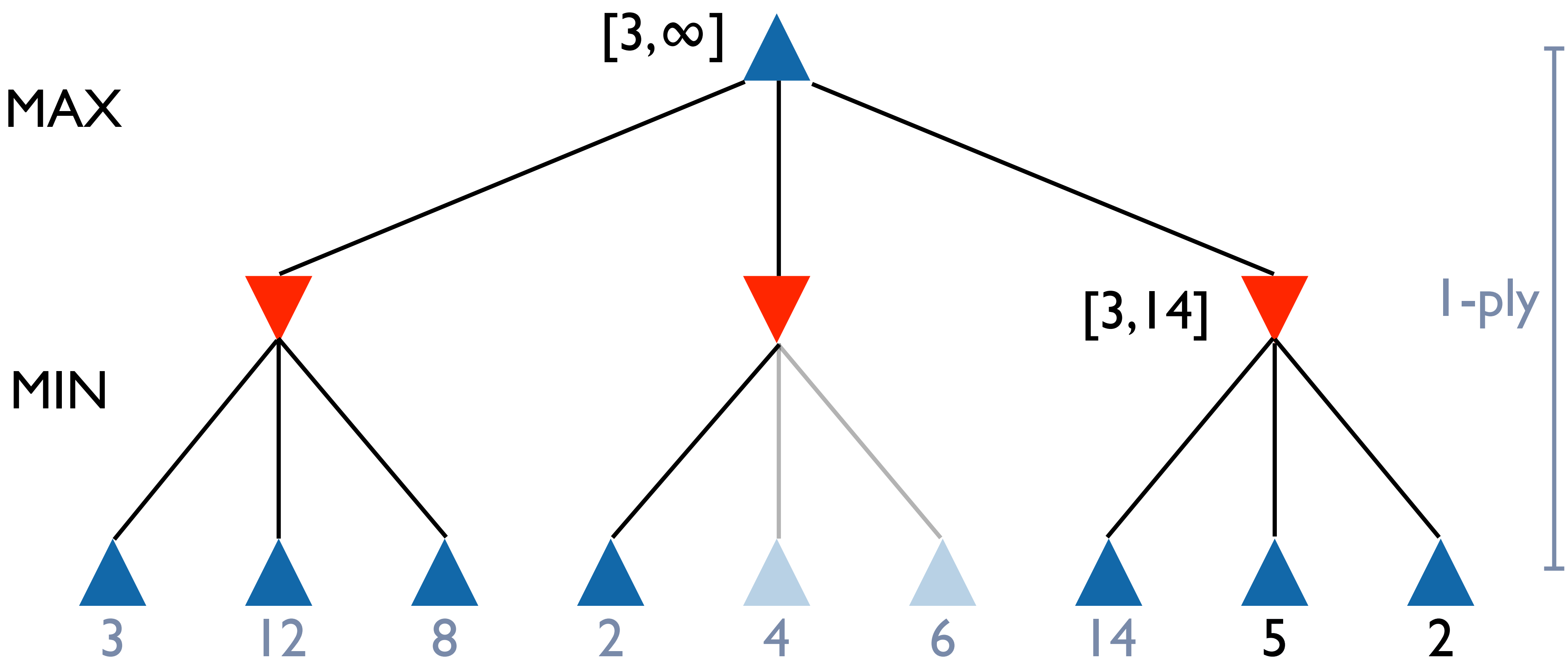
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



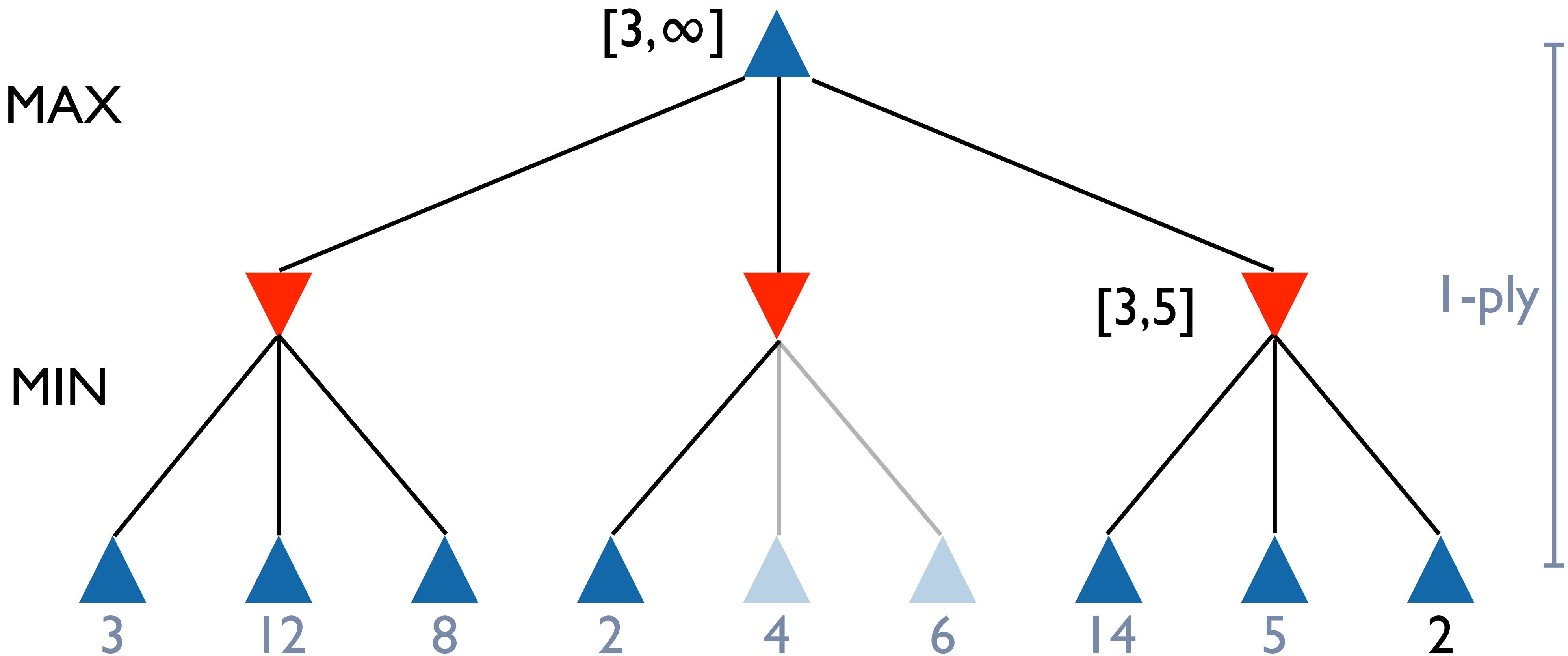
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



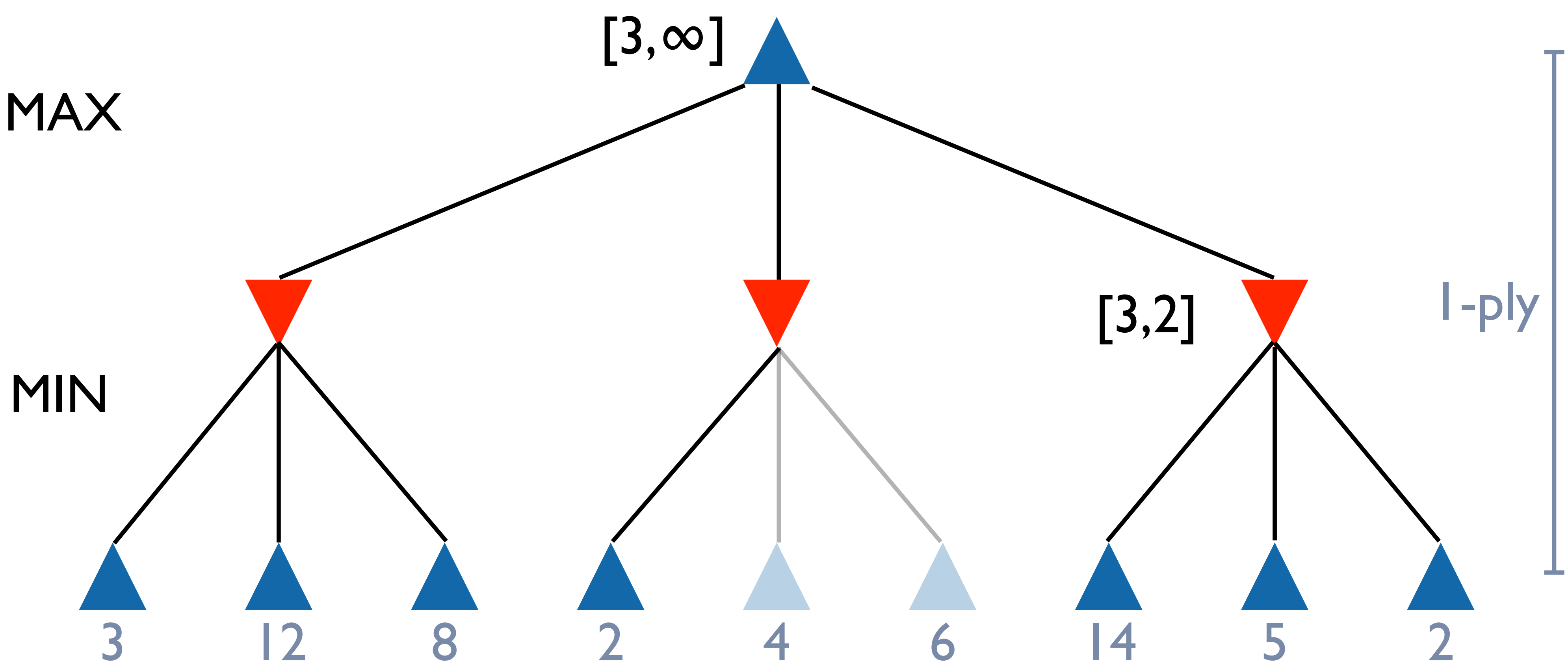
Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



Alpha-beta pruning in MINIMAX

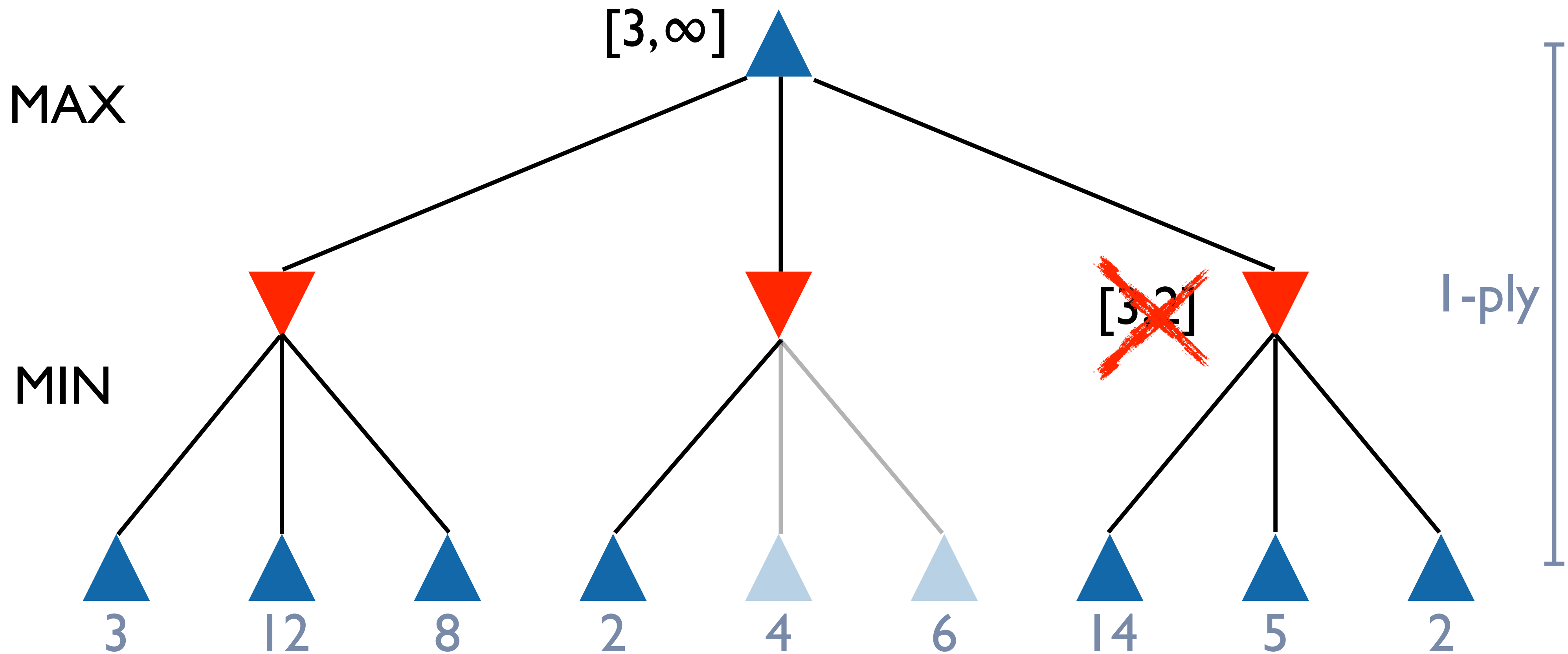
- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



Alpha-beta pruning in MINIMAX

Queen Mary
University of London

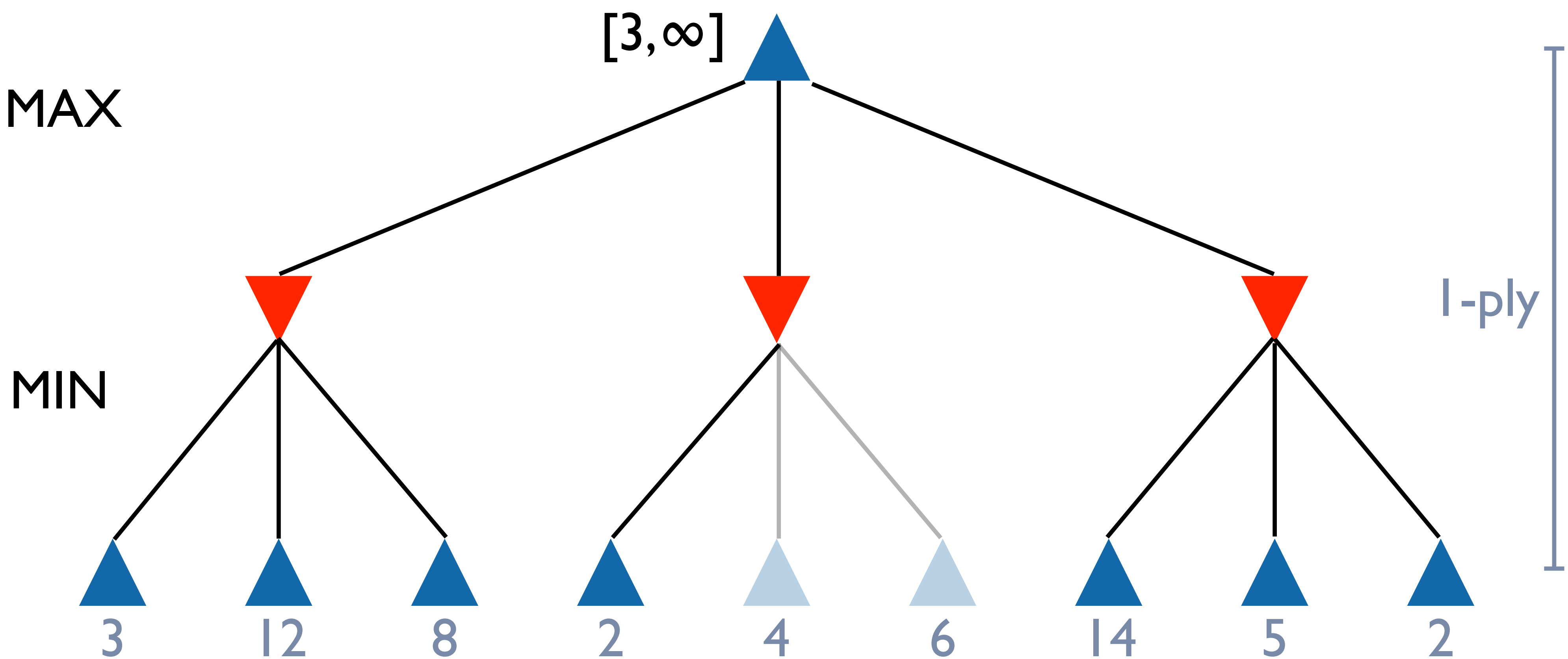
- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



Alpha-beta pruning in MINIMAX

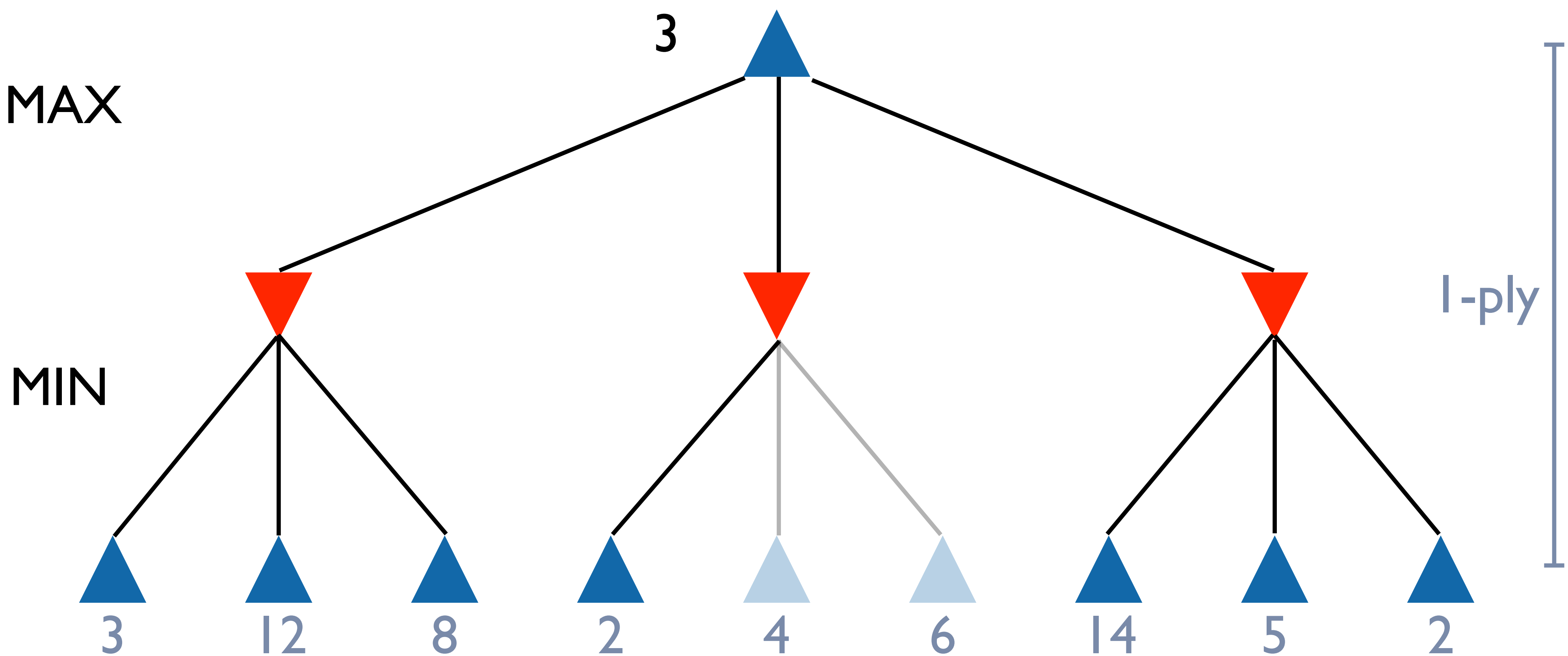
Queen Mary
University of London

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [α,β]



Alpha-beta pruning in MINIMAX

- Use the same algorithm as before, but consider *ranges* instead of values
  - range is expressed as a pair, [lowest,highest] or [**α**,**β**]



Alpha-beta algorithm

Queen Mary University of London

Initial call to the algorithm is: ALPHA-BETA-MaxPlayer (*state*, -∞, ∞)

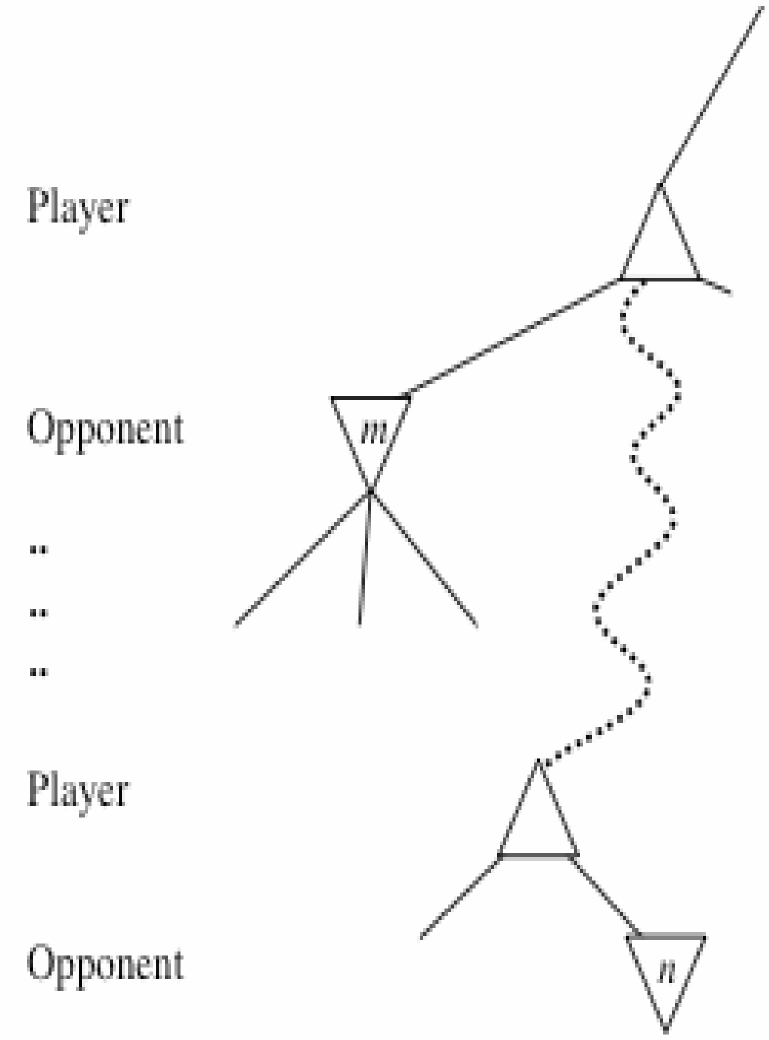
function ALPHA-BETA-MaxPlayer(*state*, **α**, **β**) **returns** an (*action*, *utility*) pair
  input: *state* - current state of game
if terminal-Test(*state*) **then return** (null, utility(*state*))
best := (null, -∞)
for *a* **in** actions(*state*) **do**
  value := ALPHA-BETA-MinPlayer(makeMove(*state*, *a*), **α**, **β**).utility
  if *value* > *best.utility* **then** *best* := (*a*, *value*)
  if *value* >= **β** **then** **return** (null, *value*)
  if *value* > **α** **then** **α** := *value*
return *best*

Alpha-beta algorithm

function ALPHA-BETA-MinPlayer(*state*, **α**, **β**) **returns** an (*action*, *utility*) pair
    input: *state* - current state of game
if terminal-Test(*state*) **then return** (null, utility(*state*))
best := (null, ∞)
for *a* **in** actions(*state*) **do**
    value := ALPHA-BETA-MaxPlayer(makeMove(*state*, *a*), **α**, **β**).utility
    if *value* < *best.utility* **then** *best* := (*a*, *value*)
    if *value* <= **α then return** (null, *value*)
    if *value* < **β then β** := *value*
return *best*

General alpha-beta pruning

- Consider a node *n* somewhere in the tree
- If player has a better choice at
  - Parent node of *n*
  - Or any choice point further up
- n will never be reached in actual play
- Hence when enough is known about *n*, it can be pruned.



Properties of alpha-beta pruning

- Pruning does not affect final results (directly)
- Entire subtrees can be pruned
- Good move ordering improves effectiveness of pruning
- With “perfect ordering”, time complexity is $O(b^{m/2})=O((b^{1/2})^m)$
  - Branching factor of $\sqrt{b}$
  - Best case alpha-beta pruning can look *twice as far* ahead as plain MINIMAX in a similar amount of time
- Repeated states are again possible
  - Store them in memory; make large gain in memory from pruning

Incomplete game trees

Queen Mary University of London

- For many games, MINIMAX and alpha-beta pruning require too many leaf-node evaluations
- Searching to terminal states will often be impossible within a reasonable amount of time
- Shannon (1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic (or *static*) evaluation function EVAL (replacing UTILITY function of MINIMAX)

Cutting off search

- Change the termination condition of MINIMAX:
  - if TERMINAL-TEST(state) then return UTILITY(state)

  into

  - if CUTOFF-TEST(state,depth) then return EVAL(state)
- Can introduce a fixed or dynamic depth limit
  - selected so (e.g.) time will not exceed what the rules of the game allow
- When cutoff occurs, and we are not at a terminal state, heuristic evaluation is performed
  - in other words, if you don't know, then take your best guess

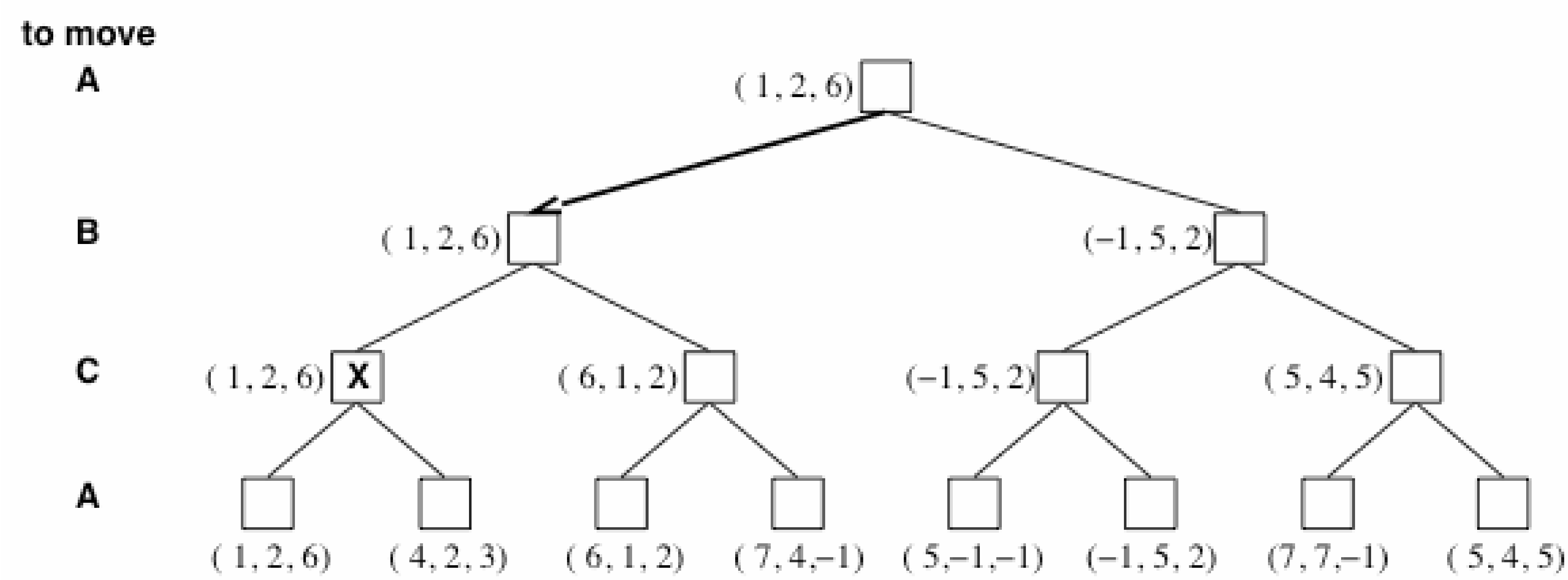
Heuristic Evaluation

Queen Mary University of London

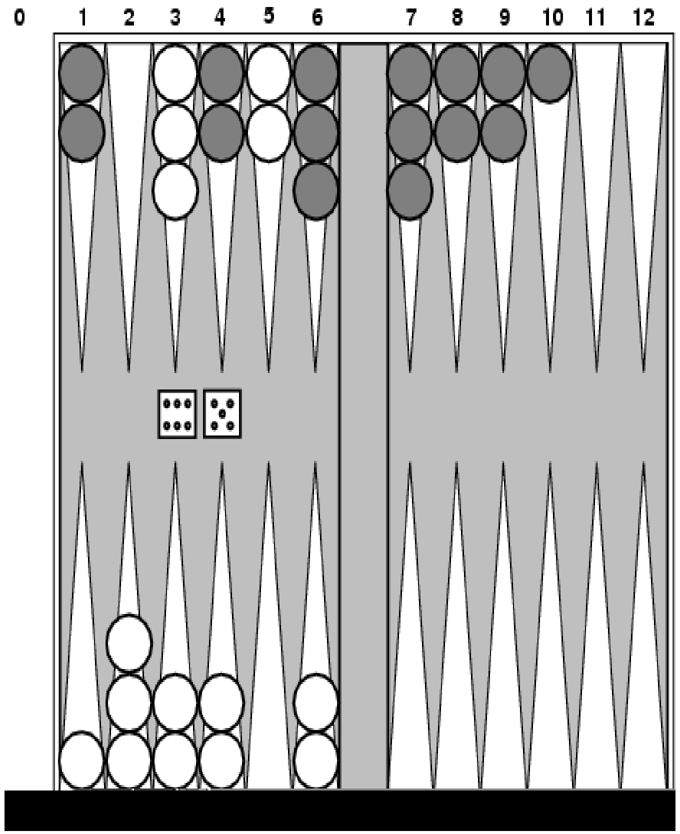
- Idea:
  - produce an estimate of the expected utility of the game from a given position
- Algorithm performance depends on quality of EVAL
- Requirements:
  - Computation may not take too long
  - EVAL should value terminal states in the same order as UTILITY
  - For non-terminal states, EVAL should be strongly correlated with the actual chance of winning
  - Ideally should only be used for *quiescent* states (=no wild swings in value in near future)

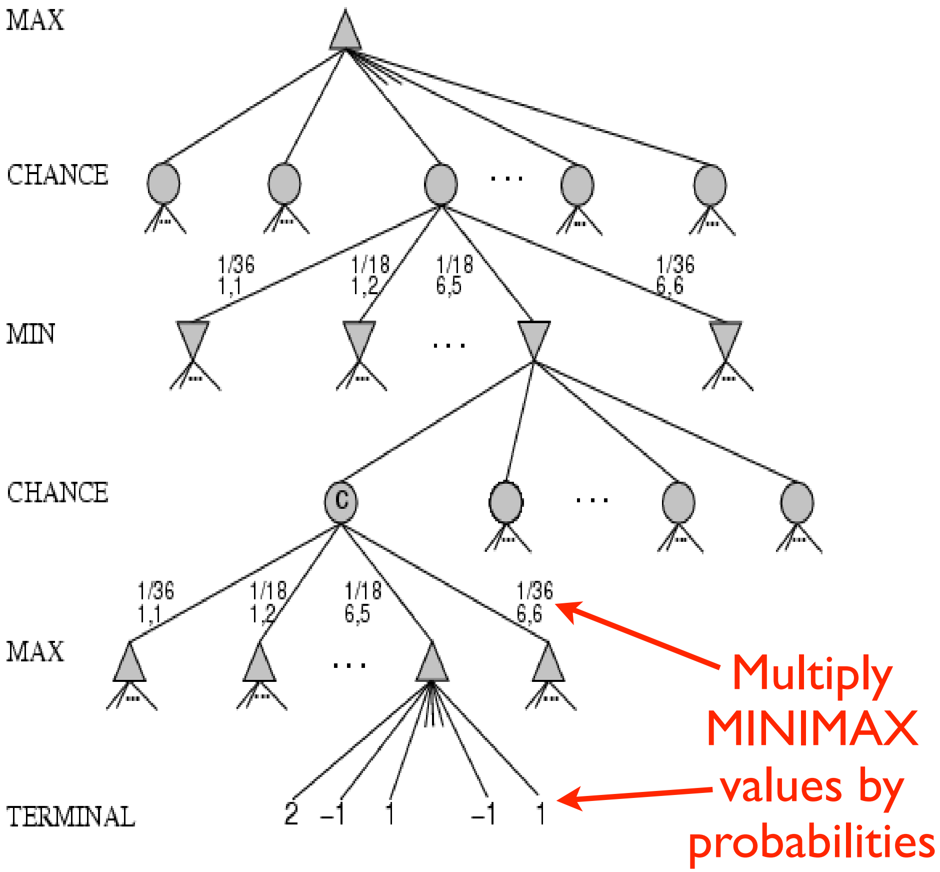
Multiplayer games

- Replace single zero-sum utility function with a function for each player
- Use a vector of values, one for each player



Games that include chance





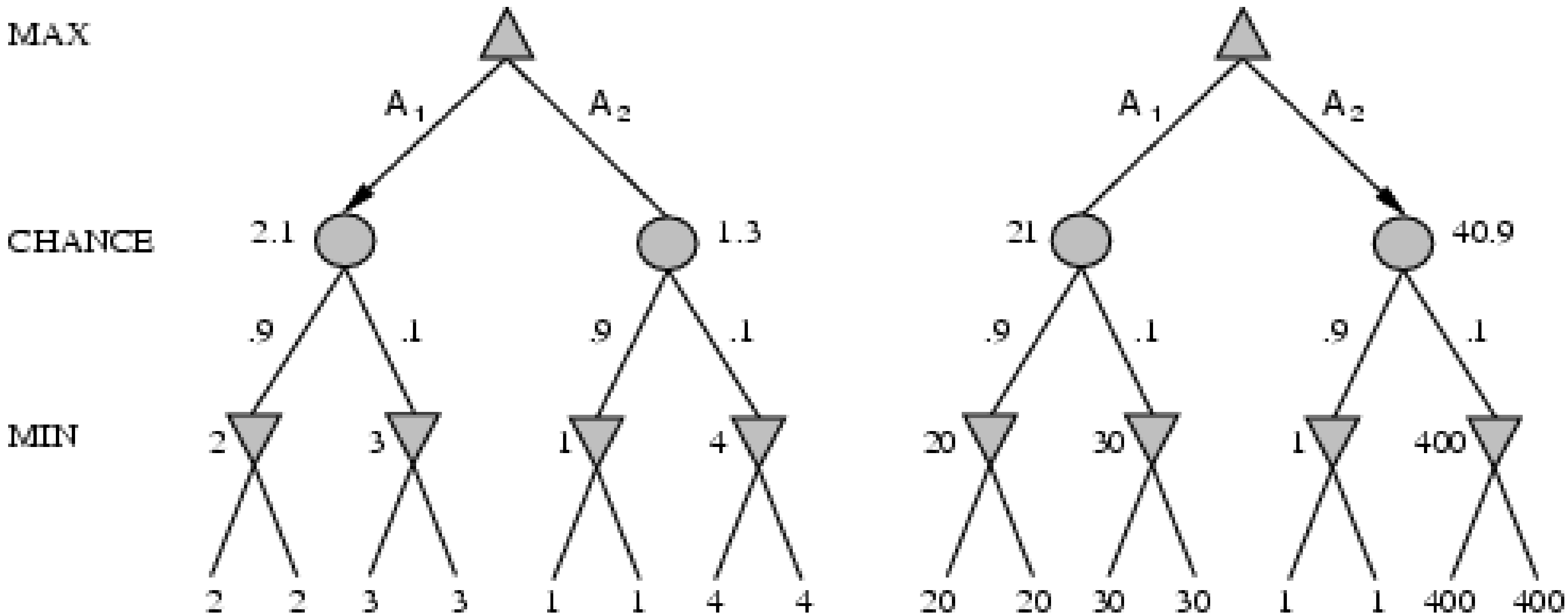
- In backgammon, you roll 2 dice, and then use both numbers in either order
  - so a double (e.g. [1,1]) has probability 1/36; all other rolls have probability 1/18
- In this tree, we calculate the *expected value* of MINIMAX

Expected MINIMAX value

- An *expected value* is a weighted average, where weights are probabilities
- Given a game tree, the optimal strategy can be determined by using the minimax value of each node, n:
  - if n is a terminal node,
    - EXPECTED-MINIMAX-VALUE(n) = UTILITY(n)
  - if n is a MAX node,
    - EXPECTED-MINIMAX-VALUE(n) = $\max_{s \in \text{successors}(n)}$ EXPECTED-MINIMAX-VALUE(s)
  - if n is a MIN node,
    - EXPECTED-MINIMAX-VALUE(n) = $\min_{s \in \text{successors}(n)}$ EXPECTED-MINIMAX-VALUE(s)
  - if n is a CHANCE node,
    - EXPECTED-MINIMAX-VALUE(n) = $\sum_{s \in \text{successors}(n)}$ P(s) EXPECTED-MINIMAX-VALUE(s)
    - where P(s) is the probability of outcome s

Position evaluation with CHANCE nodes

Queen Mary
University of London



- Note that with chance nodes included, it is not only the *order* of values that is important, but also the values themselves
- On left, A_1 is chosen; on right, A_2 is chosen
  - i.e. outcome of evaluation may change if values are scaled non-linearly

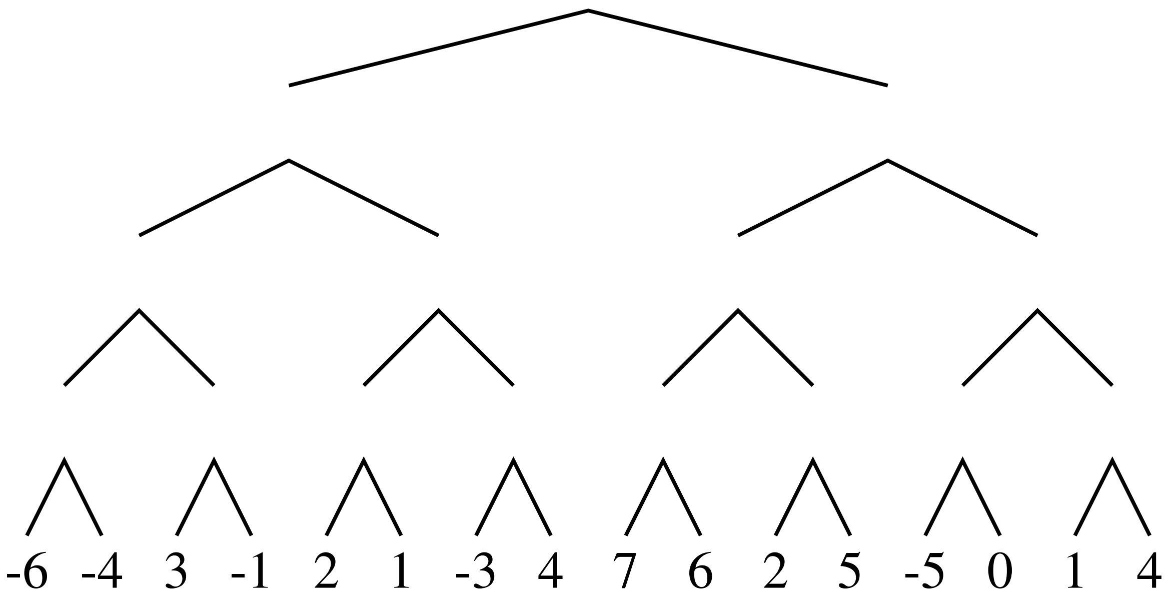
Summary

- Games are fun, useful and potentially distracting
- They illustrate many important points about AI
  - Decision-making in adverse and resource-limited situations
  - Perfection is (usually) unattainable, so approximation is necessary
  - We need a good idea of strategy to win
    - MINIMAX allows us to express that in terms of final states - relatively easy
  - The assignment of values to non-terminal states is still a hard problem
- Games are to AI as grand prix racing is to car design

Past **Exam**ple Question

Queen Mary
University of London

- *A two-player game has the following game tree for the last four moves. MAX tries to find the goal with the largest value, and MIN the smallest one. It is MAX's turn to play.*



- *Use the minimax algorithm to calculate the value of each node of the tree. (You will need to copy the whole tree into your answer book.)* [6 marks]

Past **Exam**ple Question

- *Does alpha-beta pruning result in a better game-playing agent? Explain your answer carefully.* [4 marks]

(A) Mostly yes since it reduces the number of tree's search improving computation time and prunes trees, comparing the alpha and beta (alpha min, beta max) values if alpha is bigger than beta then the tree is pruned, however it does not search all branches if they are pruned the pruned branch could possess a better move for the max or min value however they are not taken into account.

(B) No, alpha-beta pruning merely reduces the number of nodes we must visit. It produces the same outcome as minimax. It is therefore not a better game-playing agent but one that should perform better with regards to time taken and memory required.

Past **Exam**ple Question

- *Does alpha-beta pruning result in a better game-playing agent? Explain your answer carefully.* [4 marks]

(C) Alpha-beta pruning results in a better game-playing agent because it improves the speed of the agent's decision-making. This is important in adversarial search, as there may be a time element involved in the game. Without alpha-beta pruning, the minimax algorithm naively searches the entire game tree, possibly exploring whole subtrees that will never be used because they will return a utility value larger than their parent's upper bound or lower than their parent's lower bound. Alpha-beta pruning detects if the algorithm has encountered such a subtree, and saves the algorithm a significant amount of time by pruning the branch and moving on.