Lecture Sept 22, 2022

- Adversarial search (Ch 5 of R-N)
- Also see Chapter 12 of Edelkamp.
- Two-player games

Two-player games overview

- Computer programs which play 2-player games
  - game-playing as search
  - with the complication of an opponent
- General principles of game-playing and search
  - evaluation functions
  - minimax principle
  - alpha-beta-pruning
  - heuristic techniques

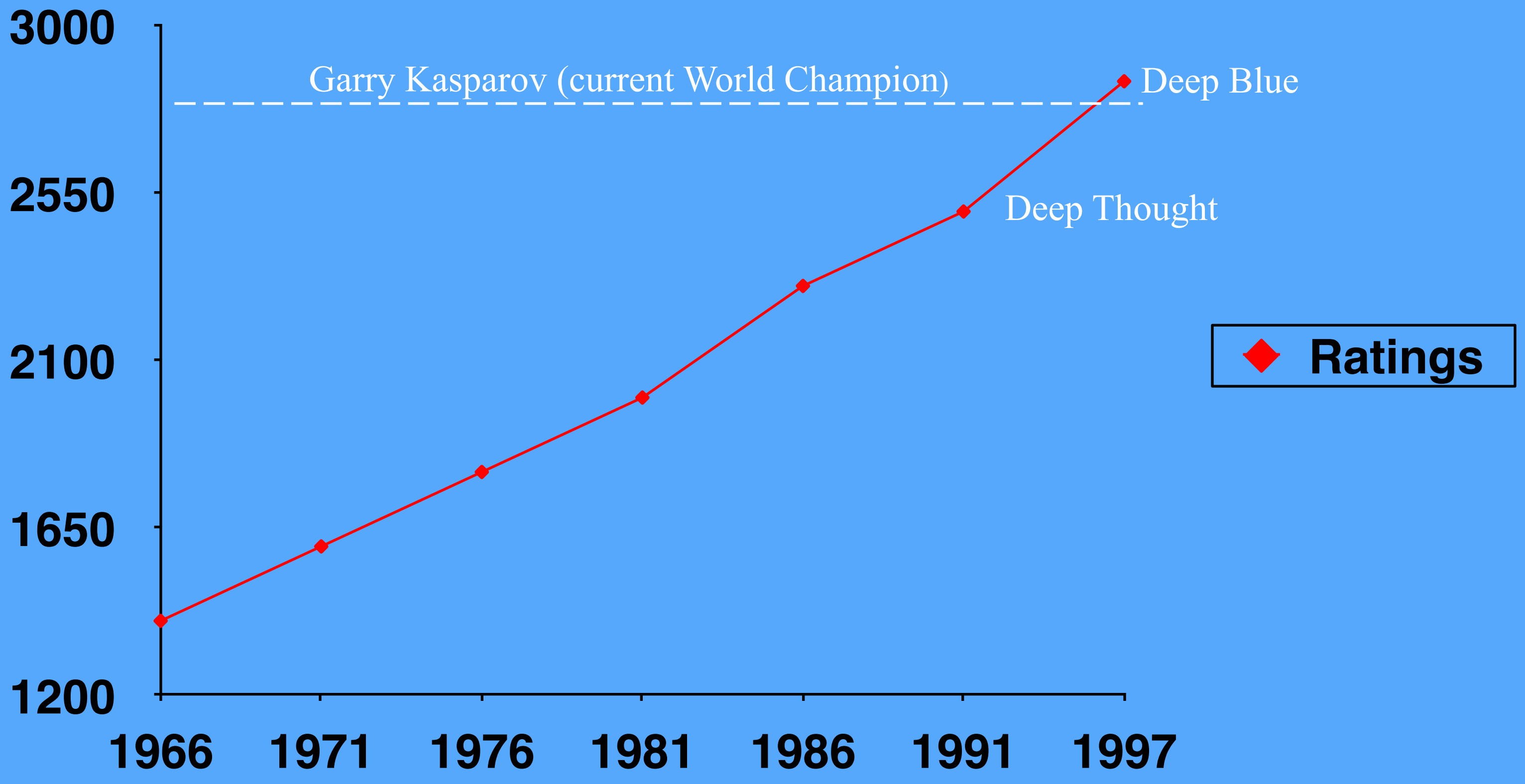
Two-player games overview

- Status of Game-Playing Systems
  - in chess, checkers, backgammon, Othello, etc, computers routinely defeat top human players (Checkers has been shown to be a draw when played by both optimally.)
- Applications?
  - think of “nature” as an opponent
  - economics, war-gaming, medical drug treatment

Games of strategy

- Deterministic rules (or deterministic rules plus probabilistic rules – these are games that combine strategy and luck, e.g. bridge, backgammon, blackjack).
- Moves are alternately made by two players A and B.
- rules define how configurations change
- A subset F of configurations is identified as final.
- typically F is partitioned into three sets: T, A and B.
  - T is tie, A (B) is win for player A (B)
- Goal is to develop a strategy for one player to win. (computer plays for that player)

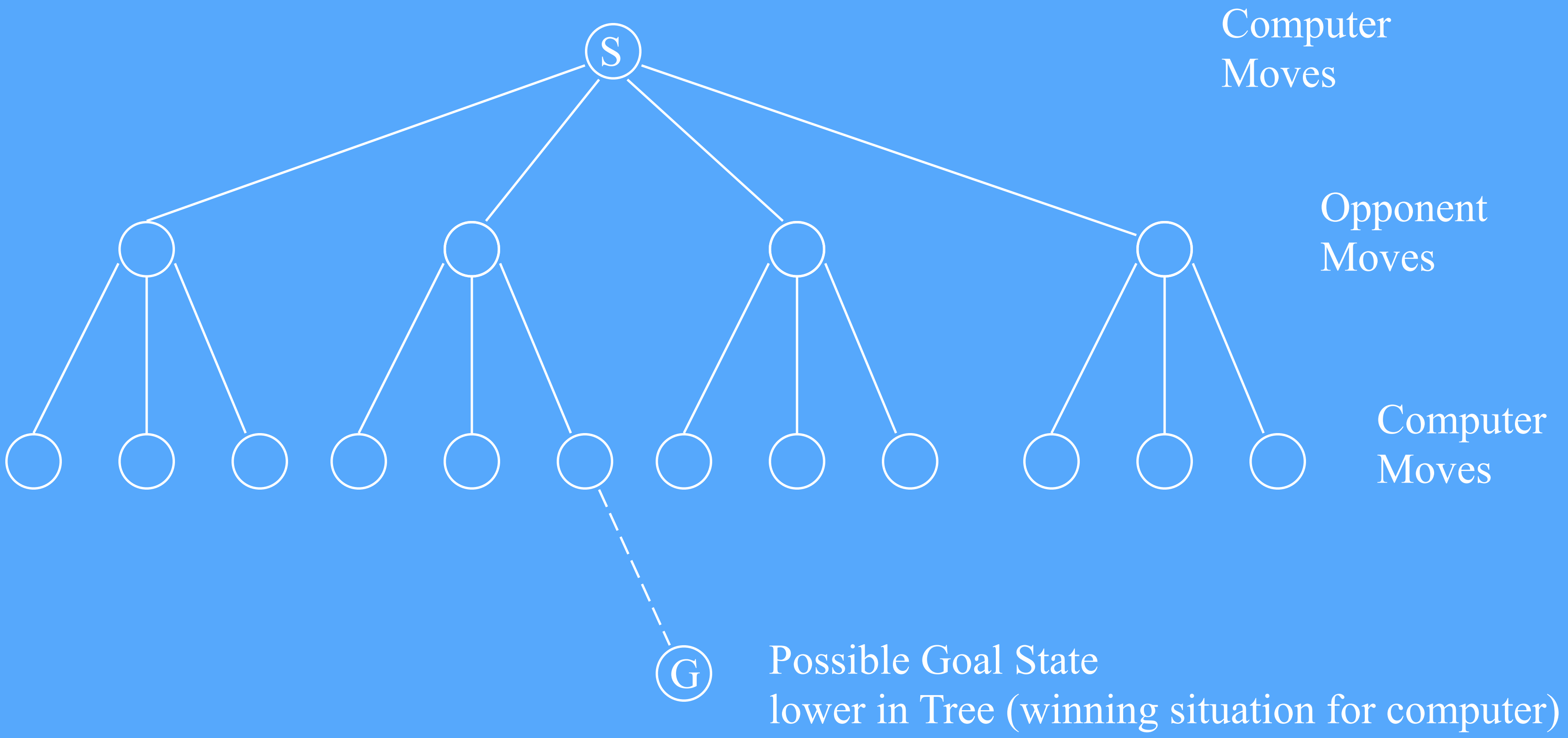
Chess Rating Scale



Two-Player Games with Complete Trees

- Let us assume that the computer has the first move. Then, the game can be described as a series of decisions, where the first decision is made by the computer, the second one by the human, the third one by the computer, and so on, until all coins are gone.
- The computer wants to make decisions that guarantee its victory (in this simple game).
- The underlying assumption is that the human always finds the optimal move.

Game Tree Representation


S
Computer Moves
Opponent Moves
Computer Moves
G
Possible Goal State lower in Tree (winning situation for computer)

- New aspect to search problem
  - there’s an opponent we cannot control
  - how can we handle this?

Game Trees

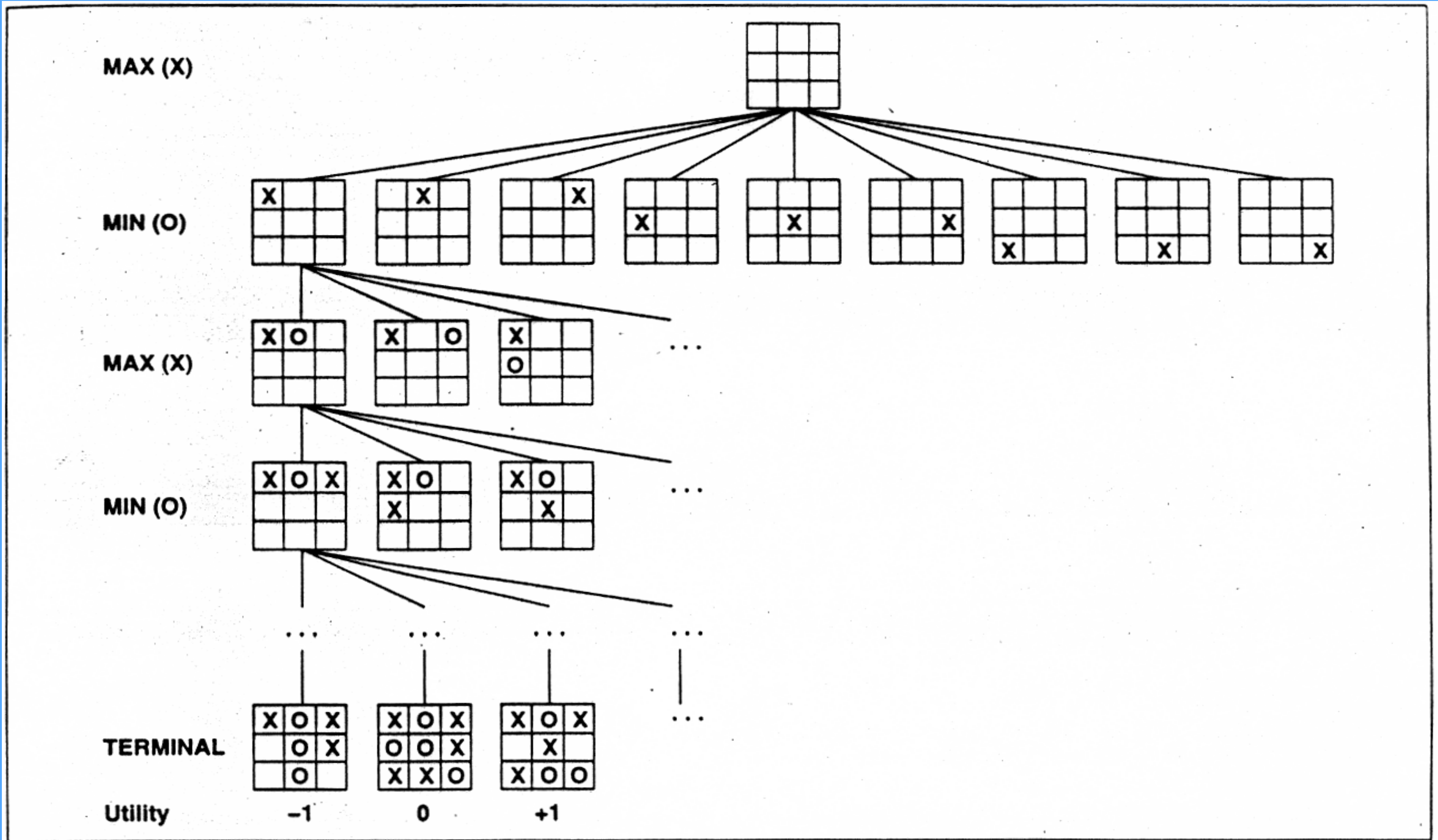
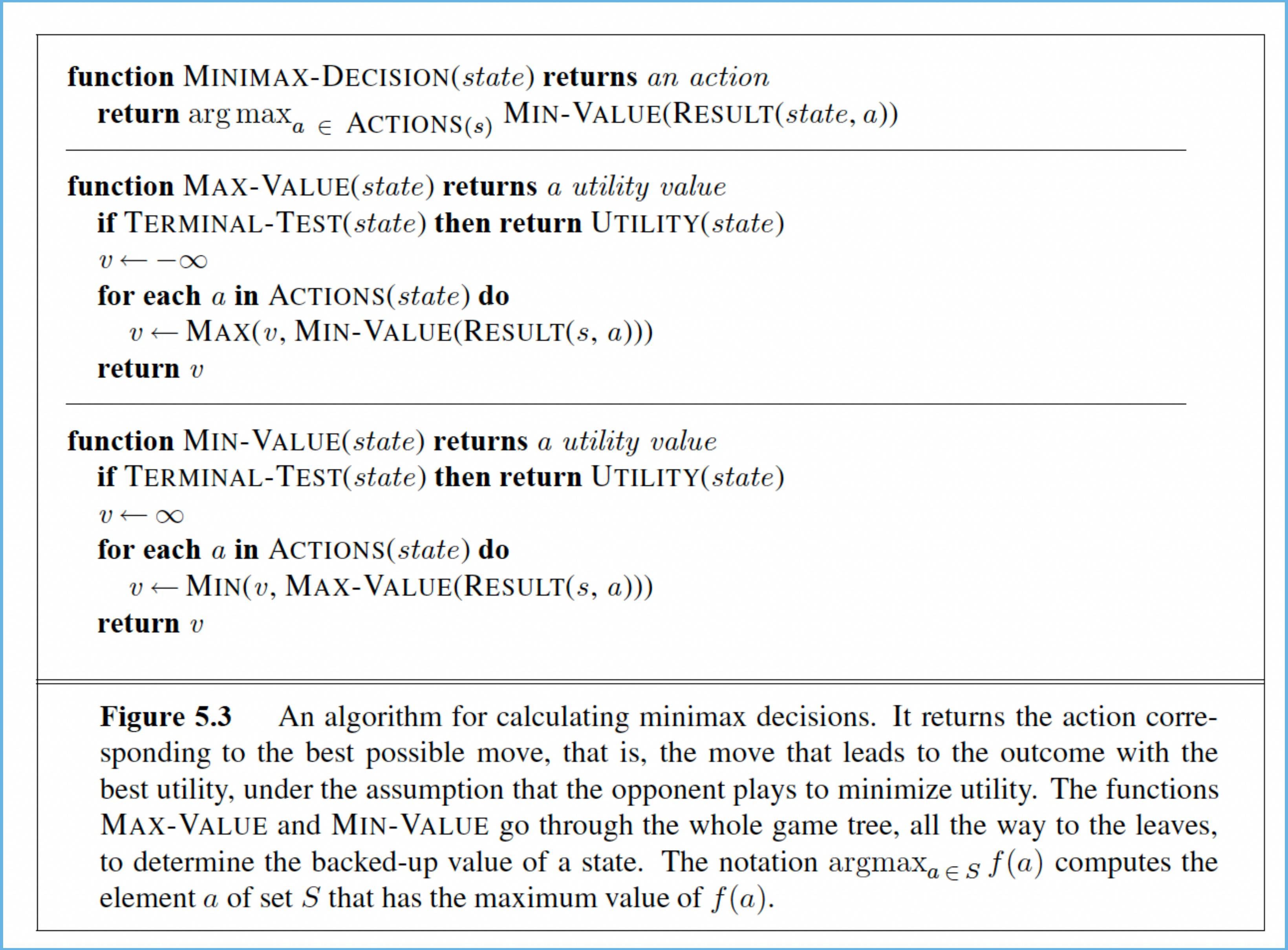


Figure 5.1 A (partial) search tree for the game of Tic-Tac-Toe. The top node is the initial state, and MAX moves first, placing an X in some square. We show part of the search tree, giving alternating moves by MIN (O) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Min-max algorithm



```
function MINIMAX-DECISION(state) returns an action
  return $\arg\max_{a \,\in\, \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
```

```
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

Game Trees

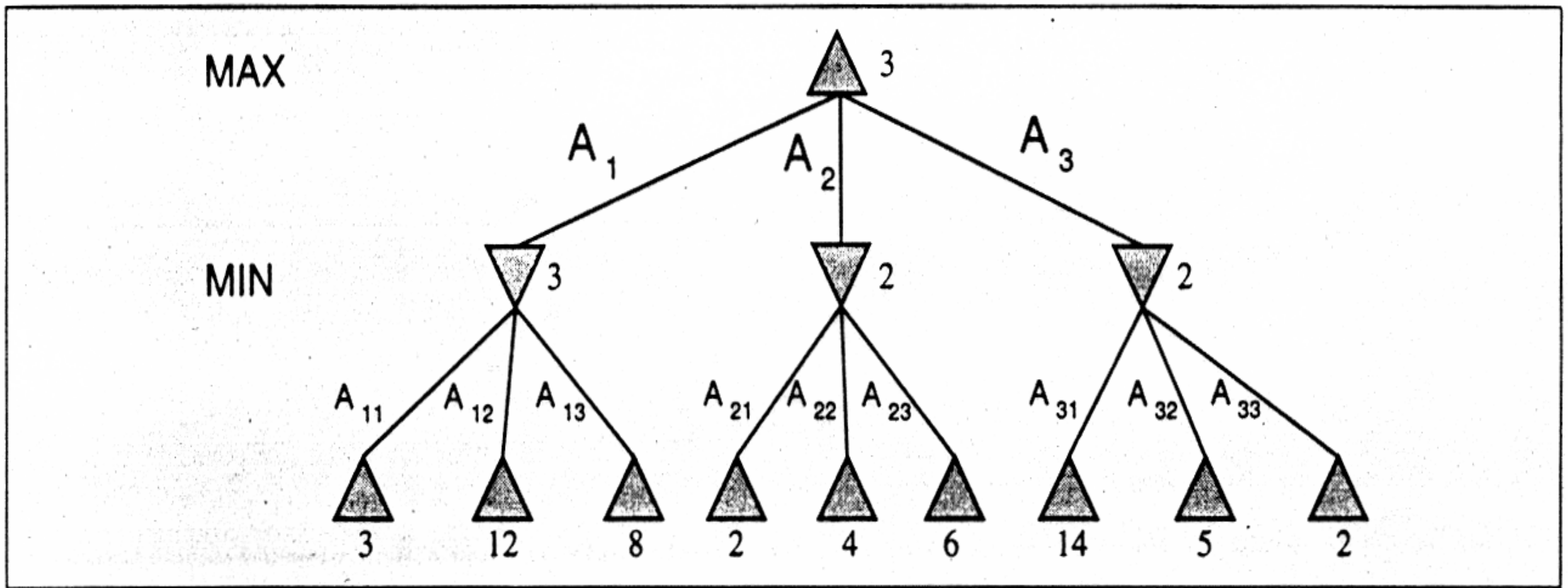


Figure 5.2 A two-ply game tree as generated by the minimax algorithm. The △ nodes are moves by MAX and the ▽ nodes are moves by MIN. The terminal nodes show the utility value for MAX computed by the utility function (i.e., by the rules of the game), whereas the utilities of the other nodes are computed by the minimax algorithm from the utilities of their successors. MAX's best move is A_1, and MIN's best reply is A_{11}.

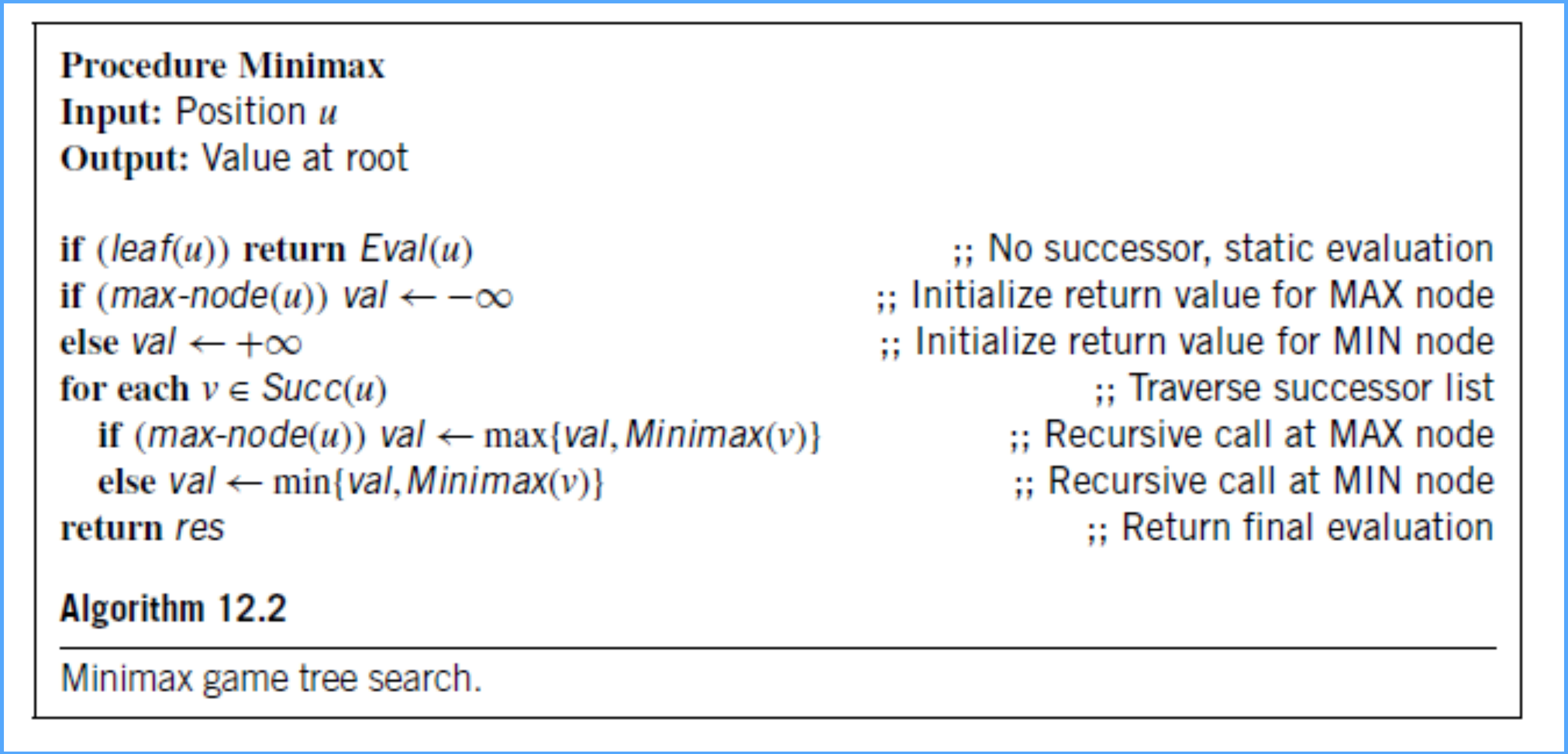
Min-max Principle

- "Assume the worst"
  - say each configuration has an evaluation number
  - high numbers favor the player (the computer)
    - so we want to choose moves which maximize evaluation
  - low numbers favor the opponent
    - so they will choose moves which minimize evaluation
- Minimax Principle
  - you (the computer) assume that the opponent will choose the minimizing move next (after your move)
  - so you now choose the best move under this assumption
    - i.e., the maximum (highest-value) option considering both your move and the opponent's optimal move.

An optimal procedure: The Min-Max method

- Designed to find the optimal strategy for Max and find best move:
  - 1. Generate the whole game tree to leaves
  - 2. Apply utility (payoff) function to leaves
  - 3. Back-up values from leaves toward the root:
    - a Max node computes the max of its child values
    - a Min node computes the Min of its child values
  - 4. When value reaches the root: choose max value and the corresponding move.
- However: It is impossible to develop the whole search tree, instead develop part of the tree and evaluate promise of leaves using a static evaluation function.

More general Minmax search algorithm



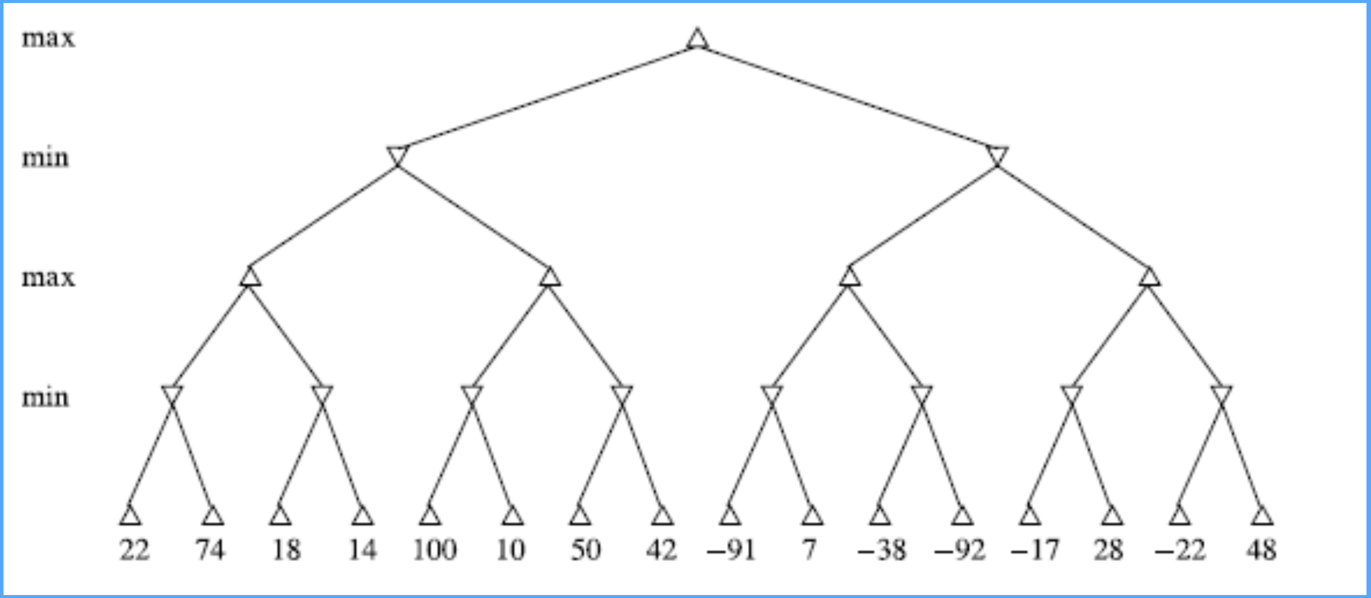
Procedure Minimax
Input: Position u
Output: Value at root

```
if ($leaf(u)$) return $Eval(u)$                              ;; No successor, static evaluation
if ($max\text{-}node(u)$) $val \leftarrow -\infty$            ;; Initialize return value for MAX node
else $val \leftarrow +\infty$                                ;; Initialize return value for MIN node
for each $v \in Succ(u)$                                     ;; Traverse successor list
   if ($max\text{-}node(u)$) $val \leftarrow \max\{val, Minimax(v)\}$   ;; Recursive call at MAX node
   else $val \leftarrow \min\{val, Minimax(v)\}$             ;; Recursive call at MIN node
return $res$                                                 ;; Return final evaluation
```

Algorithm 12.2

Minimax game tree search.

What is the minimax value of the root node?



Complexity of Game Playing

- Imagine we could predict the opponent’s moves given each computer move

- How complex would search be in this case?
  - worst case, it will be $O(b^d)$
  - Chess:
    - b ~ 35 (average branching factor)
    - d ~ 100 (depth of game tree for typical game)
    - $b^d \sim 35^{100} \sim 10^{154}$ nodes!!
  - Tic-Tac-Toe
    - ~5 legal moves, total of 9 moves
    - Number of states ~ 26000 (counting all rotations, reflections …)
  - well-known games can produce enormous search trees

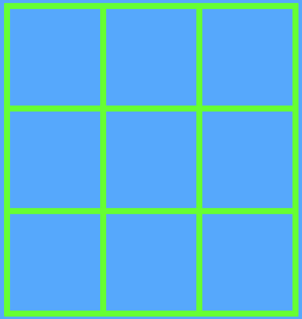
Static (Heuristic) Evaluation Functions

- An Evaluation Function:
  - estimates how good the current board configuration is for a player.
  - Typically, one figures how good it is for the player, and how good it is for the opponent, and subtracts the opponents score from the players
  - Othello: Number of white pieces - Number of black pieces
  - Chess: Value of all white pieces - Value of all black pieces
- Typical values from -infinity (loss) to +infinity (win) or [-1, +1].
- If the board evaluation is X for a player, it's -X for the opponent

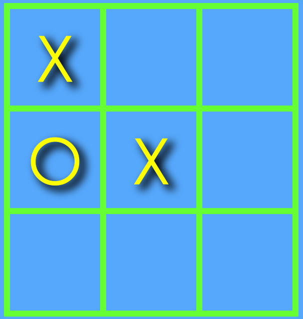
Two-Player Games

- Define a static evaluation function e(p) that tells the computer how favorable the current game position p is from its perspective.
- e(p) will assume large values if a position is likely to result in a win for the computer, and low values if it predicts its defeat.
- the computer will make a move that guarantees a maximum value for e(p) after a certain number of moves.
- We can use the Minimax procedure with a specific maximum search depth (ply-depth k for k moves of each player).

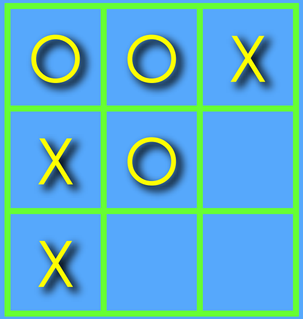
Two-Player Games Example: Tic-tac-Toe



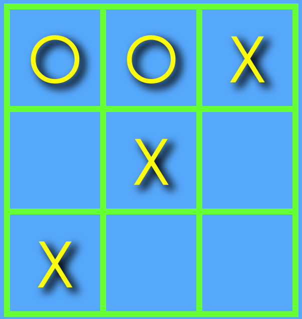
$e(p) = 8 - 8 = 0$



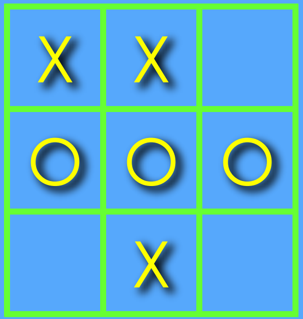
$e(p) = 6 - 2 = 4$



$e(p) = 2 - 2 = 0$



$e(p) = \infty$



$e(p) = -\infty$

Backup Values

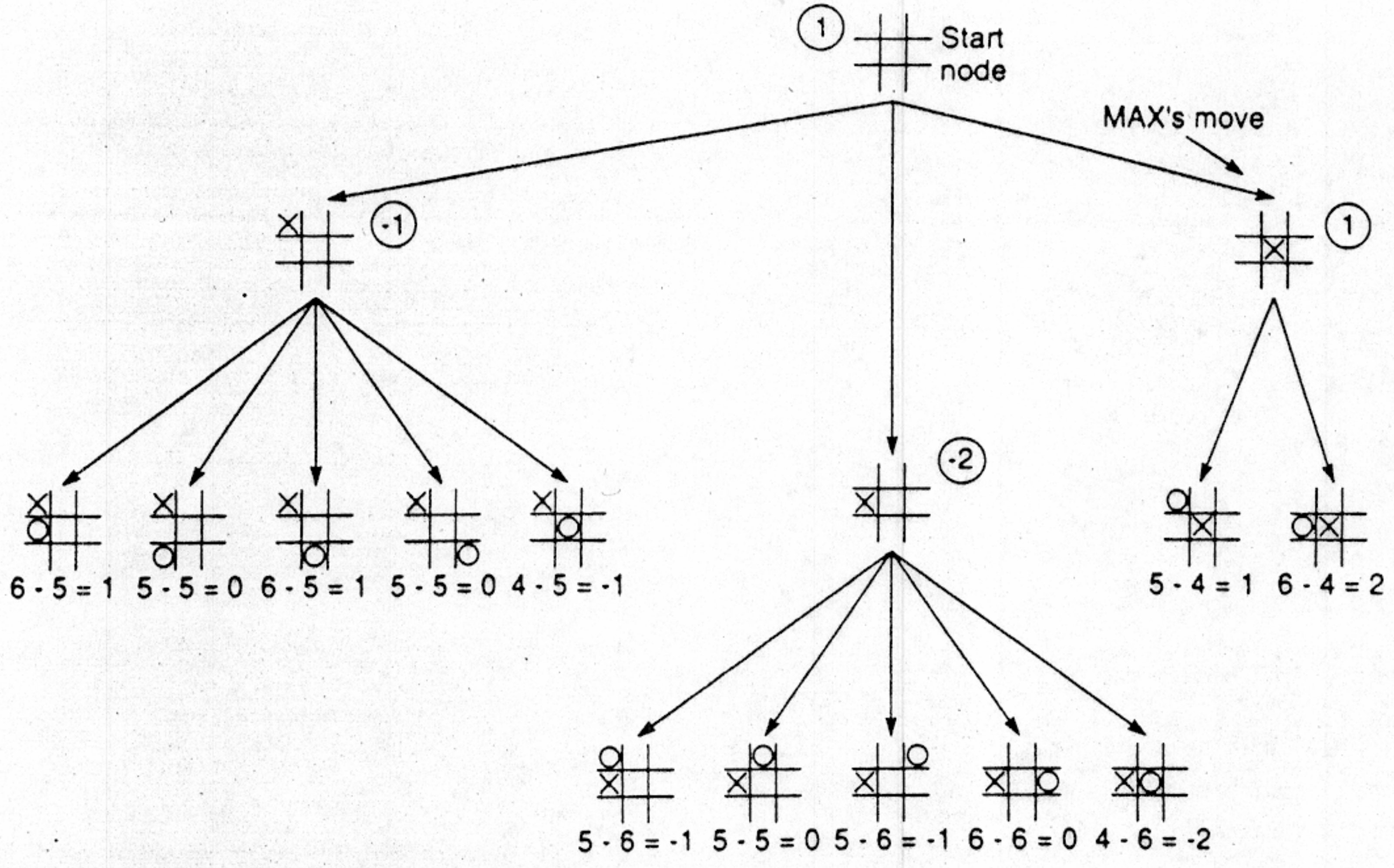


Figure 4.17 Two-ply minimax applied to the opening move of tic-tac-toe.

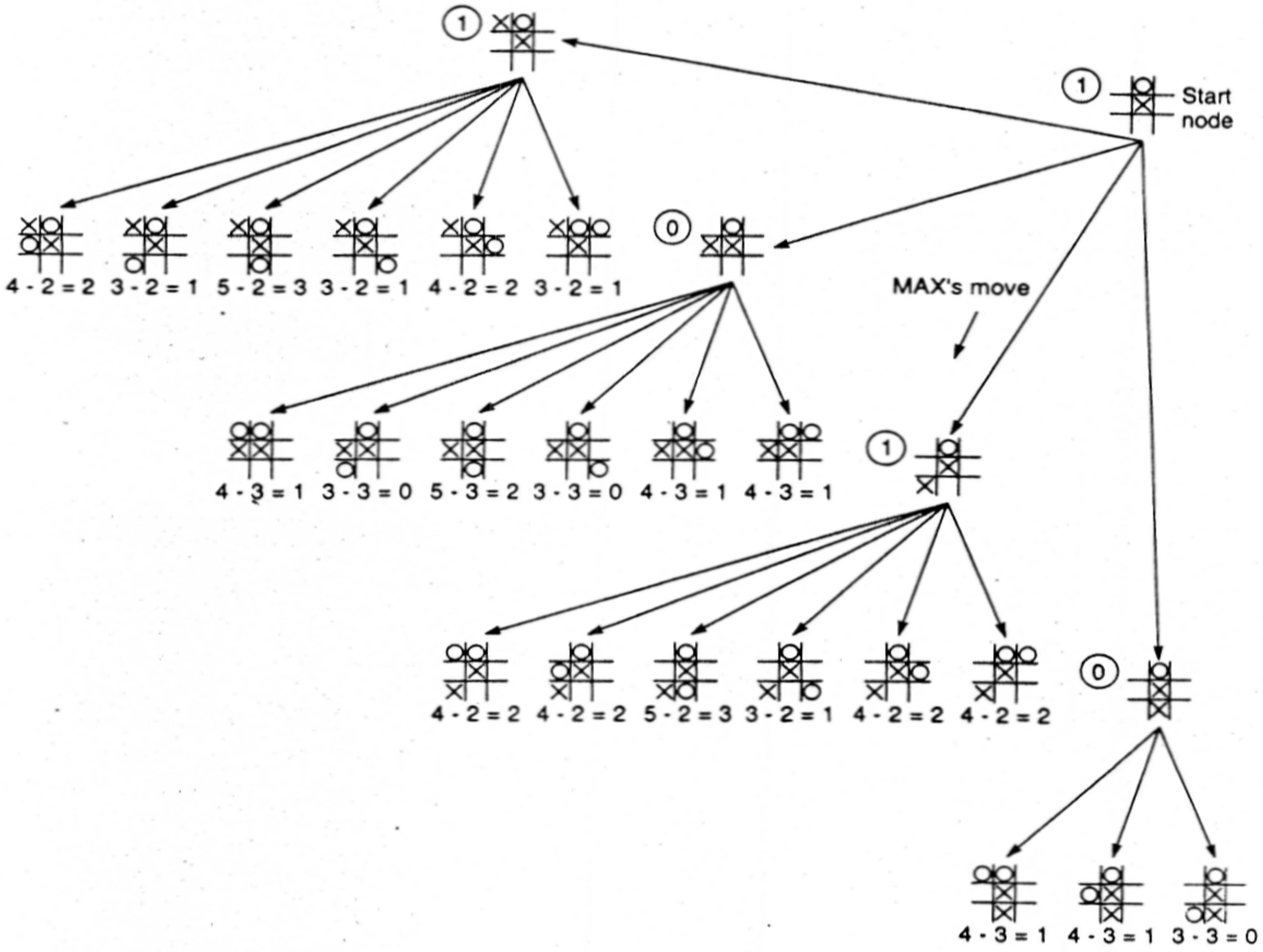


Figure 4.18 Two-ply minimax applied to X's second move of tic-tac-toe.

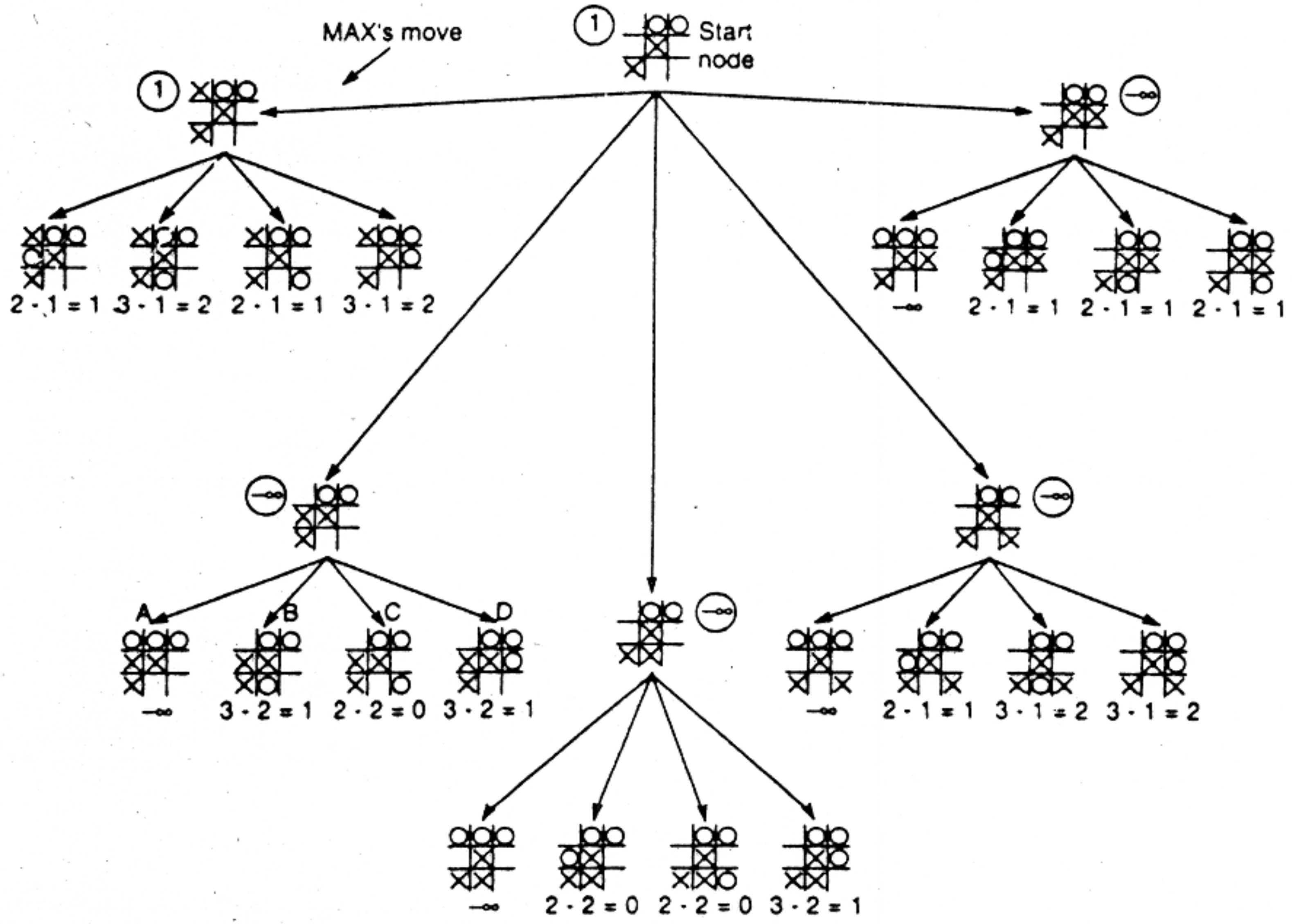


Figure 4.19 Two-ply minimax applied to **X**'s move near end game.

Alpha Beta Procedure

- Idea:
  - Do Depth first search to generate partial game tree.
  - Give static evaluation function to leaves.
  - compute bound on internal nodes.
- Alpha, Beta bounds:
  - Alpha value for Max node means that Max real value is at least alpha.
  - Beta for Min node means that Min can guarantee a value below Beta.
- Computation:
  - Alpha of a Max node is the maximum value of its seen children.
  - Beta of a Min node is the lowest value seen of its child node .

When to Prune

- Pruning
  - Below a Min node whose beta value is lower than or equal to the alpha value of a max ancestor.
  - Below a Max node having an alpha value greater than or equal to the beta value of a min ancestor.

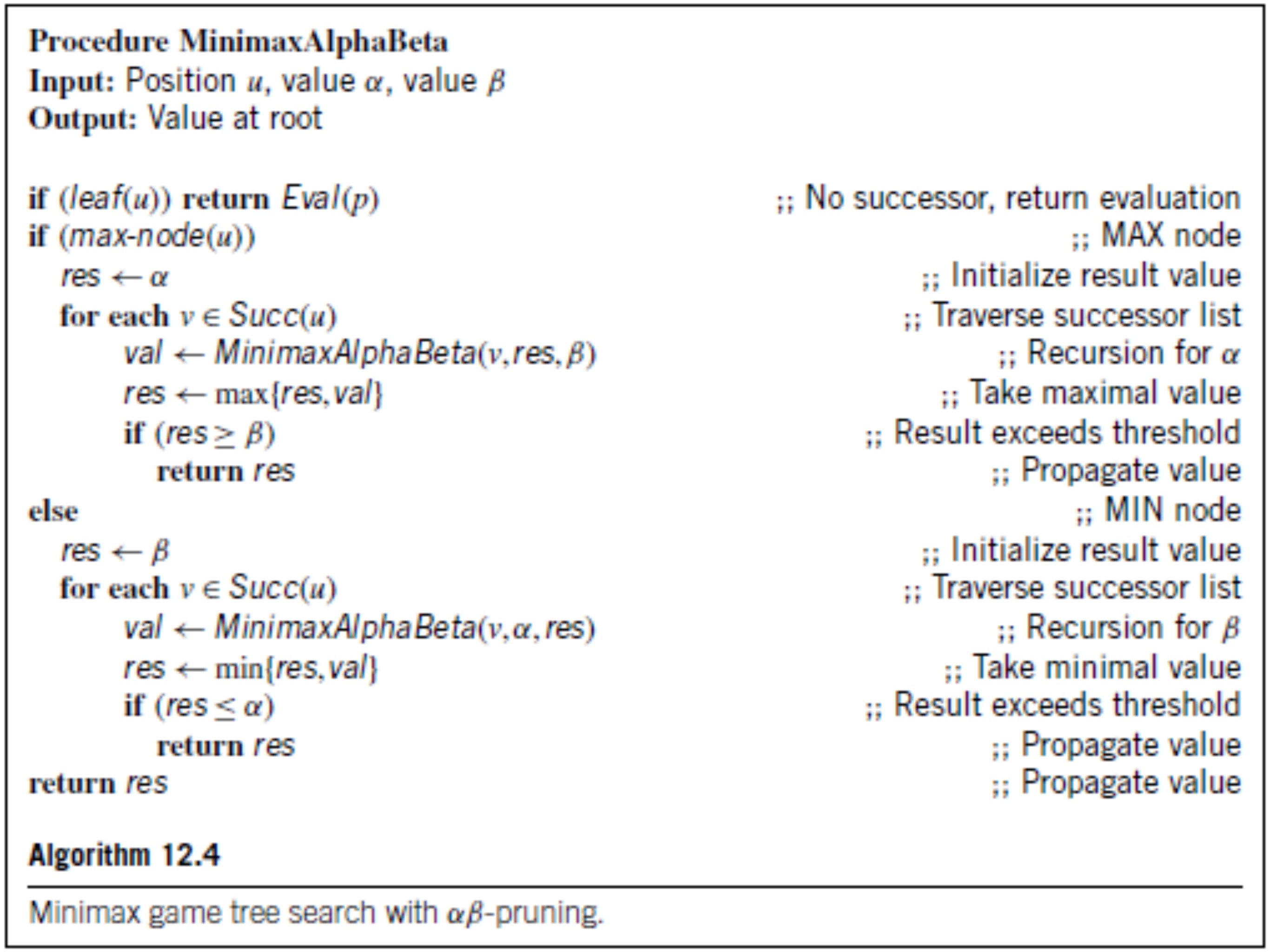
The Alpha-Beta Procedure

- Now let us specify how to prune the Minimax tree in the case of a static evaluation function.
- Use two variables alpha (associated with MAX nodes) and beta (associated with MIN nodes).
- These variables contain the best (highest or lowest, resp.) e(p) value at a node p that has been found so far.
- Notice that alpha can never decrease, and beta can never increase.

The Alpha-Beta Procedure

- There are two rules for terminating search:
- Search can be stopped below any MIN node having a beta value less than or equal to the alpha value of any of its MAX ancestors.
- Search can be stopped below any MAX node having an alpha value greater than or equal to the beta value of any of its MIN ancestors.

alpha-beta pruning algorithm

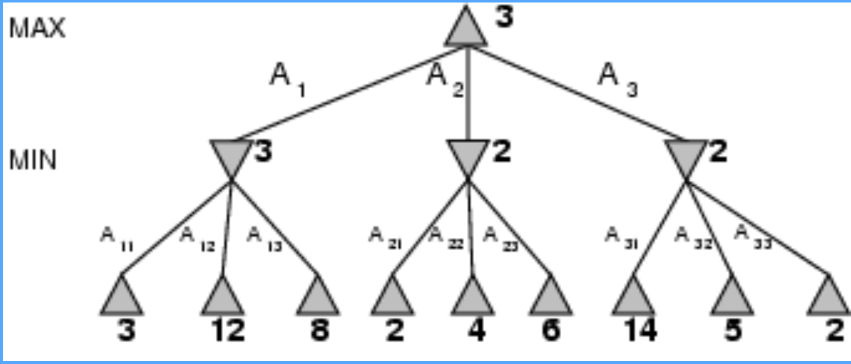


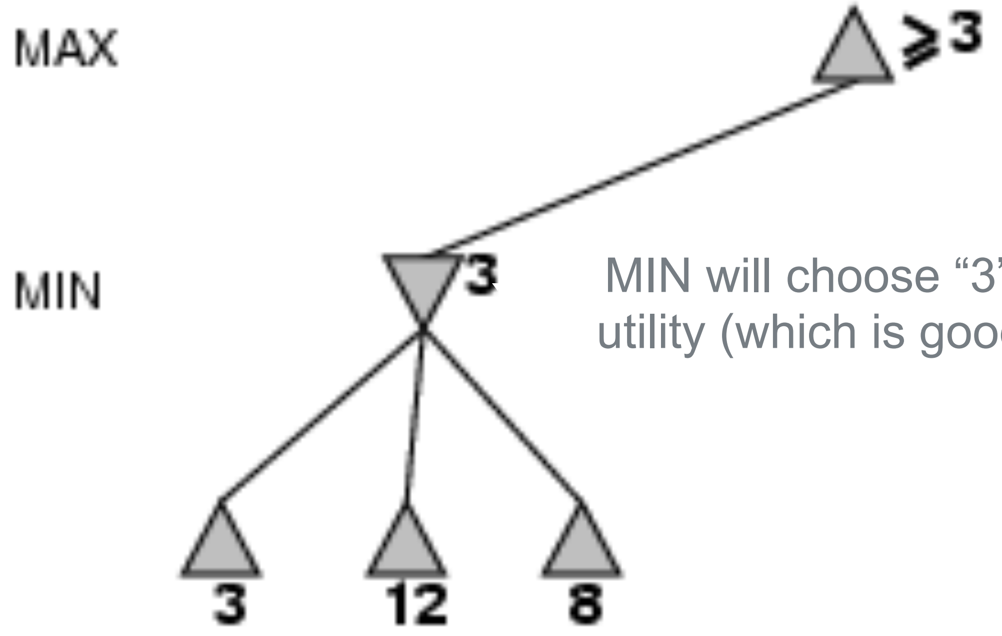
```
Procedure MinimaxAlphaBeta
Input: Position $u$, value $\alpha$, value $\beta$
Output: Value at root

if ($leaf(u)$) return $Eval(p)$                  ;; No successor, return evaluation
if ($max\text{-}node(u)$)                        ;; MAX node
  $res \leftarrow \alpha$                         ;; Initialize result value
  for each $v \in Succ(u)$                       ;; Traverse successor list
      $val \leftarrow MinimaxAlphaBeta(v, res, \beta)$   ;; Recursion for $\alpha$
      $res \leftarrow \max\{res, val\}$           ;; Take maximal value
      if ($res \geq \beta$)                      ;; Result exceeds threshold
        return $res$                             ;; Propagate value
else                                             ;; MIN node
  $res \leftarrow \beta$                          ;; Initialize result value
  for each $v \in Succ(u)$                       ;; Traverse successor list
      $val \leftarrow MinimaxAlphaBeta(v, \alpha, res)$  ;; Recursion for $\beta$
      $res \leftarrow \min\{res, val\}$           ;; Take minimal value
      if ($res \leq \alpha$)                     ;; Result exceeds threshold
        return $res$                             ;; Propagate value
return $res$                                     ;; Propagate value
```

Algorithm 12.4

Minimax game tree search with $\alpha\beta$-pruning.

α-β pruning example

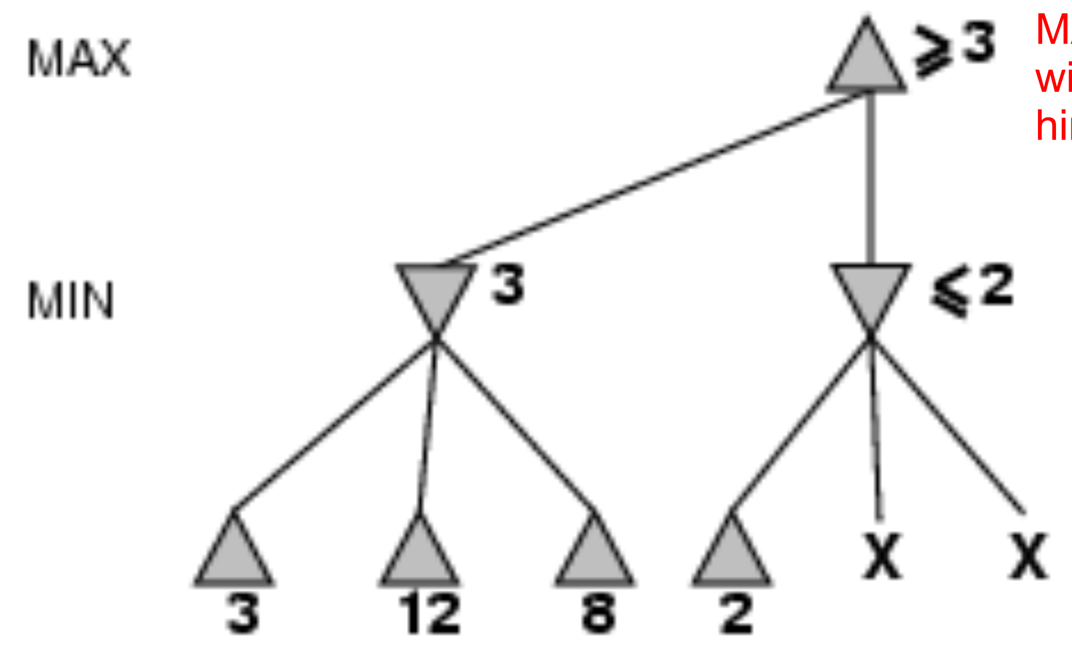




MAX knows that it can at least get "3" by playing this branch

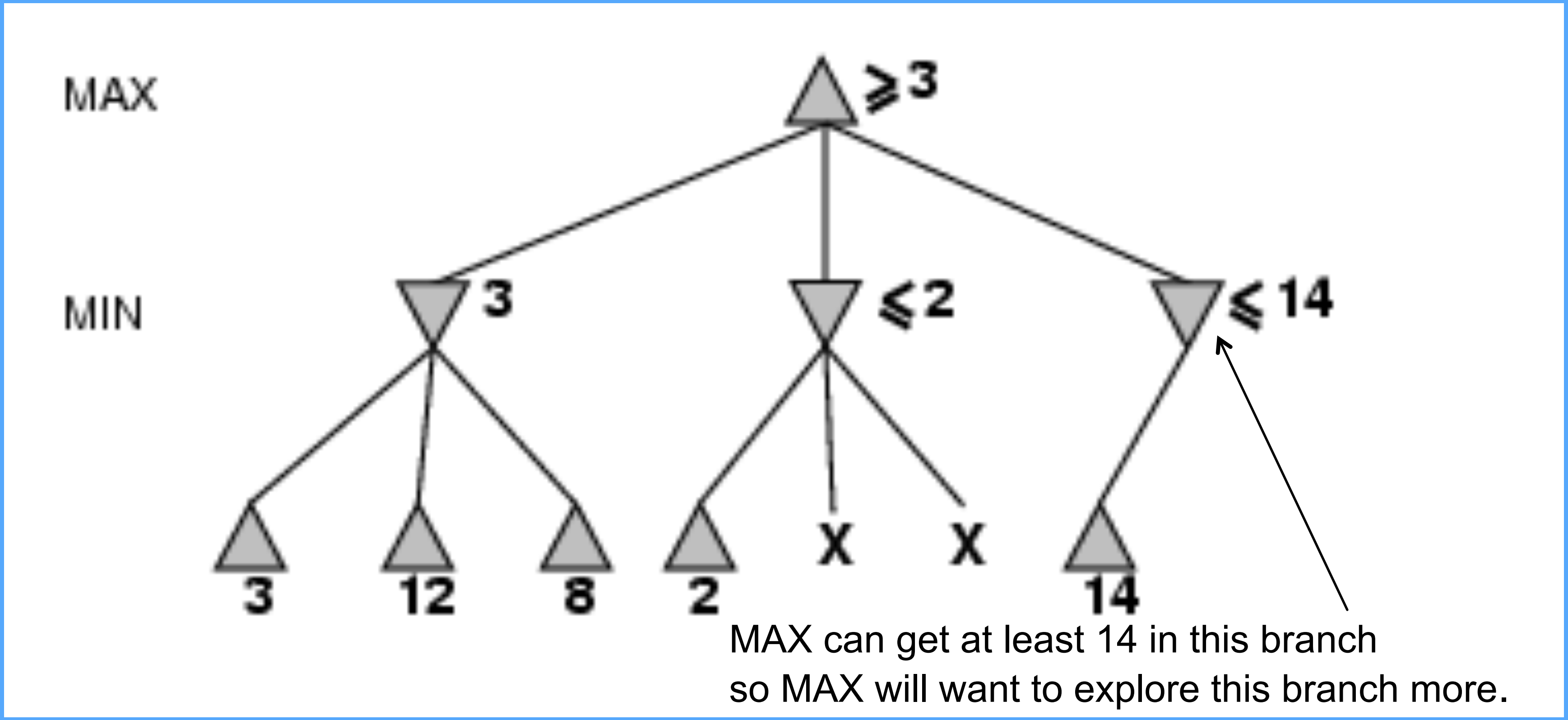
MIN will choose "3", because it minimizes the utility (which is good for MIN)

α-β pruning example



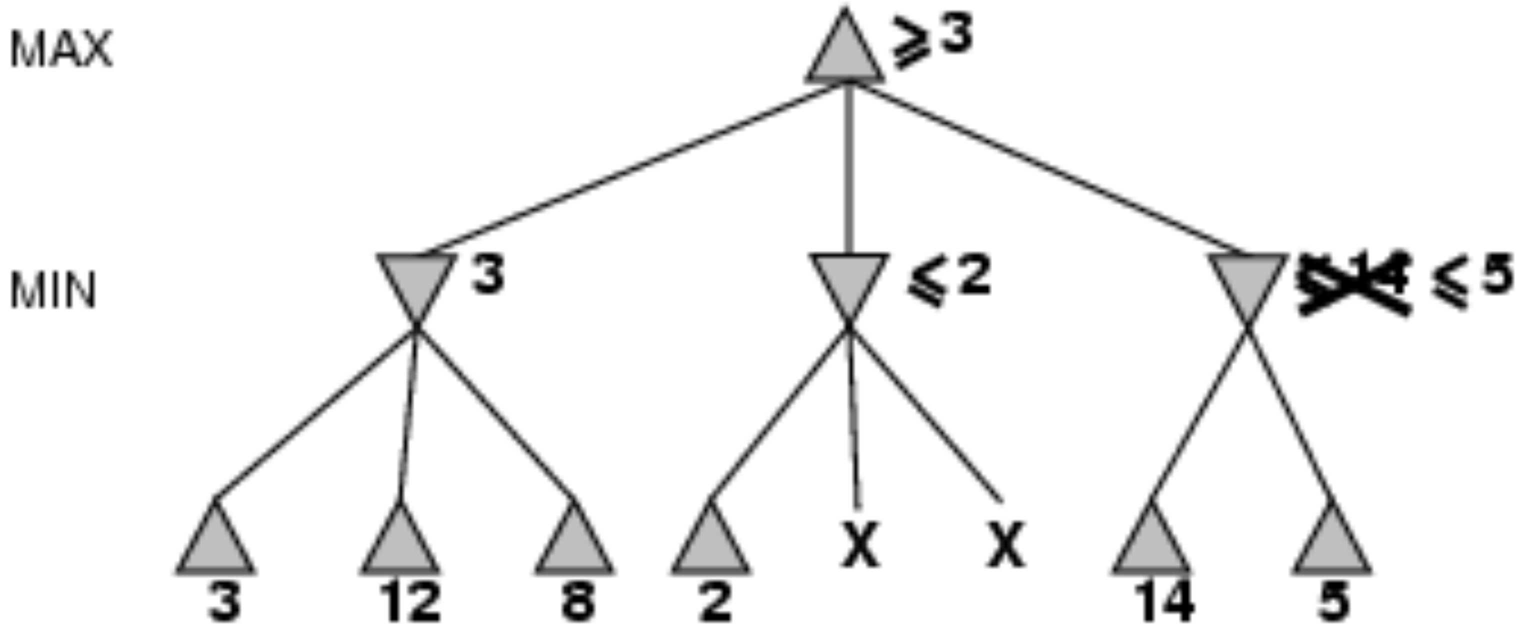
MAX knows that the new branch will never be better than 2 for him. It can *ignore* it.

α-β pruning example



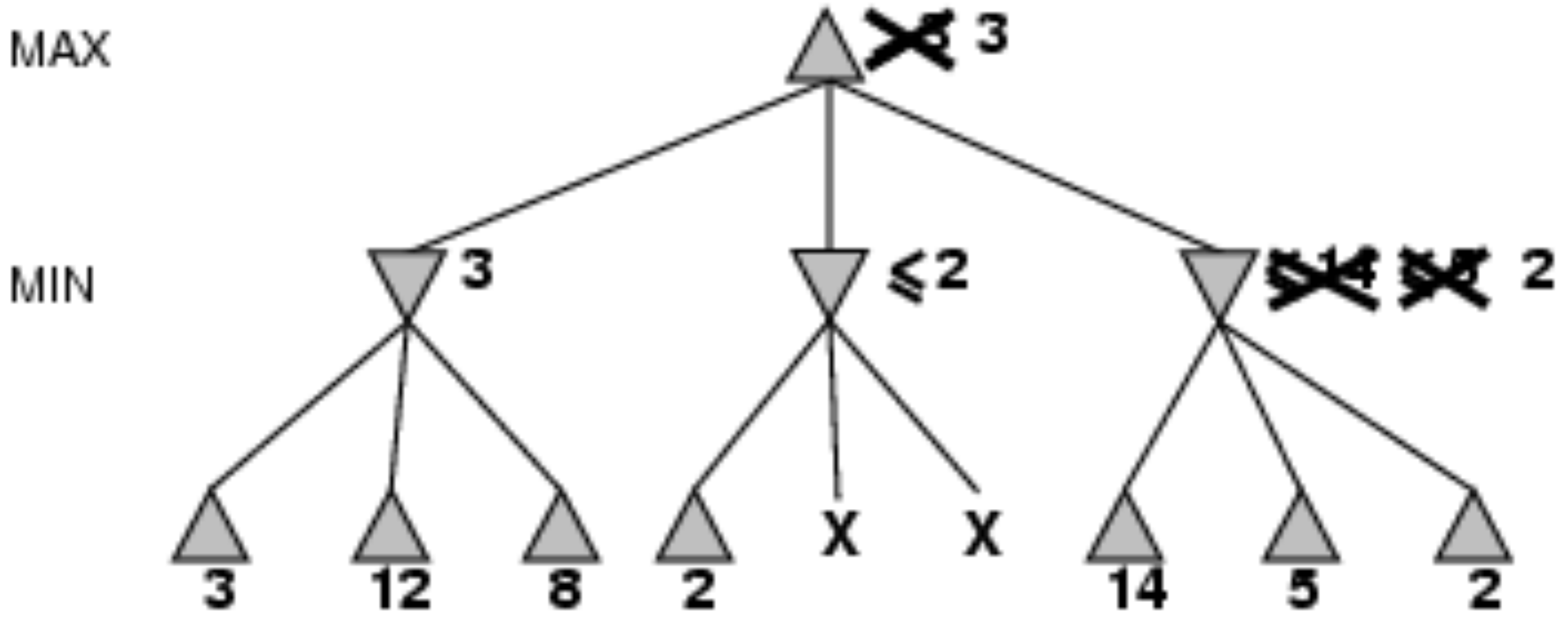
MAX can get at least 14 in this branch
so MAX will want to explore this branch more.

α-β pruning example



Should continue to explore

α-β pruning example



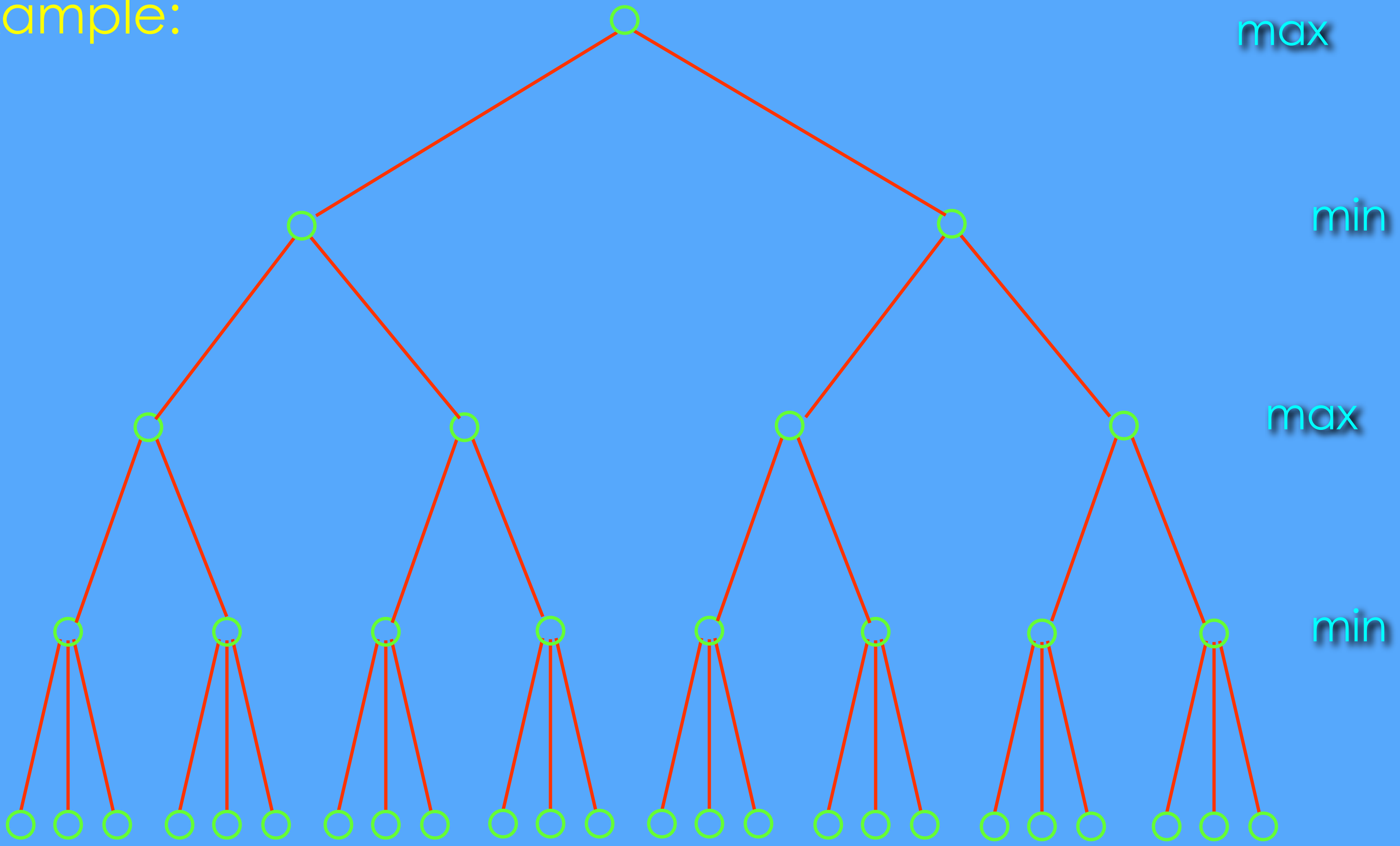
Search completed without exploring two branches

Properties of α-β pruning

- Pruning does not affect final result (it is exact).
- Good move ordering improves effectiveness of pruning (see last branch in example).
- The values at intermediate nodes may not be the same as the values computed by the minmax algorithm.

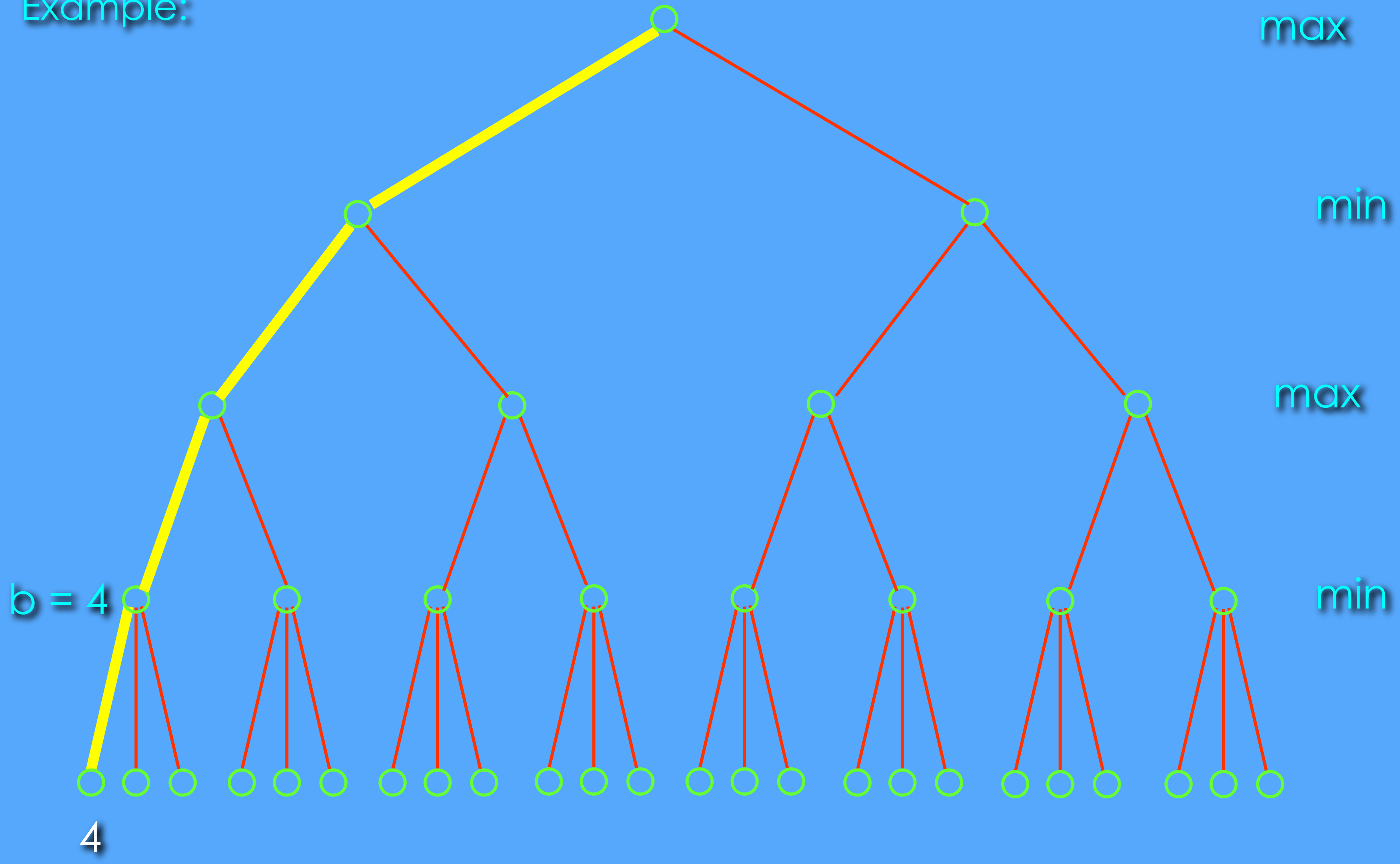
The Alpha-Beta Procedure

Example:



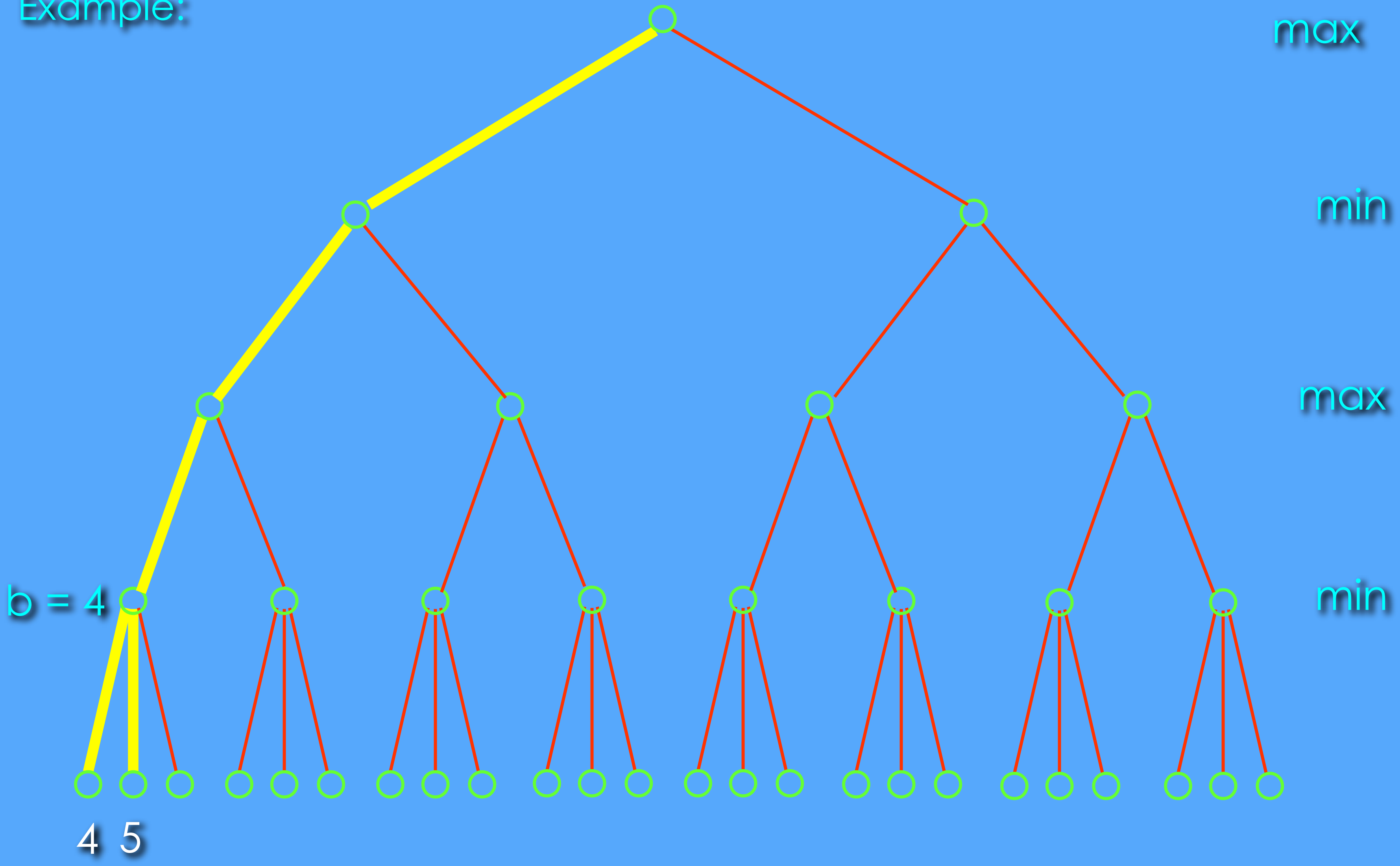
The Alpha-Beta Procedure

Example:



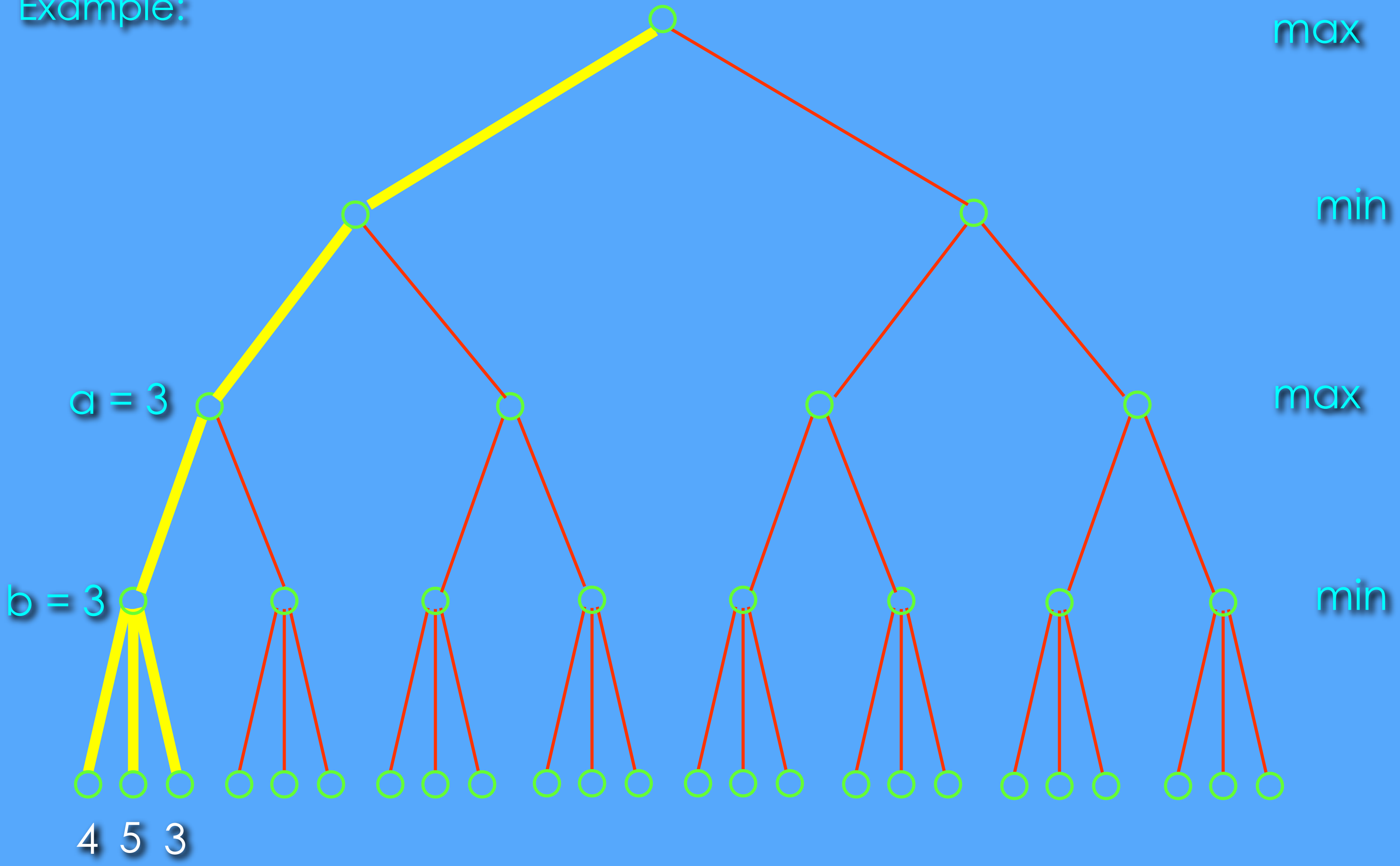
The Alpha-Beta Procedure

Example:



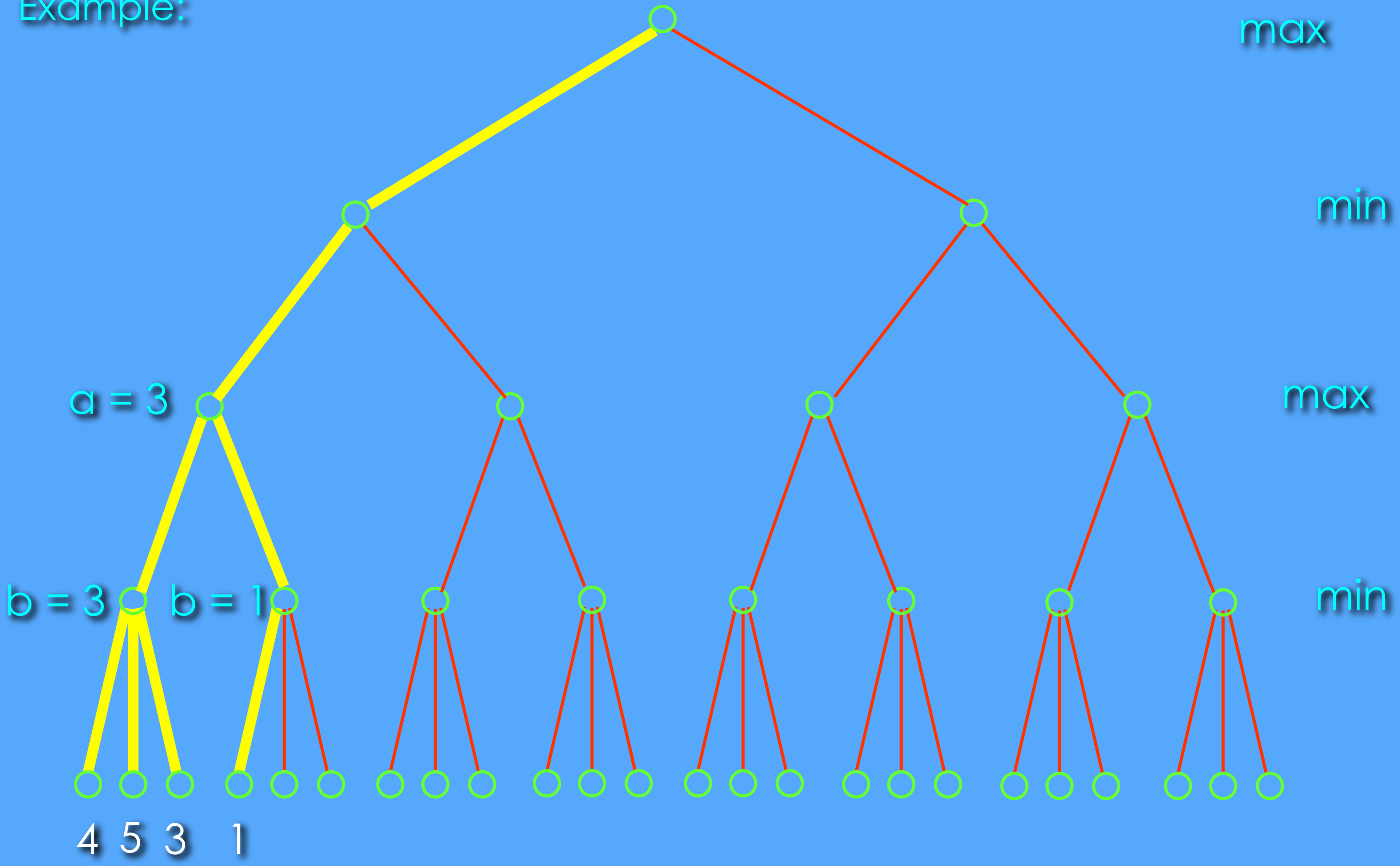
The Alpha-Beta Procedure

Example:



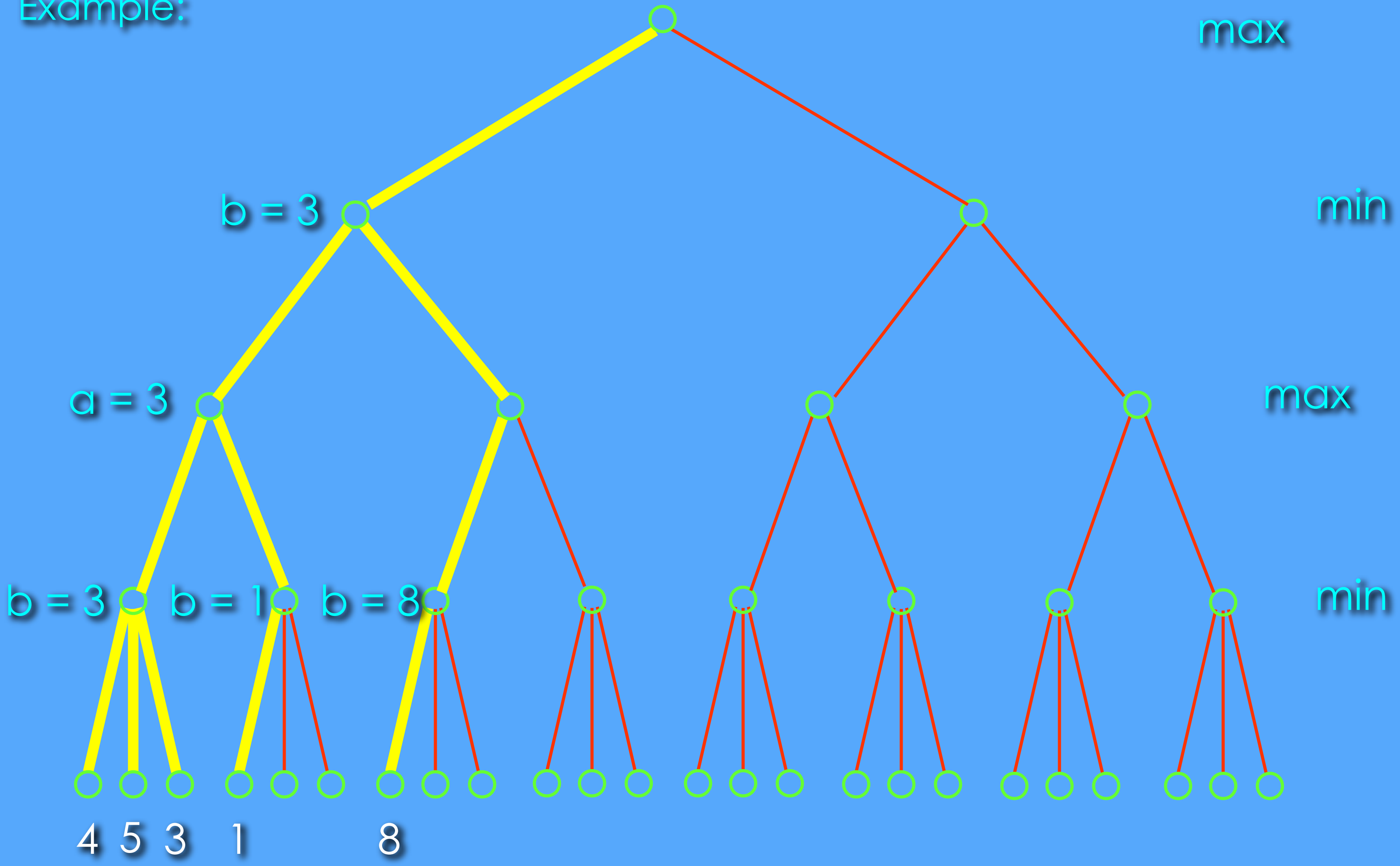
The Alpha-Beta Procedure

Example:



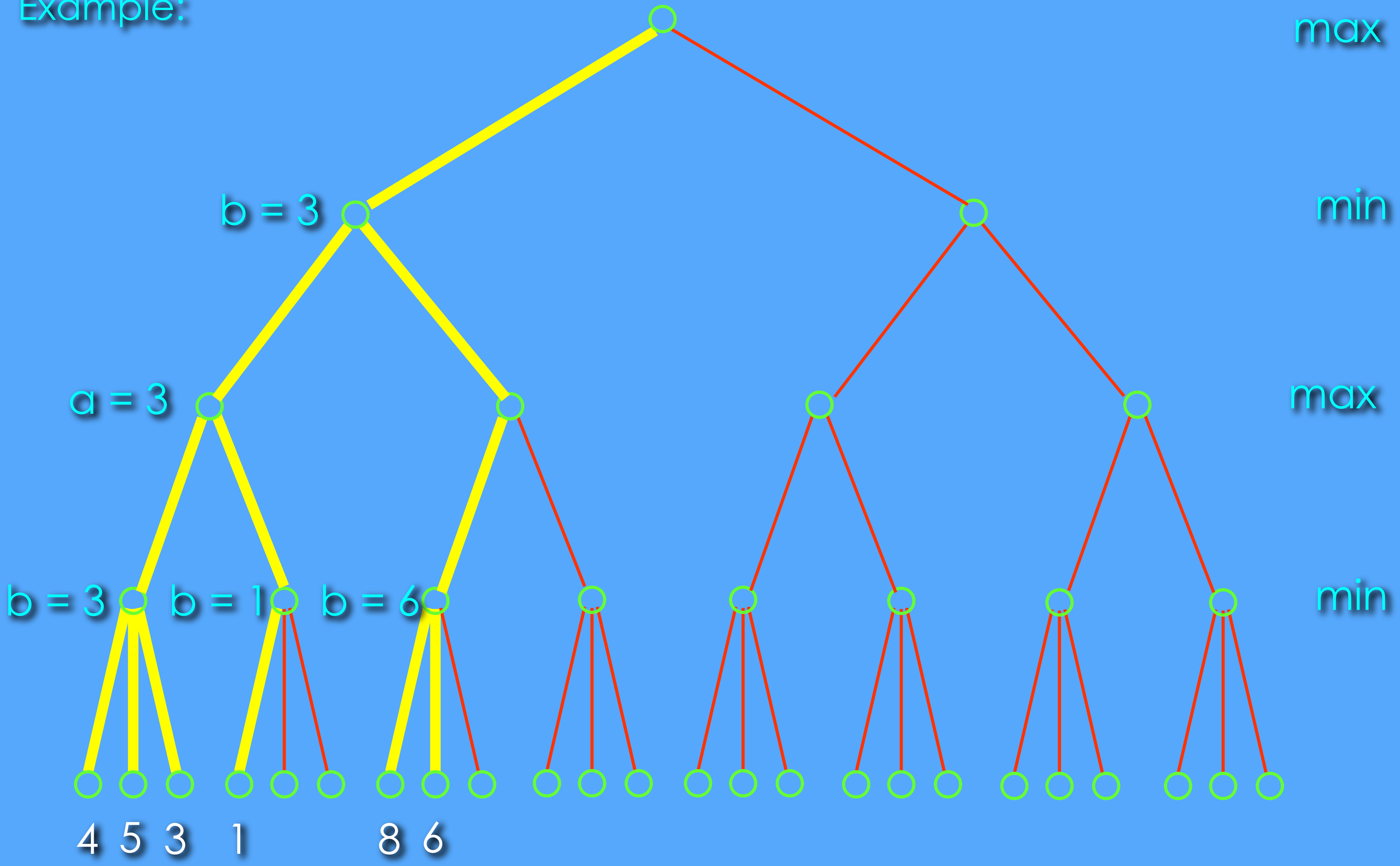
The Alpha-Beta Procedure

Example:



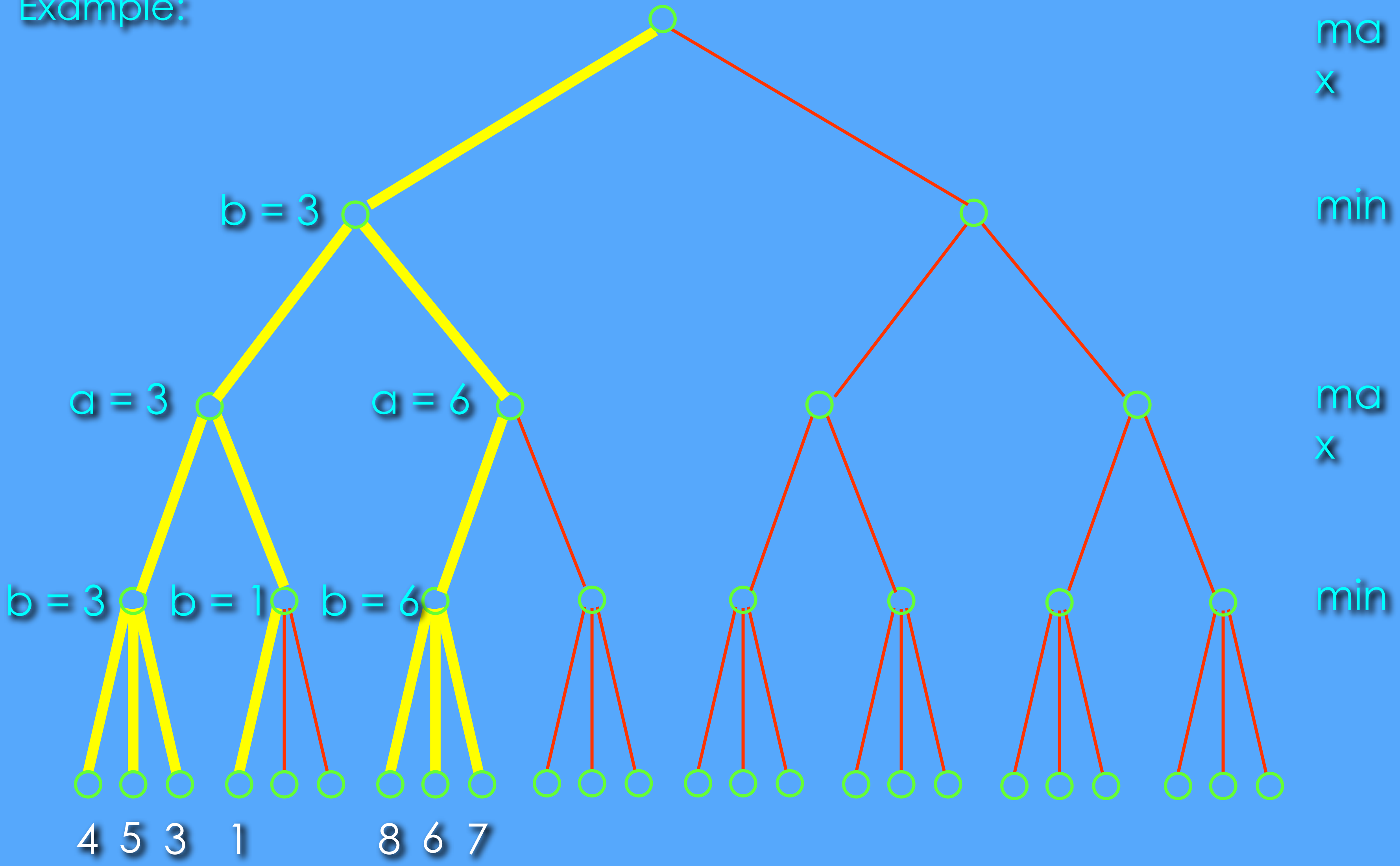
The Alpha-Beta Procedure

Example:



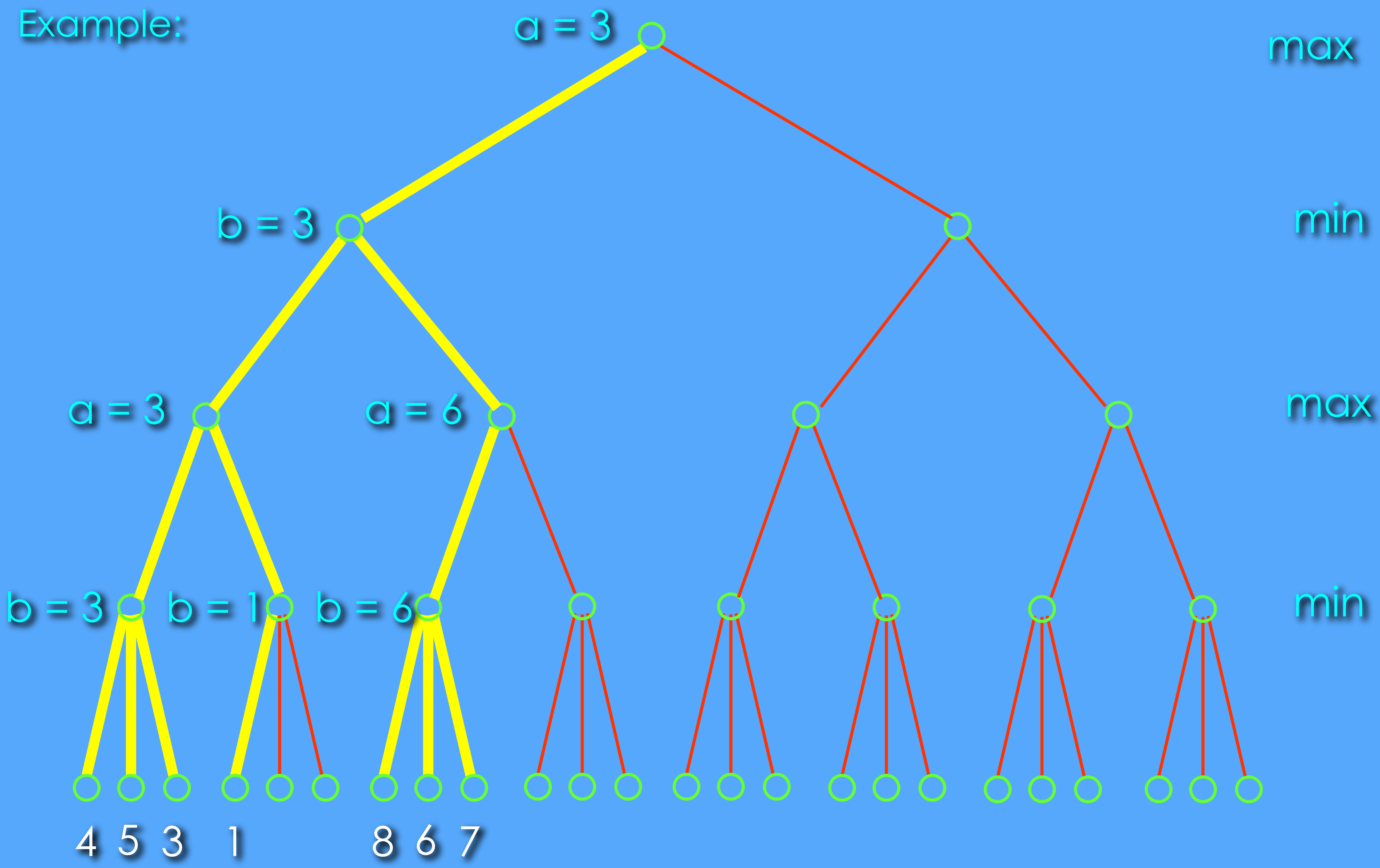
The Alpha-Beta Procedure

Example:



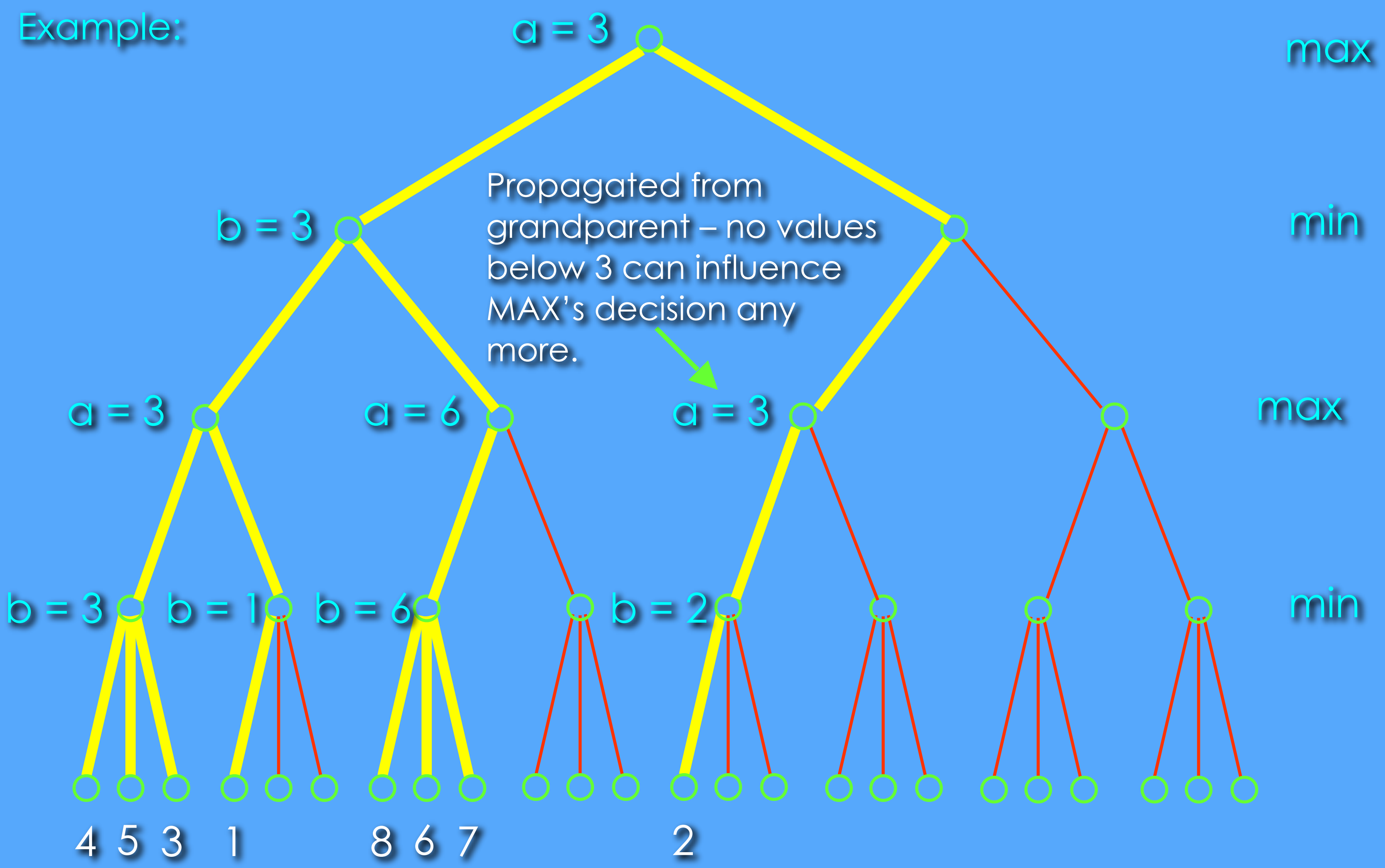
The Alpha-Beta Procedure

Example:



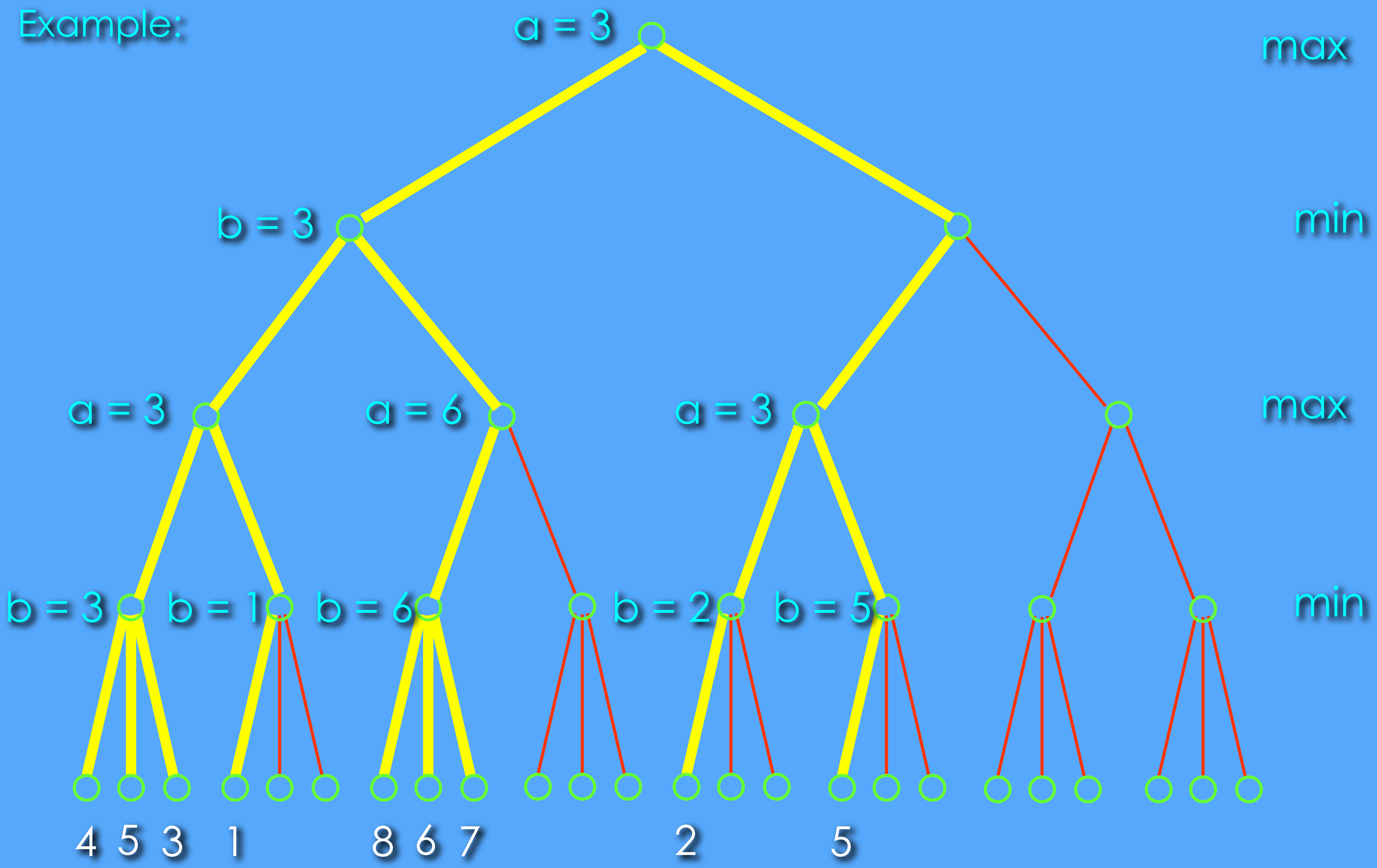
The Alpha-Beta Procedure

Example:


a = 3
max
b = 3
min
Propagated from grandparent – no values below 3 can influence MAX's decision any more.
a = 3
a = 6
a = 3
max
b = 3
b = 1
b = 6
b = 2
min
4 5 3 1
8 6 7
2

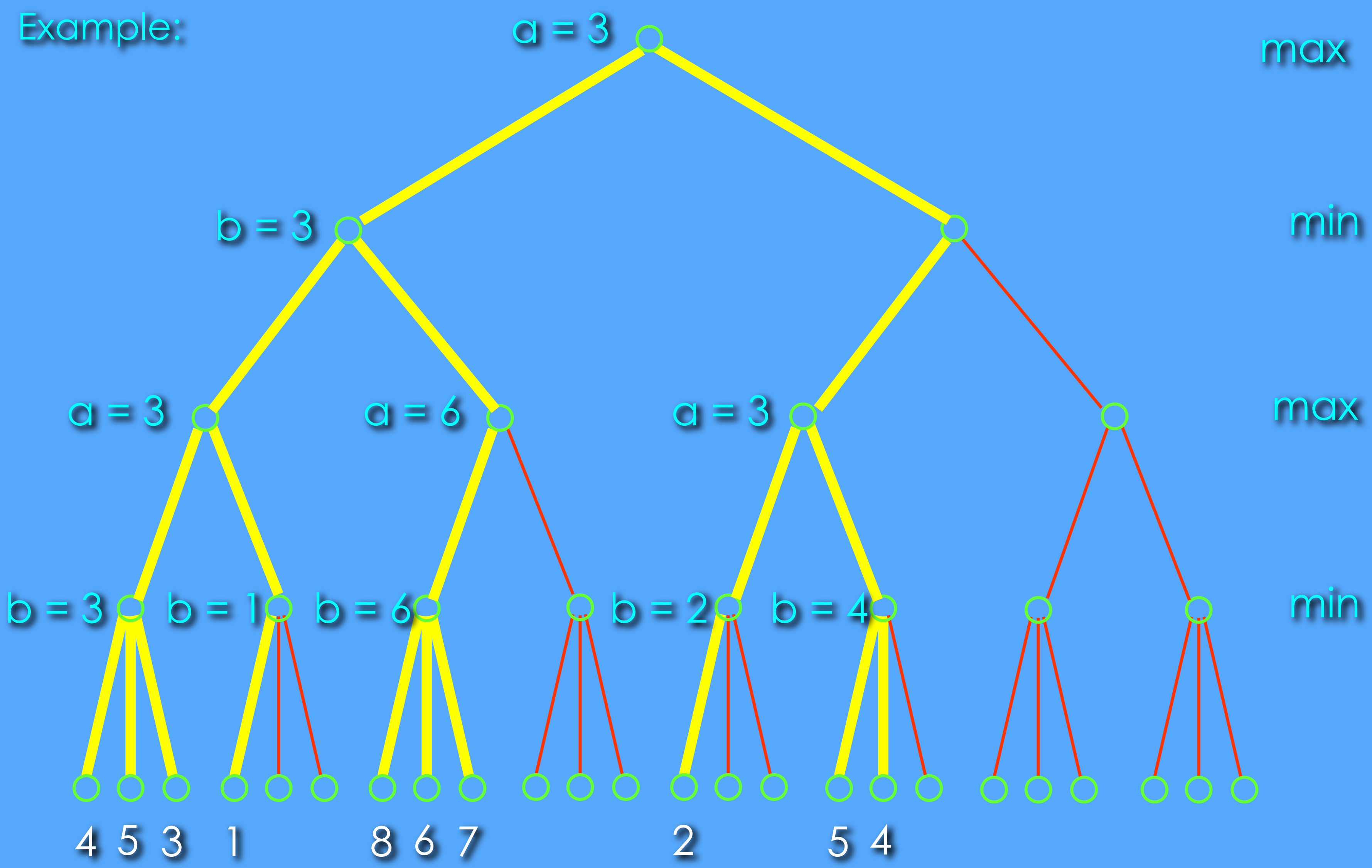
The Alpha-Beta Procedure

Example:



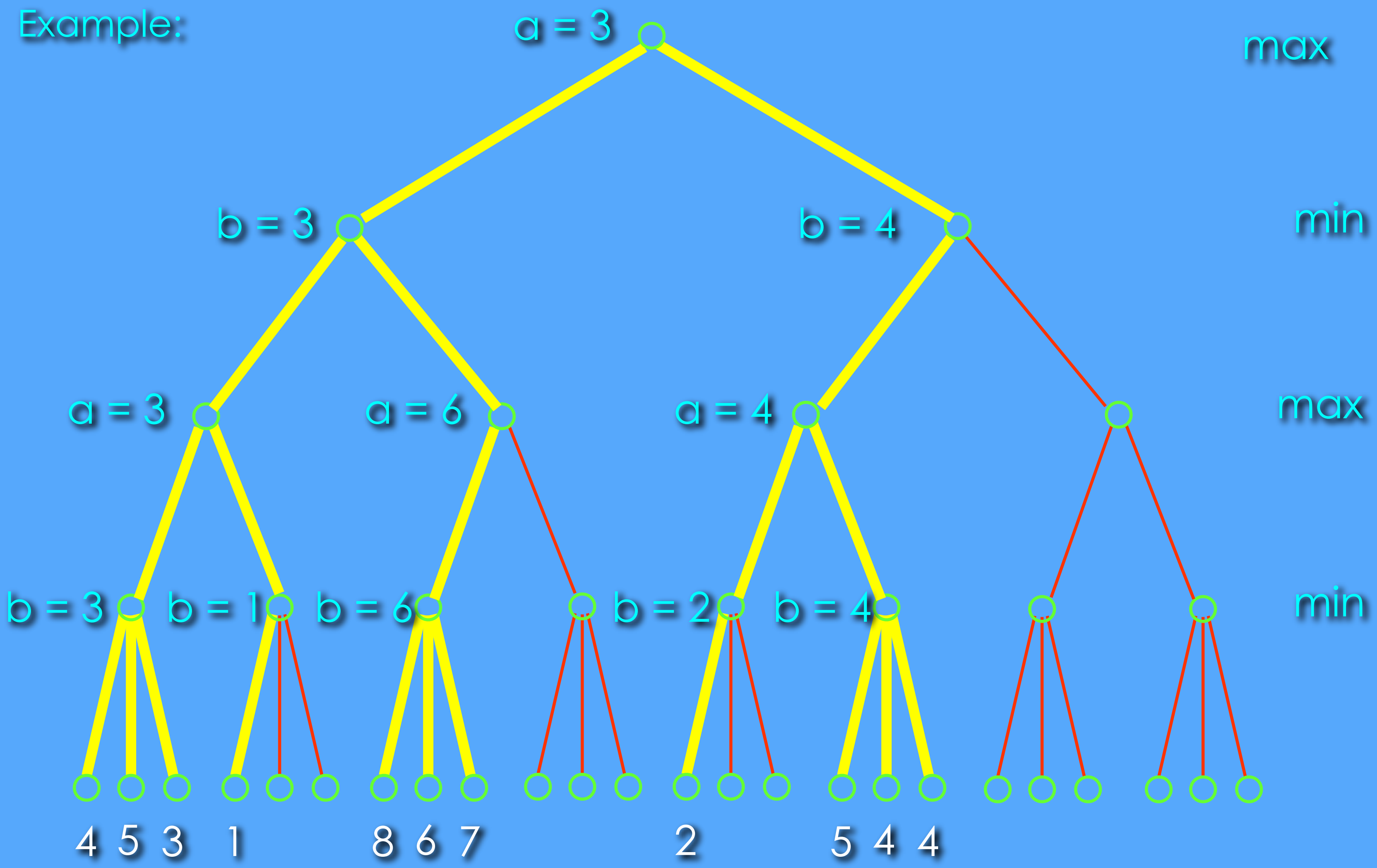
The Alpha-Beta Procedure

Example:



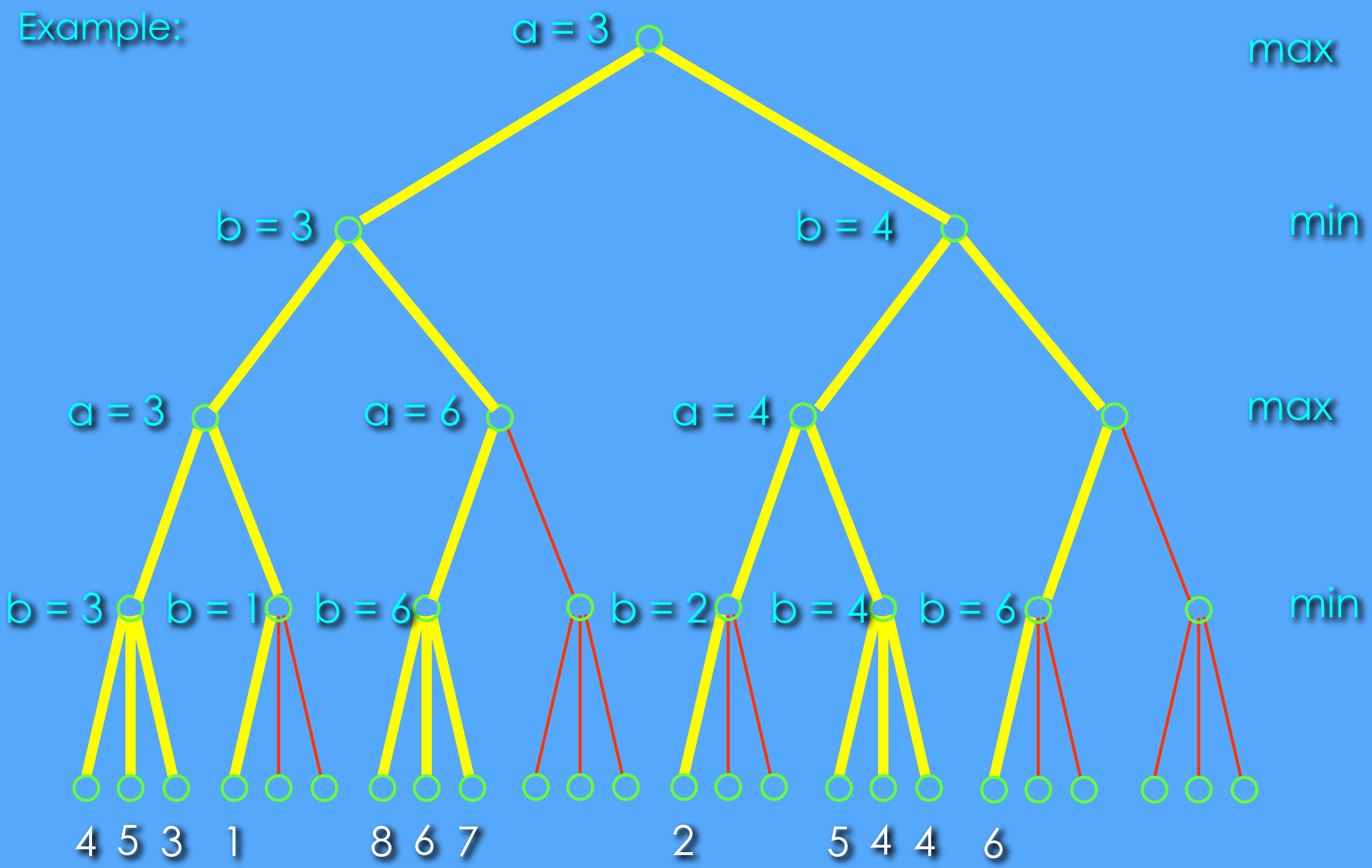
The Alpha-Beta Procedure

Example:



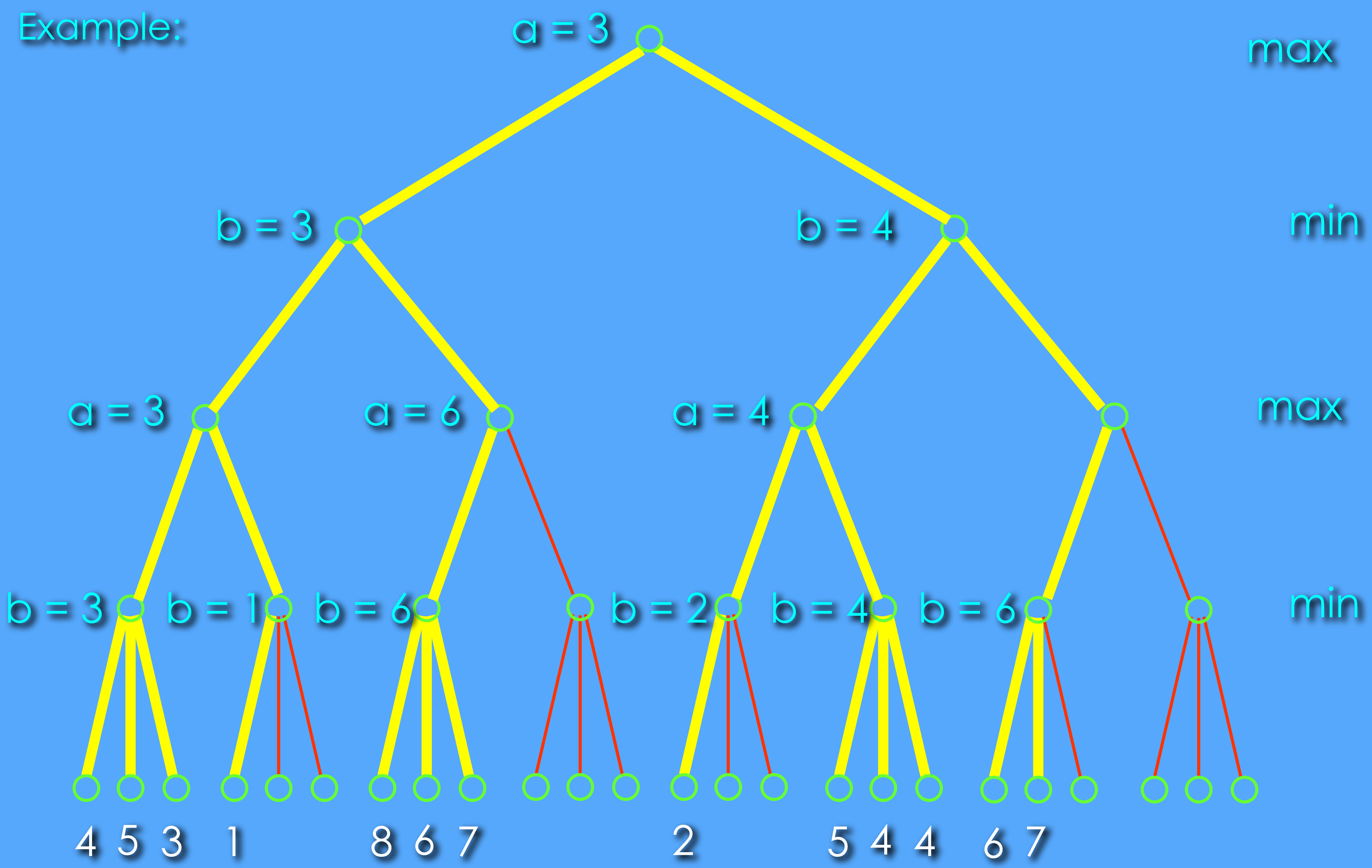
The Alpha-Beta Procedure

Example:



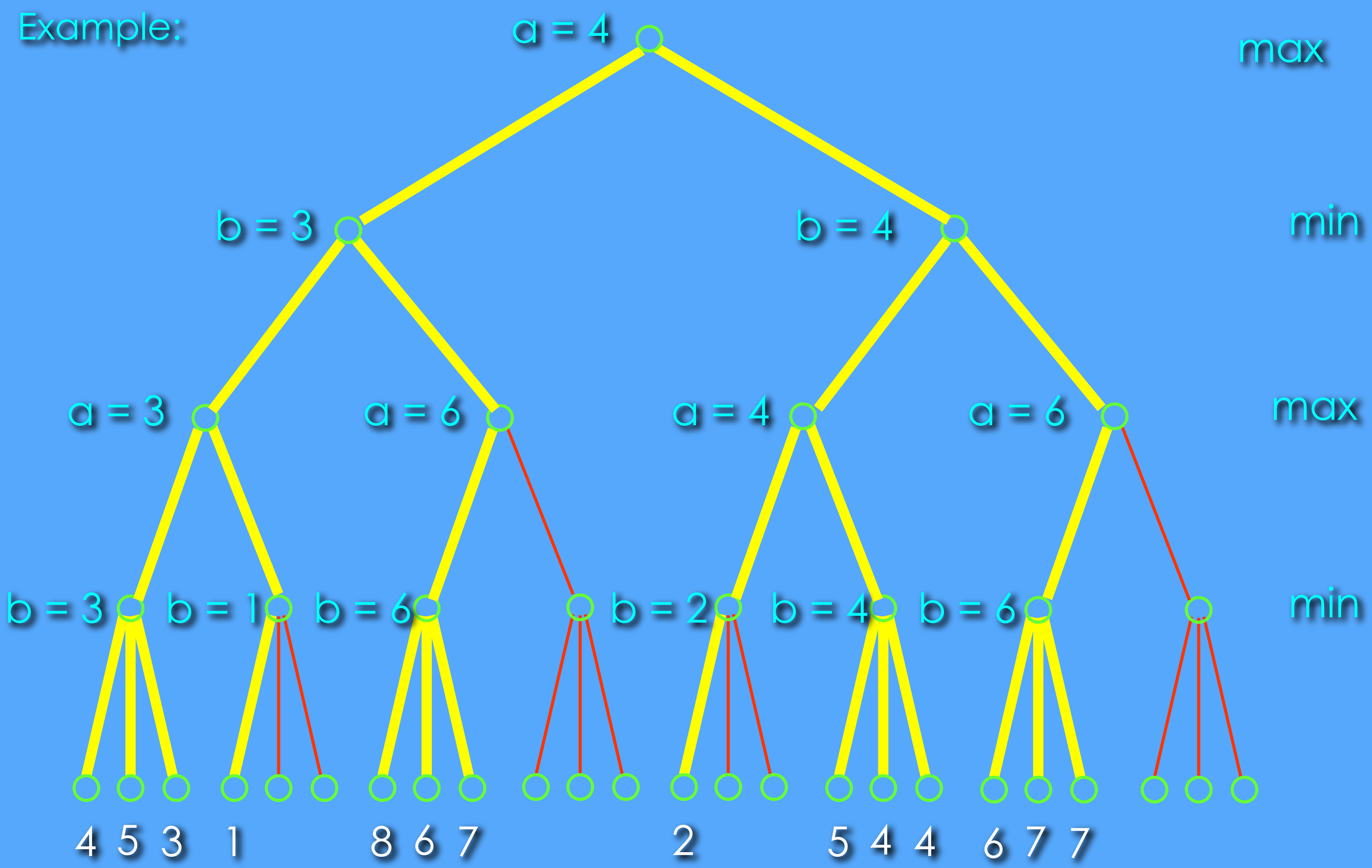
The Alpha-Beta Procedure

Example:



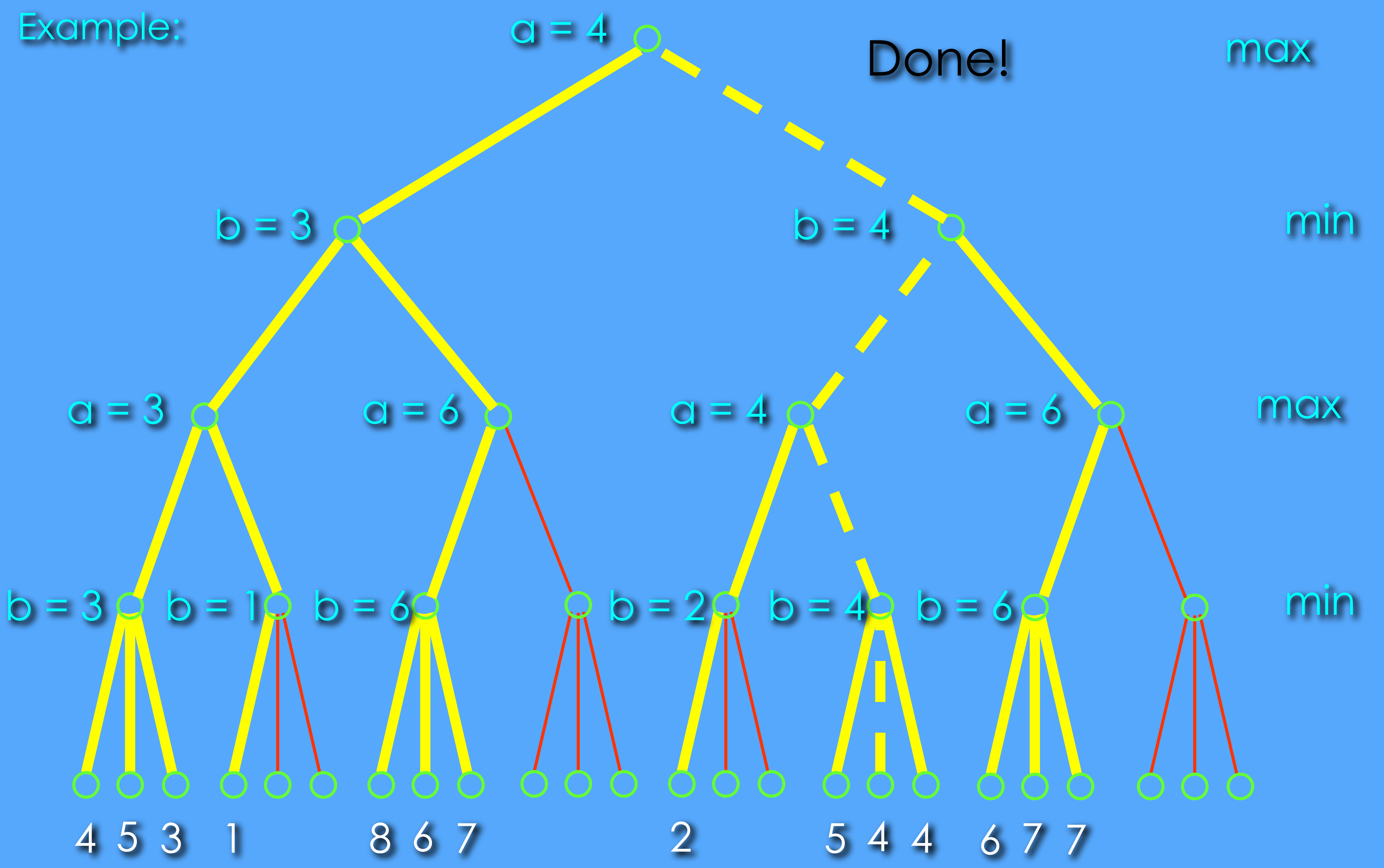
The Alpha-Beta Procedure

Example:



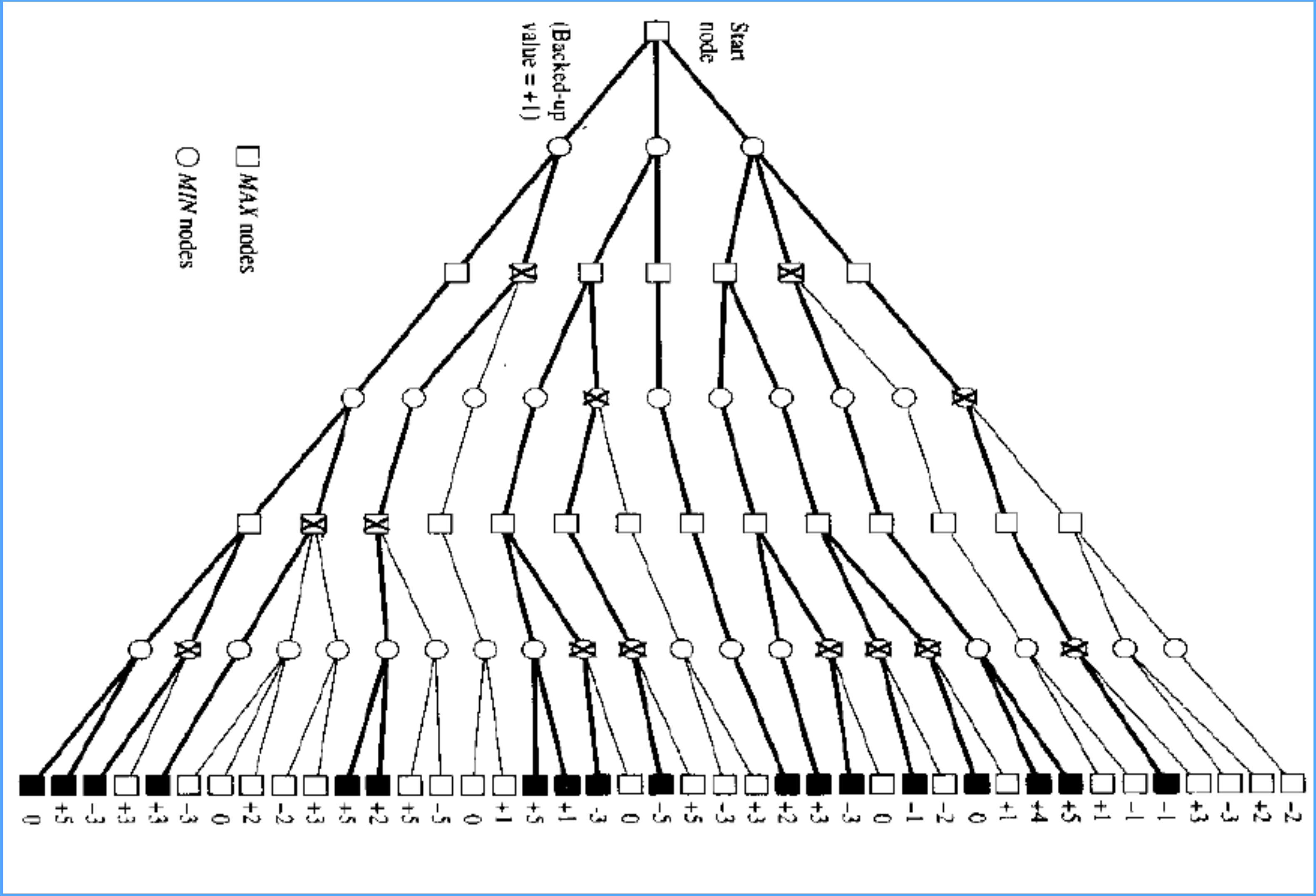
The Alpha-Beta Procedure

Example:



Nodes pruned by α–β procedure

Another example



The Alpha-Beta Procedure

- Can we estimate the efficiency benefit of the alpha-beta method?

- Suppose that there is a game that always allows a player to choose among **b** different moves, and we want to look **d** moves ahead.

- Then our search tree has **b^d leaves**.

- Therefore, if we do not use alpha-beta pruning, we would have to apply the static evaluation function $N_d = b^d$ times.

The Alpha-Beta Procedure

- Of course, the efficiency gain by the alpha-beta method always depends on the rules and the current configuration of the game.
- However, if we assume that new children of a node are explored in a particular order - those nodes p are explored first that will yield maximum values e(p) at depth d for MAX and minimum values for MIN - the number of nodes to be evaluated is:

Timing Issues

- It is very difficult to predict for a given game situation how many operations a depth d look-ahead will require.
- Since we want the computer to respond within a certain amount of time, it is a good idea to apply the idea of ***iterative deepening***.
- First, the computer finds the best move according to a ***one-move look-ahead*** search.
- Then, the computer determines the best move for a ***two-move look-ahead***, and remembers it as the new best move.
- This is ***continued*** until the time runs out. Then the currently remembered best move is executed.

How to Find Static Evaluation Functions

- Often, a static evaluation function e(p) first computes an appropriate feature vector f(p) that contains information about features of the current game configuration that are important for its evaluation.
- There is also a weight vector w(p) that indicates the weight (= importance) of each feature for the assessment of the current situation.
- Then e(p) is simply computed as the **scalar product** of f(p) and w(p).
- Both the identification of the most relevant features and the correct estimation of their relative importance are **crucial** for the strength of a game-playing program.

Evaluation function for chess

- For games like 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.

- Weights 9 for queen, 5 for rook, 3 for bishop and knight and 1 for pawn – suggested by Shannon is still widely used.

How to Find Static Evaluation Functions

- Once we have found suitable features, the weights can be adapted algorithmically.
- This can be achieved, for example, with a neural network.
- So the challenge is in extracting the most informative features from a game configuration.

Heuristics and Game Tree Search

- How to determine the depth of evaluation? (clearly the time or other resource is one guide.)

- The Horizon Effect
  - sometimes there's a major "effect" (such as a piece being captured) which is just "below" the depth to which the tree has been expanded
  - the computer cannot see that this major event could happen
  - it has a "limited horizon"
  - there are heuristics to try to follow certain branches more deeply to detect to such important events.
  - this helps to avoid catastrophic losses due to "short-sightedness"

Heuristics and Game Tree Search

Heuristics for Tree Exploration

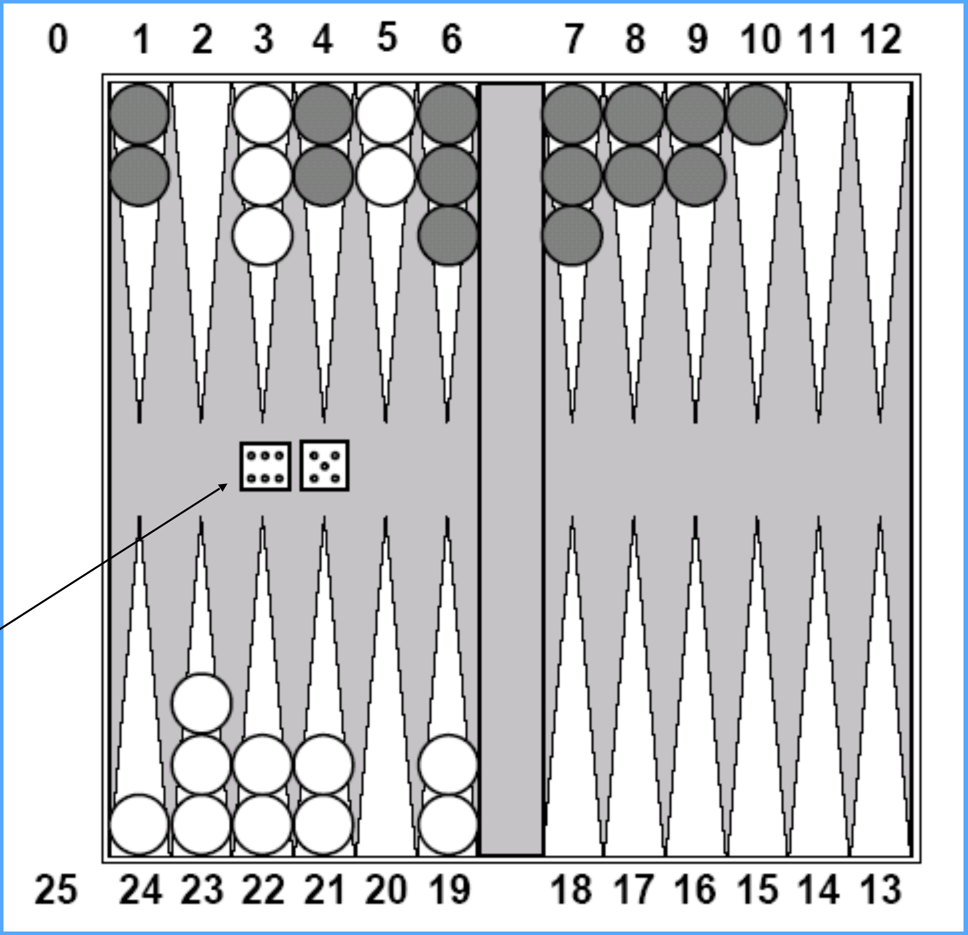
- it may be better to explore some branches more deeply in the allotted time (forward pruning)
- various heuristics exist to identify "promising" branches
  - expand to some depth k, and rank the nodes by the static evaluation function and choose the best m nodes to expand deeper.
  - for each node, determine the uncertainty of the evaluation function and go deeper on the most uncertain ones.
  - determine if a position is "active" or "passive". For active nodes, explore deeper.

Forward Pruning & Lookup tables

- Humans don’t consider all possible moves.
- Can we prune certain branches immediately?
- Use estimates (from past experience) of the uncertainty in the estimate of the node’s value and uses that to decide if a node can be pruned.
- Instead of search one can also store game states.
- Openings in chess are played from a library
- Endgames have often been solved and stored as well.

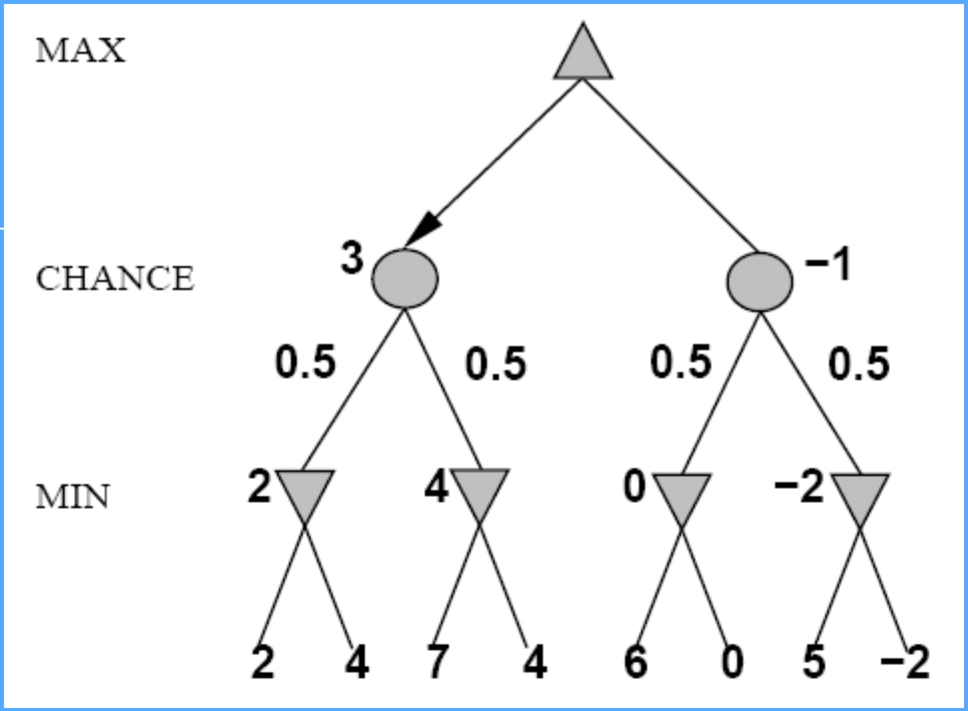
Chance Games

Backgammon



element of chance

Expected Minimax (Expectimax)



Again, the tree is constructed bottom-up.

Now we have even more nodes to search!

Summary

- Game playing is best modeled as a search problem
- Game trees represent alternate computer/opponent moves
- Evaluation functions estimate the quality of a given board configuration for the Max player.
- Minmax is a procedure which chooses moves by assuming that the opponent will always choose the move which is best for them
- Alpha-Beta is a procedure which can prune large parts of the search tree and allow search to go deeper
- For many well-known games, computer algorithms based on heuristic search match or out-perform human world experts.