

# Adversarial Search and Game-Playing

# Search versus Games

- Search — no adversary
  - Solution is (heuristic) method for finding goal
  - Heuristic techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities
- Games — adversary
  - Solution is **strategy** (strategy specifies move for every possible opponent reply).
  - **Optimality depends on opponent.** Why?
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate “goodness” of game position
  - Examples: chess, checkers, Othello, backgammon

# Adversarial Search

- Examine the problems that arise when we try to plan ahead in a world where other agents are planning against us.
- A good example is in board games.
- Two agents whose actions alternate
- Utility values for each agent are the opposite of the other
  - creates the adversarial situation
- Fully observable environments
- In game theory terms: Zero-sum games of perfect information.

# Types of game in AI

	Deterministic	Chance Moves
Perfect information	Chess,	Backgammon,
Imperfect information	Battleships,	poker

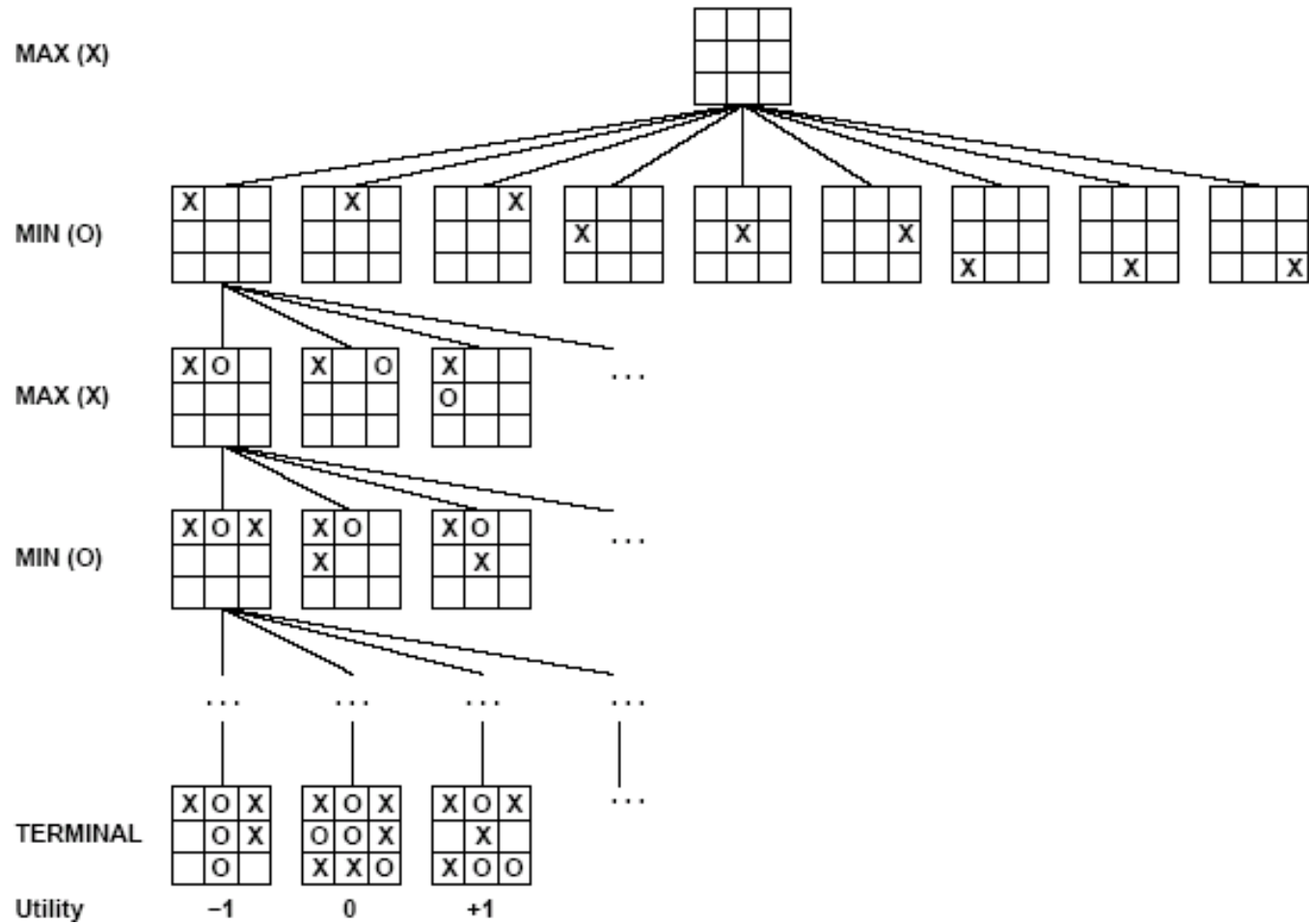
# Game Setup

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over
  - Winner gets award, loser gets penalty.
- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (move,state) pairs specifying legal moves.
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe or chess

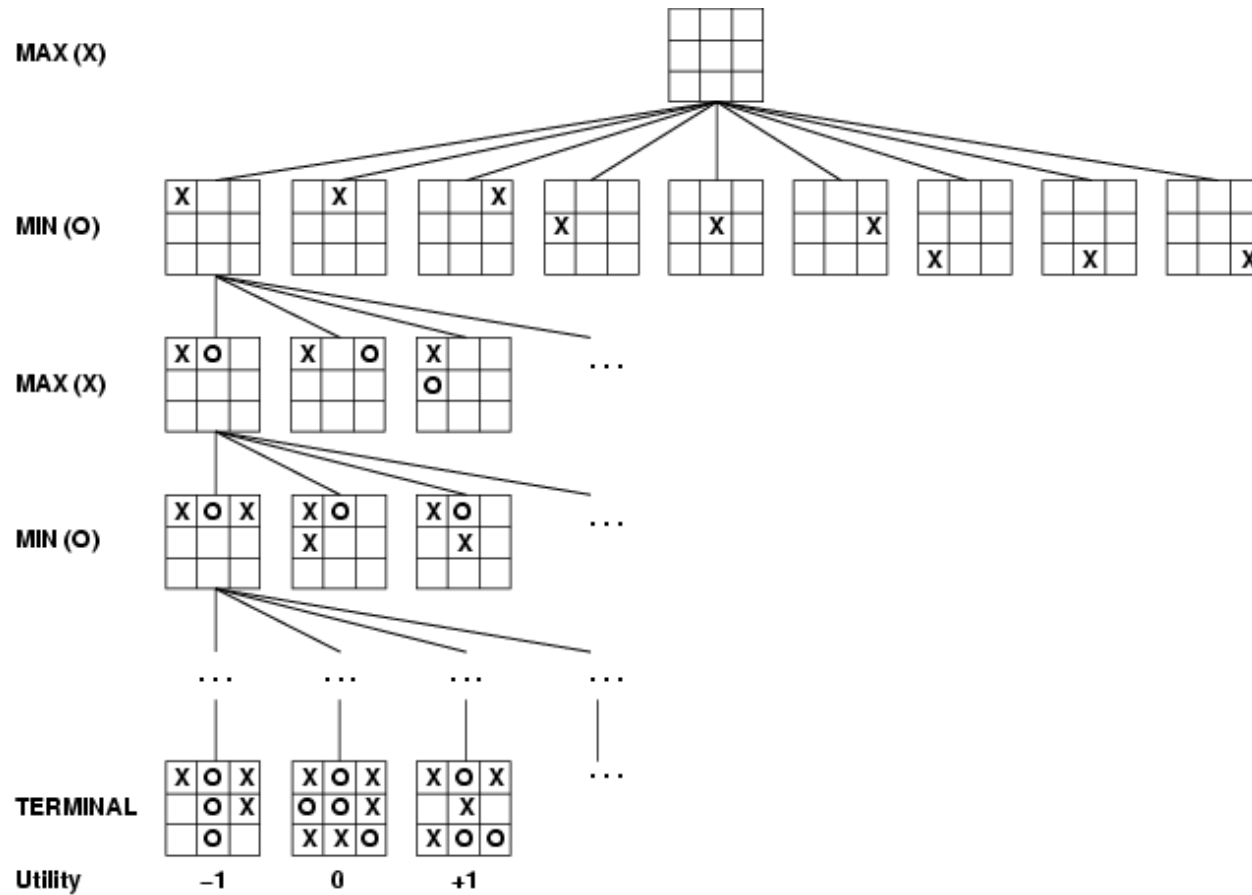
# Size of search trees

- $b$  = branching factor
- $d$  = number of moves by both players
- Search tree is  $O(b^d)$
- Chess
  - $b \sim 35$
  - $D \sim 100$ 
    - search tree is  $\sim 10^{154}$  (!!)
    - completely impractical to search this
- Game-playing emphasizes being able to make optimal decisions in a finite amount of time
  - Somewhat realistic as a model of a real-world agent
  - Even if games themselves are artificial

## Partial Game Tree for Tic-Tac-Toe



# Game tree (2-player, deterministic, turns)



How do we search this tree to find the optimal move?



## Minimax strategy: Look ahead and reason backwards

- Find the optimal *strategy* for MAX assuming an infallible MIN opponent
  - Need to compute this all the way down the tree
- Assumption: Both players play optimally!
- Given a game tree, the optimal strategy can be determined by using the **minimax value of each node**.

.

# Mini-max algorithm

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

# Pseudocode for Minimax Algorithm

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

**inputs:** *state*, current state in game

$v \leftarrow \text{MAX-VALUE}(\textit{state})$

**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**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$

**return**  $v$

---

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

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

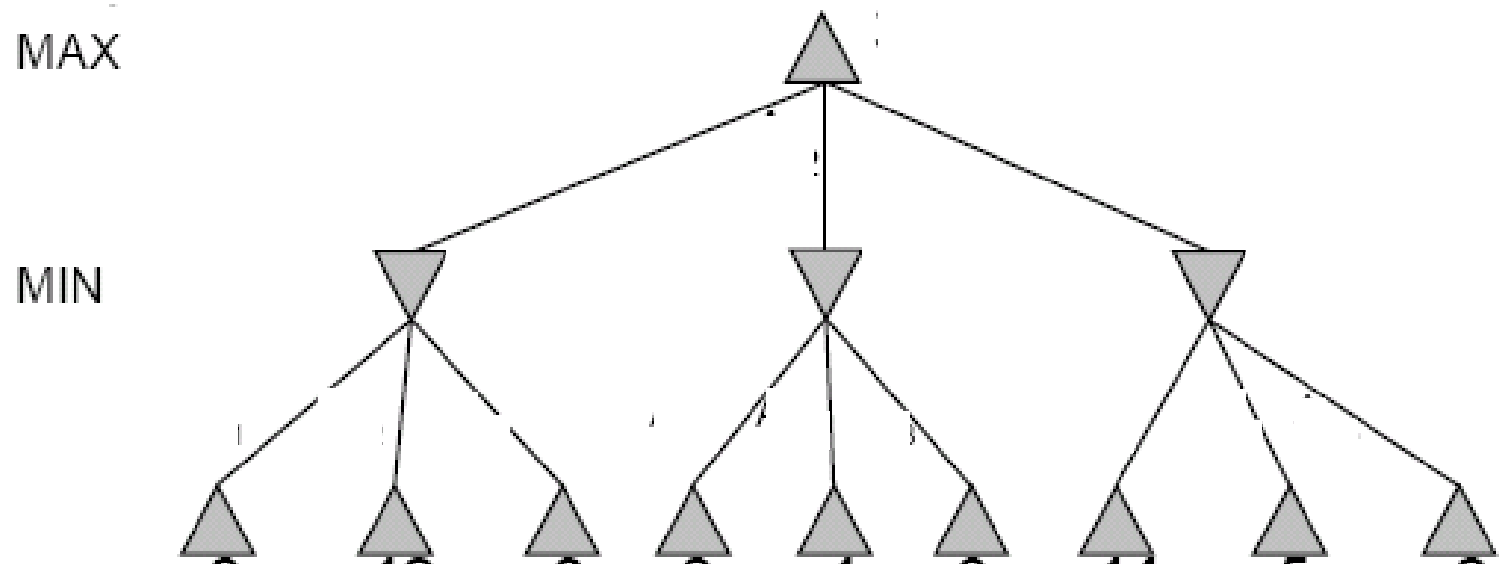
$v \leftarrow \infty$

**for**  $a, s$  in SUCCESSORS(*state*) **do**

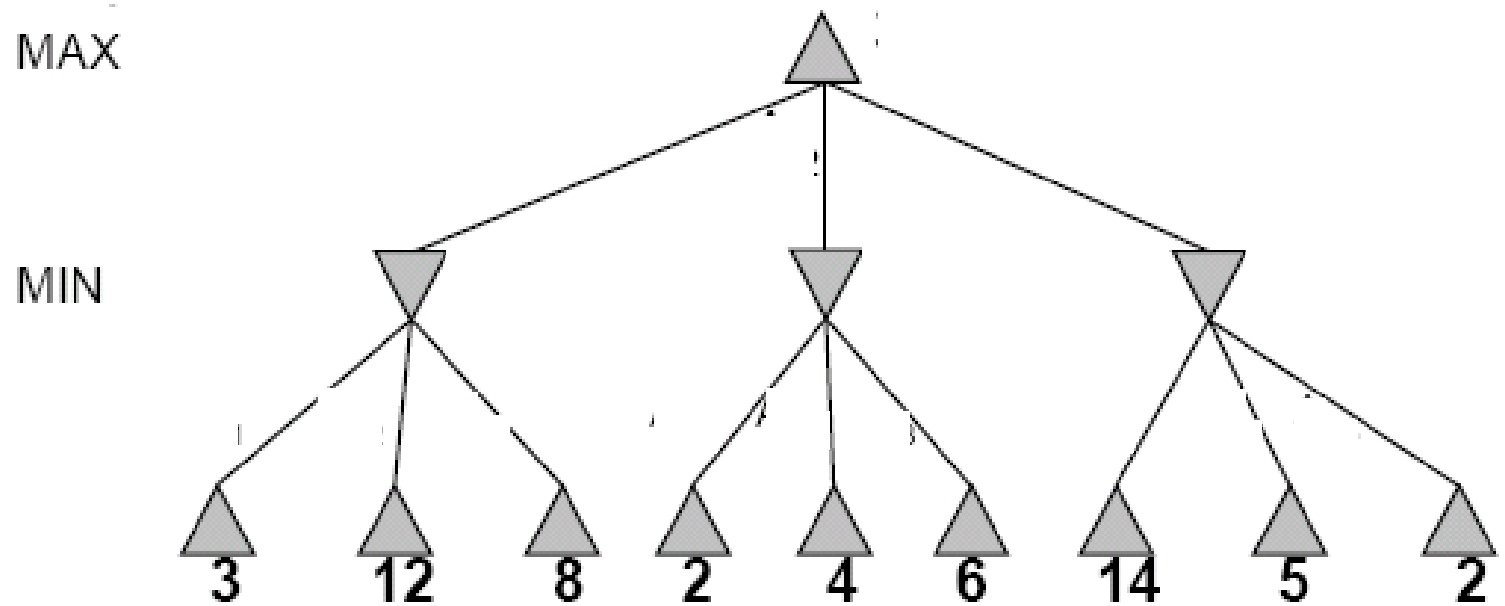
$v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$

**return**  $v$

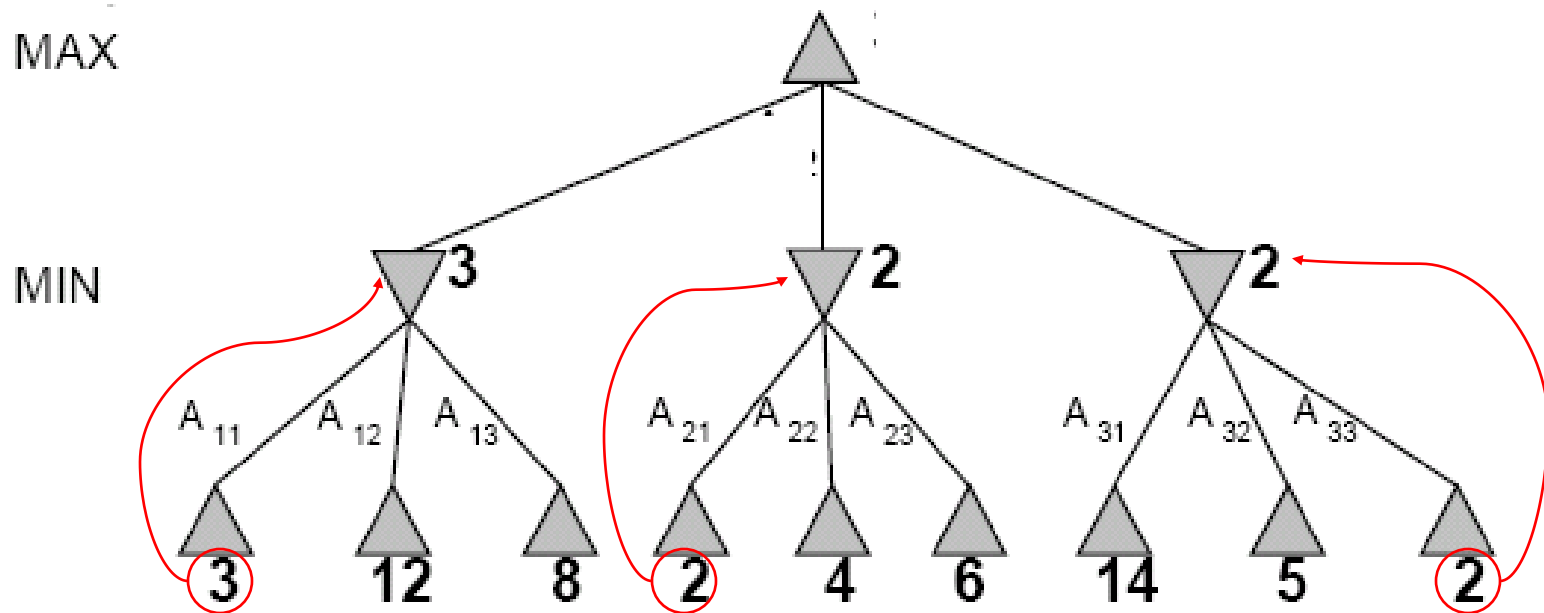
# Two-Ply Game Tree



# Two-Ply Game Tree

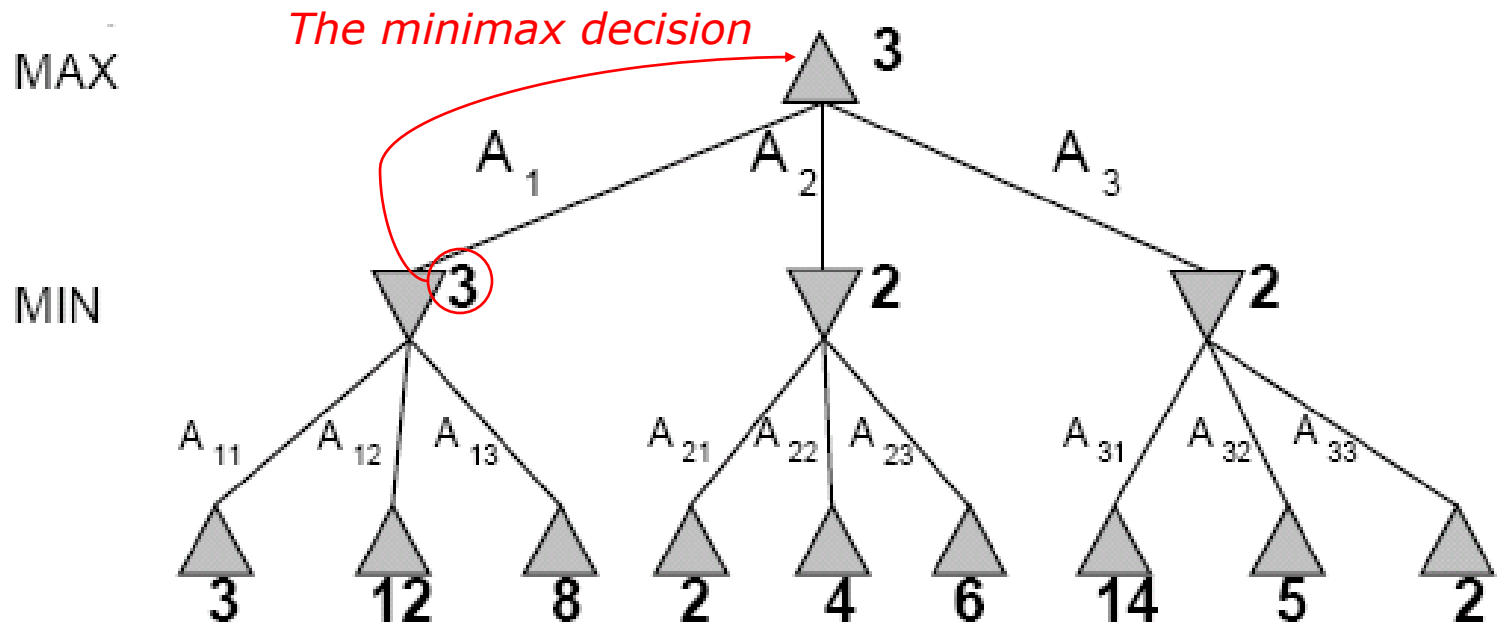


# Two-Ply Game Tree



# Two-Ply Game Tree

**Minimax maximizes the utility for the worst-case outcome for max**



# Properties of Mini-Max algorithm:

- **Complete**- Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal**- Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity**- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is  $O(b^m)$ , where  $b$  is branching factor of the game-tree, and  $m$  is the maximum depth of the tree.
- **Space Complexity**- Space complexity of Mini-max algorithm is also similar to DFS which is  $O(bm)$ .

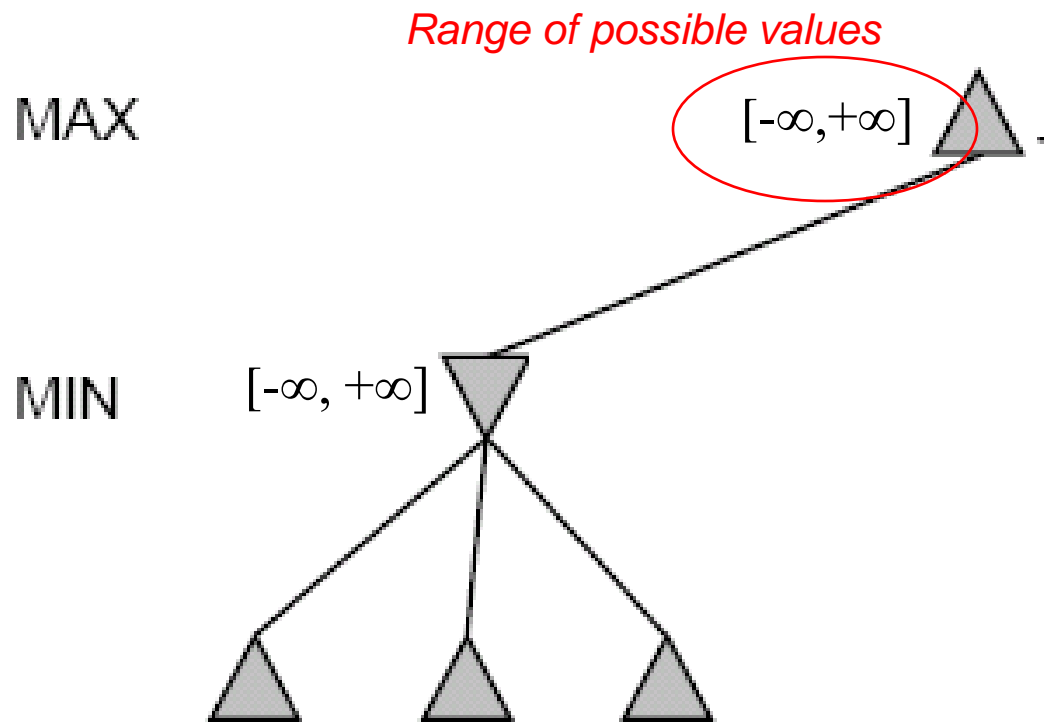


# Practical problem with minimax search

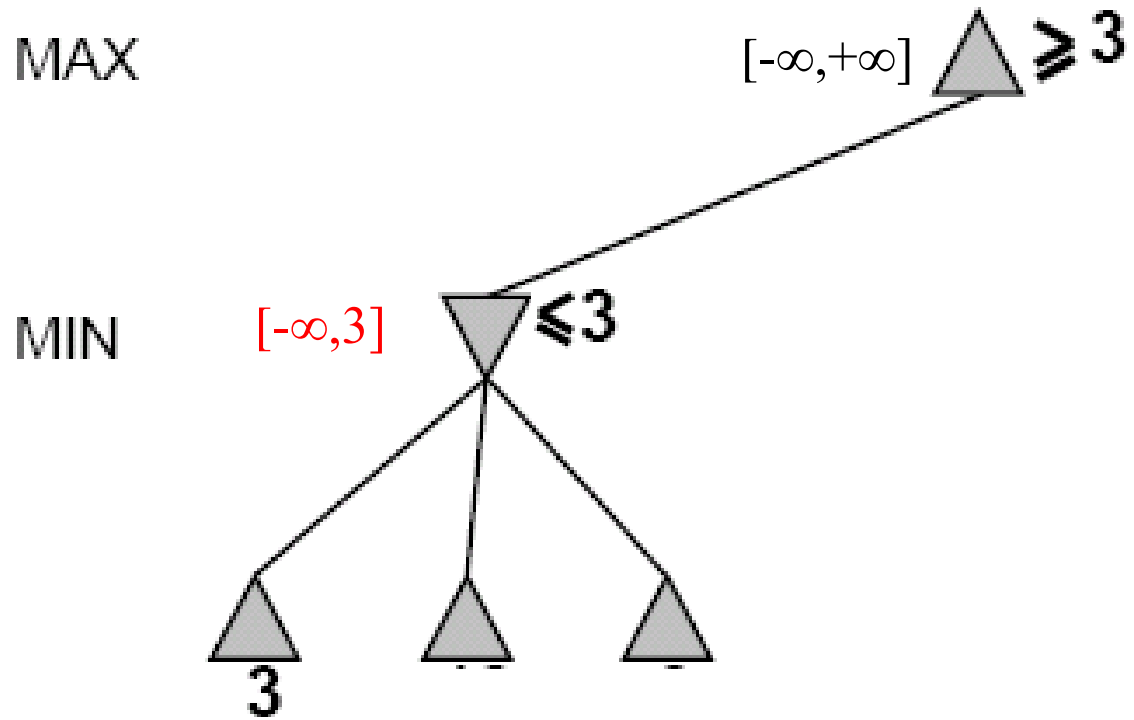
- Number of game states is exponential in the number of moves.
  - Solution: Do not examine every node
    - => pruning
      - Remove branches that do not influence final decision
- Revisit example ...

# Alpha-Beta Example

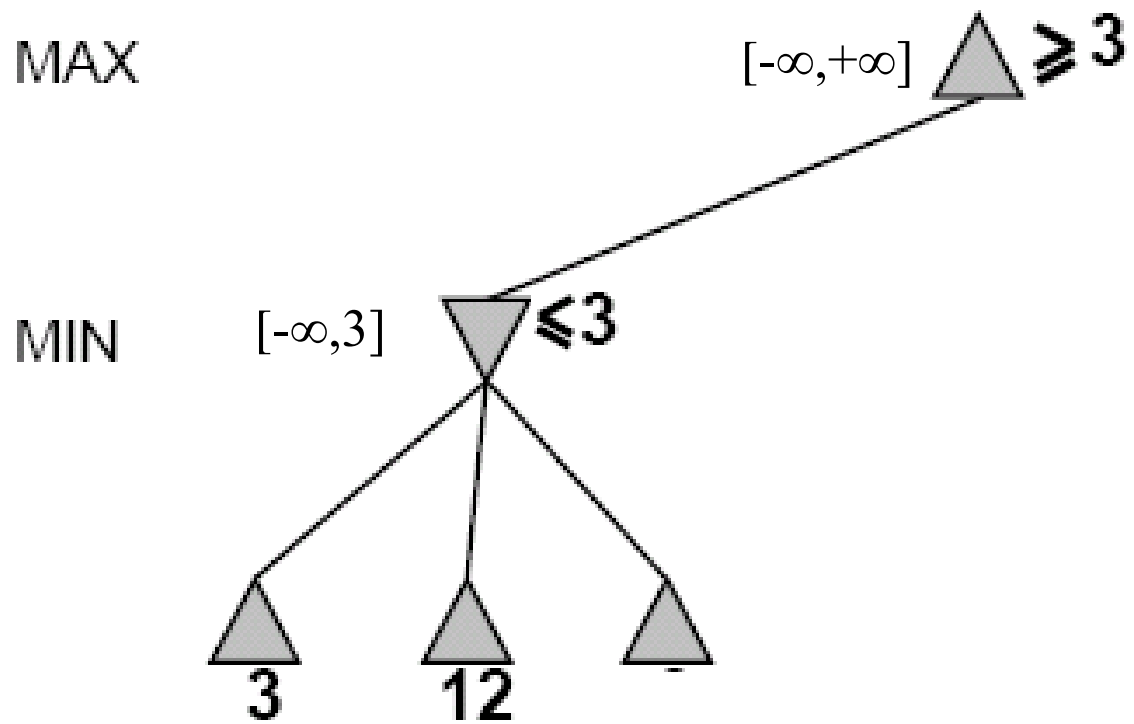
**Do DF-search until first leaf**



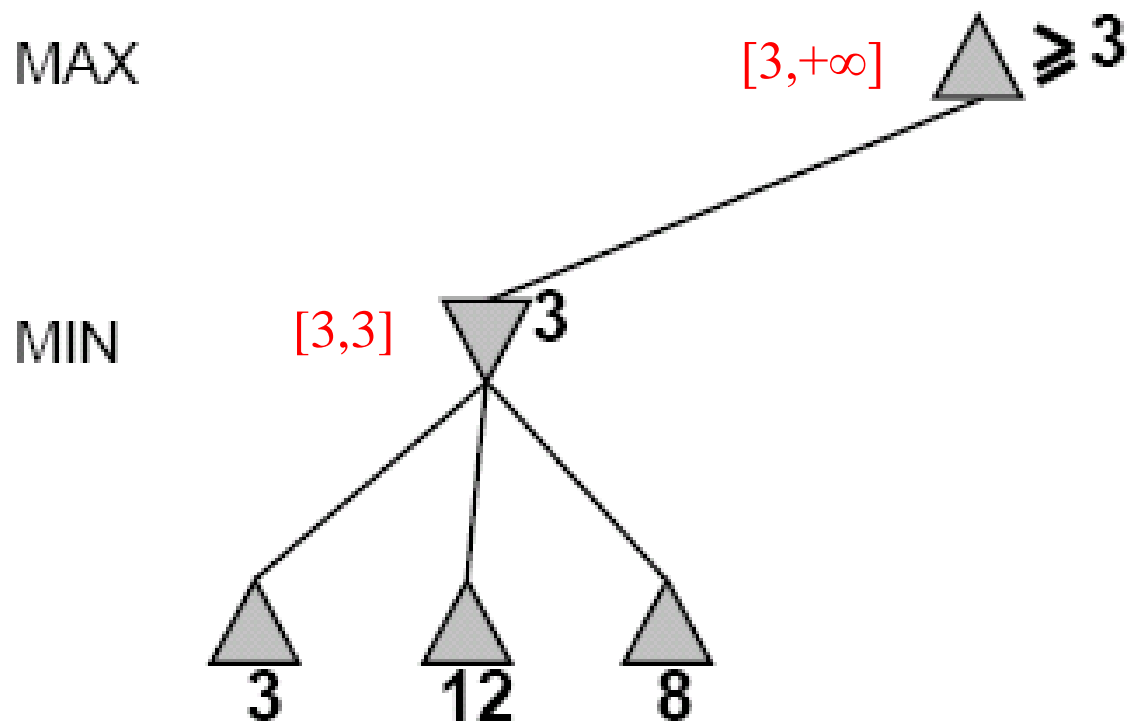
## Alpha-Beta Example (continued)



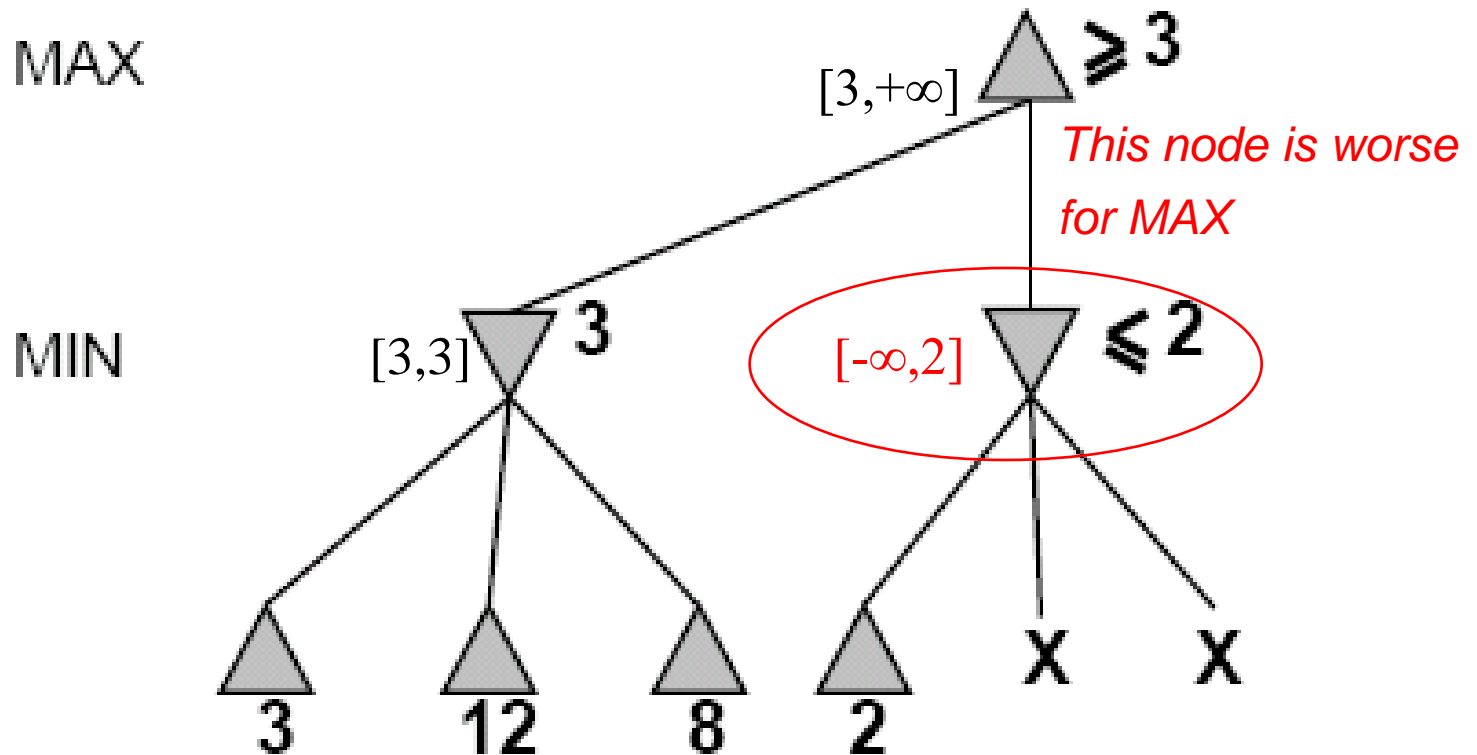
## Alpha-Beta Example (continued)



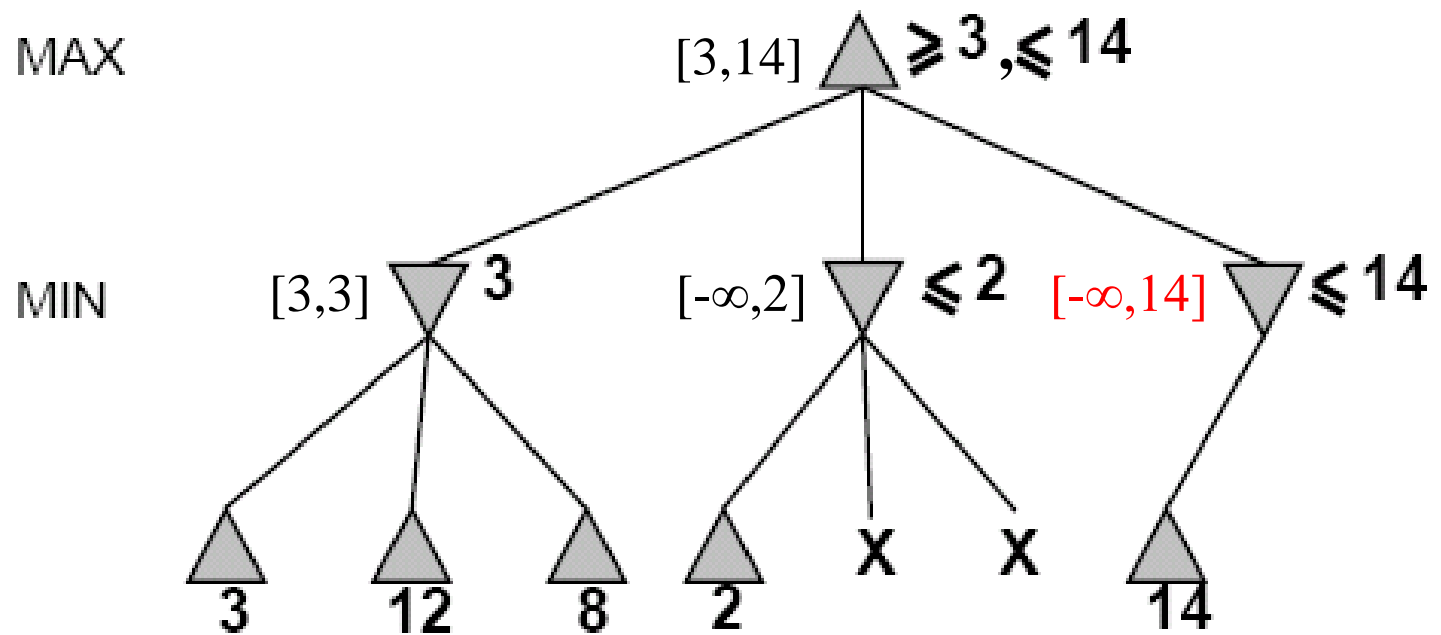
## Alpha-Beta Example (continued)



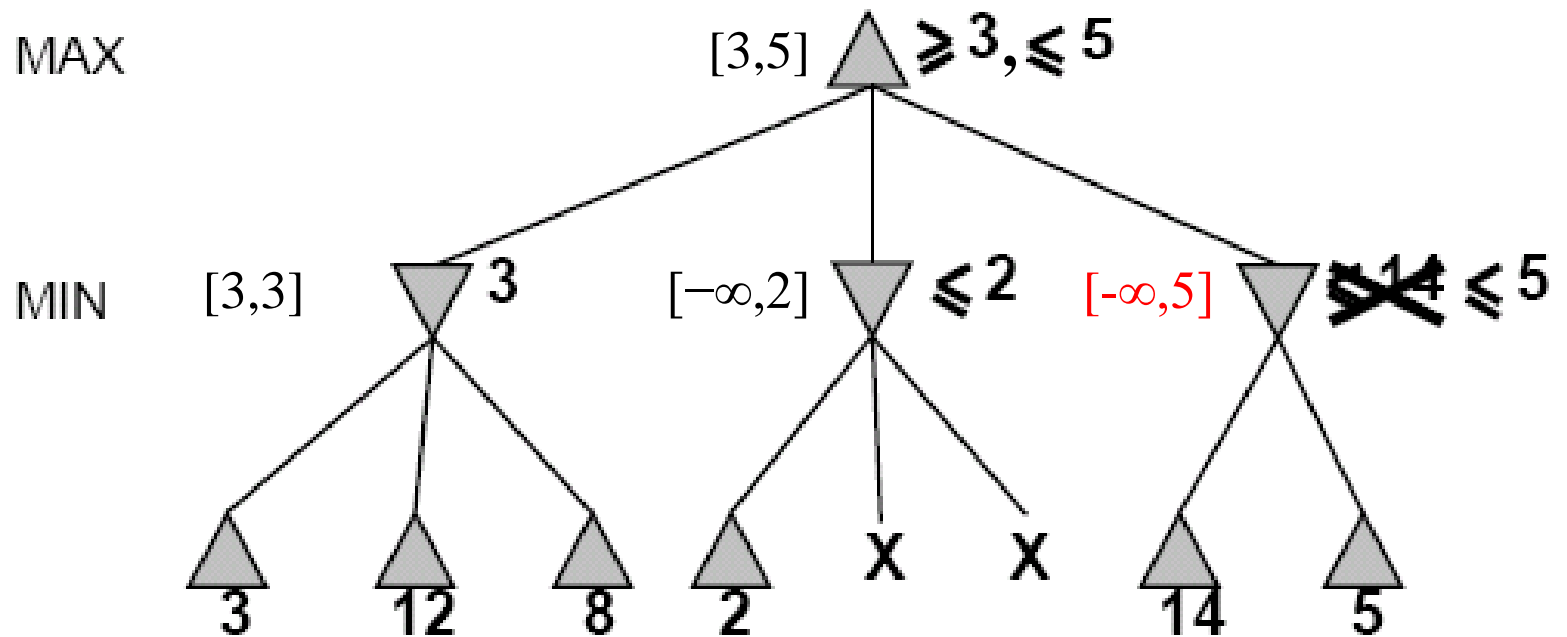
## Alpha-Beta Example (continued)



## Alpha-Beta Example (continued)

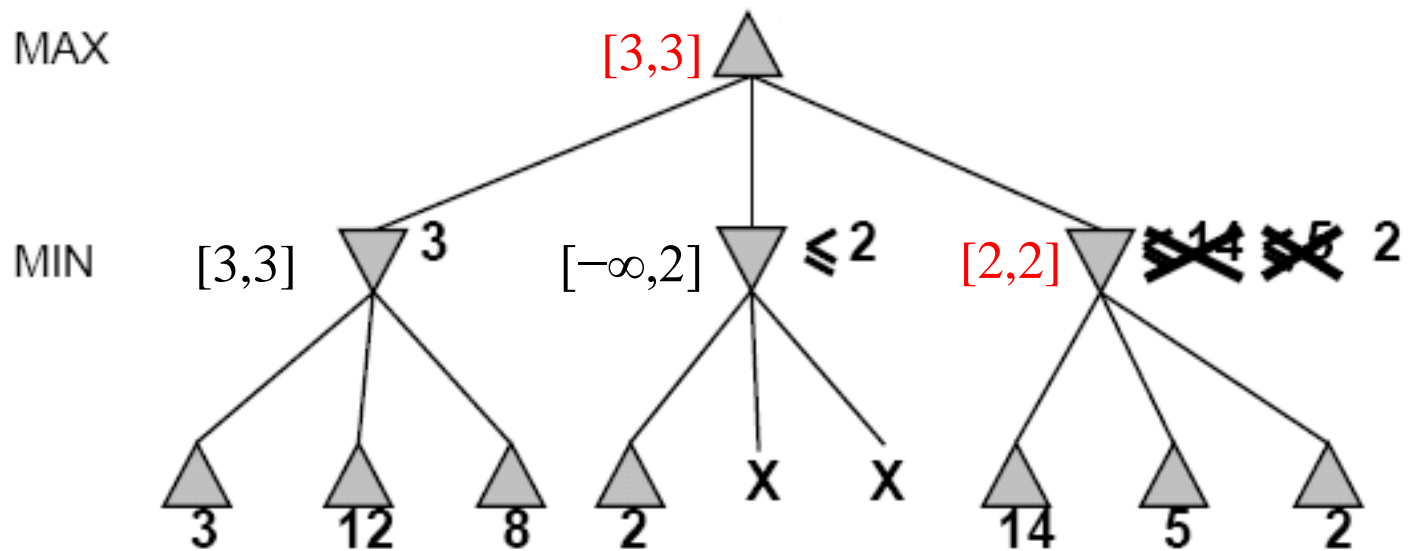


## Alpha-Beta Example (continued)

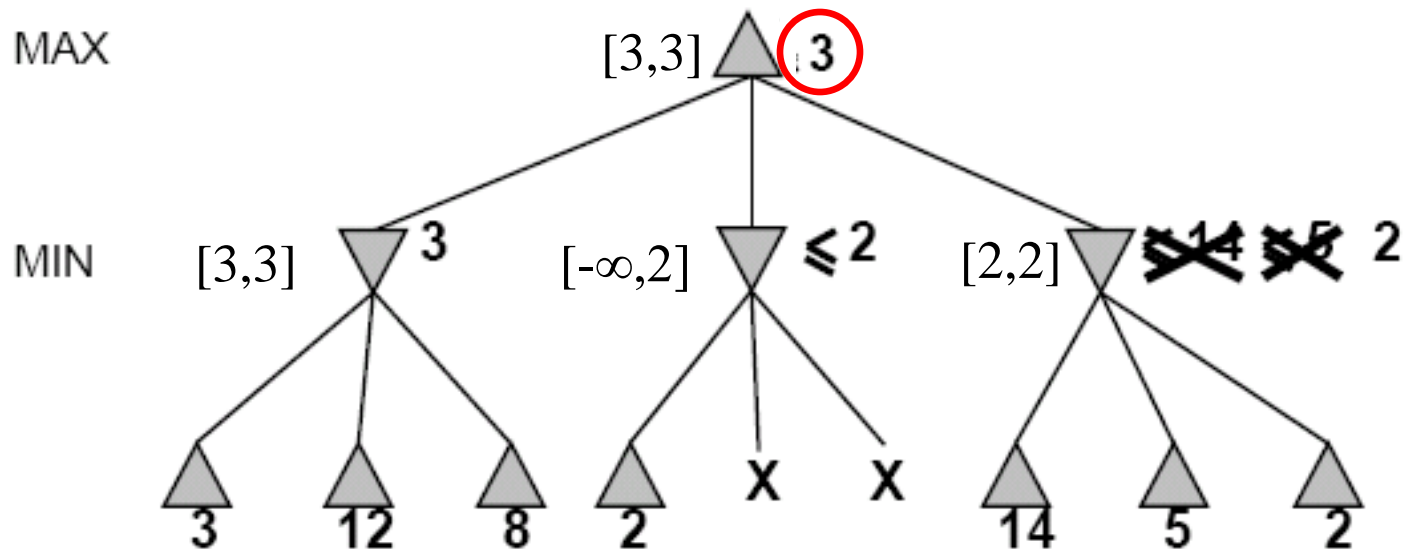




## Alpha-Beta Example (continued)



## Alpha-Beta Example (continued)



# Alpha-beta Algorithm

- Depth first search – only considers nodes along a single path at any time

$\alpha$  = highest-value choice that we can guarantee for MAX so far in the current subtree.

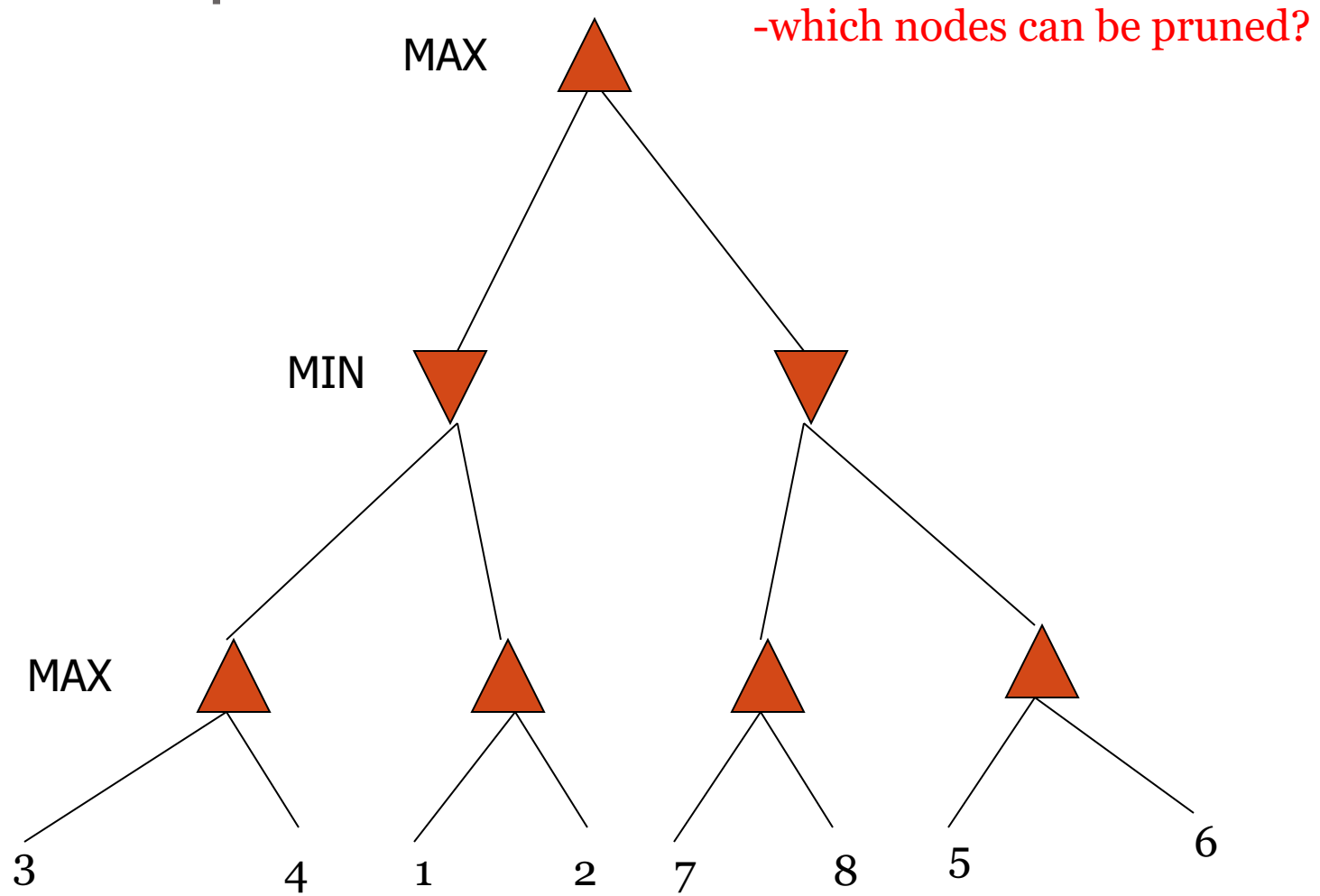
$\beta$  = lowest-value choice that we can guarantee for MIN so far in the current subtree.

- update values of  $\alpha$  and  $\beta$  during search and prunes remaining branches as soon as the value is known to be worse than the current  $\alpha$  or  $\beta$  value for MAX or MIN.
- Alpha-beta Demo.

# Effectiveness of Alpha-Beta Search

- Worst-Case
  - branches are ordered so that no pruning takes place. In this case alpha-beta gives no improvement over exhaustive search
- Best-Case
  - each player's best move is the left-most alternative (i.e., evaluated first)
  - in practice, performance is closer to best rather than worst-case
- In practice often get  $O(b^{(d/2)})$  rather than  $O(b^d)$ 
  - this is the same as having a branching factor of  $\sqrt{b}$ ,
    - since  $(\sqrt{b})^d = b^{(d/2)}$
    - i.e., we have effectively gone from  $b$  to square root of  $b$
  - e.g., in chess go from  $b \sim 35$  to  $b \sim 6$ 
    - this permits much deeper search in the same amount of time
    - Typically twice as deep.

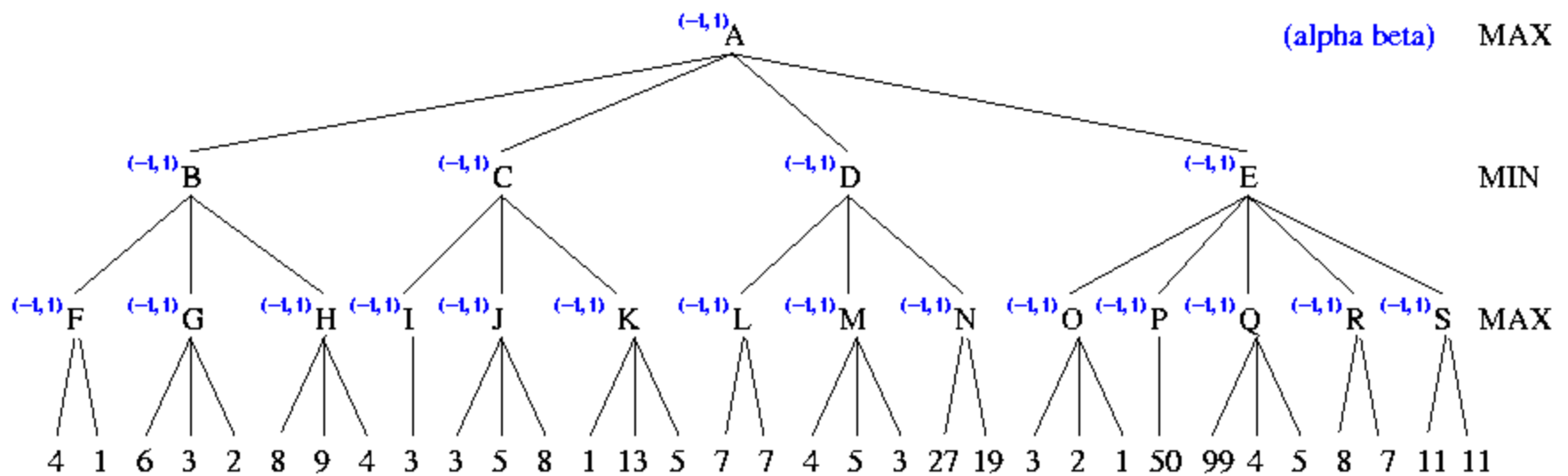
# Example



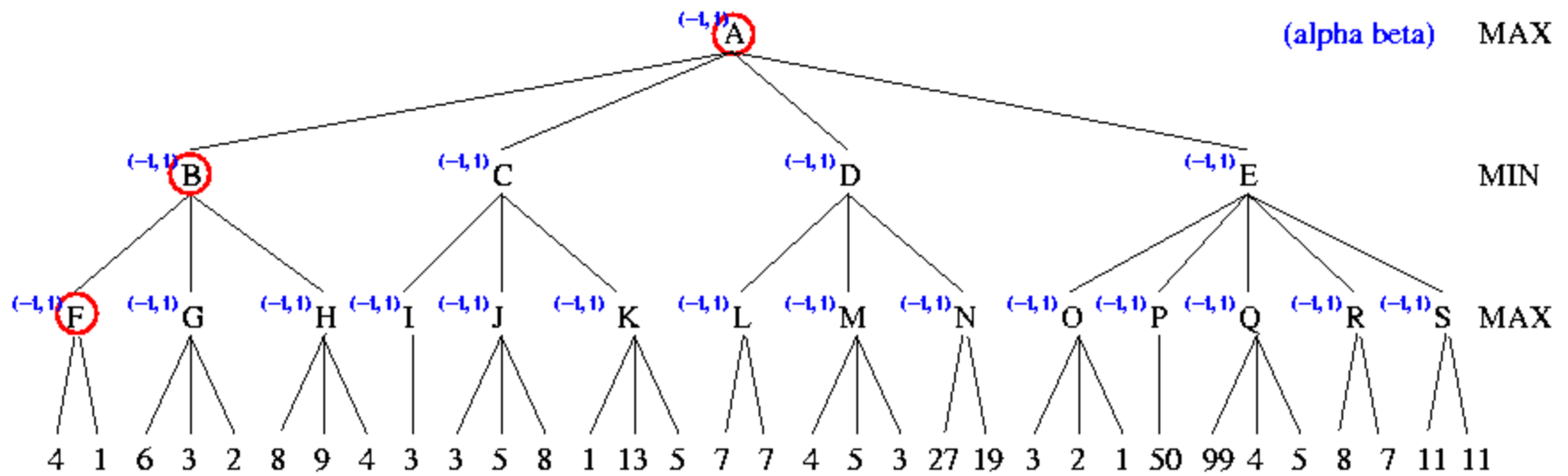
## Final Comments about Alpha-Beta Pruning

- Pruning does not affect final results
- Entire subtrees can be pruned.
- Good move *ordering* improves effectiveness of pruning
- Repeated states are again possible.
  - Store them in memory = transposition table

# Example

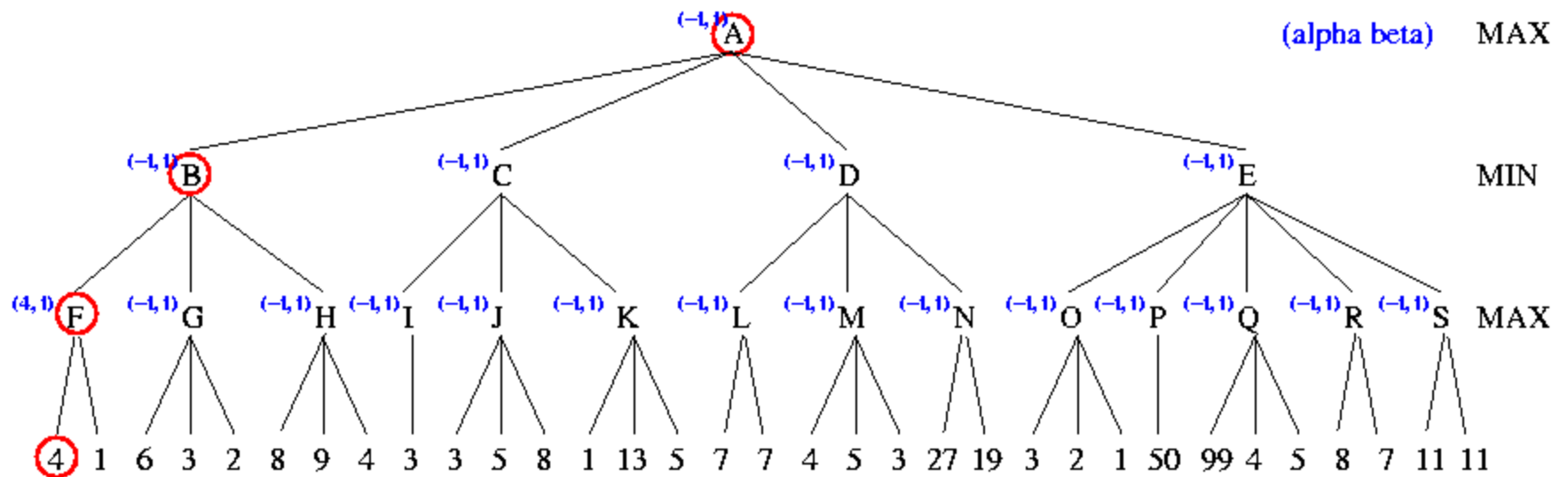


# Example

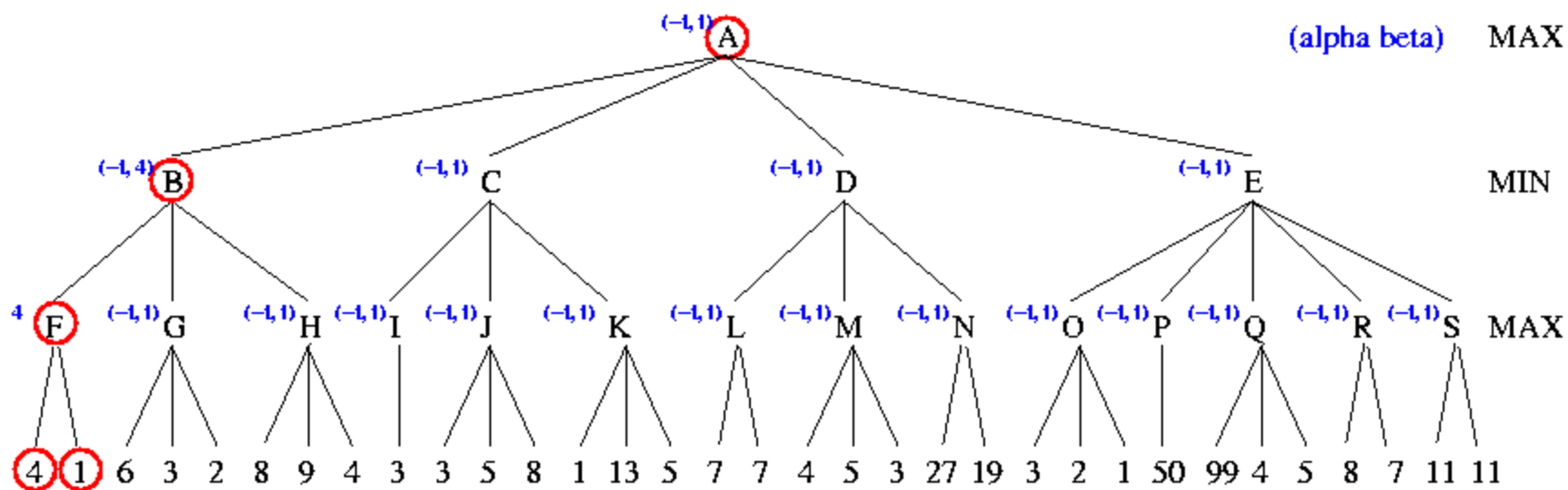




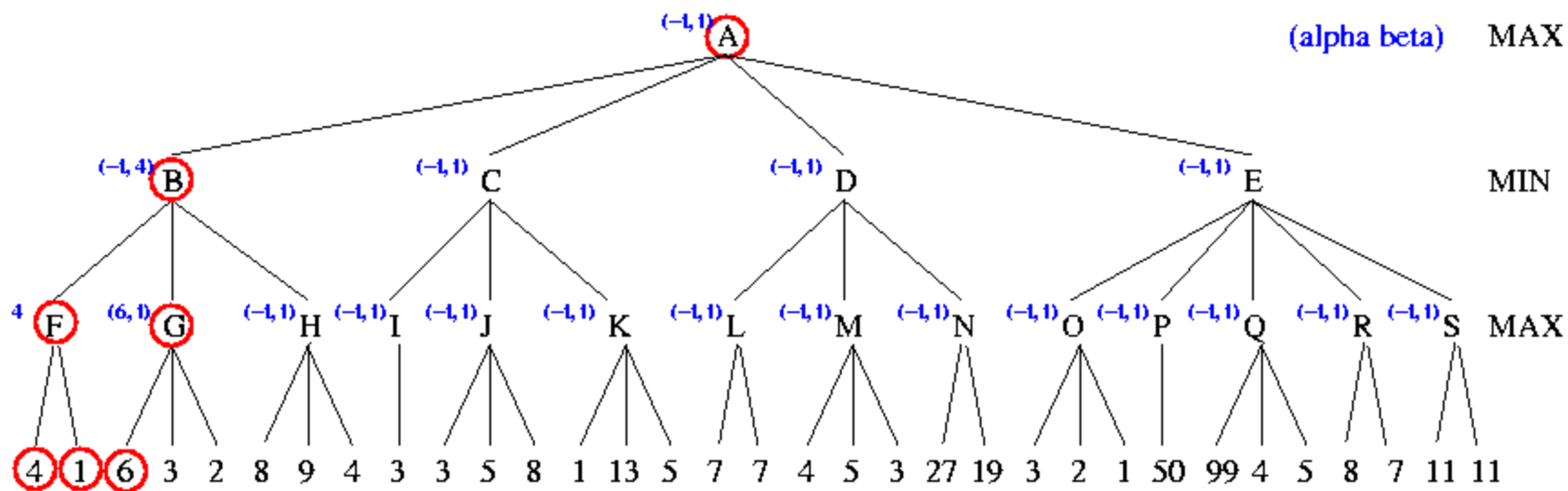
# Example



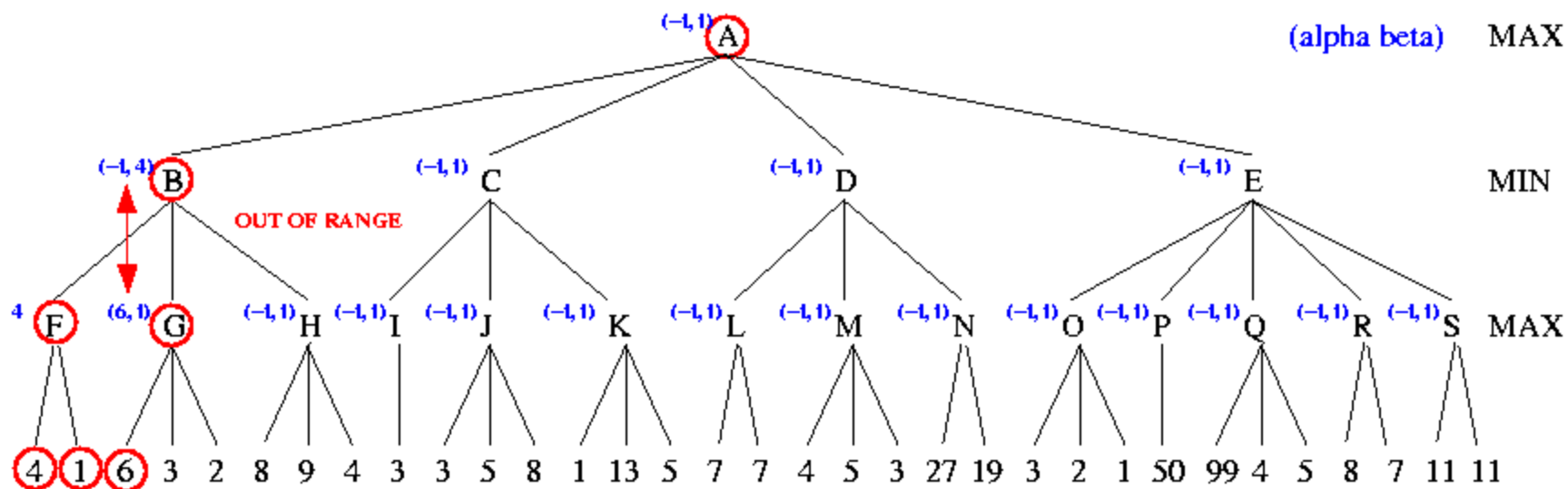
# Example



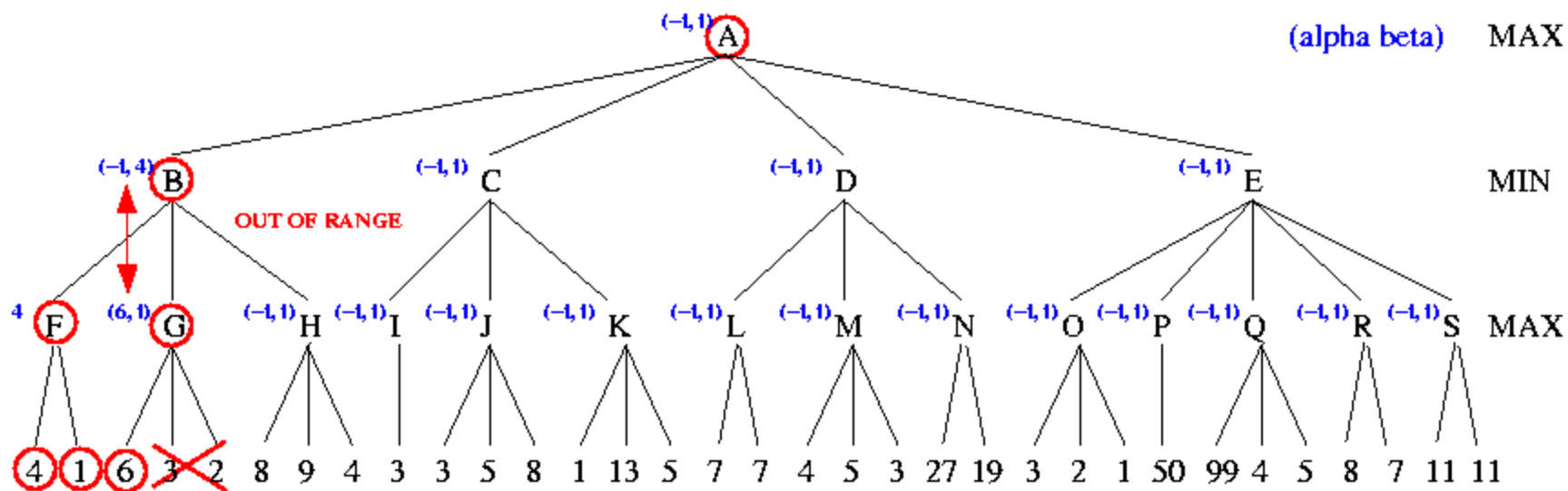
# Example



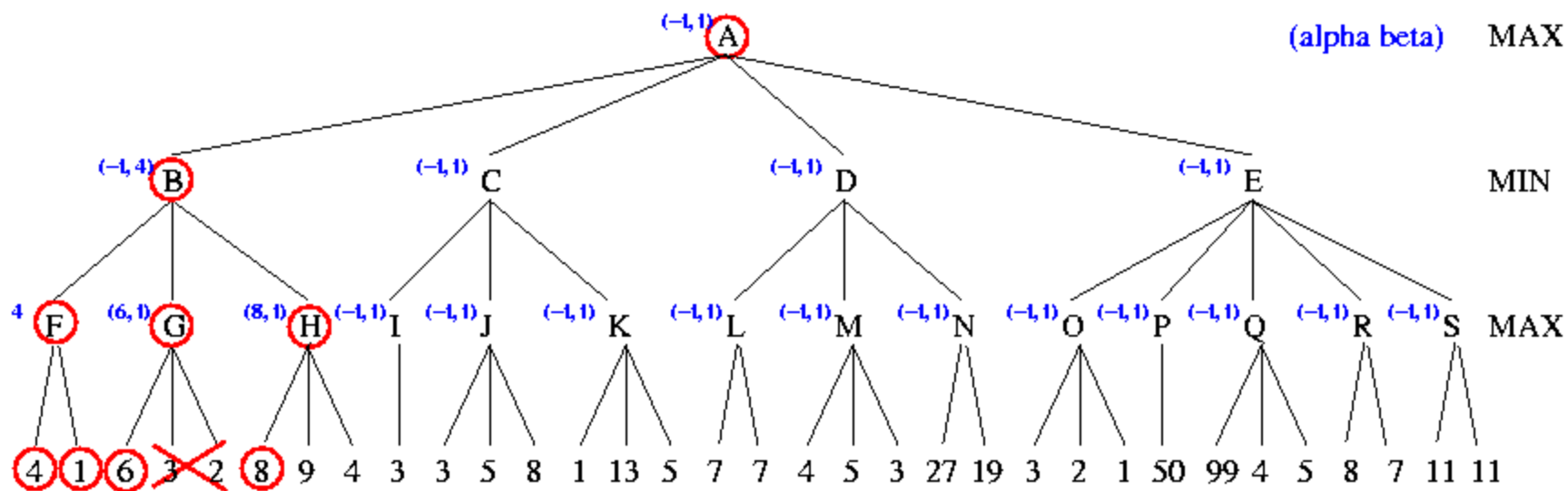
# Example



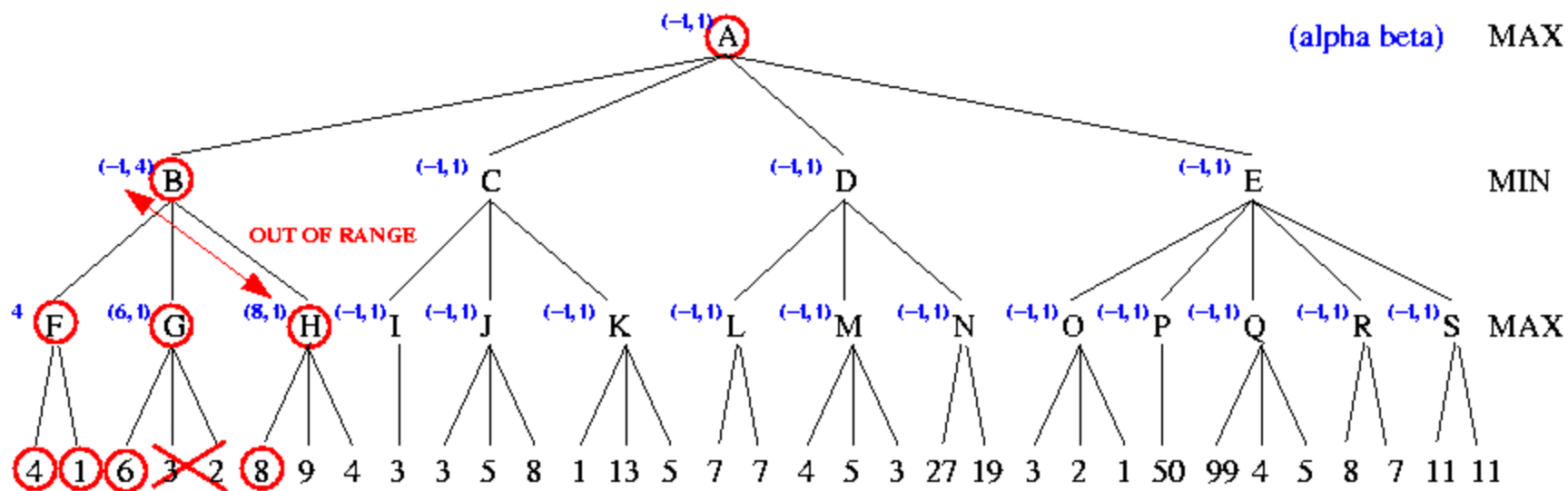
# Example



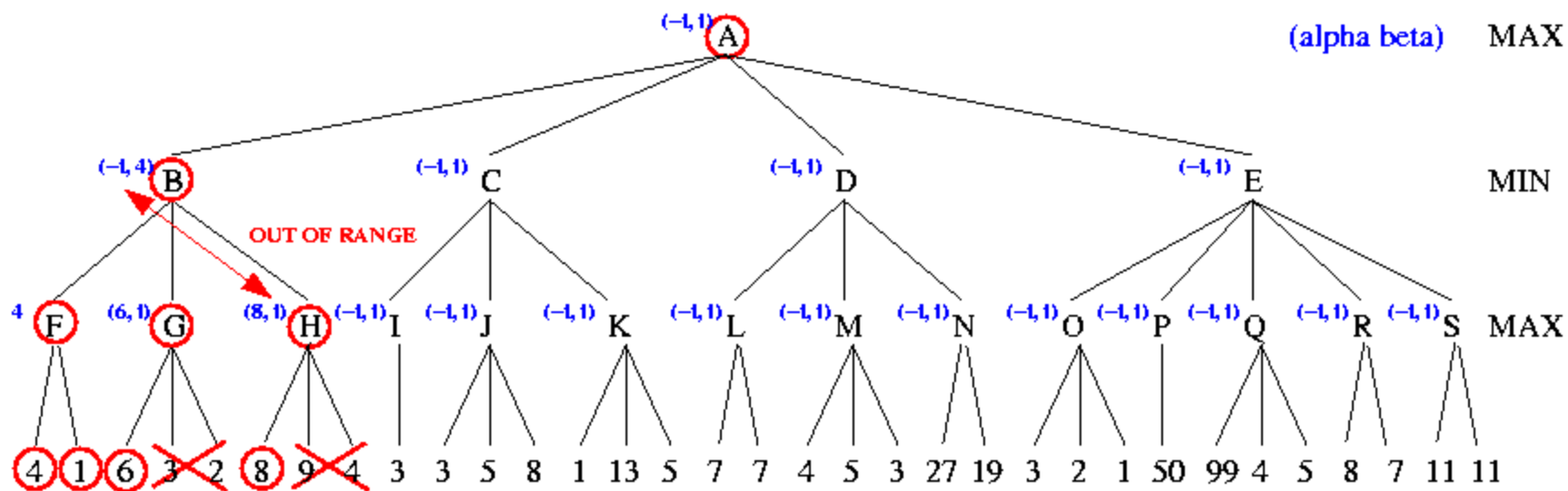
# Example



# Example

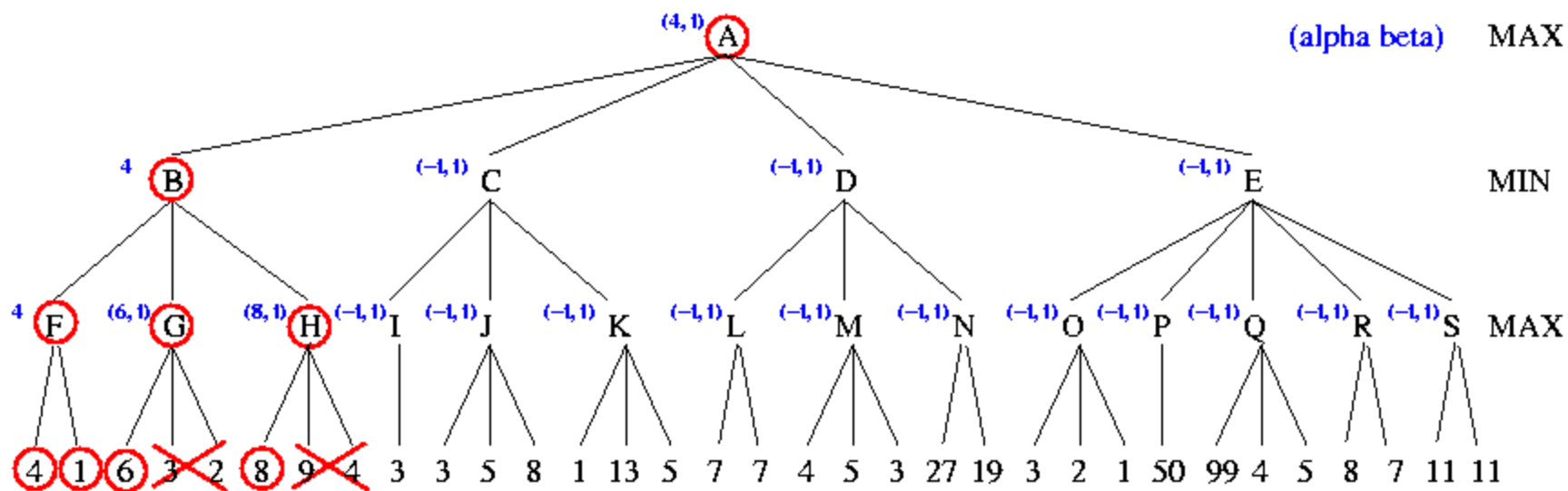


# Example

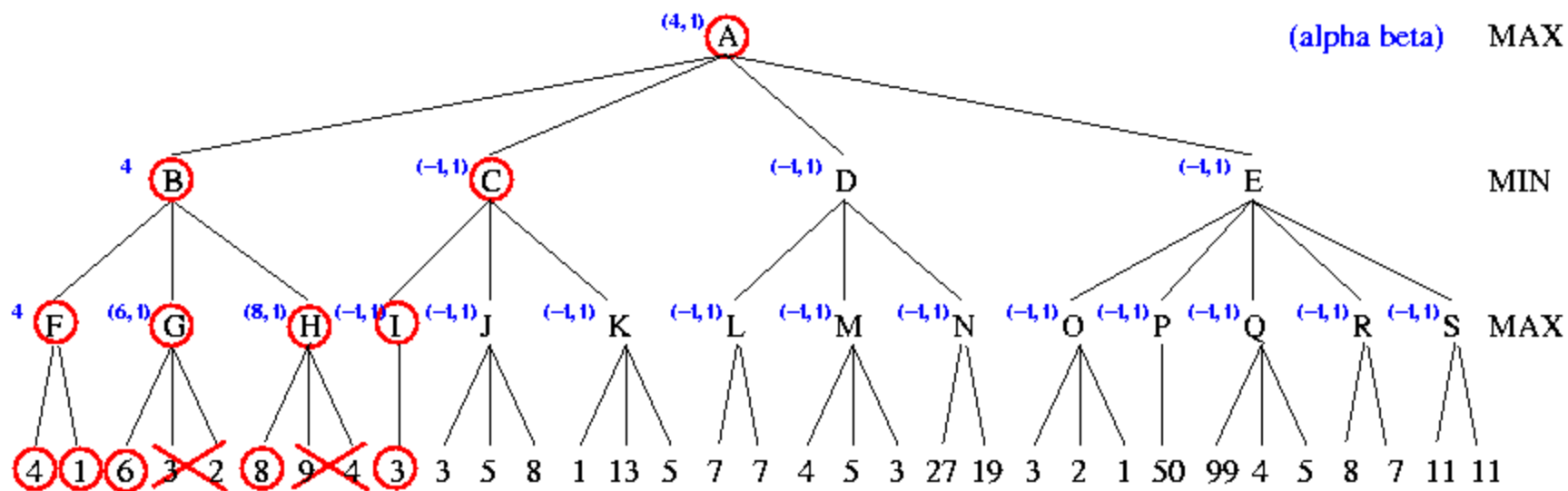




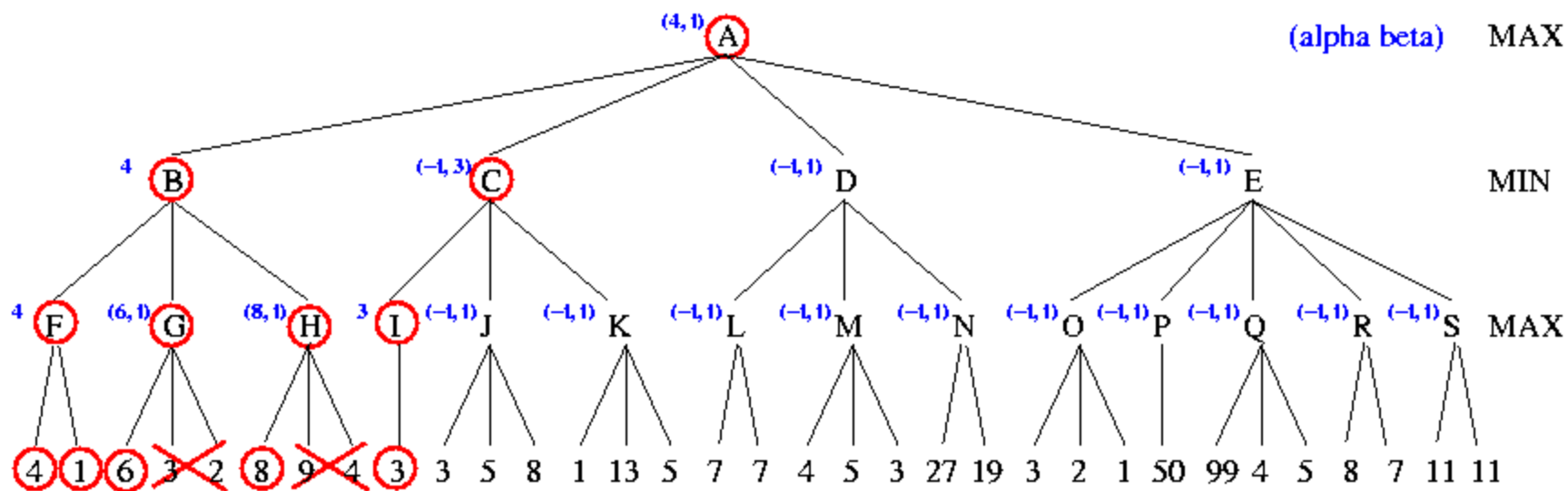
# Example



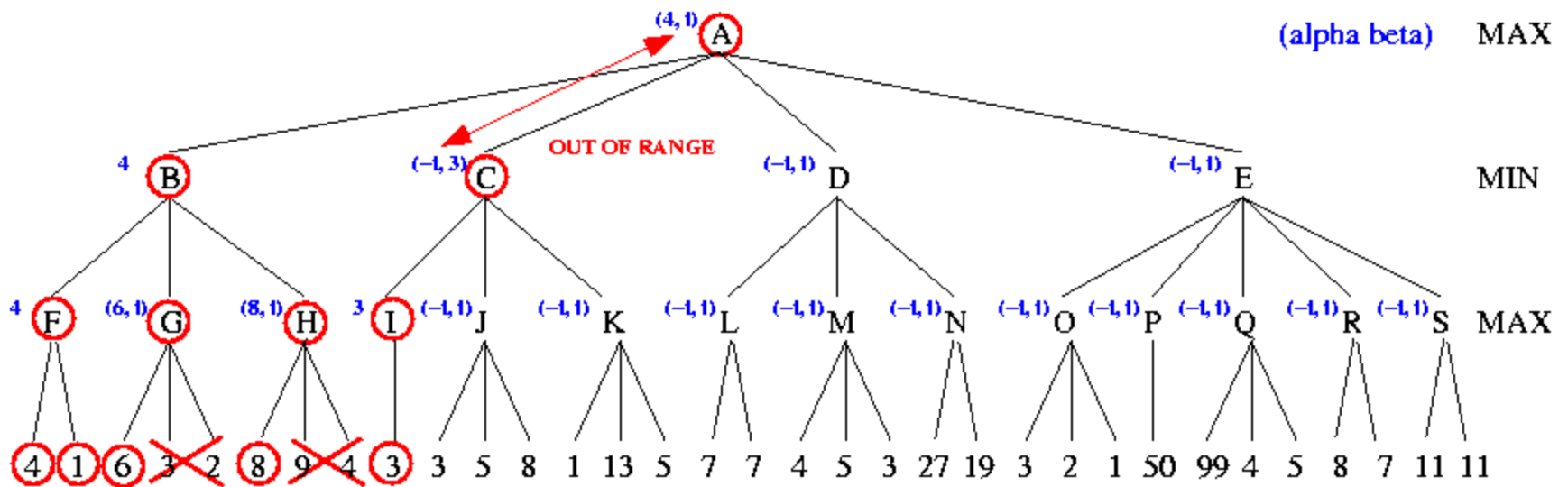
# Example



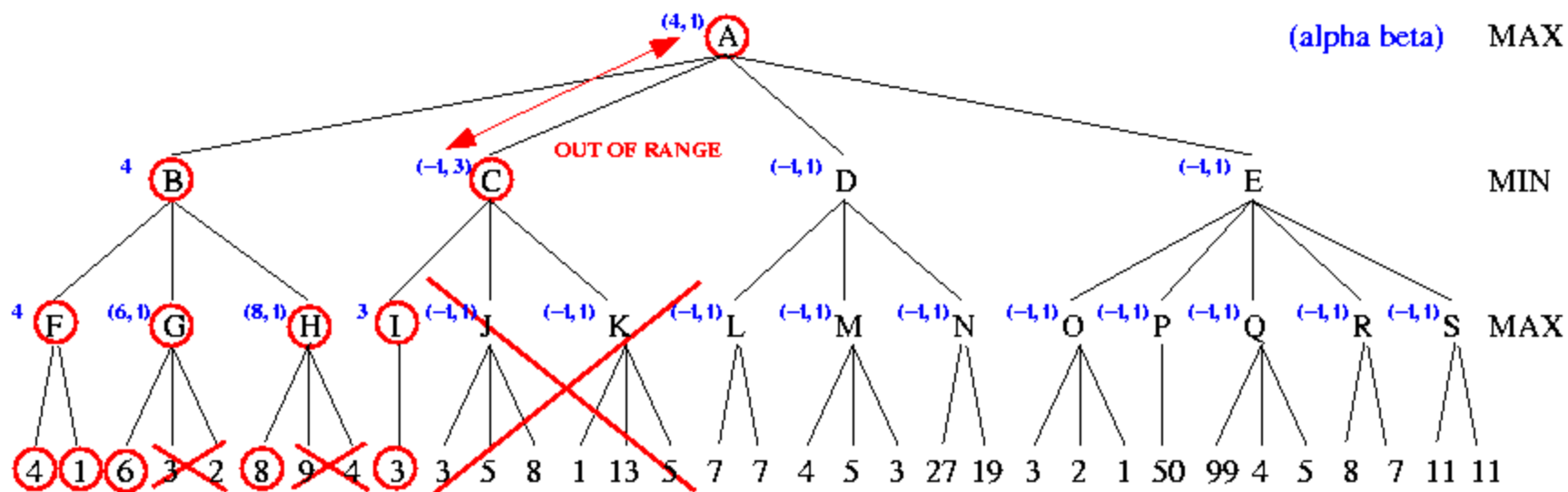
# Example



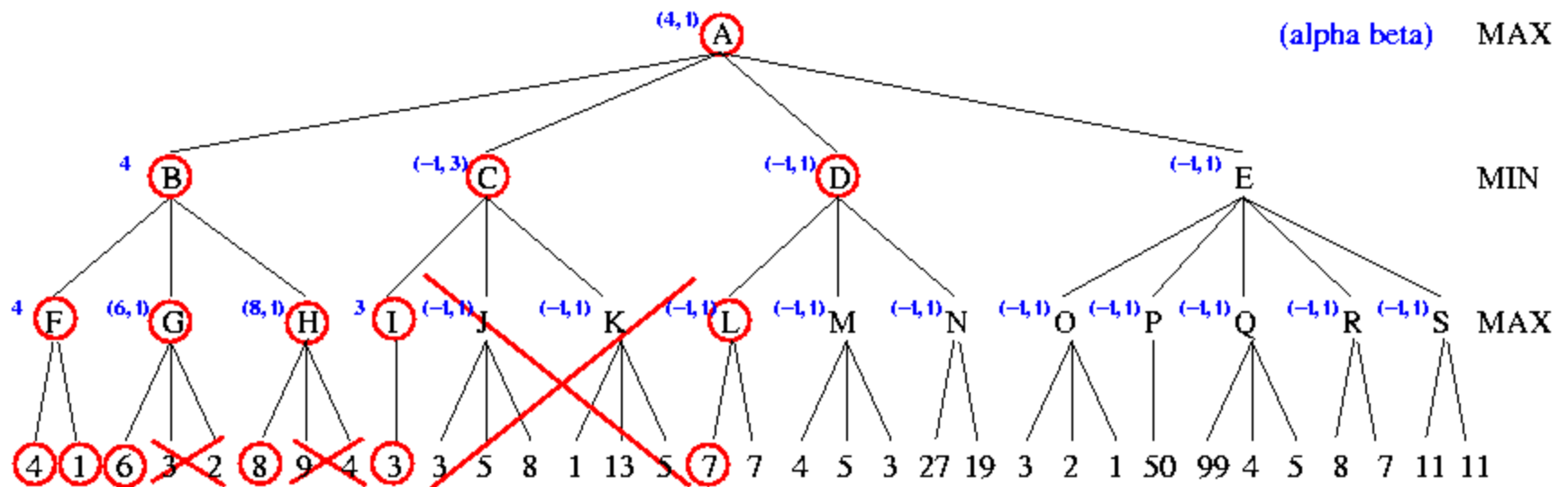
# Example



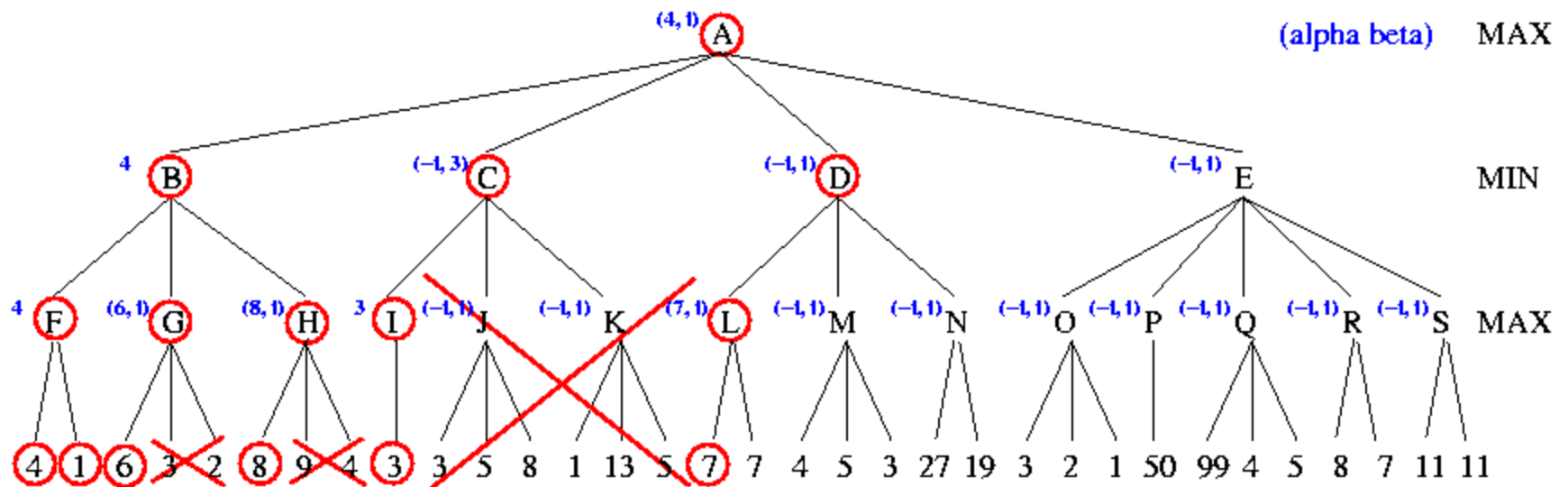
# Example



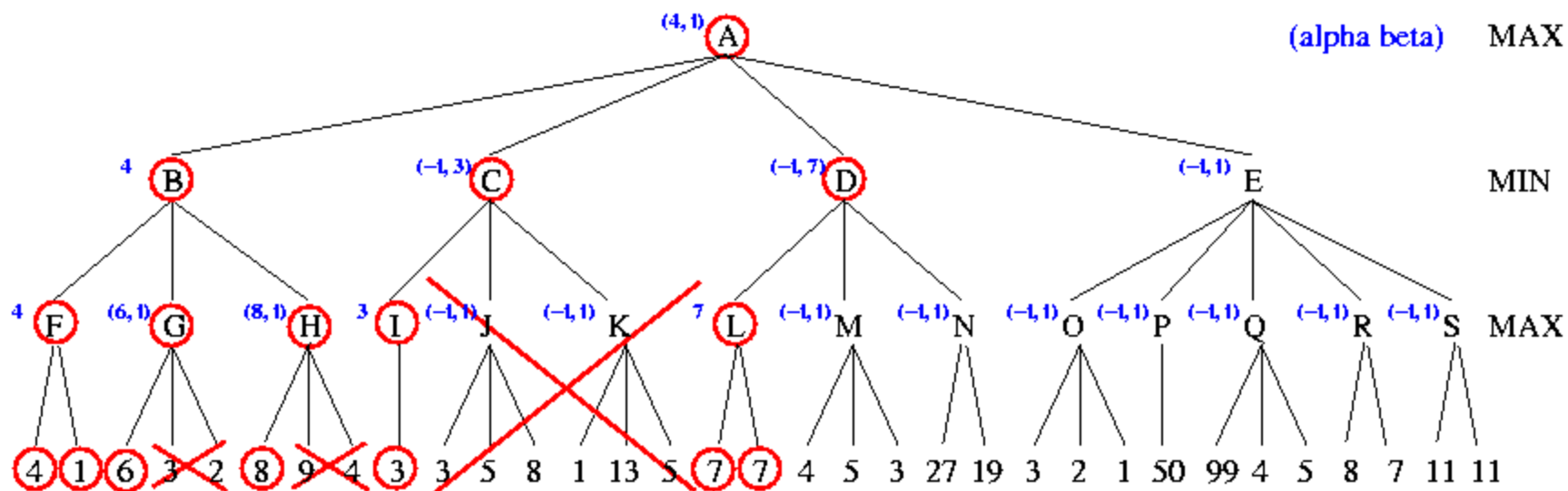
# Example



# Example

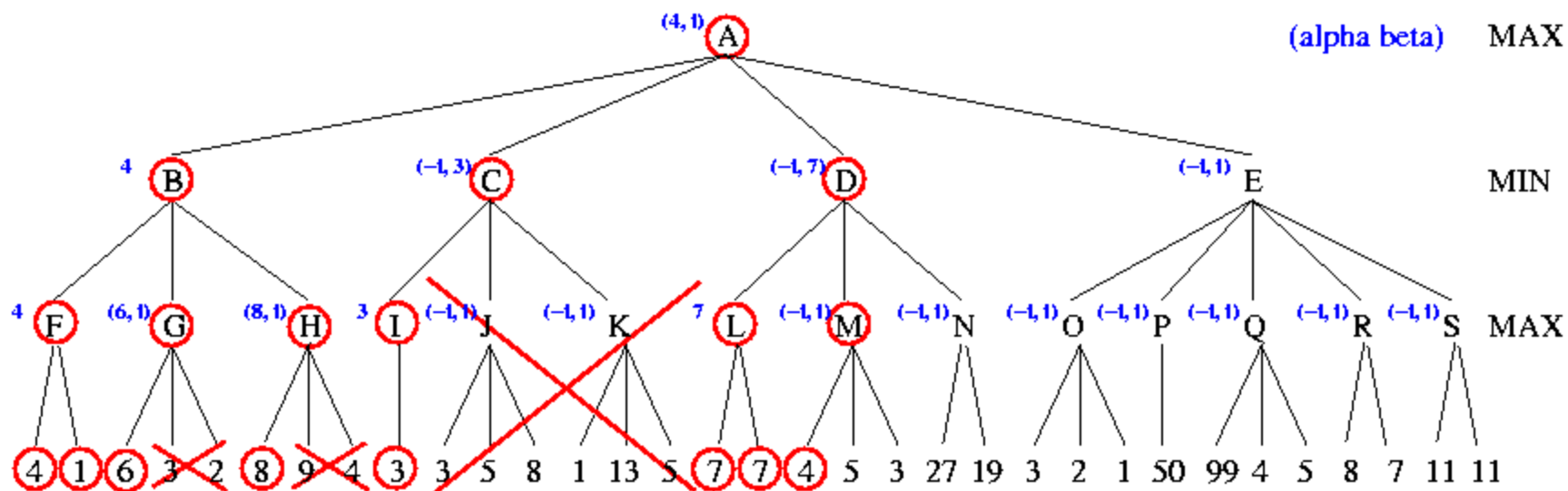


# Example

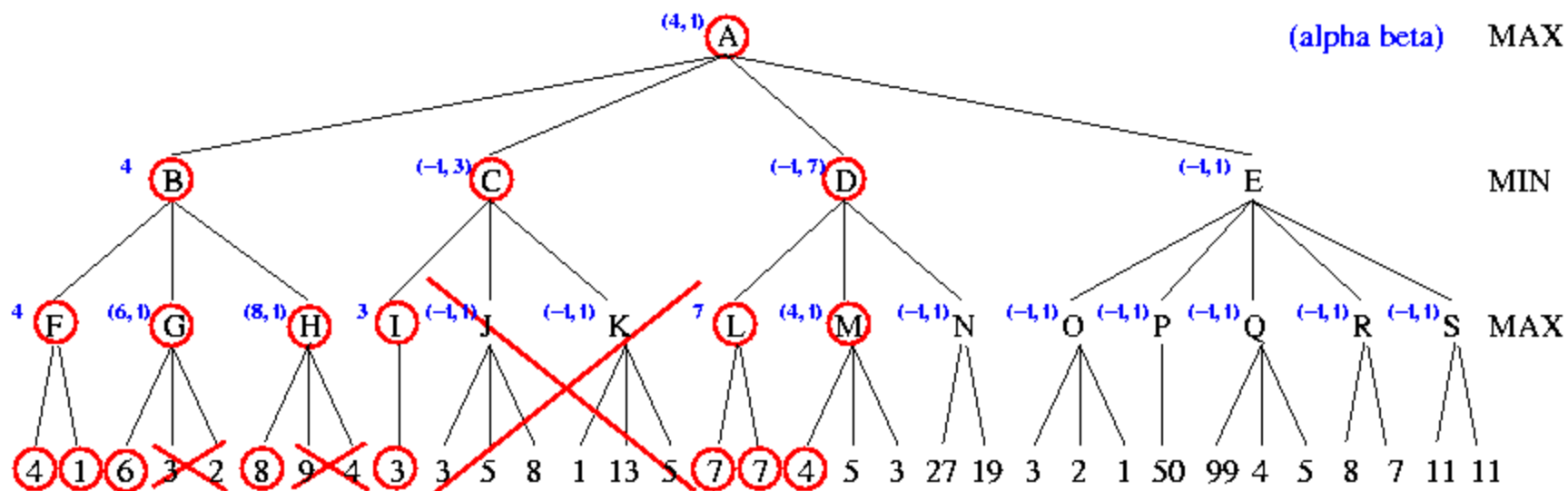




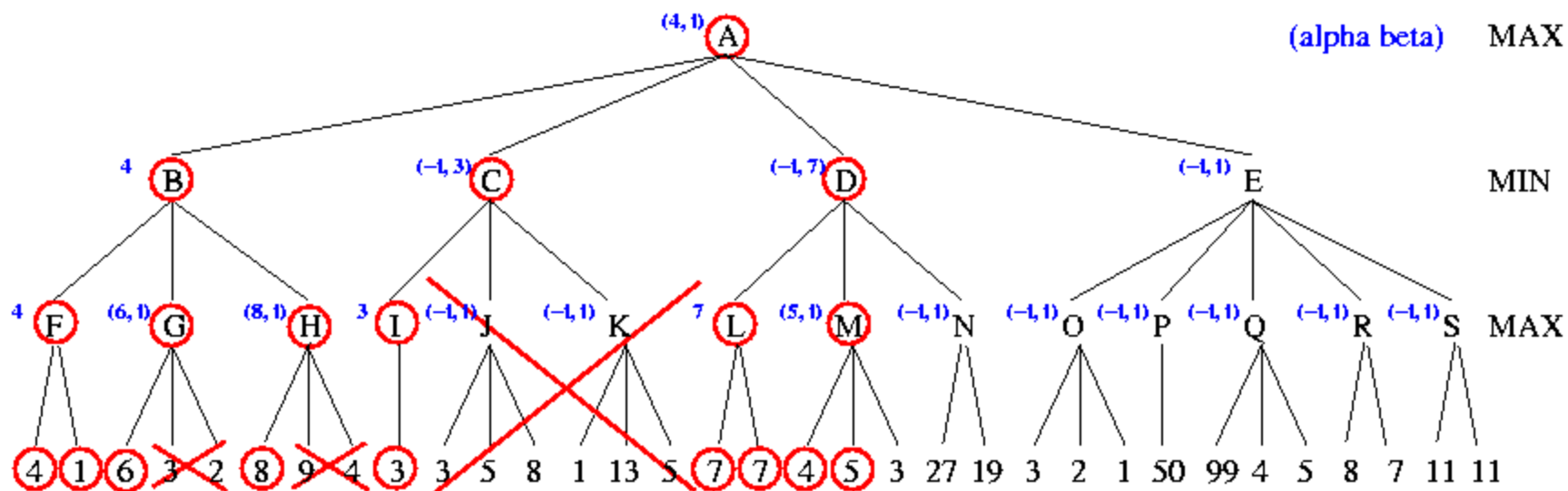
# Example



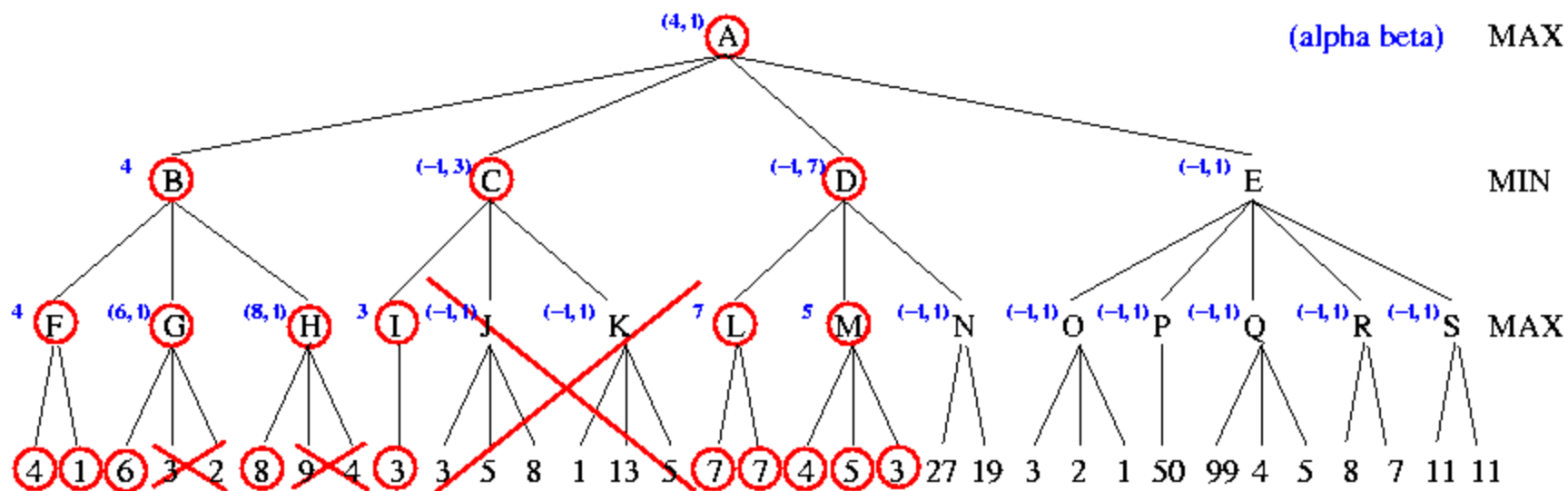
# Example



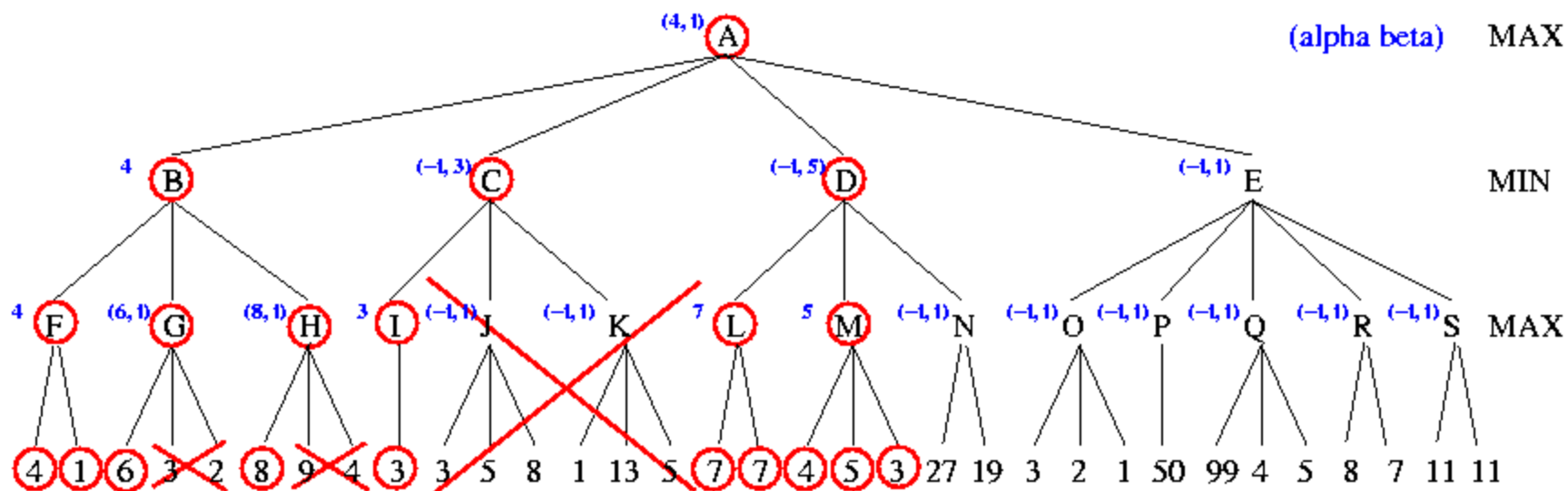
# Example



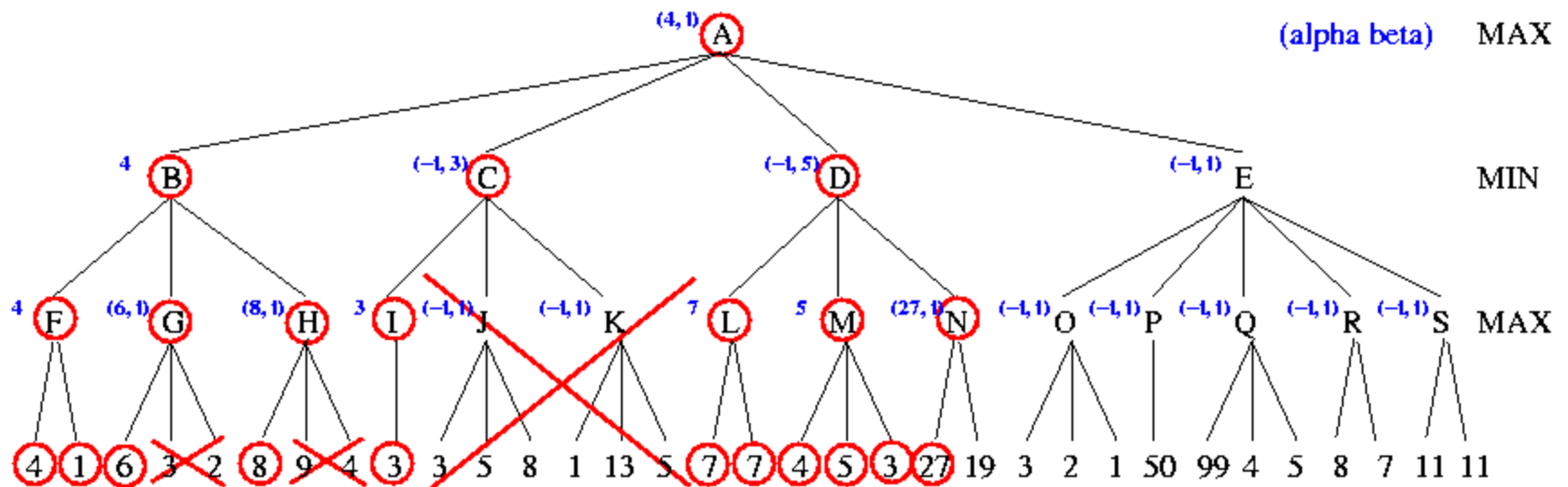
# Example



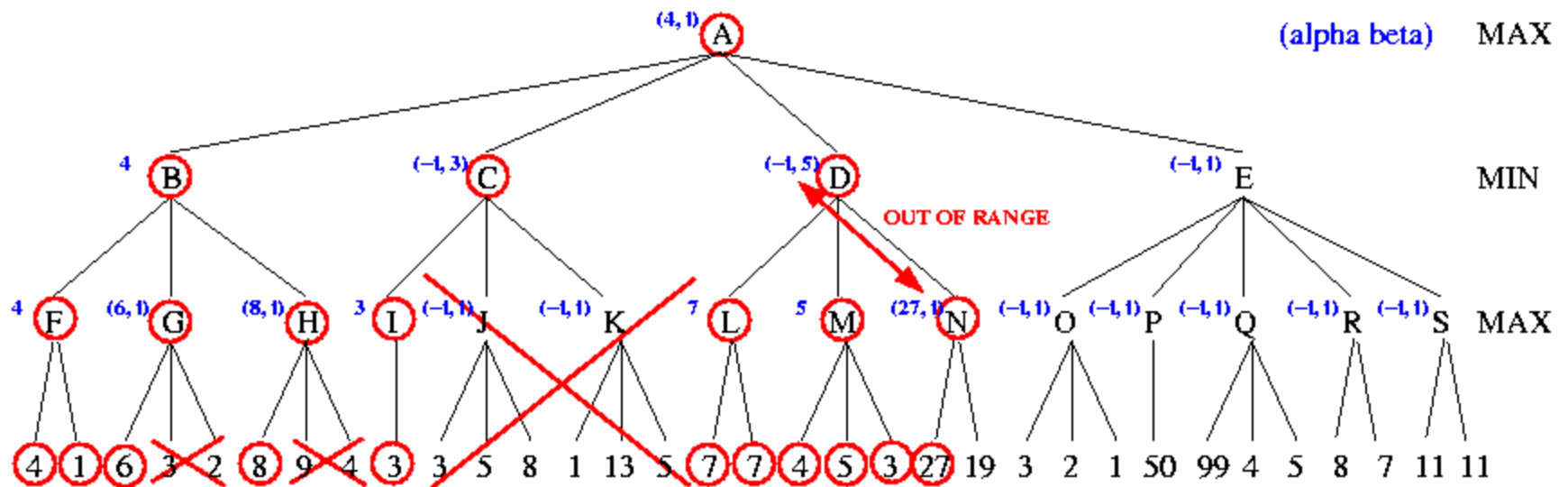
# Example



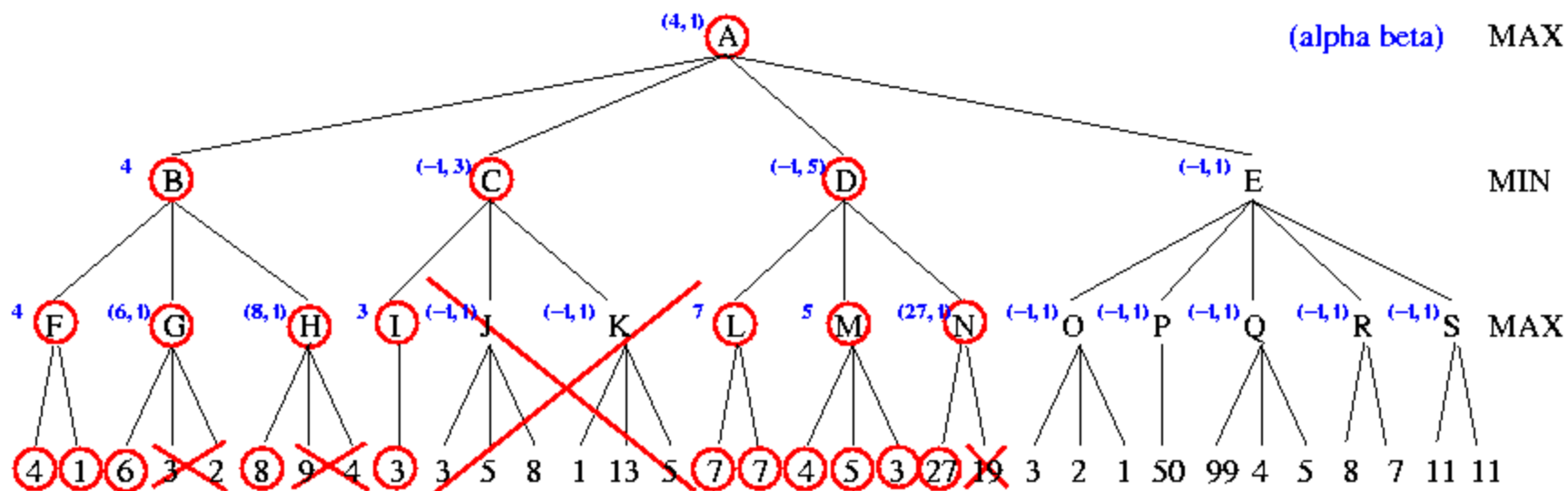
# Example



# Example

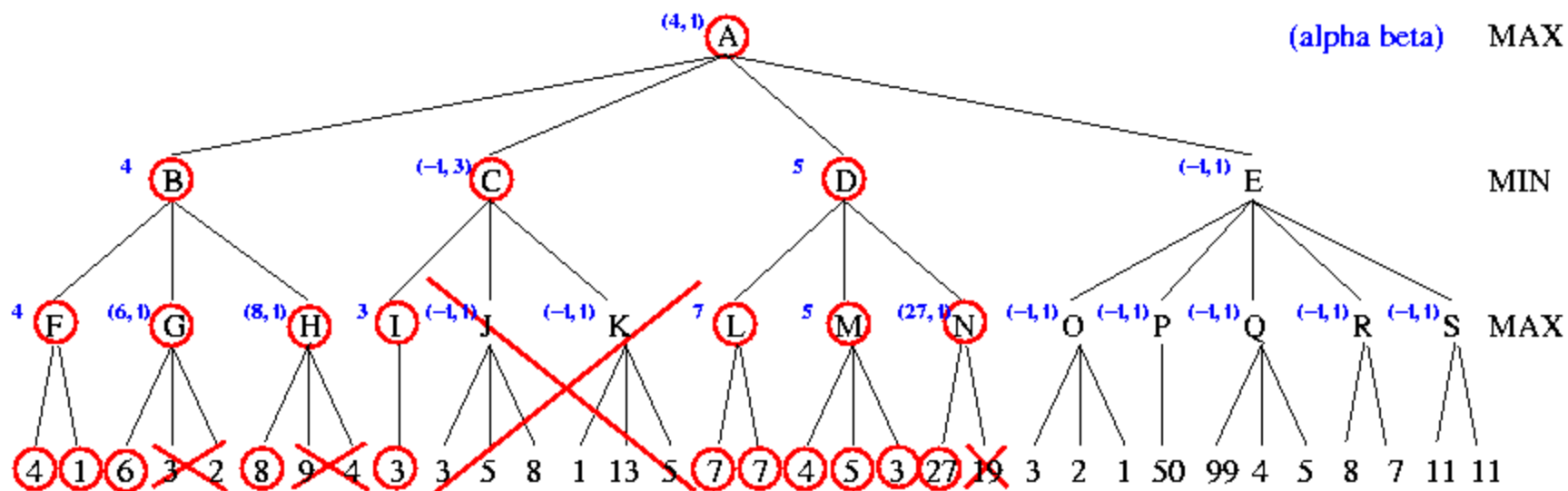


# Example

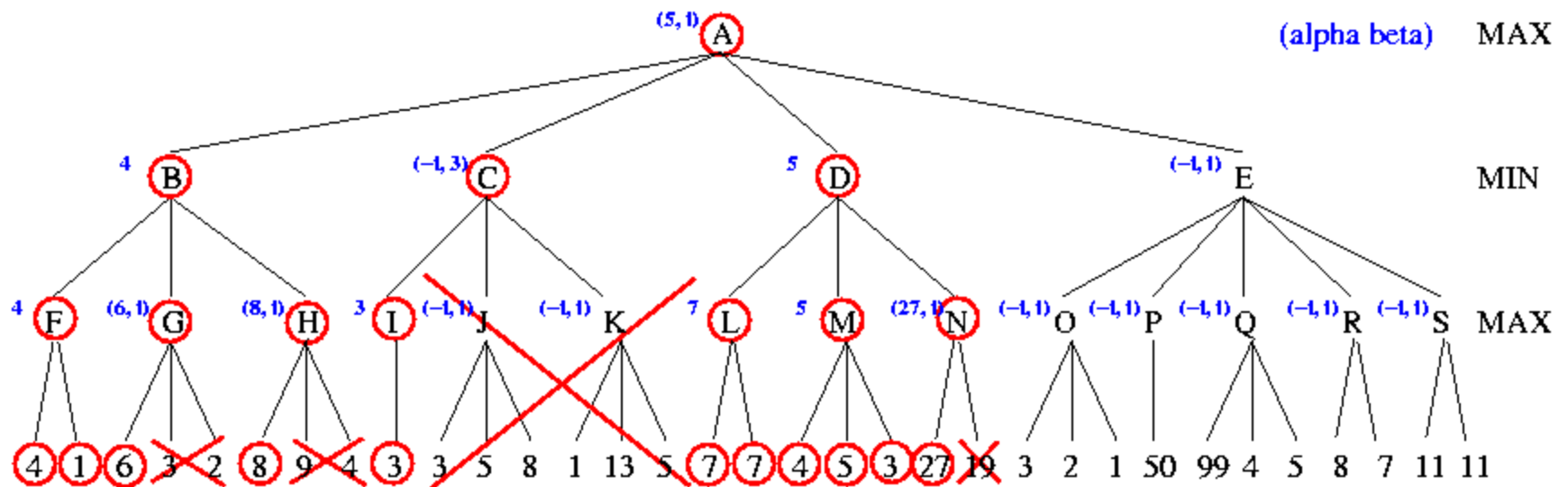




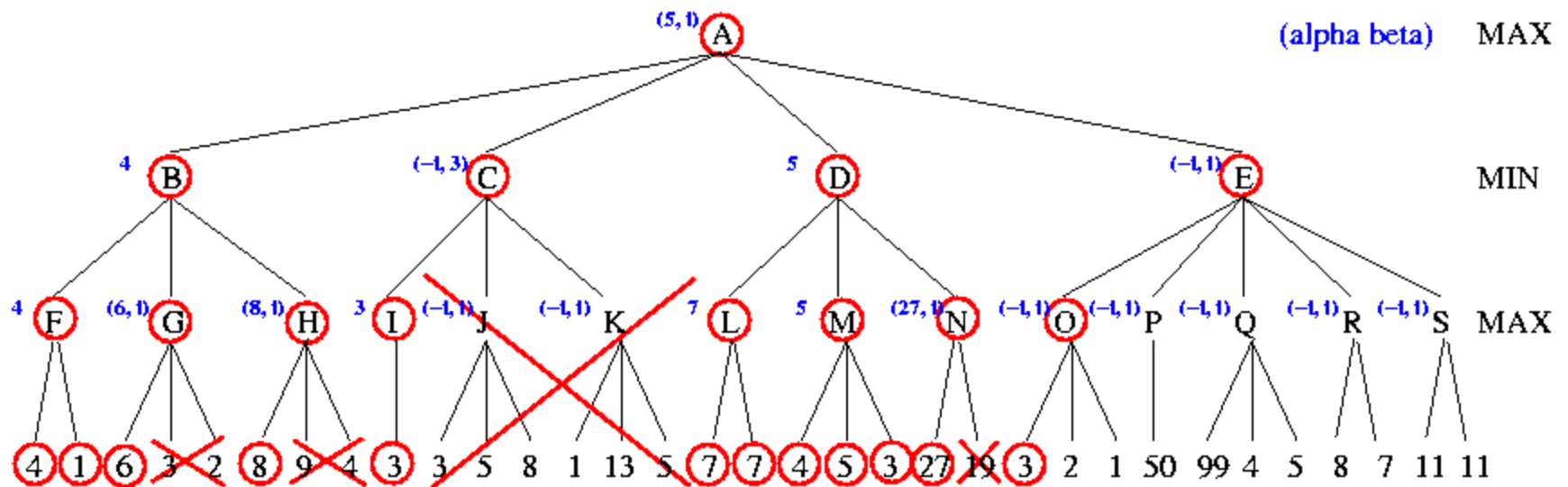
# Example



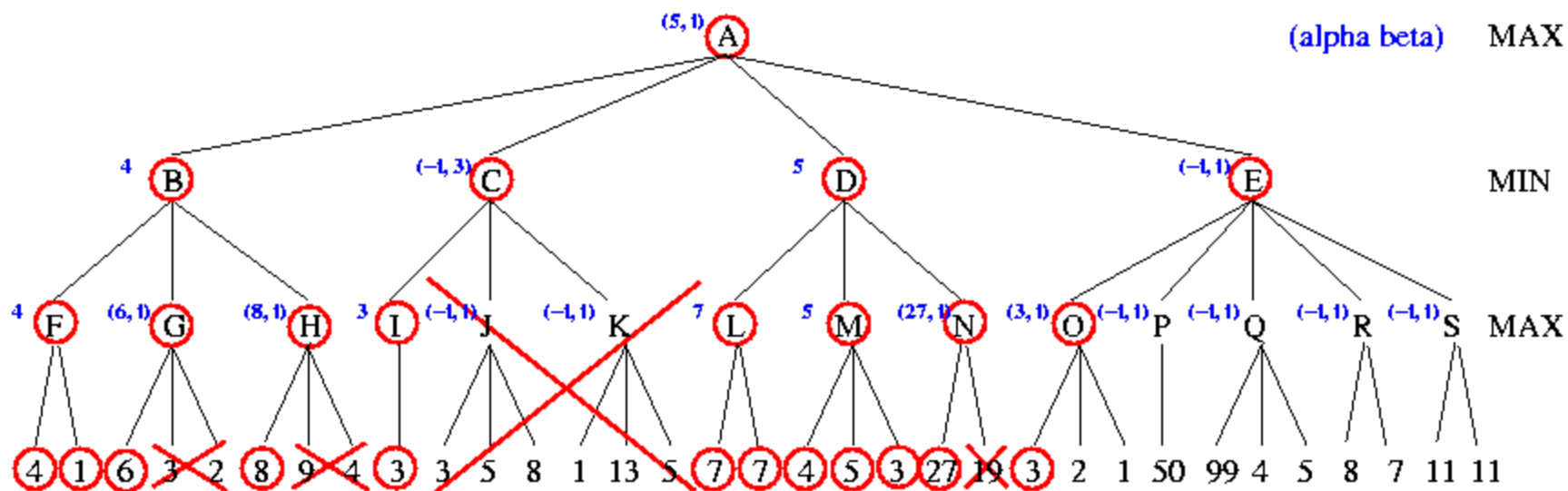
# Example



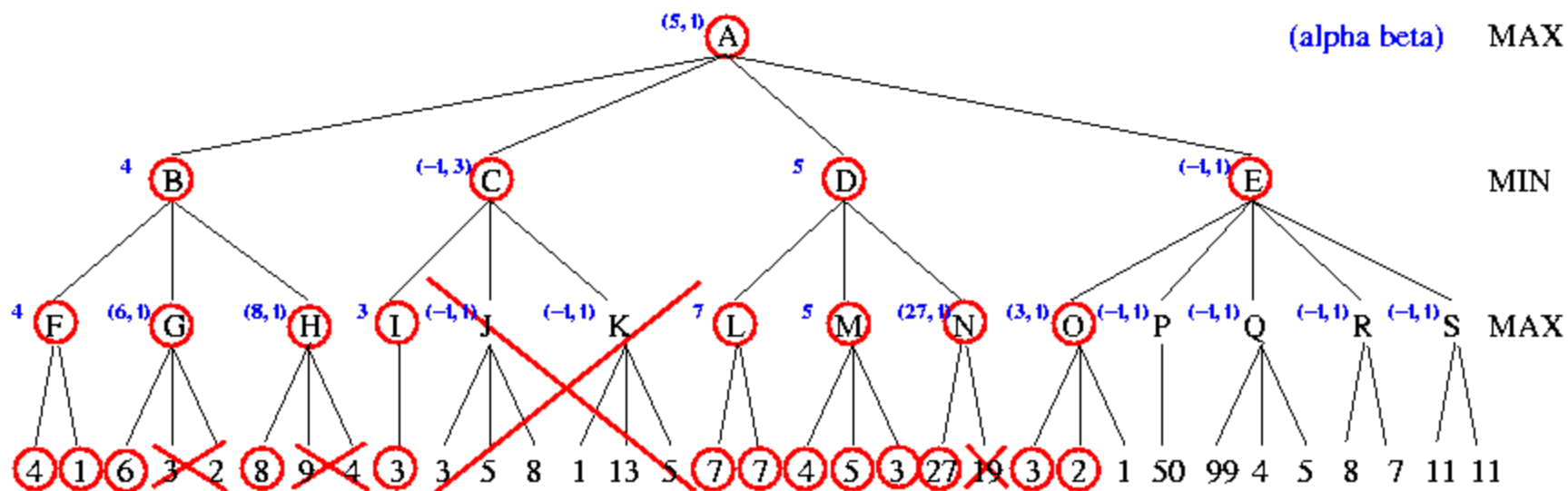
# Example



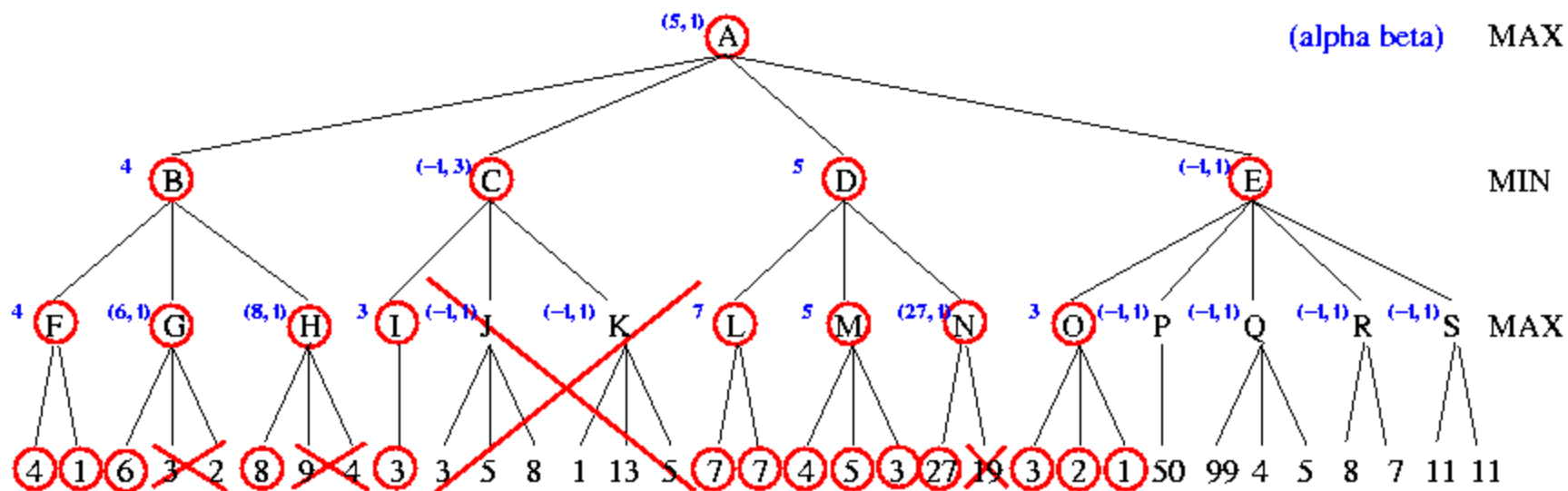
# Example



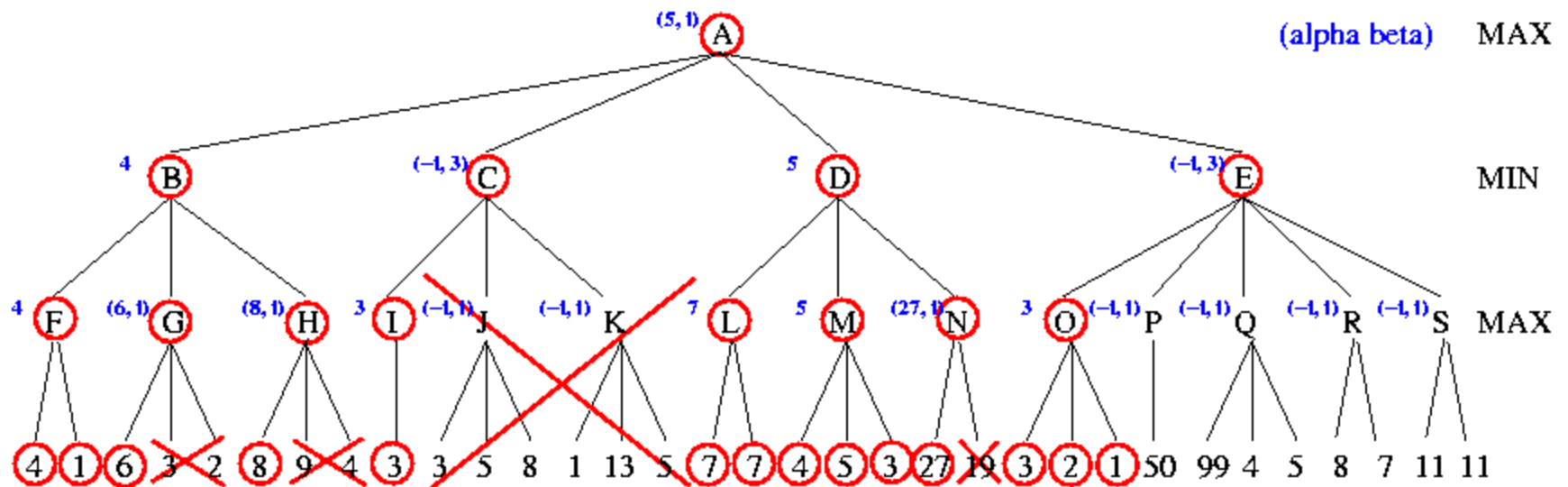
# Example



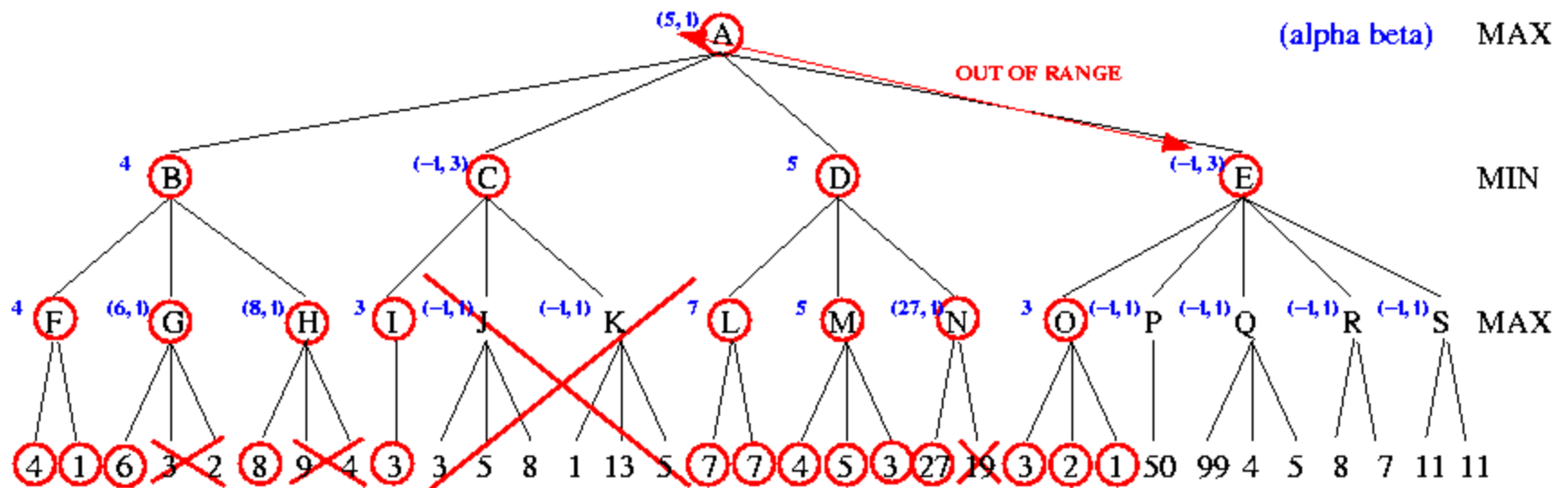
# Example



# Example

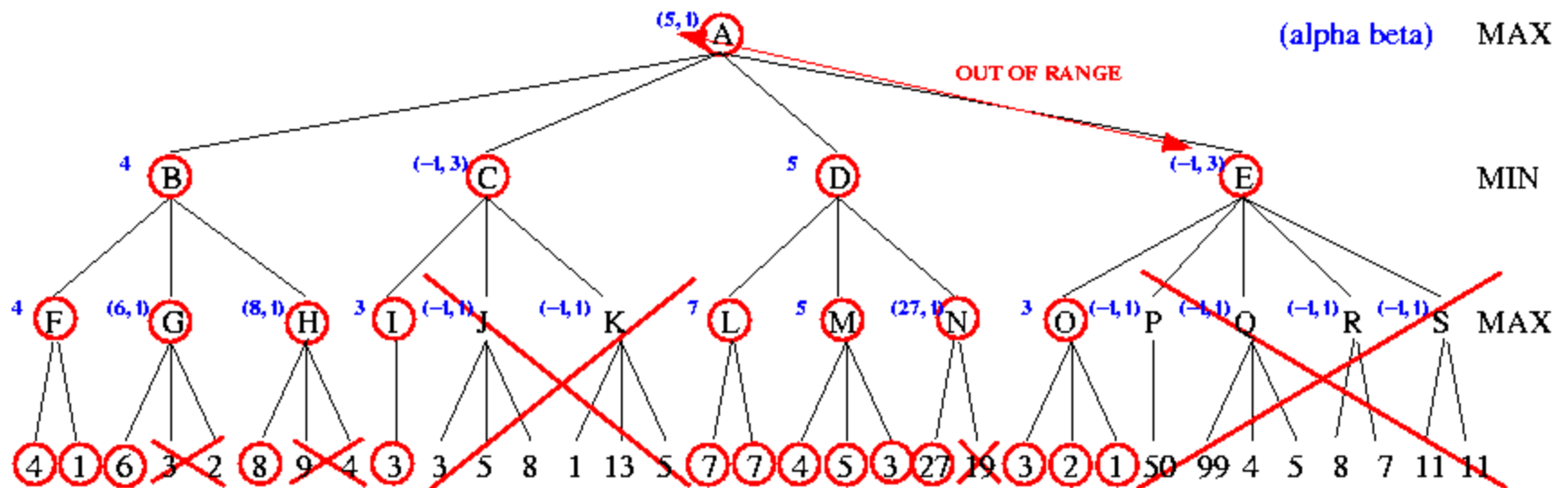


# Example

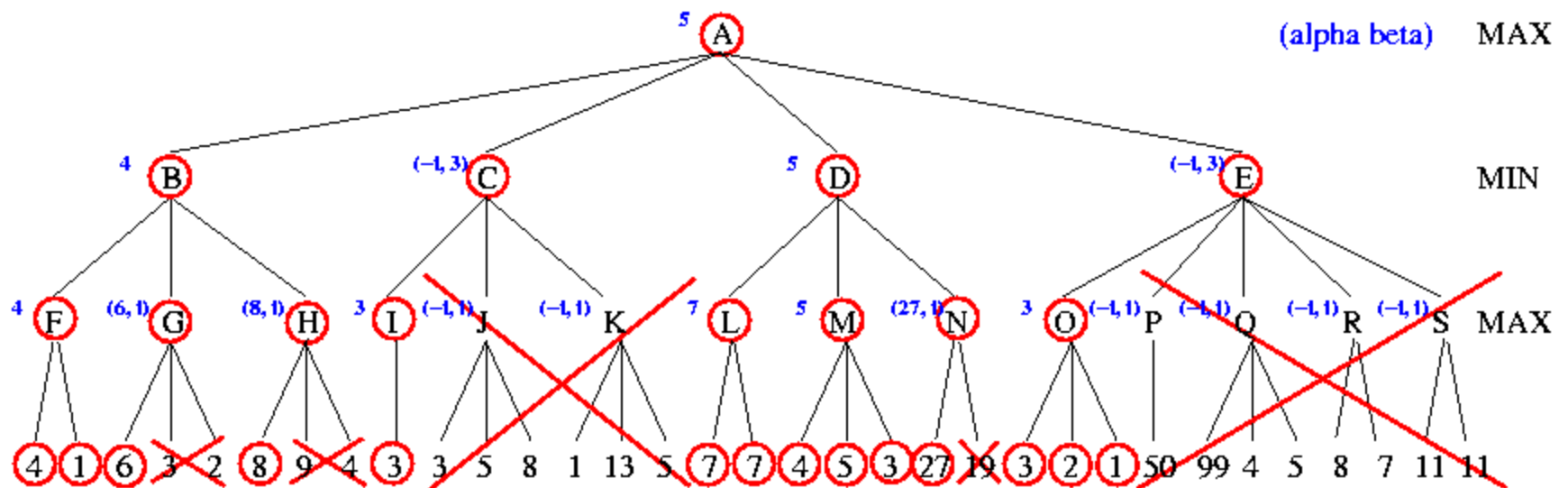




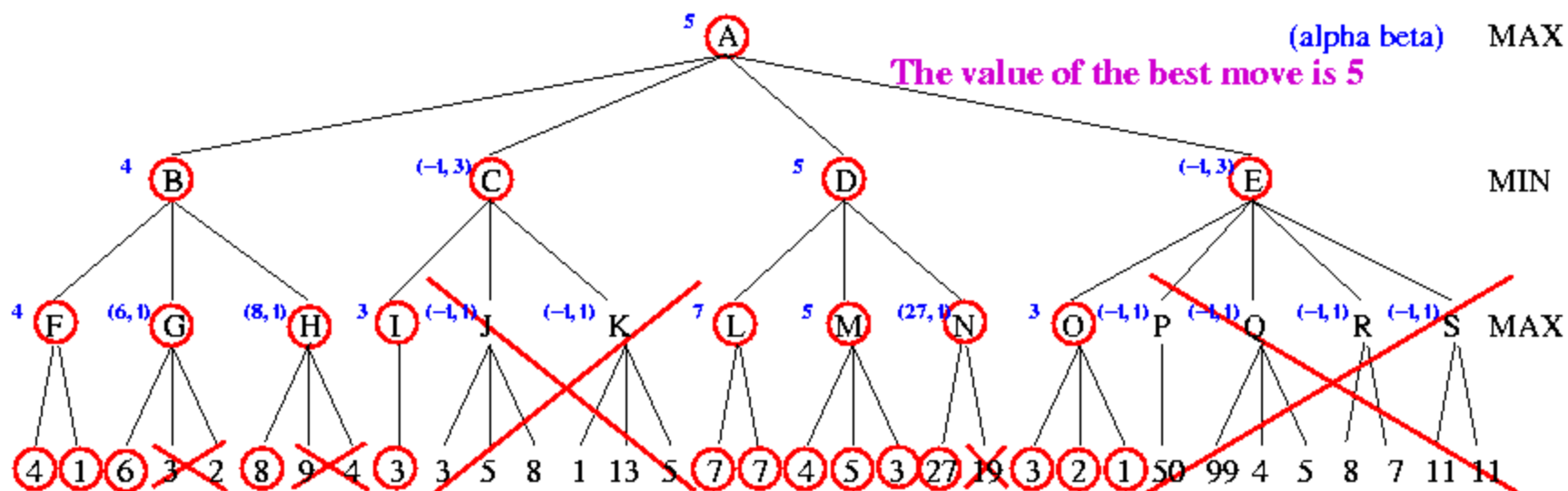
# Example



# Example



# Example



# Practical Implementation

How do we make these ideas practical in real game trees?

Standard approach:

- **cutoff test:** (where do we stop descending the tree)
  - depth limit
  - better: iterative deepening
  - cutoff only when no big changes are expected to occur next (**quiescence search**).
- **evaluation function**
  - When the search is cut off, we evaluate the current state by estimating its utility using **an evaluation function**.

# 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  $-\infty$  (loss) to  $+\infty$  (win) or  $[-1, +1]$ .
- If the board evaluation is  $X$  for a player, it's  $-X$  for the opponent.
- Many clever ideas about how to use the evaluation function.
  - e.g. null move heuristic: let opponent move twice.
- Example:
  - Evaluating chess boards,
  - Checkers
  - Tic-tac-toe

# Summary

- Game playing can be effectively 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.
- Minimax 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.

# Constraint Satisfaction Problems

# Constraint satisfaction problems (CSPs)

- CSP:
  - **state** is defined by **variables**  $X_i$  with **values** from **domain**  $D_i$
  - **goal test** is a set of **constraints** specifying allowable combinations of values for subsets of variables
- Allows useful **general-purpose** algorithms with more power than standard search algorithms

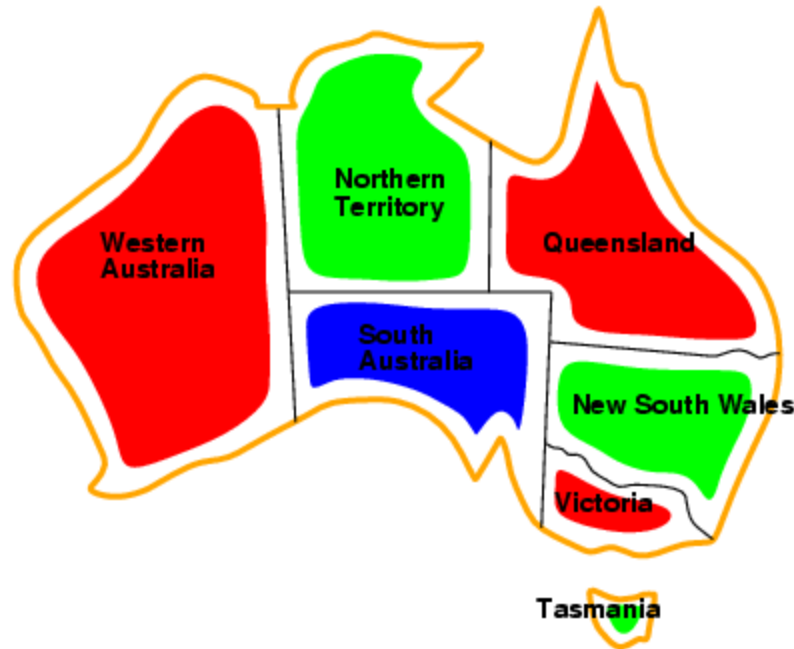


# Example: Map-Coloring



- **Variables**  $WA, NT, Q, NSW, V, SA, T$
- **Domains**  $D_i = \{\text{red, green, blue}\}$
- **Constraints**: adjacent regions must have different colors
- e.g.,  $WA \neq NT$

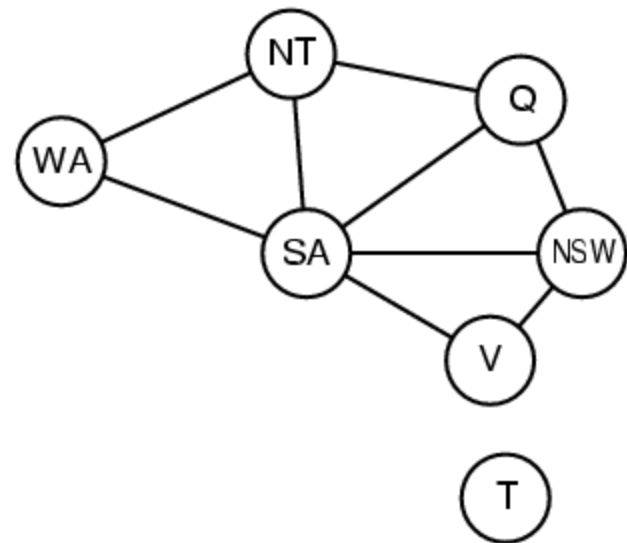
# Example: Map-Coloring



- Solutions are **complete** and **consistent** assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

# Constraint graph

- **Binary CSP:** each constraint relates two variables
- **Constraint graph:** nodes are variables, arcs are constraints



# Varieties of CSPs

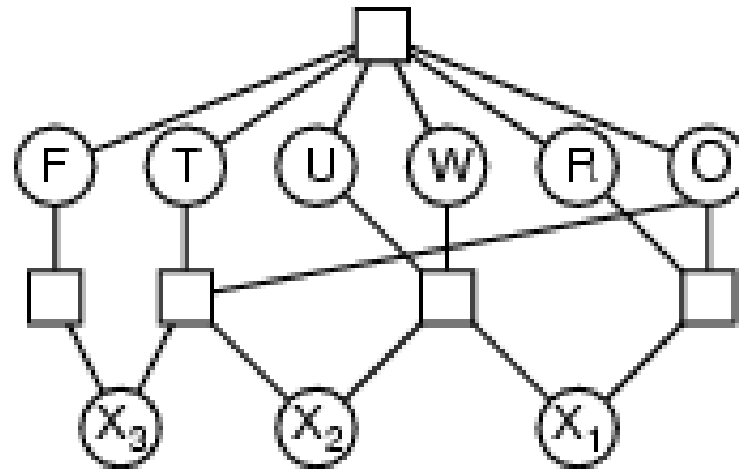
- Discrete variables
  - finite domains:
    - $n$  variables, domain size  $d \rightarrow O(d^n)$  complete assignments
    - e.g., 3-SAT (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g.,  $StartJob_1 + 5 \leq StartJob_3$
- Continuous variables
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming

# Varieties of constraints

- **Unary** constraints involve a single variable,
  - e.g.,  $SA \neq \text{green}$
- **Binary** constraints involve pairs of variables,
  - e.g.,  $SA \neq WA$
- **Higher-order** constraints involve 3 or more variables,
  - e.g.,  $SA \neq WA \neq NT$

# Example: Cryptarithmic

$$\begin{array}{r}
 \text{TWO} \\
 + \text{TWO} \\
 \hline
 \text{FOUR}
 \end{array}$$



- **Variables:**  $F T U W R O$   $X_1 X_2 X_3$
- **Domains:**  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$   $\{0, 1\}$
- **Constraints:**  $Alldiff(F, T, U, W, R, O)$ 
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F, T \neq 0, F \neq 0$

- SEND+MORE=MONEY
- BASE+BALL=GAMES
- LOGIC+LOGIC=PROLOG

# Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

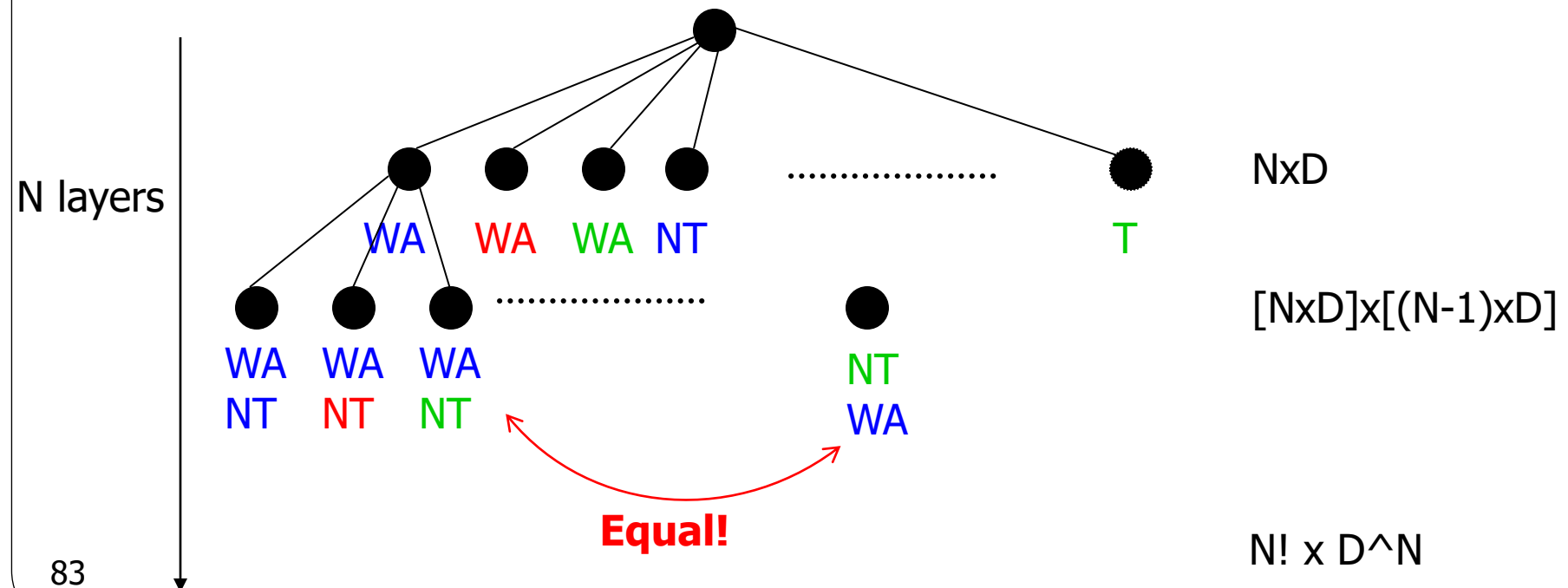
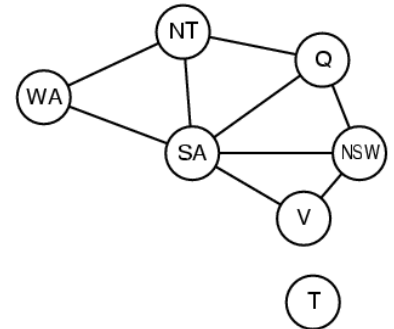


# Standard search formulation

Let's try the standard search formulation.

We need:

- Initial state: none of the variables has a value (color)
- Successor state: one of the variables without a value will get some value.
- Goal: all variables have a value and none of the constraints is violated.

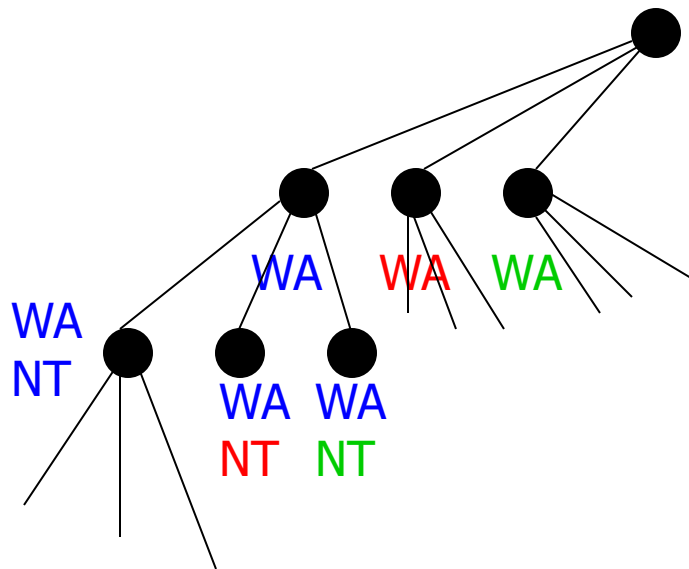


There are  $N! \times D^N$  nodes in the tree but only  $D^N$  distinct states??

# Backtracking (Depth-First) search

- Special property of CSPs: They **are commutative**:  
This means: the order in which we assign variables does not matter.
- Better search tree: First **order** variables, then assign them values **one-by-one**.

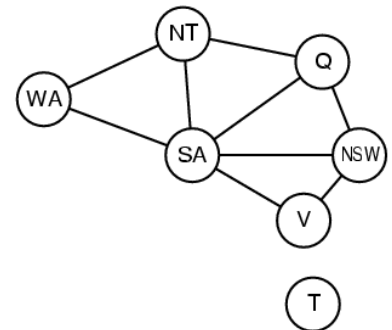
$$\begin{matrix} \text{NT} \\ \text{WA} \end{matrix} = \begin{matrix} \text{WA} \\ \text{NT} \end{matrix}$$



D

$D^2$

$D^N$



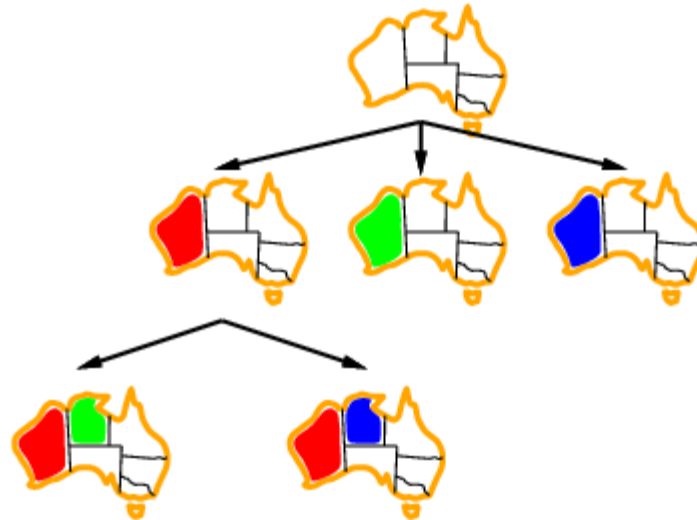
# Backtracking example



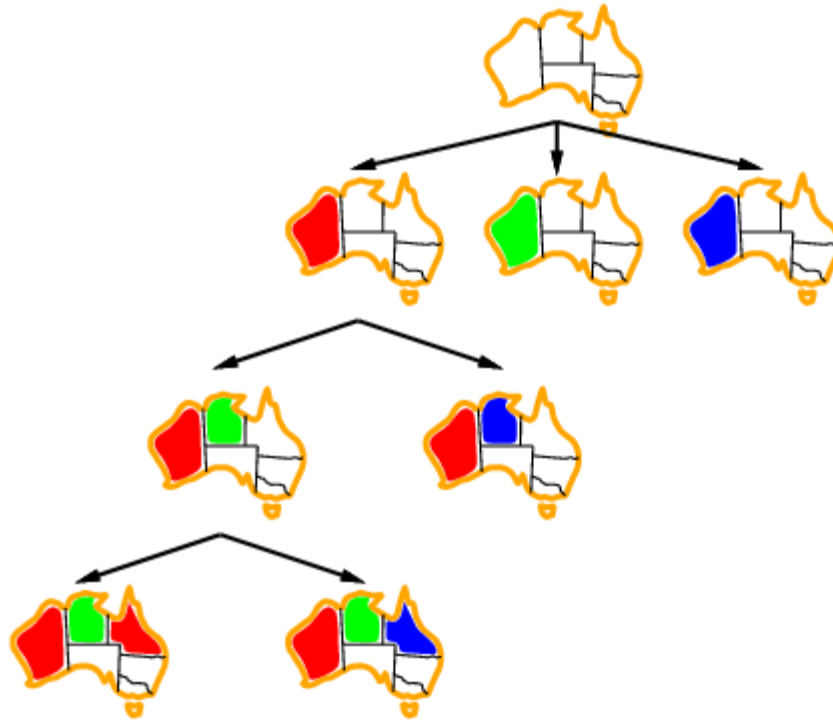
# Backtracking example



# Backtracking example



# Backtracking example

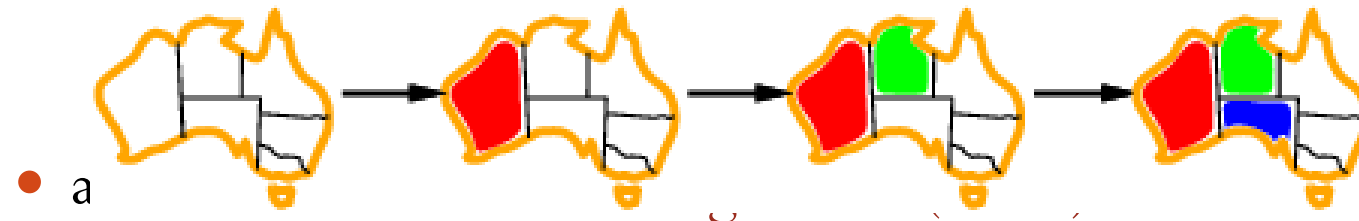


# Improving backtracking efficiency

- **General-purpose** methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

# Most constrained variable

- Most constrained variable:  
choose the variable with the fewest legal values

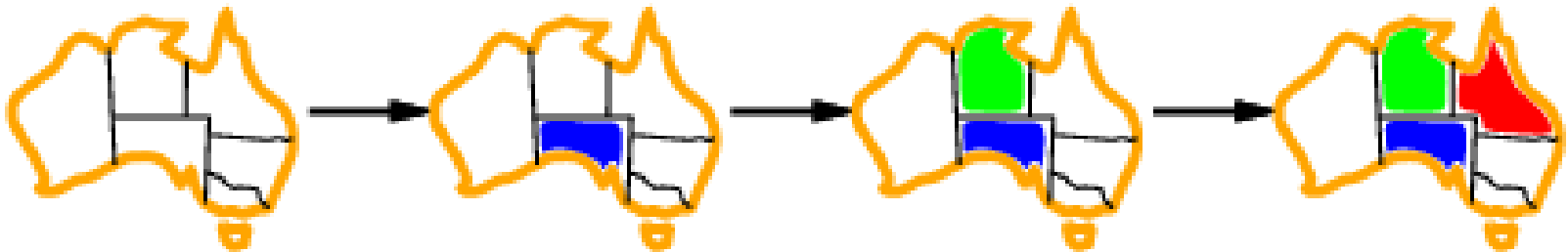
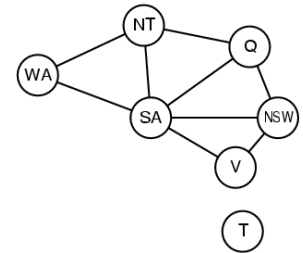


- Picks a variable which will cause failure as soon as possible, allowing the tree to be pruned.



# Most constraining variable

- Tie-breaker among most constrained variables
- Most constraining variable:
  - choose the variable with the most constraints or variables (most edges in graph)



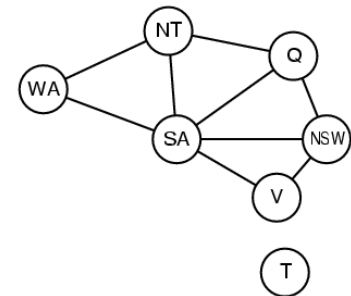
# Least constraining value

- Given a variable, choose the least constraining value
  - the one that rules out the fewest values in the remaining variables



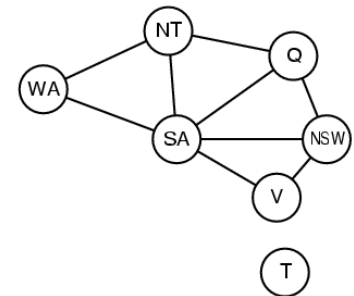
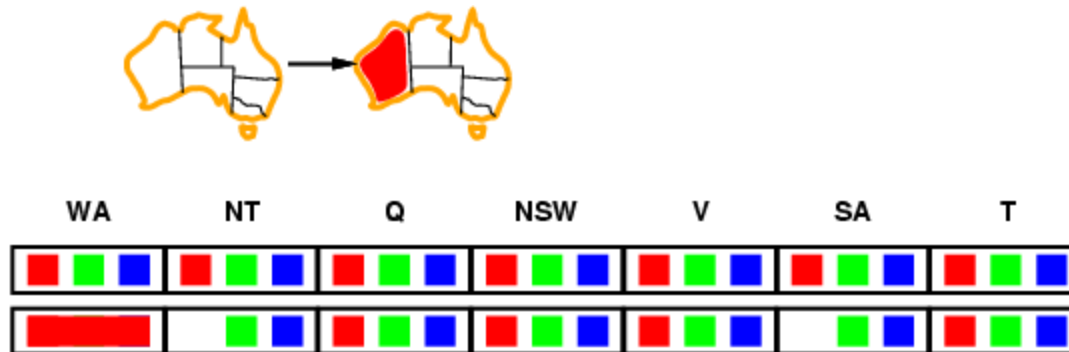
# Forward checking

- **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



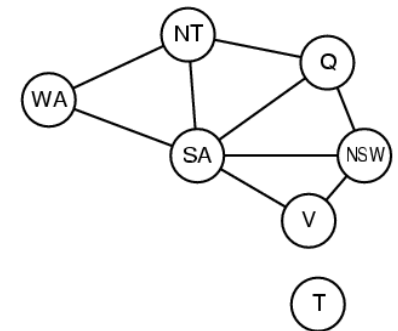
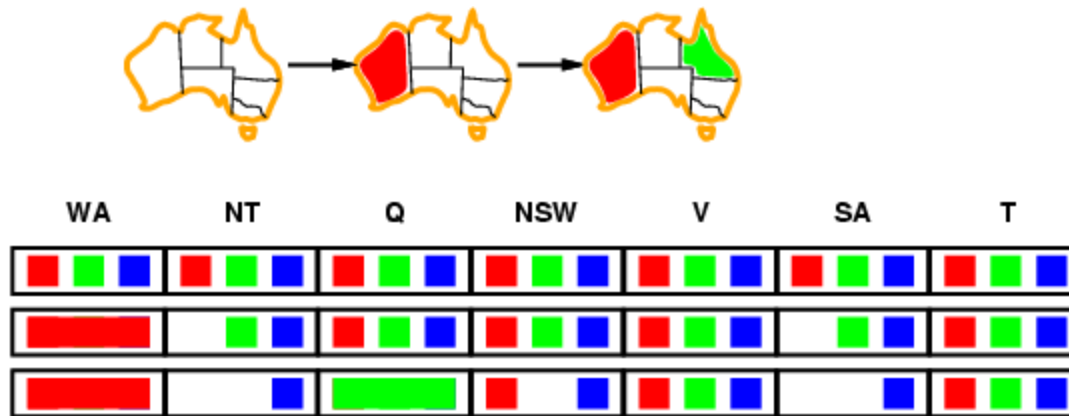
# Forward checking

- **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



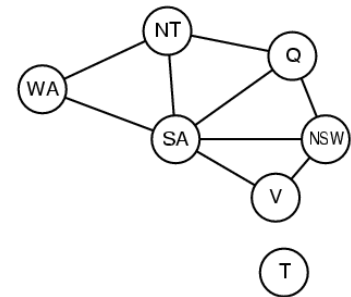
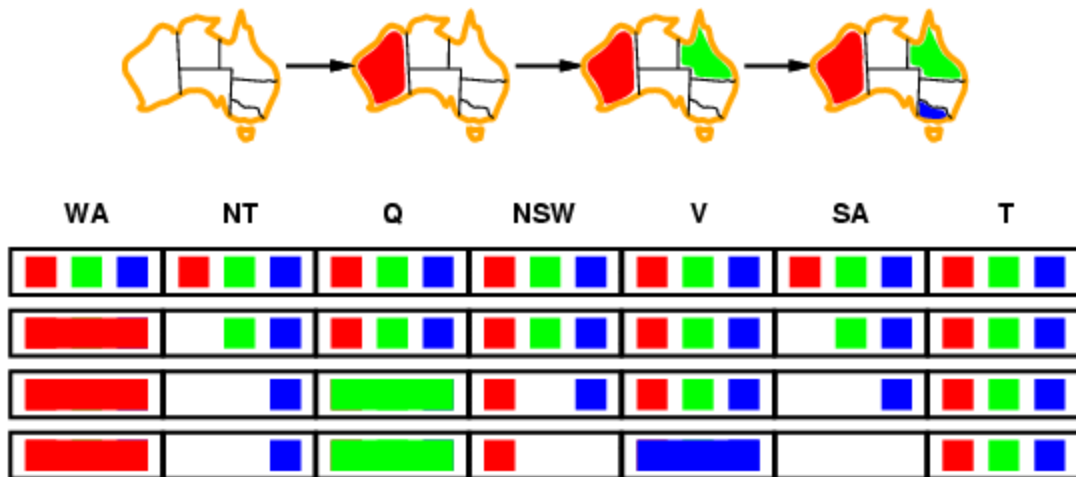
# Forward checking

- **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



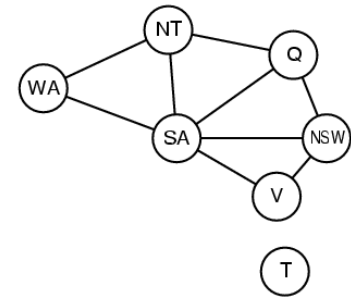
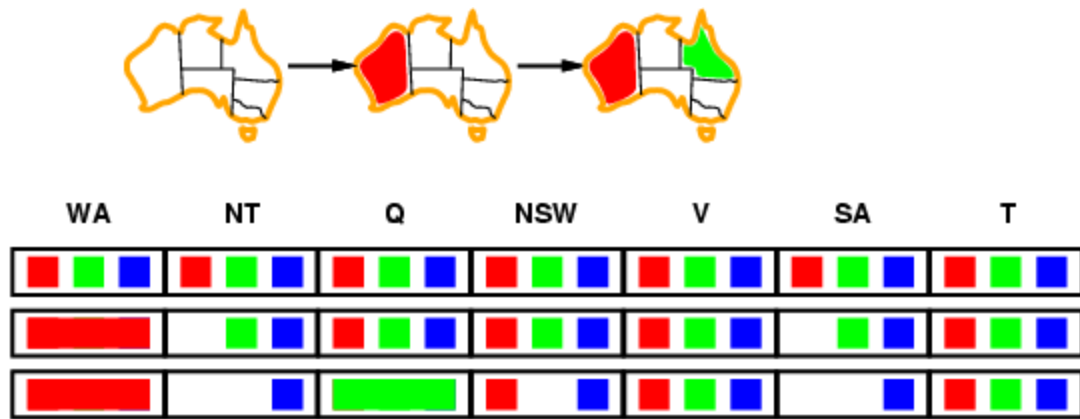
# Forward checking

- **Idea:**
  - Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values



# Constraint propagation

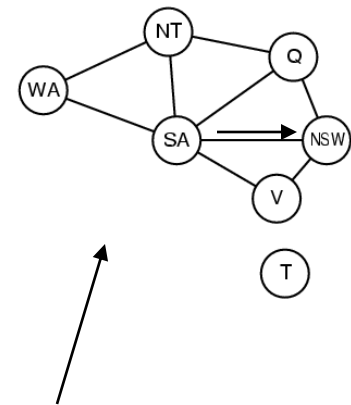
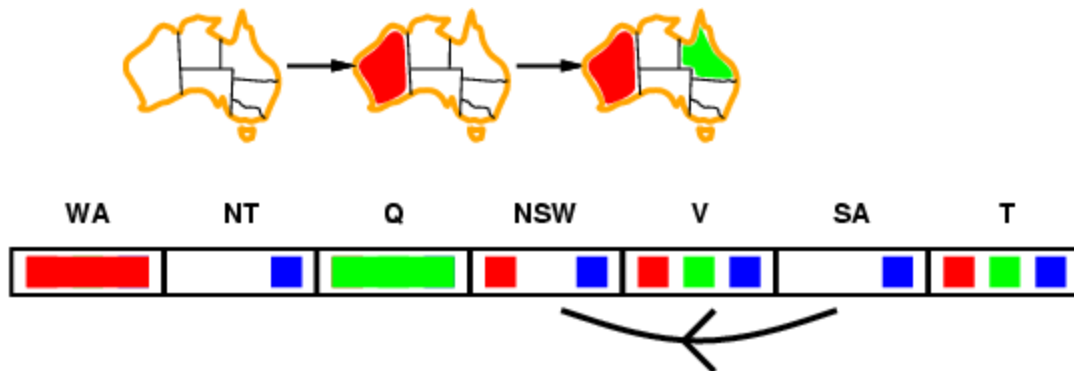
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



- NT and SA cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints locally

# Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$  is consistent iff  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$

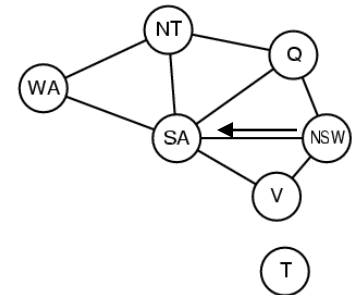
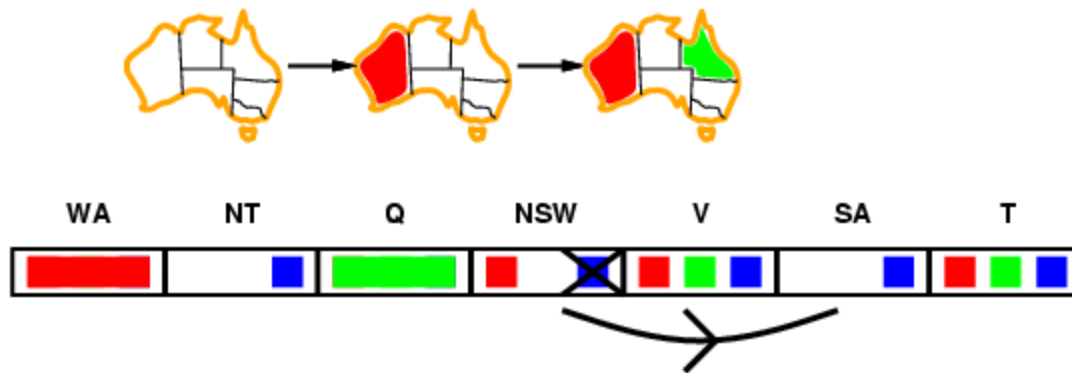


constraint propagation propagates arc consistency on the graph.



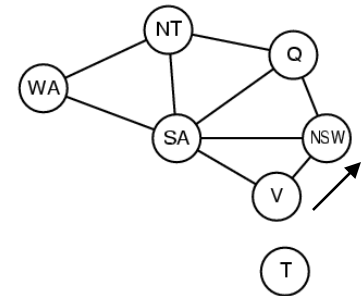
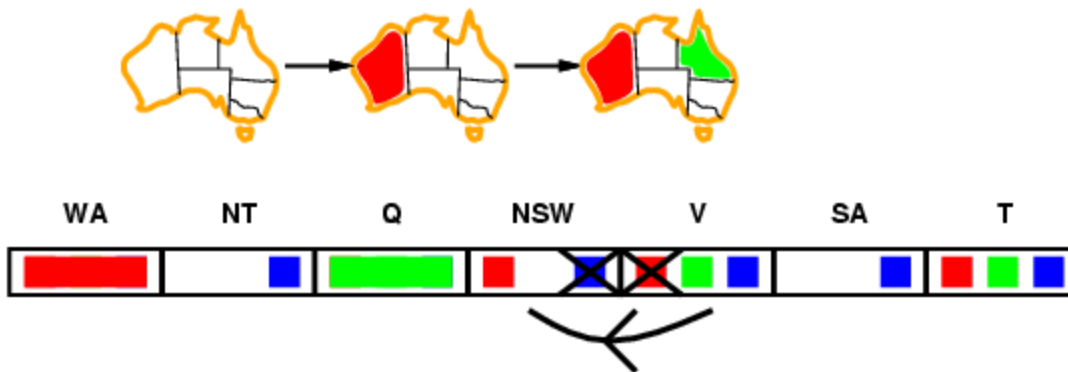
# Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$  is consistent iff  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$



# Arc consistency

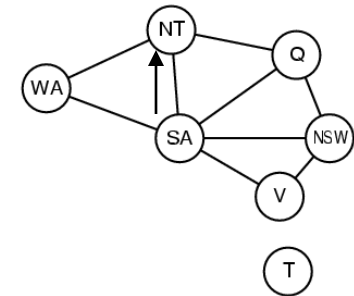
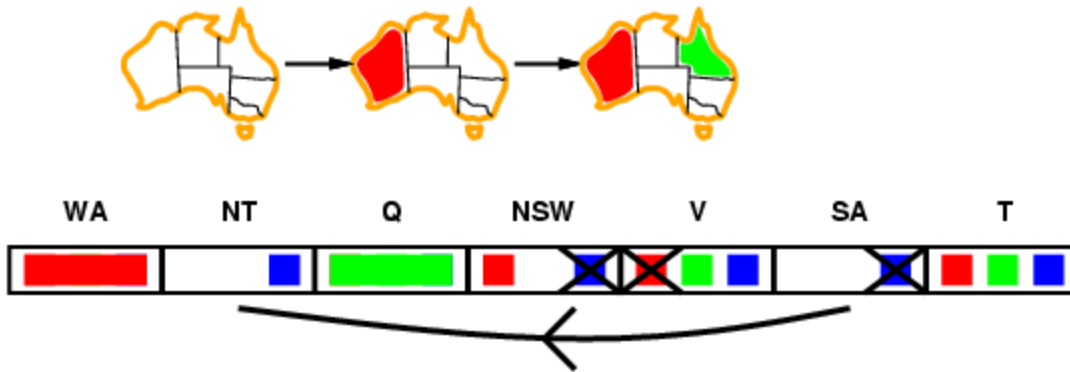
- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$  is consistent iff  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$



- If  $X$  loses a value, neighbors of  $X$  need to be rechecked

# Arc consistency

- Simplest form of propagation makes each arc **consistent**
- $X \rightarrow Y$  is consistent iff  
for **every** value  $x$  of  $X$  there is **some** allowed  $y$



- If  $X$  loses a value, neighbors of  $X$  need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment