Adversarial Search
Games

- Programs that can play competitive board games
- Will be based on Adversarial Search

General Game Playing

General Intelligence in Game-Playing Agents (GIGA'13)

(http://giga13.ru.is)

General Information

Artificial Intelligence (AI) researchers have for decades worked on building game-playing agents capable of matching wits with the strongest humans in the world, resulting in several success stories for games like chess and checkers. The success of such systems has been partly due to years of relentless knowledge-engineering effort on behalf of the program developers, manually adding application-dependent knowledge to their game-playing agents. The various algorithmic enhancements used are often highly tailored towards the game at hand.

Research into general game playing (GGP) aims at taking this approach to the next level: to build intelligent software agents that can, given the rules of any game, automatically learn a strategy for playing that game at an expert level without any human intervention. In contrast to software systems designed to play one specific game, systems capable of playing arbitrary unseen games cannot be provided with game-specific domain knowledge a priori. Instead, they must be endowed with high-level abilities to learn strategies and perform abstract reasoning. Successful realization of such programs poses many interesting research challenges for a wide variety of artificial-intelligence sub-areas including (but not limited to):

- knowledge representation and reasoning
- heuristic search and automated planning
- computational game theory
- multi-agent systems
- machine learning

The aim of this workshop is to bring together researchers from the above sub-fields of AI to discuss how best to address the challenges of and further advance the state-of-the-art of general game-playing systems and generic artificial intelligence.

The workshop is one-day long and will be held onsite at IJCAI during the scheduled workshop period August 3rd-5th (exact day is to be announced later).

Game Playing

- Many different kinds of games!

- Choices:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Perfect information (can you see the state)?

- Want algorithms for calculating a strategy (policy) which recommends a move in each state

Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: S x A → S
  - Terminal Test: S → {t,f}
  - Terminal Utilities: S x P → R
- Solution for a player is a policy: S → A

- Class of games well-studied in AI:

  board games, which can be characterized as
  deterministic, turn-taking, two-player, zero-sum
  games with ***perfect information***

Games as Adversarial Search

- Components:
  - States:
  - Initial state:
  - Successor function:
  - Terminal test:
  - Utility function:

Games as Adversarial Search

- Components:
  - States: board configurations
  - Initial state: the board position and which player will move
  - Successor function: returns list of *(move, state)* pairs, each indicating a legal move and the resulting state
  - Terminal test: determines if the game is over
  - Utility function: gives a numeric value to terminal states (e.g., -1, 0, +1 in chess for loss, tie, win)

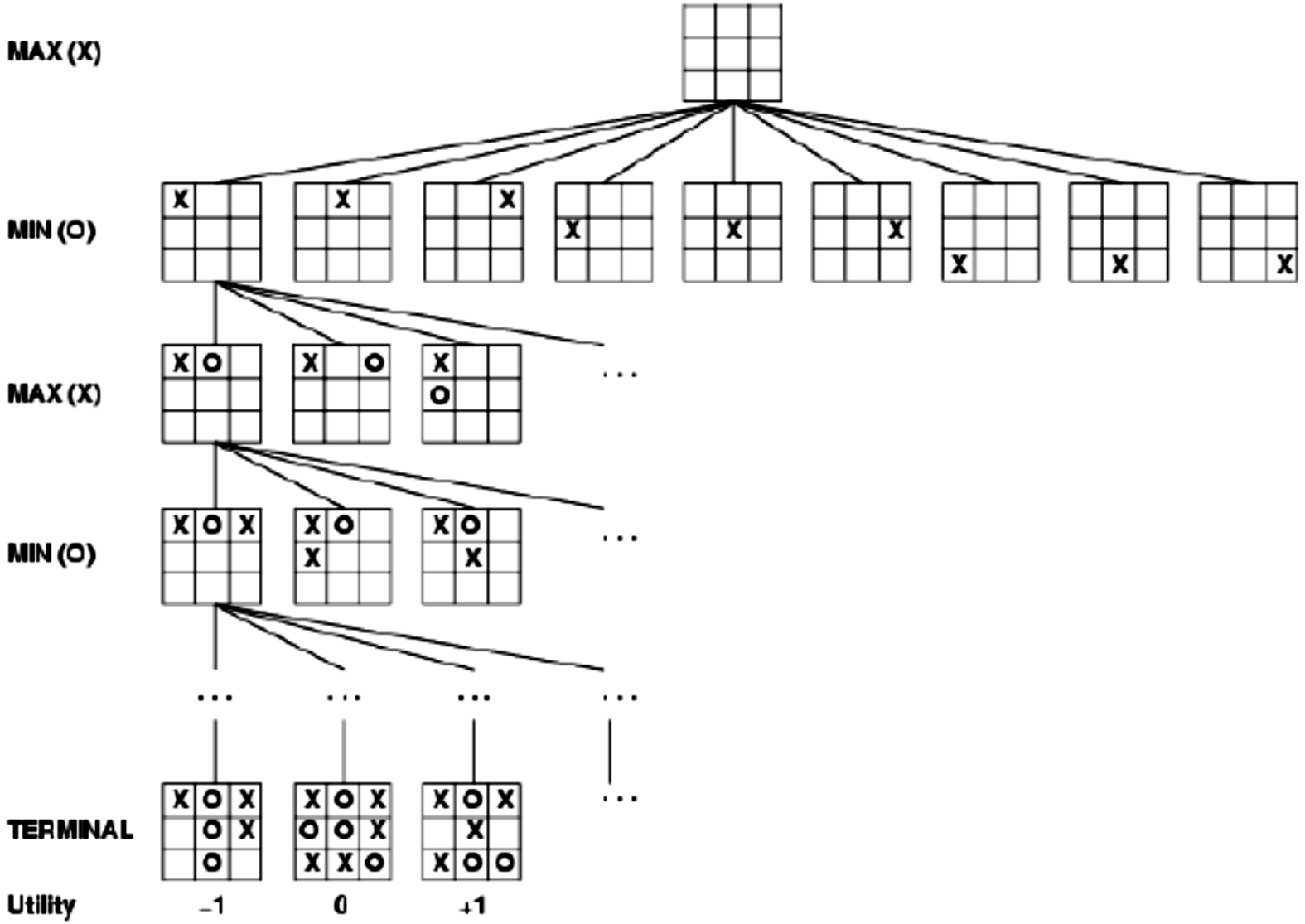
Games as Adversarial Search

Convention: first player is MAX,

2nd player is MIN

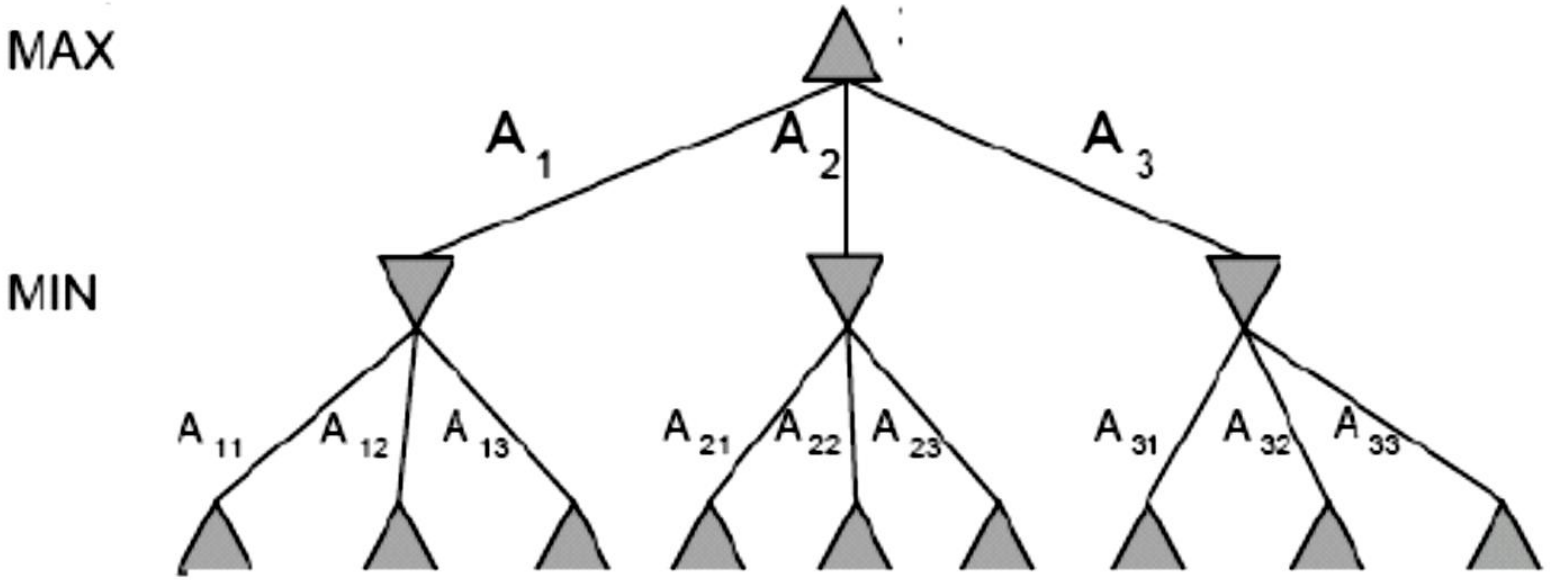
- MAX moves first and they take turns until the game is over
- Winner gets reward, loser gets penalty
- Utility values are from MAX's perspective
- Initial state + legal moves define the *game tree*
- MAX uses game tree to determine next move

Game Tree for Tic-Tac-Toe



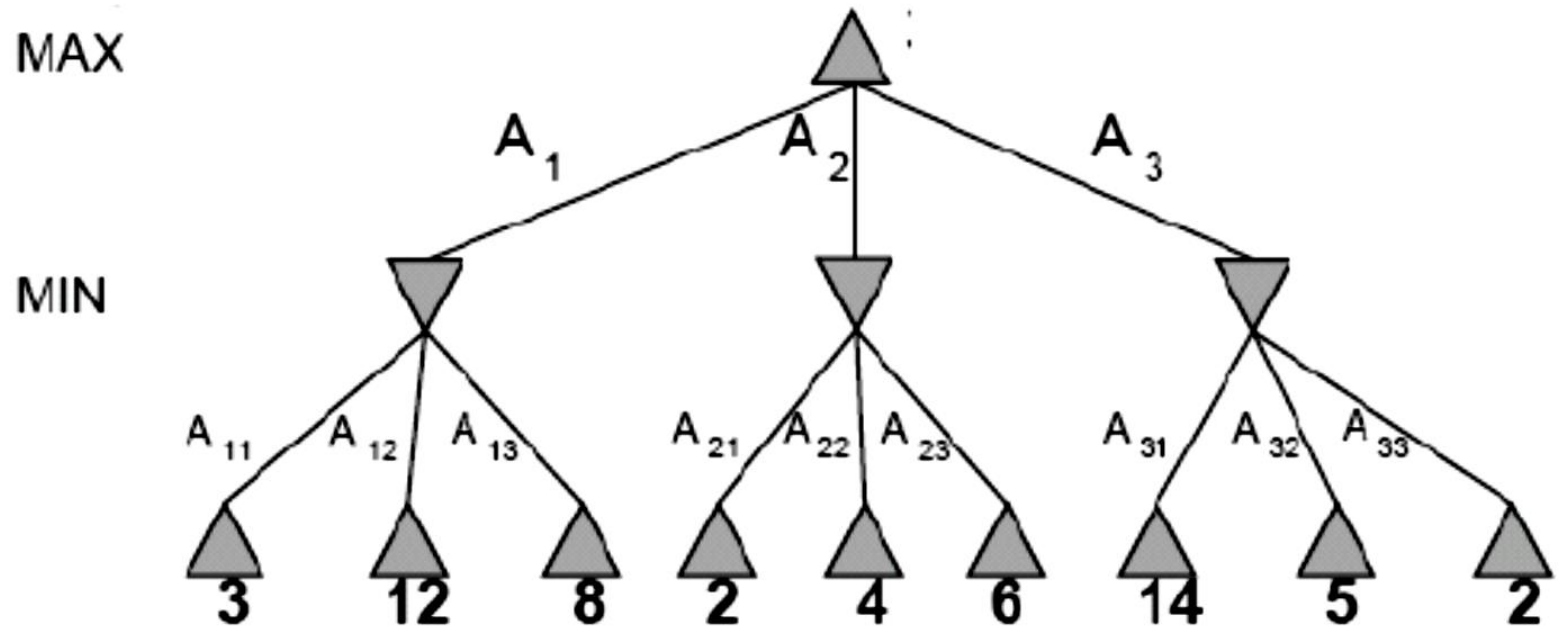
Minimax Example

Two-Ply Game Tree



"ply" = one action by one player, "move" = two plies.

Two-Ply Game Tree



Minimax Algorithm

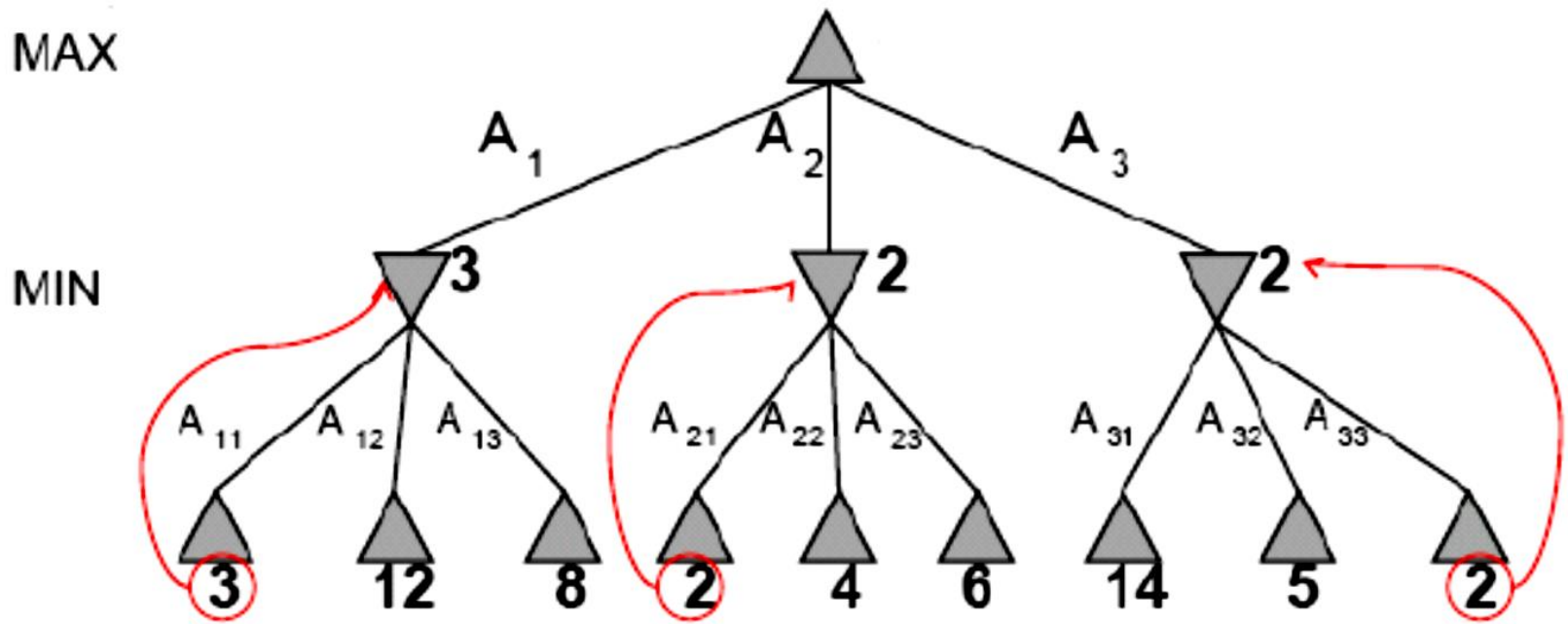
MINIMAX-VALUE(n)=

UTILITY(n)	if n is a terminal
$\max_{s \in succ(n)}$ MINIMAX-VALUE(s)	if n is a MAX node
$\min_{s \in succ(n)}$ MINIMAX-VALUE(s)	if n is a MIN node

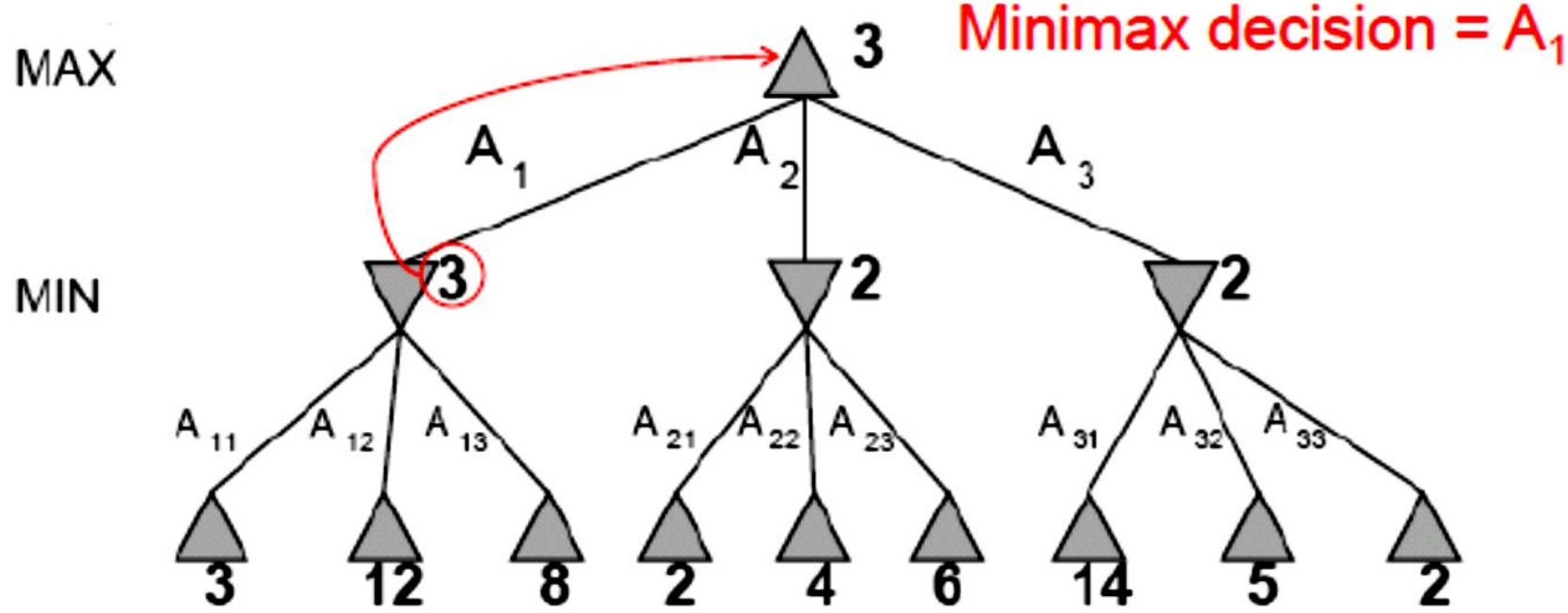
For the MAX player

1. Generate the game to terminal states
2. Apply the utility function to the terminal states
3. Back-up values
   - At MIN ply assign minimum payoff move
   - At MAX ply assign maximum payoff move
4. At root, MAX chooses the operator that led to the highest payoff

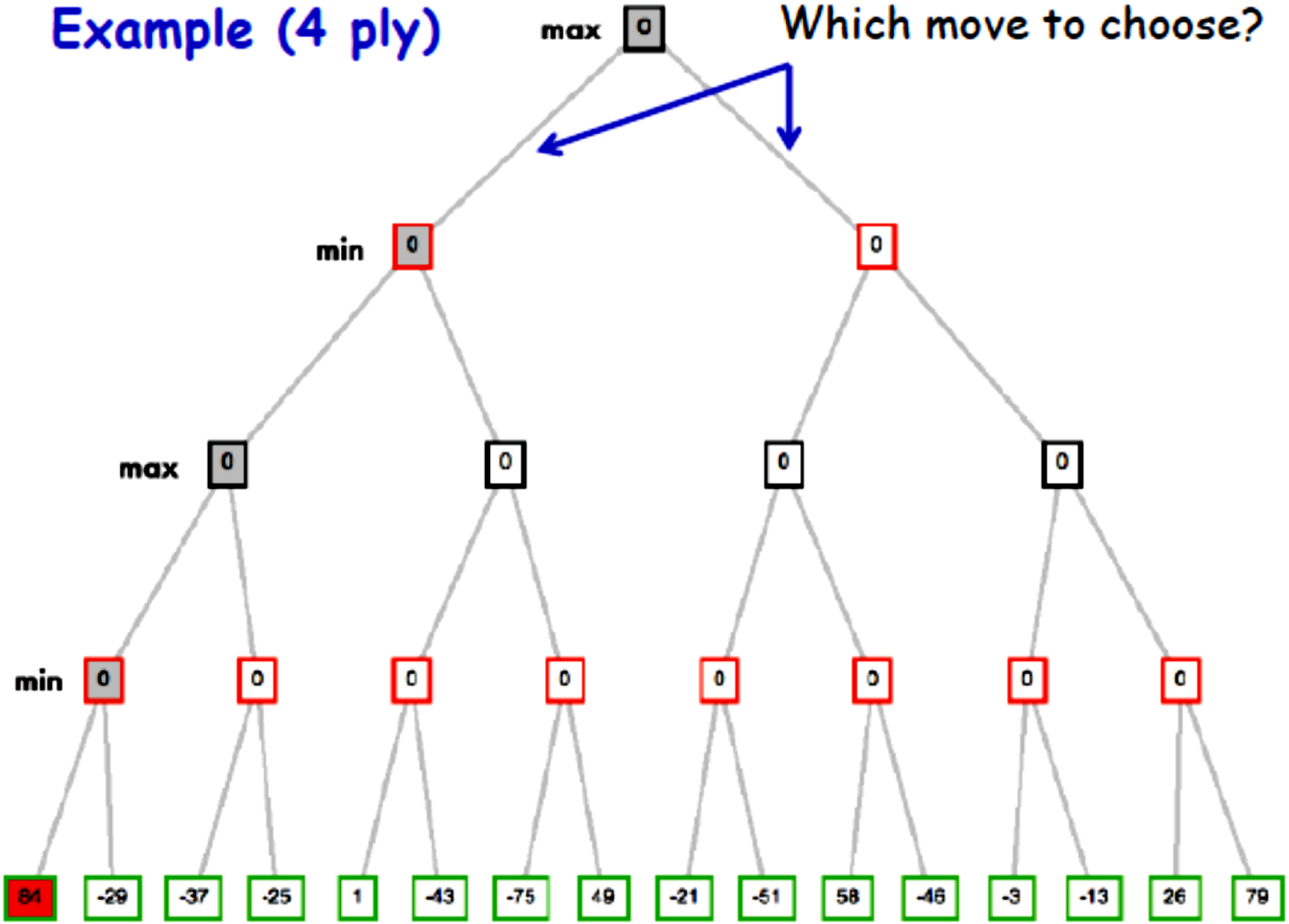
Two-Ply Game Tree

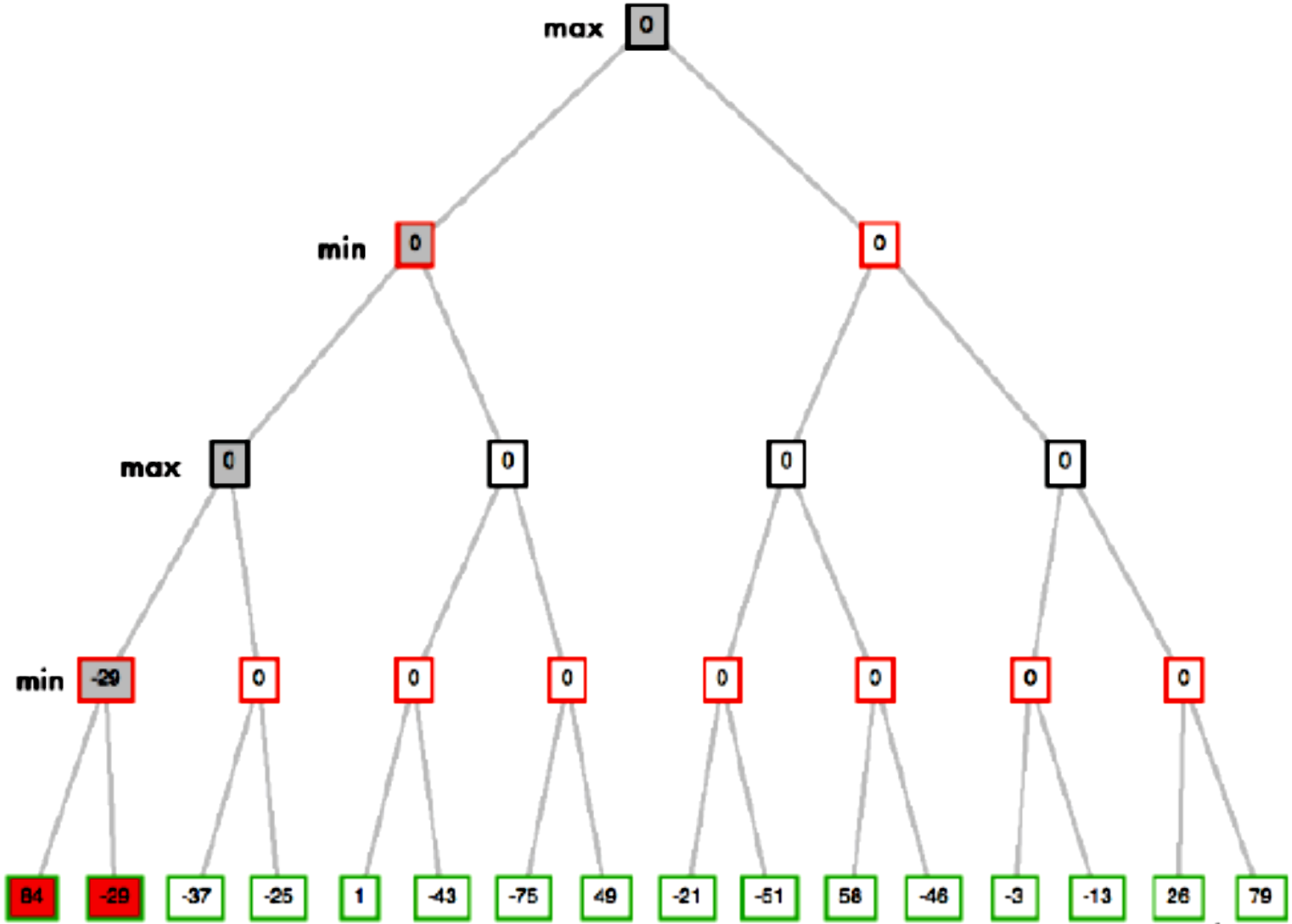


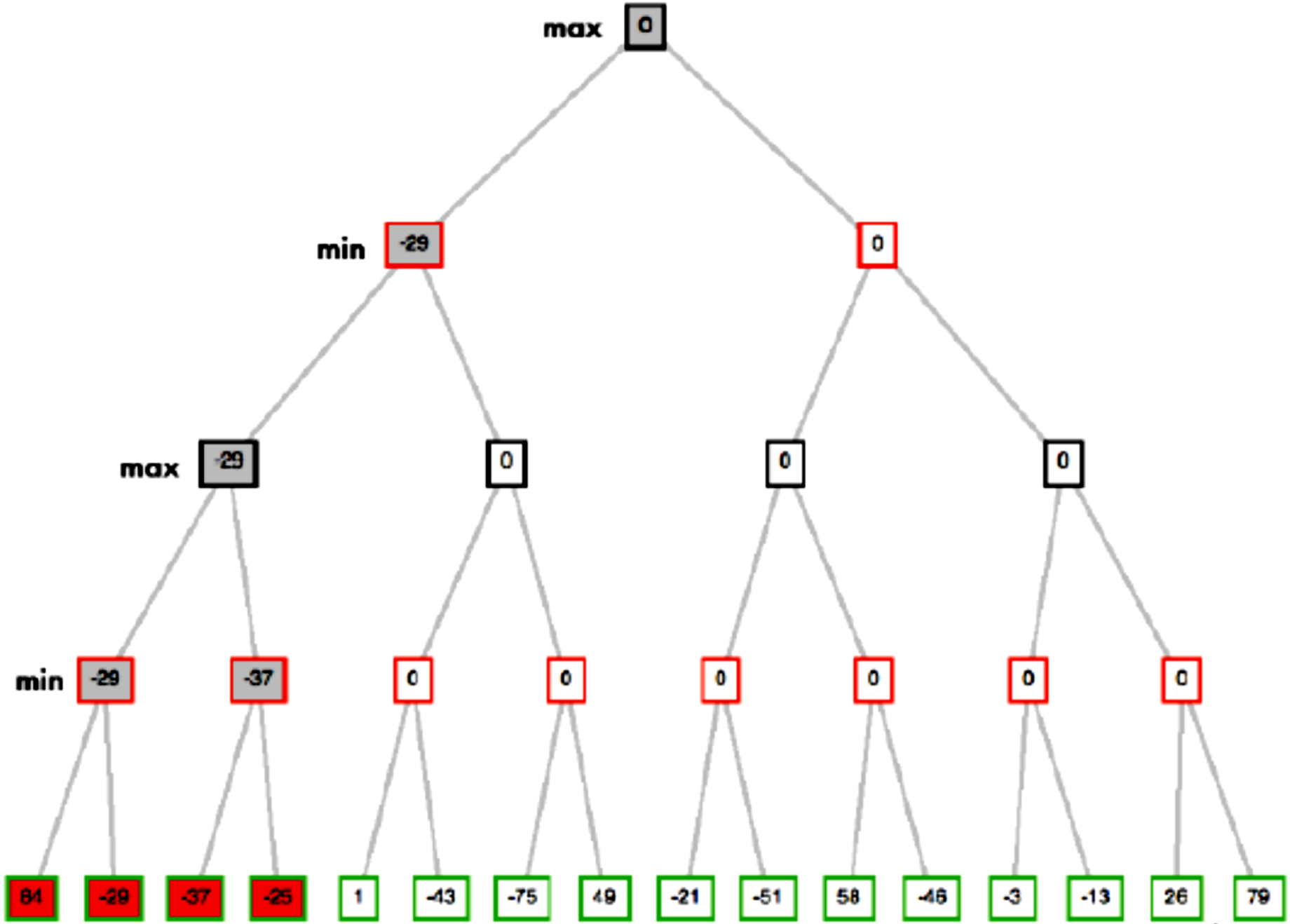
Two-Ply Game Tree

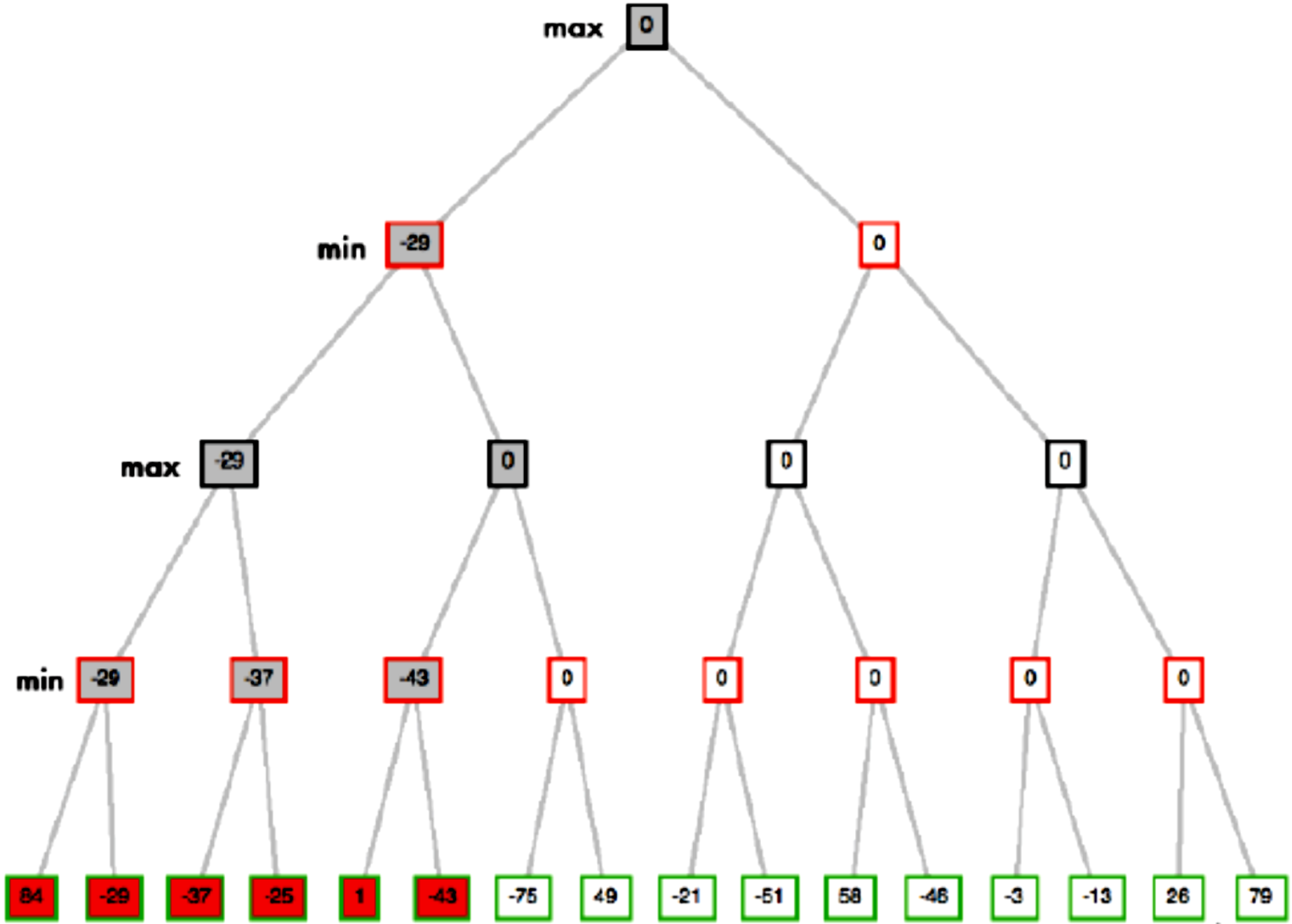


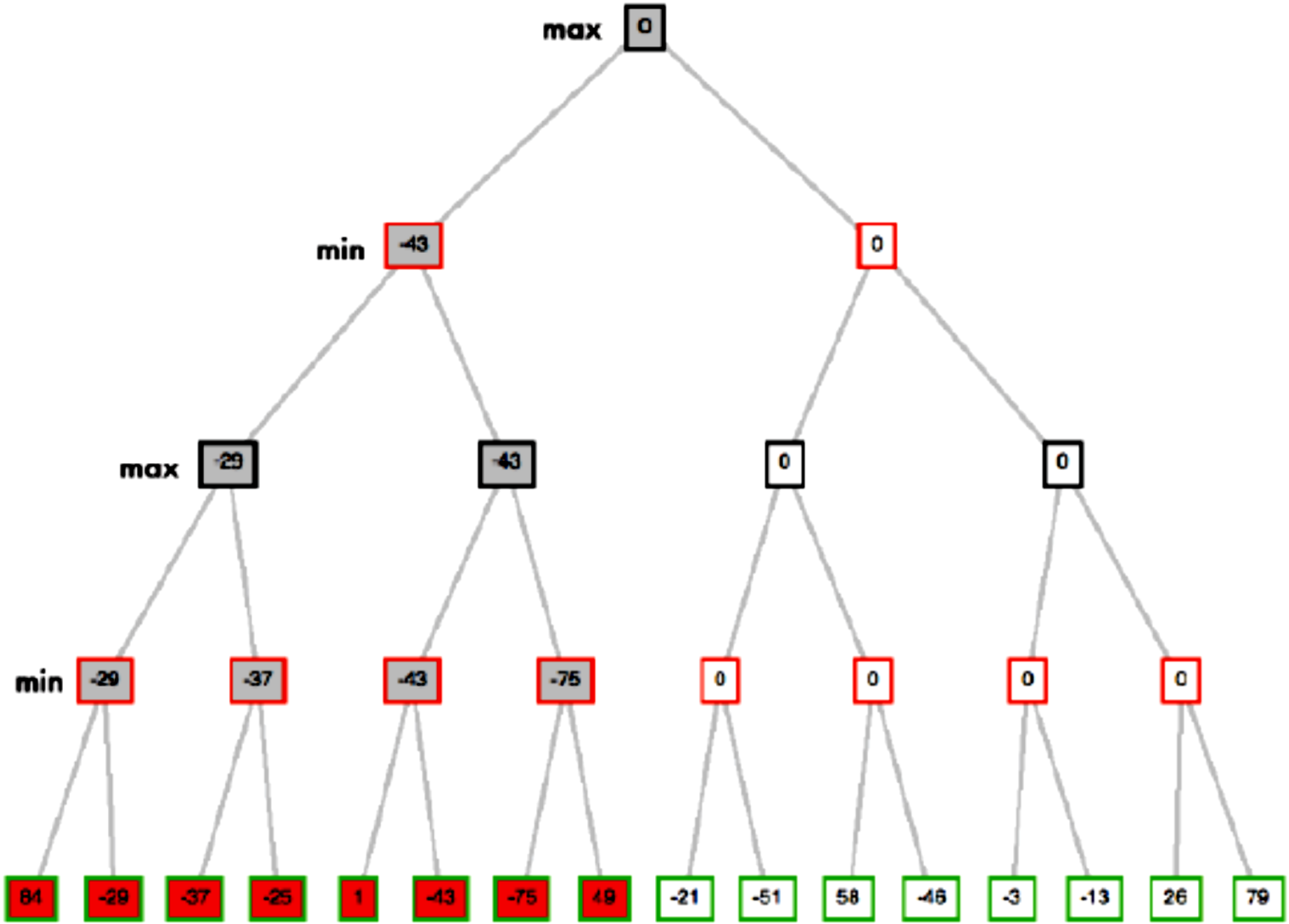
Example (4 ply)

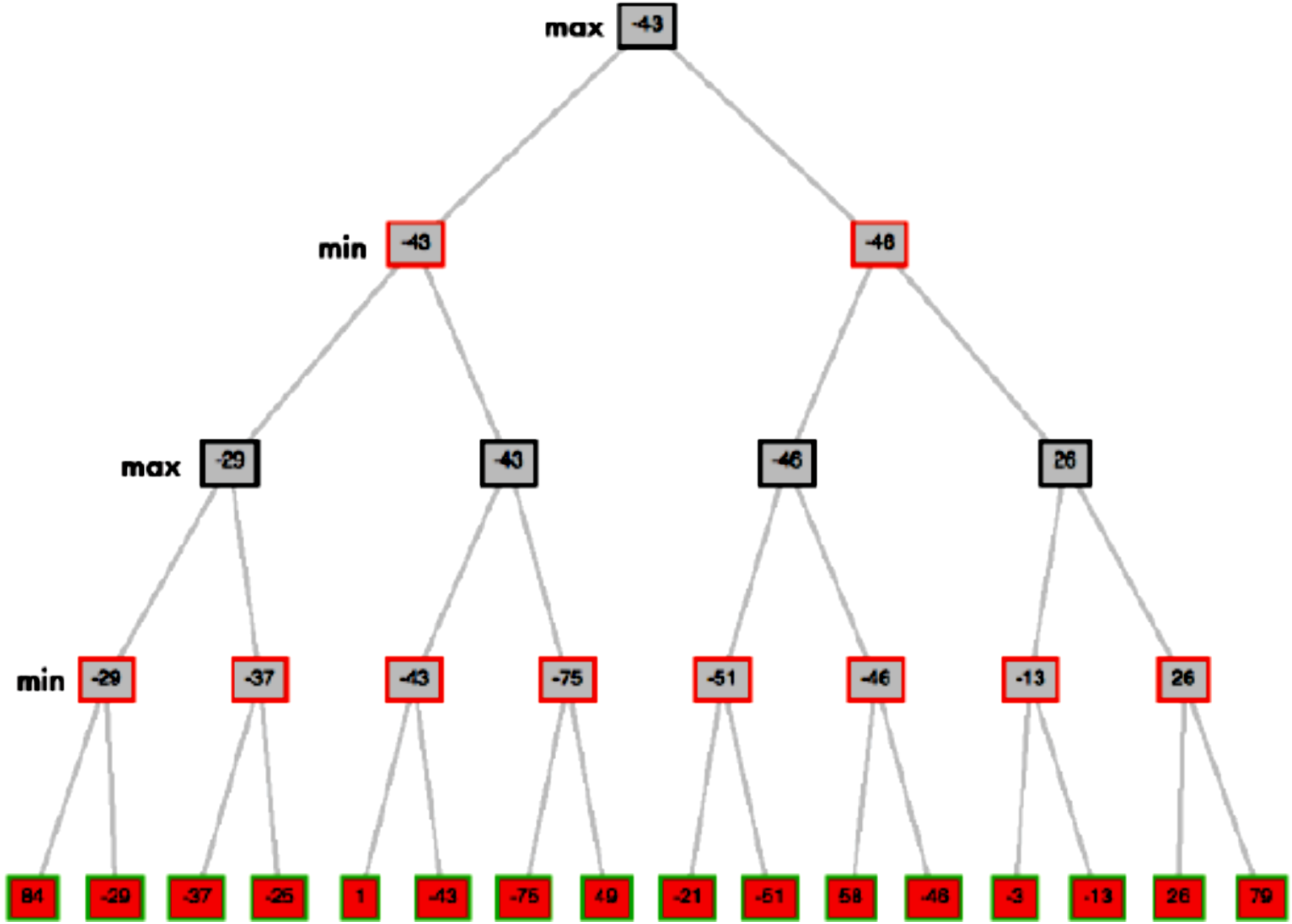


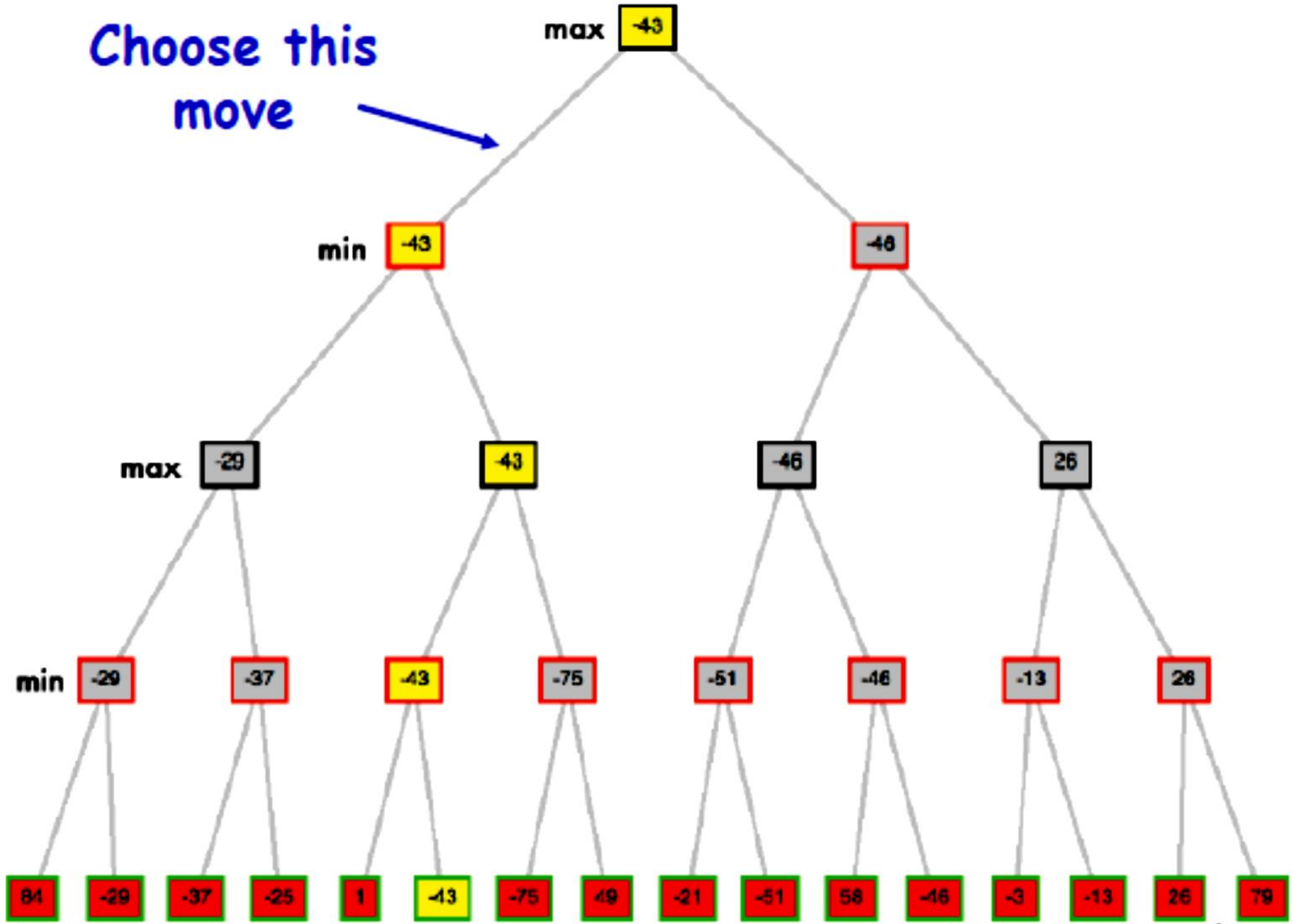

max
0
min
0
0
max
0
0
0
0
min
-29
0
0
0
0
0
0
0
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79


max
0
min
-29
0
max
-29
0
0
0
min
-29
-37
0
0
0
0
0
0
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79


max
0
min
-29
0
max
-29
0
0
0
min
-29
-37
-43
0
0
0
0
0
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79


max
0
min
-43
0
max
-29
-43
0
0
min
-29
-37
-43
-75
0
0
0
0
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79


max
-43
min
-43
-46
max
-29
-43
-46
26
min
-29
-37
-43
-75
-51
-46
-13
26
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79


Choose this move
max
-43
min
-43
-46
max
-29
-43
-46
26
min
-29
-37
-43
-75
-51
-46
-13
26
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79

Minimax Properties

- Optimal? Yes
- Complete? Yes (if tree is finite)

b is branching factor and m is depth

- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (depth-first exploration)
  - Chess:
    - branching factor b ≈ 35
    - game length m ≈ 100
    - search space $b^m \approx 35^{100} \approx 10^{154}$

Generating the tree is often impractical

But do we need to explore the entire tree ?

Is Minimax good enough?

- Chess:
  - branching factor b ≈ 35
  - game length m ≈ 100
  - search space $b^m \approx 35^{100} \approx 10^{154}$

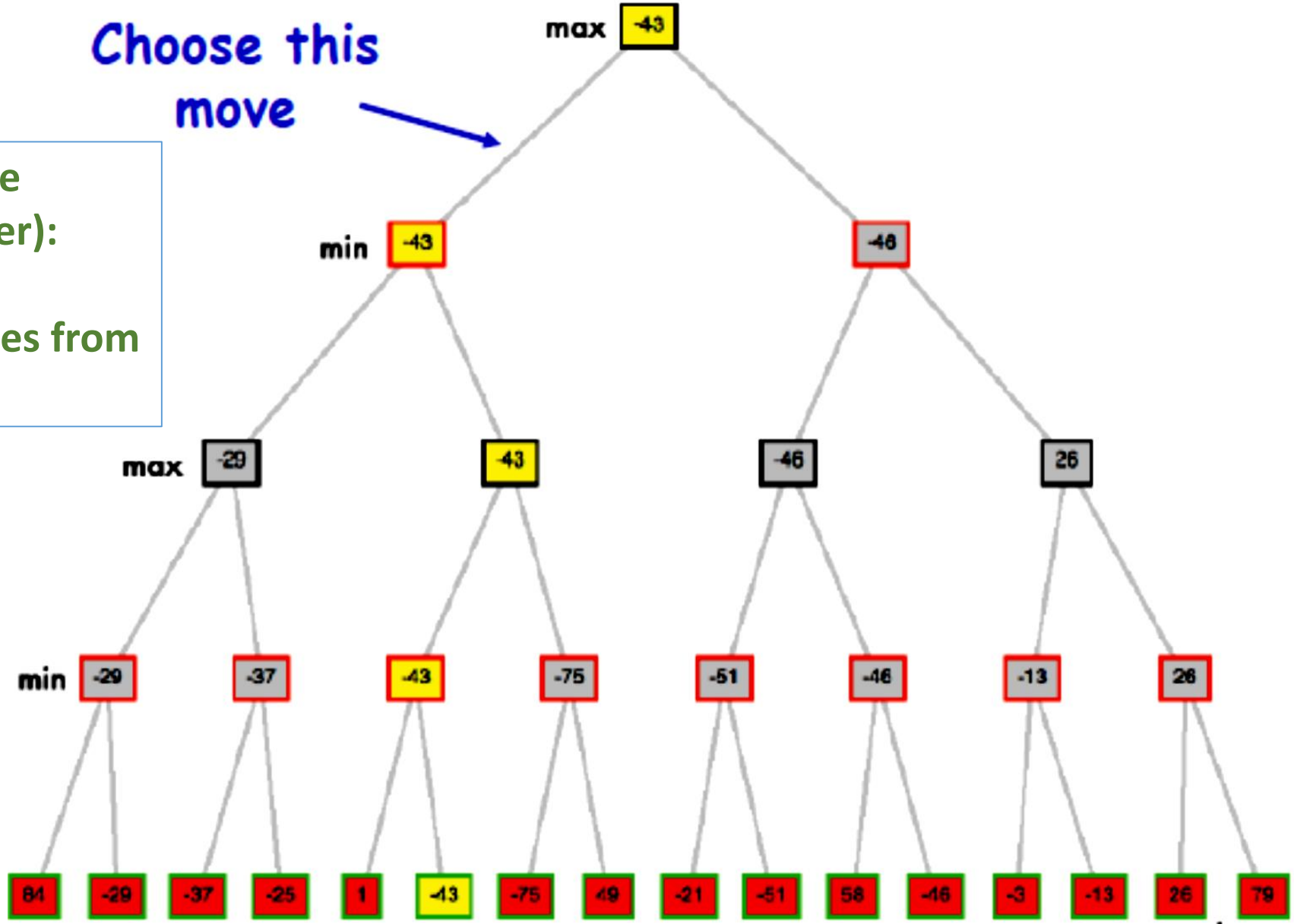
- The Universe:
  - number of atoms ≈ 10^{78}
  - age ≈ 10^{21} milliseconds

Generating the tree is often impractical

But do we need to explore the entire tree ?

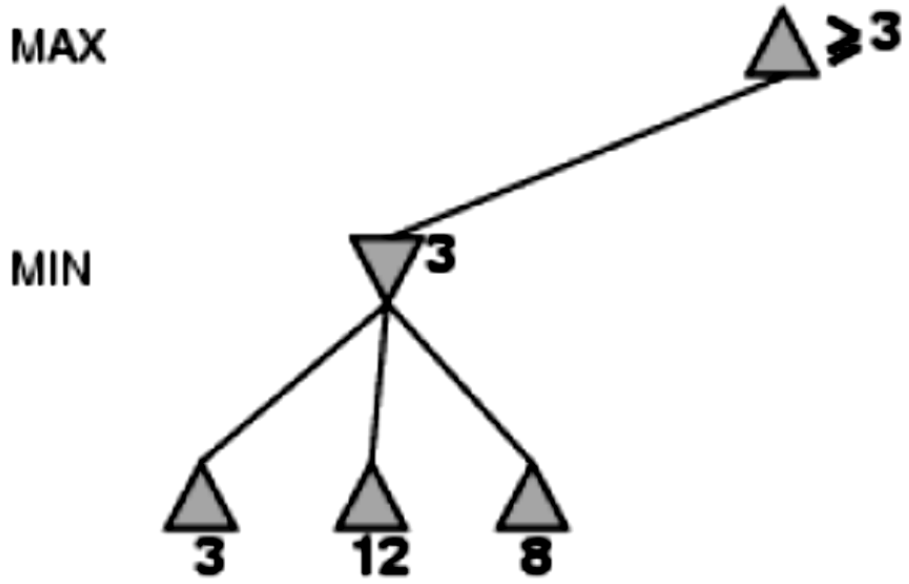
Minimax operates in two separate phases (one followed by the other):

1. Generate the game tree
2. Full propagation of utility values from all the leaf nodes to root

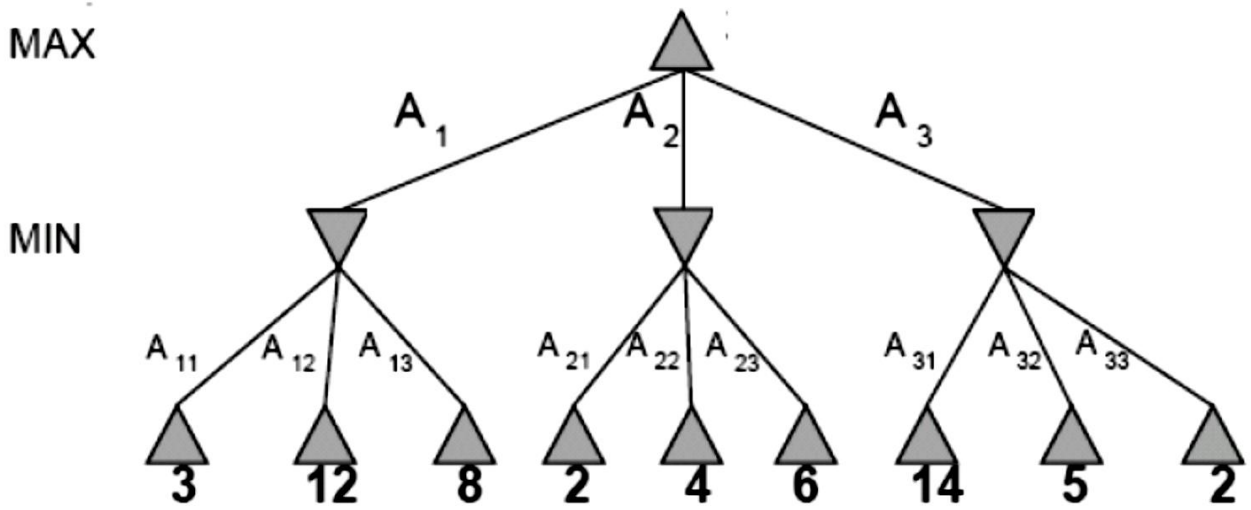


Idea:
Interleave Generation & Propagation

Pruning trees

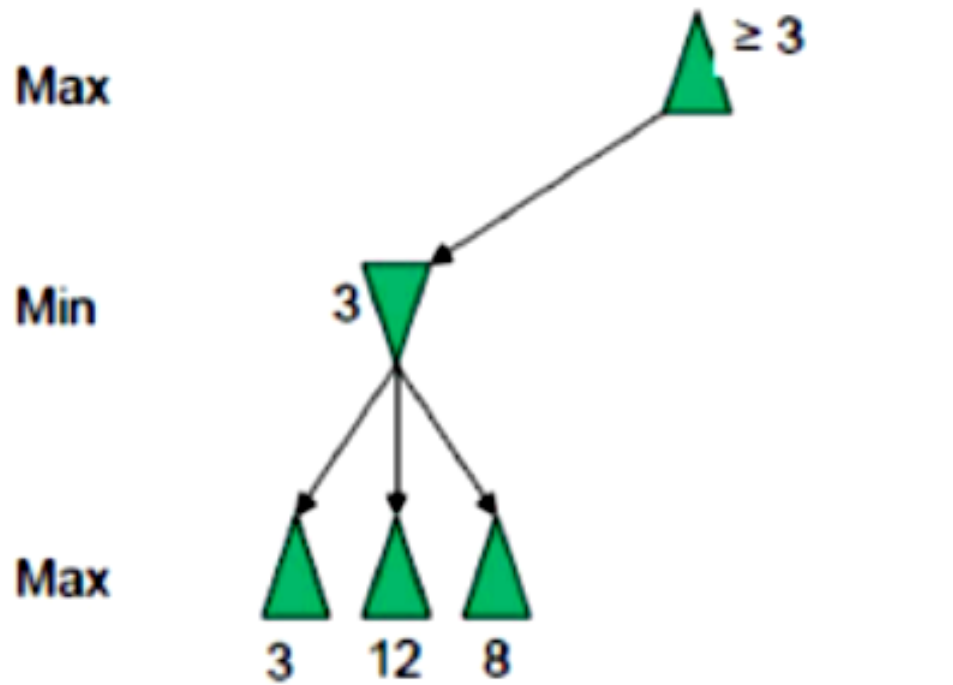


Minimax algorithm explores depth-first



Example

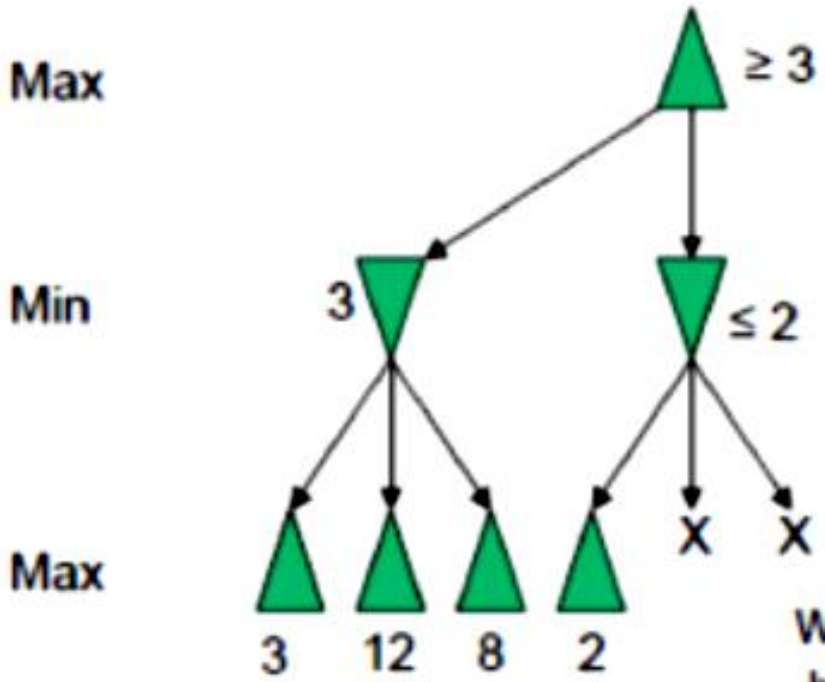
Two-ply game

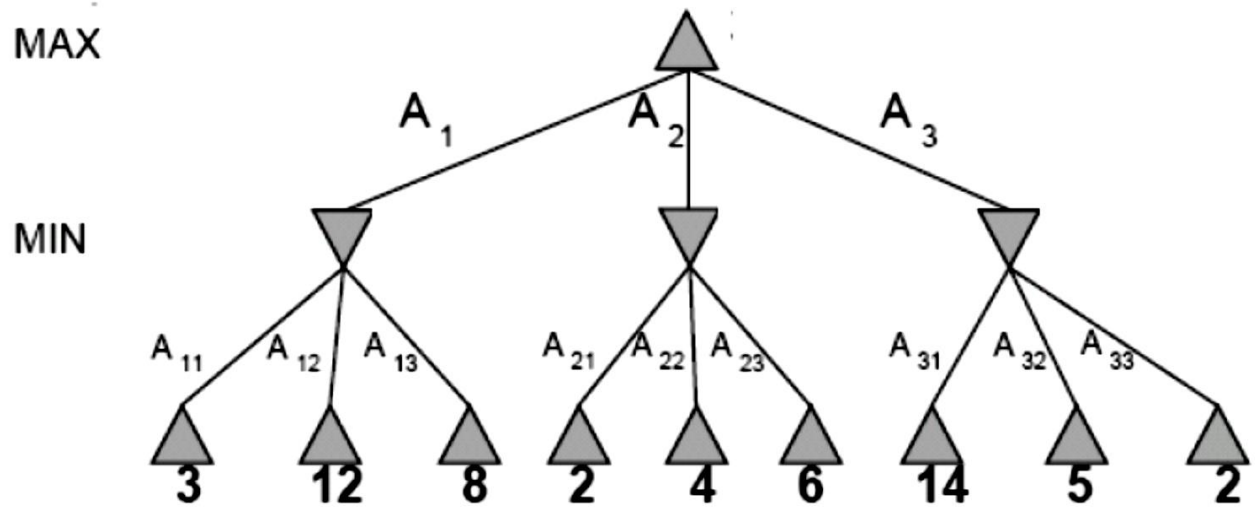


Explore tree depth-first:

Example

Two-ply game



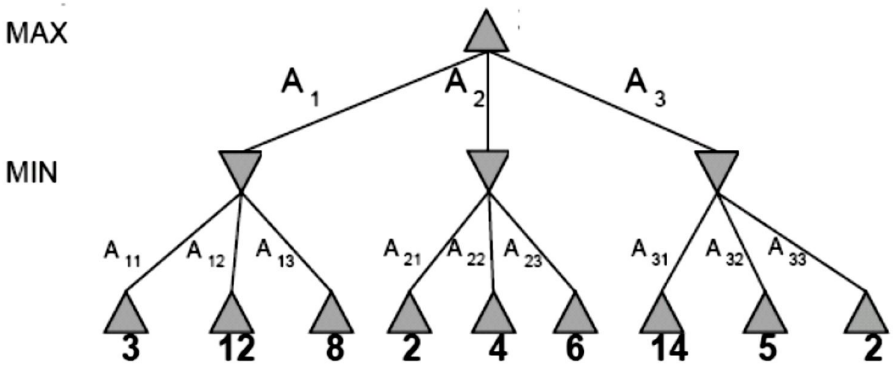


When Min is examining its moves, and it gets one back that is less than alpha (i.e., *worse* for Max), then its parent, Max, would not make that move because the move that gave alpha is better.
So Min can abandon this node right now before examining any more moves from it

This form of tree pruning is known as alpha-beta pruning

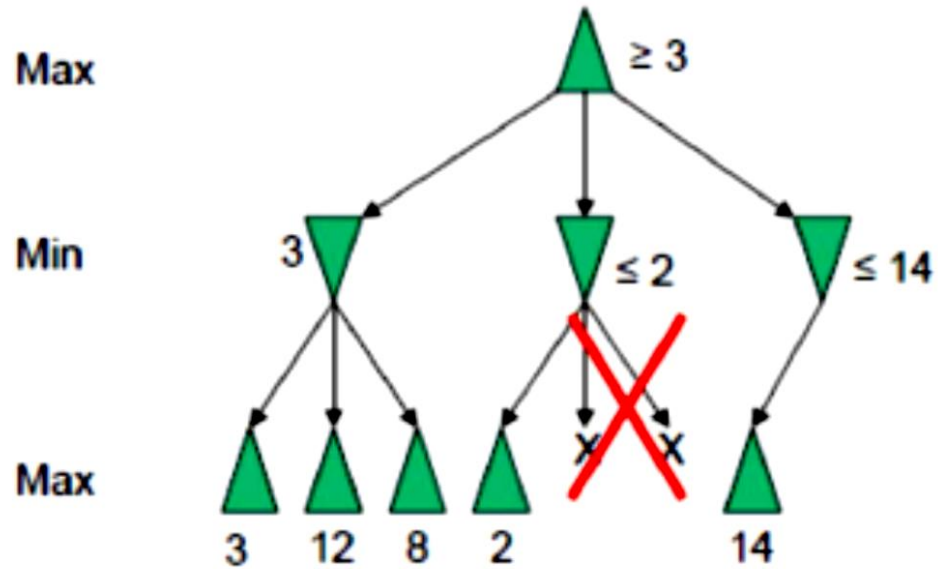
alpha = highest (best) value for MAX along current path from root

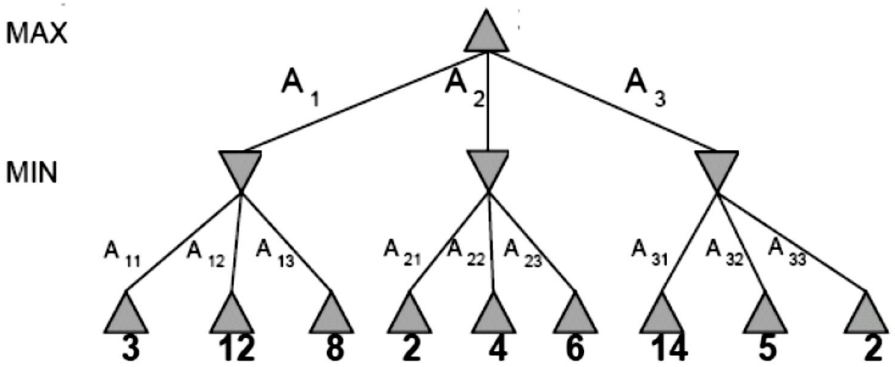
beta = lowest (best) value for MIN along current path from root



Alpha-Beta Pruning Example

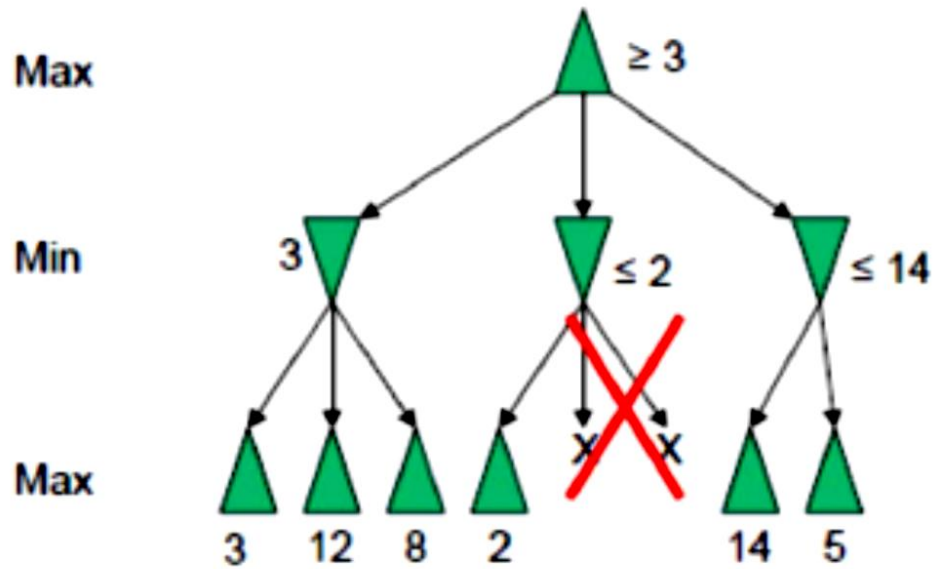
Two-ply game

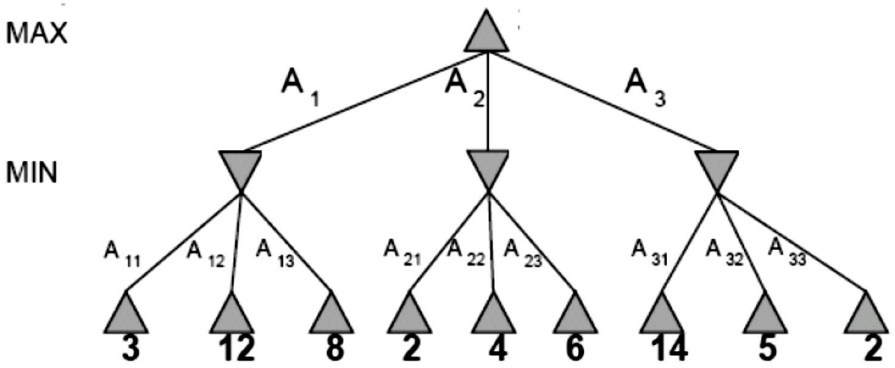




Alpha-Beta Pruning Example

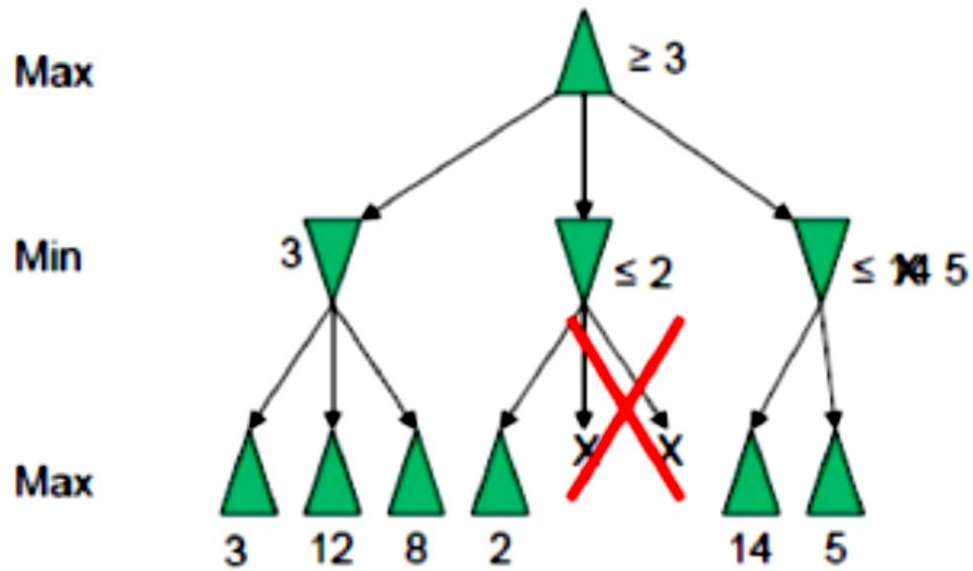
Two-ply game

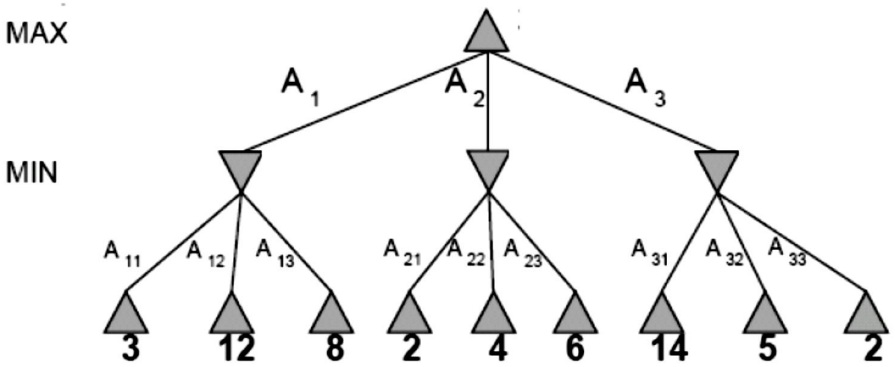




Alpha-Beta Pruning Example

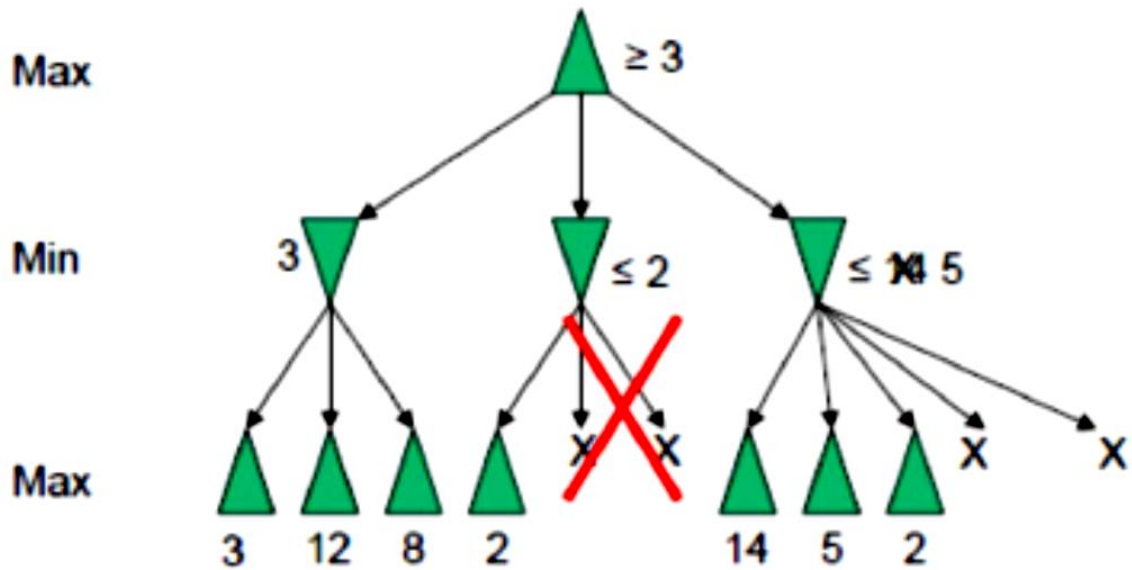
Two-ply game

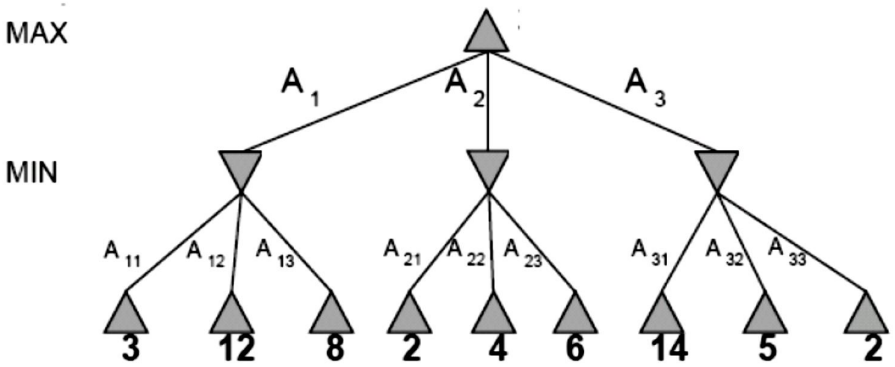




Alpha-Beta Pruning Example

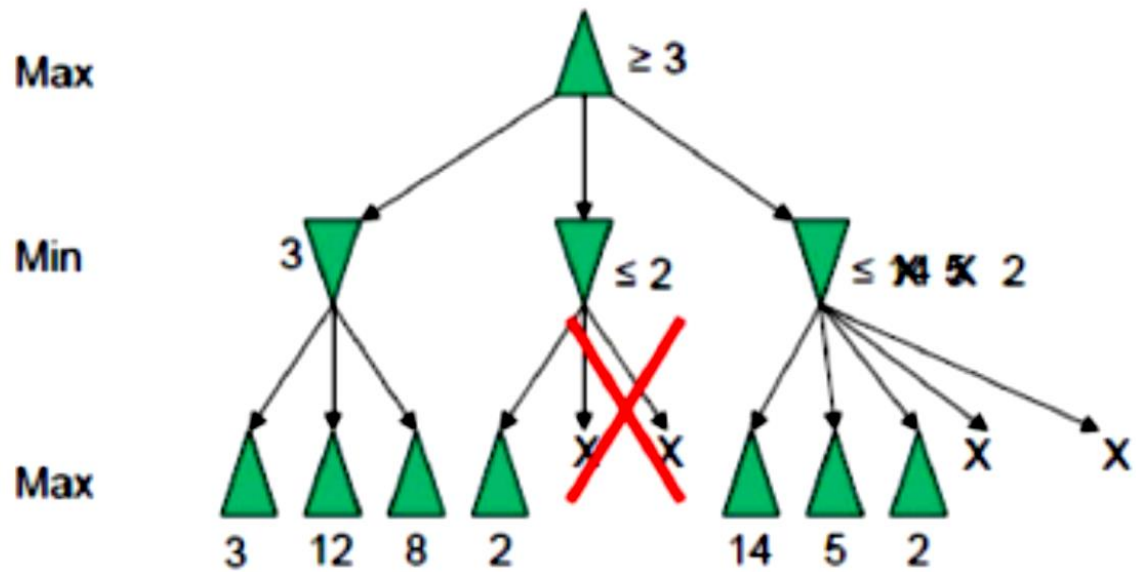
Two-ply game

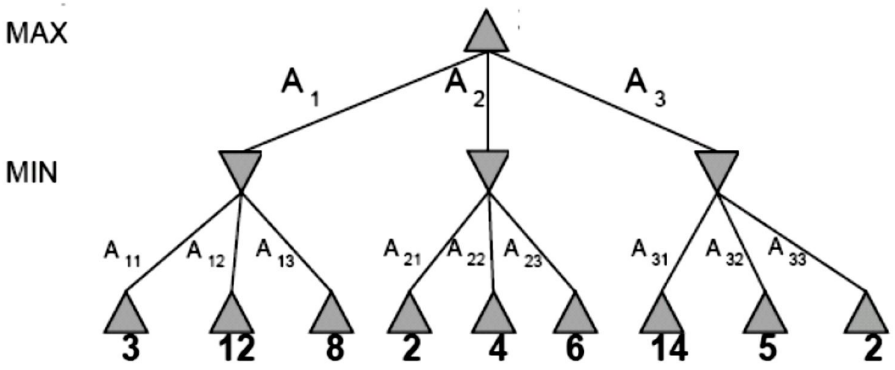




Alpha-Beta Pruning Example

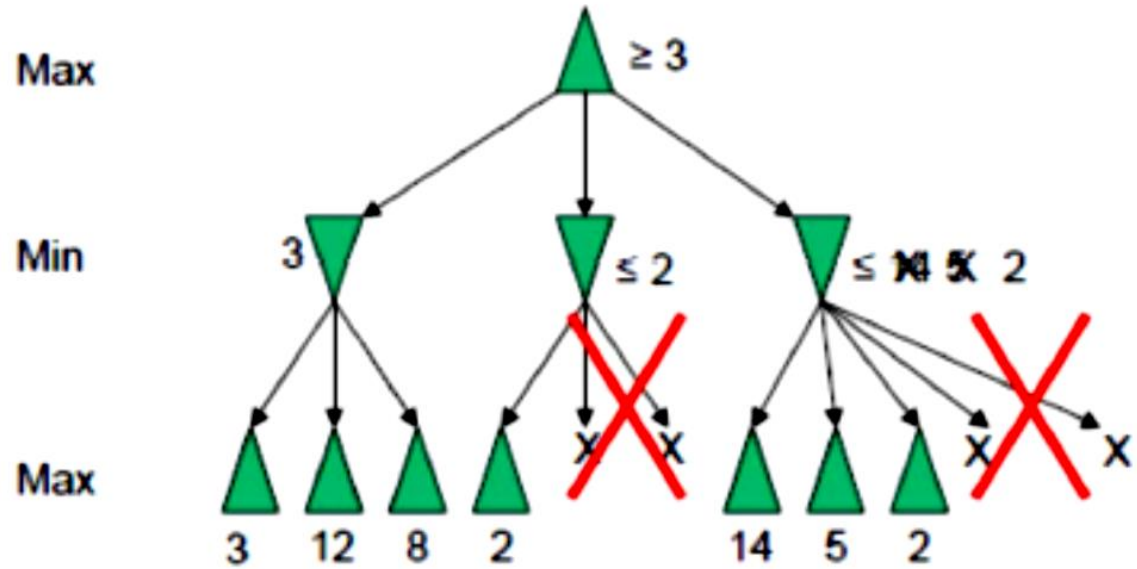
Two-ply game



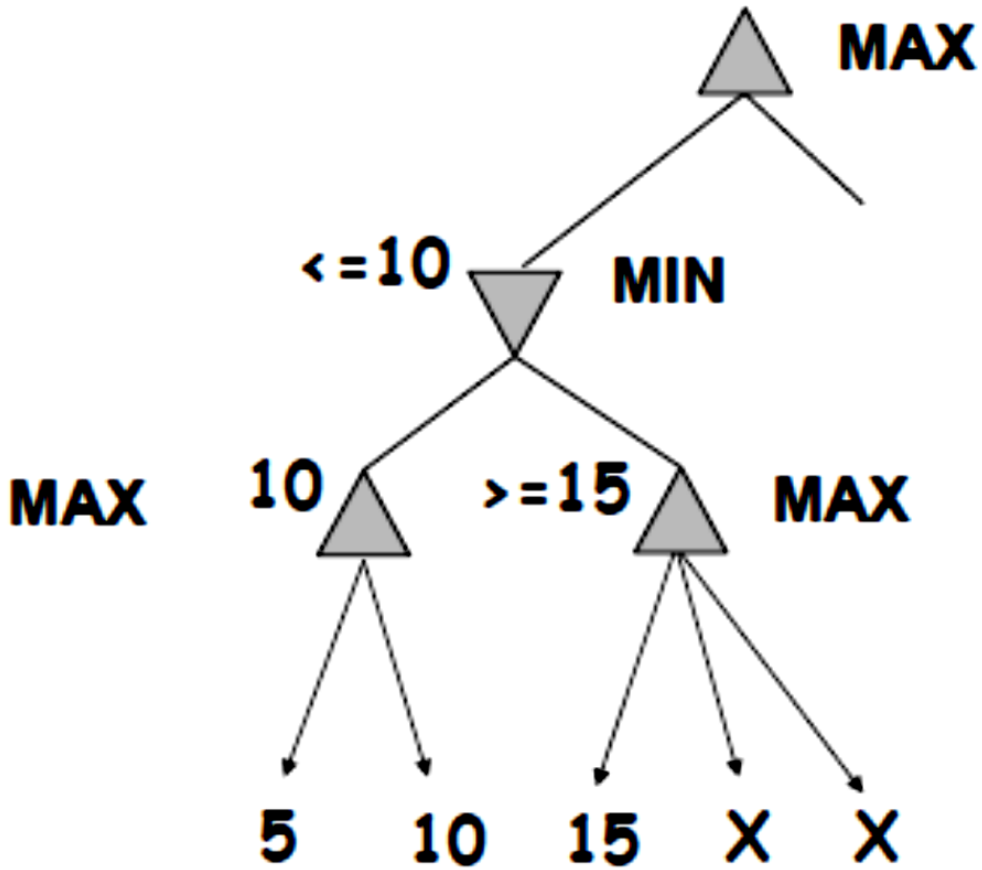


Alpha-Beta Pruning Example

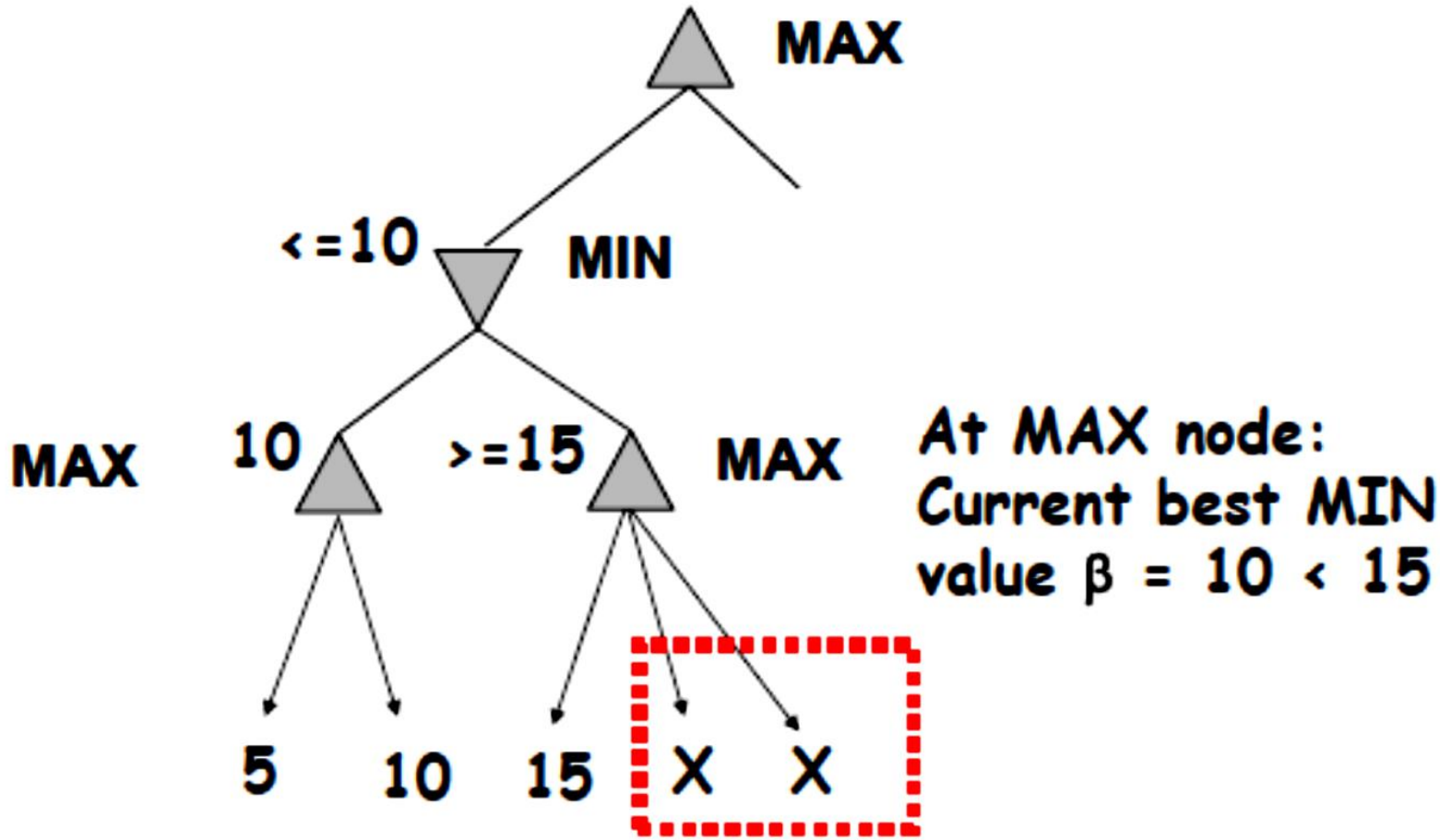
Two-ply game



One more example



One more example

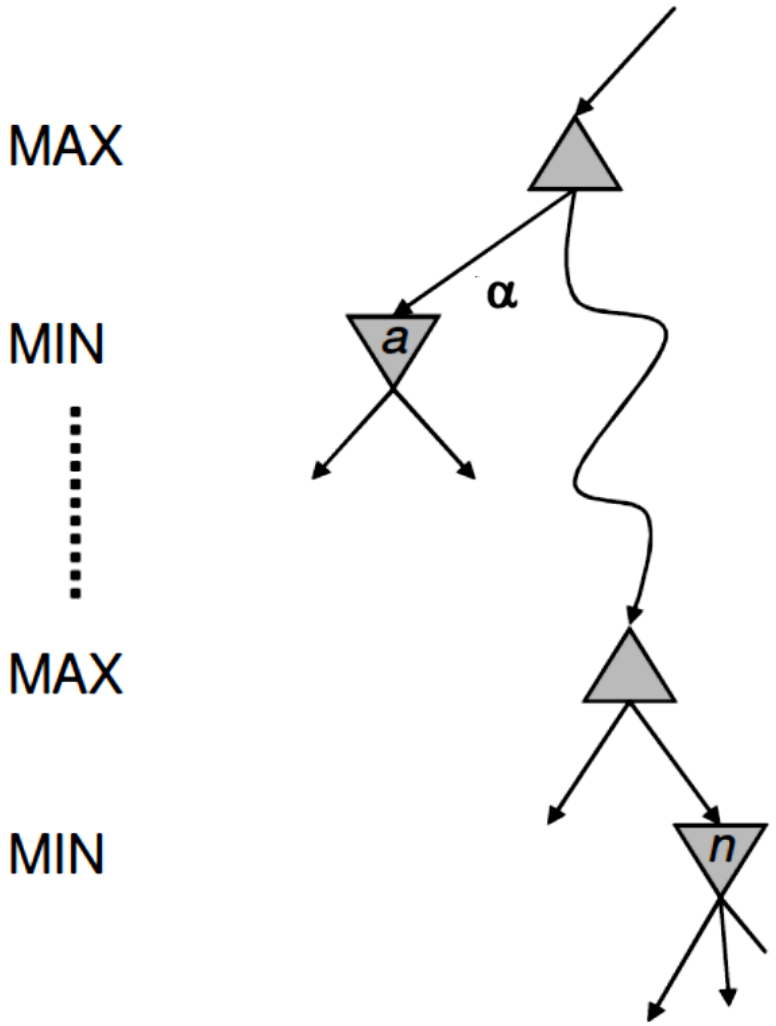


At MAX node:
Current best MIN value $\beta = 10 < 15$

No need to look at these nodes!! (these nodes can only increase MAX value from 15)

Alpha-Beta Pruning

- General configuration
  - We're computing the MIN-VALUE at *n*
  - We're looping over *n*'s children
  - *n*'s value estimate is dropping
  - *a* is the best value that MAX can get at any choice point along the current path
  - If *n* becomes worse than *a*, MAX will avoid it, so can stop considering *n*'s other children
  - Define *b* similarly for MIN



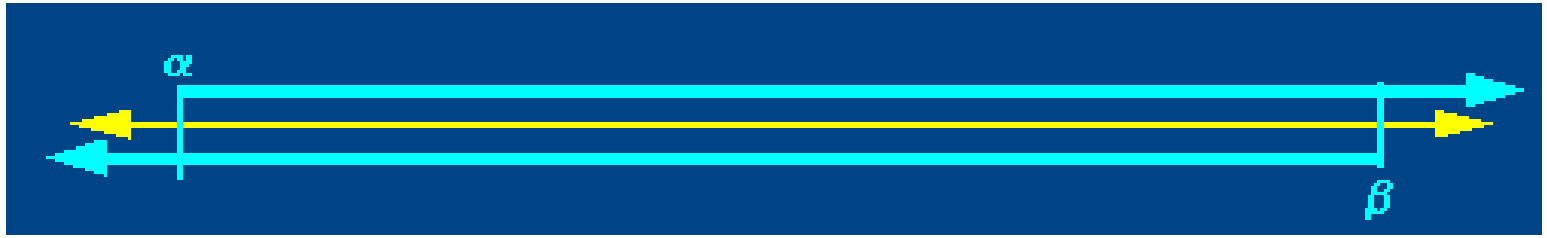
Alpha (α), Beta (β)

- We maintain two values: **α** and **β**
- α is primarily associated with Max nodes
- β is primarily associated with Min nodes

- α is a lower bound on Max's current value
- β is a upper bound on Min node's current value
- α can only increase
- β can only decrease

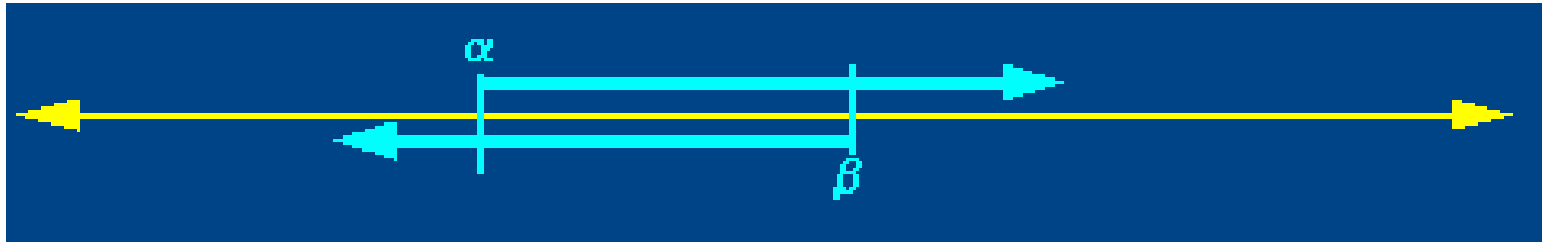
Alpha-Beta Pruning

- Initially, α = -infinity, β=infinity



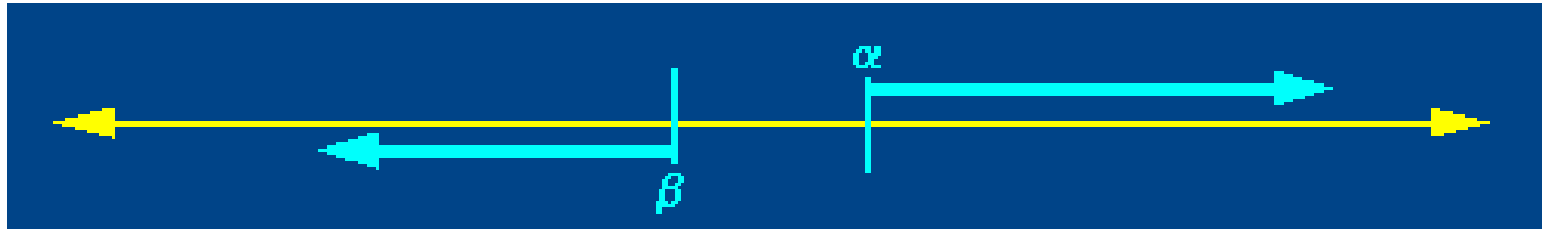
Alpha-Beta Pruning

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



Alpha-Beta Pruning

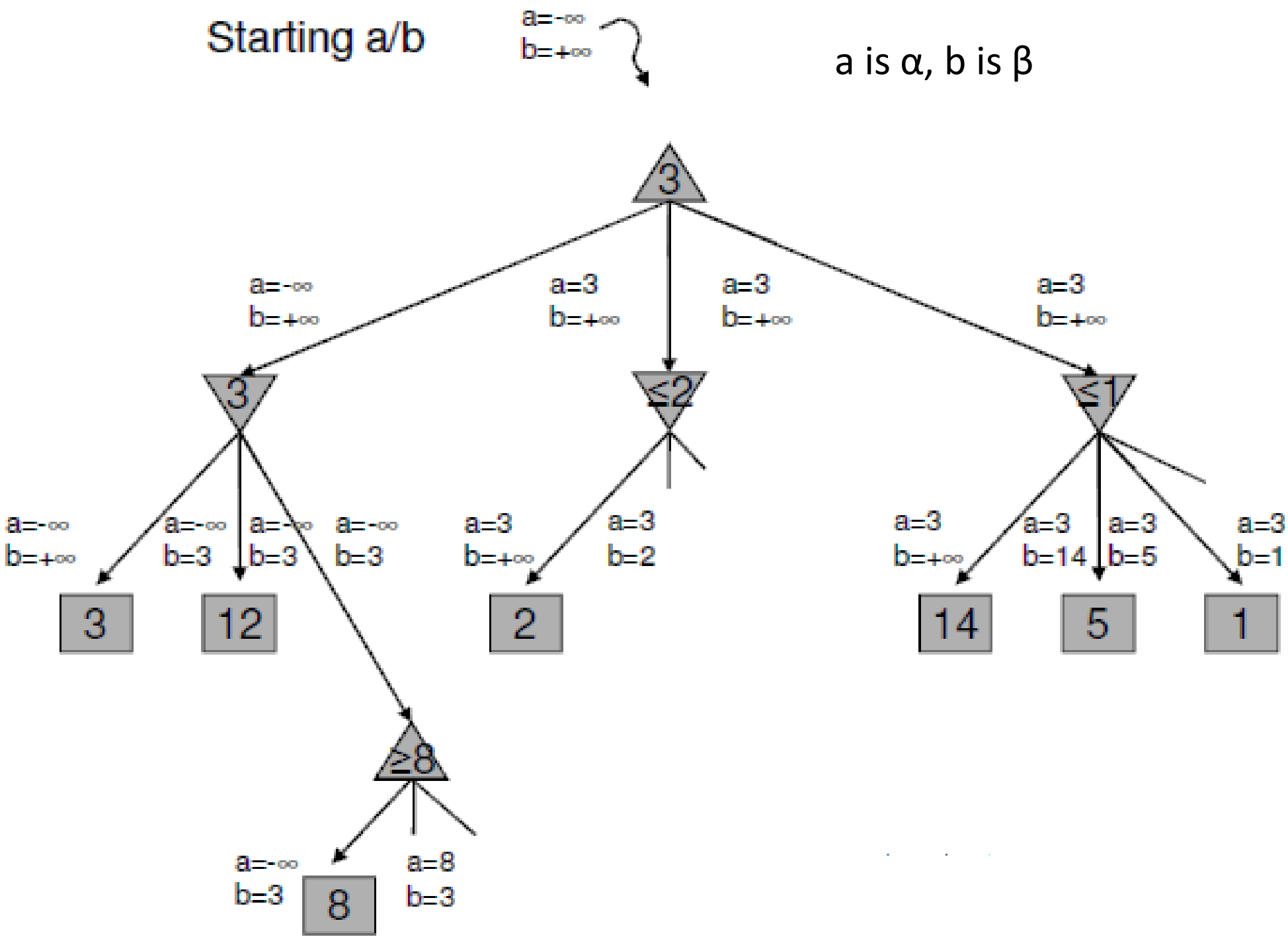
- Once there is no longer any overlap in the possible ranges of alpha and beta, it is safe to conclude that the current node is a dead end



Phases of Alpha-Beta

- Exploration and Propagate α and β
- Backup and Update

- Questions?
  - When to Explore?
  - What to Update?

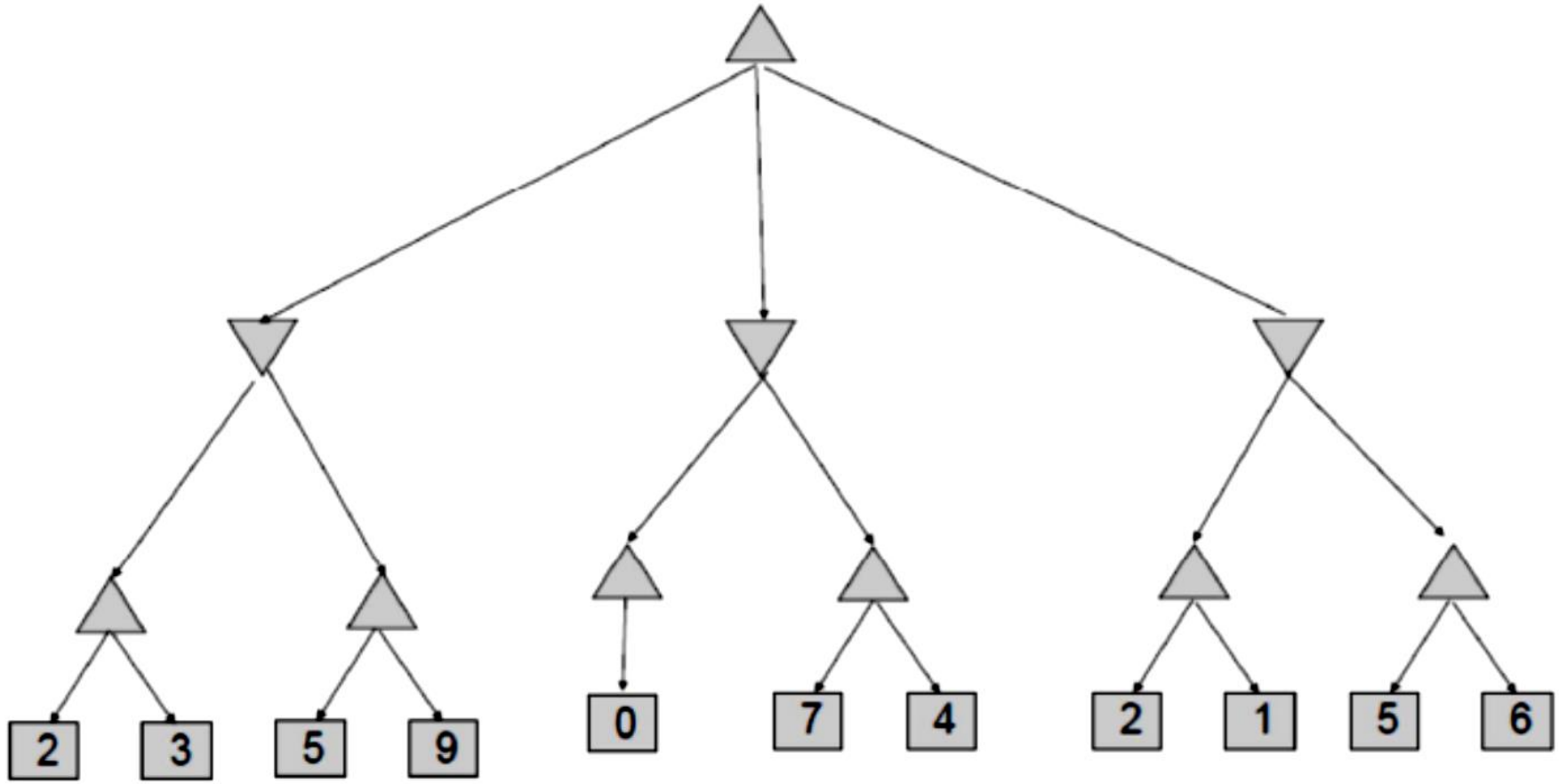

Starting a/b
a=-∞
b=+∞
a is α, b is β
3
a=-∞
b=+∞
a=3
b=+∞
a=3
b=+∞
a=3
b=+∞
3
≤2
≤1
a=-∞
b=+∞
a=-∞
b=3
a=-∞
b=3
a=-∞
b=3
a=3
b=+∞
a=3
b=2
a=3
b=+∞
a=3
b=14
a=3
b=5
a=3
b=1
3
12
2
14
5
1
≥8
a=-∞
b=3
8
a=8
b=3

Alpha-Beta Algorithm Sketch:

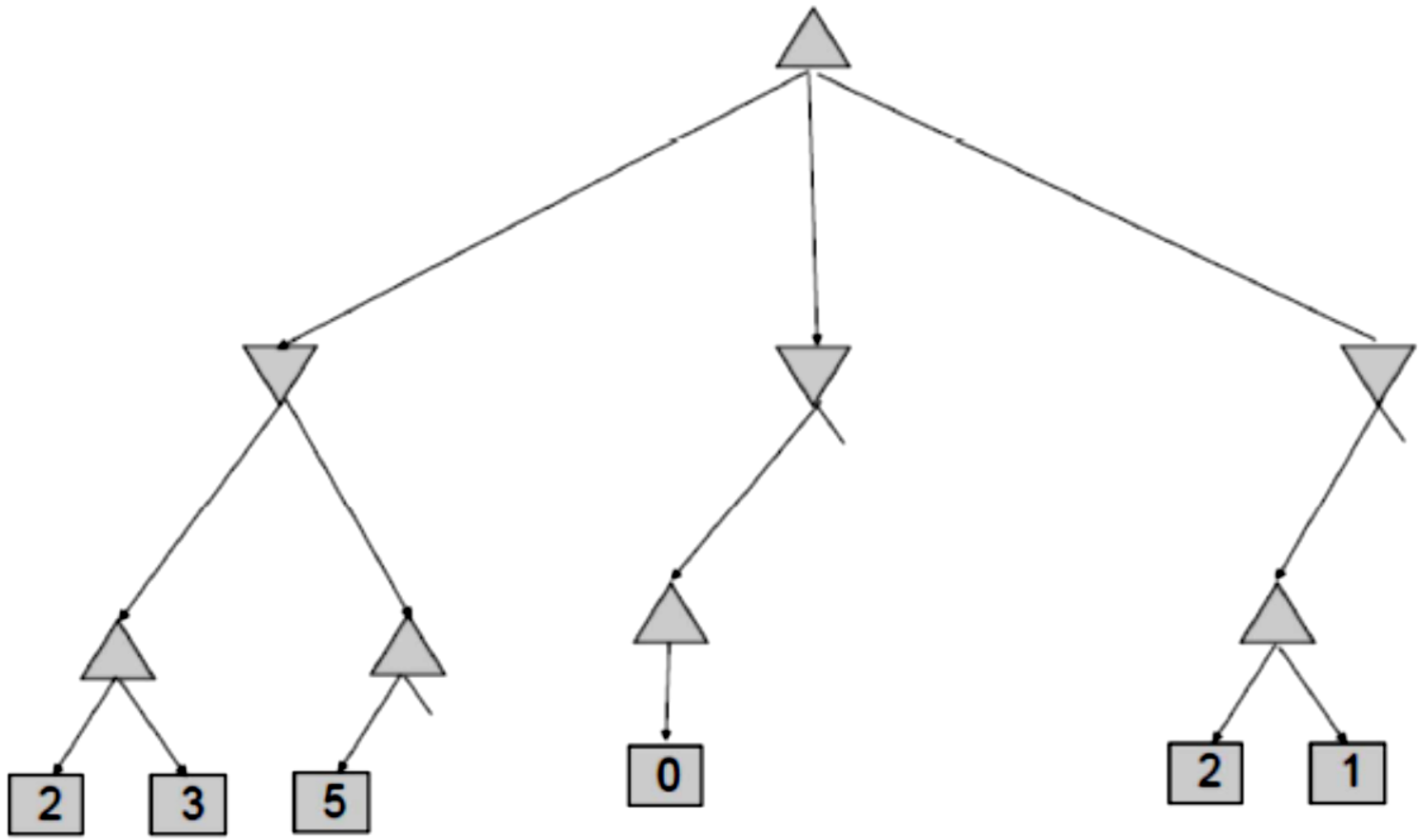
Algorithm 1 Max's turn to play

1: **procedure** MAXMOVE(S)
2: Assume MAX has to make a move from initial state S
3: $S.\alpha = -\infty$
4: $S.\beta = +\infty$
5:
6: **Exploration of Children from a Node:**
7: Explore child nodes of a node N (Initially $N = S$)
8: **if** $(\alpha < \beta)$ **then**
9: Pick the leftmost child C of N that remains to be explored
10: Propagate α and β values from N to child C
11: **end if**
13: **On Completing Exploration of Children:**
14: When you back up away from a node N to its parent P (when no children of N remains to be explored):
15: **if** N is MIN node **then**
16: send back up to P the updated β value
17: **end if**
18: **if** N is a MAX node **then**
19: send back up to P the updated α value
20: **end if**
22: **Updating α/β Values:**
23: Use the backed up value sent by N to update the α, β value of parent P
24: **if** P is MIN node **then**
25: $\beta_{new} = min\{\beta_{current}$, backed up value sent by $N\}$
26: $\alpha_{current}$ value in P does not change
27: **end if**
28: **if** P is MAX node **then**
29: $\alpha_{new} = max\{\alpha_{current}$, backed up value sent by $N\}$
30: $\beta_{current}$ value in P does not change
31: **end if**
32: **end procedure**

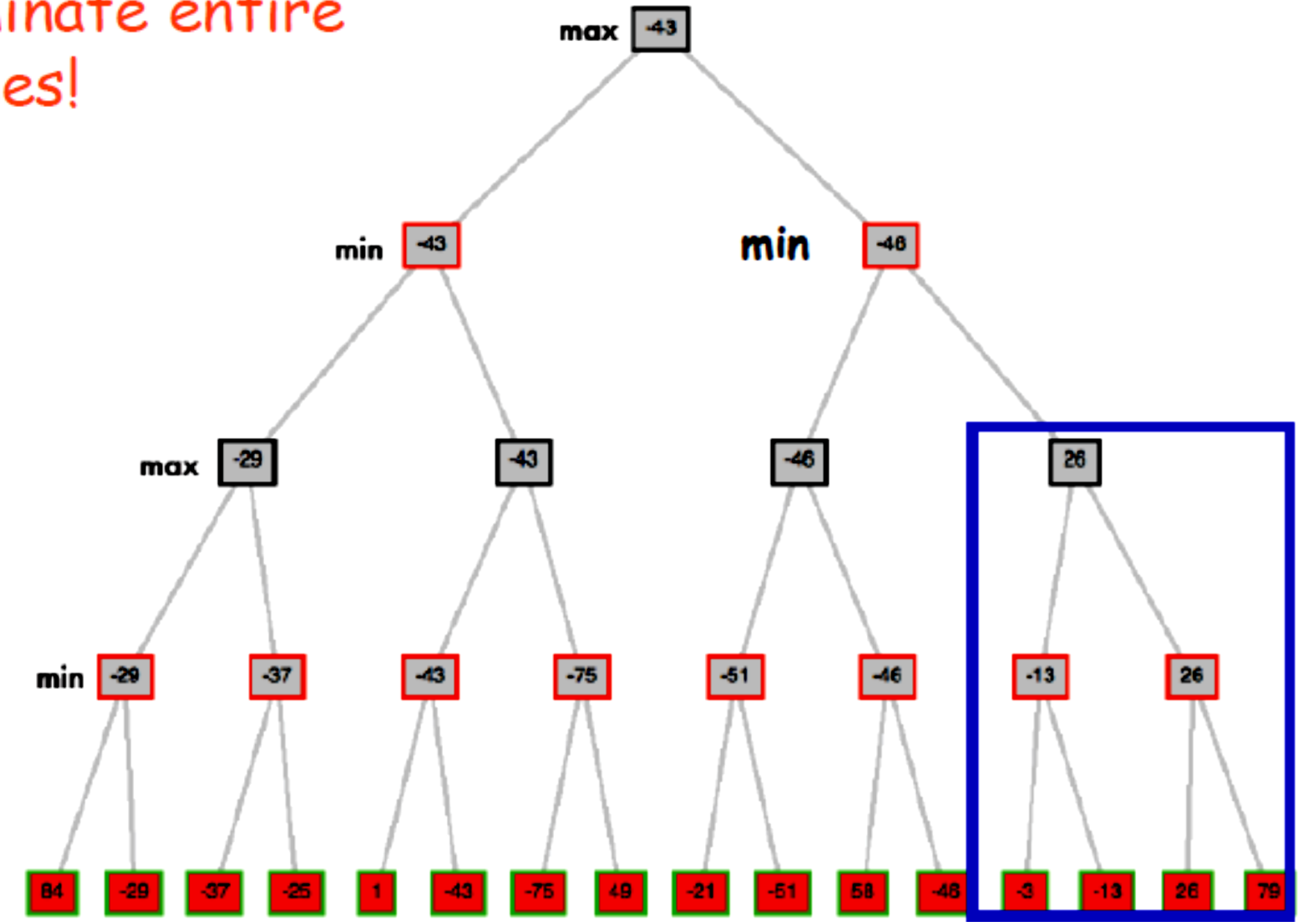
Alpha-Beta Pruning Example



Alpha-Beta Pruning Example



Pruning can eliminate entire subtrees!



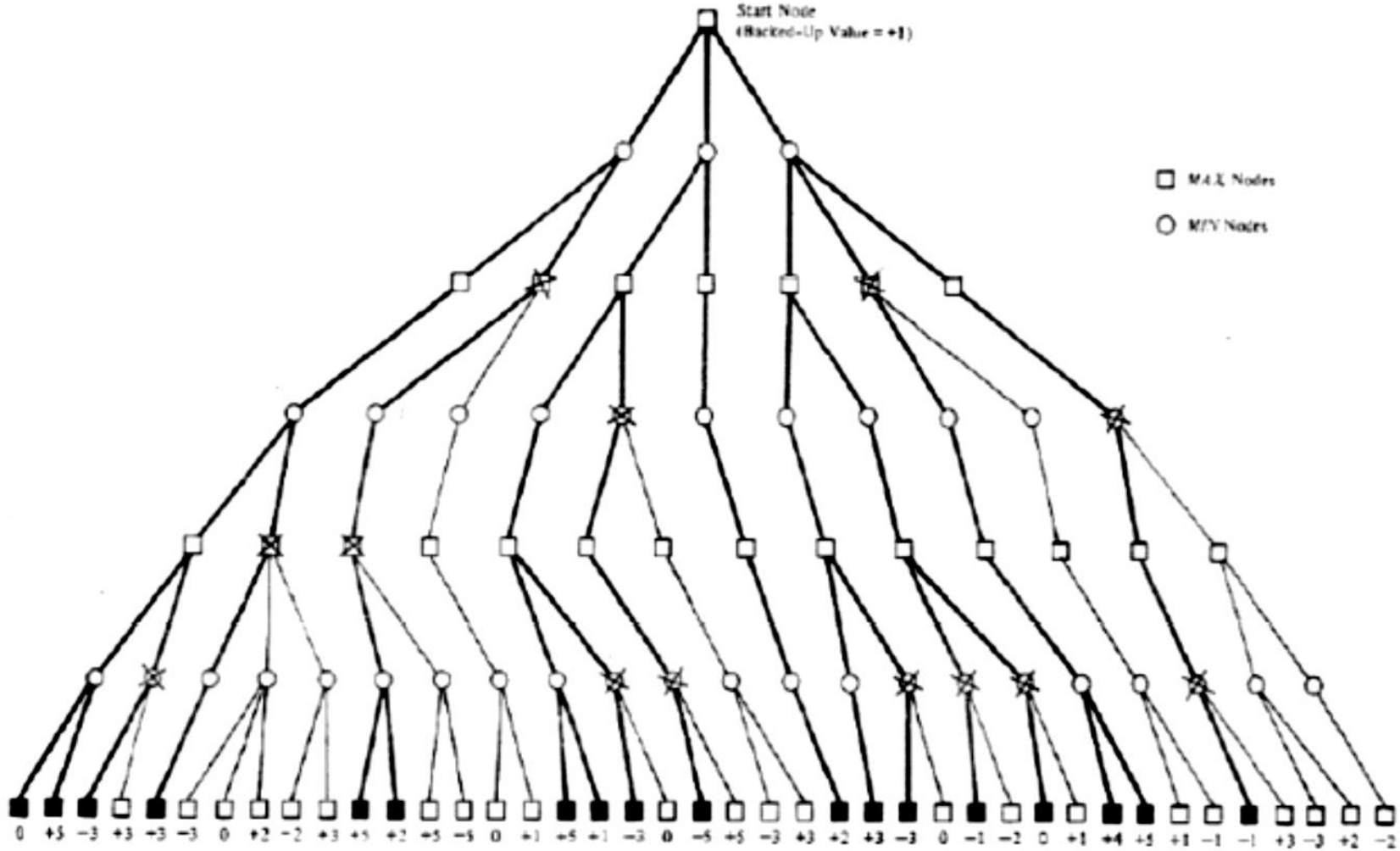


Fig. 3.12 An example illustrating the alpha-beta search procedure.

Alpha-Beta Pruning Properties

- This pruning has no effect on final result at the root
- Values of intermediate nodes might be wrong!
  - but, they are bounds
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!

Good enough?

- Chess:
  - branching factor $b \approx 35$
  - game length $m \approx 100$
  - α-β search space $b^{m/2} \approx 35^{50} \approx 10^{77}$

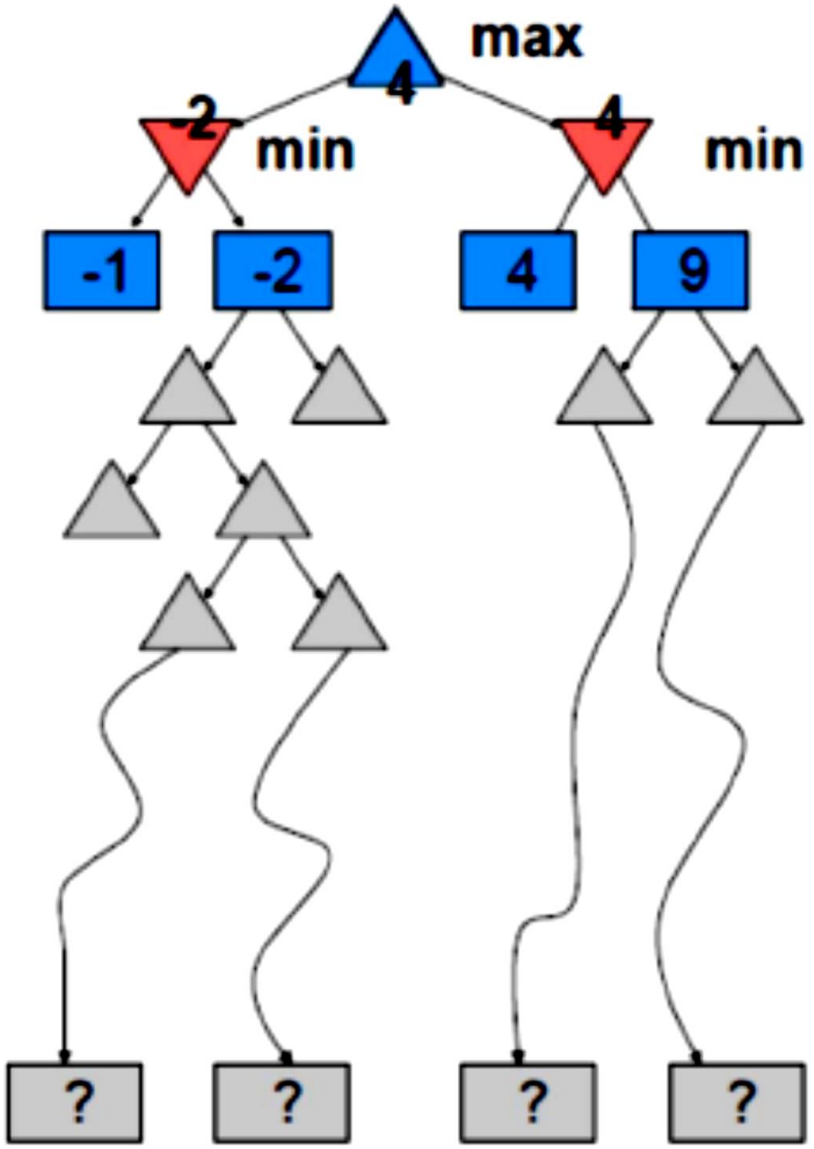
- The Universe:
  - number of atoms $\approx 10^{78}$
  - age $\approx 10^{21}$ milliseconds

- **Chess:**
  - branching factor b ≈ 35
  - game length m ≈ 100
  - search space $b^m \approx 35^{100} \approx 10^{154}$

Assuming a modern computer can process 1000000 board positions a second it will take 10^{140} years to search the entire tree.

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions

Note: can't guarantee optimality of move



Partial Search Tree

- In a real game, we can only look ahead a few ply!
- The depth of search is determined by the time allowed per move.
- Suppose we can process 1000000 positions a second and we're allowed one minute per move, then we can search 5 ply.

For chess with b=35

Heuristic Evaluation Functions

- Motivation: When search space is too large, create game tree up to a certain depth only.
- Art is to estimate utilities of positions that are not terminal states.

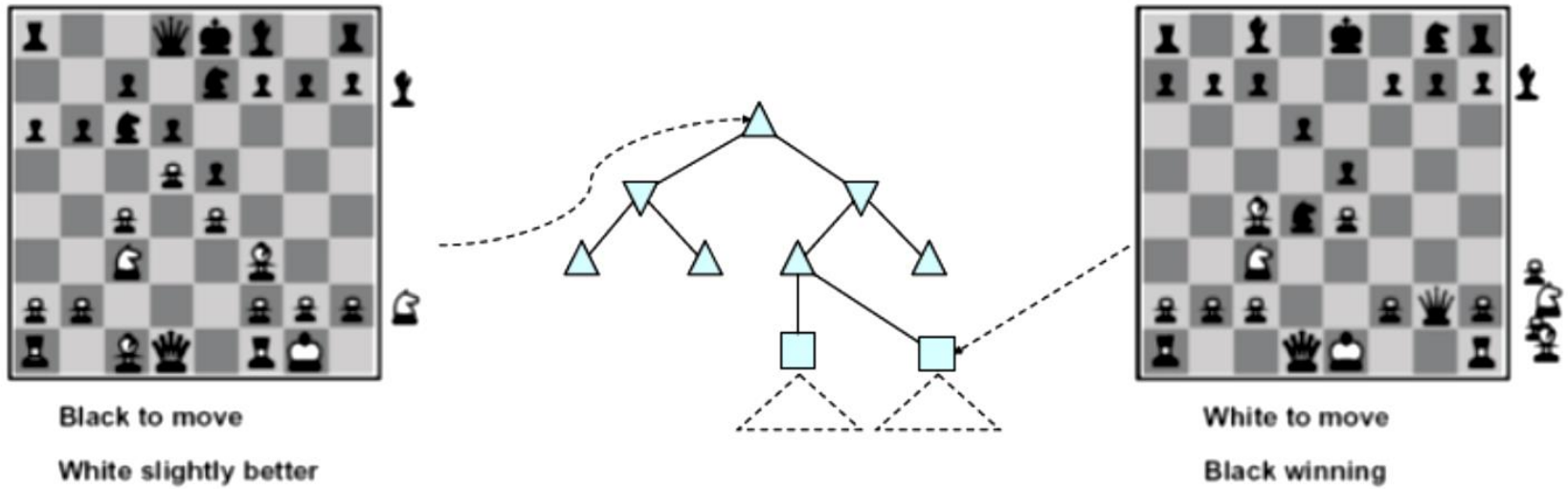
- An Evaluation Function:
  - Estimates how good the current board configuration is for a player.
  - Typically, evaluate how good it is for the player, how good it is for the opponent, then subtract the opponent's score from the player's.
  - Preserve the relative ordering of the terminal states utilities.
  - 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
- "zero-sum game"

Evaluation Function

- Function which scores non-terminals



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

- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:
  - e.g. $f_1(s)$ = (num white queens – num black queens), etc.

"material" evaluation function:

9*(#WQ - #BQ) + 5*(#WR - #BR) + 3*(#WB - #BB) + 3*(#WK - #BK) + (#WP - #BP)

To construct this type of function, first pick the features, and then adjust the weights until the program plays well

You can adjust the weights automatically by having the program play lots of games against itself ("reinforcement learning")

Minimax Cutoff

- Does it work in practice?
  - Time complexity: $O(b^m)$
- Chess:
  - $b = 35$
  - Suppose we limit our search to 1.5 million nodes per move
  - $m = 4$
  - 4-ply chess player is a lousy player!
  - 4-ply = novice chess player
  - 8-ply = typical PC, human master
  - 12-ply = Deep Blue, Kasparov

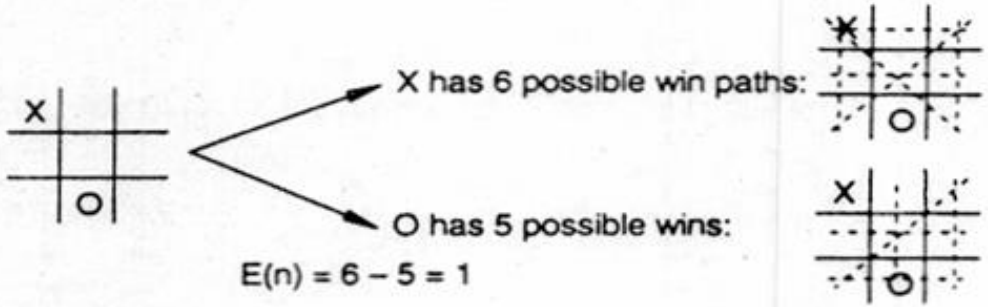
Revised Minimax Algorithm

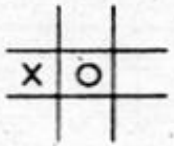
For the MAX player

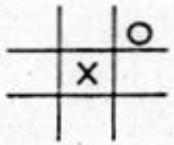
1. Generate the game as deep as time permits
2. Apply the evaluation function to the leaf states
3. Back-up values
   - At MIN ply assign minimum payoff move
   - At MAX ply assign maximum payoff move
4. At root, MAX chooses the operator that led to the highest payoff

Revised Minimax: Tic-Tac-Toe

- The static evaluation function heuristic







Heuristic is E(n) = M(n) – O(n)

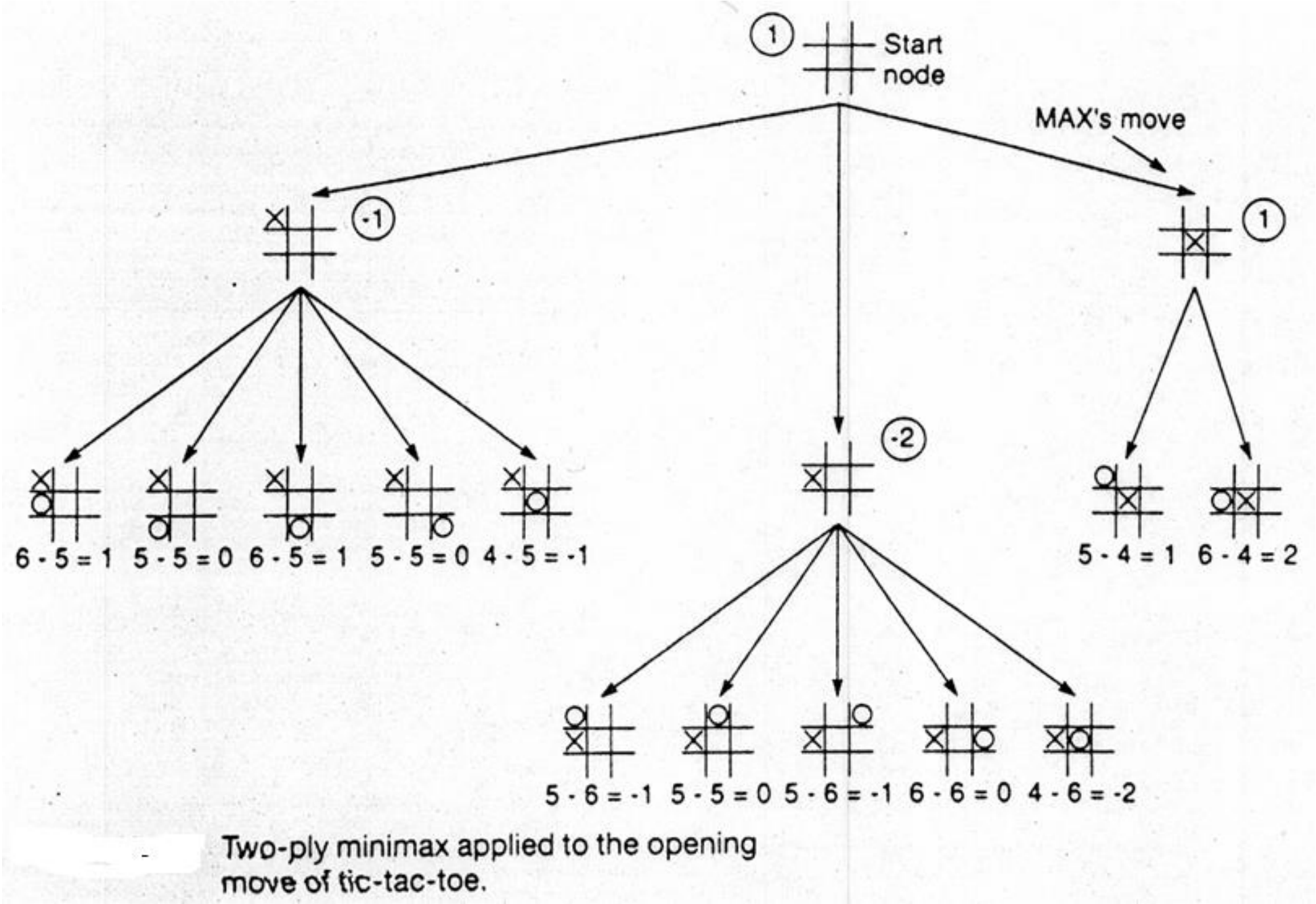
where M(n) is the total of My possible winning lines

O(n) is total of Opponent's possible winning lines

E(n) is the total Evaluation for state n

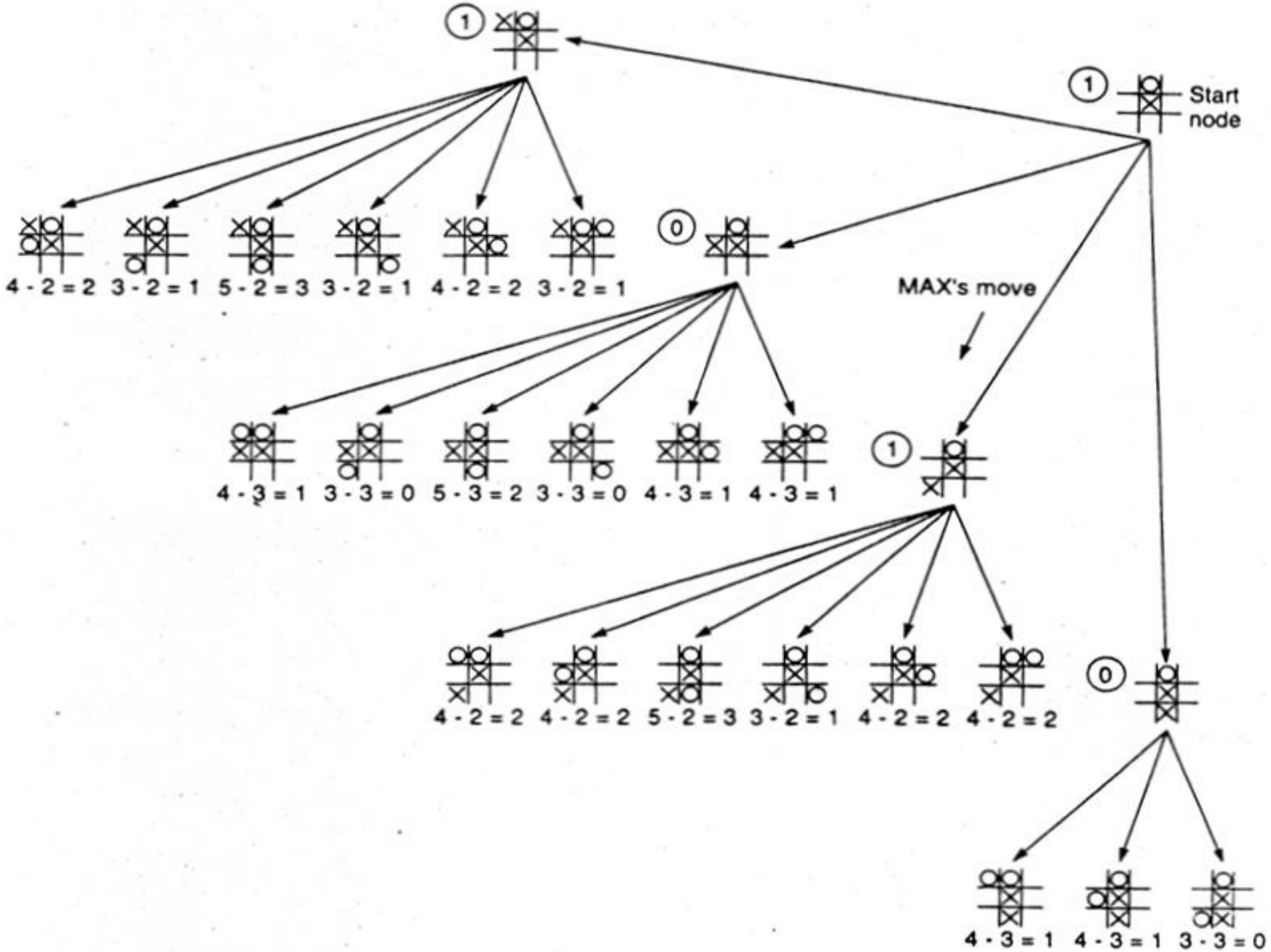
Heuristic measuring conflict applied to states of tic-tac-toe.

1

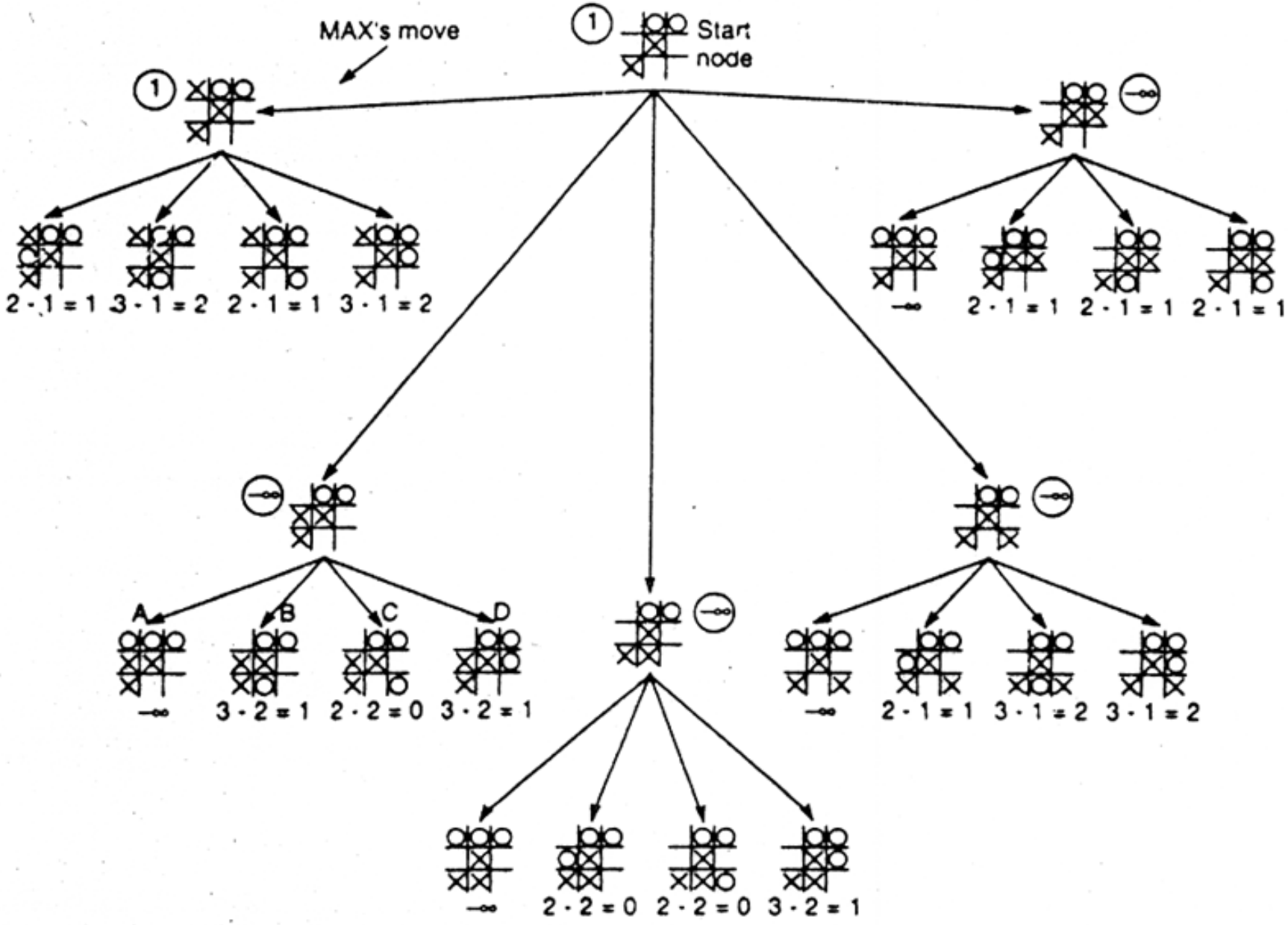


Two-ply minimax applied to the opening move of tic-tac-toe.

2



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



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

Miscellaneous Optimizations

- Transposition Tables
  - Games Trees have repeated States since
  - Different permutations of moves can result in previously seen games
  - Maintain a Transposition Table to record such States (akin to Explored Sets in Search) and do table lookup for previously seen games
- End Game/Opening Databases
  - Precompute choices for smaller games and store them
  - Gains are substantial – Deep Blue uses them
- Iterative Deepening
  - is frequently used with Alpha-Beta so that searches to successively deeper plies can be attempted if there is time, and the move selected is the one computed by the deepest search completed when the time limit is reached.

Game Playing – State of the Art

- **Checkers:** Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions. Checkers is now solved!
- **Chess:** Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Othello:** Human champions refuse to compete against computers, which are too good.
- **Go:** Human champions are beginning to be challenged by machines,

Google's machine has won GO – the last frontier

Modeling the Opponent

- So far assumed

  Opponent = rational, optimal (always picks MIN values)

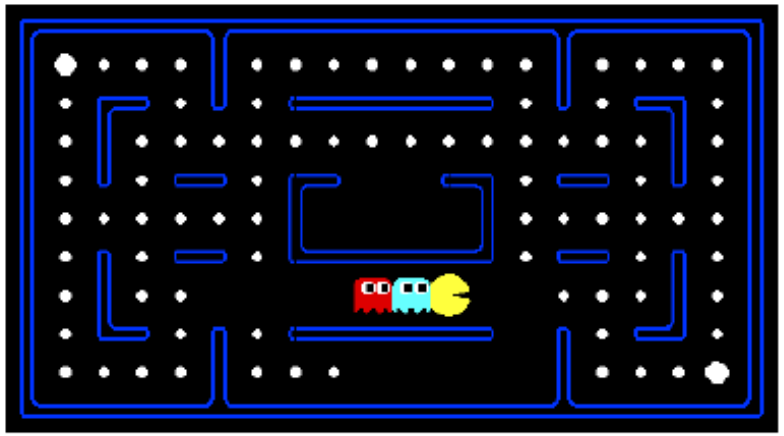
- What if

  Opponent = random? (picks action randomly)

  2 player w/ random opponent = 1 player stochastic

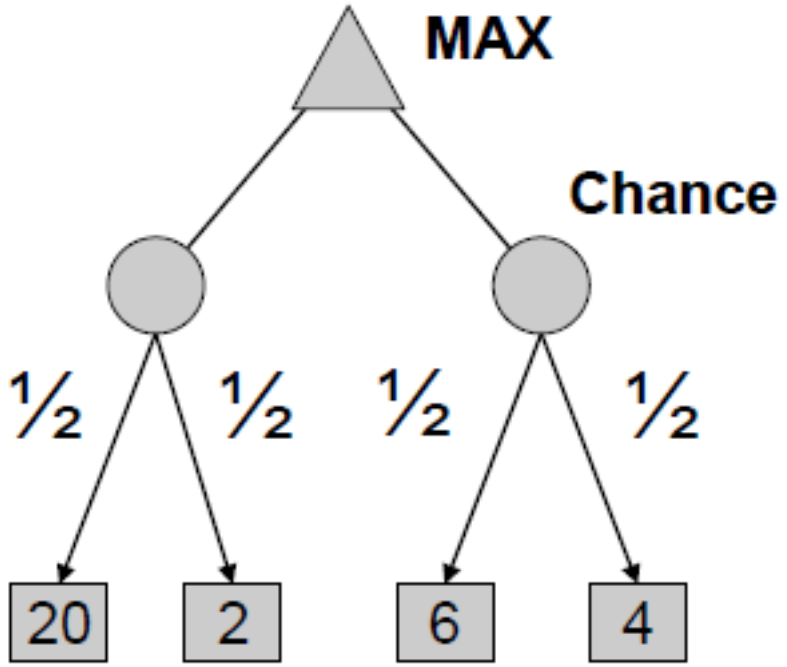
Stochastic Single Player

- Don’t know what the result of an action will be. E.g.,
  - In backgammon, don’t know result of dice throw; In solitaire, card shuffle is unknown; in minesweeper, mine locations are unknown
  - In Pac-Man, suppose the ghosts behave randomly



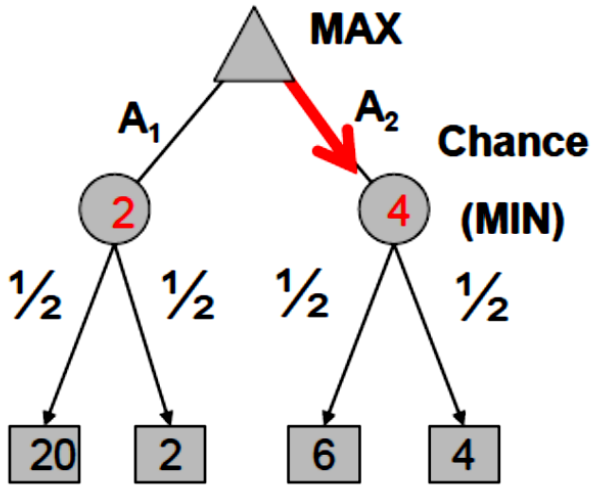
Adversarial Search: Expectimax and Expectiminimax

- Game Tree has
  - MAX nodes as before
  - **Chance** nodes



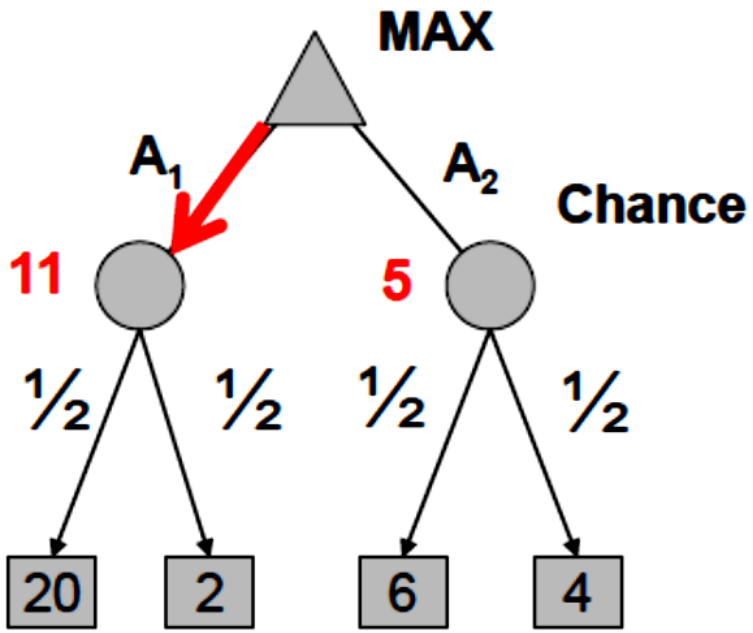
Minimax with Chance Nodes

- Suppose you pick MIN value move at each chance node
- Which move (action) would MAX choose?
- MAX would always choose A_2
  - Average utility = 5
- If MAX had chosen A_1
  - Average utility = 11



Expectimax Search

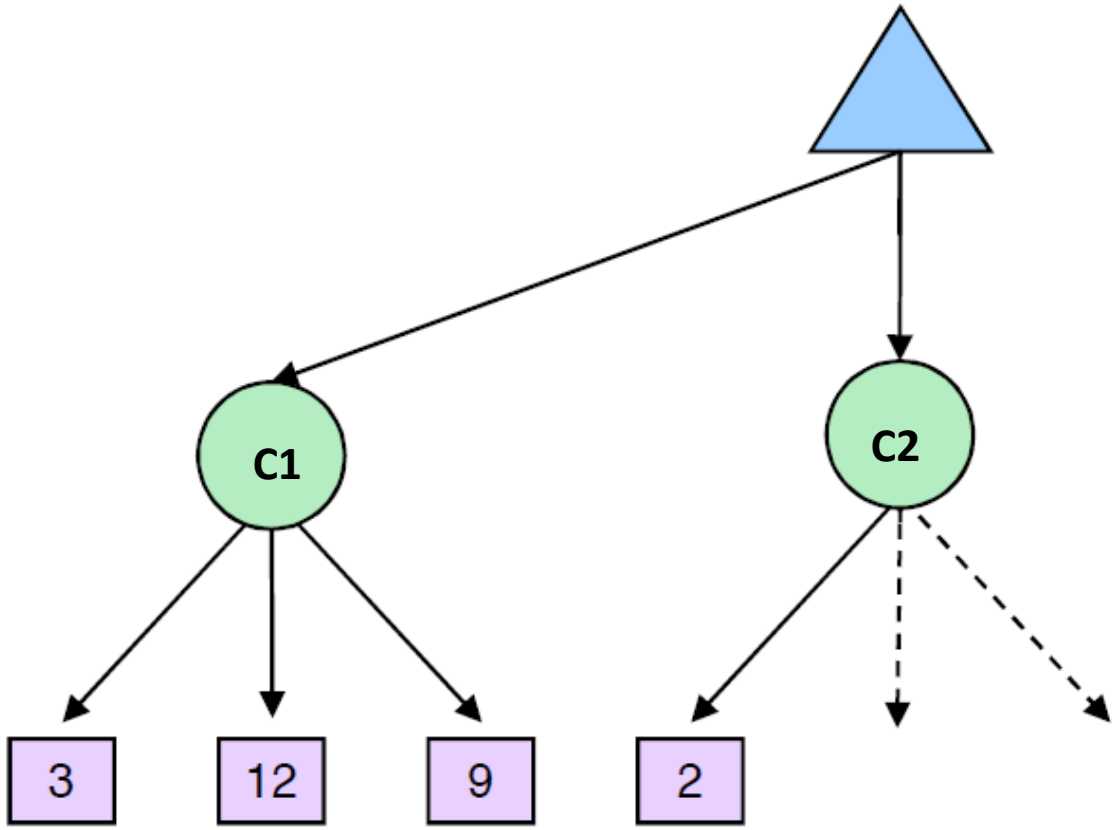
- **Expectimax search:**
  Chance nodes take average (expectation) of value of children
- MAX picks move with *maximum expected value*



Expectimax Search Sketch

if *state* is a MAX node **then**
    return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
if *state* is a MIN node **then**
    return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
if *state* is a chance node **then**
    return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)

Expectimax Pruning



Assuming Uniform probability

Expected value @ C1 = 1/3*(3+12+9)=8

Can we prune the 2 leaf nodes of C2
to the right of leaf node with utility value 2 ?

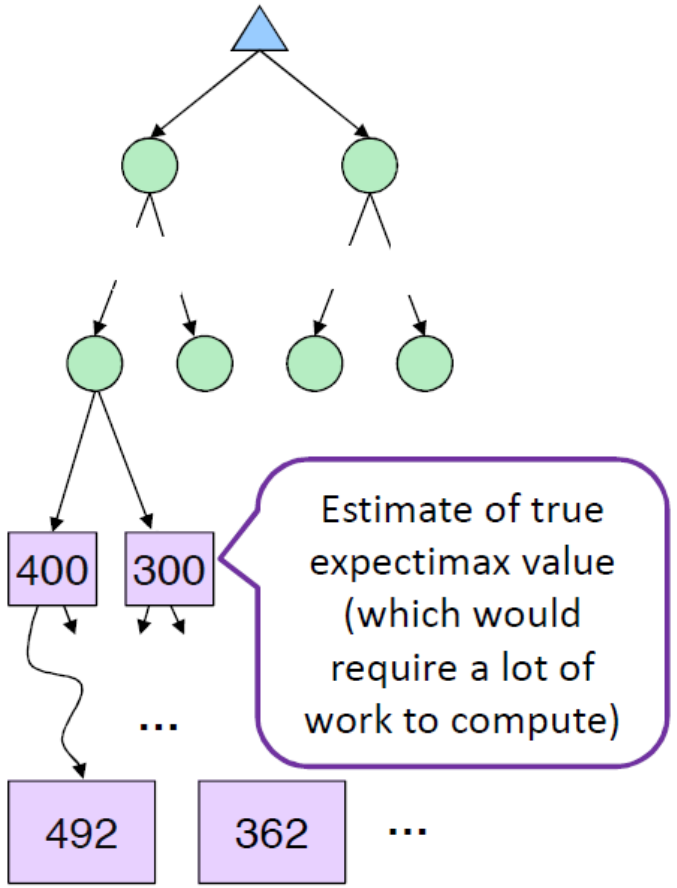
Standard α-β will prune them

But these are expected values.

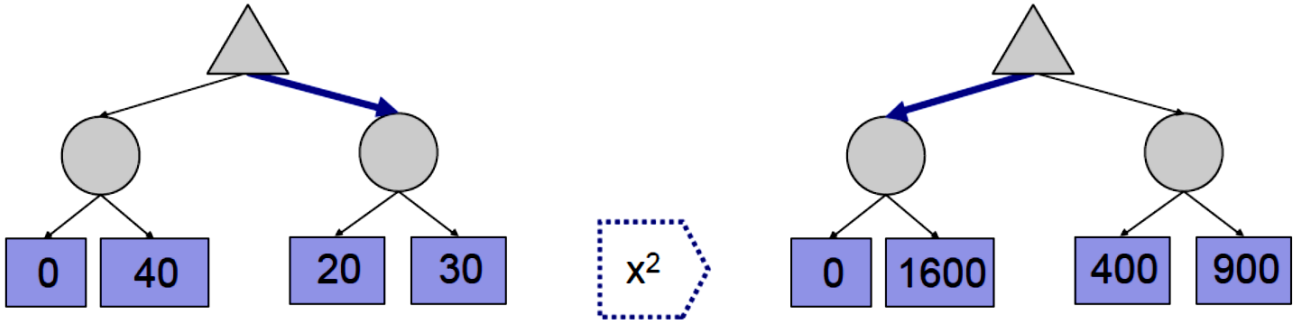
What if the 2nd sibling node of 2 has value say 100?

Not Easy – Need Bounds on Exact Values

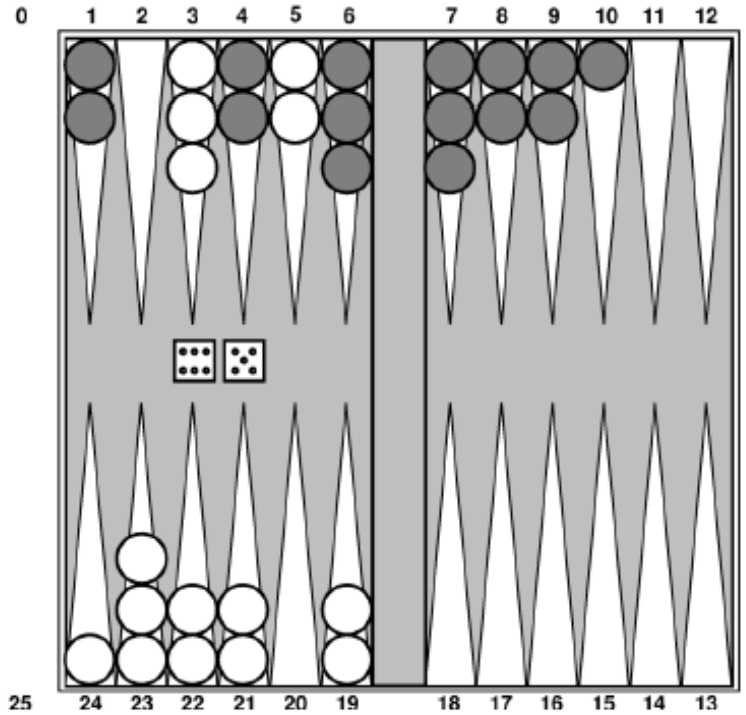
Depth-limited Expectimax



But Evaluation functions need to be designed carefully



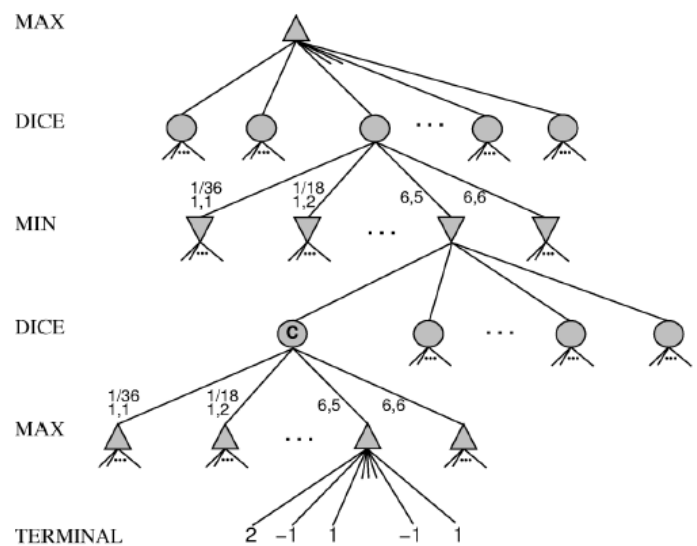
Extending Expectimax to Stochastic Two Player Games



White has just rolled 6-5 and has 4 legal moves.

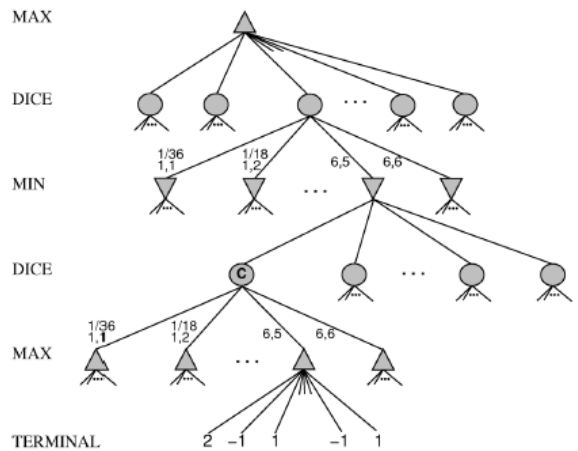
Expectiminimax Search

- In addition to MIN- and MAX nodes, we have chance nodes (e.g., for rolling dice)
- Chance nodes take expectations, otherwise like minimax



Expectiminimax Search

if *state* is a MAX node **then**
    return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
if *state* is a MIN node **then**
    return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
if *state* is a chance node **then**
    return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)



Search costs increase: Instead of $O(b^d)$, we get $O((bn)^d)$, where n is the number of chance outcomes

Adversarial Search: Summary

- Basic idea: Minimax
- Too slow for most games
- Alpha-Beta pruning can increase max depth by factor up to 2
- Limited depth search necessary for most games
- Static evaluation functions necessary for limited depth search; opening game and end game databases can help
- Computers have beaten humans in all known board games: checkers, chess, Othello and most recently: Go
- Expectimax and Expectiminimax allow search in stochastic games