Artificial Intelligence
ENCS 434

Adversarial Search & Games

Aziz M. Qaroush - Birzeit University

Game Playing and AI

☞ **Why would game playing be a good problem for AI research?**

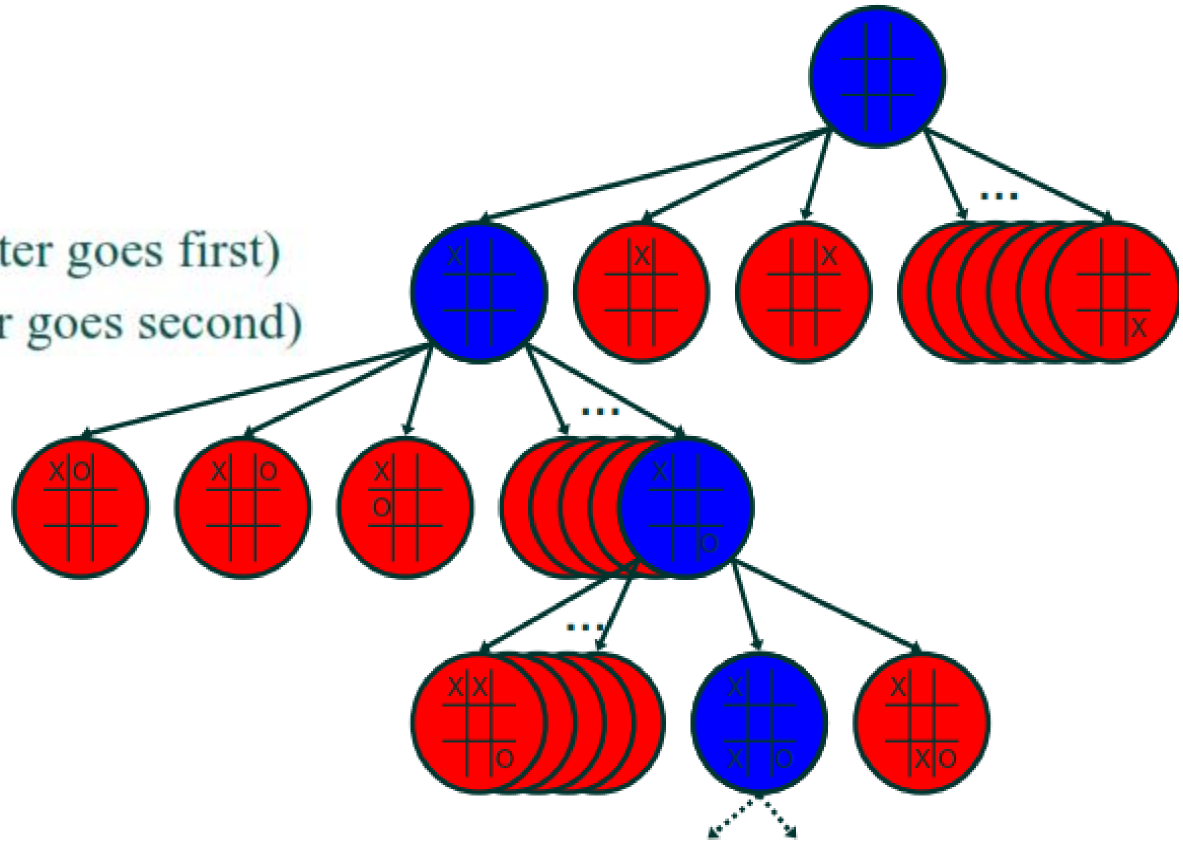
- Game-playing is non-trivial
  - Need to display “human-like” intelligence
  - Some games (such as chess) are very complex
  - Requires decision-making within a time-limit
    - More realistic than other search problems
- Games are played in a controlled environment
  - Can do experiments, repeat games, etc
  - Good for evaluating research systems
- Can compare humans and computers directly
  - Can evaluate percentage of wins/losses to quantify performance
- All the information is available
  - Human and computer have equal information

How Does a Computer Play a Game?

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

Tic-Tac-Toe Game

- Tic-Tac-Toe
  - b ~ 5 legal moves,
  - d ~ total of 9 moves
  - $5^9 = 1{,}953{,}125$
  - 9! = 362,880 (Computer goes first)
  - 8! = 40,320 (Computer goes second)



Game Playing: Adversarial Search

❑ Introduction

- Different kinds of games:

	Deterministic	Chance
Perfect Information	Chess, Checkers Go, Othello	Backgammon, Monopoly
Imperfect Information	Battleship	Bridge, Poker, Scrabble,

- Games with perfect information. No randomness is involved.

- Games with imperfect information. Random factors are part of the game.

Games as Adversarial Search

- many games can be formulated as search problems

- the zero-sum utility function leads to an adversarial situation
  - in order for one agent to win, the other necessarily has to lose

- factors complicating the search task
  - potentially huge search spaces
  - elements of chance
  - multi-person games, teams
  - time limits
  - imprecise rules

Difficulties with Games

- games can be very hard search problems
  - yet reasonably easy to formalize
  - finding the *optimal* solution may be impractical
    - a solution that beats the opponent is "good enough"
  - unforgiving
    - a solution that is "not good enough" not only leads to higher costs, but to a loss to the opponent

- example: chess
  - size of the search space
    - branching factor around 35
    - about 50 moves per player
    - about 35^{100} or 10^{154} nodes
      - about 10^{40} *distinct* nodes (size of the search graph)

Single-Person Game

- conventional search problem
  - identify a sequence of moves that leads to a winning state
  - examples: Solitaire, dragons and dungeons, Rubik’s cube
  - little attention in AI

- some games can be quite challenging
  - some versions of solitaire
  - a heuristic for Rubik’s cube was found by the Absolver program

Searching in a two player game

- Traditional (single agent) search methods only consider how close the agent is to the goal state (e.g. best first search).
- In two player games, decisions of both agents have to be taken into account: a decision made by one agent will affect the resulting search space that the other agent would need to explore.
- Question: Do we have randomness here since the decision made by the opponent is NOT known in advance?
- ☺ No. Not if *all* the moves or choices that the opponent can make are finite and can be known in advance.

Searching in a two player game

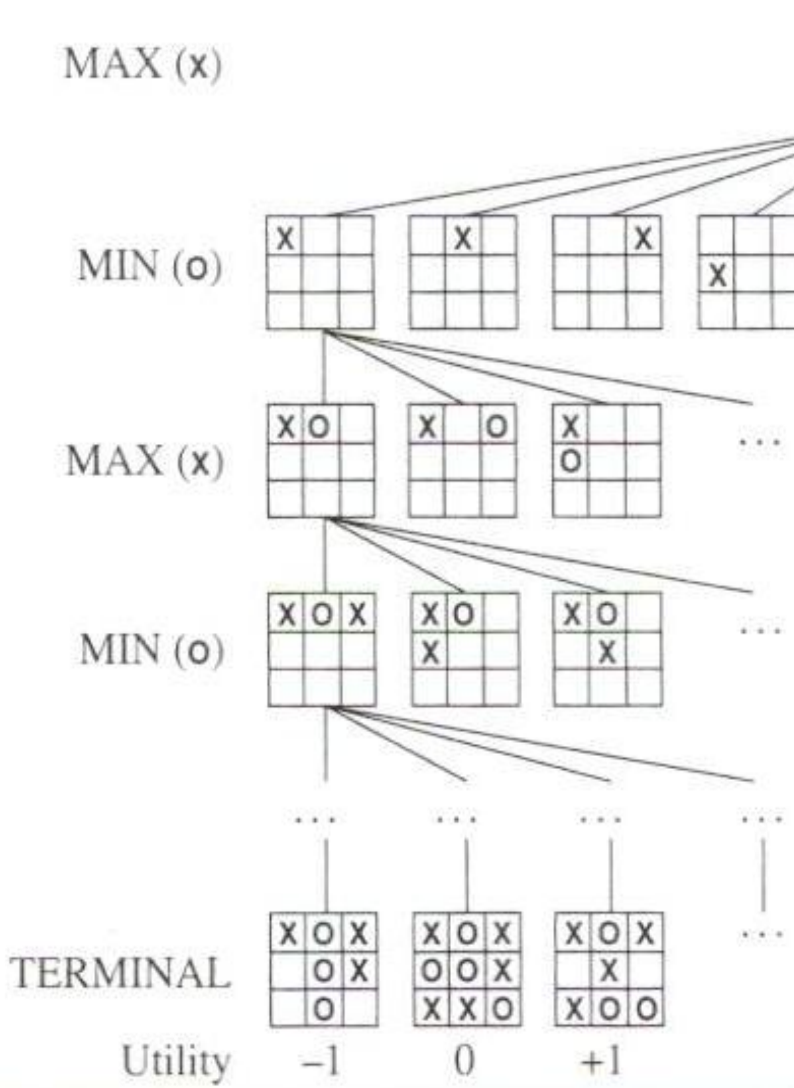
To formalize a two player game as a search problem an agent can be called **MAX** and the opponent can be called **MIN.**

Problem Formulation:

- **Initial state:** board configurations and the player to move.
- **Successor function:** list of pairs (move, state) specifying legal moves and their resulting states. (moves + initial state = game tree)
- **A terminal test:** decide if the game has finished.
- **A utility function:** produces a numerical value for (only) the terminal states. Example: In chess, outcome = win/loss/draw, with values +1, -1, 0 respectively.

- Players need search tree to determine next move.

Partial game tree for Tic-Tac-Toe

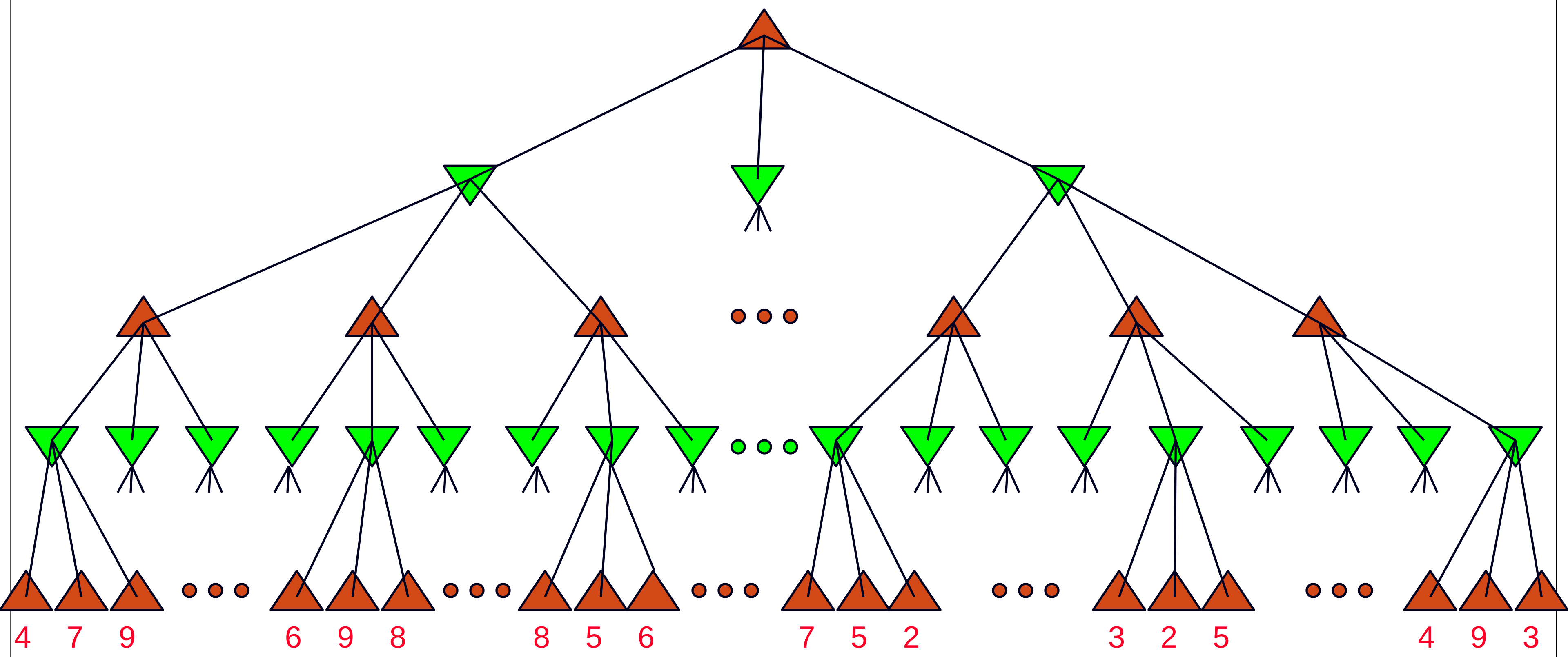


- Root node represents the current board configuration; player must decide the best single move to make next
- Each level of search nodes in the tree corresponds to all possible board configurations for a particular player MAX or MIN.
- If it is my turn to move, then the root is labeled a "MAX" node; otherwise it is labeled a "MIN" node, indicating opponent's turn.
- Utility values found at the end can be returned back to their parent nodes.
- **Idea**: MAX chooses the board with the max utility value, MIN the minimum.

MiniMax Algorithm

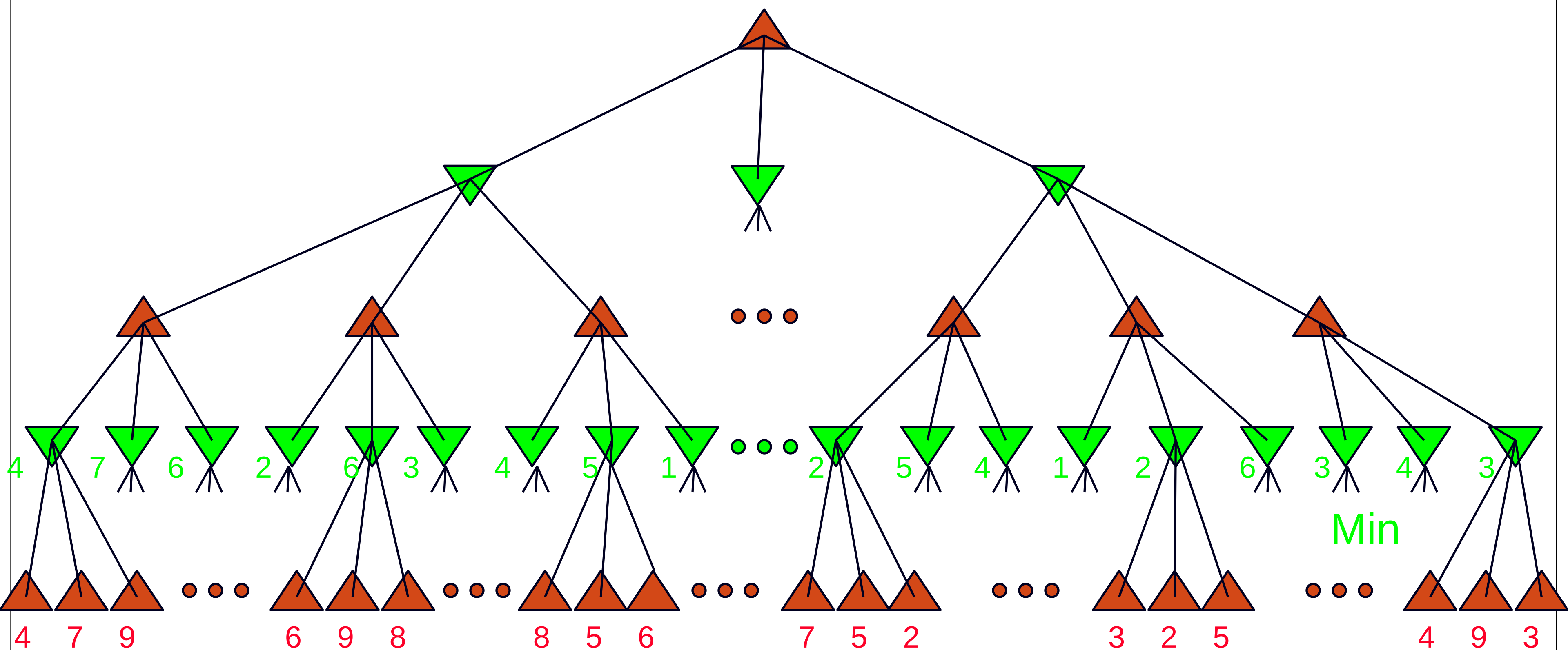
- Create start node as a MAX node with current board configuration
- Expand nodes down to some depth of lookahead in the game
- Apply the evaluation function at each of the leaf nodes
- "Back up" values for each of the non-leaf nodes until a value is computed for the root node.
  - At MIN nodes, the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

MiniMax Example



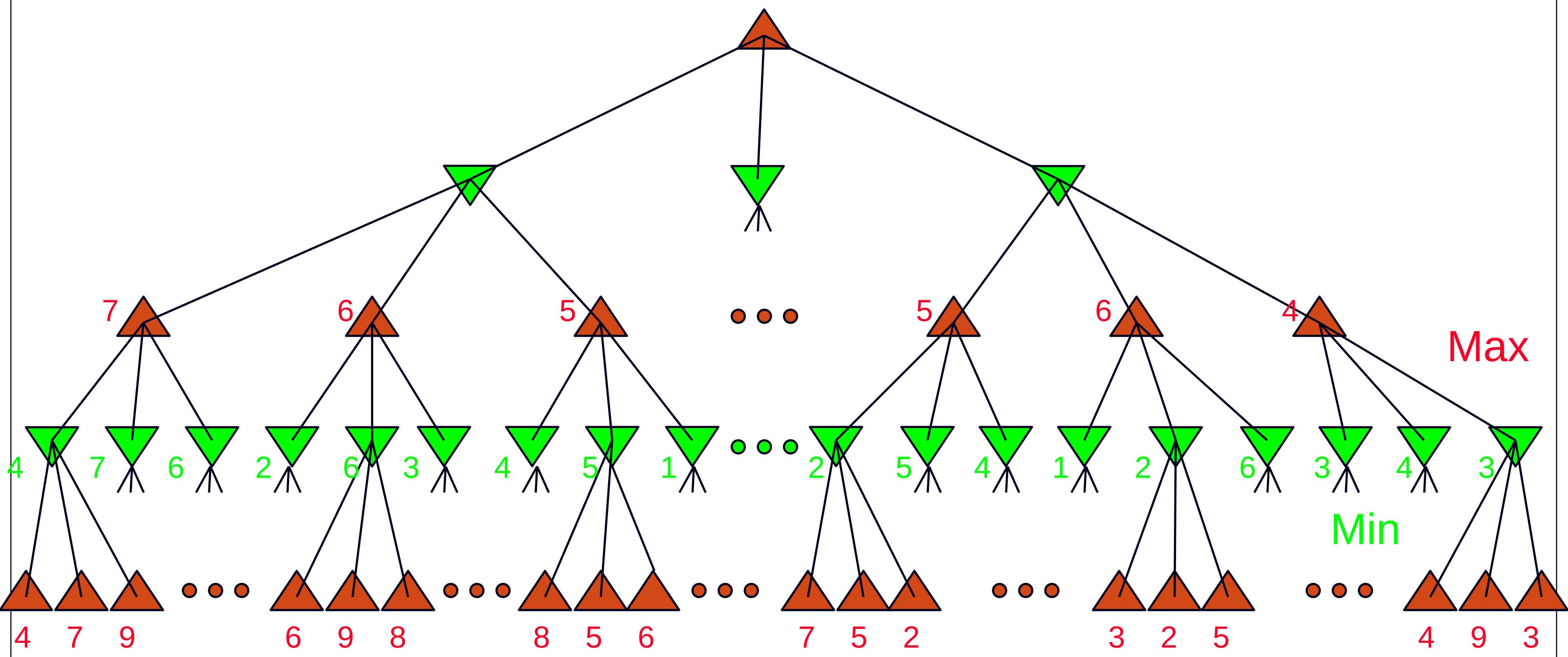
terminal nodes: values calculated from the utility function

MiniMax Example

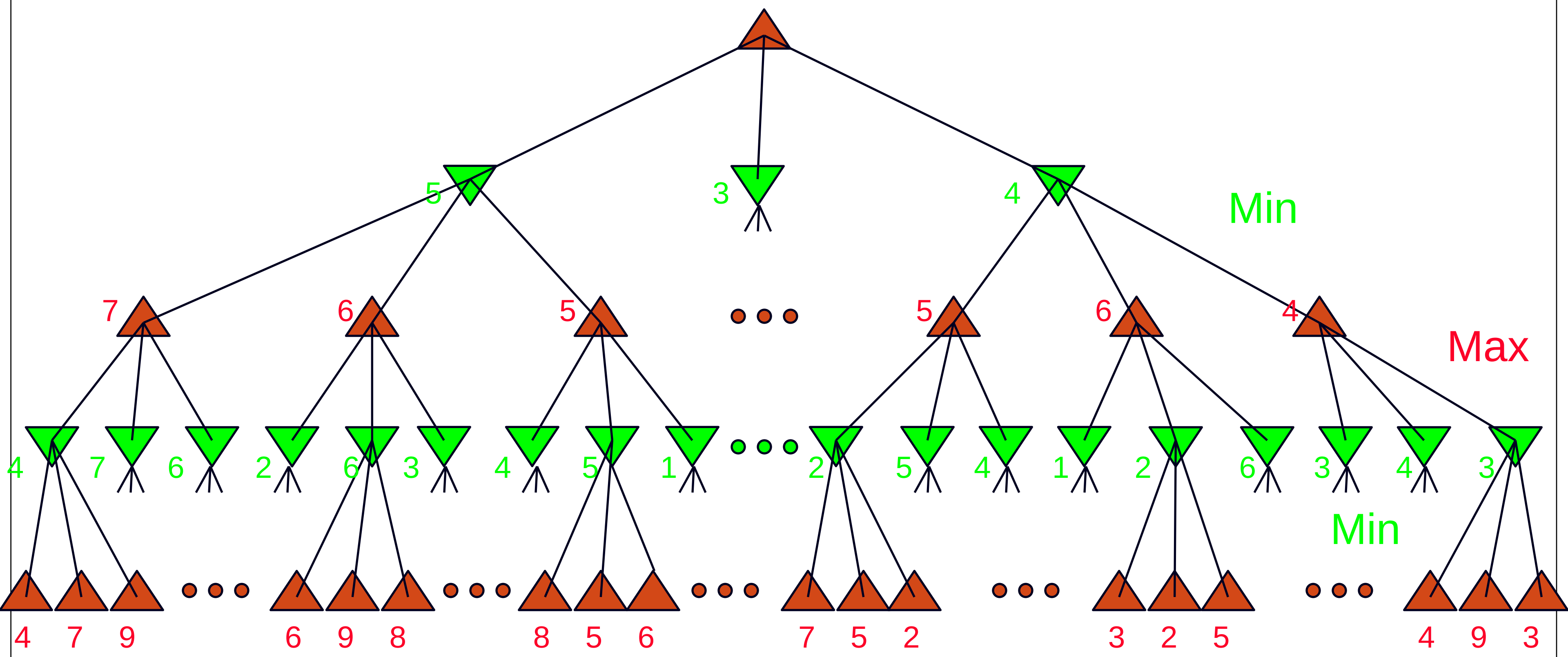


other nodes: values calculated via minimax algorithm

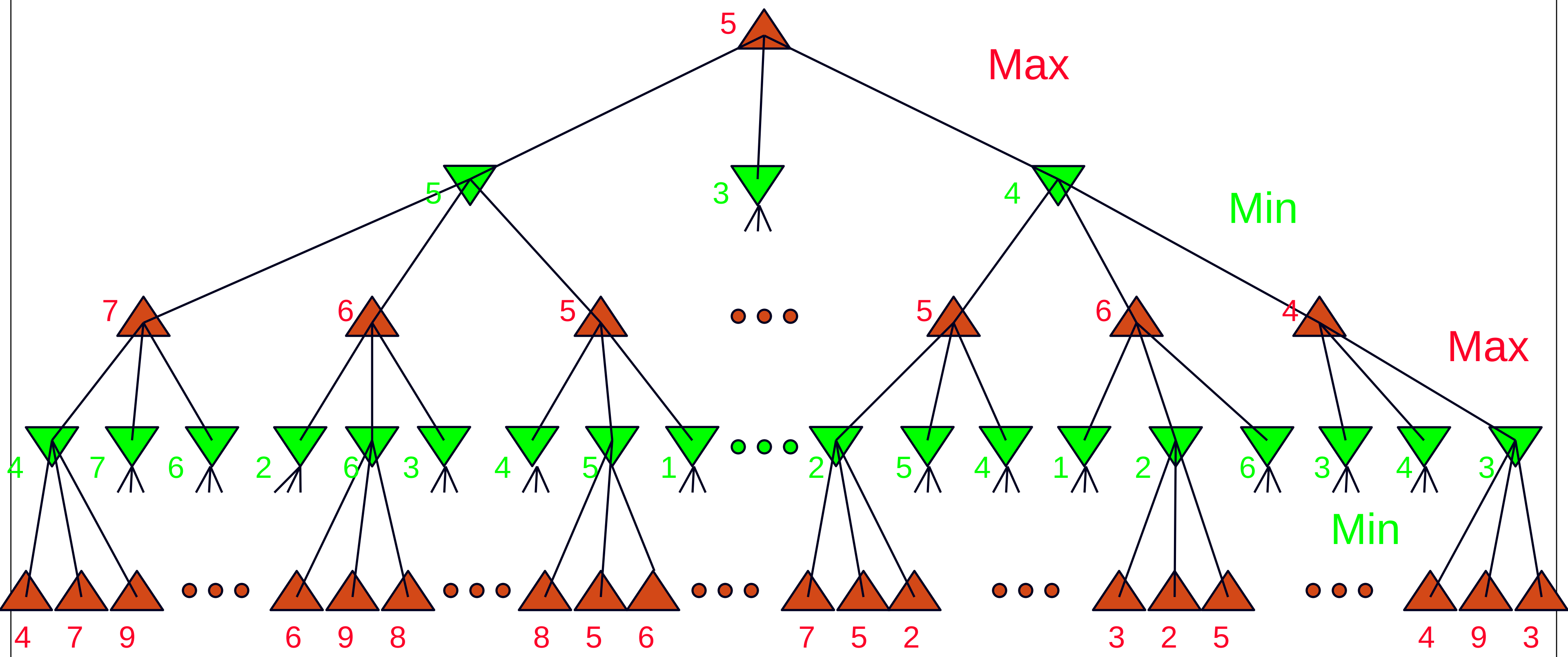
MiniMax Example



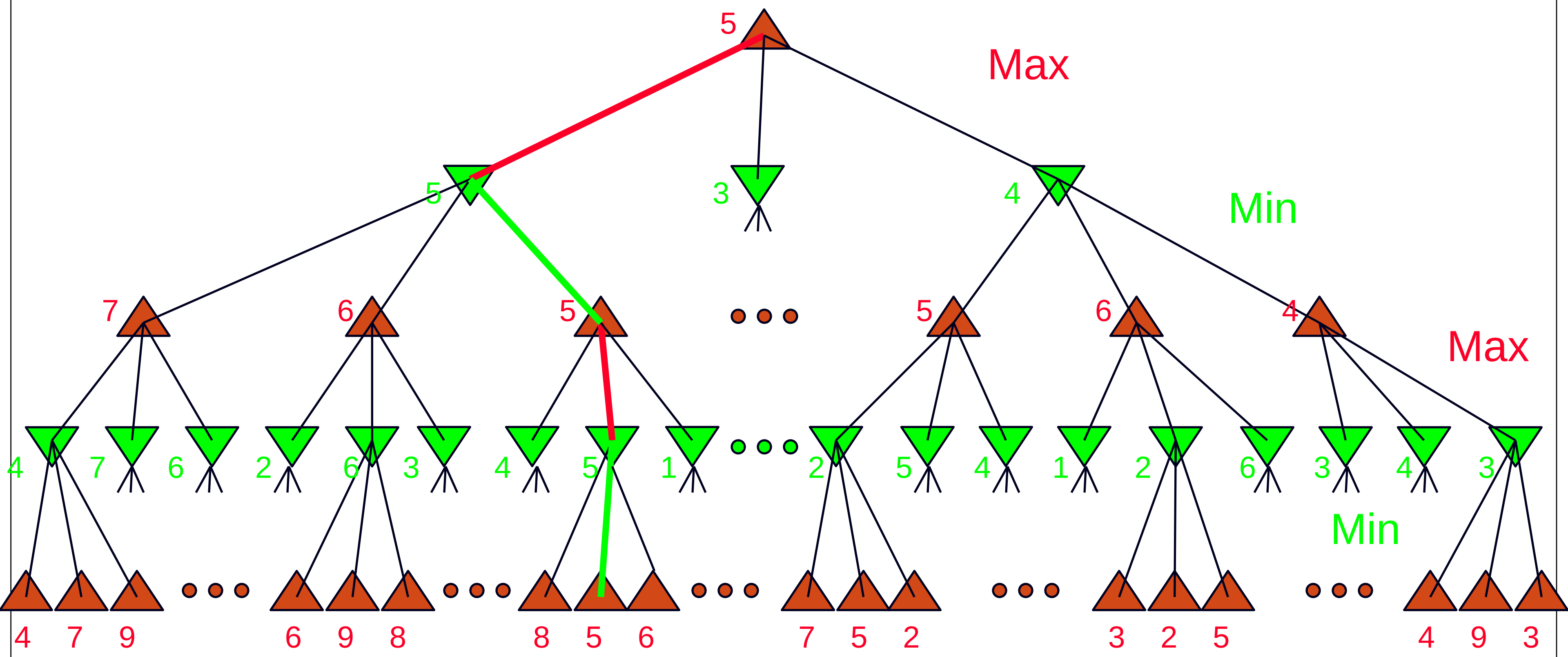
MiniMax Example



MiniMax Example



MiniMax Example



moves by Max and countermoves by Min

MiniMax Properties

Assume all terminal states are at depth d

☞ Space complexity?

Depth-first search, so O(bd)

☞ Time complexity?

Given branching factor b, so $O(b^d)$

✱ Time complexity is a major problem!

Computer typically only has a finite amount of time to make a move.

❑ Direct mini-max also is impractical in practice

✱ Static Board Evaluator (SBE) function

Uses heuristics to estimate the value of non-terminal states.

Pruning

- **Discards parts of the search tree**
  - Guaranteed not to contain good moves
  - Guarantee that the solution is not in that branch or sub-tree
    - If both players make optimal decisions, they will never end up in that part of the search tree
- **Use pruning to ignore those branches.**
- **Certain moves are not considered**
  - Won't result in a better evaluation value than a move further up in the tree
  - They would lead to a less desirable outcome
- **Applies to moves by both players**
  - α **(alpha)** indicates the best choice for Max so far never decreases
    - Highest Evaluation value seen so far (initialize to -infinity)
  - β (**beta**) indicates the best choice for Min so far never increases
    - Lowest Evaluation value seen so far (initialize to +infinity)

Alpha-Beta Pruning

- Beta cutoff pruning occurs when maximizing

  if *child's alpha >= parent's beta*

  Stop expanding children. Why?

  - Opponent won't allow computer to take this move

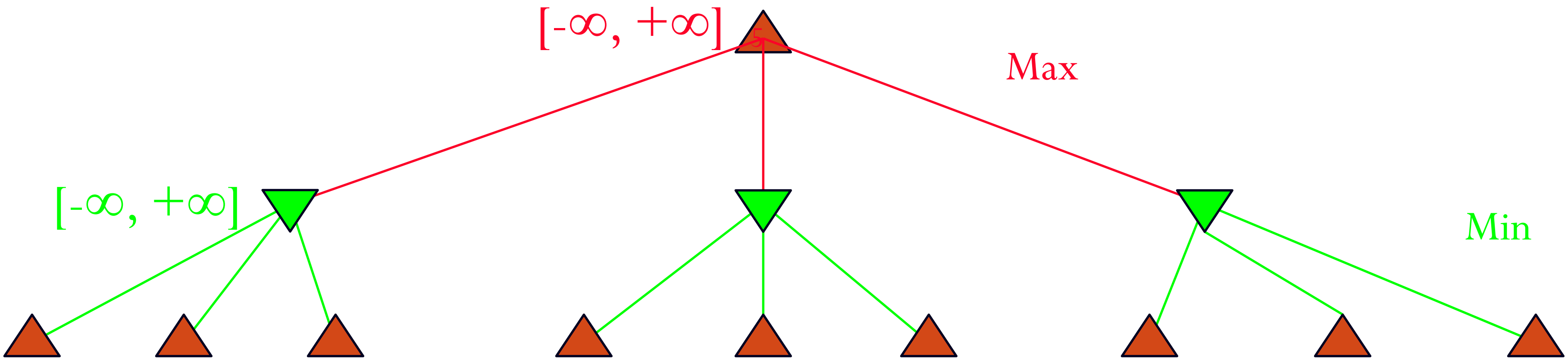
- Alpha cutoff pruning occurs when minimizing

  if *parent's alpha >= child's beta*

  Stop expanding children. Why?

  - Computer has a better move than this

Alpha-Beta Example 1

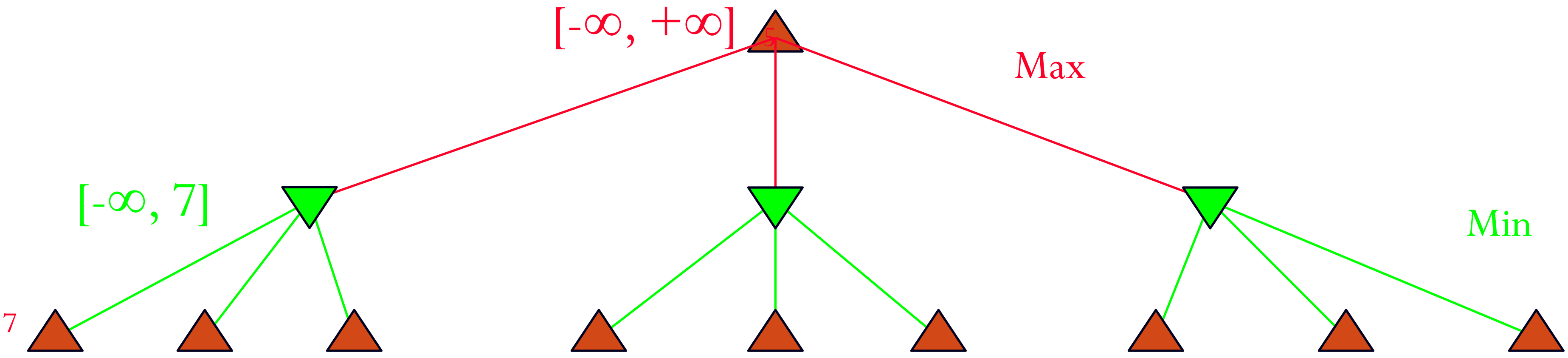


α best choice for Max ?

β best choice for Min ?

- we assume a depth-first, left-to-right search as basic strategy
- the range of the possible values for each node are indicated
  - initially [-∞, +∞]
  - from Max's or Min's perspective
  - these *local* values reflect the values of the sub-trees in that node; the *global* values α and β are the best overall choices so far for Max or Min

Alpha-Beta Example 2

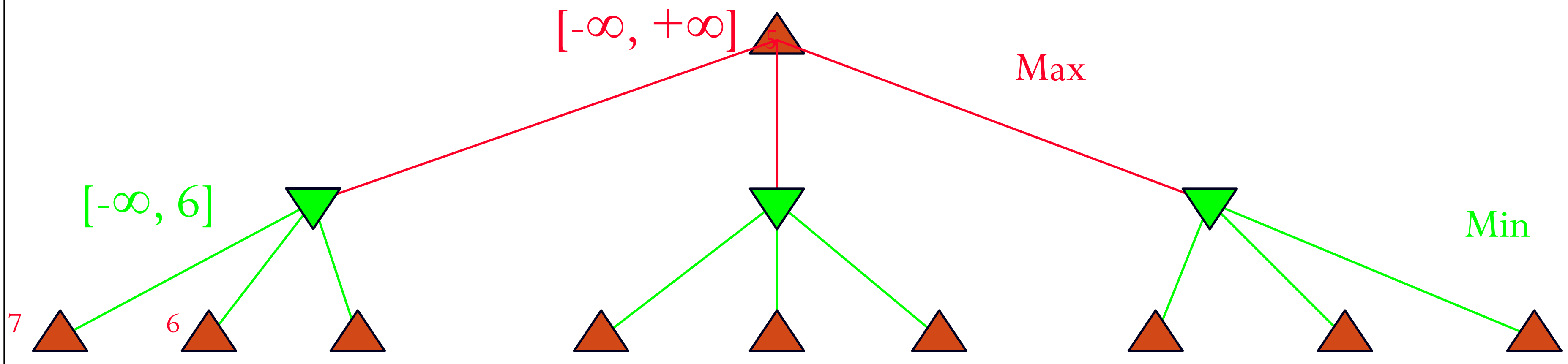


α best choice for Max ?

β best choice for Min 7

- Min obtains the first value from a successor node

Alpha-Beta Example 3

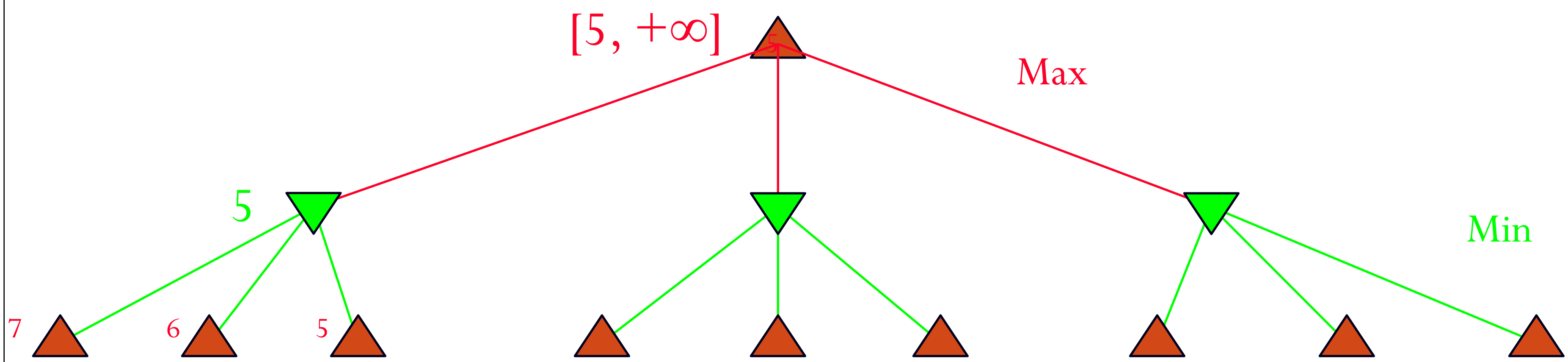


α best choice for Max ?

β best choice for Min 6

- Min obtains the second value from a successor node

Alpha-Beta Example 4

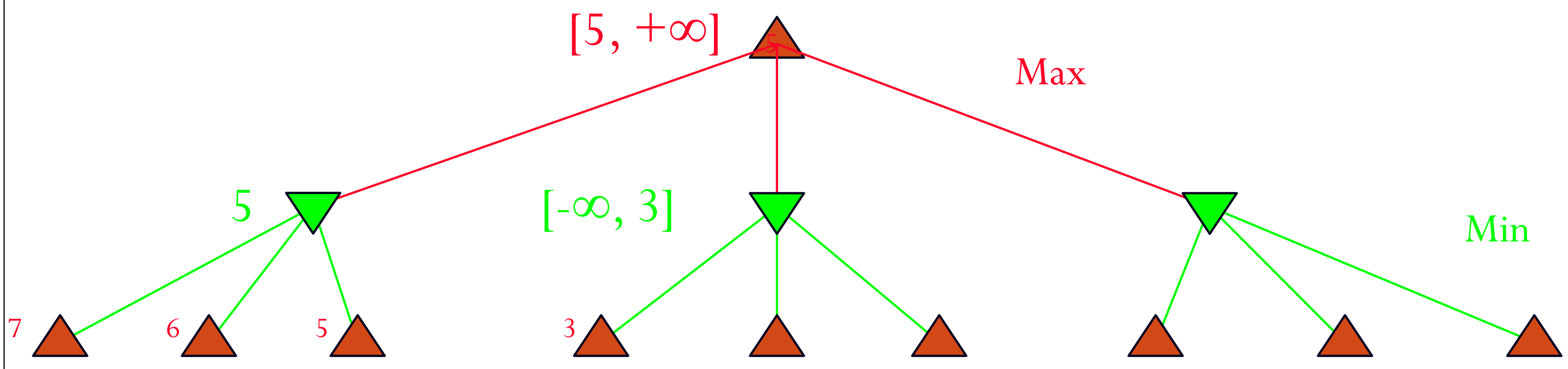


α best choice for Max 5

β best choice for Min 5

- Min obtains the third value from a successor node
- this is the last value from this sub-tree, and the exact value is known
- Max now has a value for its first successor node, but hopes that something better might still come

Alpha-Beta Example 5

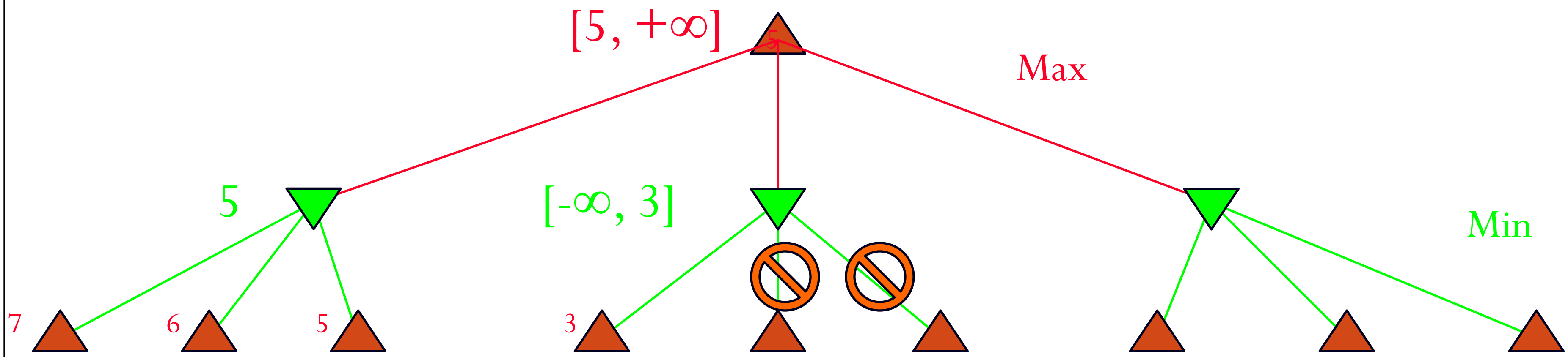


α best choice for Max 5

β best choice for Min 3

- Min continues with the next sub-tree, and gets a better value
- Max has a better choice from its perspective, however, and will not consider a move in the sub-tree currently explored by Min
  - initially [-∞, +∞]

Alpha-Beta Example 6

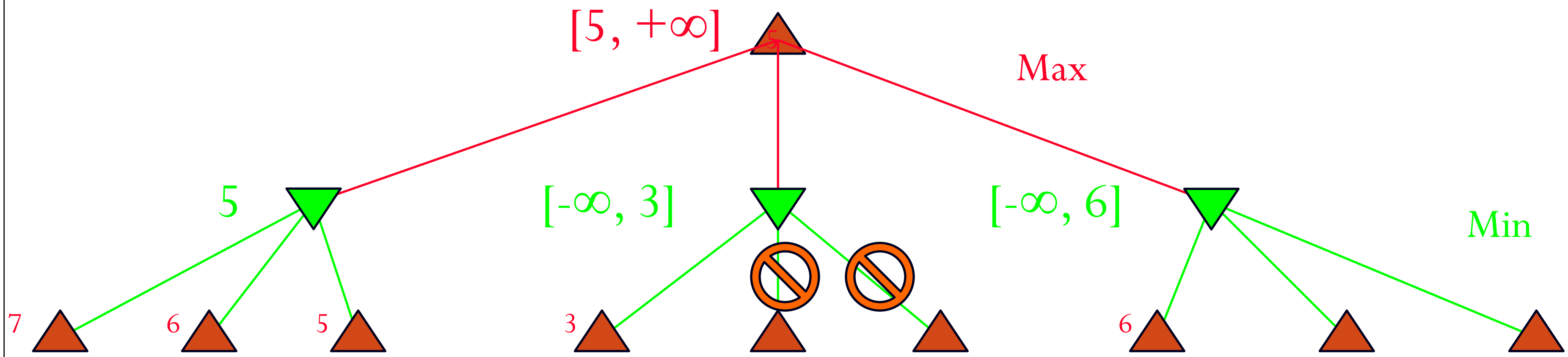


α best choice for Max 5

β best choice for Min 3

- Min knows that Max won't consider a move to this sub-tree, and abandons it
- this is a case of *pruning*, indicated by

Alpha-Beta Example 7

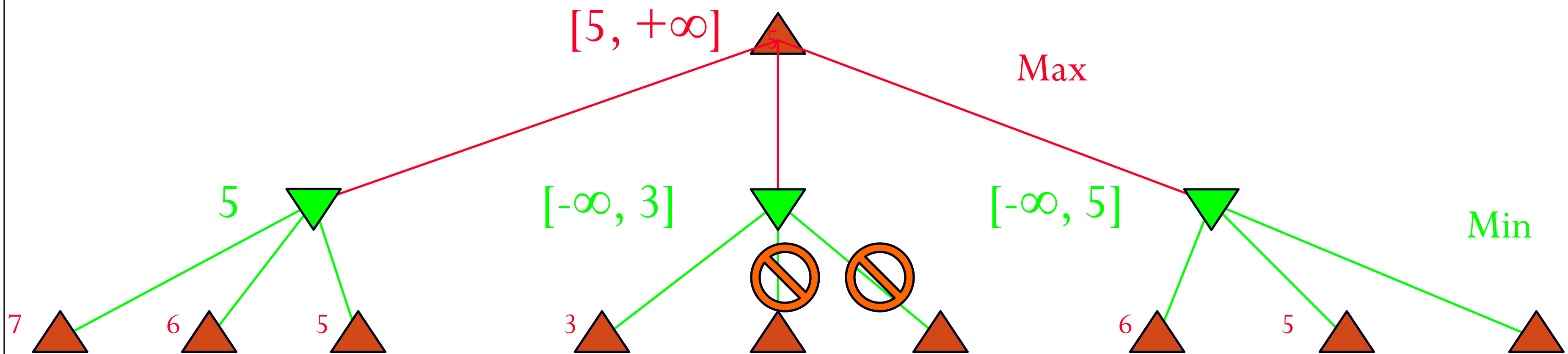


α best choice for Max 5

β best choice for Min 3

- Min explores the next sub-tree, and finds a value that is worse than the other nodes at this level
- if Min is not able to find something lower, then Max will choose this branch, so Min must explore more successor nodes

Alpha-Beta Example 8

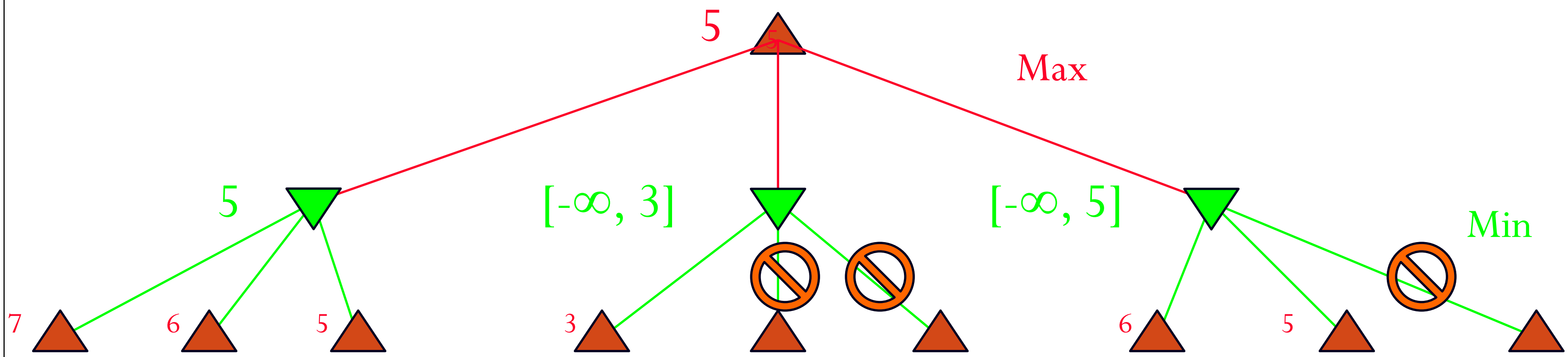


α best choice for Max 5

β best choice for Min 3

- Min is lucky, and finds a value that is the same as the current worst value at this level
- Max can choose this branch, or the other branch with the same value

Alpha-Beta Example 9



α best choice for Max 5

β best choice for Min 3

- Min could continue searching this sub-tree to see if there is a value that is less than the current worst alternative in order to give Max as few choices as possible
  - this depends on the specific implementation
- Max knows the best value for its sub-tree

Properties of Alpha-Beta Pruning

- in the ideal case, the best successor node is examined first
  - results in $O(b^{d/2})$ nodes to be searched instead of $O(b^d)$
  - alpha-beta can look ahead twice as far as minimax
  - in practice, simple ordering functions are quite useful
- assumes an idealized tree model
  - uniform branching factor, path length
  - random distribution of leaf evaluation values
- transpositions tables can be used to store permutations
  - sequences of moves that lead to the same position
- requires additional information for good players
  - game-specific background knowledge
  - empirical data

Imperfect Decisions

- complete search is impractical for most games
- alternative: search the tree only to a certain depth
  - requires a cutoff-test to determine where to stop
    - replaces the terminal test
    - the nodes at that level effectively become terminal leave nodes
  - uses a heuristics-based evaluation function to estimate the expected utility of the game from those leave nodes

Evaluation Function

- determines the performance of a game-playing program
- must be consistent with the utility function
  - values for terminal nodes (or at least their order) must be the same
- tradeoff between accuracy and time cost
  - without time limits, minimax could be used
- should reflect the actual chances of winning
- frequently weighted linear functions are used
  - $E = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$
  - combination of features, weighted by their relevance

Example: Tic-Tac-Toe

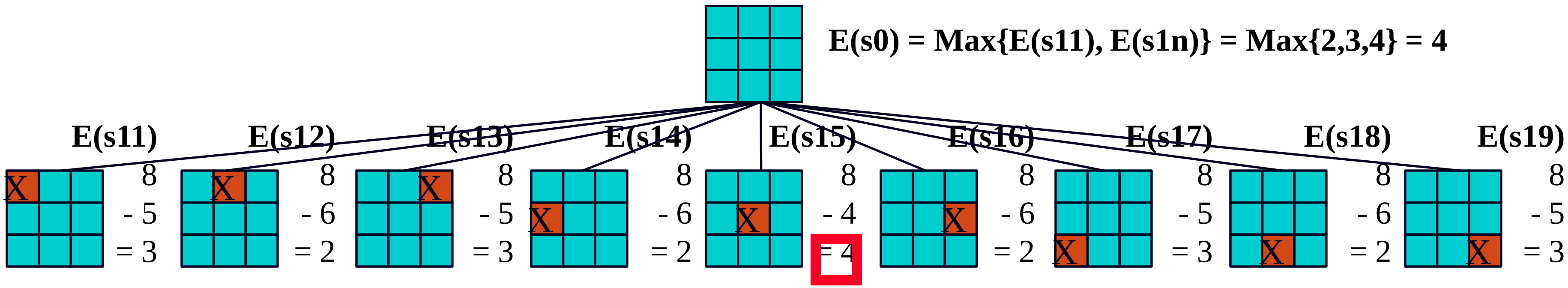
- simple evaluation function

$$E(s) = (rx + cx + dx) - (ro + co + do)$$

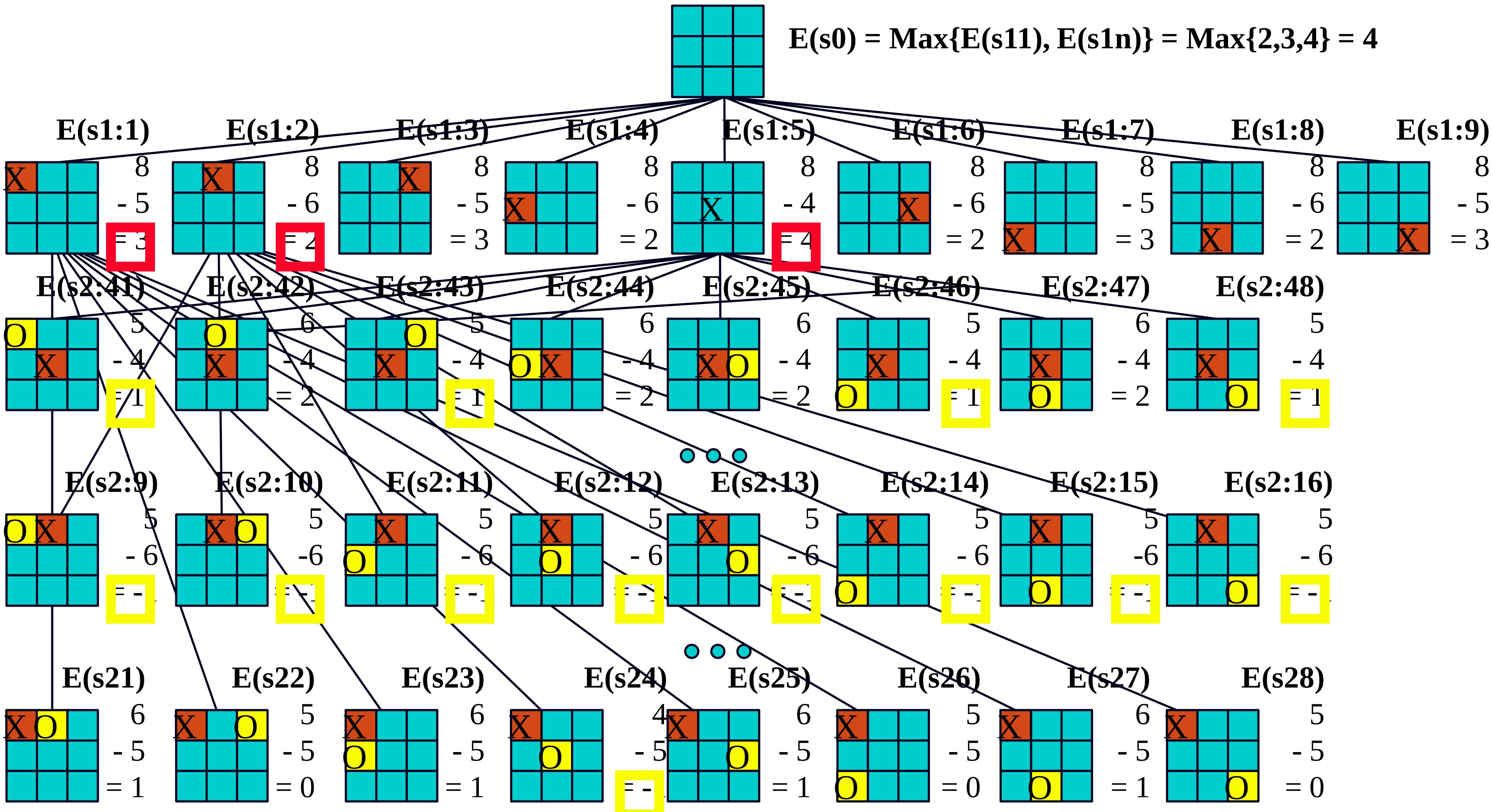
(number of rows, columns, and diagonals open for MAX) – (number of rows, columns, and diagonals open for MIN)

- 1-ply lookahead
  - start at the top of the tree
  - evaluate all 9 choices for player 1
  - pick the maximum E-value
- 2-ply lookahead
  - also looks at the opponents possible move
    - assuming that the opponents picks the minimum E-value

Tic-Tac-Toe 1-Ply



Tic-Tac-Toe 2-Ply



Checkers Case Study

- initial board configuration
  - Black — single on 20, single on 21, king on 31
  - Red — single on 23, king on 22
  - evaluation function

    $E(s) = (5\,x_1 + x_2) - (5r_1 + r_2)$

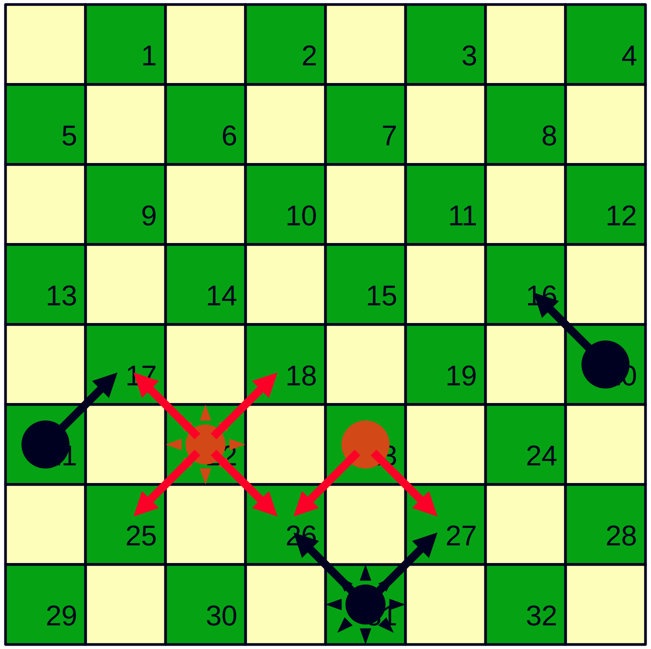
    where

    x_1 = black king advantage,

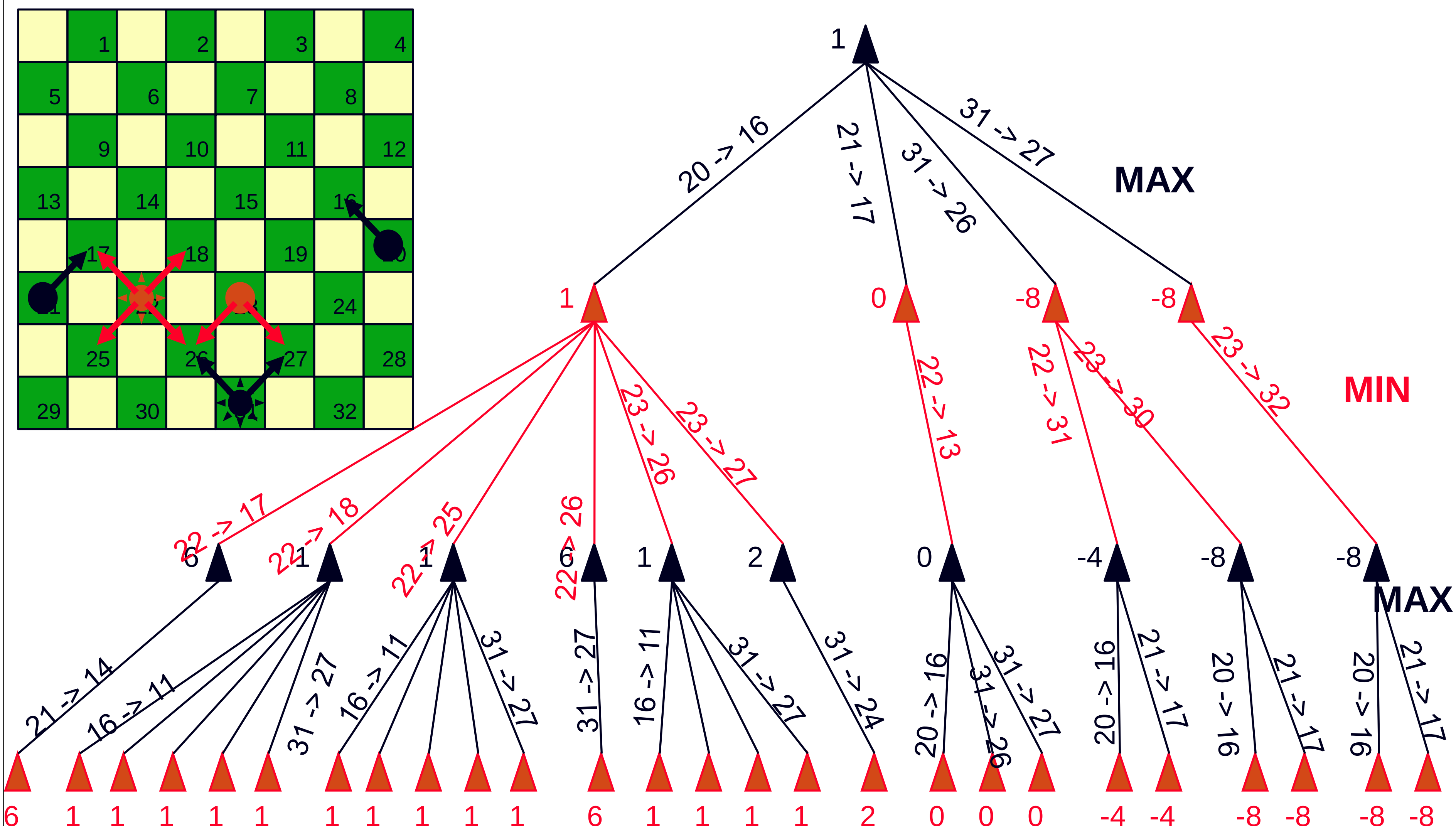
    x_2 = black single advantage,

    r_1 = red king advantage,

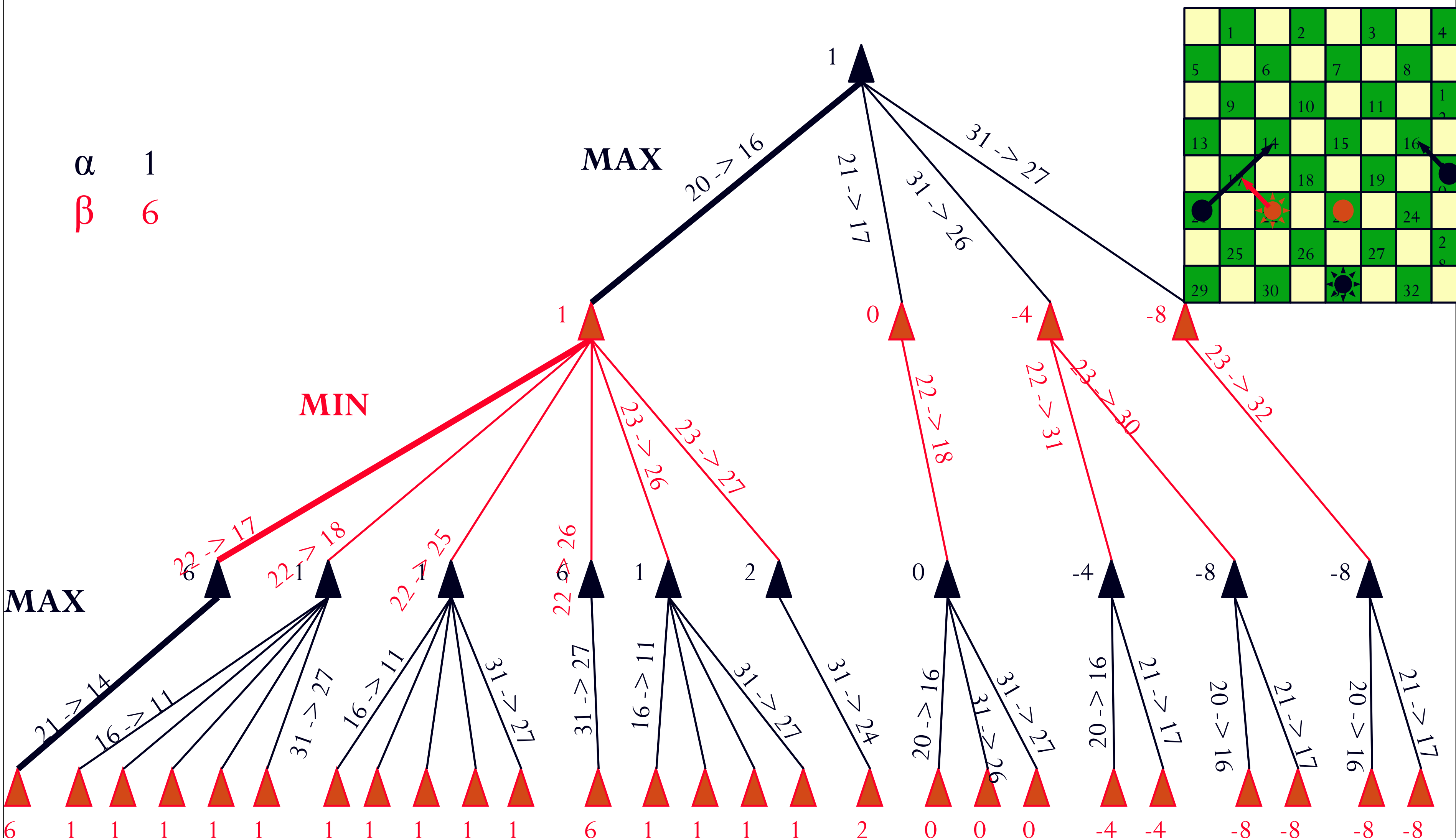
    r_2 = red single advantage



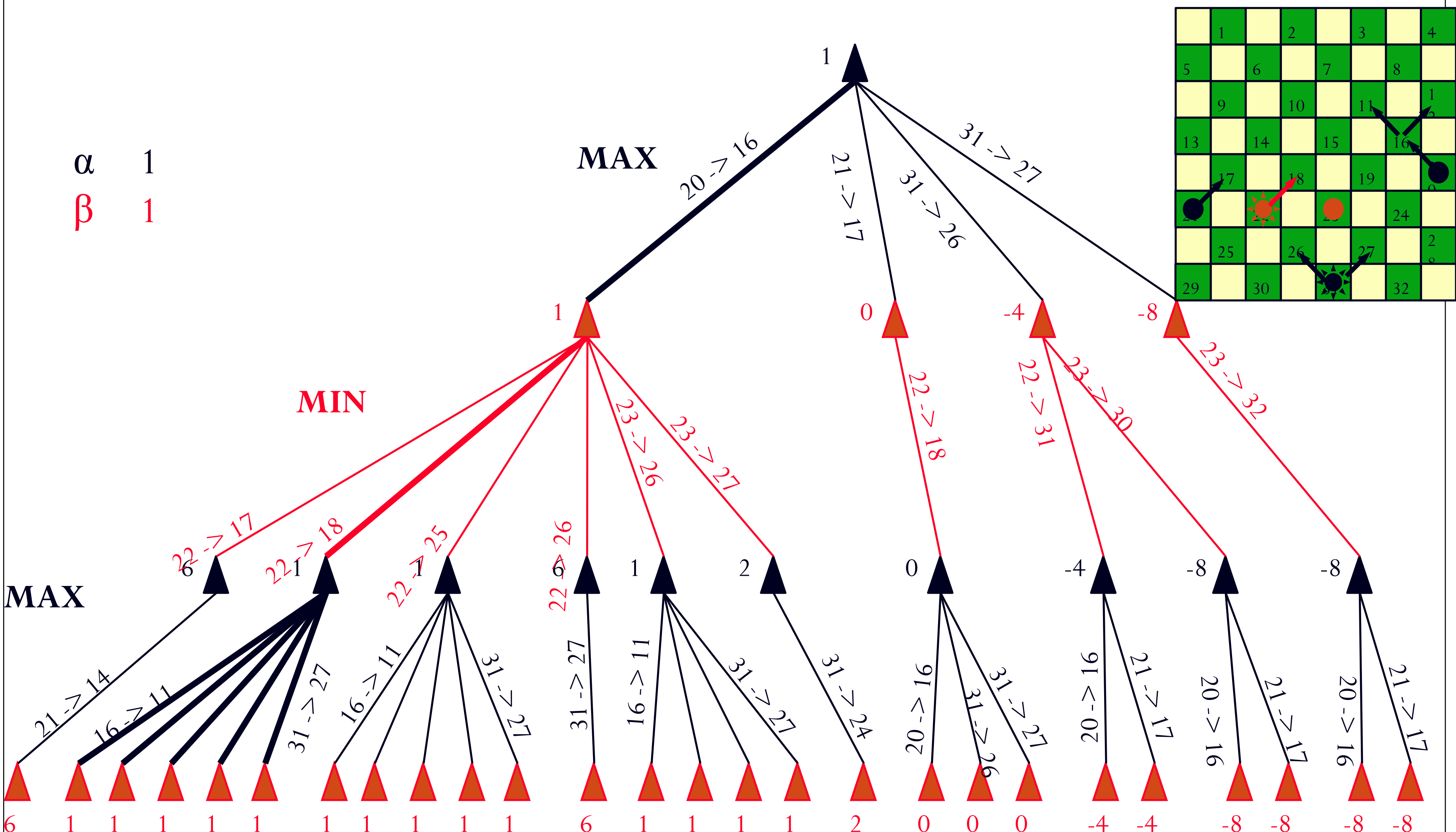
Checkers MiniMax Example



Checkers Alpha-Beta Example



Checkers Alpha-Beta Example

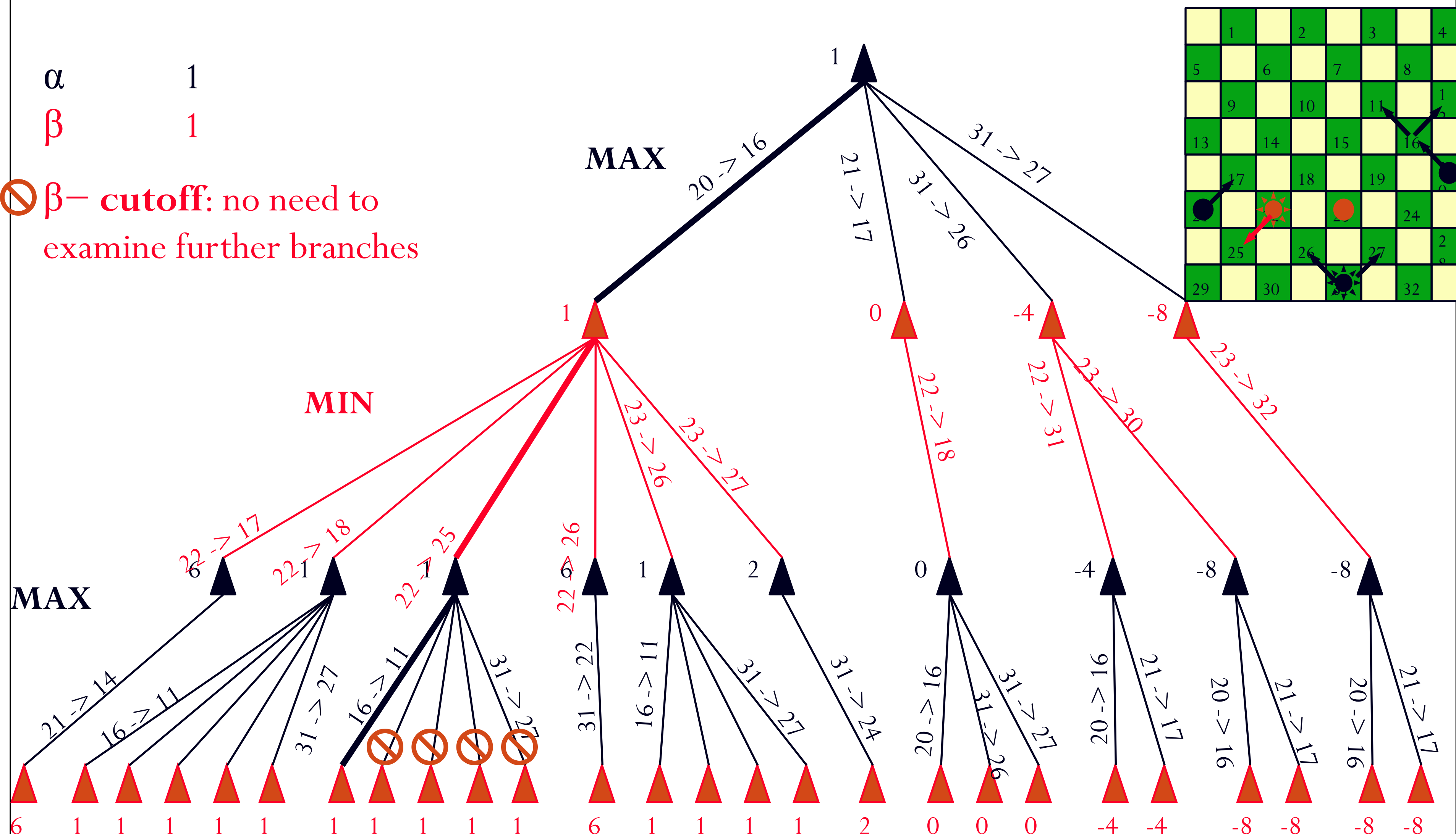


Checkers Alpha-Beta Example

α 1

β 1

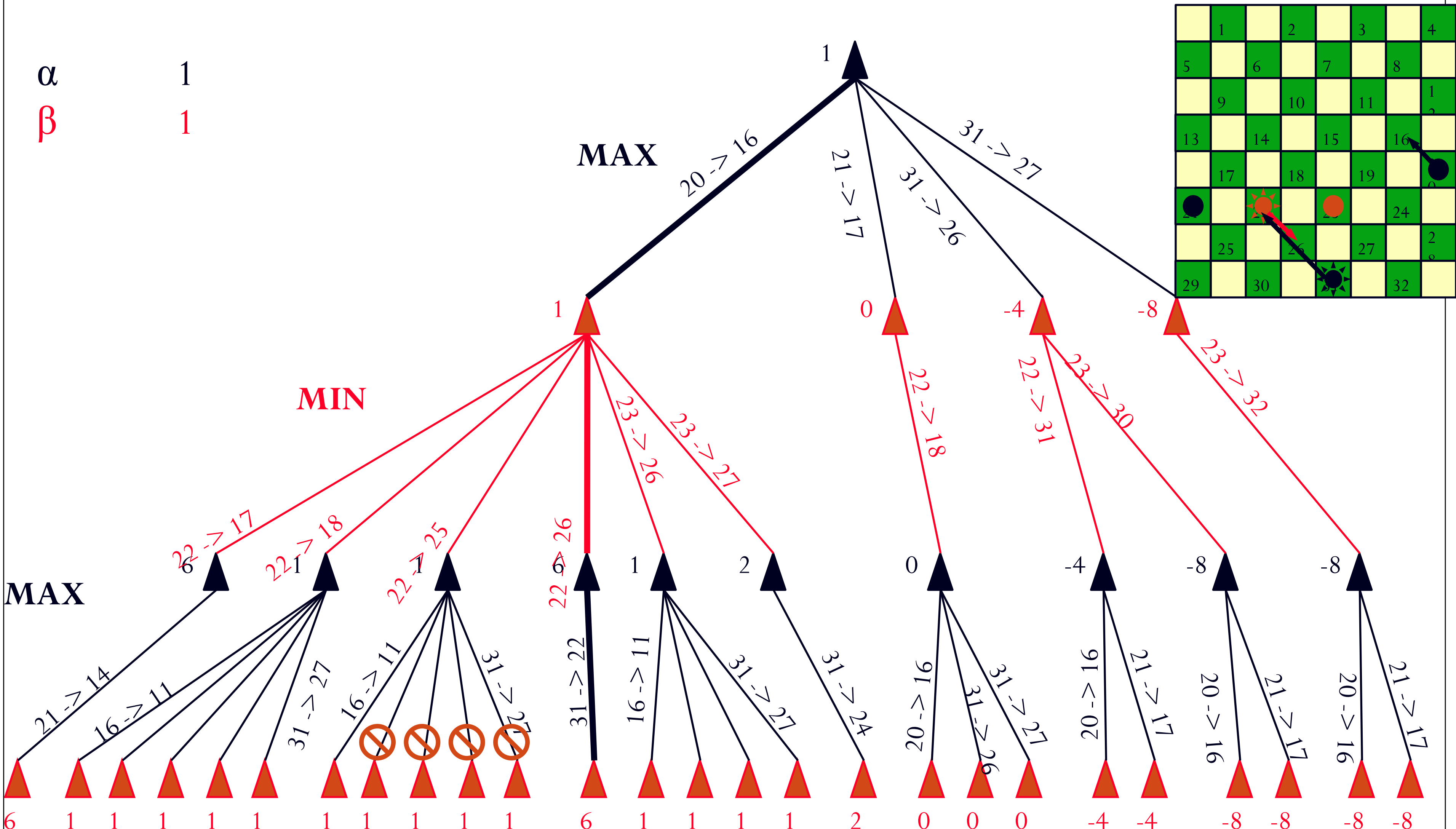
β− **cutoff**: no need to examine further branches



Checkers Alpha-Beta Example

α 1

β 1

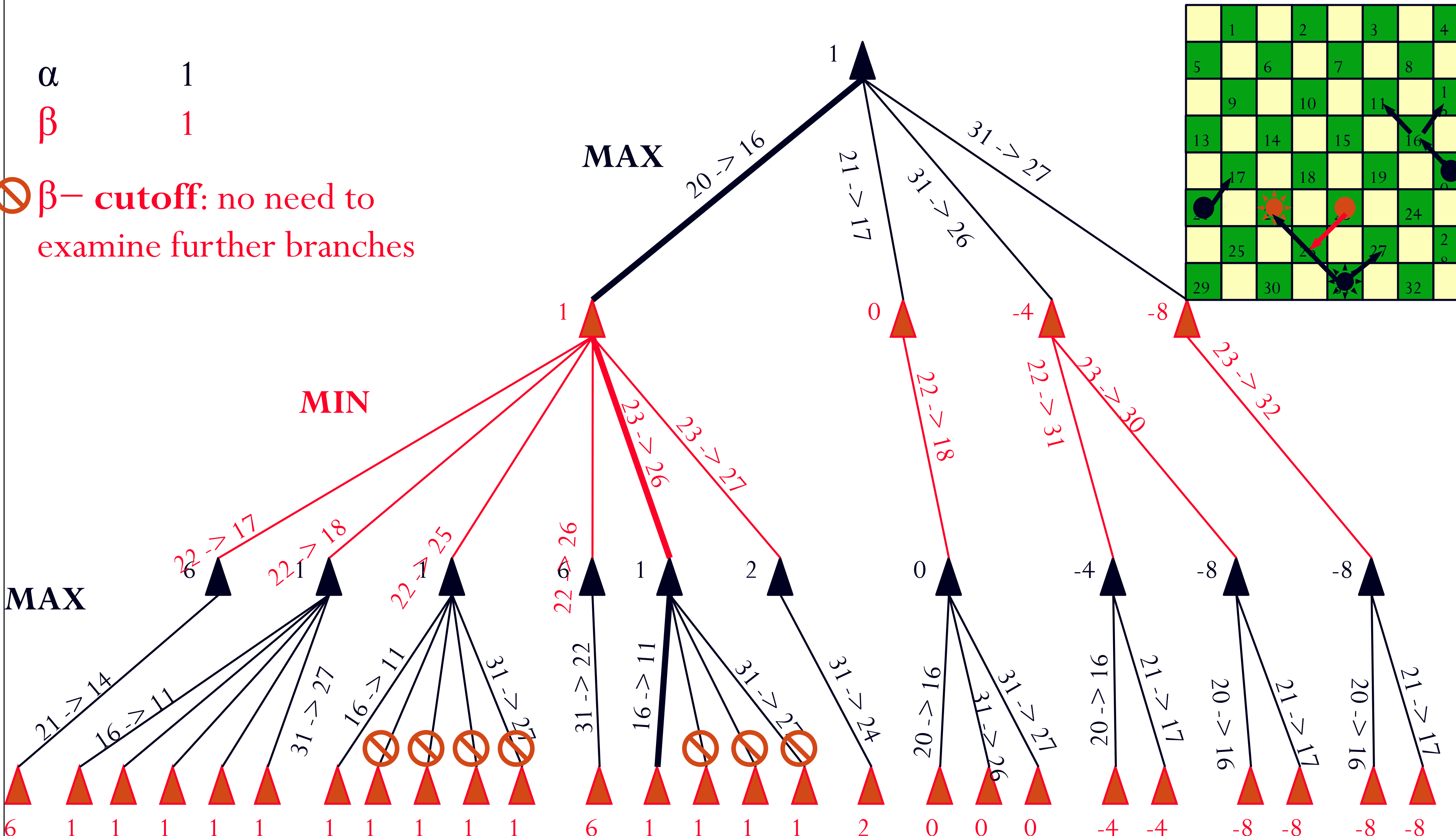


Checkers Alpha-Beta Example

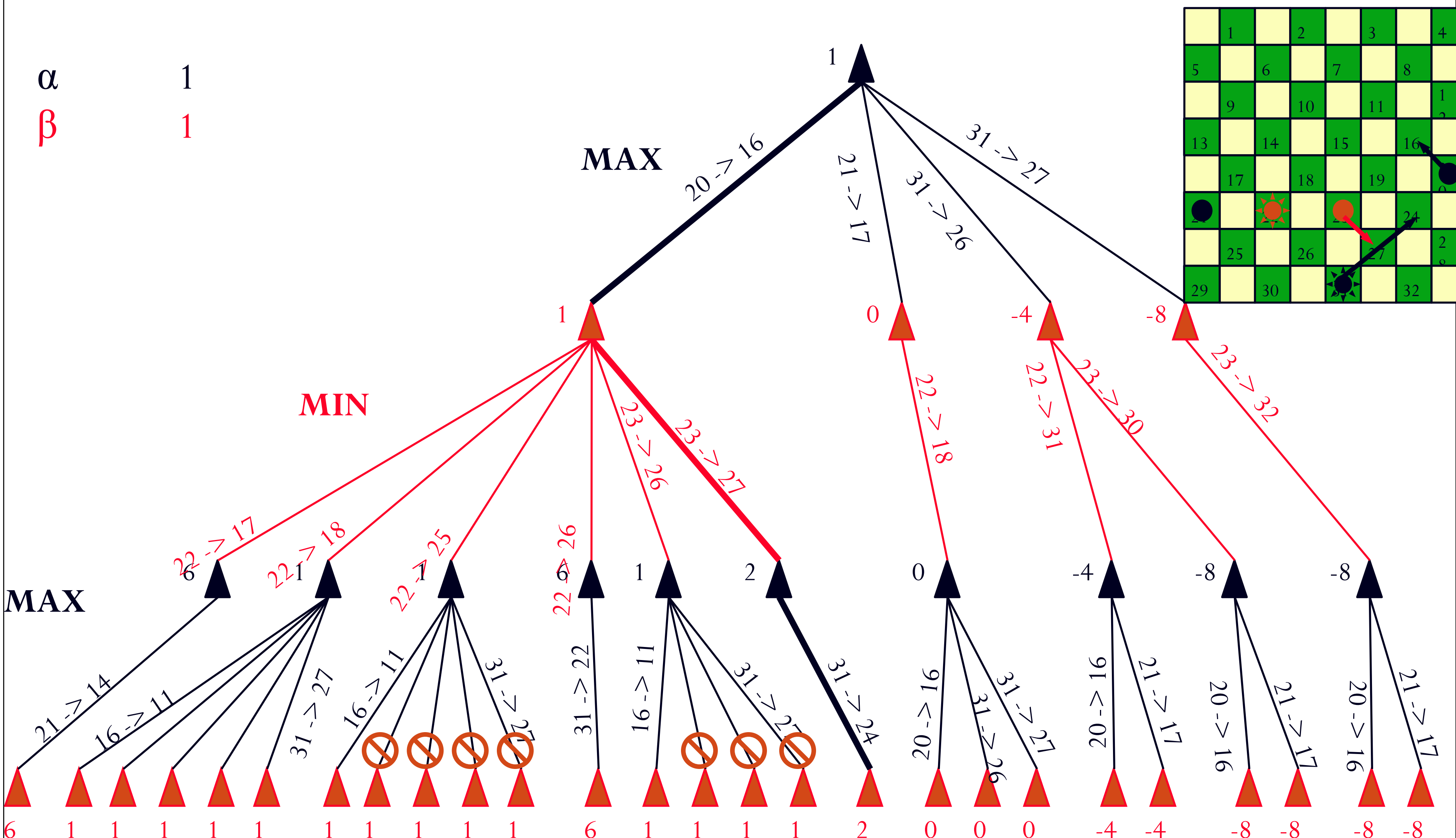
α 1

β 1

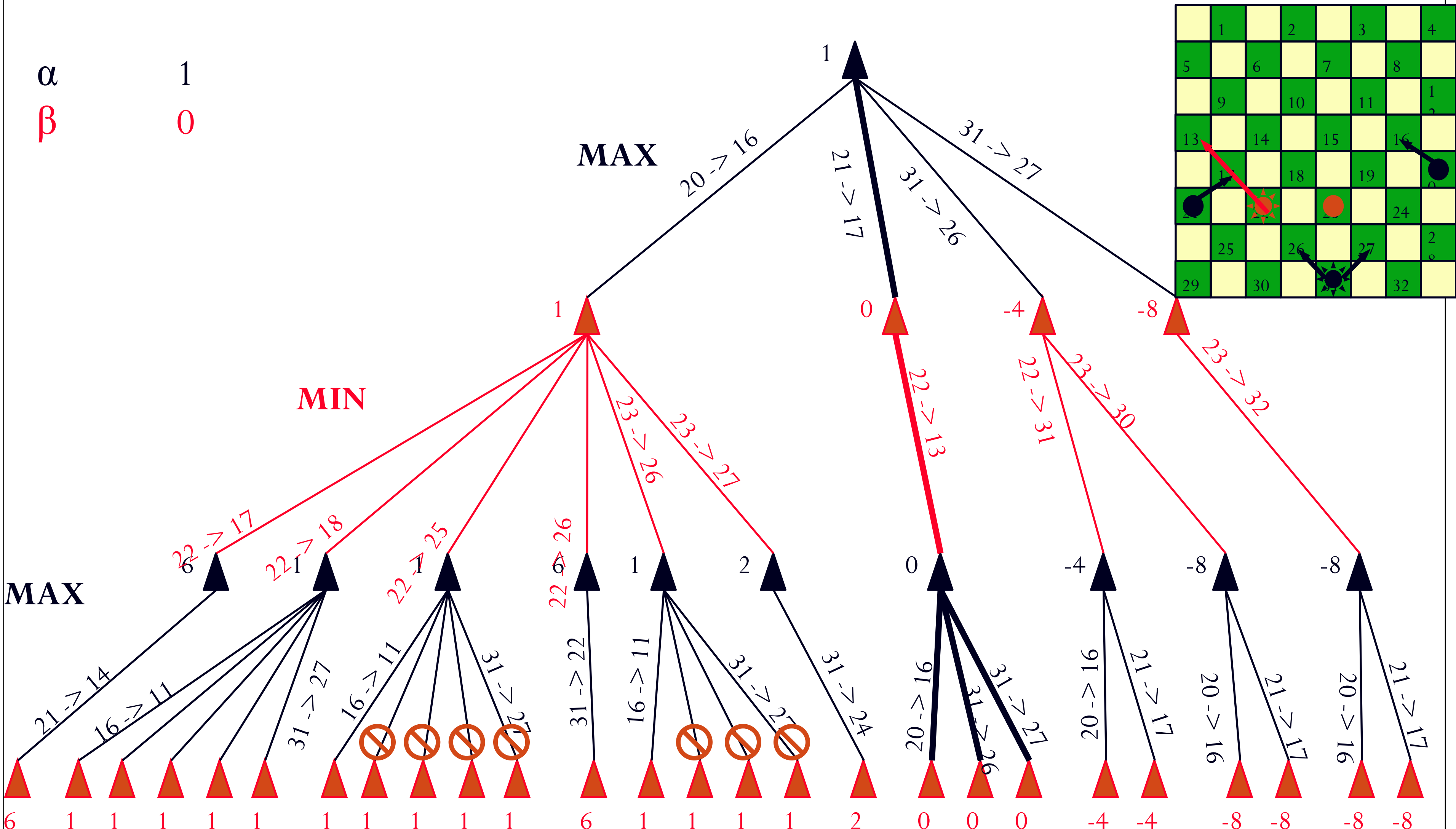
β− **cutoff**: no need to examine further branches



Checkers Alpha-Beta Example



Checkers Alpha-Beta Example

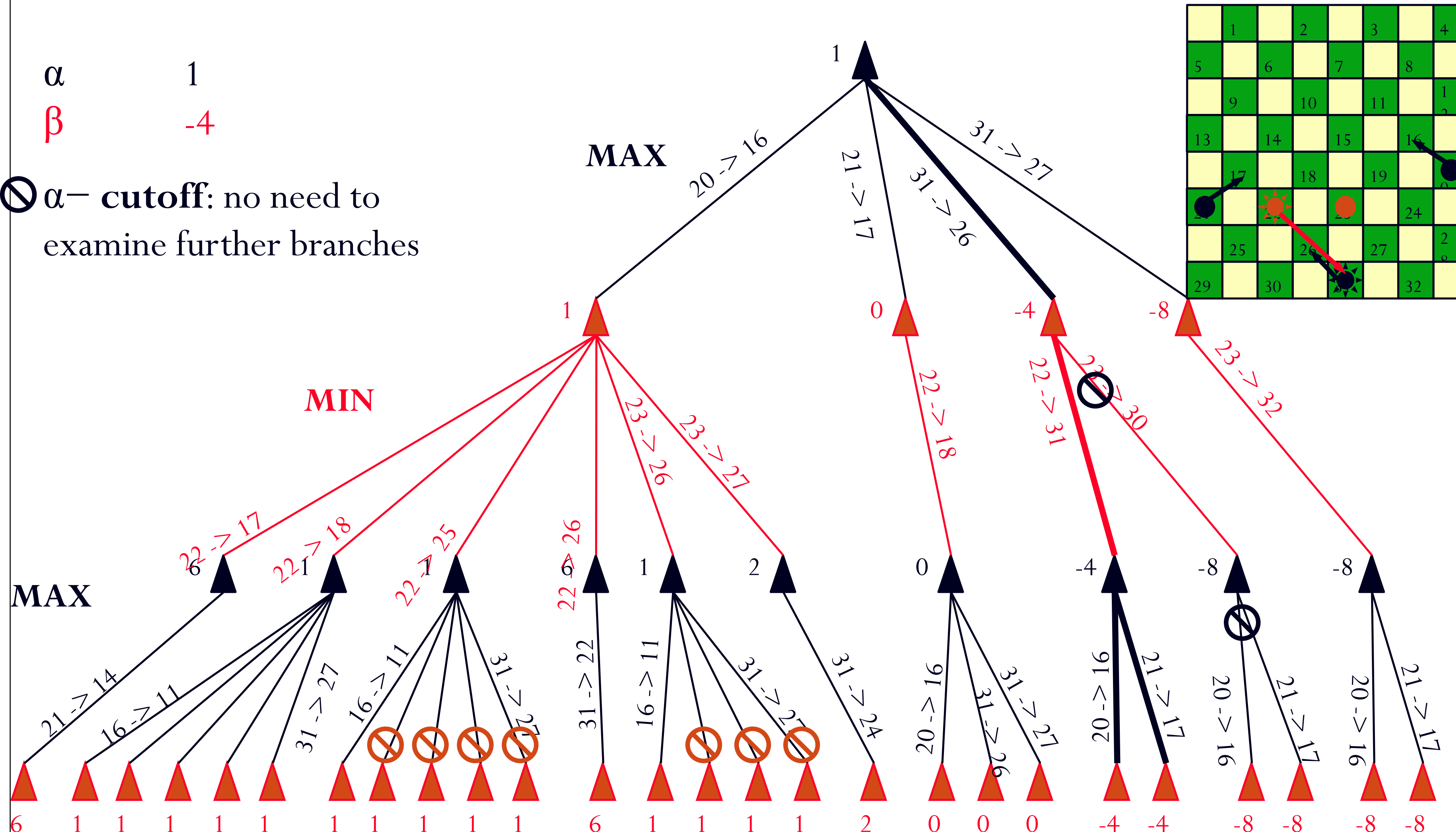


Checkers Alpha-Beta Example

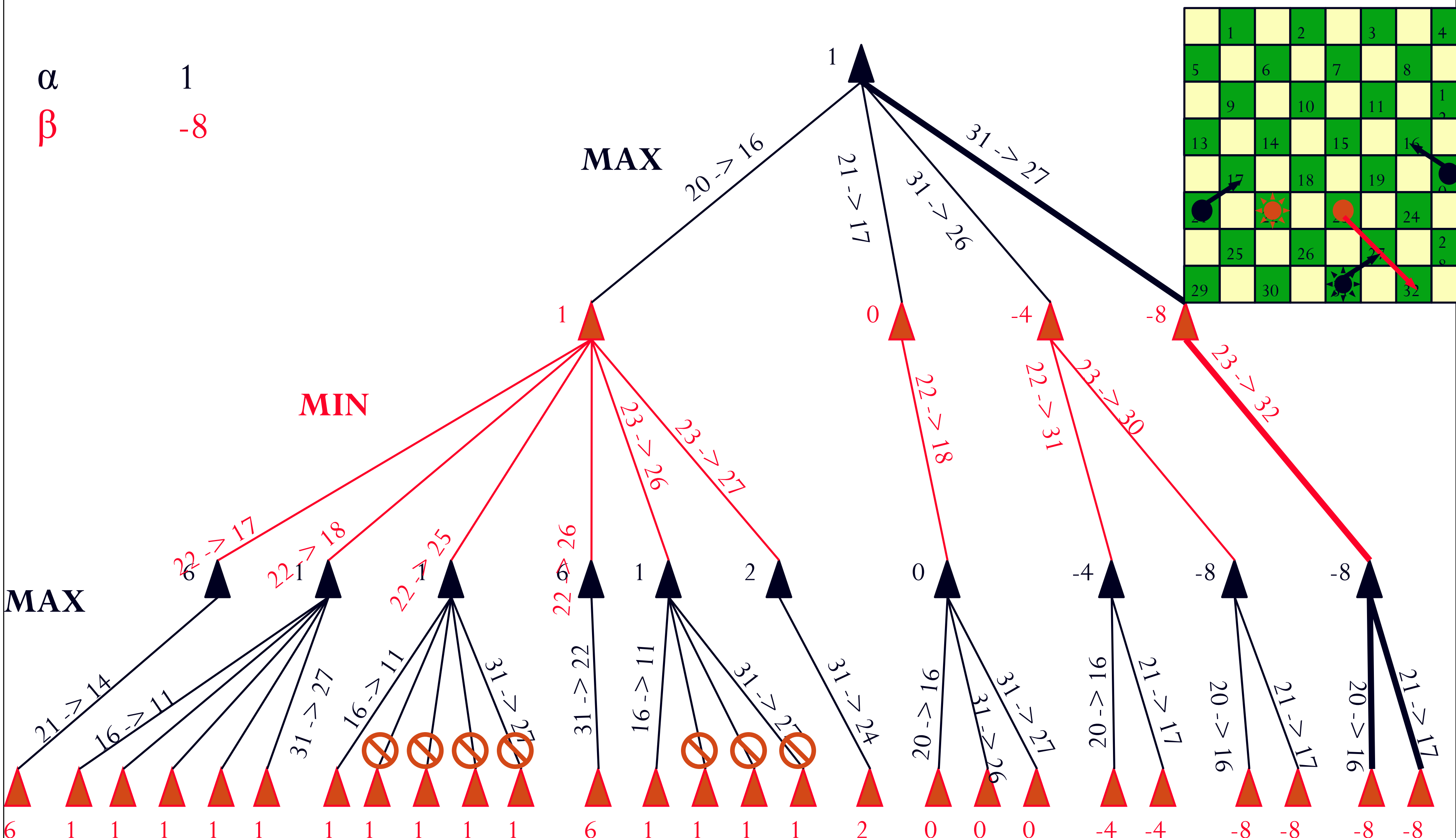
α 1

β -4

α− **cutoff**: no need to examine further branches



Checkers Alpha-Beta Example



Search Limits

- search must be cut off because of time or space limitations
- strategies like depth-limited or iterative deepening search can be used
  - don't take advantage of knowledge about the problem
- more refined strategies apply background knowledge
  - quiescent search
    - cut off only parts of the search space that don't exhibit big changes in the evaluation function

Horizon Problem

- moves may have disastrous consequences in the future, but the consequences are not visible
  - the corresponding change in the evaluation function will only become evident at deeper levels
    - they are "beyond the horizon"
- determining the horizon is an open problem without a general solution
  - only some pragmatic approaches restricted to specific games or situation

Games with Chance

- in many games, there is a degree of unpredictability through random elements
  - throwing dice, card distribution, roulette wheel, …
- this requires *chance nodes* in addition to the Max and Min nodes
  - branches indicate possible variations
  - each branch indicates the outcome and its likelihood

Decisions with Chance

- the utility value of a position depends on the random element
  - the definite minimax value must be replaced by an expected value
- calculation of expected values
  - utility function for terminal nodes
  - for all other nodes
    - calculate the utility for each chance event
    - weigh by the chance that the event occurs
    - add up the individual utilities

Chapter Summary

- many game techniques are derived from search methods
- the minimax algorithm determines the best move for a player by calculating the complete game tree
- alpha-beta pruning dismisses parts of the search tree that are provably irrelevant
- an evaluation function gives an estimate of the utility of a state when a complete search is impractical
- chance events can be incorporated into the minimax algorithm by considering the weighted probabilities of chance events