Outline

- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - α-β pruning
  - NegaScout
- Real-time Game-Tree Search
  - evaluation functions
  - practical enhancements
  - selective search
- Games of imperfect information and games of chance
- Simulation Search
  - Monte-Carlo search
  - UCT search

What are and why study games?

- Games are a form of multi-agent environment
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
  - Competitive multi-agent environments give rise to adversarial search a.k.a. games

- Why study games?
  - Fun; historically entertaining
  - Interesting subject of study because they are hard
  - Easy to represent and agents restricted to small number of actions
  - Problem (and success) is easy to communicate

Relation of Games to Search

- Search – no adversary
  - Solution is method for finding goal
  - Heuristics and CSP 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
  - Time limits force an *approximate* solution
  - Evaluation function:
    - evaluate “goodness” of game position
  - Examples:
    - chess, checkers, Othello, backgammon, ...

Types of Games

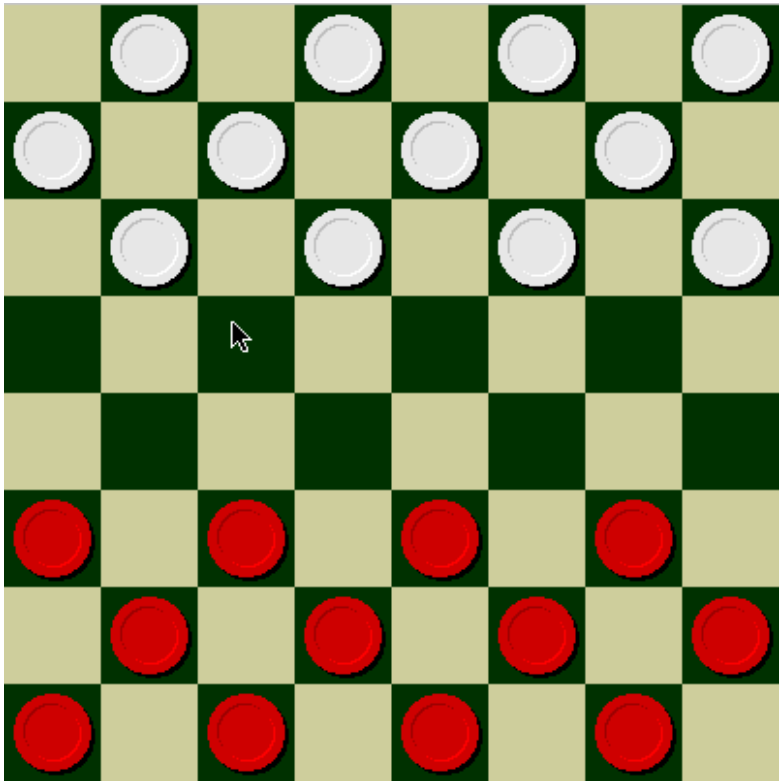
- Zero-Sum Games
  - one player's gain is the other player's (or players') loss
- turn-taking
  - players alternate moves
- deterministic games vs. games of chance
  - do random components influence the progress of the game?
- perfect vs. imperfect information
  - does every player see the entire game situation?

	deterministic	chance
perfect information	chess, checkers, Go, Othello	backgammon, monopoly
imperfect information	battleship, kriegspiel, matching pennies, Roshambo	bridge, poker, scrabble

A Brief History of Search in Game Playing

- Computer considers possible lines of play
  (Babbage, 1846)
- Algorithm for perfect play
  (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation
  (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program
  (Turing, 1951)
- Machine learning to improve evaluation accuracy
  (Samuel, 1952-57)
- Selective Search Programs
  (Newell, Shaw, Simon 1958; Greenblatt, Eastake, Crocker 1967)
- Pruning to allow deeper search
  (McCarthy, 1956)
- Breakthrough of Brute-Force Programs
  (Atkin & Slate, 1970-77)

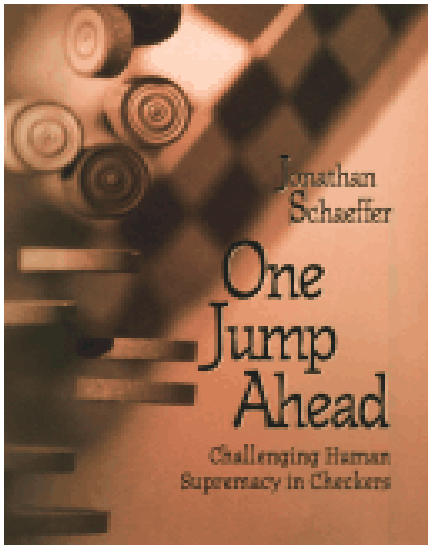
Checkers



Chinook vs. Tinsley

Name: Marion Tinsley
Profession: Teach
  mathematics
Hobby: Checkers
Record: Over 42 years
  loses only 3 (!)
  games of checkers

Chinook

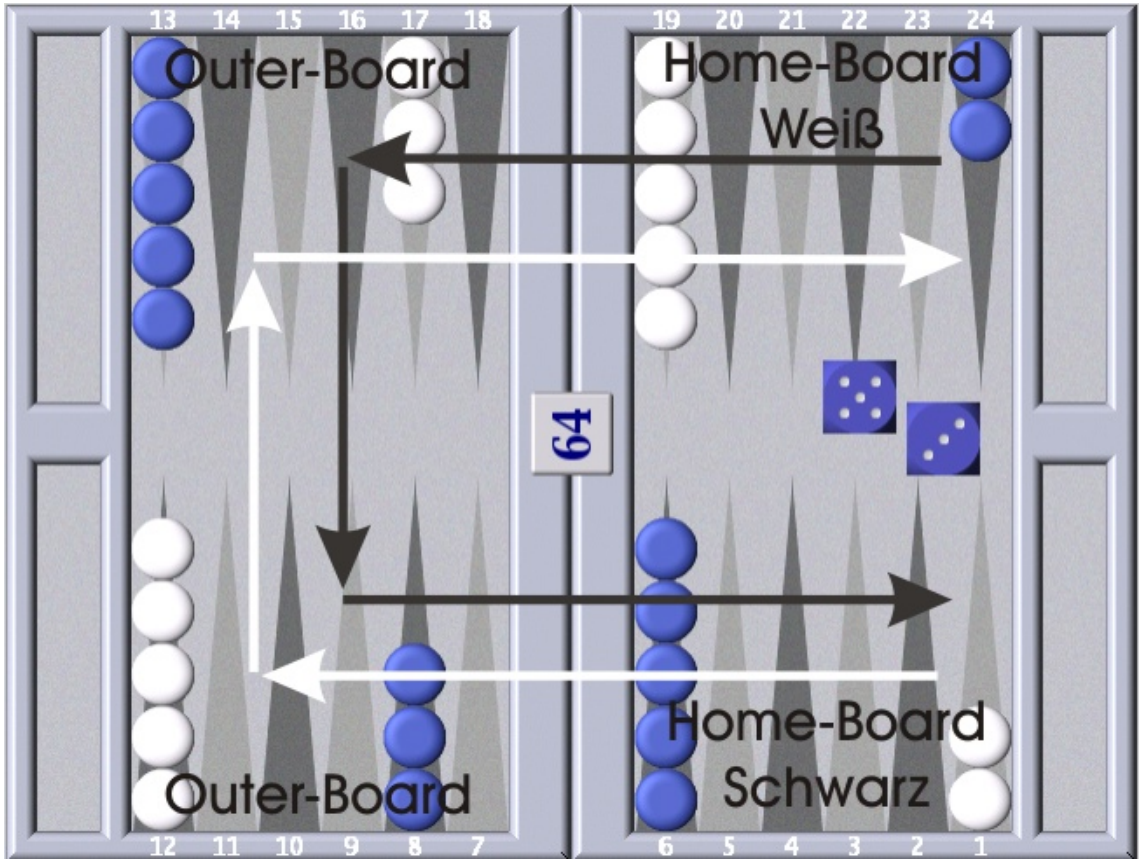


First computer to win human world championship!

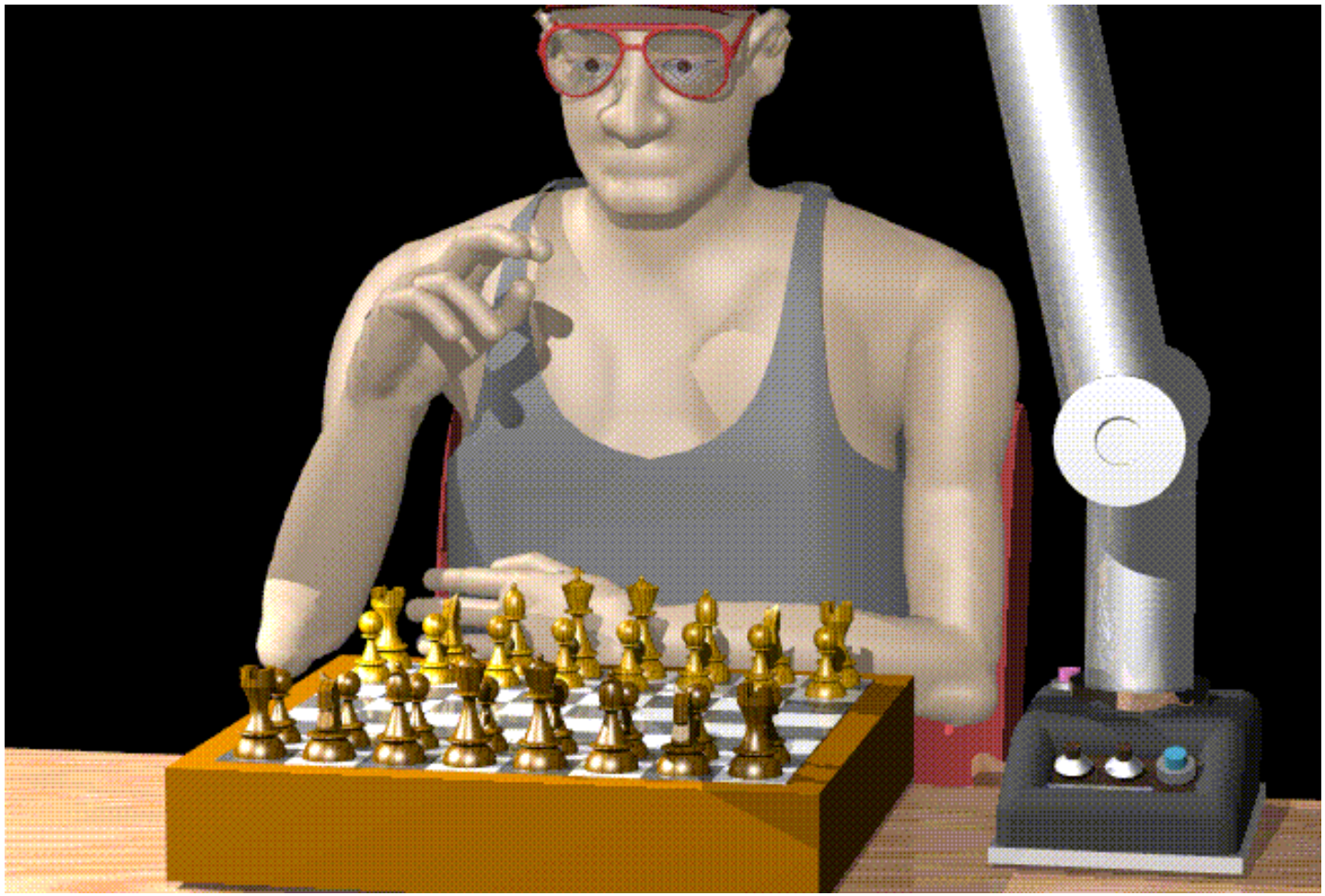
Visit http://www.cs.ualberta.ca/~chinook/ to play a version of Chinook over the Internet.

Backgammon

- branching factor several hundred
- TD-Gammon v1 – 1-step lookahead, learns to play games against itself
- TD-Gammon v2.1 – 2-ply search, does well against world champions
- TD-Gammon has changed the way experts play backgammon.



Chess



Man vs. Machine

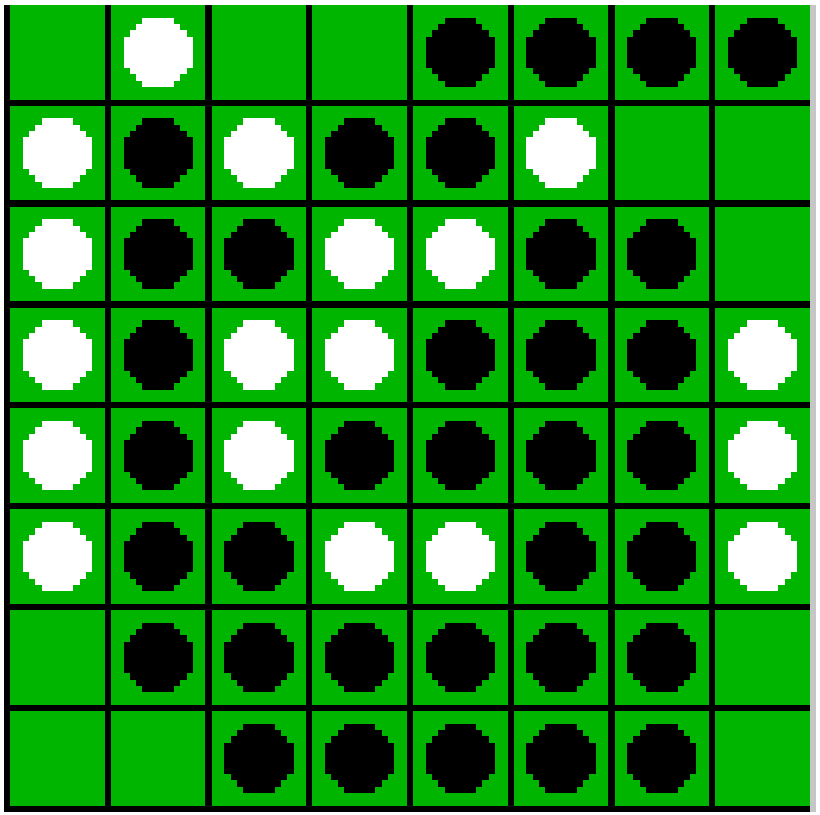
Kasparov	Name	Deep Blue
5'10"	Height	6' 5"
176 lbs	Weight	2,400 lbs
34 years	Age	4 years
50 billion neurons	Computers	512 processors
2 pos/sec	Speed	200,000,000 pos/sec
Extensive	Knowledge	Primitive
Electrical/chemical	Power Source	Electrical
Enormous	Ego	None

Chess

Name: Garry Kasparov
Title: World Chess Champion
Crime: Valued greed over common sense

- http://www.wired.com/wired/archive/9.10/chess.htm
- http://www.byte.com/art/9707/sec6/art6.htm

Reversi/Othello



Othello

Name: Takeshi Murakami
Title: World Othello Champion
Crime: Man crushed by machine

Go: On the One Side

Name: Chen Zhixing
Author: Handtalk (Goemate)
Profession: Retired
Computer skills: self-taught assembly language programmer
Accomplishments: dominated computer go for 4 years.

Go: And on the Other

Gave Handtalk a 9 stone handicap and still easily beat the program, thereby winning $15,000

Outline

- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - α-β pruning
  - NegaScout
- Real-time Game-Tree Search
  - evaluation functions
  - practical enhancements
  - selective search
- Games of imperfect information and games of chance
- Simulation Search
  - Monte-Carlo search
  - UCT search

Status Quo in Game Playing

- Solved
  - Tic-Tac-Toe, Connect-4, Go-Moku, 4-men Morris
  - Most recent addition: Checkers is a draw
    - Solved with almost 20 years of computation time (first endgame databases were computed in 1989)
  - http://www.sciencemag.org/cgi/content/abstract/1144079
- Partly solved
  - Chess
    - all 6-men endgames, some 7-men endgames
    - longest win: position in KQN vs. KRBN after 517 moves
      - http://www.gothicchess.com/javascript_8x8_chess_endings.html
- World-Championship strength
  - Chess, Checkers, Backgammon, Scrabble, Othello
- Human Supremacy
  - Go, Shogi, Bridge, Poker (probably the next to fall)

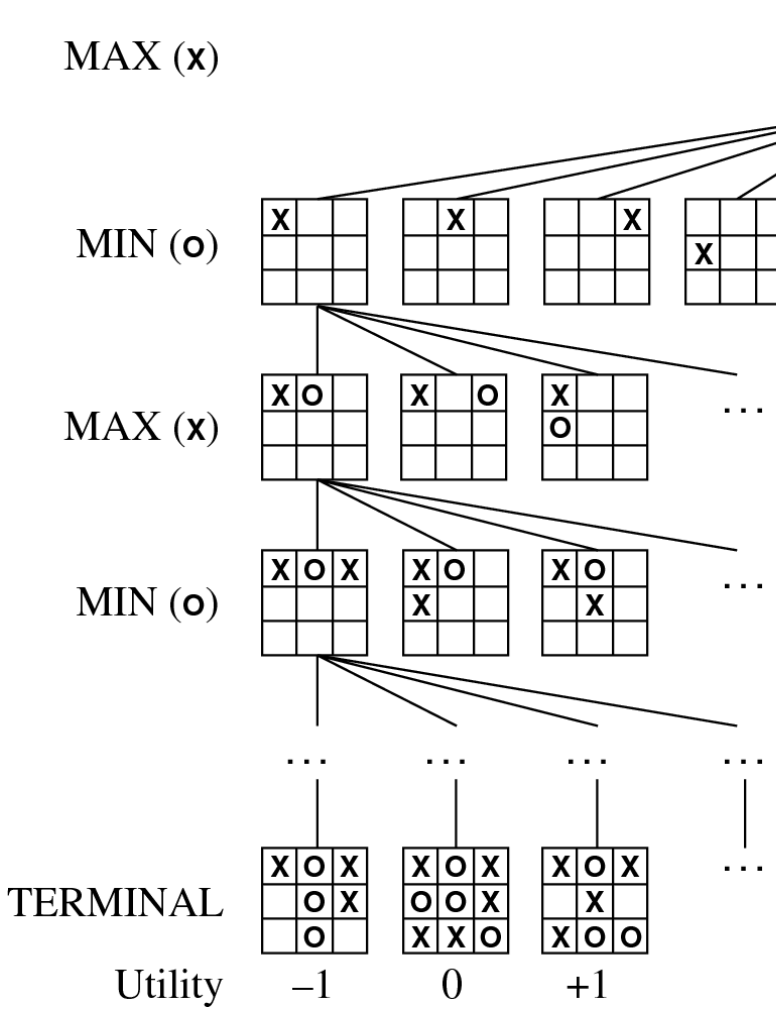
Solving a Game

- Ultra-weak
  - prove whether the first player will win, lose, or draw from the initial position, given perfect play on both sides
  - could be a non-constructive proof, which does not help in play
  - could be done via a complete minimax or alpha-beta search
  - Example:
    - chess when first move may be a pass
- Weak
  - provide an algorithm which secures a win for one player, or a draw for either, against any possible moves by the opponent, from the initial position only
- Strong
  - provide an algorithm which can produce perfect play from any position
    - often in the form of a database for all positions

Game setup

- Two players: MAX and MIN
  - MAX moves first and they take turns until the game is over.
    - ply: a half-move by one of the players
    - move: two plies, one by MAX and one by MIN
  - Winner gets award, looser 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 (*objective function, payoff function*)
    - Gives numerical value of terminal states
    - E.g. win (+1), loose (-1) and draw (0) in tic-tac-toe (next)
    - typically from the point of view of MAX

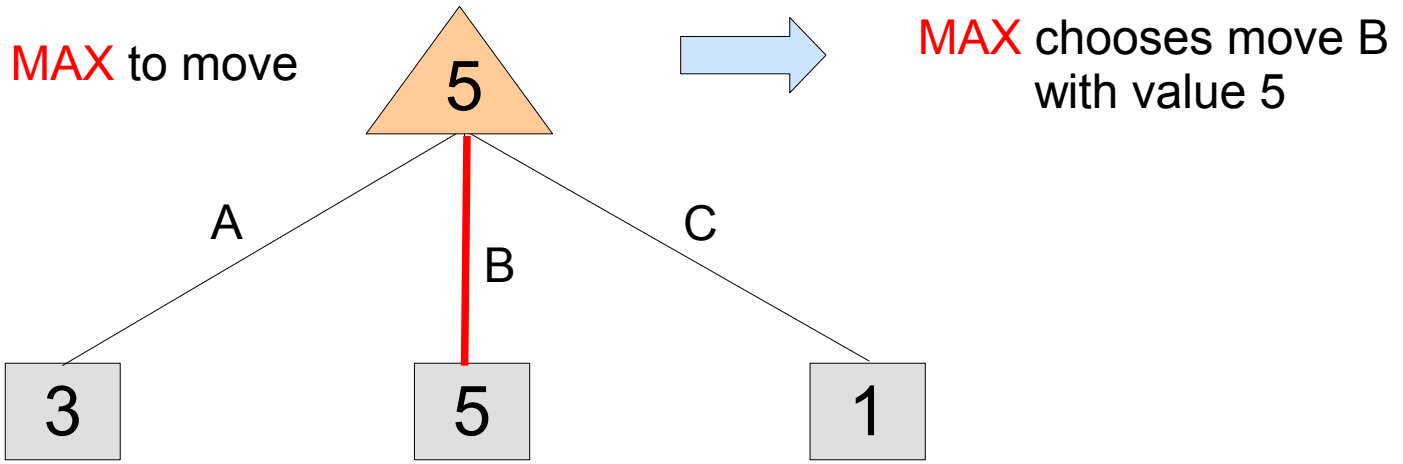
Partial Game Tree for Tic-Tac-Toe



- MAX is to move at odd depths
- MIN is to move at even depths
- Terminal nodes are evaluated from MAX's point of view

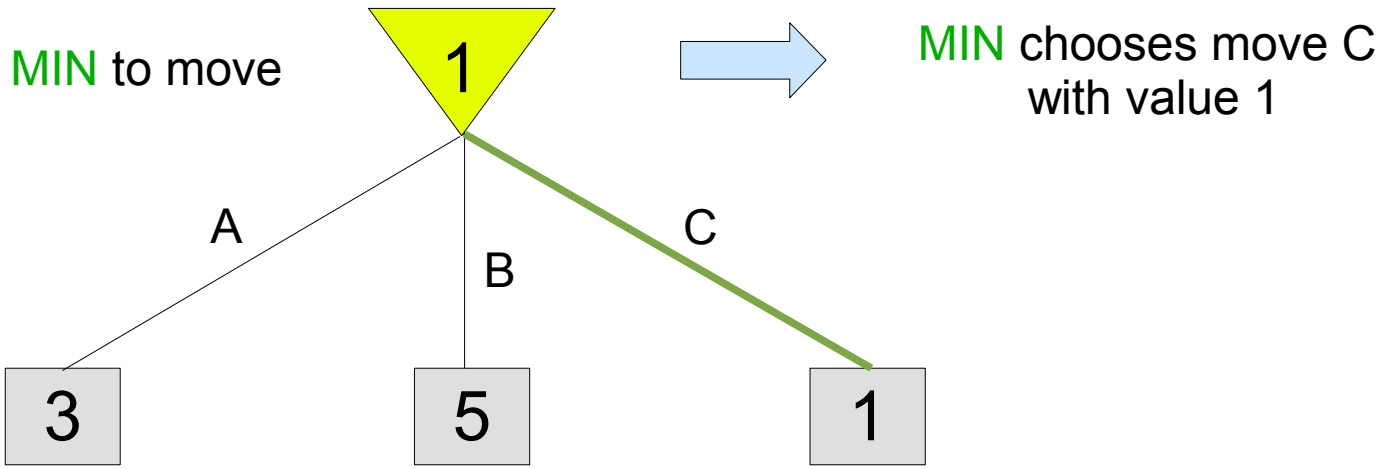
Optimal strategies

- Perfect play for deterministic, perfect-information games
  - Find the best strategy for MAX assuming an infallible MIN opponent.
  - Assumption: Both players play optimally
- Basic idea:
  - the terminal positions are evaluated form MAX's point of view
  - MAX player tries to maximize the evaluation of the position



Optimal strategies

- Perfect play for deterministic, perfect-information games
  - Find the best strategy for MAX assuming an infallible MIN opponent.
  - Assumption: Both players play optimally
- Basic idea:
  - the terminal positions are evaluated form MAX's point of view
  - MAX player tries to maximize the evaluation of the position
  - MIN player tries to minimize MAX's evaluation of the position

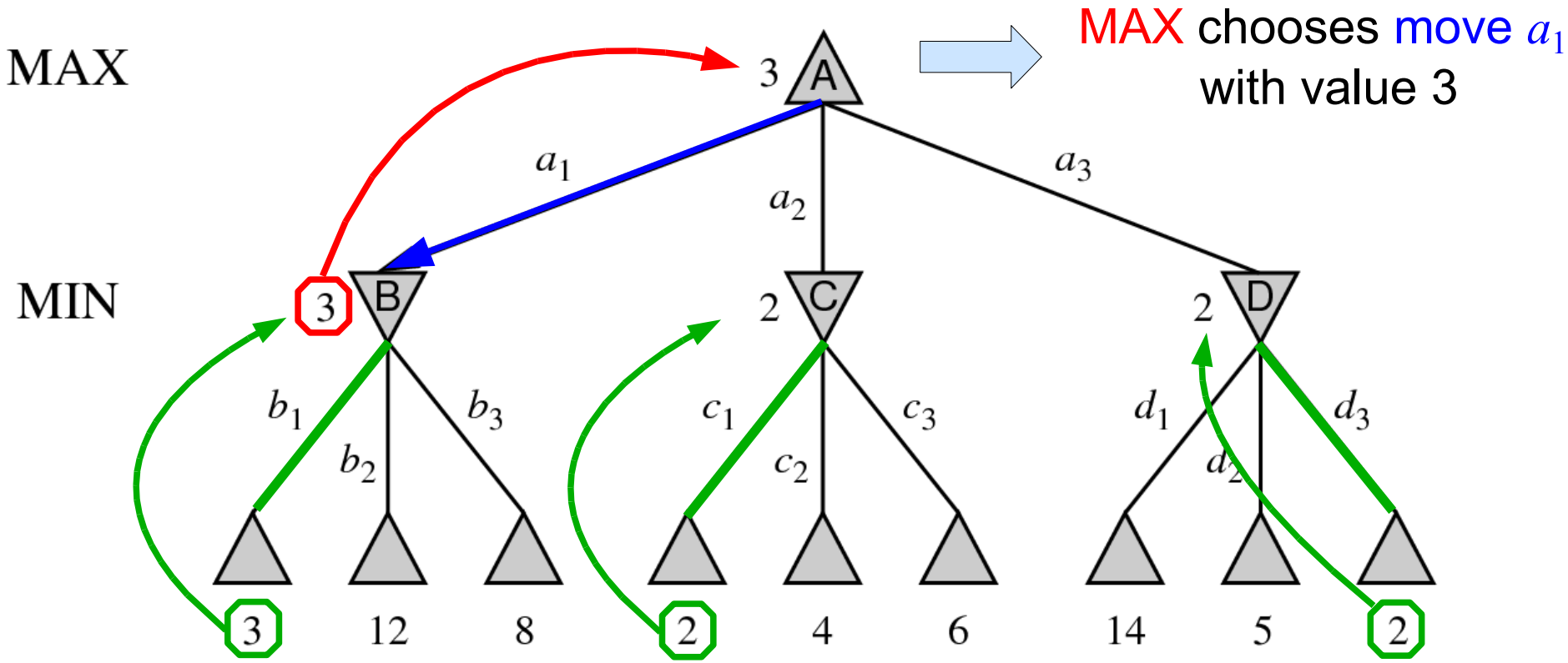


Optimal strategies

- Perfect play for deterministic, perfect-information games
  - Find the best strategy for MAX assuming an infallible MIN opponent.
  - Assumption: Both players play optimally
- Basic idea:
  - the terminal positions are evaluated form MAX's point of view
  - MAX player tries to maximize the evaluation of the position
  - MIN player tries to minimize MAX's evaluation of the position
- Minimax value
  - Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

$$\mathrm{MINIMAX}(n) = \begin{cases} \mathrm{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in SUCCESSORS(n)} \mathrm{MINIMAX}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in SUCCESSORS(n)} \mathrm{MINIMAX}(s) & \text{if } n \text{ is a MIN node} \end{cases}$$

Depth-Two Minimax Search Tree



Minimax maximizes the worst-case outcome for MAX.

Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    v ← MAX-VALUE(state)
    return action a which has value v and a, s is in SUCCESSORS(state)
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
    return v
```

NegaMax Formulation

- The minimax algorithm can be reformulated in a simpler way
  - for evaluation functions that are symmetric around 0 (zero-sum)
- Basic idea:
  - Assume that evaluations in all nodes (and leaves) are always from the point of view of the player that is to move
    - the MIN-player now also maximizes its value
  - As the values are zero-sum, the value of a position for MAX is equal to minus the value of position for MIN

    → NegaMax = Negated Maximum

$$\text{NEGAMAX}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in SUCCESSORS(n)}(-\text{NEGAMAX}(s)) & \text{if } n \text{ is an internal node} \end{cases}$$

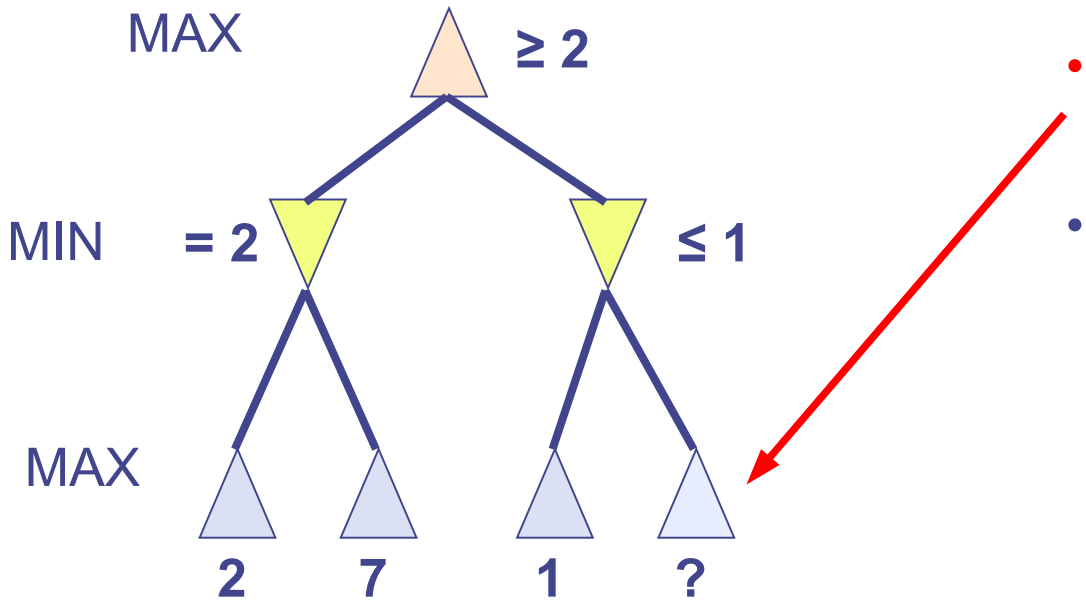
Properties of Minimax Search

- Completeness
  - Yes, if tree is finite
    - e.g., chess guarantees this through separate rules (3-fold repetition or 50 moves w/o irreversible moves are draw)
  - Note that there might also be finite solutions in infinite trees
- Optimality
  - Yes, if the opponent also plays optimally
    - If not, there might be better strategies (→ *opponent modeling*)
- Time Complexity
  - $O(b^m)$
  - has to search all nodes up to maximum depth (i.e., until terminal positions are reached)
  - for many games unfeasible (e.g., chess: $b \approx 35, m \approx 60$)
- Space Complexity
  - search proceeds depth-first → $O(m \cdot b)$

Alpha-Beta Pruning

- Minimax needs to search an exponential number of states
- Possible solution:
  - Do not examine every node
  - remove nodes that can not influence the final decision

> *"If you have an idea that is surely bad, don't take the time to see how truly awful it is."* -- Pat Winston



- We don't need to compute the value at this node.
- No matter what it is, it can't affect the value of the root node.

Alpha-Beta Pruning

Maintains two values $[\alpha,\beta]$ for all nodes in the current path

- Alpha:
  - the value of the best choice (i.e., highest value) for the MAX player at any choice node for MAX in the current path

  → MAX can obtain a value of at least α
- Beta:
  - the value of the best choice (i.e., lowest value) for the MIN player at any choice node for MIN in the current path

  → MIN can make sure that MAX obtains a value of at most β

The values are initialized with $[-\infty, +\infty]$

Alpha-Beta Pruning

Alpha and Beta are used for pruning the search tree:

- Alpha-Cutoff:
  - if we find a move with value $\leq \alpha$ at a MIN node, we do not examine alternatives to this move
  - we already know that MAX can achieve a better result in a different variation
- Beta-Cutoff:
  - if we find a move with value $\geq \beta$ at a MAX node, we do not examine alternatives to this move
  - we already know that MIN can achieve a better result in a different variation

Alpha-Beta Algorithm

function ALPHA-BETA-DECISION(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty$, $+\infty$)
  return action *a* which has value *v* and *a*, *s* is in SUCCESSORS(*state*)

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a*, *s* in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

if TERMINAL-TEST(*state*) **return** UTILITY(*state*)
 $v \leftarrow +\infty$
for *a*, *s* in SUCCESSORS(*state*) **do**
 $v \leftarrow$ MIN(v,MAX-VALUE(s, α , β))
 if $v \leq \alpha$ **then return** v
 $\beta \leftarrow$ MIN(β ,v)
return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  same as MAX-VALUE but with roles of α, β reversed

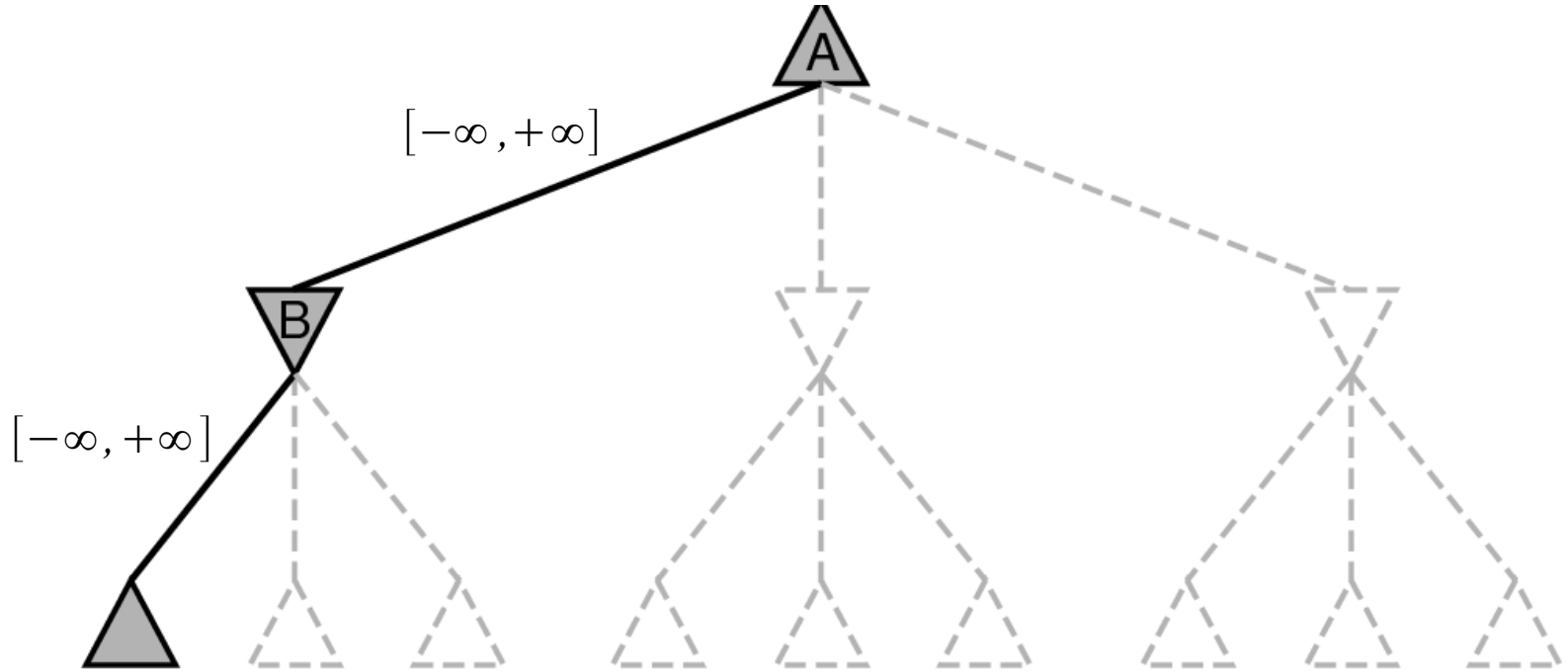
Alpha-Beta – NegaMax Formulation

```
int AlphaBeta ( position p; int α, β );
{                                   /* compute minimax value of position p */
        int a, t, i;
        determine successors p_1,...,p_w of p;
        if ( w == 0 )
                return ( Evaluate(p) );                    /* leaf node */
        a = α;

        for ( i = 1; i <= w; i++ ) {
                t = -AlphaBeta ( p_i, - β, -a );

                a = max( a, t );
                if ( a >= β )
                        return ( a );                      /* cut-off */

        }
        return ( a );
}
```

Recursive call with negated window
$[\alpha_{MIN}=-\beta_{MAX}, \beta_{MIN}=-\alpha_{MAX}]$
Note the negated return value!

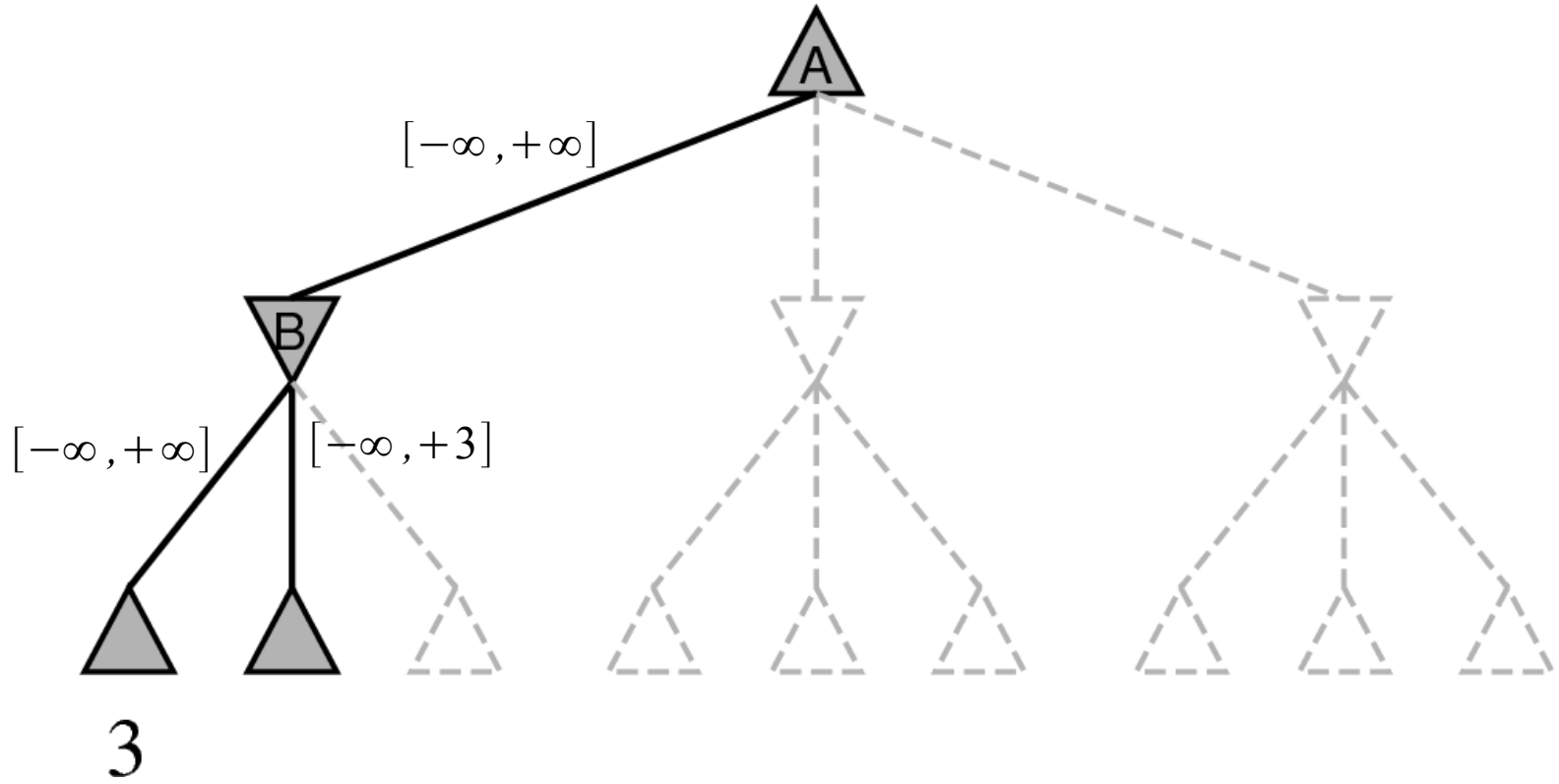
Example: Alpha-Beta

- The window is initialized with $[-\infty, +\infty]$
- search runs depth-first
- until first leaf is found (value 3)



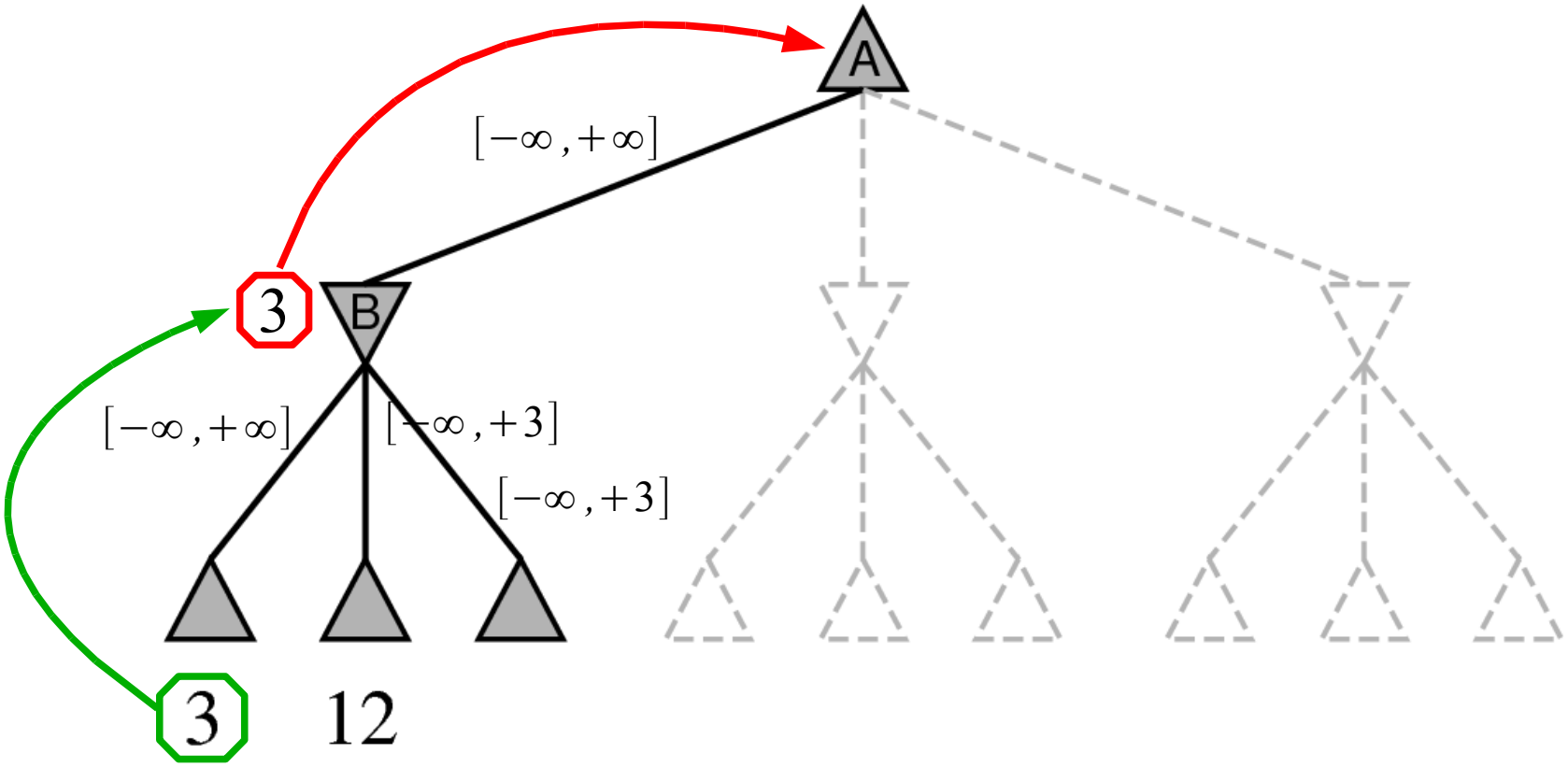
Example: Alpha-Beta

- It is followed that at node B, MIN can obtain at least 3
- Subsequent search below B is now initialized with $[-\infty, +3]$
- The leaf node (value 12) is worse for MIN (higher value for MAX)



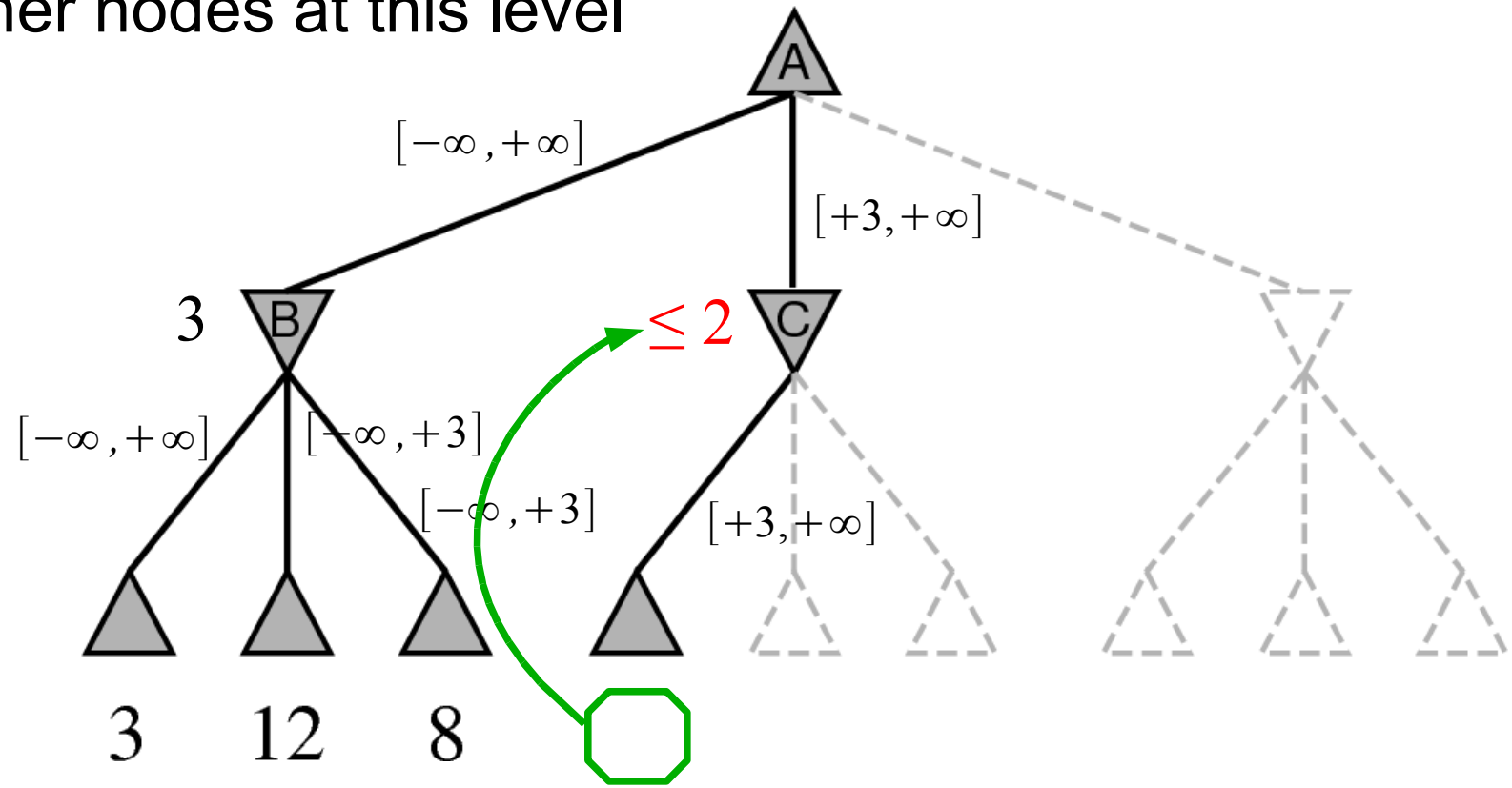
Example: Alpha-Beta

- The next leaf is also worse for MIN (value 8)
- Node B is now completed, and evaluated with 3
- The value is propagated up to A as a new minimum for MAX



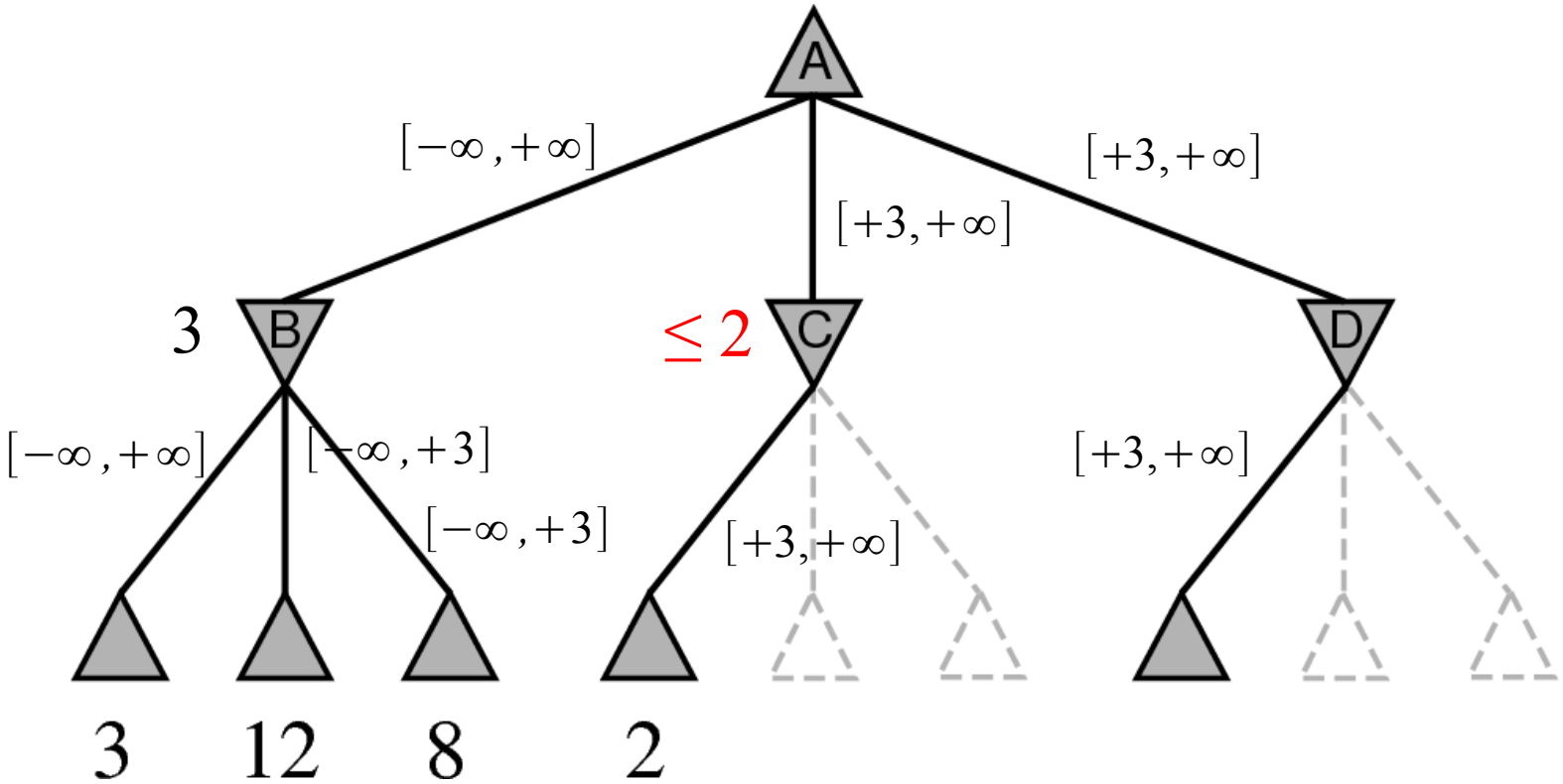
Example: Alpha-Beta

- Subsequent searches now know that MAX can achieve at least 3, i.e., the alpha-beta window is $[+3, +\infty]$
- The value 2 is found below the min node
- As the value is outside the window ($2 < 3$), we can prune all other nodes at this level



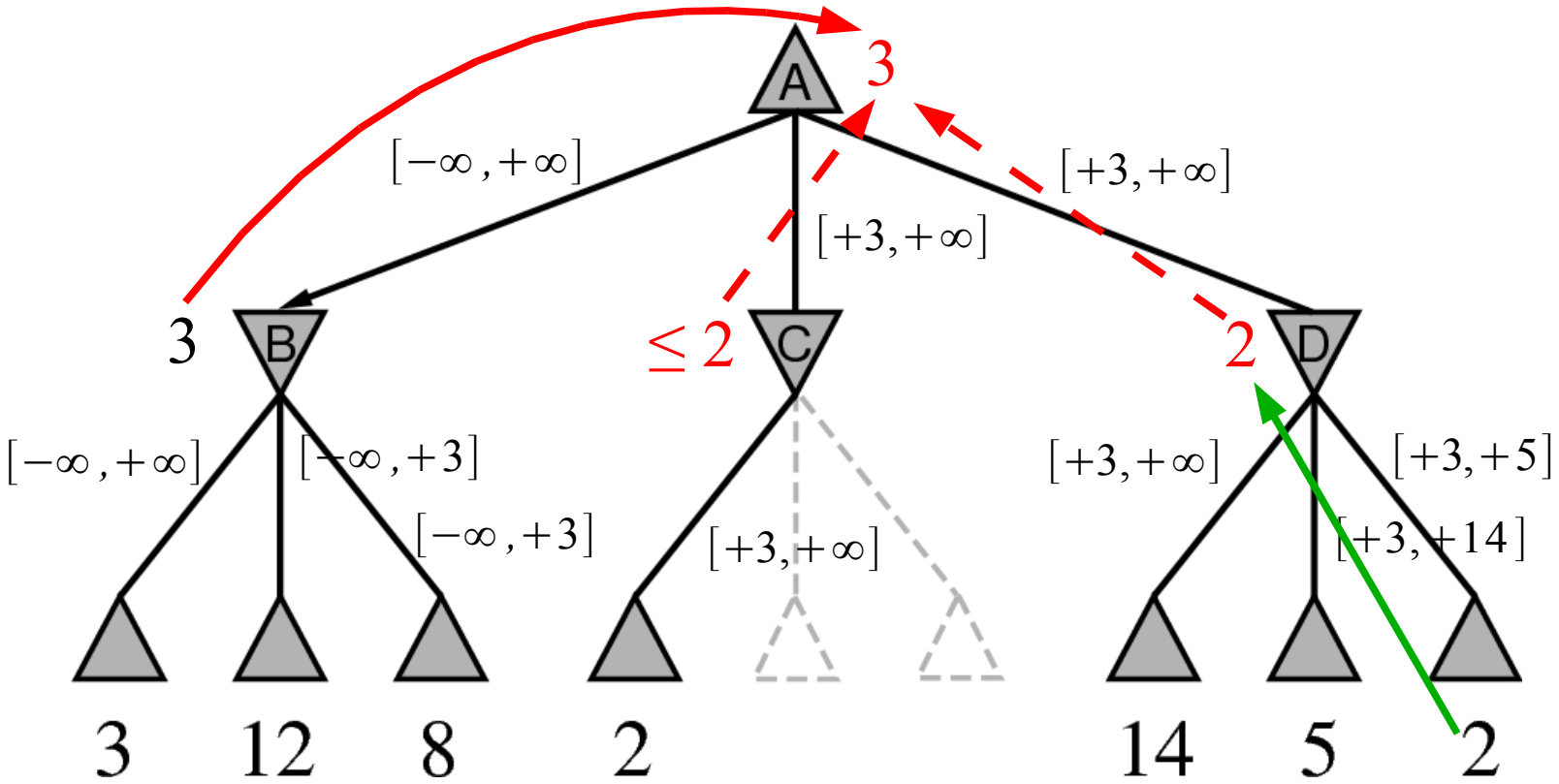
Example: Alpha-Beta

- Subsequent searches now know that MAX can achieve at least 3, i.e., the alpha-beta window is $[+3, +\infty]$
- The value 14 is found below the min node



Example: Alpha-Beta

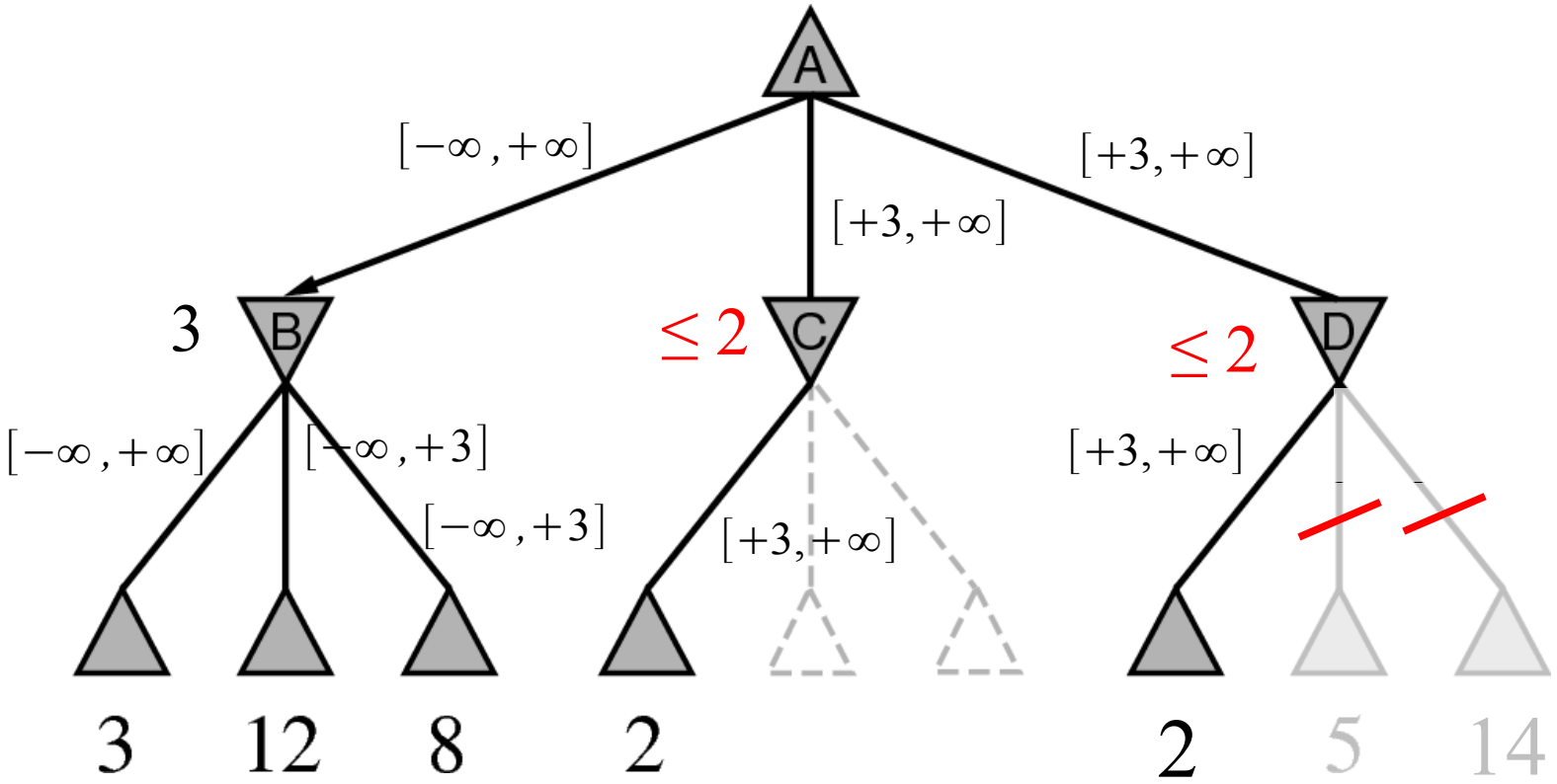
- The next search now knows that MAX can achieve at least 3 but MIN can hold him down to 14
- i.e., the alpha-beta window is $[+3, +14]$
- For the final node the window is $[+3, +5]$



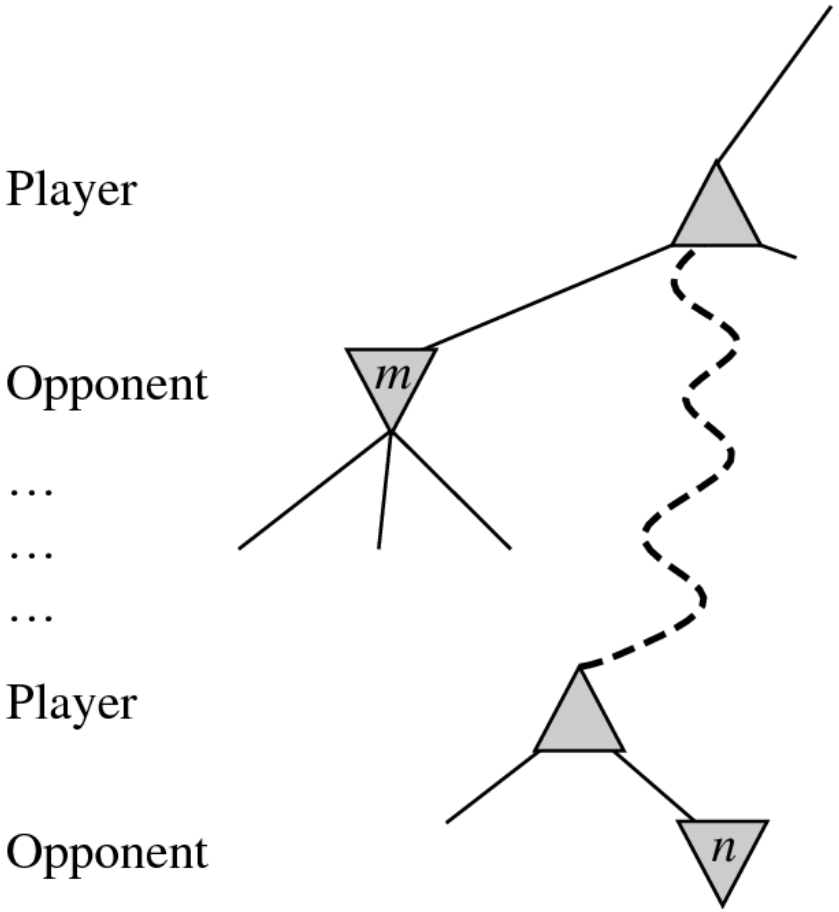
Evaluation Order

- Note that the order of the evaluation of the nodes is crucial
- e.g., if in node D, the node with evaluation 2 is seached first, another cutoff would have been possible

→ good move order is crucial for good performance



General Alpha-Beta Pruning

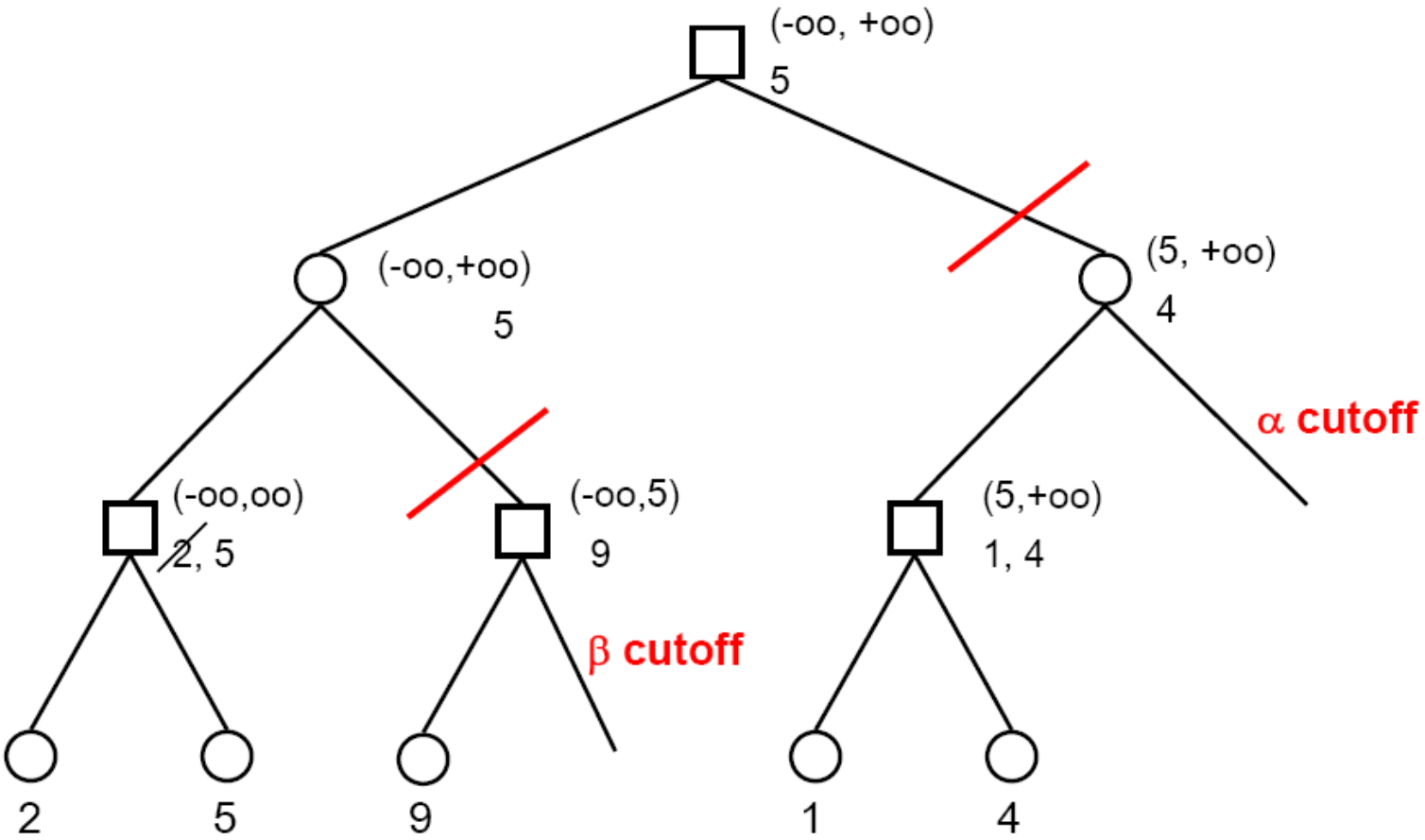


- Consider a node n somewhere in the tree
- If Player has a better choice
  - at parent node of n
  - or at any choice point further up

  n will never be reached in actual play.
- Hence we can prune n
  - as soon as we can establish that there is a better choice

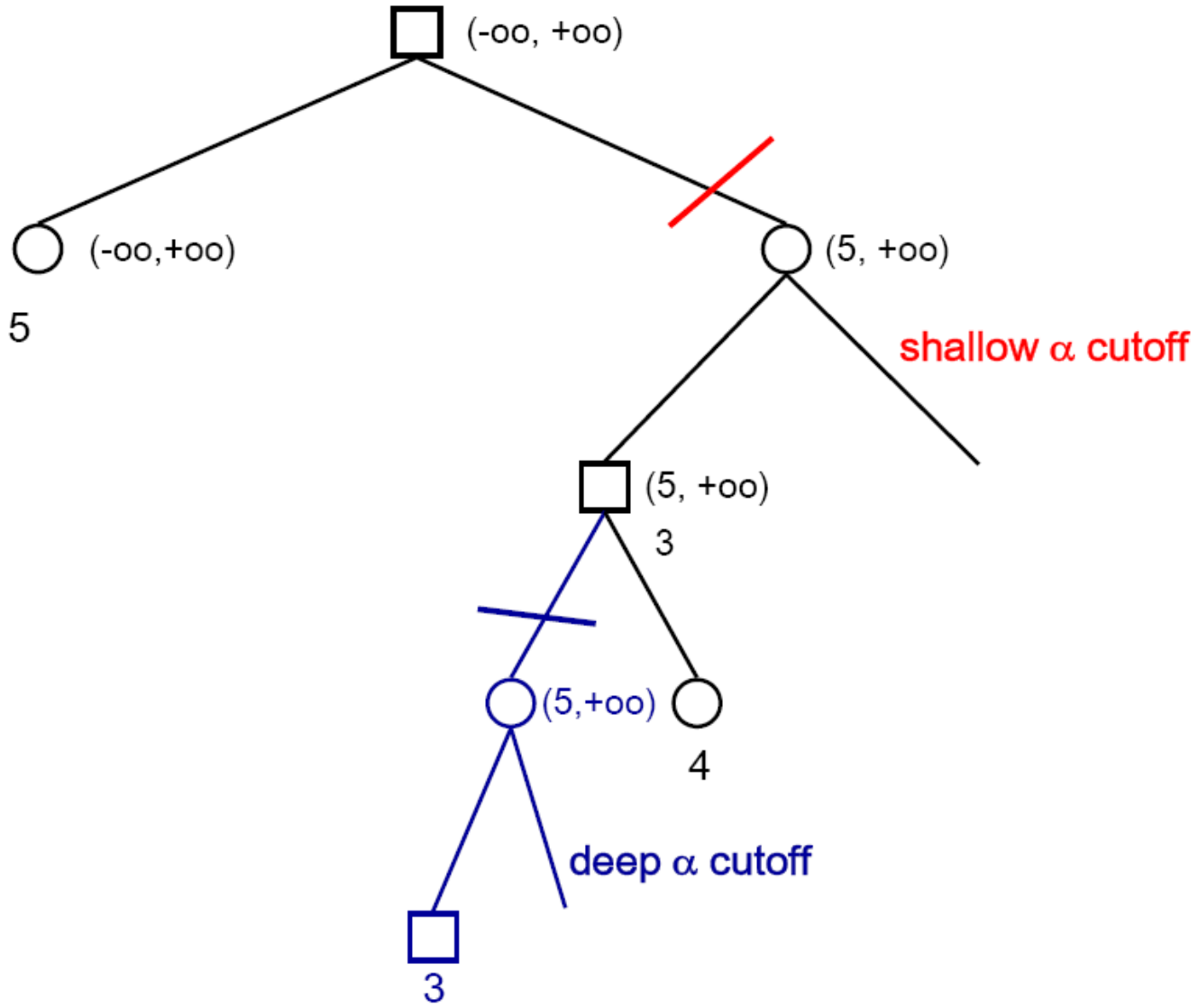
Alpha-Cutoff vs. Beta-Cutoff

- Of course, cutoffs can also occur at MAX-nodes

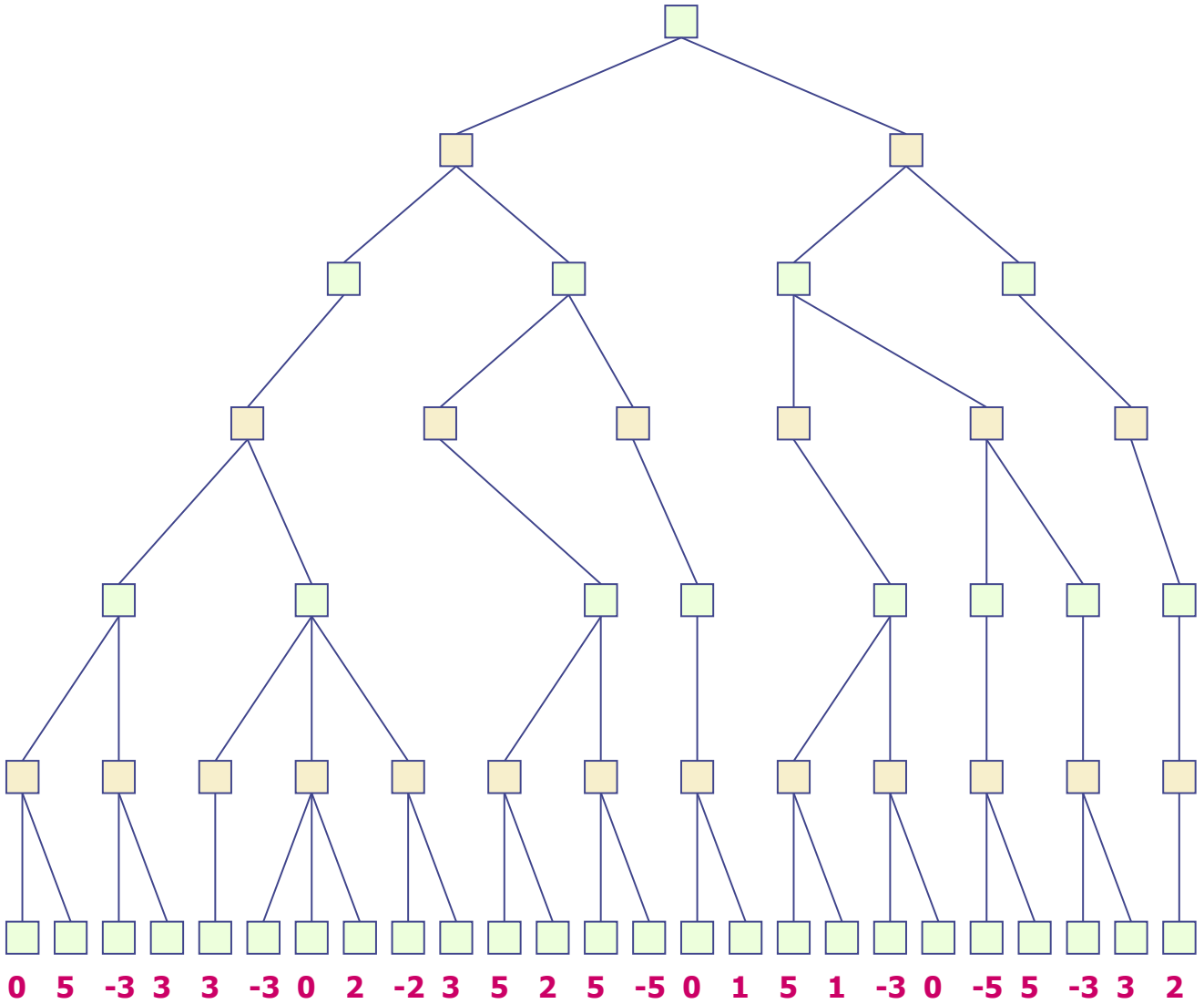


Shallow vs. Deep Cutoffs

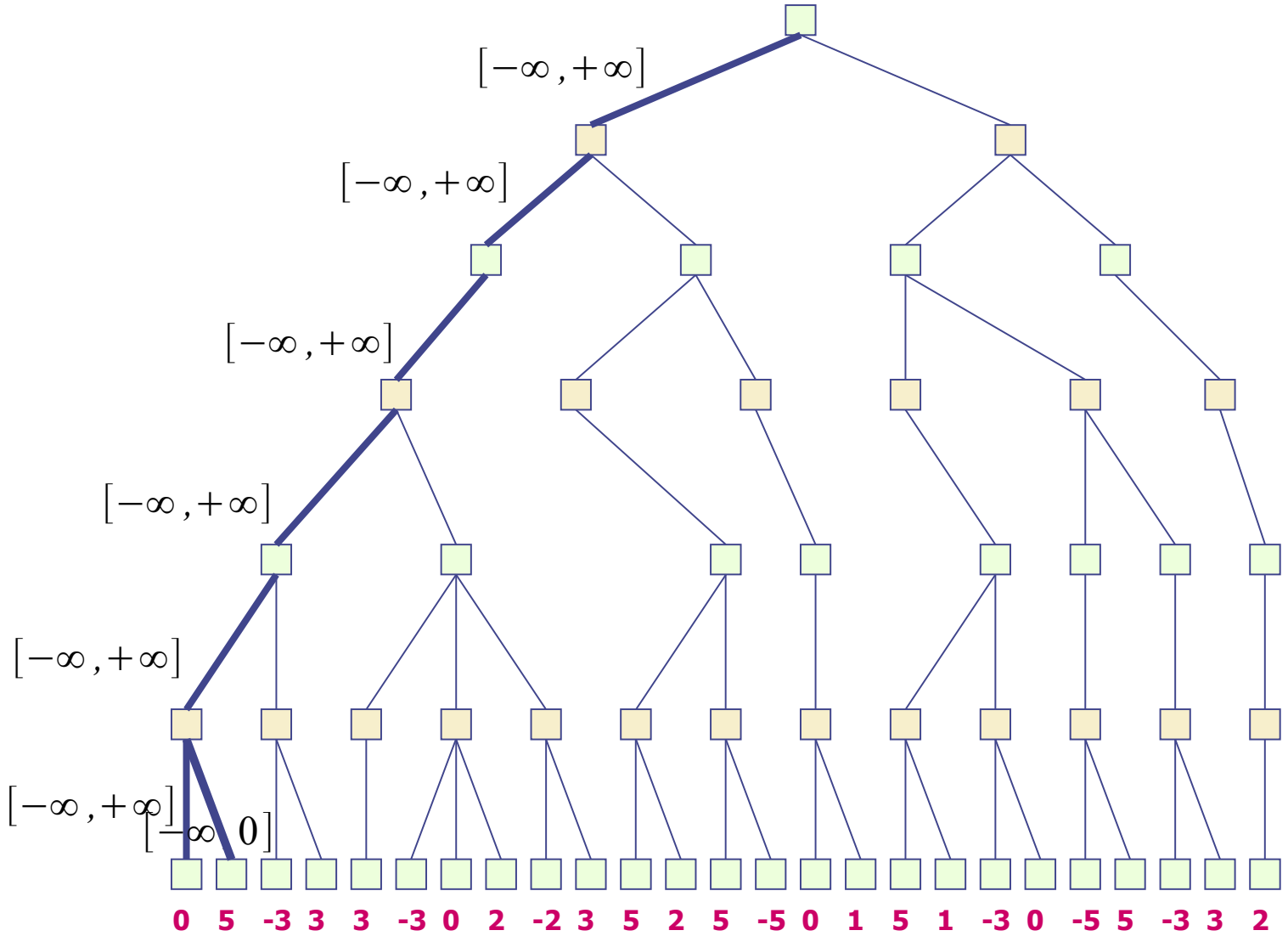
- Cutoffs may occur arbitrarily deep in (sub-)trees



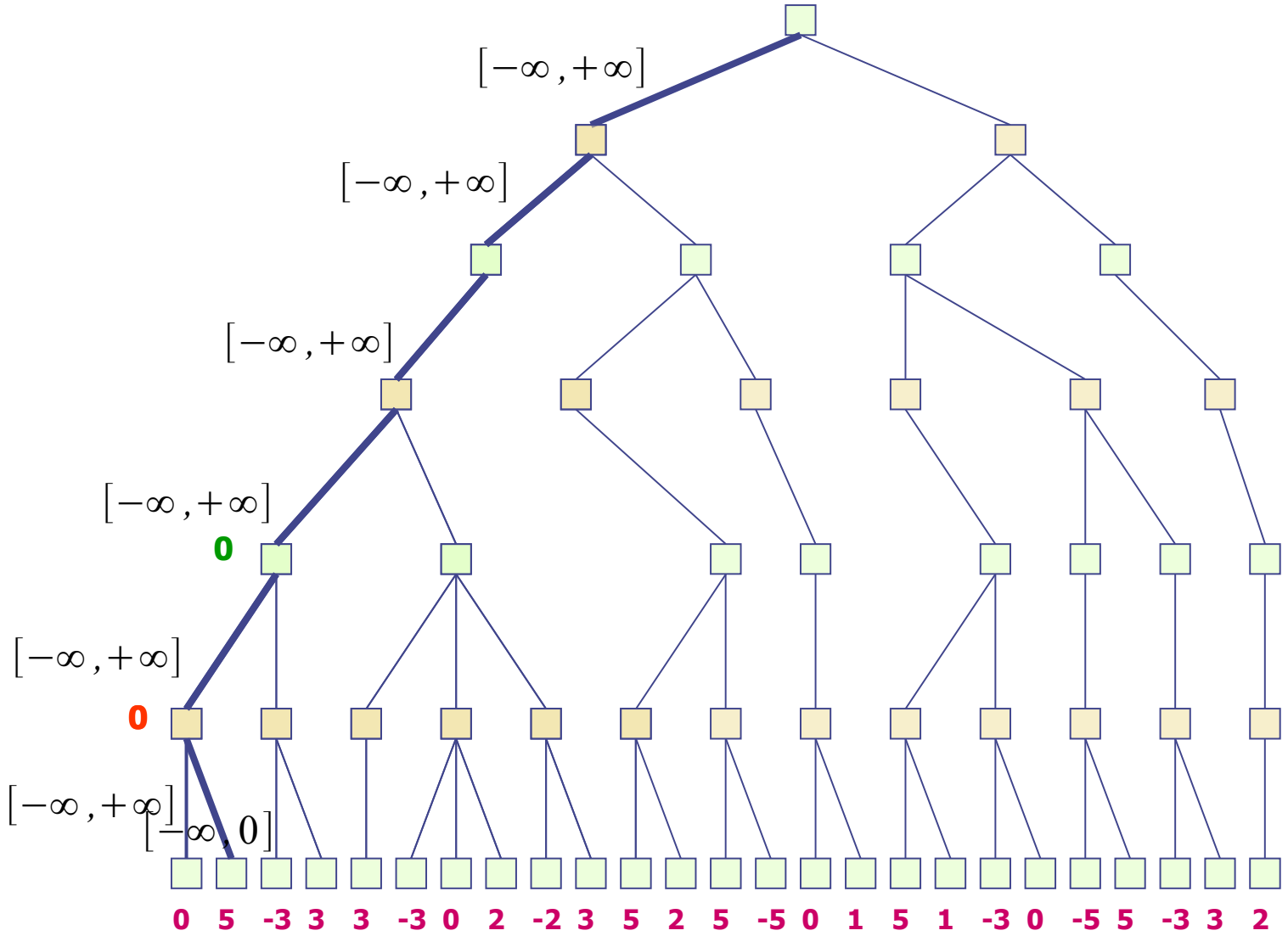
Alpha-Beta Example



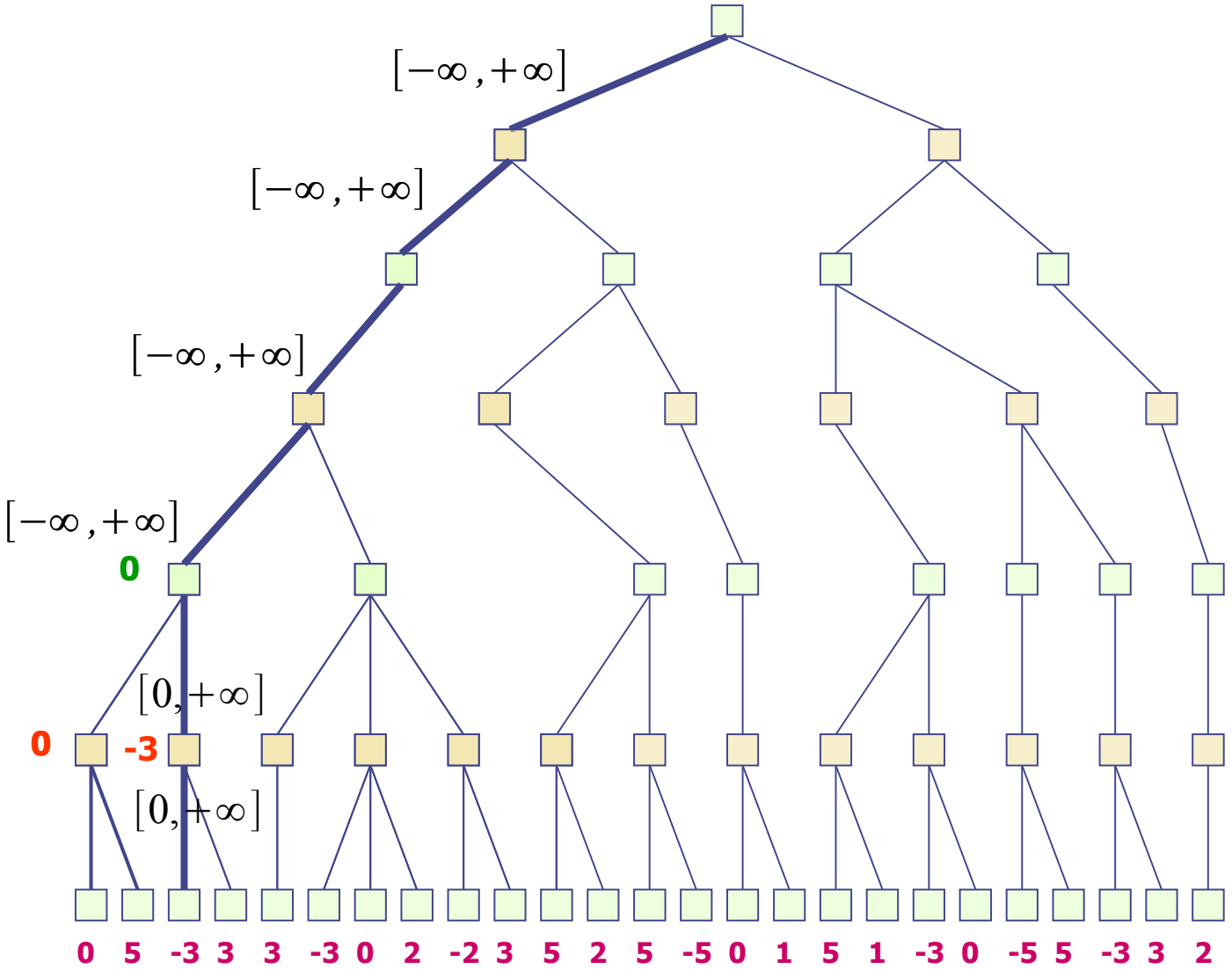
Alpha-Beta Example



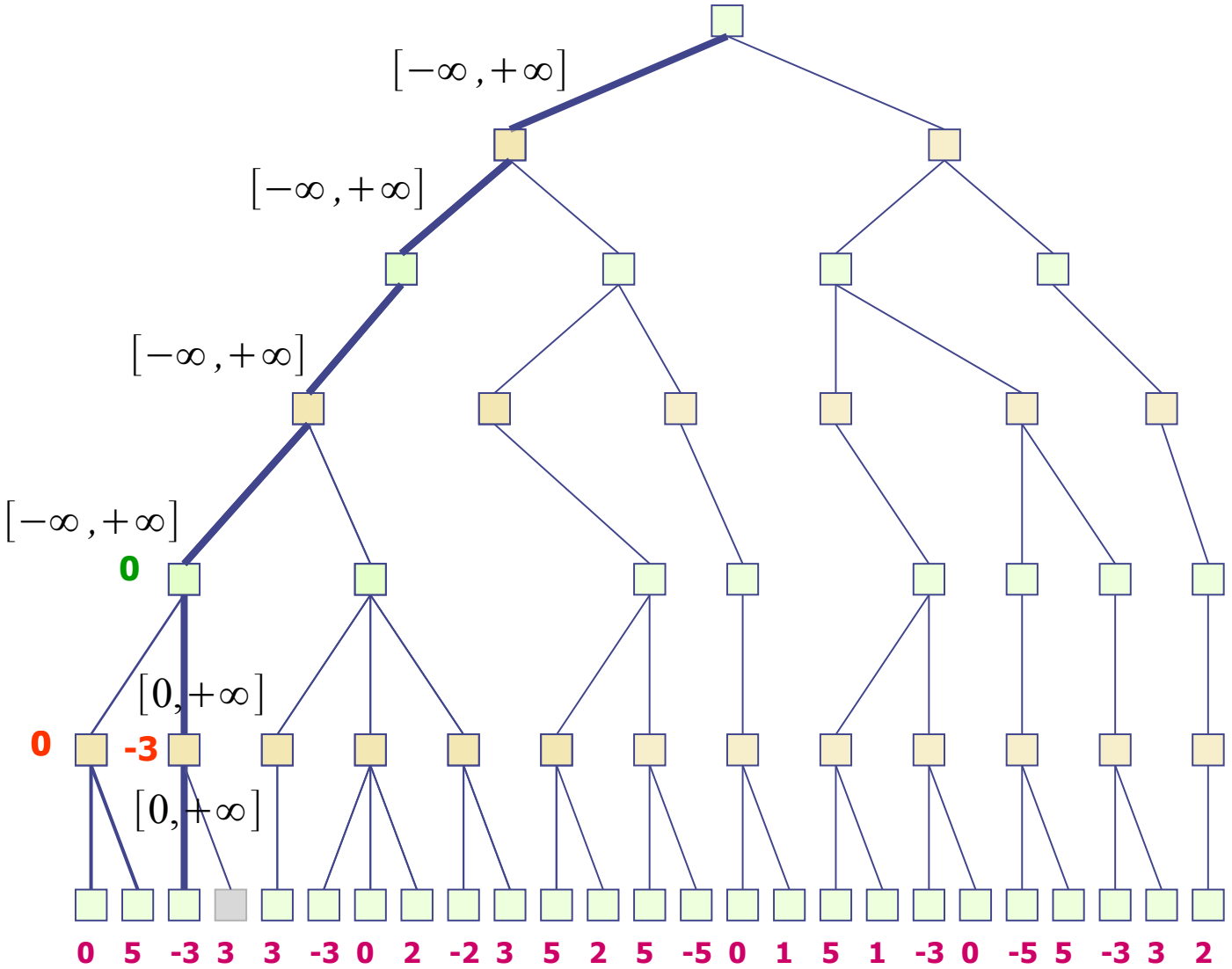
Alpha-Beta Example



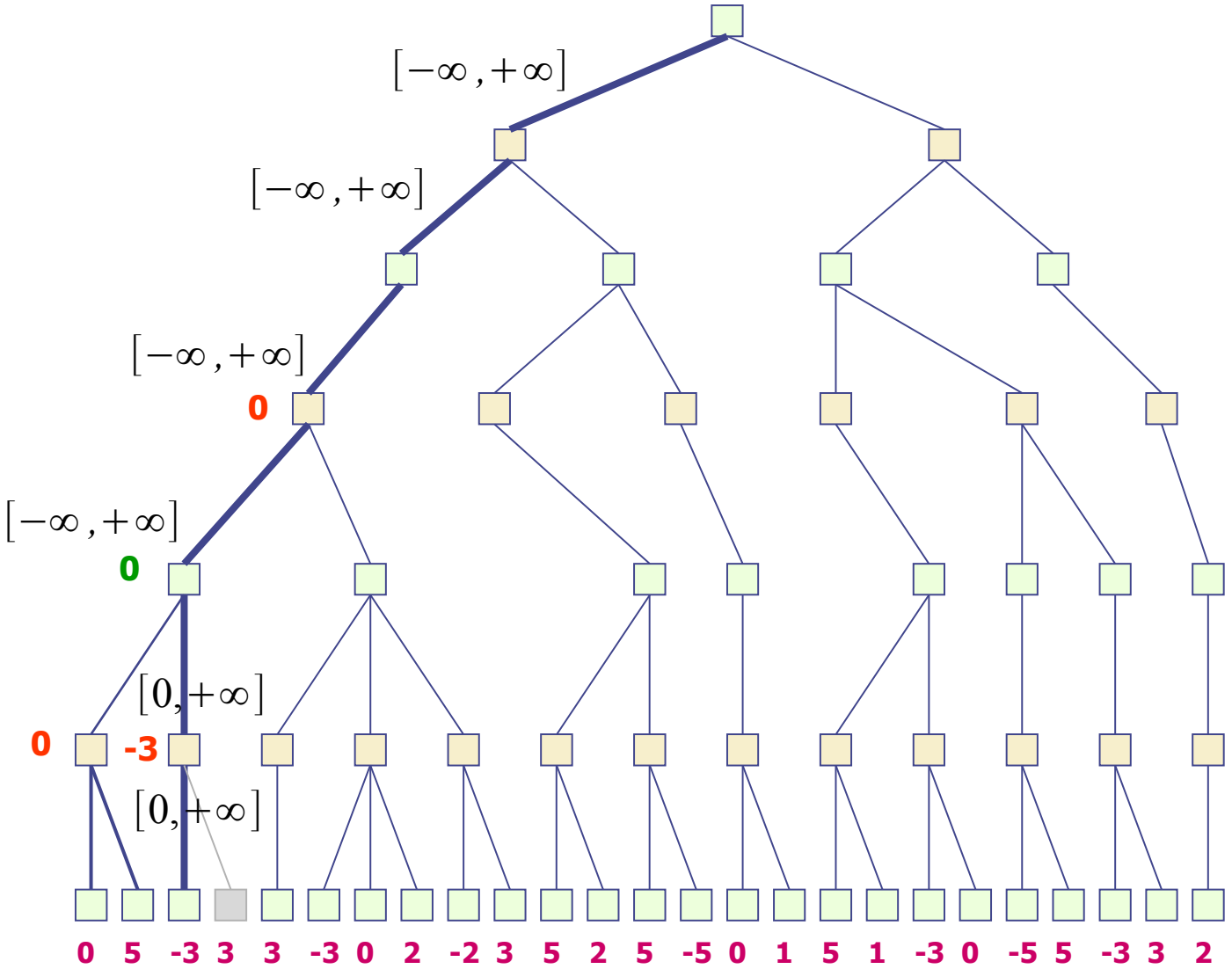
Alpha-Beta Example



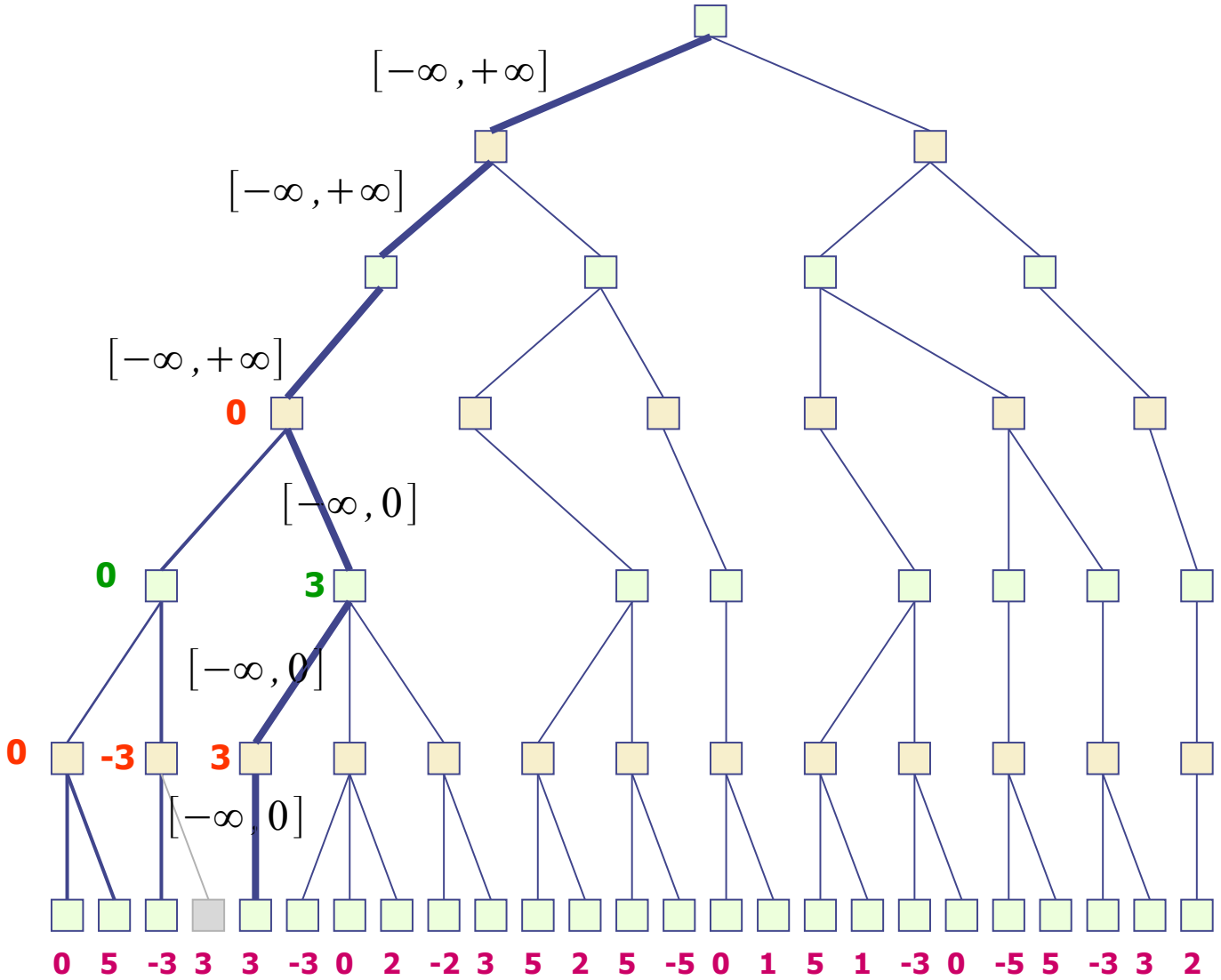
Alpha-Beta Example



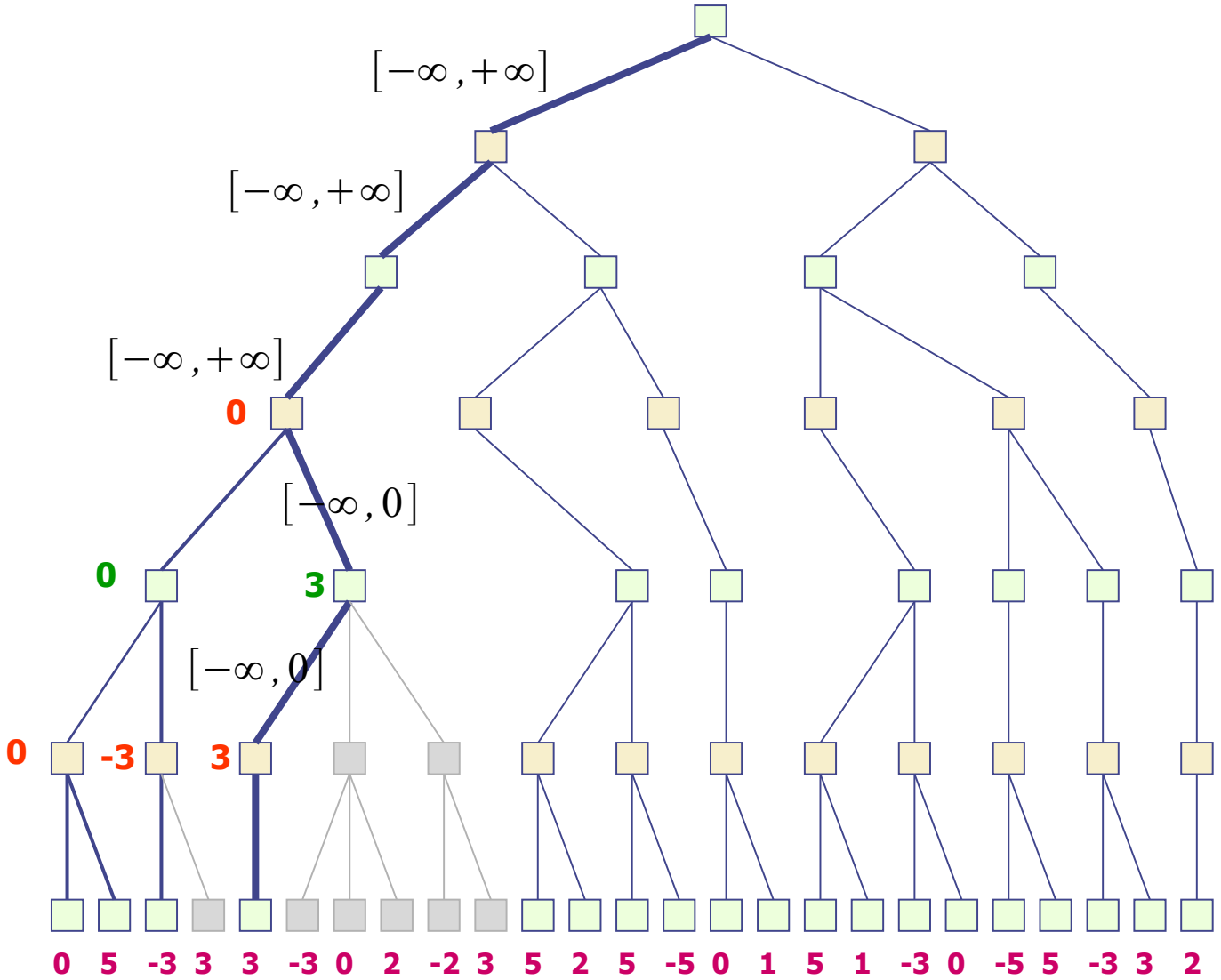
Alpha-Beta Example



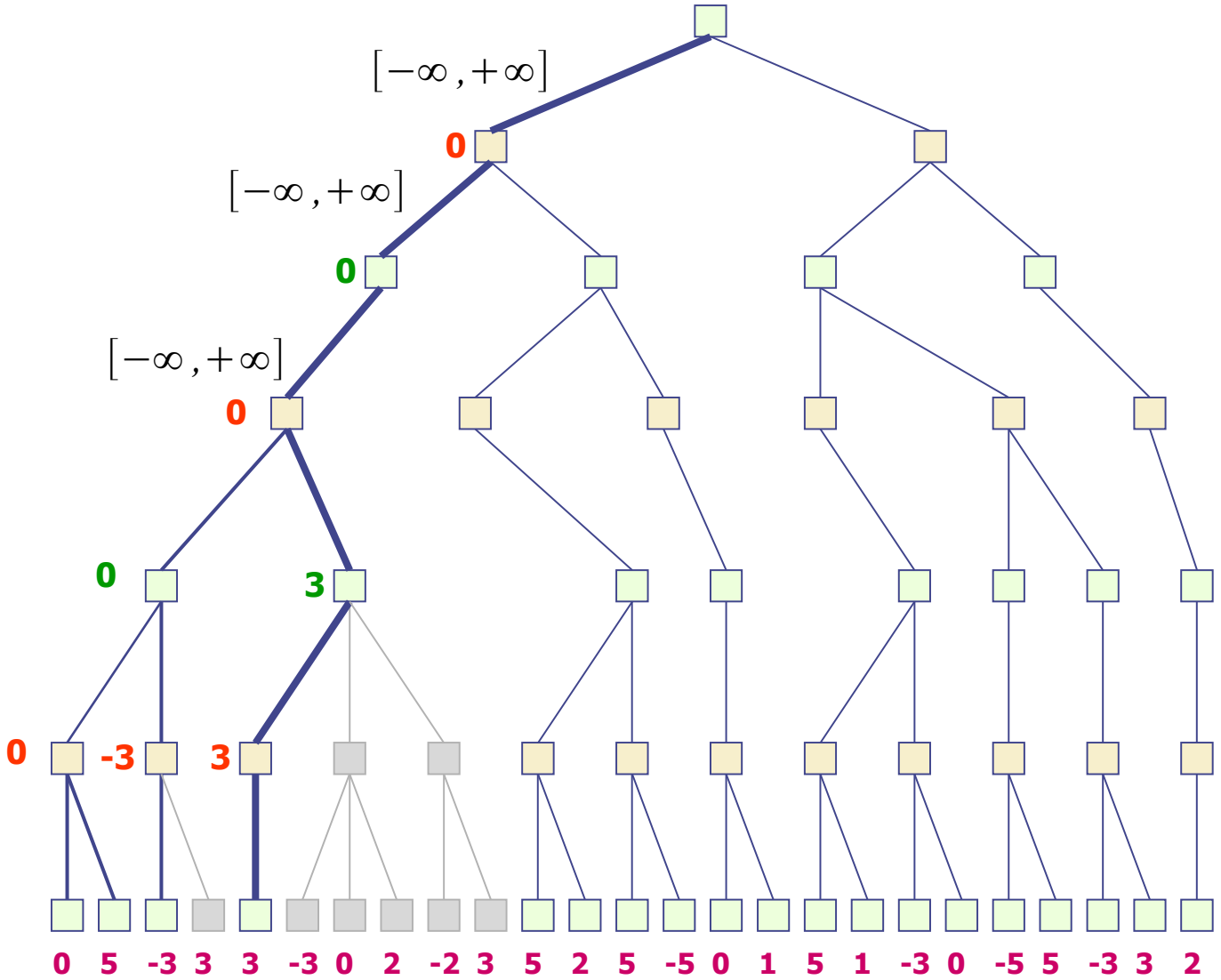
Alpha-Beta Example



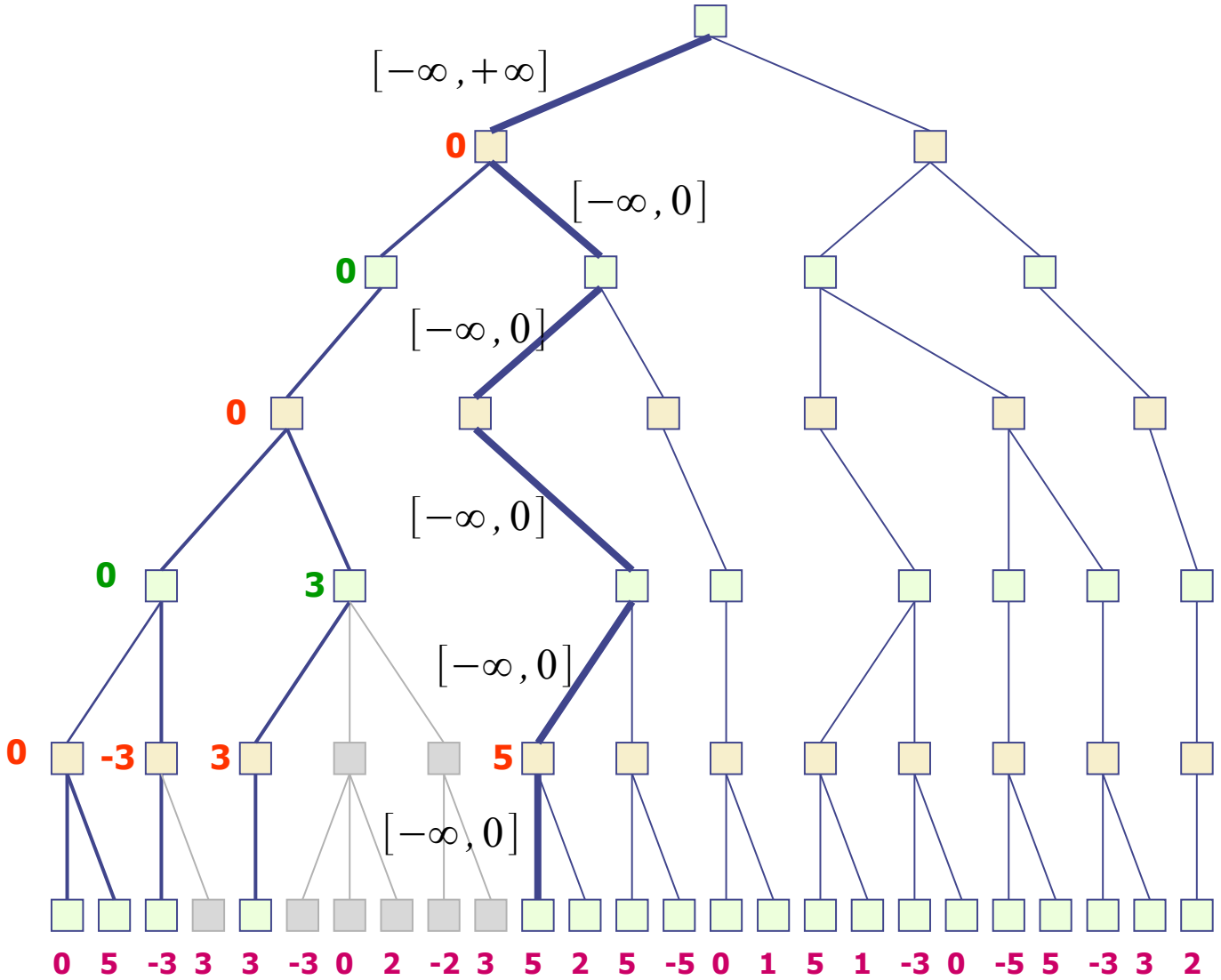
Alpha-Beta Example



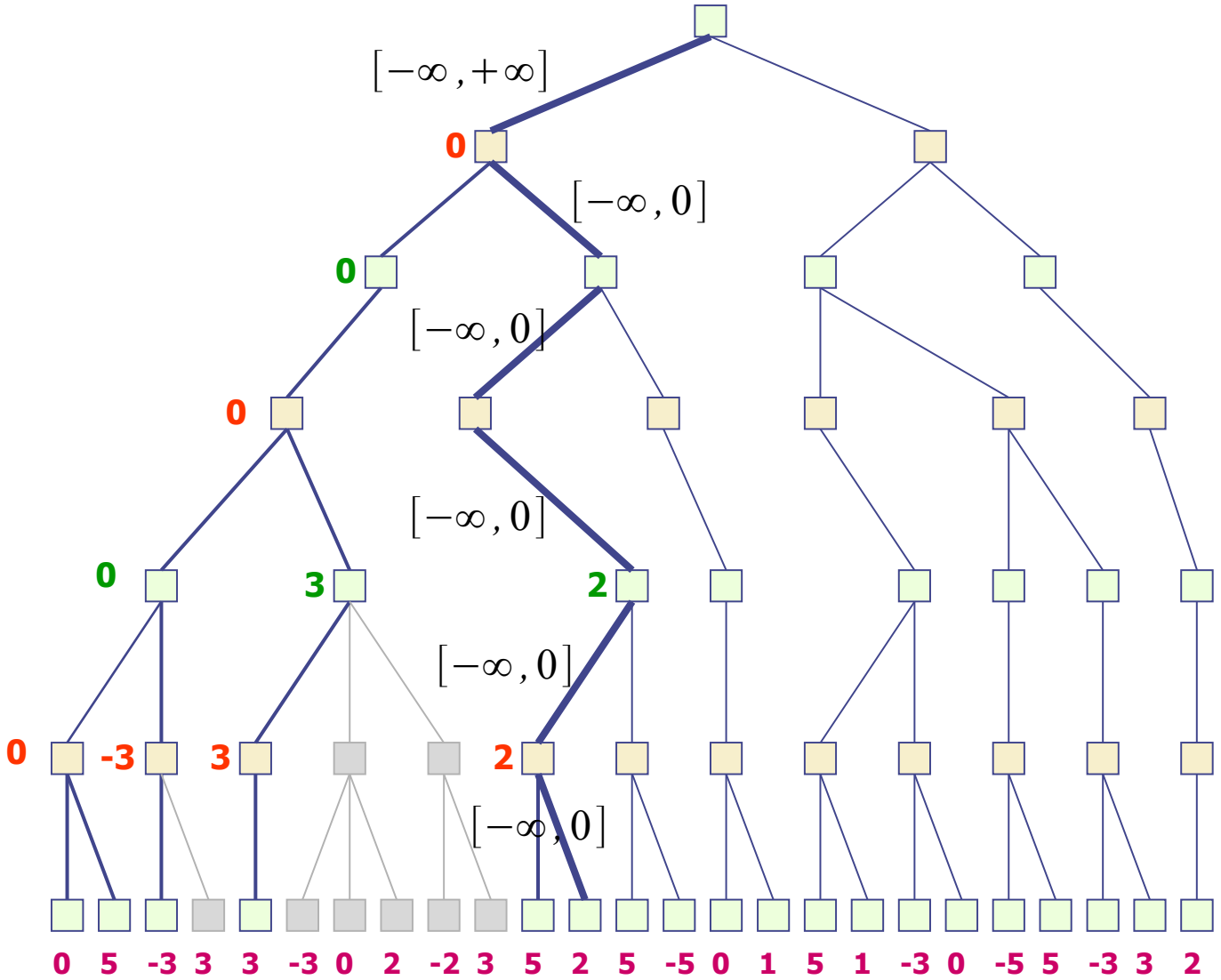
Alpha-Beta Example



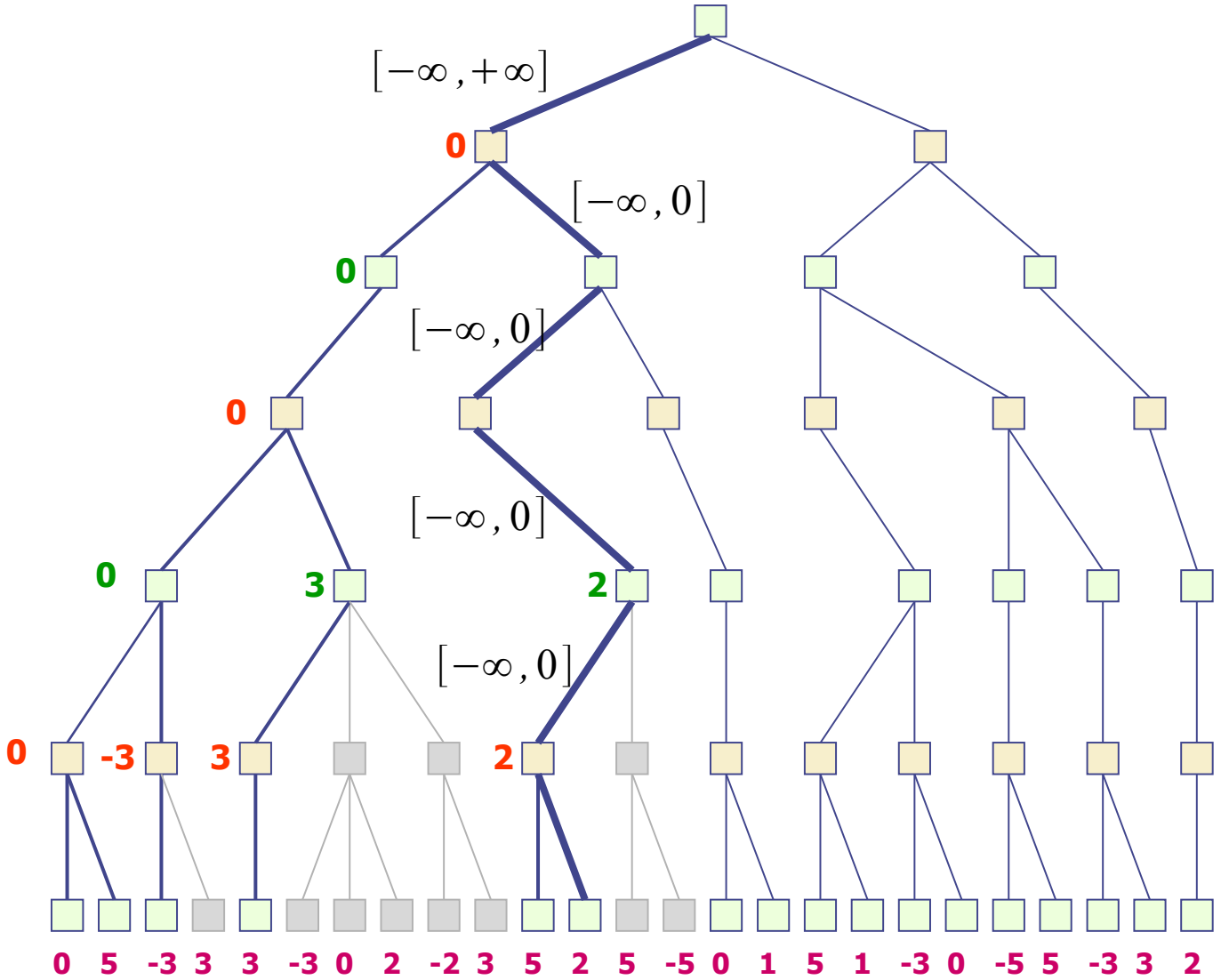
Alpha-Beta Example



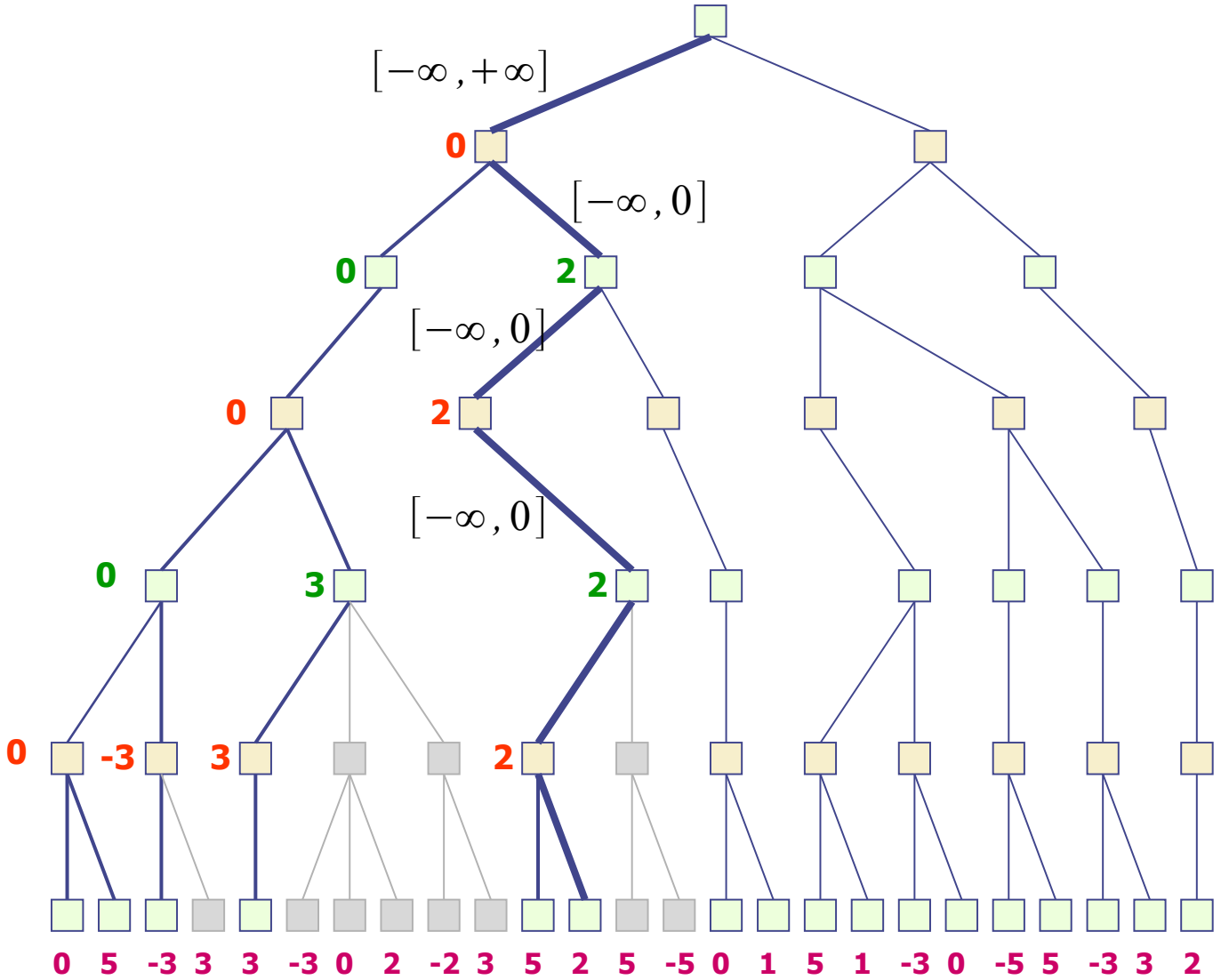
Alpha-Beta Example



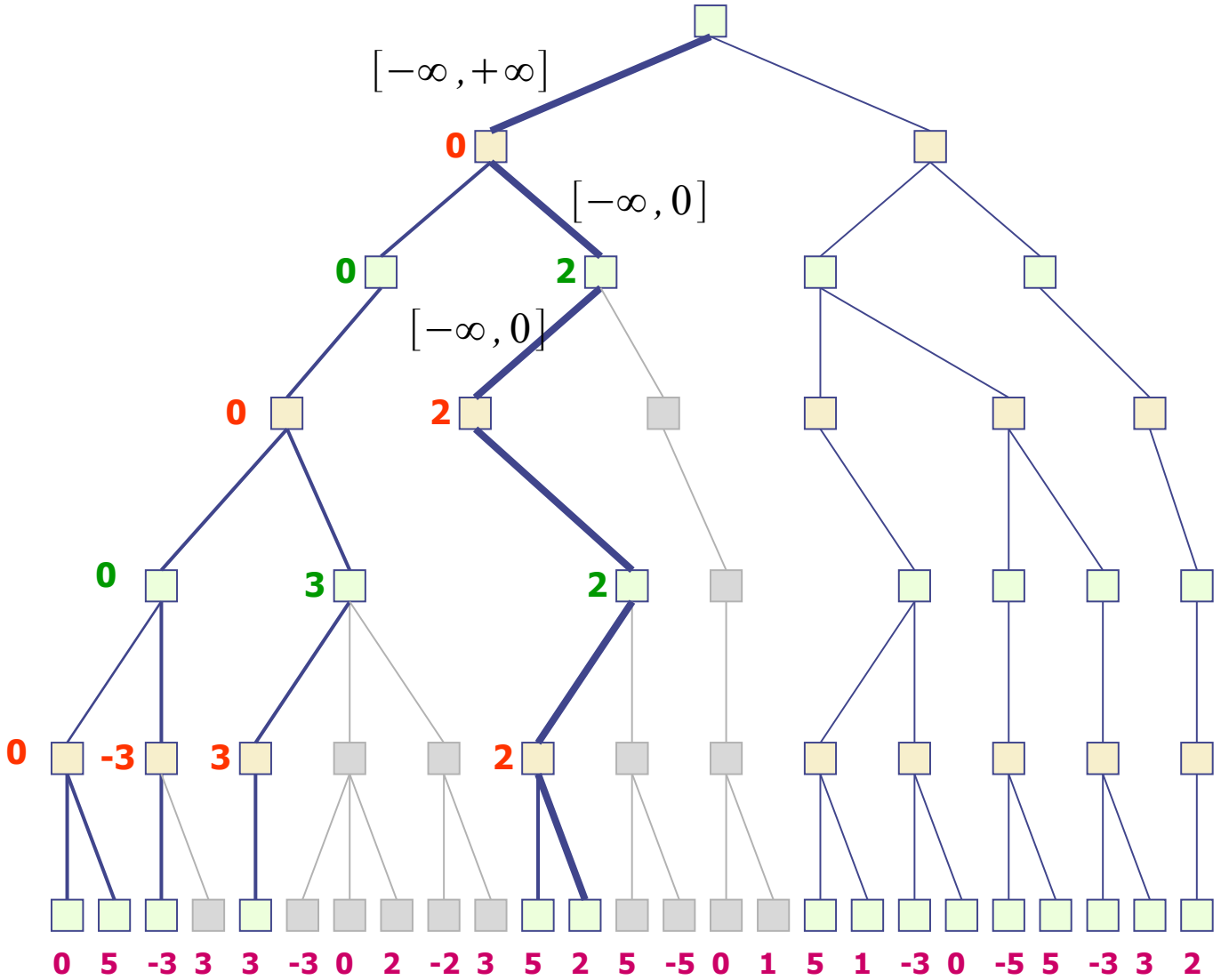
Alpha-Beta Example



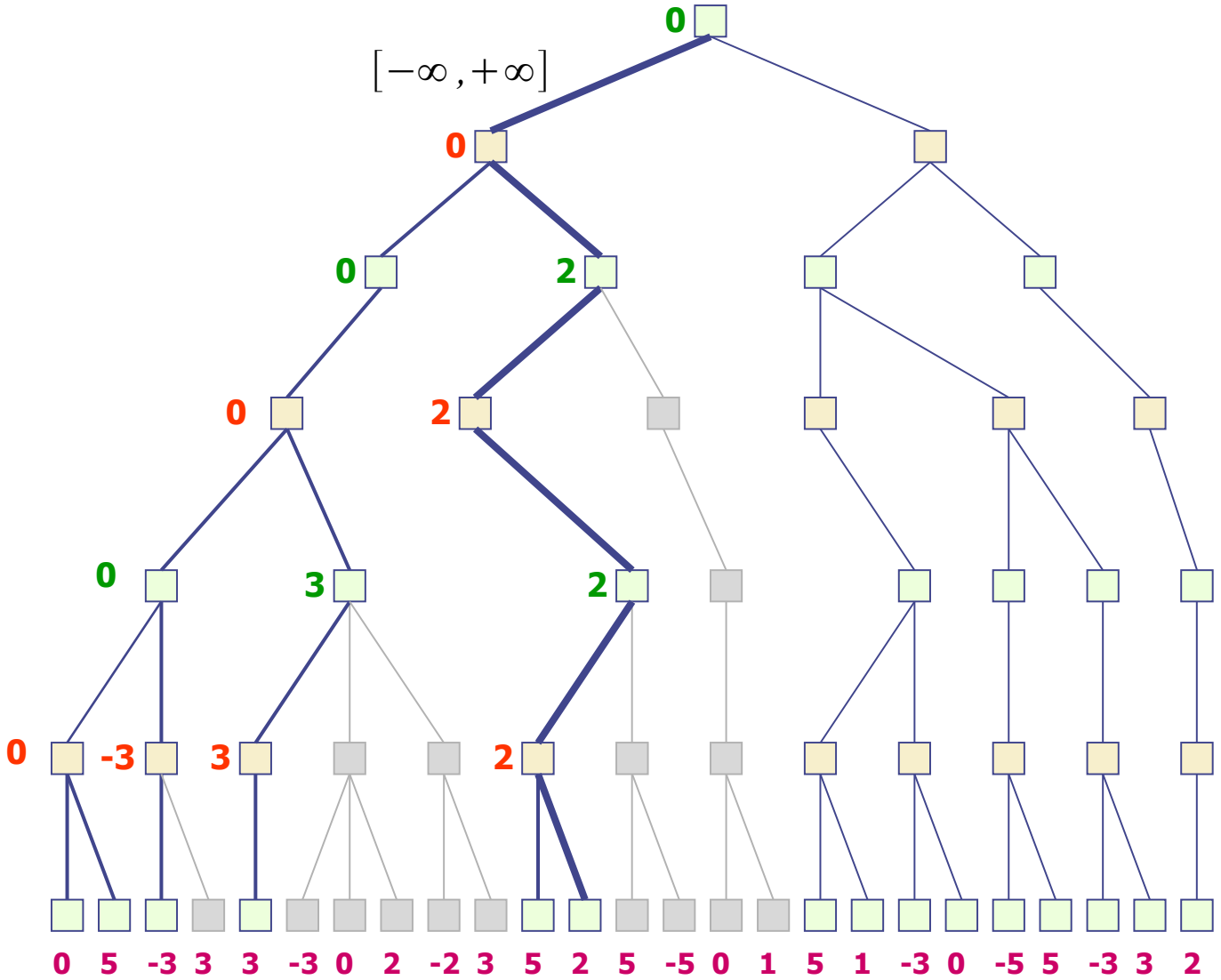
Alpha-Beta Example



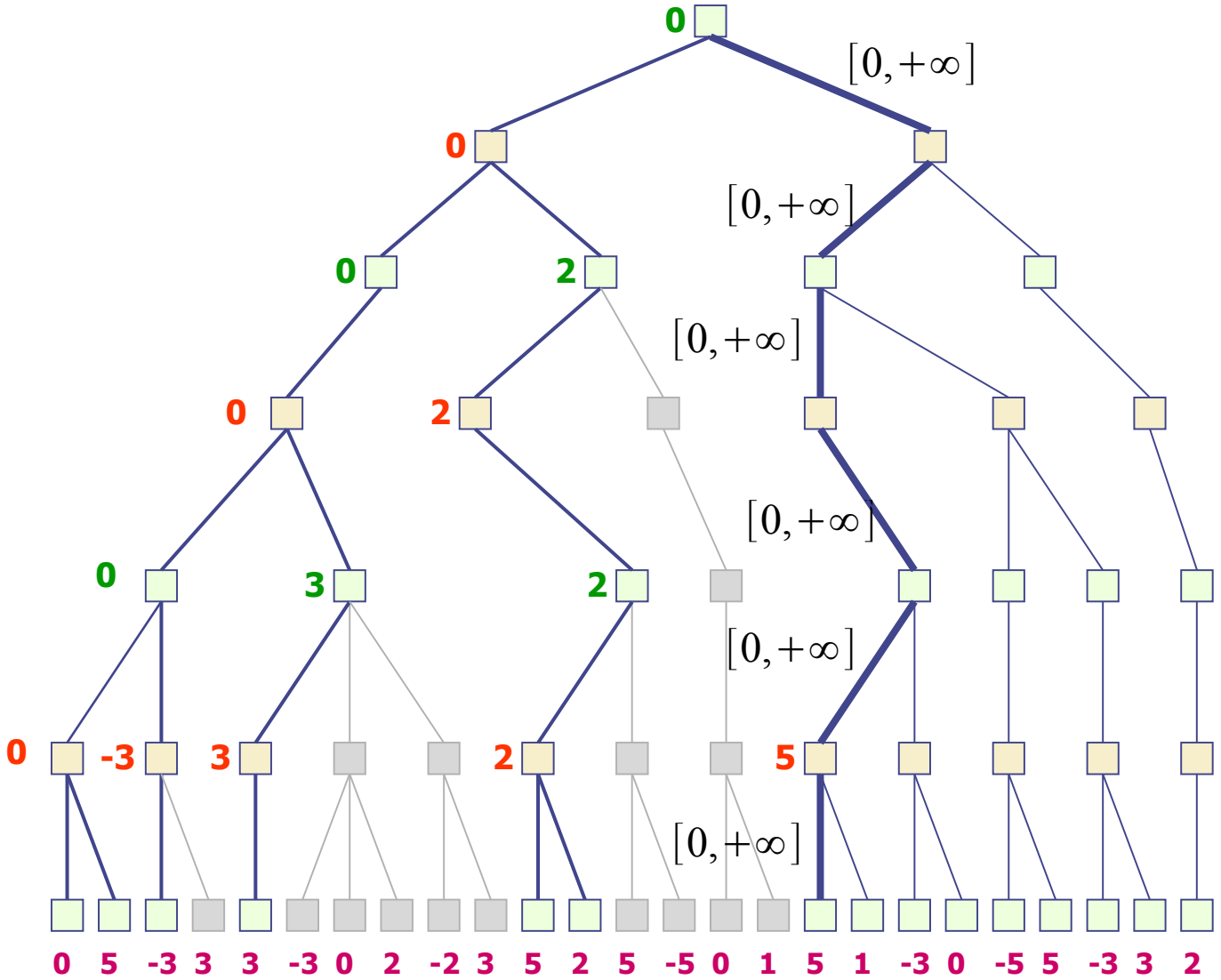
Alpha-Beta Example



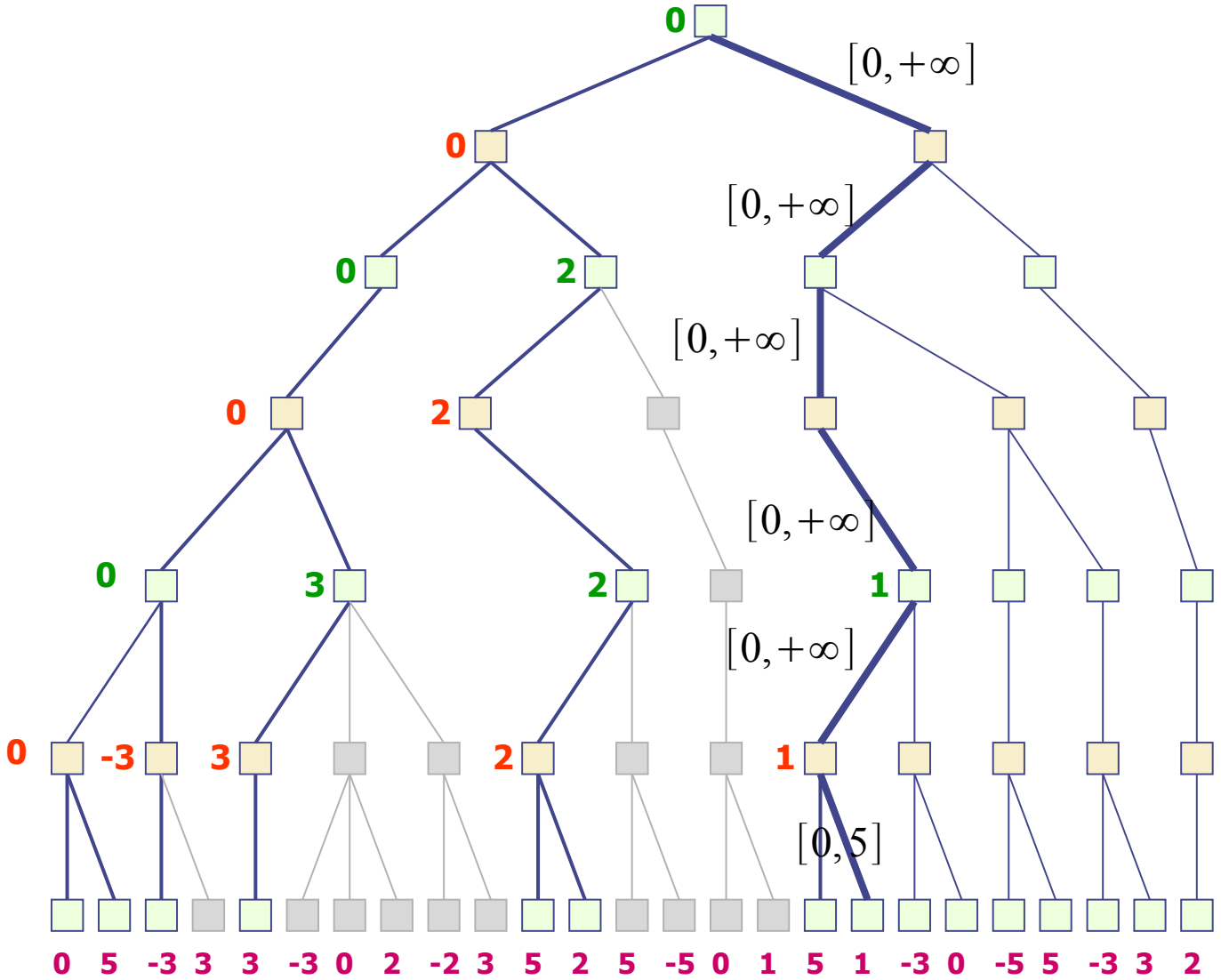
Alpha-Beta Example



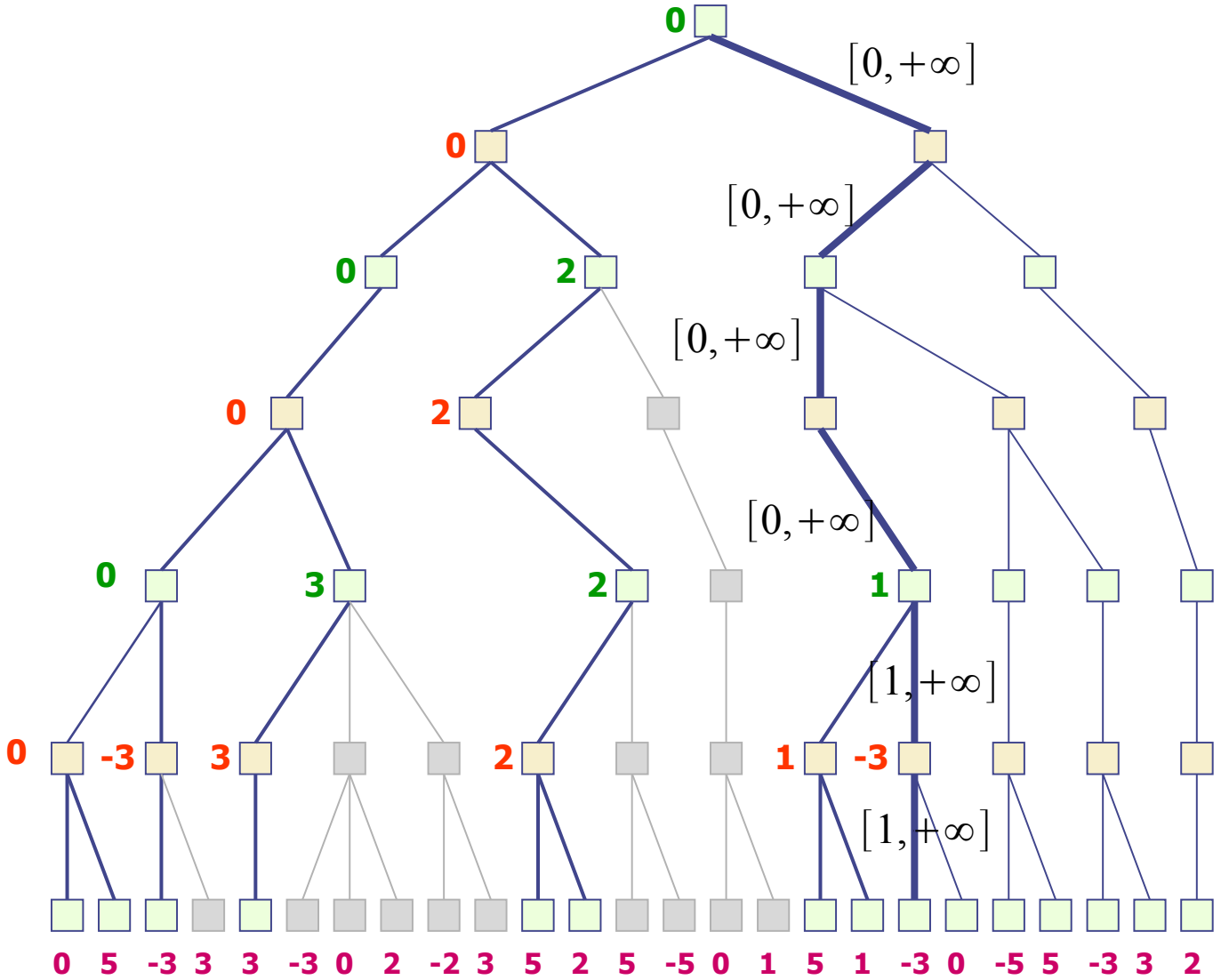
Alpha-Beta Example



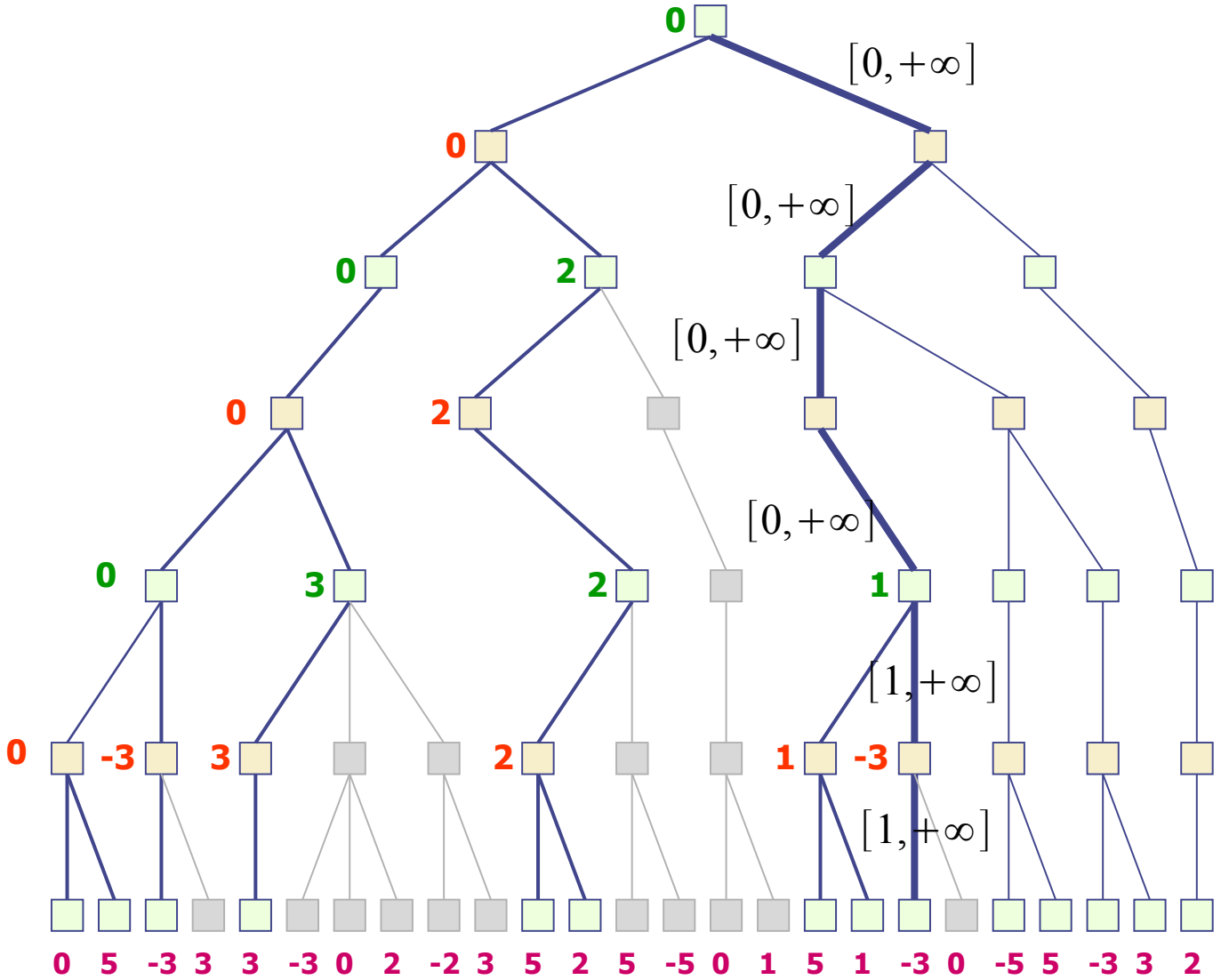
Alpha-Beta Example



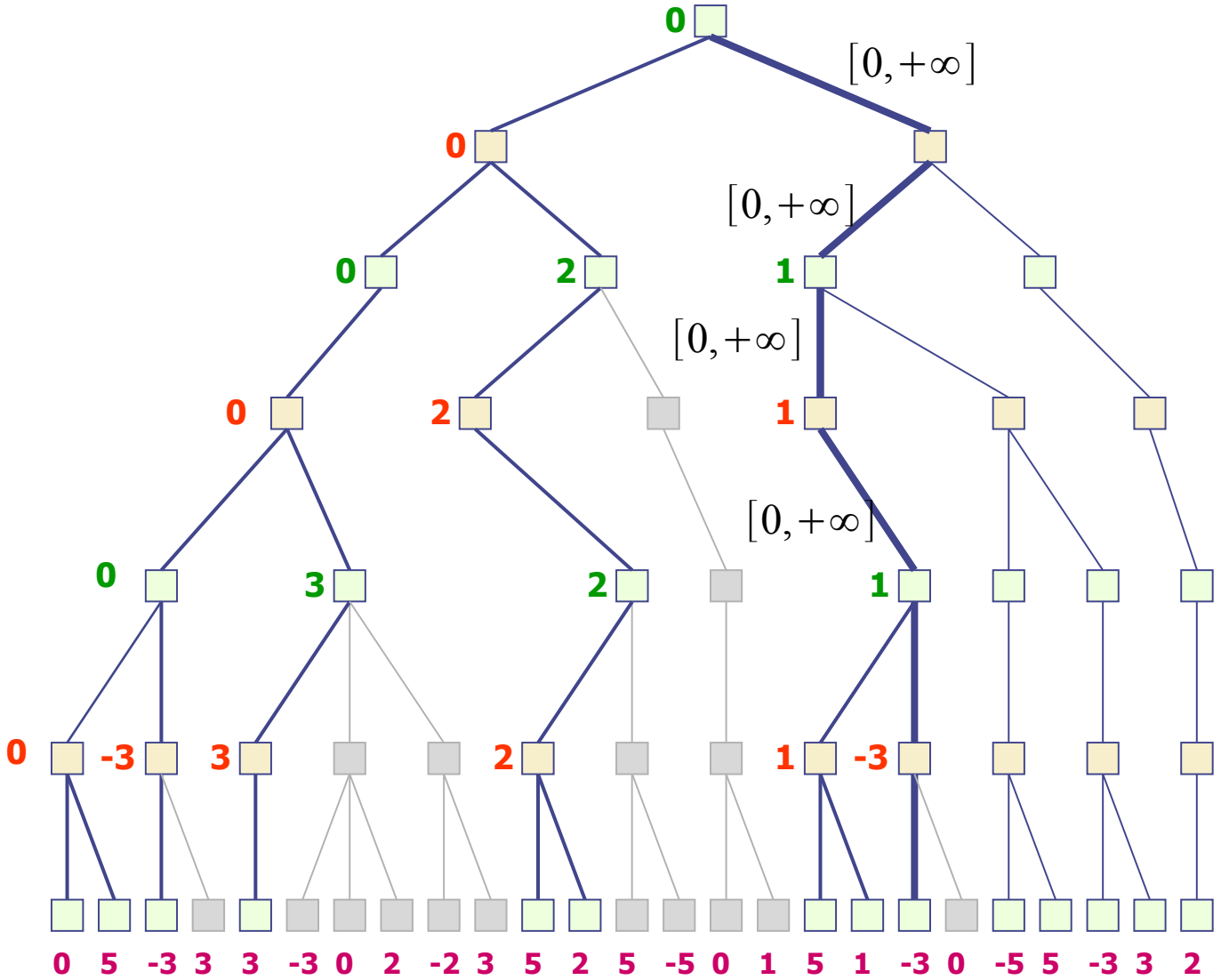
Alpha-Beta Example



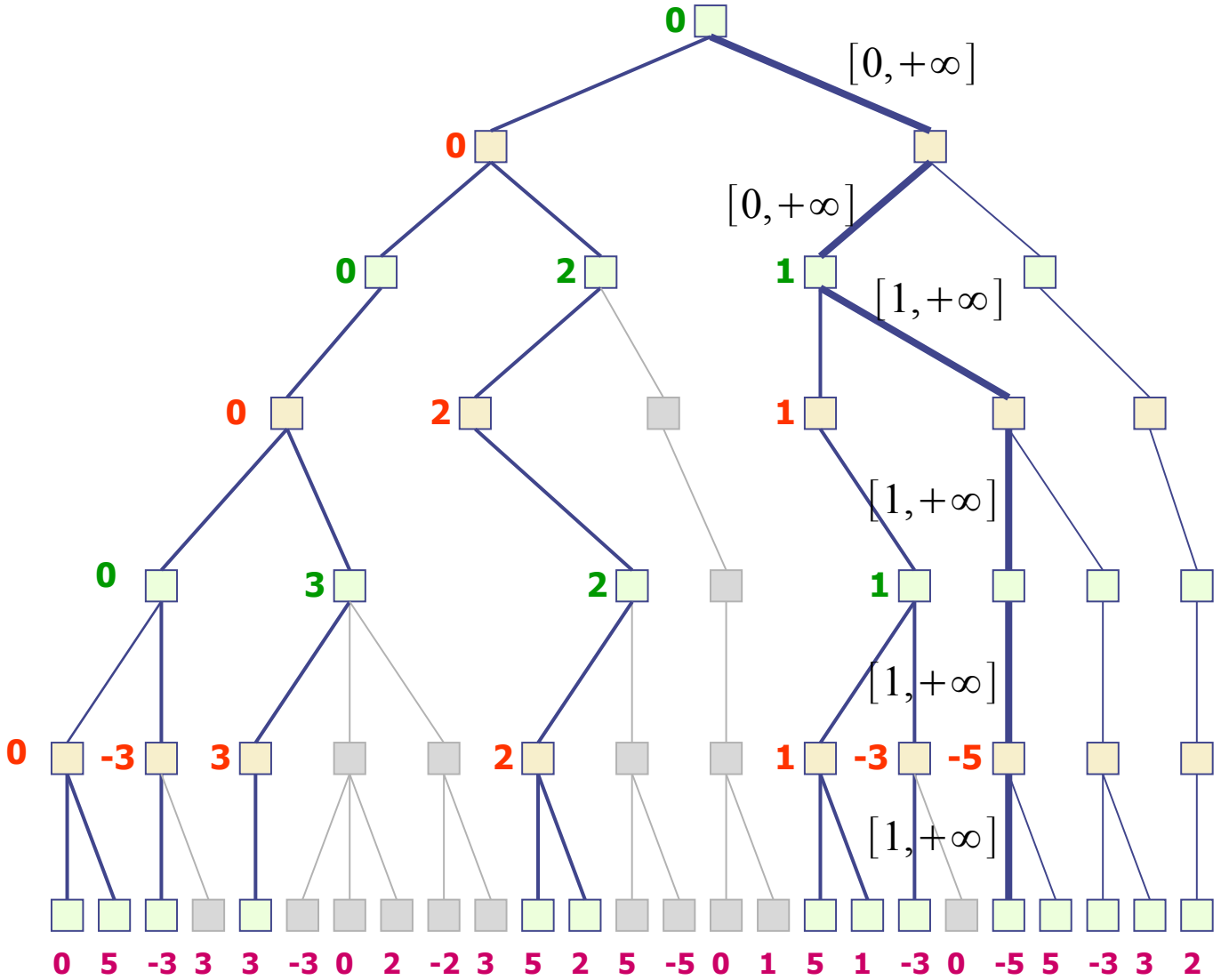
Alpha-Beta Example



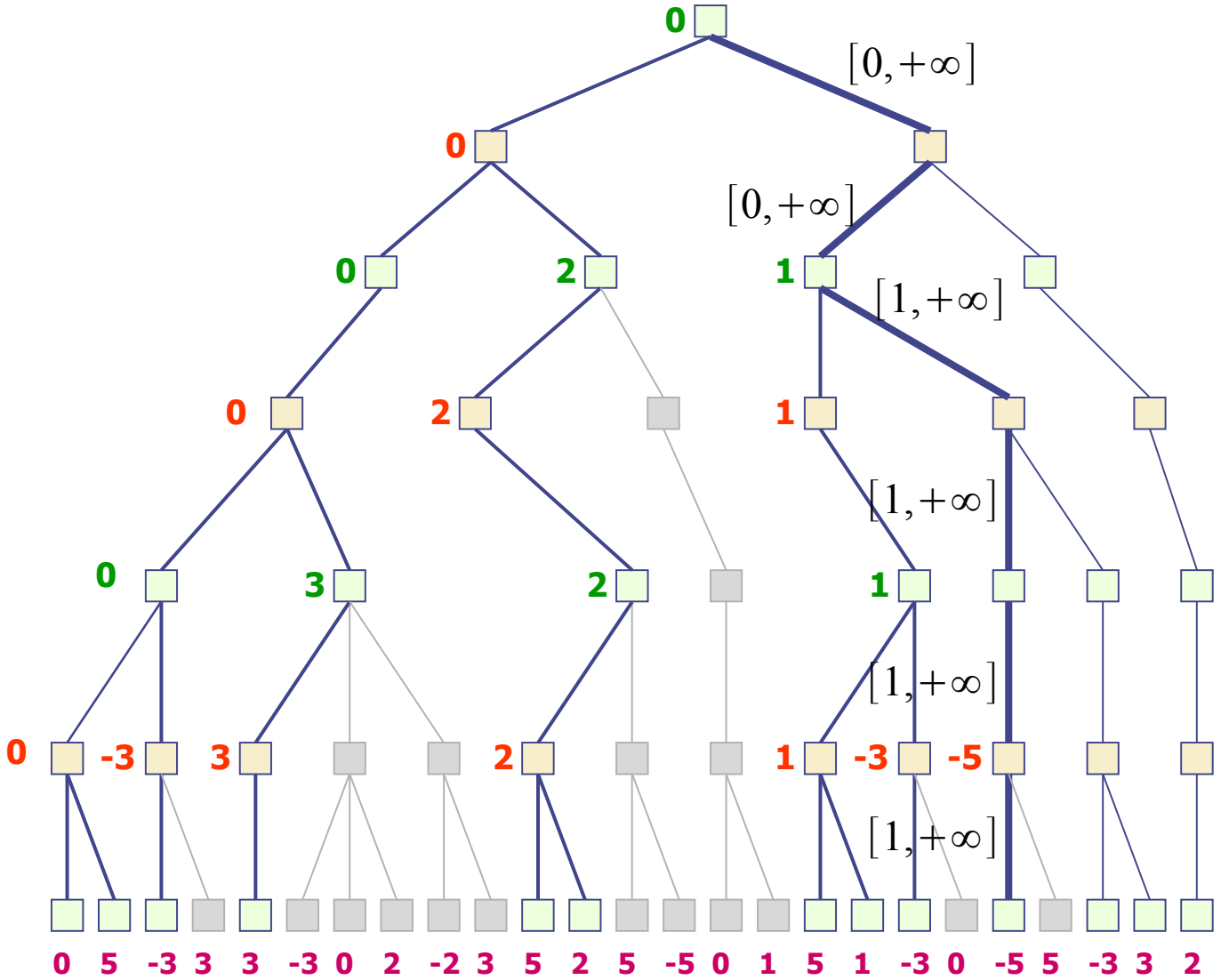
Alpha-Beta Example



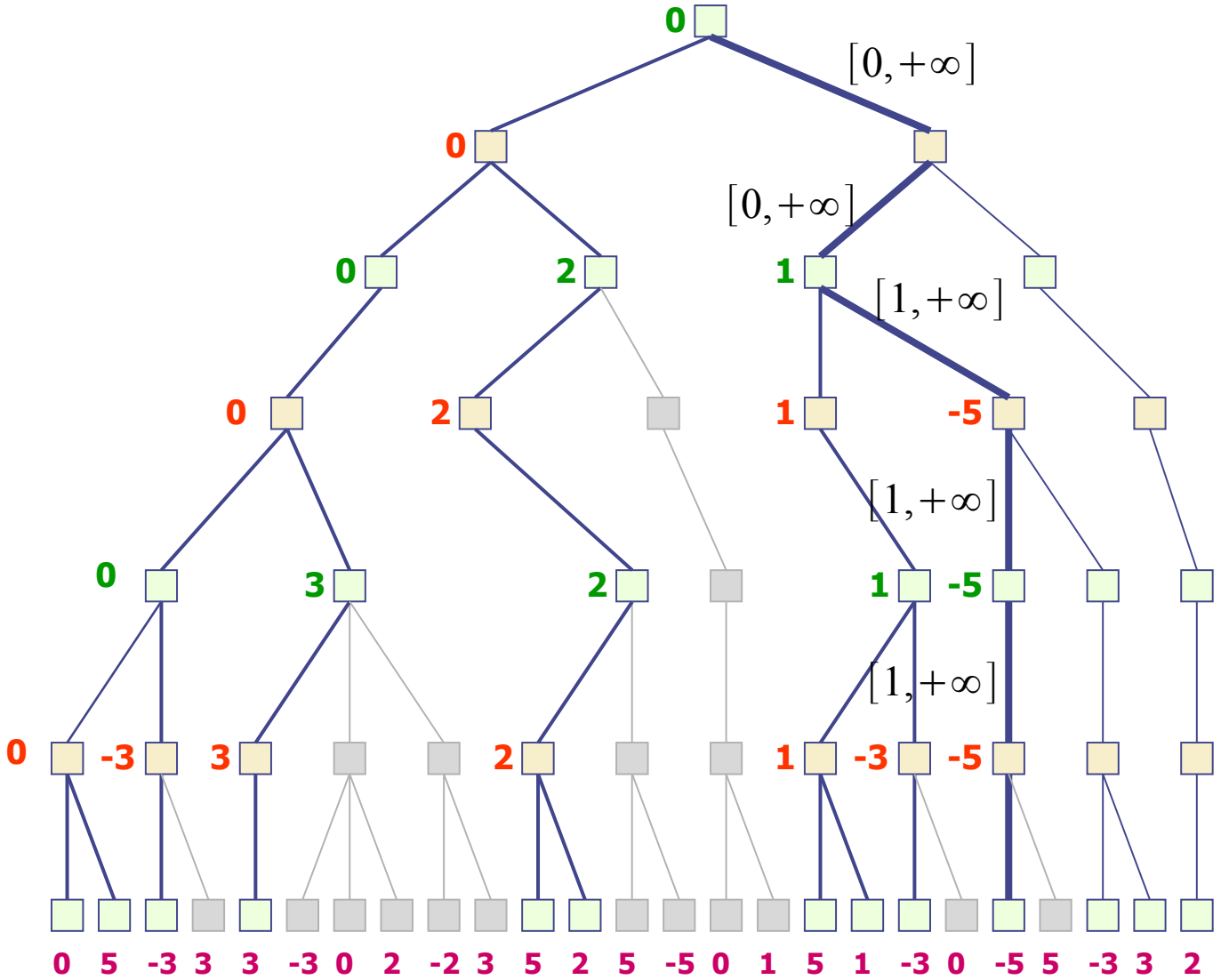
Alpha-Beta Example



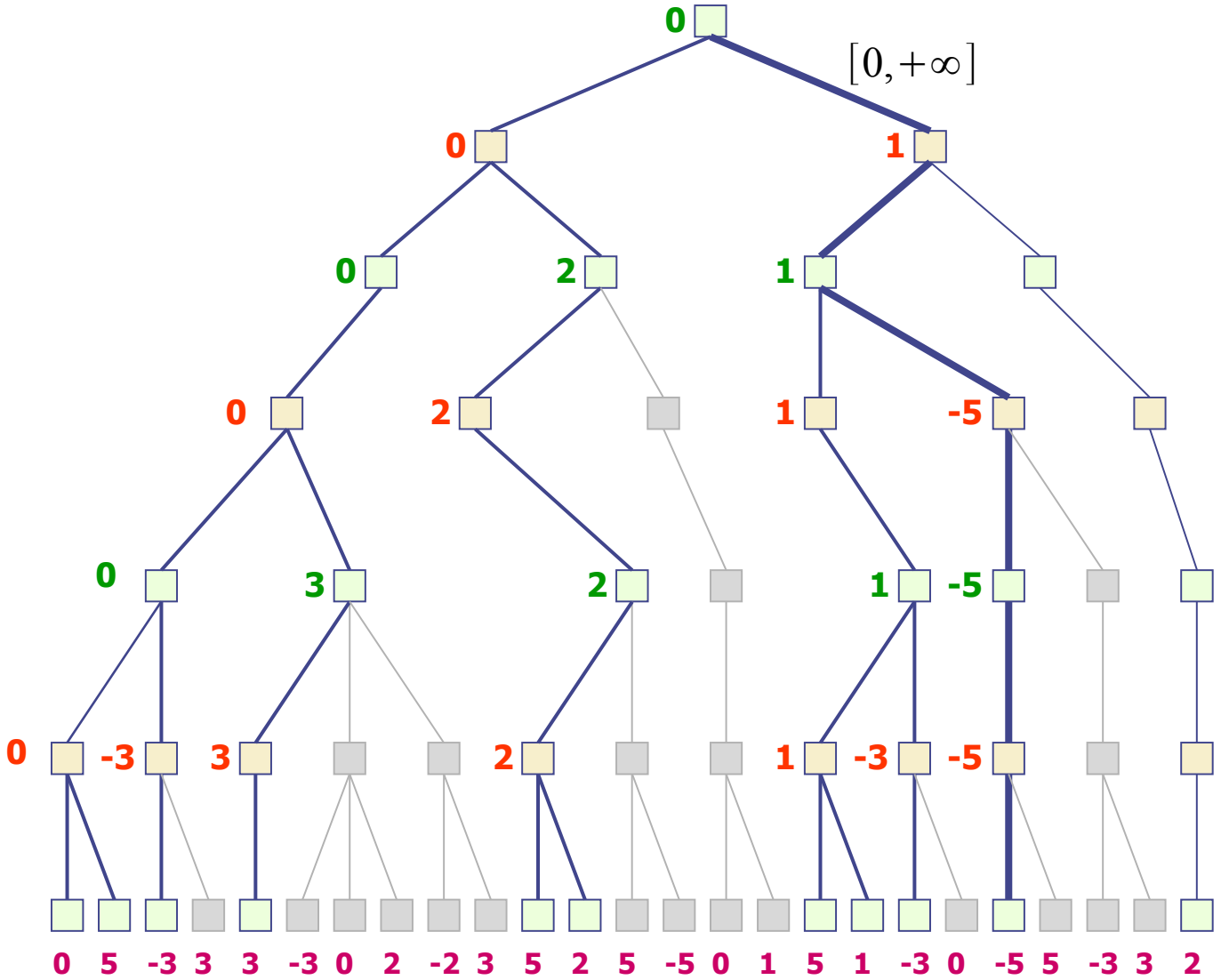
Alpha-Beta Example



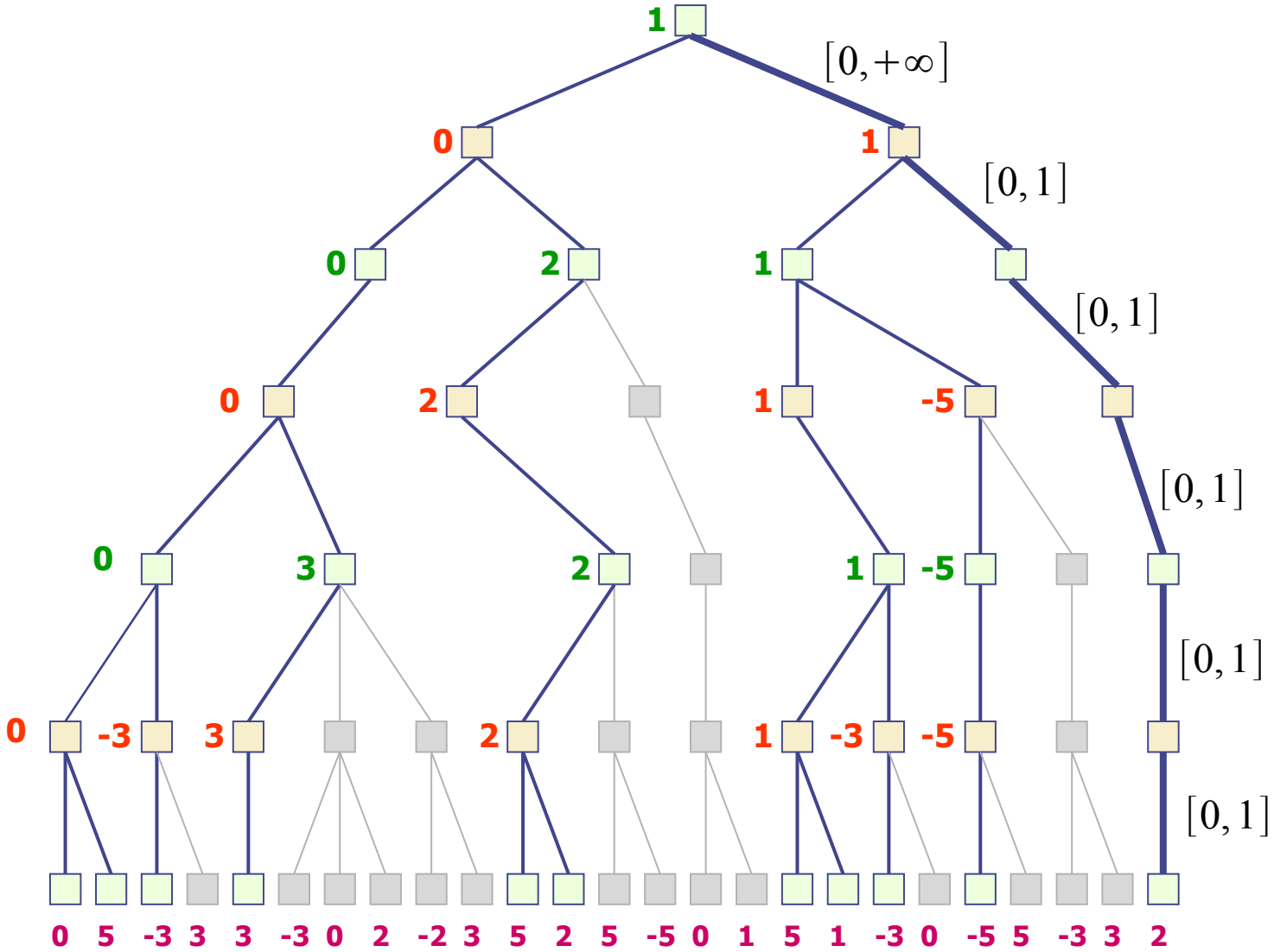
Alpha-Beta Example



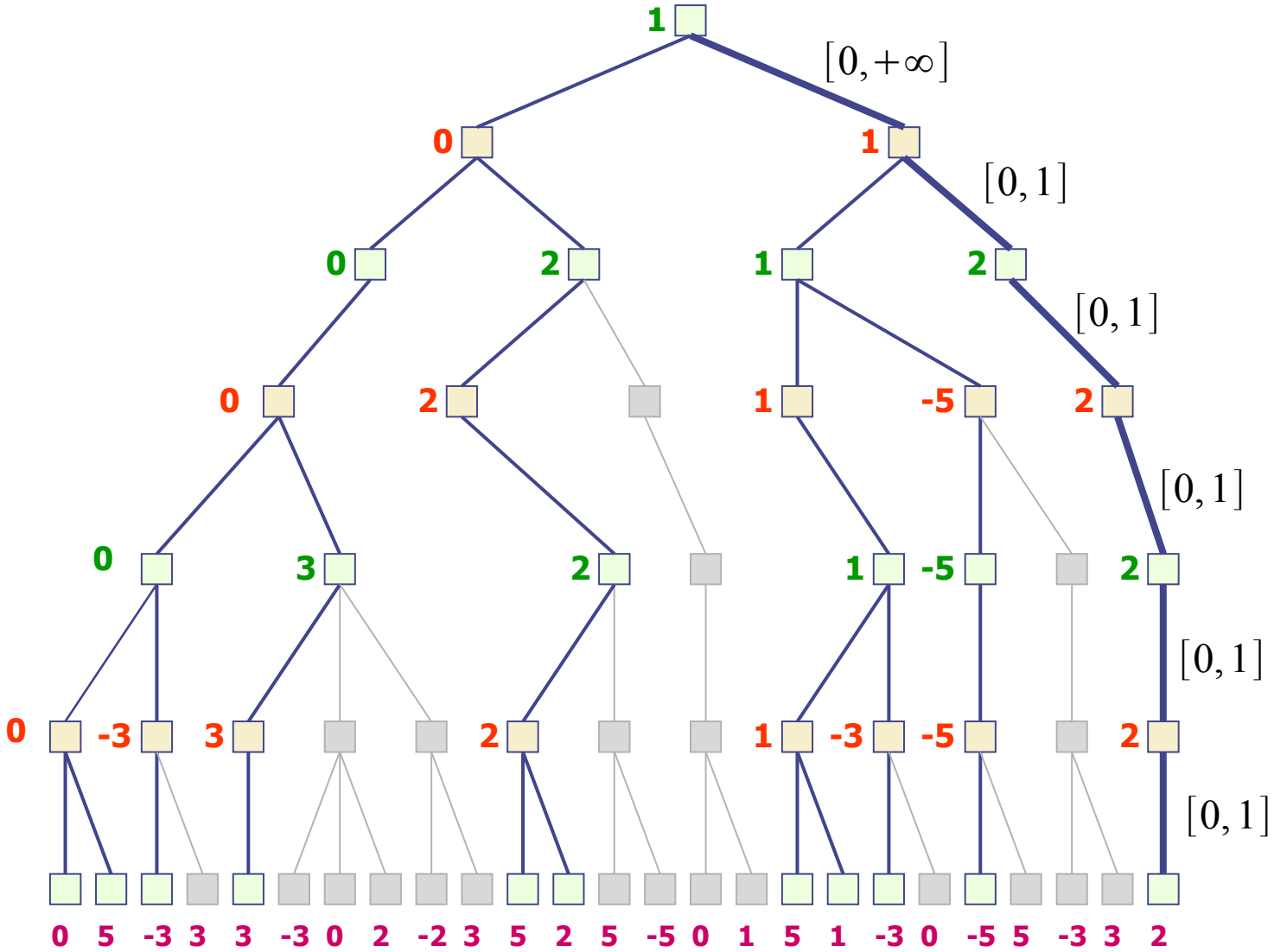
Alpha-Beta Example



Alpha-Beta Example

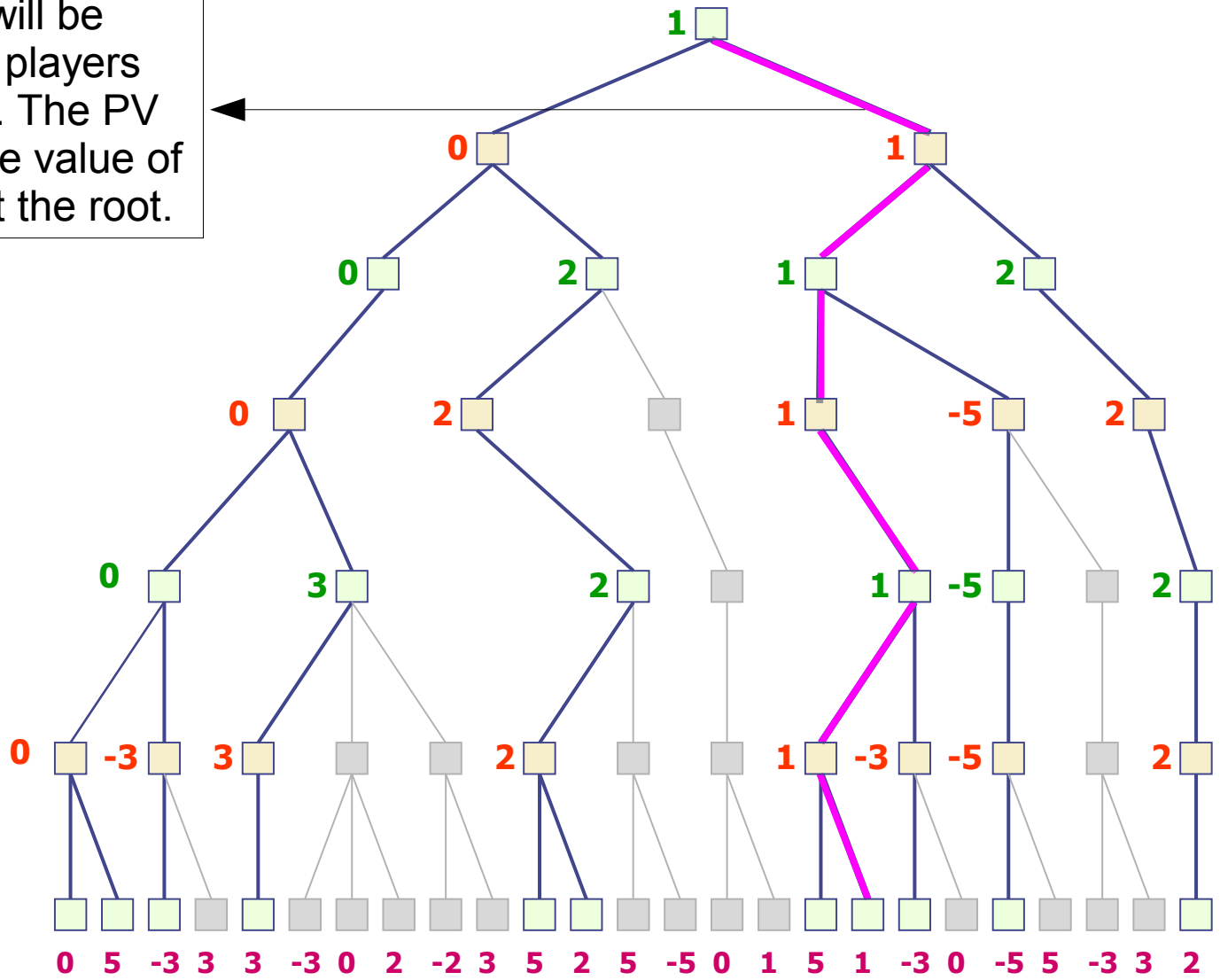


Alpha-Beta Example



Alpha-Beta Example

Principal Variation
The line that will be played if both players play optimally. The PV determines the value of the position at the root.



Properties of Alpha-Beta Pruning

- Pruning does not affect final results
- Entire subtrees can be pruned.
- Effectiveness depends on ordering of branches
  - Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity is $O(b^{m/2})$
  - this corresponds to a branching factor of $\sqrt{b}$

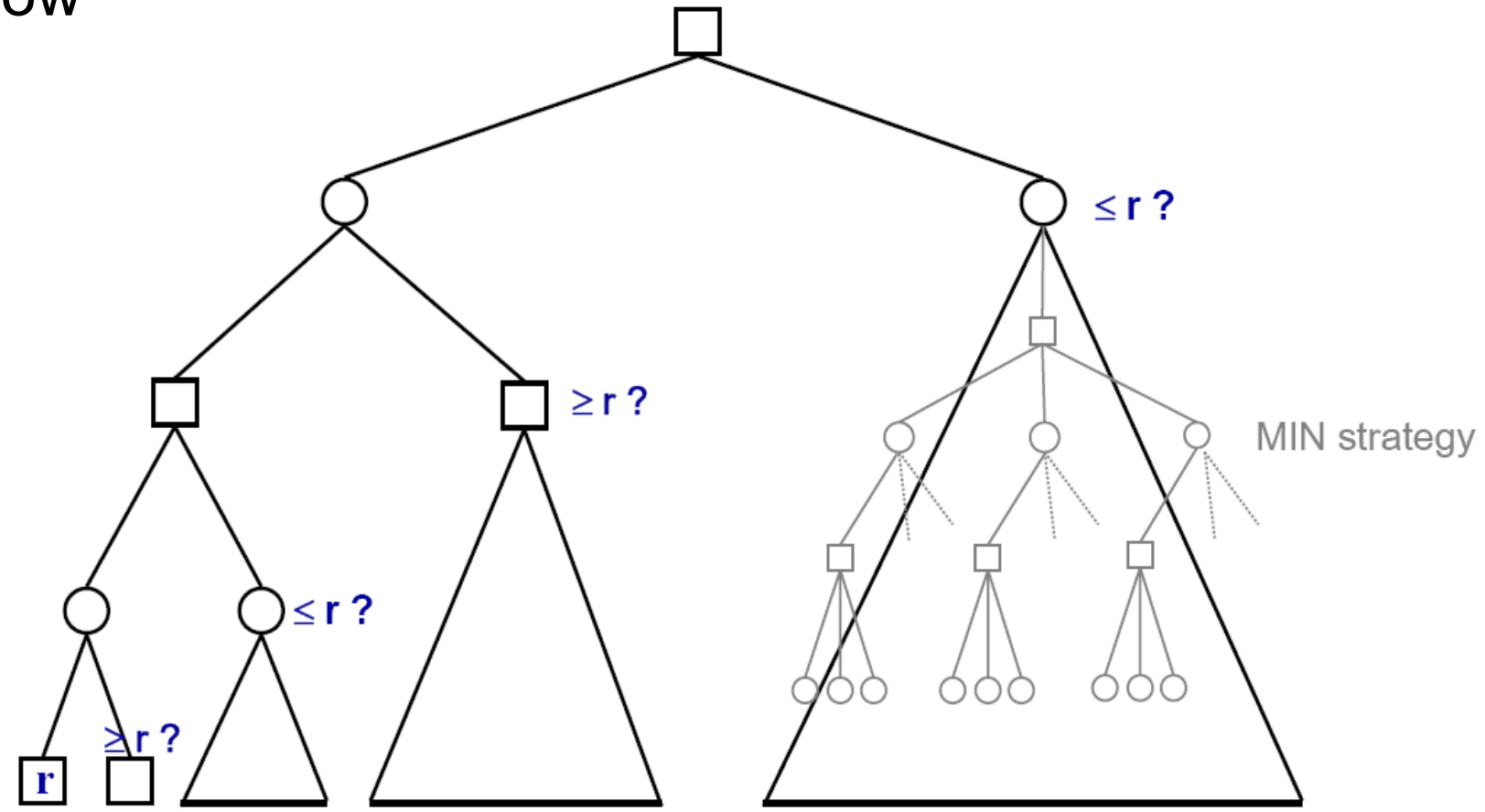
  → Alpha-beta pruning can look twice as deep as minimax in the same amount of time
- However, perfect ordering not possible
  - perfect ordering implies perfect play w/o search
  - random orders have a complexity of $O(b^{3m/4})$
  - crude move orders are often possible and get you within a constant factor of $O(b^{m/2})$
    - e.g., in chess: captures and pawn promotions first, forward moves before backward moves

Minimal Window Search

- If we have a good guess about the value of the position, we can further increase efficiency of Alpha-Beta by starting with a narrower interval than $[-\infty, +\infty]$
  - such an aspiration window will result in more cut-offs
  - with the danger that they may not be correct
- Extreme case: Minimal Window $\beta = \alpha + 1$
  - No value can be between these two values
    - assuming an integer-valued evaluation function
  - Possible results:
    - FAIL HIGH: $\text{Value} \geq \beta = \alpha + 1 \Rightarrow \text{Value} > \alpha$
    - FAIL LOW: $\text{Value} \leq \alpha$
  - Thus, MWS tests *efficiently* (many cutoffs) whether a position is better than a given value or not

NegaScout (Principal Variation Search)

- if we can establish that the value of a node is lower (FAIL LOW), we can prune the node
- If FAIL-HIGH, we need to re-search the tree with a bigger window

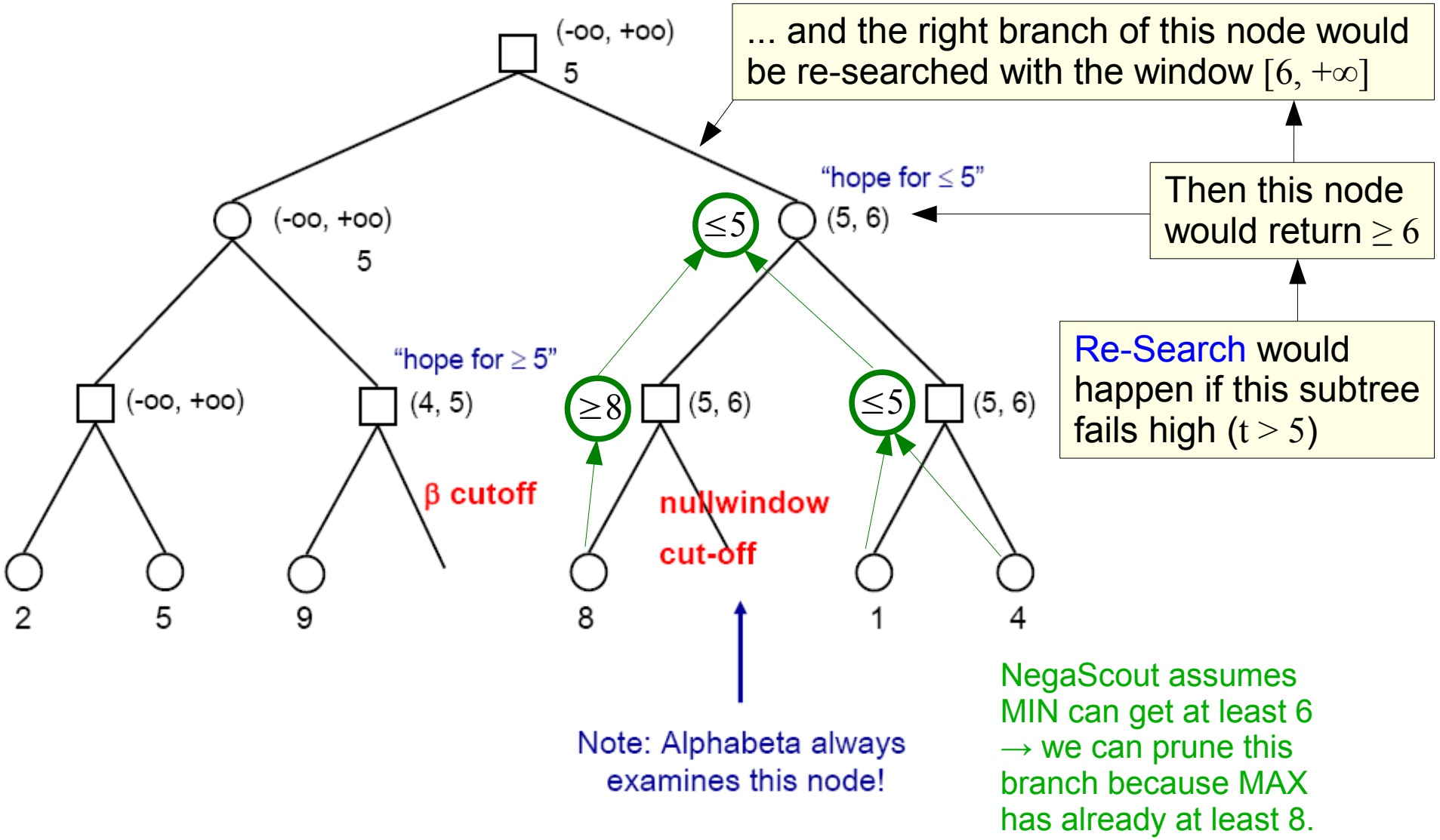


NegaScout (Reinefeld 1982)

```
int NegaScout ( position p; int α, β );
{                                   /* compute minimax value of position p */
        int a, b, t, i;
        determine successors p1,...,pw of p;
        if ( w == 0 )
                return ( Evaluate(p) );                    /* leaf node */
        a = α;
        b = β;
        for ( i = 1; i <= w; i++ ) {
                t = -NegaScout ( pi, -b, -a );
                if ( t > a && t < β && i > 1 && d < maxdepth-1 )
                        a = -NegaScout ( pi, -β, -t );     /* re-search */
                a = max( a, t );
                if ( a >= β )
                        return ( a );                          /* cut-off */
                b = a + 1;                              /* set new null window */
        }
        return ( a );
}
```

FAIL-HIGH:
`t` is outside the null window
(but still within the original window)

NegaScout Example



Performance of NegaScout

- Essentially, NegaScout assumes that the first node is best (i.e., the first node is in the principal variation)
  - if this assumption is wrong, it has to do re-searches
  - if it is correct, it is much more efficient than Alpha-Beta

→ it works best if the move ordering is good

  - for random move orders it will take longer than Alpha-Beta
  - 10% performance increase in chess engines
- It can be shown that NegaScout prunes every node that is also pruned by Alpha-Beta

- Various other algorithms were proposed, but NegaScout is still used in practice
  - SSS*: based on best-first search
  - MTD(f): improves NegaScout by returning upper or lower bounds on the true value, needs memory (TTable) for that

Outline

- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - α-β pruning
  - NegaScout
- Real-time Game-Tree Search
  - evaluation functions
  - practical enhancements
  - selective search
- Games of imperfect information and games of chance
- Simulation Search
  - Monte-Carlo search
  - UCT search

Move Ordering

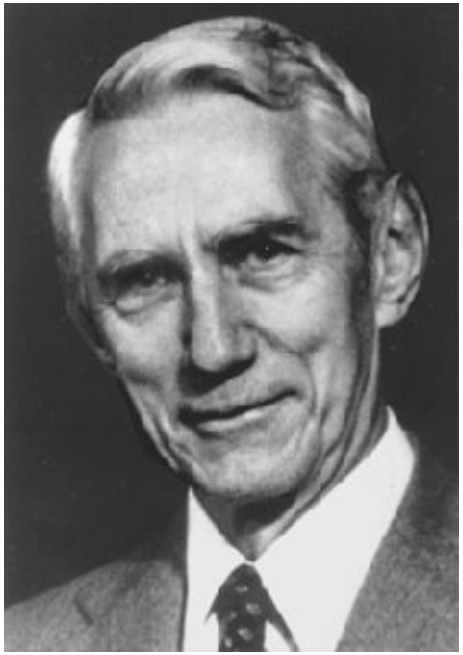
- The move ordering is crucial to the performance of alpha-beta search
- Domain-dependent heuristics:
  - capture moves first
    - ordered by value of capture
  - forward moves first
- Domain-independent heuristics:
  - Killer Heuristic
    - manage a list of moves that produced cutoffs at the current level of search
    - Idea: if there is a strong threat, this should be searched first
  - History Heuristic
    - maintain a table of all possible moves (independent of current position)
    - if a move produces a cutoff, its value is increased by a value that grows fast with the search depth (e.g., d^2 or 2^d)

Imperfect Real-World Decisions

- In general the search tree is too big to make it possible to reach the terminal states!
  - even though alpha-beta effectively doubles the search depth
- Examples:
  - Checkers: ~10^{40} nodes
  - Chess: ~10^{120} nodes
- For most games, it is not practical within a reasonable amount of time
- Key idea (Shannon 1950):
  - Cut off search earlier
    - replace TERMINAL-TEST by CUTOFF-TEST
    - which determines whether the current position needs to be searched deeper
  - Use heuristic evaluation function EVAL
    - replace calls to UTILITY with calls to EVAL
    - which evaluates how promising the position at the cutoff is

Brute-Force vs. Selective Search

- Shannon Type-A (Brute Force)
  - search all positions until a fixed horizon
  - CUTOFF-TEST test only tests for the depth of a position
- Shannon Type-B (Selective Search)
  - CUTOFF-TEST prunes uninteresting lines (as humans do)



- Selective Search preferred by Shannon and contemporaries
  - early program limit branching factor (e.g., Newell/Simon/Show to the „magical number“ 7)
- Brute-Force Search was shown to outperform selective search in the 70s
- Current programs use a mixture
    - selective search near the leaves

Fixed-Depth Alpha-Beta

Cutoff the search at a pre-determined depth

- CUTOFF-TEST compares the current search depth to a fixed maximum depth *D* and returns true if the depth has been reached or if the position is a terminal position

- At a terminal position:
  - return the game-theoretic score
- At a max-depth position:
  - return the value of the evaluation function EVAL
- At an interior node:
  - recursively call alpha-beta
  - increment the current search depth by one

Note:

- the incrementation of the search depth is often realized with a decrement of an initial search depth, and a cutoff at 0.

Evaluation Function

- Evaluation function or static evaluator is used to evaluate the "goodness" of a game position.
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node

- The zero-sum assumption allows us to use a single evaluation function to describe the goodness of a board with respect to both players.
  - $f(n) >> 0$: position n good for me and bad for you
  - $f(n) << 0$: position n bad for me and good for you
  - $f(n) \approx 0$: position n is a neutral position
  - $f(n) = +\infty$: win for me
  - $f(n) = -\infty$: win for you

Heuristic Evaluation Function

- Idea:
  - produce an estimate of the expected utility of the game from a given position.
- Performance:
  - depends on quality of EVAL.
- Requirements:
  - EVAL should order terminal-nodes in the same way as UTILITY.
  - Computation should not take too long (many leaf nodes have to be evaluated)
  - For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.

Linear Evaluation Functions

- Most evaluation functions are linear combinations of features
  - $\mathrm{EVAL}(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \ldots + w_n \cdot f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$
- a feature f_i encodes a certain characteristic of the position
  - e.g., # white queens/rooks/knights,..., # of possible moves, # of center squares under control, etc.
  - originate from experience with the game
- Advantages:
  - conceptually simple, typically fast to compute
- Disadvantages:
  - tuning of the weights may be very hard (→ machine learning)
  - adding up the weighted features makes the assumption that each feature is independent of the other features

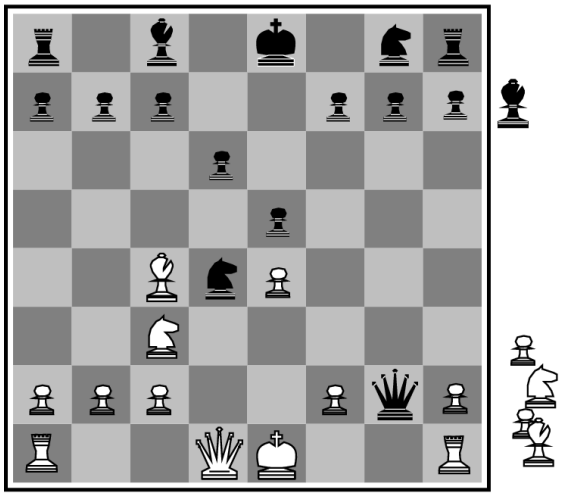
Evaluation Function Examples

- Example of an evaluation function for Tic-Tac-Toe:
  - $f(n)$ = [# 3-lengths open for me] − [# 3-lengths open for you]
  - where a 3-length is a complete row, column, or diagonal
- Alan Turing's function for chess
  - $f(n) = w(n)/b(n)$ where
    - $w(n)$ = sum of the point value of white's pieces
    - $b(n)$ = sum of black's
  - Chess champion program Deep Blue has about 6000 features in its evaluation function
- Current state-of-the-art programs use non-linear functions
  - e.g. different feature weights in different game phases

Evaluation functions

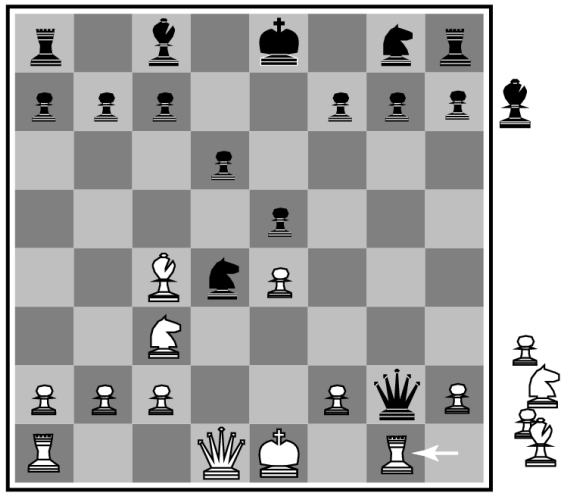
Evaluation is typically very brittle

- small changes in the position may cause large leaps in the evaluation



(a) White to move

Black is clearly winning
(up in material)



(b) White to move

White is clearly winning
(can take black's queen)

→ Evaluation and Search are not independent:

- What is taken care of by search need not be in EVAL

→ Evaluation only applied to stable „quiescent“ position

Quiescence Search

- Evaluation only useful for quiescent states
  - states w/o wild swings in value in near future
  - e.g.: states in the middle of an exchange are not quiet
- Algorithm
  - When search depth reached, compute quiescence state evaluation heuristic
  - If state quiescent, then proceed as usual; otherwise increase search depth if quiescence search depth not yet reached
- Example:
  - In chess, typically all capturing moves, and all pawn promotions are followed
  - no depth parameter needed, because there is only a finite number of captures and pawn promotions
  - Note that this is different with checks!

Iterative Deepening

Repeated fixed-depth searches for depths $d = 1, ..., D$

- as for single-agent search
- frequently used in game-playing programs

Advantages:

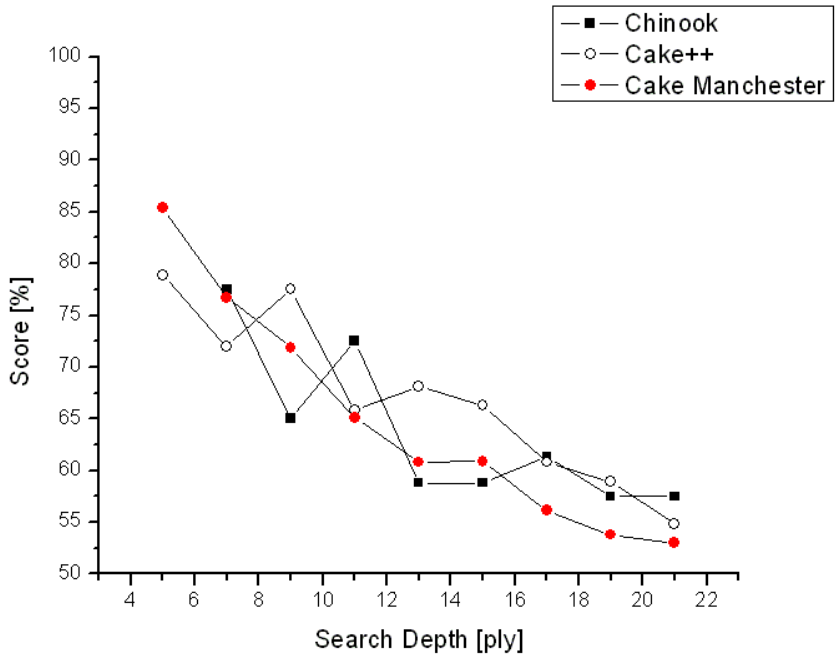
- works well with transposition tables
- improved dynamic move-ordering in alpha-beta
  - what worked well in the previous iteration is tried first in the next iteration
- simplifies time managements
  - if there is a fixed time limit per move, this can be handled flexibly by adjusting the number of iterations during the search
  - previous iterations provide useful information that allow to guess whether the next iteration can be completed in time

→ Quite frequently the the total number of nodes searched is smaller than with non-iterative search!

Why Should Deeper Search Work?

- If we have a perfect evaluation function, we do not need search.
- If we have an imperfect evaluation function, why should its performance get better if we search deeper?
- Game Tree Pathologies
  - One can construct situations or games where deeper search results in bad performance
- Diminishing returns:
  - the gain of deeper searches goes down with the depth
  - can be observed in most games
  - various different explanations

Results of Checkers pograms that play with depth d against themselves with depth d-2



Transposition Tables

- Repeated states may occur
  - different permutations of the move sequences lead to the same positions
- Can cause exponential growth in search cost

Transposition Tables:

- Basic idea:
  - store found positions in a hash table
  - if it occurs a second time, the value of the node does not have to be recomputed
- Essentially identical to the *closed* list in GRAPH-SEARCH
- May increase the efficiency by a factor of 2
- Various strategies for swapping positions once the table size is exhausted

Transposition Tables - Implementation

Each entry in the hash table stores

- State evaluation value (including whether this was as exact value or a fail high/low value)
- Search depth of stored value (in case we search deeper)
- Hash key of position (to eliminate collisions)
- (optional) Best move from position

Zobrist Hash Keys:

- Generate 3d-array of random 64-bit numbers
  - One key for each combination of piece type, location and color
- Start with a 64-bit hash key initialized to 0
- Loop through current position, XOR'ing hash key with Zobrist value of each piece found
  - Can be updated incrementally by XORing the "from" location and the "to" location to move a piece

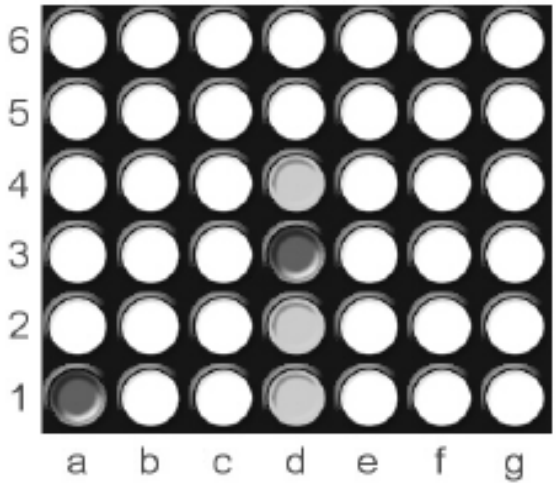
Zobrist Keys for Connect-4

- Key Table:

piece-square	64-Bit-Zufallszahl
Weiß auf a1	1010100000101011011111101011101100111111010001100011000000100100
Schwarz auf a1	1010111111110111100001000100011111010100011101101110110110111011
Weiß auf a2	0111001010111111101100111011100010101000000100011101010110111001
Schwarz auf a2	0000001000011011110111100001011000011111011101001101000100011010
Weiß auf a3	1110100000110010011010100101101010011111100010101010000101111010
...	...
Weiß auf d1	0101101001010111011111000001100011000011111011001100000100010110
Schwarz auf d1	1101100001111000010011001111110010010100011110001000001101010011
Weiß auf d2	0010000001100000111100011011001110001101110010100000000000001011
Schwarz auf d2	0000111101011001110011011111010011110011101010100100100001100100
Weiß auf d3	1000101100110111110001110110100000001001100001011000101001000000
Schwarz auf d3	1100101101011110100001001000100100011001000010011100001111011011
...	...
Schwarz auf g6	0110100110010001111100000001101100111010010111111100010100101110

Zobrist Keys for Connect-4

- Computation of a position key:




```
  10101111111101111000010001000111110101000111011011101101110111011  (Schwarz auf a1)
⊕ 01011010010101110111110000011000110000011110110011000001000010110  (Weiß auf d1)
⊕ 00100000011000001111000110110011100011011100101000000000000001011  (Weiß auf d2)
⊕ 11001011010111101000010010001001000110010000100111000011110111011  (Schwarz auf d3)
⊕ 01001010011000100001101100111110010010111111010000001111011101111  (Weiß auf d4)
= 01010100111111001001011001011011110010001001100010100011000010100
```

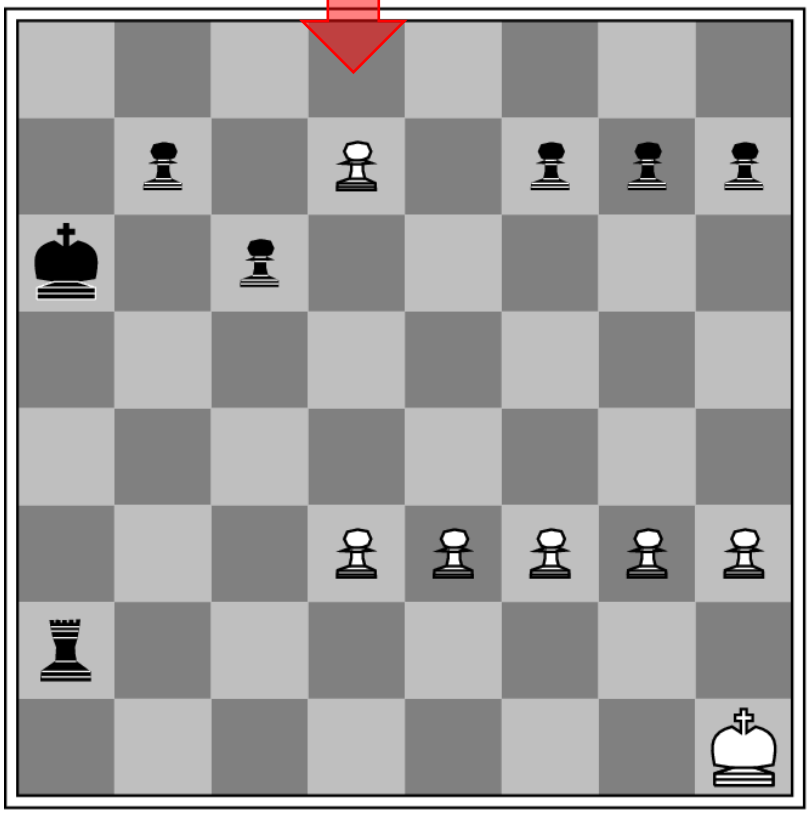
hash key for above position

Horizon Effect

- Problem with fixed-depth search:
  - if we only search n moves ahead, it may be possible that the catastrophy can be delayed by a sequence of moves that do not make any progress
  - also works in other direction (good moves may not be found)
- Examples:
  - computer starts to give away its pieces in hopeless positions (because this avoids the mate)
  - checks:

Fixed depth search thinks it can avoid the queening move

Black can give many consecutive checks before white escapes



Black to move

Search Extensions

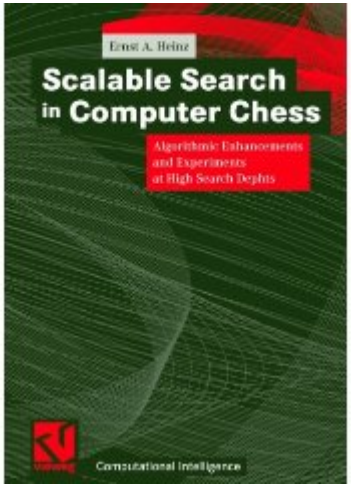
- game-playing programs sometimes extend the search depth
  - typically by skipping the step that increments the current search depth
    - increments with fractional values are also possible (multiple fractional extensions are needed for an extension by 1)
  - search is then continued as usual (until horizon is reached)
    - but the depth of the of the horizon may be different in different branches of the trees
- Danger:
  - extensions have to be designed carefully so that the search will always terminate (within reasonable time)
- Typical idea:
  - extend the search when a forced move is found that limits the possible replies to one (or very few) possible actions
- Examples in chess:
  - checks, recaptures, moves with passed pawns

Forward Pruning

- Alpha-Beta only prunes search trees when it is safe to do so
  - the evaluation will not change (guaranteed)
- Human players prune most of the possible moves
  - and make many mistakes by doing so...

- Several variants of forward pruning techniques are used in state-of-the-art chess programs
  - Null-move pruning
  - Futility pruning
  - Razoring

- See, e.g.,
  - Ernst A. Heinz: *Scalable Search in Computer Chess*. Vieweg 2000.



Null-Move Pruning

- Idea: in most games, making a move improves the position
- Approach:
  - add a „null-move“ to the search, i.e., assume that current player does not make a move
  - if the null-move search (sometimes at reduced depth) results in a cutoff, assume that making a move will do the same
- Danger:
  - sometimes it is good to make no move (Zugzwang)
- Improvements:
  - do not make a null-move if
    - in check
    - in endgame
    - previous move was a null-move
  - verified null-move-pruning: do not cut off but reduce depth
  - adaptive null-move pruning:
    - use variable depth reduction for the null-move search

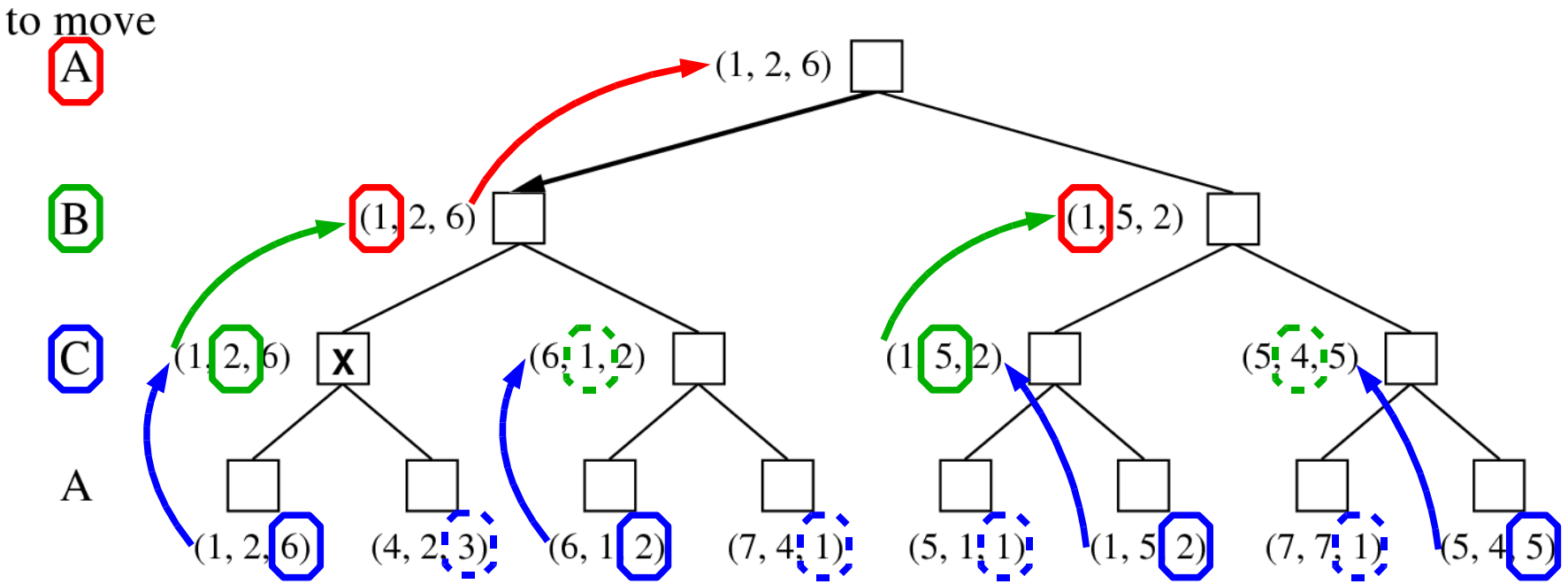
Outline

- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - α-β pruning
  - NegaScout
- Real-time Game-Tree Search
  - evaluation functions
  - practical enhancements
  - selective search
- Games of imperfect information and games of chance
- Simulation Search
  - Monte-Carlo search
  - UCT search

Multiplayer games

- Games allow more than two players
- Single minimax values become vectors
  - one evaluation value for each player
- Example:
  - three players (A, B, C) → $f(n) = (f_A(n), f_B(n), f_C(n))$

Two-Player 0-sum are a special case where $f_A(n) = -f_B(n)$ (hence only one value is needed)



Retrograde Analysis

- Retrograde Analysis Algorithm (goes back to Zermelo 1912)
  - builds up a database if we want to strongly solve a game

0. Generate all possible positions
1. Find all positions that are won for MAX
   i. mark all terminal positions that are won for MAX
   ii. mark all positions where MAX is to move and can make a move that leads to a marked position
   iii. mark all positions where MIN is to move and all moves lead to a marked position
   iv. if there are positions that have not yet been considered goto ii.
2. Find all positions that are won for MIN
   - analogous to 1.
3. All remaining positions are draw

Games of Chance

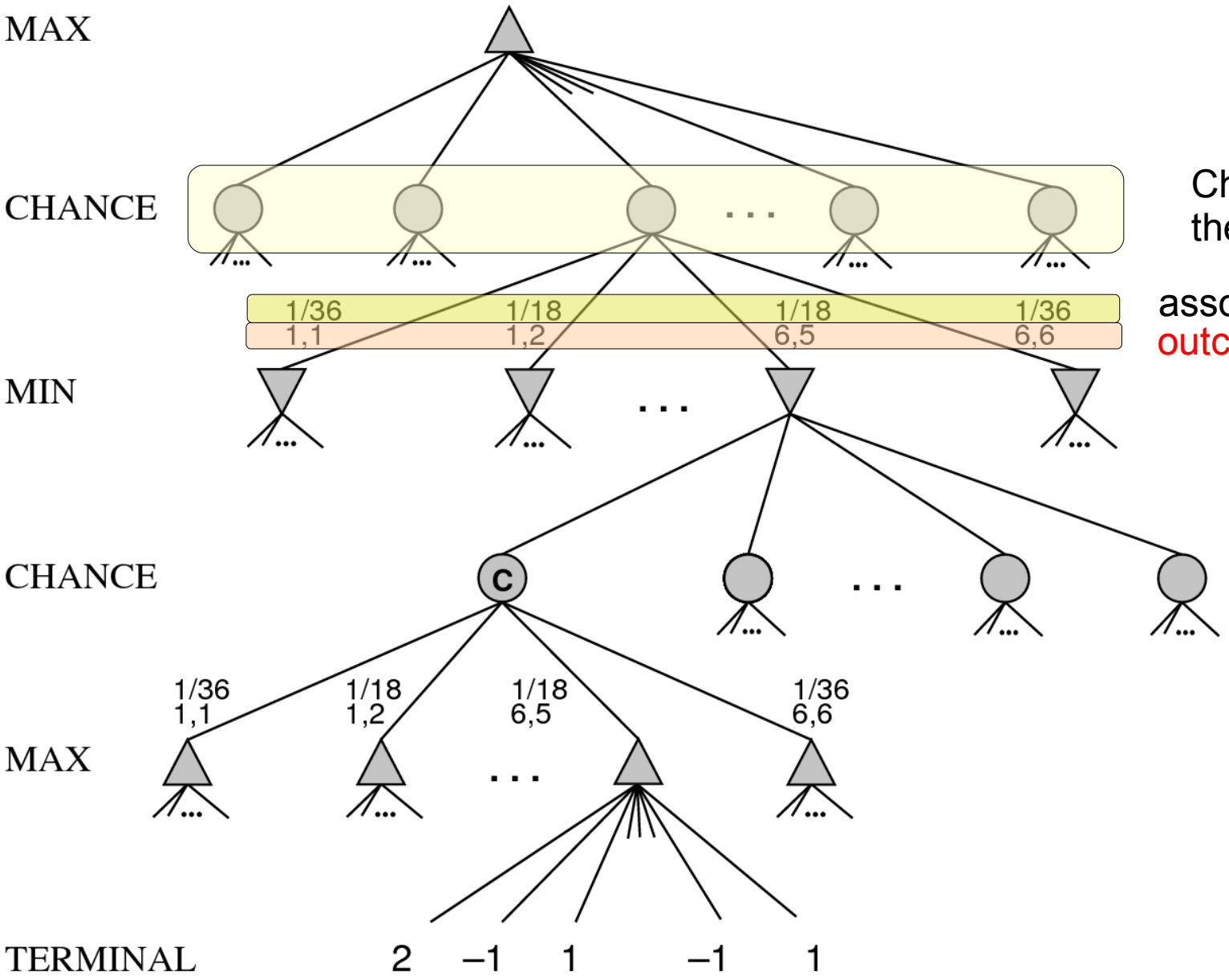
- Many games combine skill and chance
  - i.e., they contain a random element like the roll of dice
- This brings us closer to real-life
  - in real-life we often encounter unforeseen situations
- Examples
  - Backgammon, Monopoly, ...

- Problem
  - Player MAX cannot directly maximize his gain because he does not know what MIN's legal actions will be
    - MIN makes a roll of the dice *after* MAX has completed his ply
  - and vice versa (MIN cannot minimize)

  → Minimax or Alpha-Beta no longer applicable

→ Standard game trees are extended with chance nodes

Game-Tree with Chance Nodes



Chance nodes for the roll of two dice

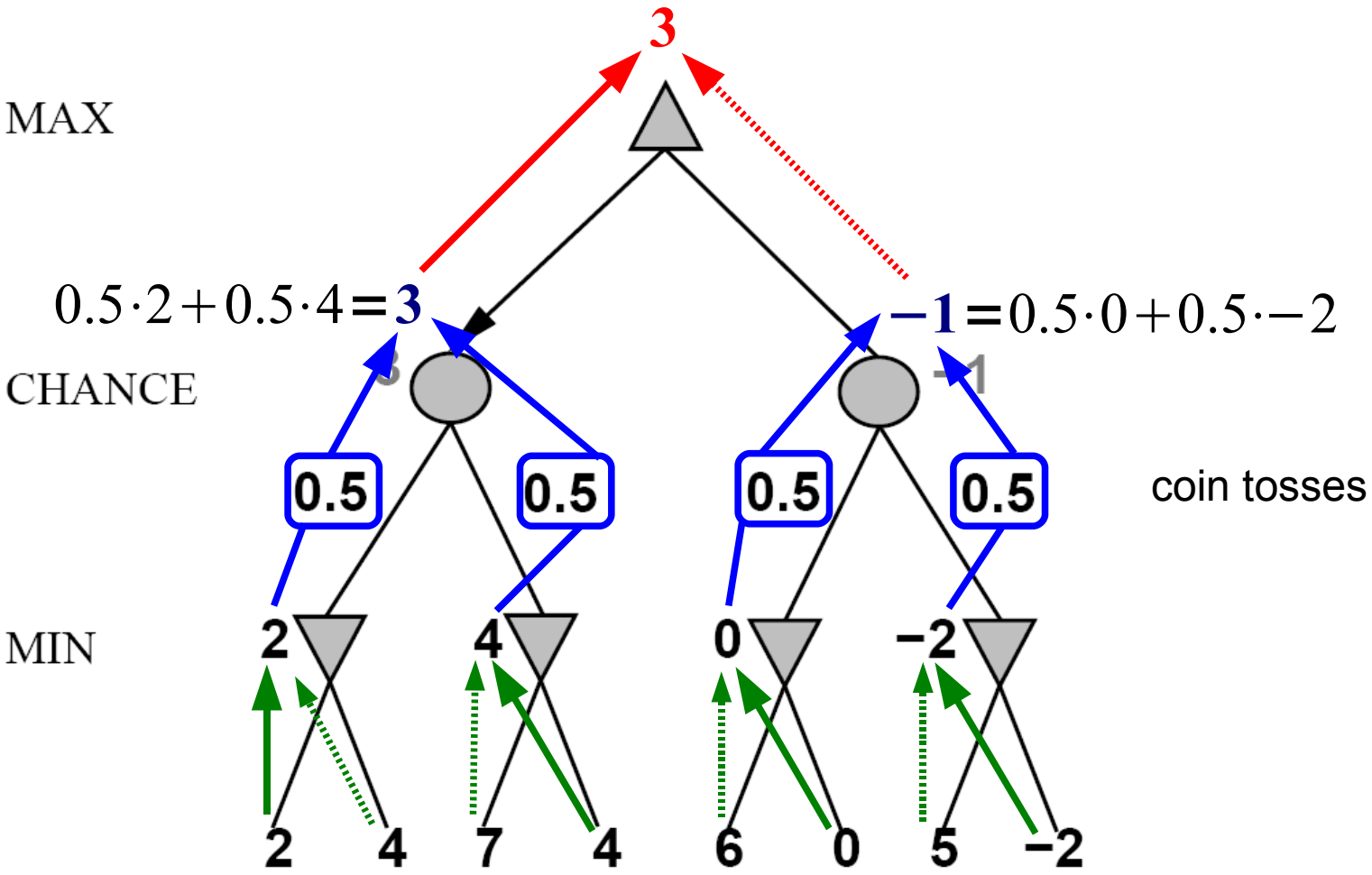
associated probability outcome of the dice roll

Optimal Strategy with Chance Nodes

- MAX wants to play the move that maximizes his chances of winning
- Problem:
  - the exact outcome of a MAX-node cannot be computed because each MAX-node is followed by a chance node
  - analogously for MIN-nodes
- Expected Minimax value
  - compute the expected value of the outcome at each chance node

$$\text{EXPECTIMINIMAX}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in SUCCESSORS(n)} \text{EXPECTIMINIMAX} & \text{if } n \text{ is a MAX node} \\ \min_{s \in SUCCESSORS(n)} \text{EXPECTIMINIMAX} & \text{if } n \text{ is a MIN node} \\ \sum_{s \in SUCCESSORS(n)} P(s) \cdot \text{EXPECTIMINIMAX} & \text{if } n \text{ is a chance node} \end{cases}$$

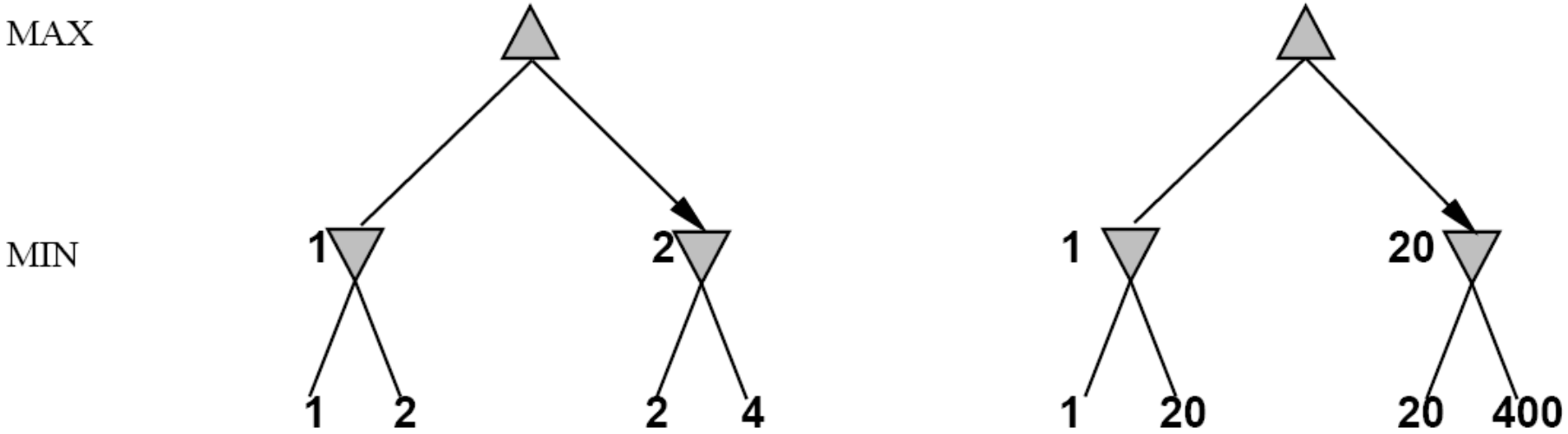
Example



EXPECTIMINIMAX gives perfect play, like MINIMAX

Re-Scaling of Evaluation Functions

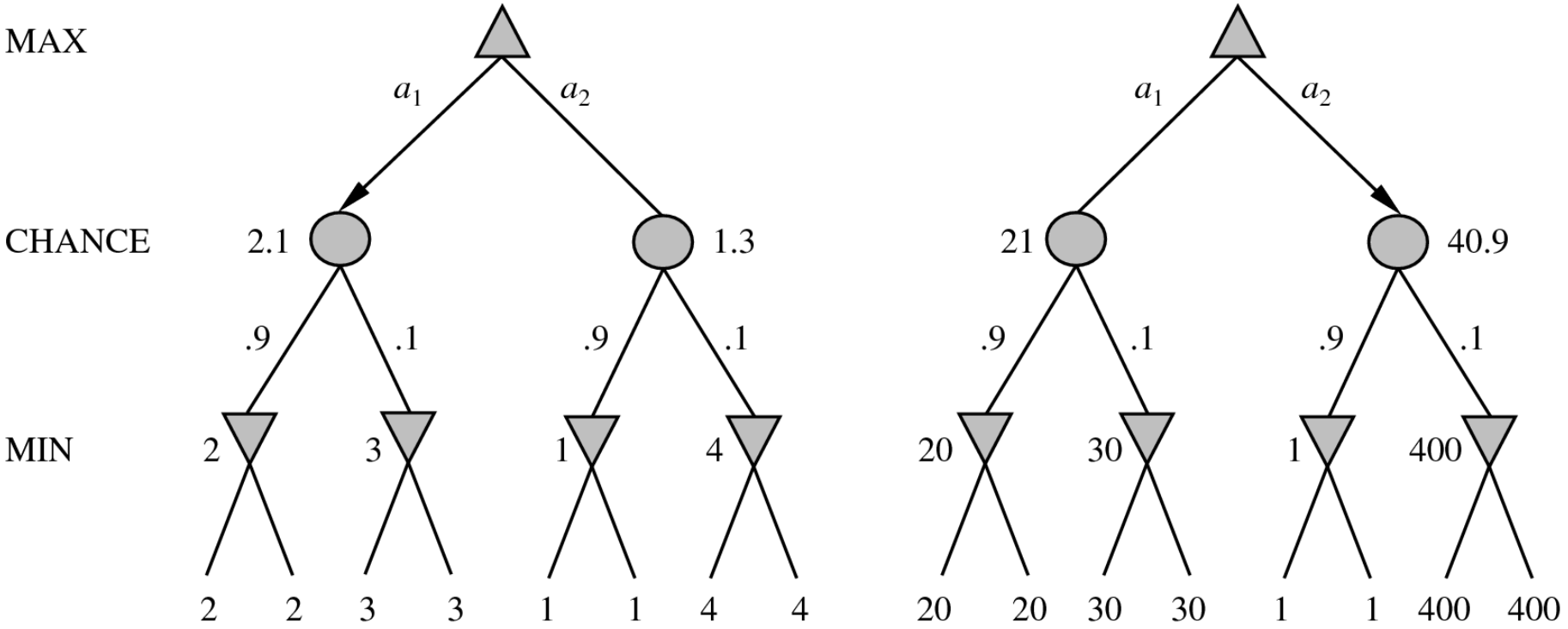
- Minimax:
  - no problem, as long as values are ordered in the same way (monotonic transformations)



MAX plays the same move in both cases

Re-Scaling of Evaluation Functions

- Expectiminimax:
  - Monotonic transformations may change the result



  - only positive *linear* transformations preserve behavior

→ EVAL should be proportional to the expected outcome!

Nondeterministic Games in Practice

- Complexity
  - In addition to the branching factor, the number of different outcomes c adds at each chance node to the complexity
  - Total complexity is $O(b^m c^m)$
    → deep look-ahead not feasible

$c = 2$	for coin flip
$c = 6$	for rolling one die
$c = 21$	for rolling two dice

- prob. of reaching a given node shrinks with increasing depth
  - forming plans is not that important
    → deep look-ahead is also not that valuable
  - Example:
    - TD-Gammon uses only 2-ply look-ahead + very good EVAL
- Alpha-Beta Pruning is also possible (but less effective)
  - at MIN and MAX nodes as usual
  - at chance nodes, expected values can be bounded before all nodes have been searched if the value range is bounded

Games of Imperfect Information

- The players do not have access to the entire world state
  - e.g., card games, when opponent's initial cards are unknown
- We can calculate a probability for each possible deal
  - seems just like one big dice roll at the beginning of the game
- Intuitive Idea:
  - compute the minimax value of each action in each deal
  - choose the action with the highest expected value over all deals
- Main problem:
  - too many possible deals to do this efficiently

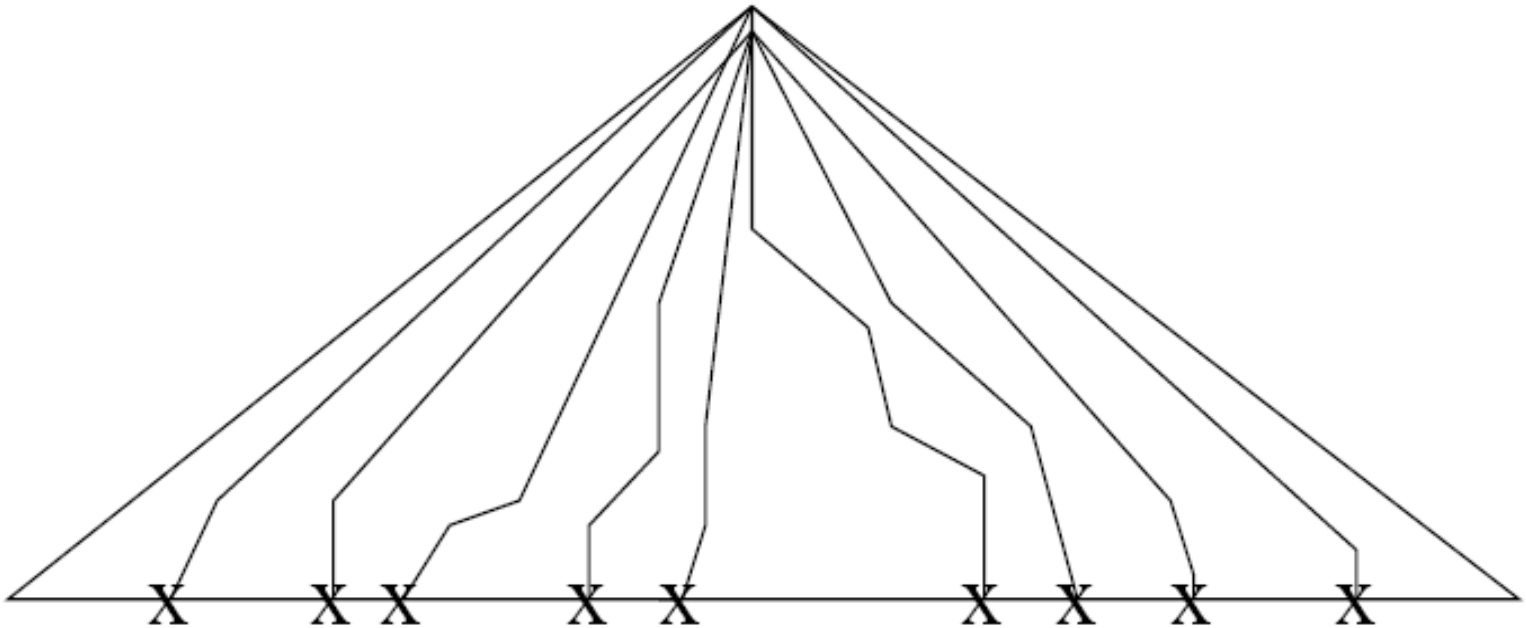
  → take a sample of all possible deals
- Example:
  - GIB (currently the best Bridge program) generates 100 deals consistent with bidding information (this also restricts!)
  - picks the move that wins the most tricks on average

Outline

- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - α-β pruning
  - NegaScout
- Real-time Game-Tree Search
  - evaluation functions
  - practical enhancements
  - selective search
- Games of imperfect information and games of chance
- Simulation Search
  - Monte-Carlo search
  - UCT search

Simulation Search – Key Idea

- The complete tree is not searchable
  - thus minimax/alpha-beta limit the depth of the search tree
    - search all variations to a certain depth
  - alternatively, we can limit the breadth of the search tree
    - sample some lines to the full depth



Simulation Search

- Algorithm Sketch:
  - estimate the expected value of each move by counting the number of wins in a series of complete games
  - at each chance node select one of the options at random (according to the probabilities)
  - at MAX and MIN nodes make moves (e.g., guided by a fast evaluation function)

- Examples:
  - roll-out analysis in Backgammon
    - play a large number of games from the same position
    - each game has different dice rolls
  - in Scrabble:
    - different draws of the remaining tiles from the bag
  - in card games (e.g., GIB in Bridge)
    - different distributions of the opponents' cards

Simulation Search

- Algorithm Sketch:
  - estimate the expected value of each move by counting the number of wins in a series of complete games
  - at each chance node select one of the options at random (according to the probabilities)
  - at MAX and MIN nodes make moves (e.g., guided by a fast evaluation function)

- Properties:
  - We need a fast algorithm for making the decisions at each MAX and each MIN node
    - the program plays both sides, of course
  - Often works well even if the program is not that strong

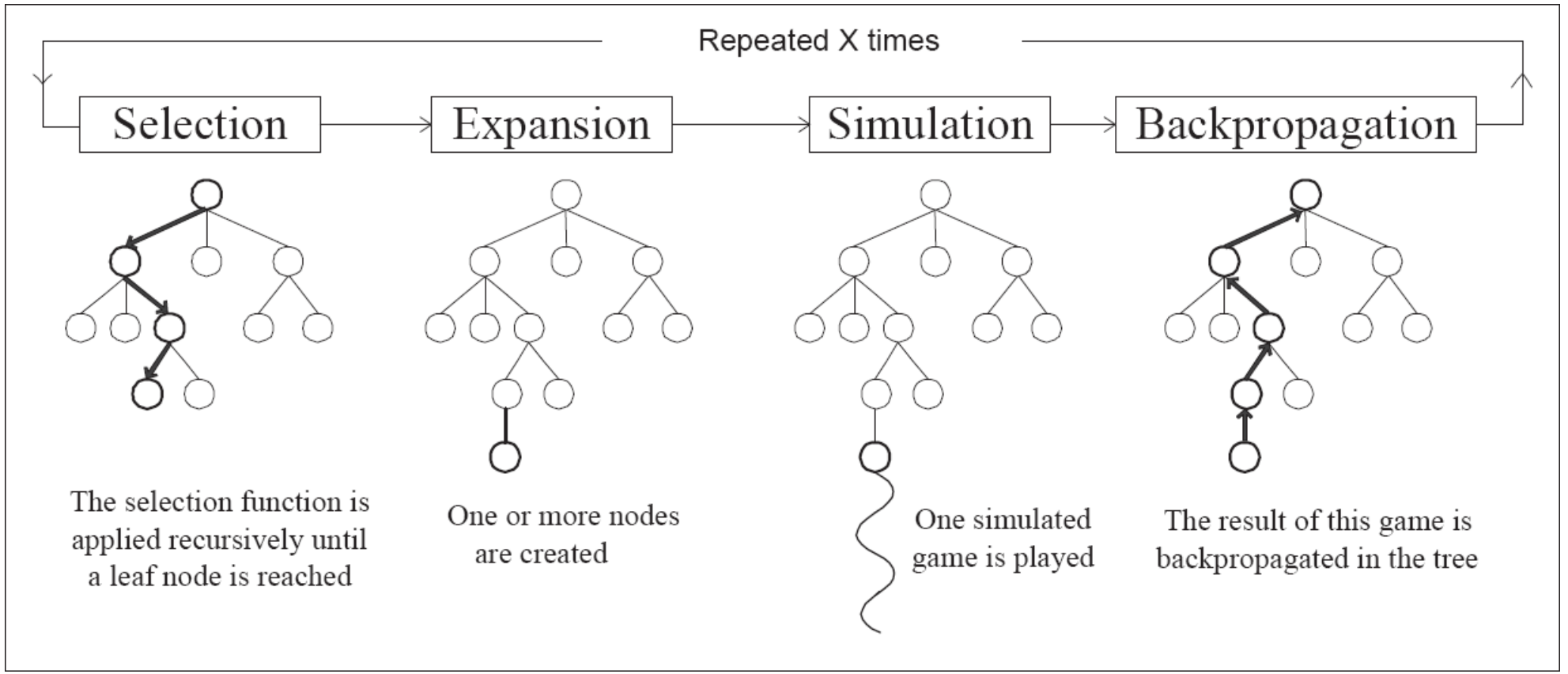
    → fast is possible
  - Easily parallelizable

Monte-Carlo Search

- Extreme case of Simulation search:
  - play a large number of games where both players make their moves randomly
  - average the scores of these games
  - make the move that has the highest average score
- Has been treen with some success in Go
  - e.g., Bruegmann 1993

Integrating Simulation Search and Game Tree Search

- Monte-Carlo Search can be integrated with conventional game-tree search algorithms:



G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3), 2008.

UCT Search
(Kocsis & Szepesvari, 2006)

- Selection
  - Select the node $s_{max} = \arg\max_{s \in \text{Successors}(n)} \text{value}(s) + C \cdot \sqrt{\frac{\ln \#\text{visits}(n)}{\#\text{visits}(s)}}$
  - Parameter C trades off between
    - **Exploitation**: Try to play the best possible move
      - maximize $\text{value}(s)$
    - **Exploration**: Try new moves to learn something new
      - s gets a high value when the number of visits in the node is low
        - in relation to the number of visits in the parent node n
  - Sometimes:
    - only use UCT if the node has been visited at least T times
      - frequently used value $T = 30$

- UCT is an adaptation of a solution to the Multi-Armed Bandit Problem to game tree search
  - you are in a Casino with k one-armed bandits with different winning probabilities
  - try to maximize your winnings

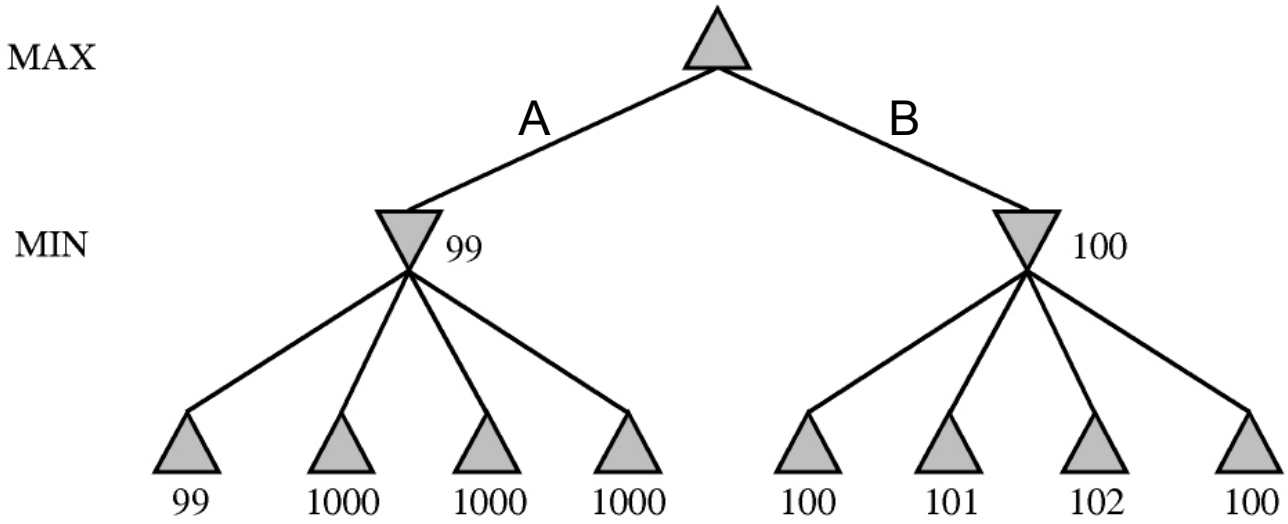
UCT Search
(Kocsis & Szepesvari, 2006)

- Expansion
  - add a randomly selected node to the game tree
- Simulation
  - perform one iteration of a Monte-Carlo search starting from the selected node
- Backpropagation
  - adapt $\text{value}(n)$ for each node n in the partial game tree
  - the value is just the average result of all games that pass through this node
- Move Choice
  - make the move that has been visited most often (reliability)
  - not necessarily the one with the highest value (high variance)

- UCT is currently very popular in Computer Go Research
  - e.g., MoGo (Gelly, Wang, Munos, Teytaud, 2006)

Minimax is Conservative

- It always assumes that the opponent plays its best response
  - according to MINIMAX's evaluation
- This may be a bad idea:

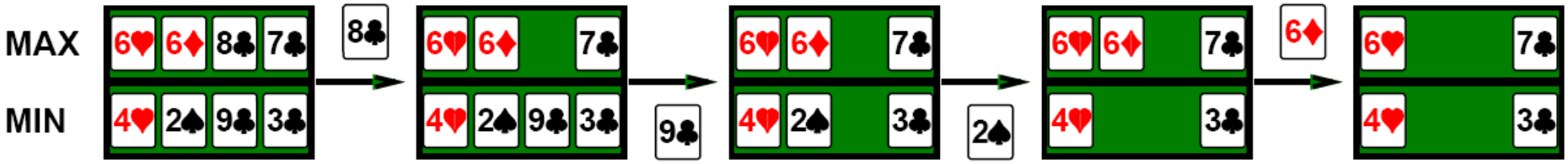


  - MAX will play move B
  - If there is a small chance that MIN does not play according to MAX's evaluation
    - because the evaluation is wrong or MIN makes a mistake

  then A would be the better choice!

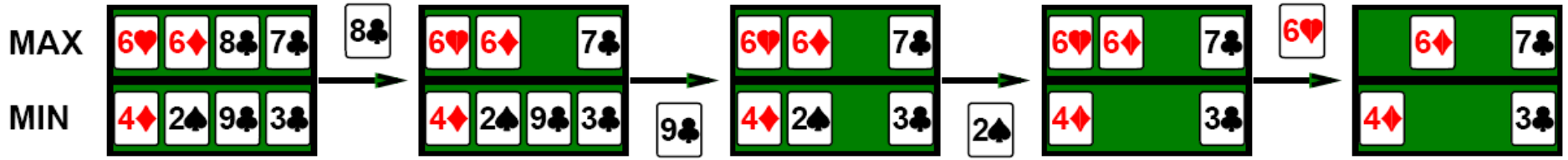
Expectimax is conservative too

Scenario a) MIN has 4♥



→ both players will make two tricks

Scenario b) MIN has 4♦

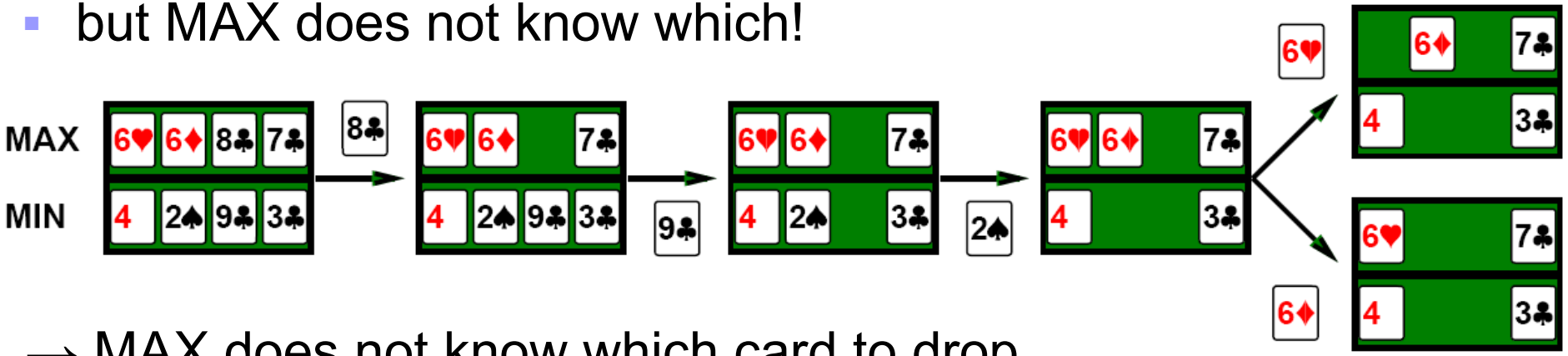


→ both players will make two tricks

Expectimax is conservative too

Scenario c) MIN has either 4♥ or 4♦

- but MAX does not know which!



→ MAX does not know which card to drop and has a 50% chance of losing the game!

- Lesson:
  - The intuition that the value of an action is the average of its value in all actual states is *wrong*!
  - the value of an action also depends on the agents' belief state
    - if I know that it is more probable that he has 4♥, the expected value should be adjusted accordingly
  - may lead to information-gathering or information-disclosing actions (e.g., signalling bids or unpredictable (random) play)

Opponent Modeling

- For simple games we know optimal solutions
  - Complete search through Minimax tree
  - Game-Theory: Nash-Equilibrium
- Optimal solutions are not Maximal!
  - Example: Roshambo (Rock/Paper/Scissors)

    - Optimal Solution: Pick a random move
    - clearly suboptimal against a player that always plays rock!

  → Roshambo Computer Tournament (1999, 2000)
- Opponent Modeling
  - try to predict the opponent's next move
  - try to predict what move the opponent predicts that your next move will be, ...
- For some games, opponent modeling is essential for success
  - Poker (Schaeffer et al., University of Alberta)

Perspective on Games: Pro

"Saying Deep Blue doesn't really think about chess is like saying an airplane doesn't really fly because it doesn't flap its wings"

Drew McDermott

Perspective on Games: Con

"Chess is the Drosophila of artificial intelligence. However, computer chess has developed much as genetics might have if the geneticists had concentrated their efforts starting in 1910 on breeding racing Drosophila. We would have some science, but mainly we would have very fast fruit flies."

John McCarthy

Additional Reading

- Jonathan Schaeffer. The Games Computers (and People) Play, *Advances in Computers 50* , Marvin Zelkowitz (ed.) Academic Press, pp. 189-266, 2000. http://www.cs.ualberta.ca/~jonathan/Papers/Papers/advances.ps
  - excellent survey paper
- Jonathan Schaeffer and Jaap van den Herik (eds.) *Chips Challenging Champions: Games, Computers and Artificial Intelligence*, North-Holland 2002.
  - very good collection of state-of-the-art papers
- Jonathan Schaeffer: *One Jump Ahead: Challenging Human Supremacy in Checkers*, Springer 1998.
  - non-technical first-hand account on the Chinook project
- Feng-Hsiung Hsu: *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*, Princeton 2002
  - non-technical first-hand account on Deep Blue