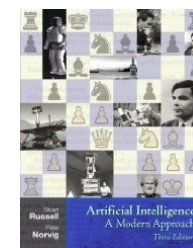


# Outline

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



Many slides based on  
Russell & Norvig's slides  
**Artificial Intelligence:  
A Modern Approach**

# 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

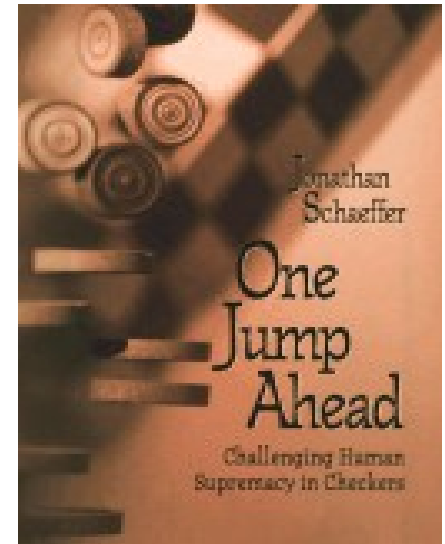
- Computer considers possible lines of play  
(Babbage, 1846)
- Algorithm for perfect play  
(Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation  
(Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First computer chess game  
(Turing, 1951)
- Machine learning to improve evaluation accuracy  
(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



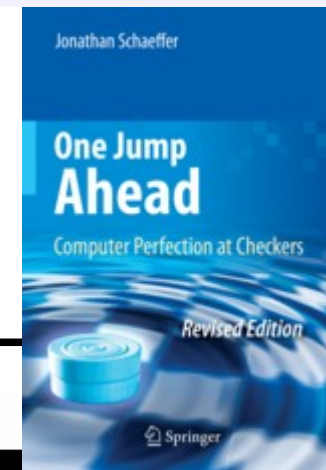
First computer to win human world championship!

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

# Chinook

- July 19 2007, after 18 years of computation:

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.5393>



## Scienceexpress

## Research Article

### Checkers Is Solved

Jonathan Schaeffer,\* Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, Steve Sutphen  
Department of Computing Science, University of Alberta, Edmonton, Alberta T6G 2E8, Canada.

\*To whom correspondence should be addressed. E-mail: [jonathan@cs.ualberta.ca](mailto:jonathan@cs.ualberta.ca)

The game of checkers has roughly 500 billion billion possible positions ( $5 \times 10^{20}$ ). The task of solving the game, determining the final result in a game with no mistakes made by either player, is daunting. Since 1989, almost continuously, dozens of computers have been working on solving checkers, applying state-of-the-art artificial intelligence techniques to the proving process. This paper announces that checkers is now solved: **perfect play by both sides leads to a draw.** This is the most challenging popular game to be solved to date, roughly one million times more complex than Connect Four. Artificial intelligence technology has been used to generate strong heuristic-based game-playing programs, such as DEEP BLUE for chess. Solving a game takes this to the next level, by replacing the heuristics with perfection.

1992, over 200 processors were being used simultaneously. The end result is one of the longest running computations completed to date.

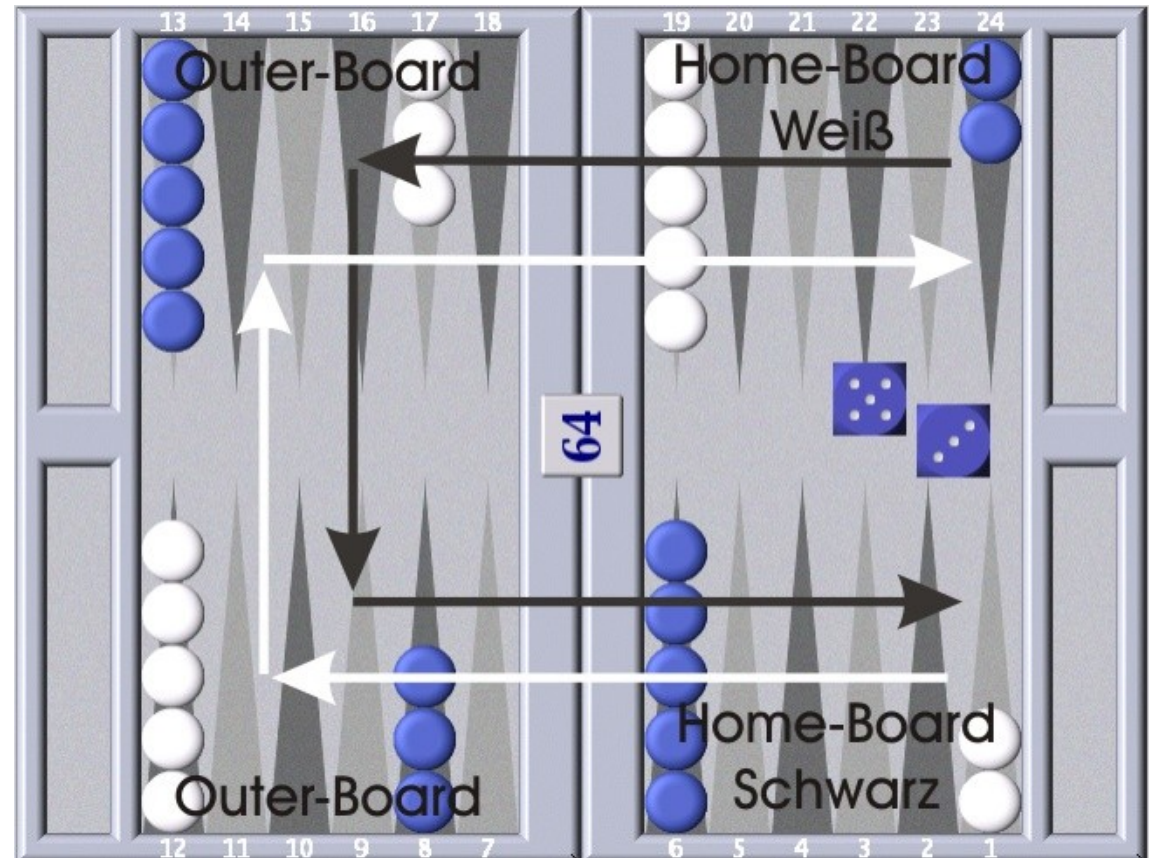
This paper announces that checkers has been weakly solved. From the starting position (Fig. 1A), we have a computational proof that checkers is a draw. The proof consists of an explicit strategy that never loses – the program can achieve at least a draw against *any* opponent, playing either the black or white pieces. That checkers is a draw is not a surprise; grandmaster players have conjectured this for decades.

The checkers result pushes the boundary of artificial intelligence (AI). In the early days of AI research, the easiest path to achieving high performance was seen to be emulating the human approach. This was fraught with difficulty, especially the problems of capturing and encoding human knowledge. Human-like strategies are not



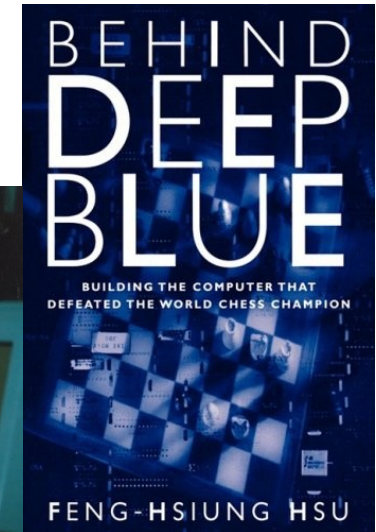
# Backgammon

- 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



## Kasparov

5'10"

176 lbs

34 years

50 billion neurons

2 pos/sec

Extensive

Electrical/chemical

Enormous

## Name

Height

Weight

Age

Computers

Speed

Knowledge

Power Source

Ego

## Deep Blue

6' 5"

2,400 lbs

4 years

512 processors

200,000,000 pos/sec

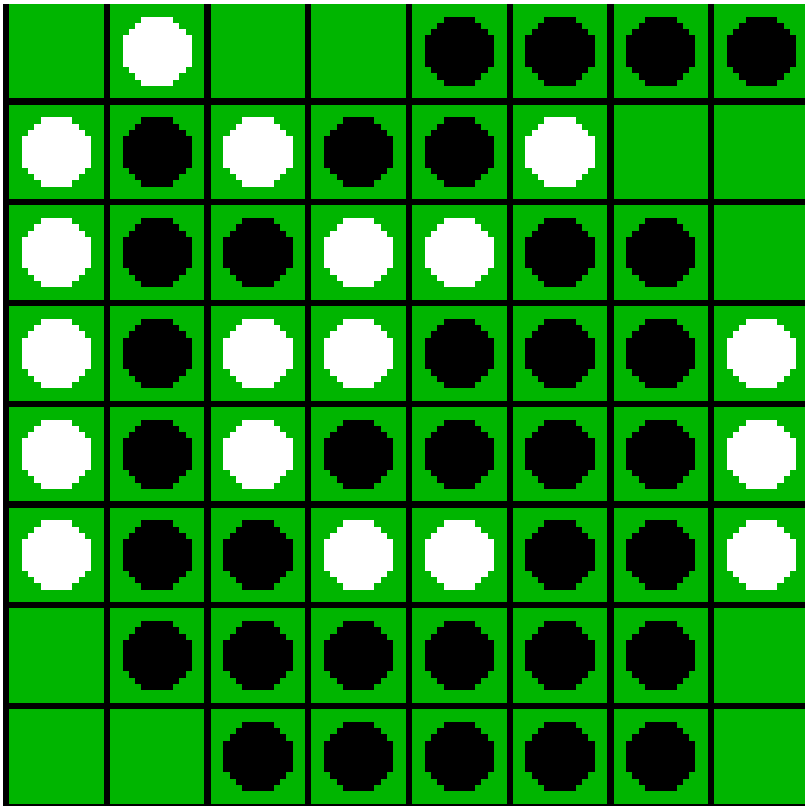
Primitive

Electrical

None

<http://www.wired.com/wired/archive/9.10/chess.htm>

# Reversi/Othello



Name: Takeshi Murakami  
 Title: World Othello Champion  
 1997: Lost 6-0 against Othello  
 Program Logistello

# Computer Go



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

# Computer Go



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

**Active Area of research**  
 Methods relying on Monte  
 Carlo tree search gave a  
 strong boost in performance,  
 Best Humans are still out of  
 reach on the 19x19 board.

Gave Handtalk a 9  
 stone handicap and  
 still easily beat  
 the program, 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**
  - **$\alpha$ - $\beta$  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

# 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

# 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 player A
  - i. mark all terminal positions that are won for A
  - ii. mark all positions where A is to move and can make a move that leads to a marked position
  - iii. mark all positions where B 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 B
  - analogous to 1.
3. All remaining positions are draw



# Retrograde Analysis

- Several Games have been solved completely using RA
  - Tic-Tac-Toe, Go-Moku, Connect-4, ...
- For other games, solutions for partial
  - Chess
    - All endgames with 7 pieces (=2 kings + 5 additional pieces) are solved since 2012
      - ca. 500 000 000 000 000 positions had to be stored, even when considering symmetries etc.
    - Accessible on-line <http://tb7.chessok.com/>
    - <https://chessprogramming.wikispaces.com/Endgame+Tablebases>
  - Checkers
    - In checkers, databases with up to 10 pieces were crucial for (weakly) solving the game
- Overall, RA is too complex for most games
  - Impossible to store all possible game states

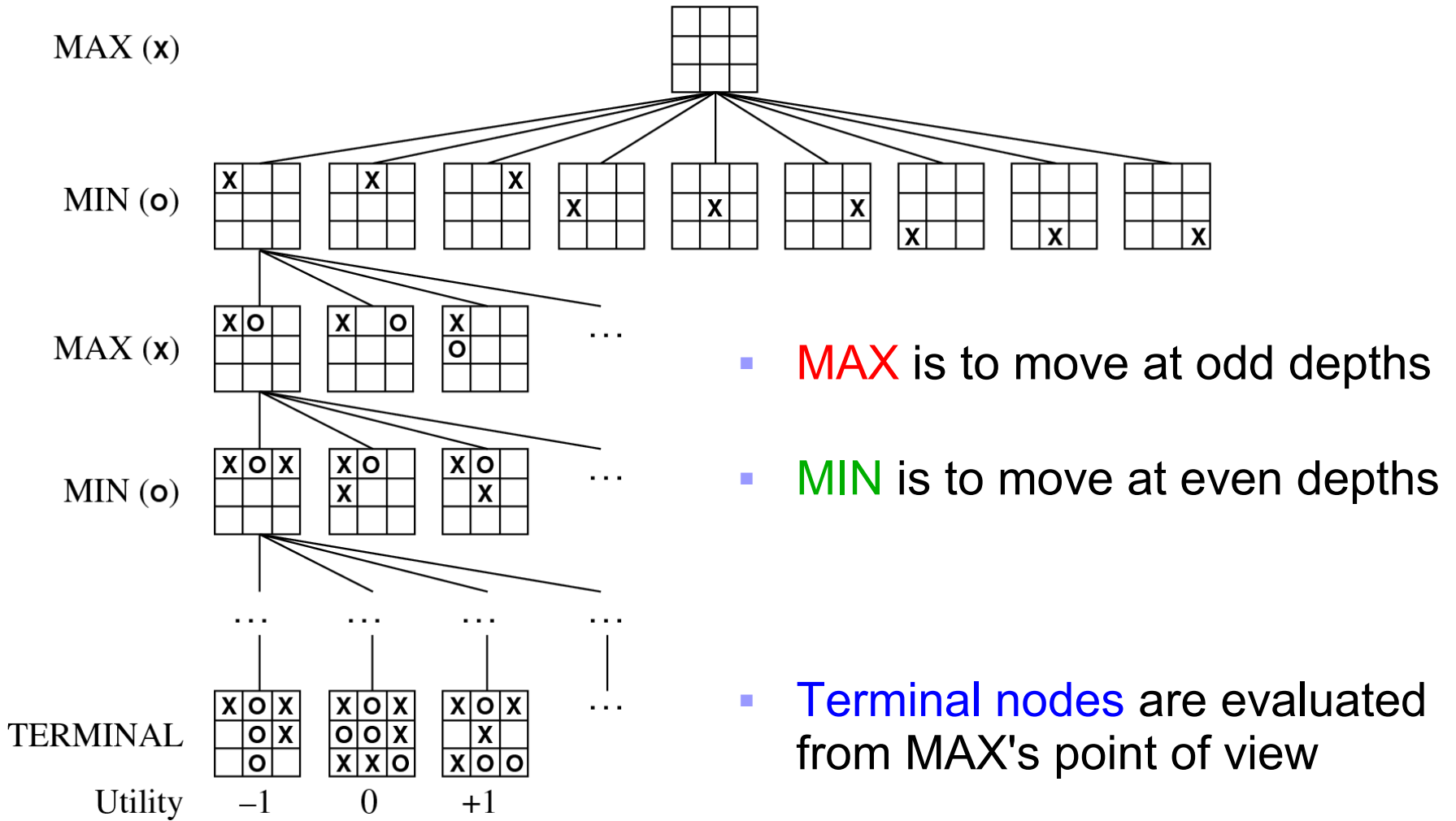
# Status Quo in Game Playing

- Solved
  - Tic-Tac-Toe, Connect-4, Go-Moku, 9-men Morris
  - Most recent addition: Checkers is a draw
    - Solved with 18 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://timkr.home.xs4all.nl/chess2/diary\\_16.htm](http://timkr.home.xs4all.nl/chess2/diary_16.htm)
- World-Championship strength
  - Chess, Backgammon, Scrabble, Othello
- Human Supremacy
  - Go, Shogi, Bridge, Poker (probably the next to fall)

# Game setup

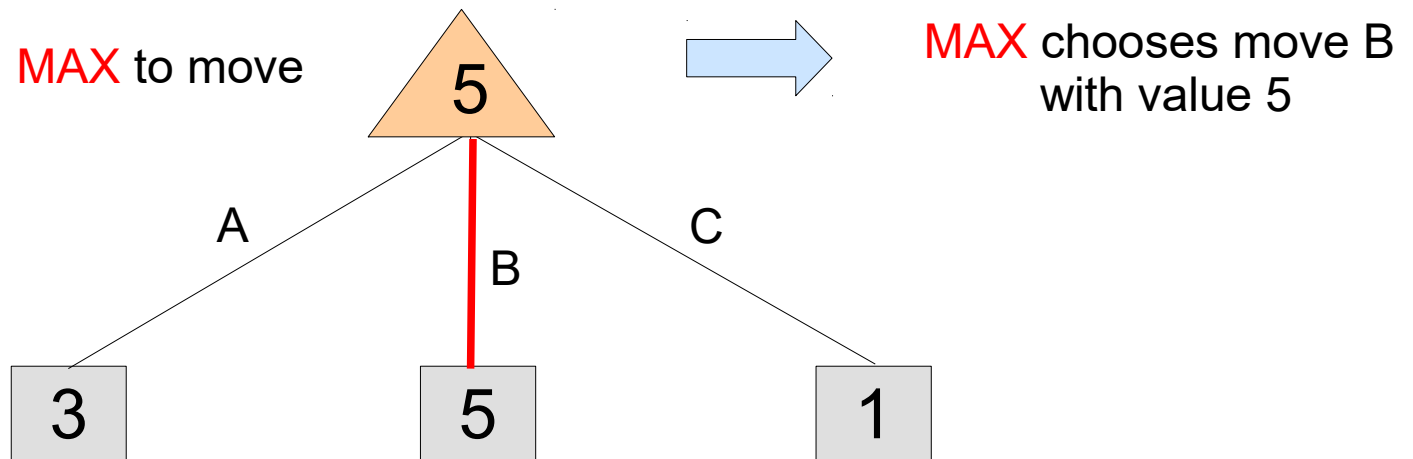
- Two players: MAX and MIN
  - MAX moves first and they take turns until the game is over.
    - **ply**: a half-move by one of the players
    - **move**: two plies, one by MAX and one by MIN
  - Winner gets award, loser gets penalty.
- Games as search:
  - **Initial state**:
    - e.g., board configuration of chess
  - **Successor function**:
    - list of (move,state) pairs specifying legal moves.
  - **Terminal test**:
    - Is the game finished?
  - **Utility function** (*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



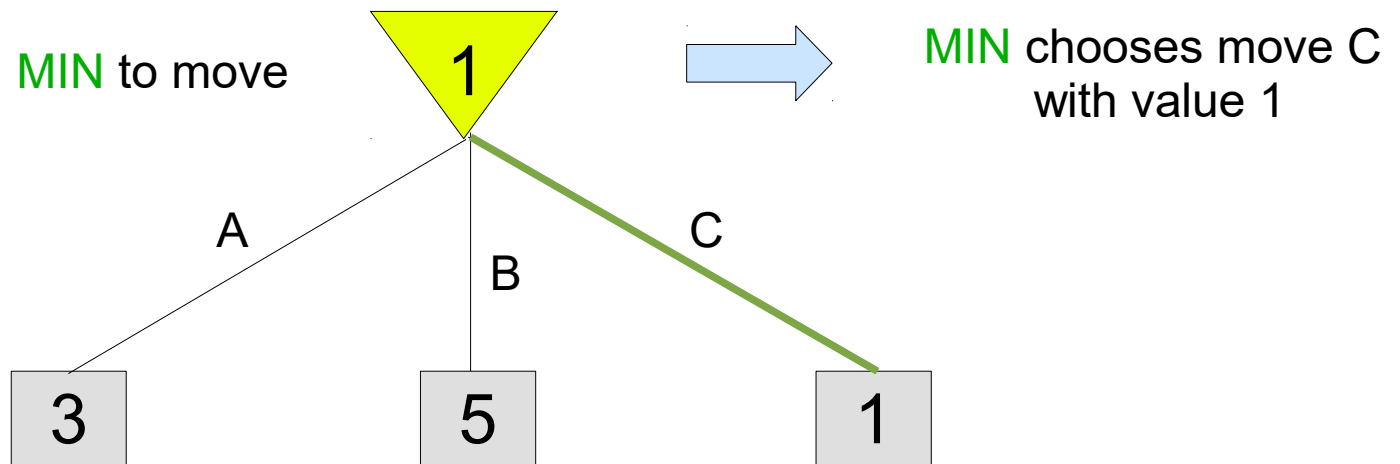
# Optimal strategies

- 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 from 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 from 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

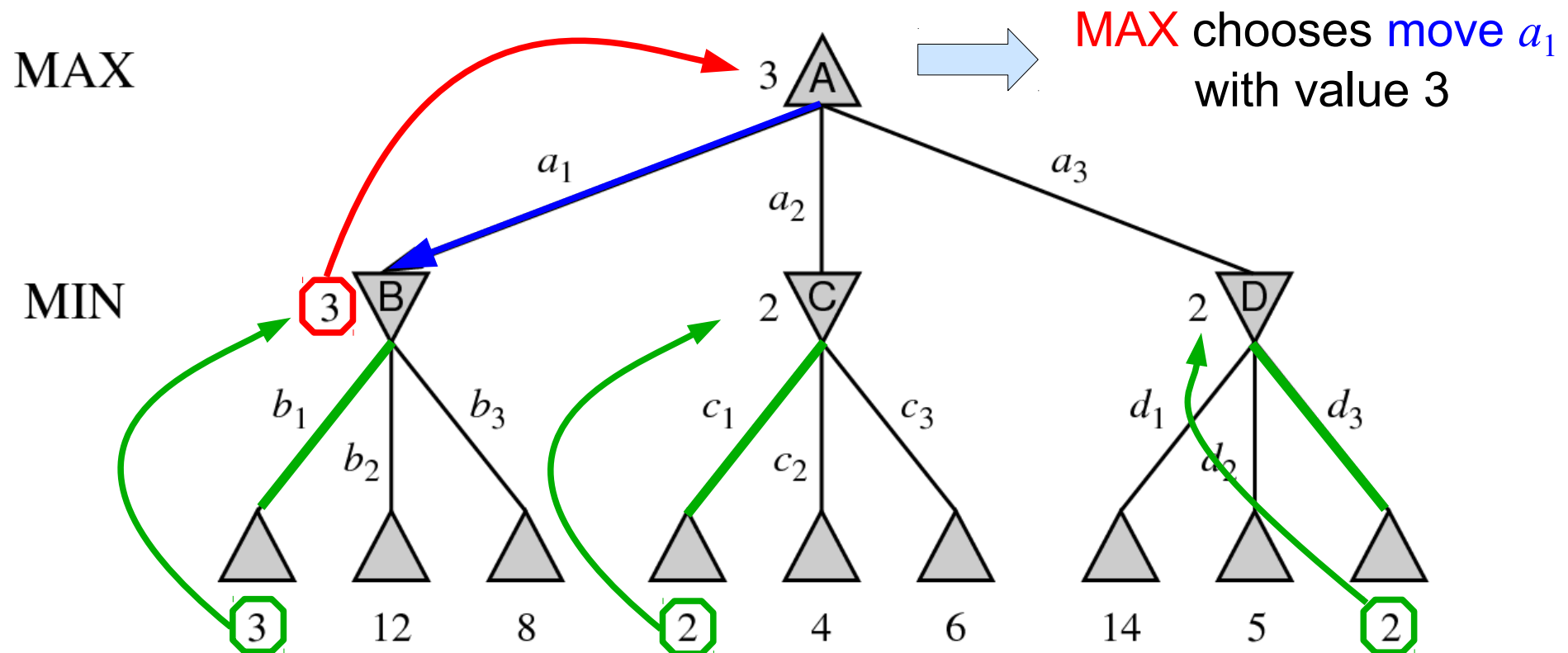


# Optimal strategies

- 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 from 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:

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

# 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

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

**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*)

$v \leftarrow -\infty$

**for** *a, s* in SUCCESSORS(*state*) **do**  $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$

**return** *v*

---

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

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

$v \leftarrow \infty$

**for** *a, s* in SUCCESSORS(*state*) **do**  $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$

**return** *v*

# 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**

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

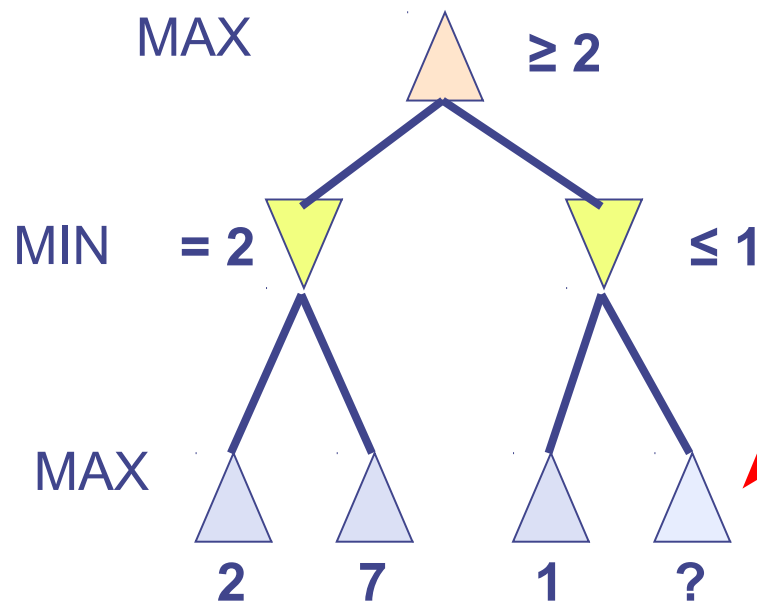
# Properties of Minimax Search

- **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 ( $\rightarrow$  *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  $\rightarrow O(m \cdot b)$

# 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  $\alpha$
- **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  $\beta$

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

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

**return** action *a* which has value *v* and *a, s* is in SUCCESSORS(*state*)

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value

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

$\alpha$ , the value of the best alternative for MAX along the path to *state*

$\beta$ , 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**

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

**if**  $v \geq \beta$  **then return** *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

**return** *v*

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

$v \leftarrow +\infty$

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

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

**if**  $v \leq \alpha$  **then return** *v*

$\beta \leftarrow \text{MIN}(\beta, v)$

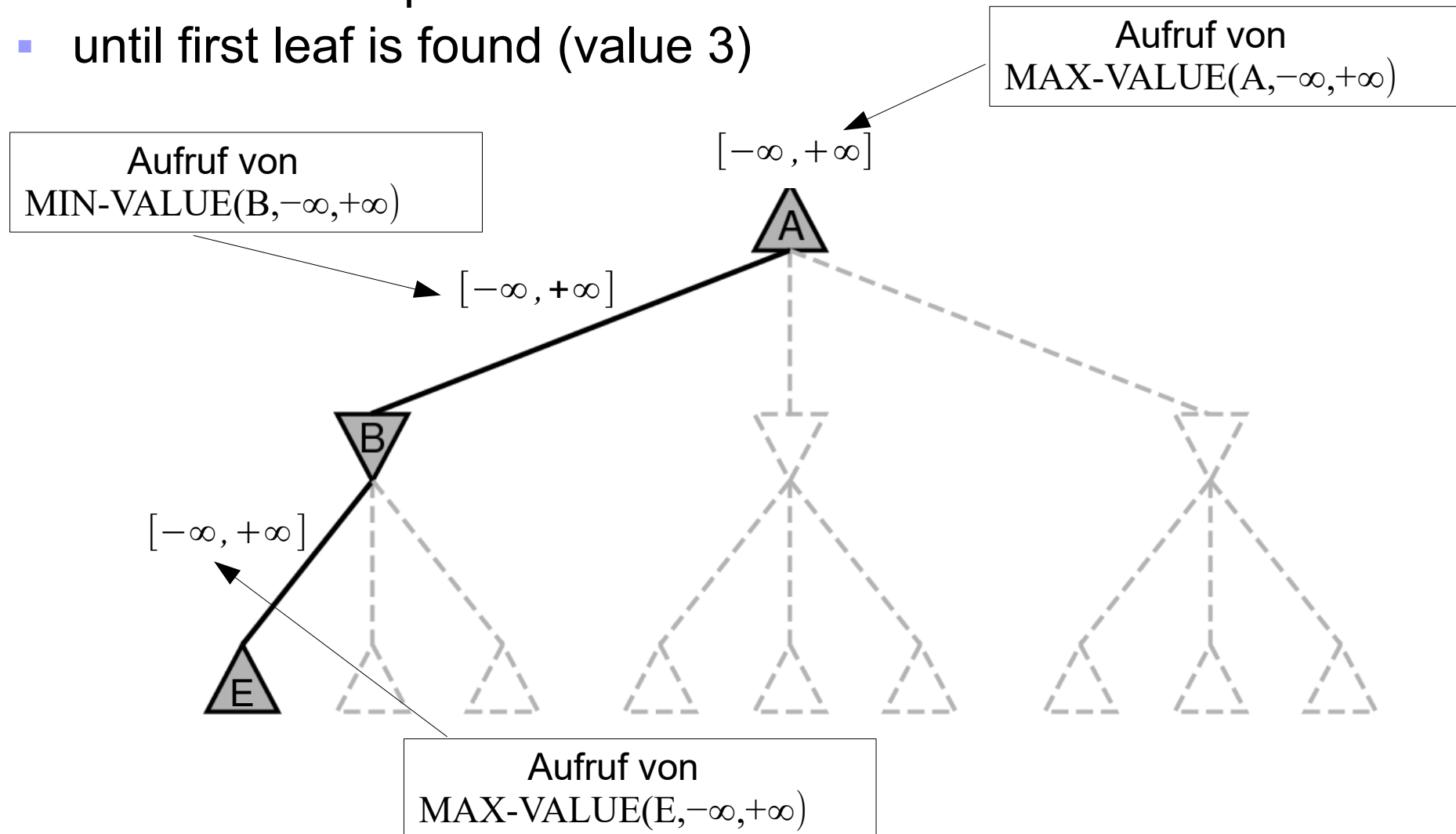
**return** *v*

**function** MIN-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value

same as MAX-VALUE but with roles of  $\alpha, \beta$  reversed

# Example: Alpha-Beta

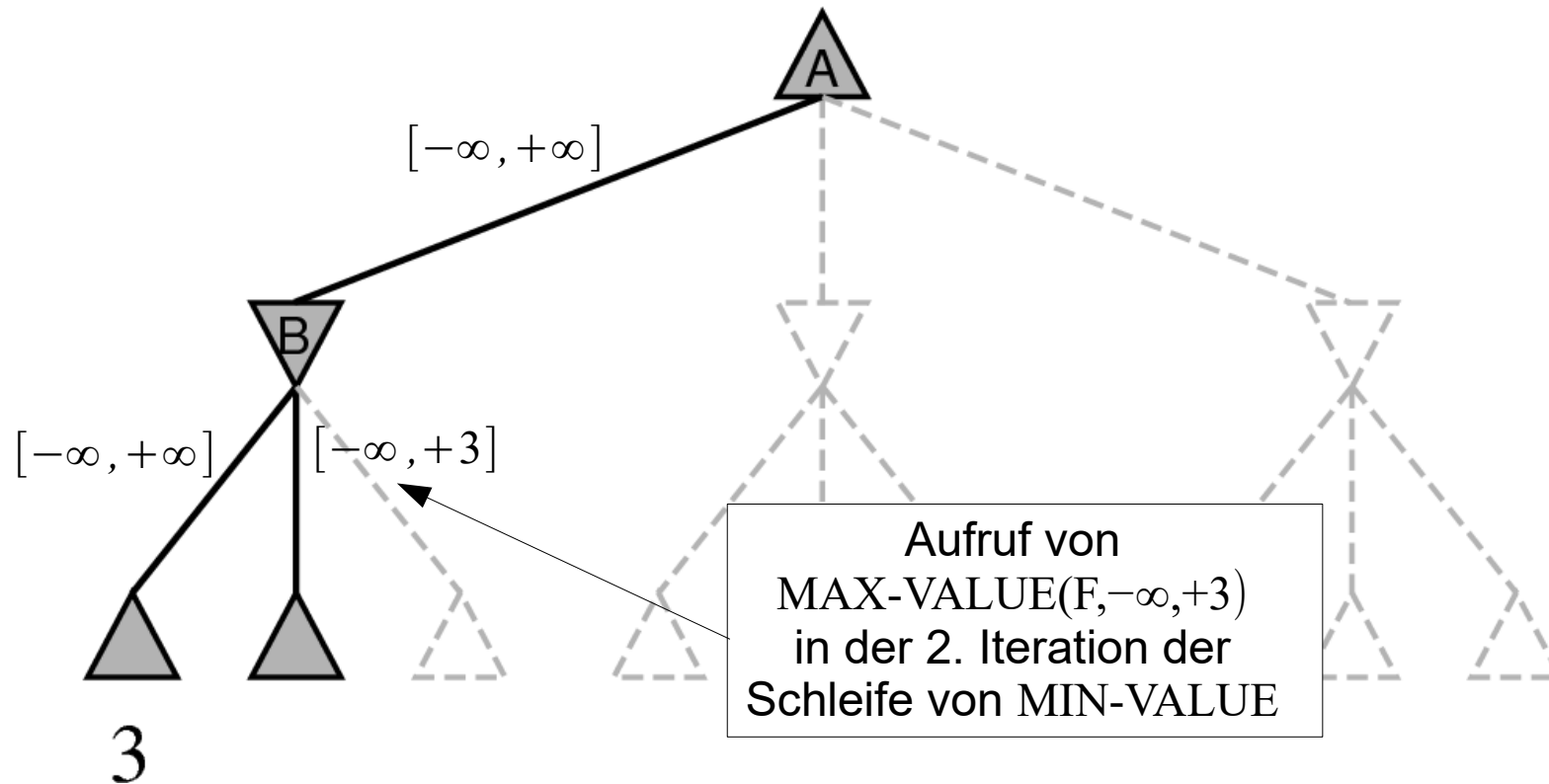
- The window is initialized with  $[-\infty, +\infty]$
- search runs depth-first
- until first leaf is found (value 3)





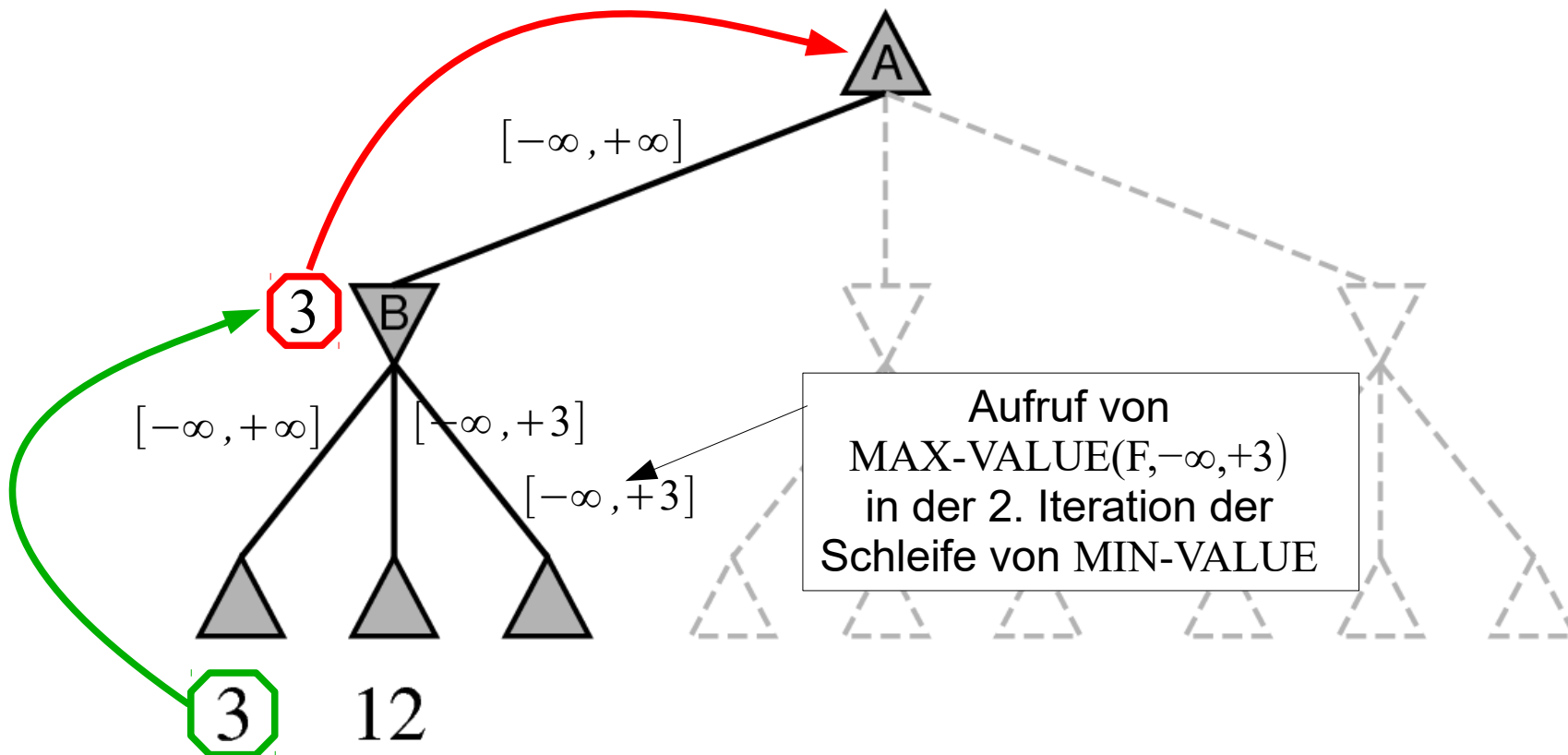
# Example: Alpha-Beta

- 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)



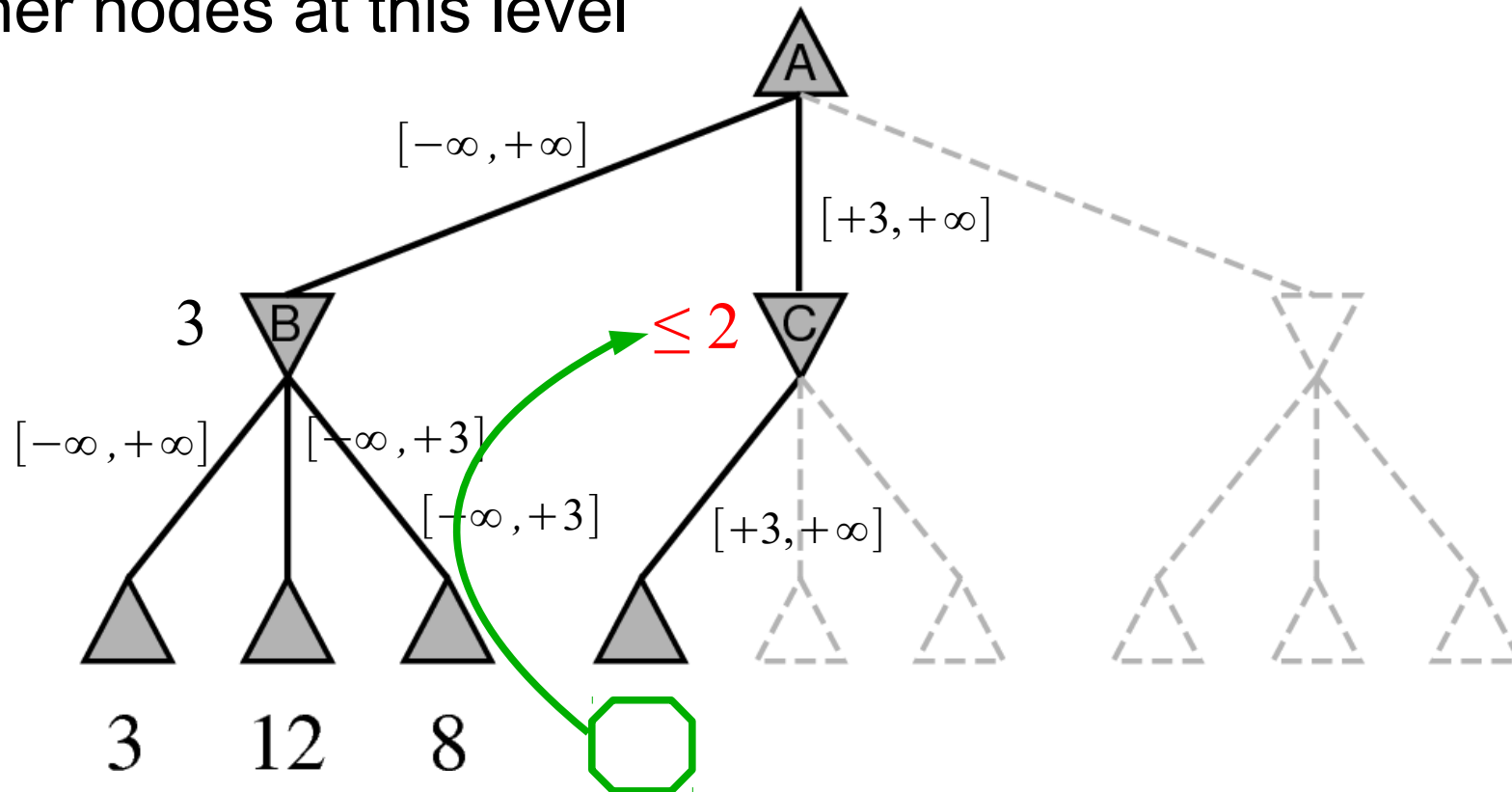
# Example: Alpha-Beta

- 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



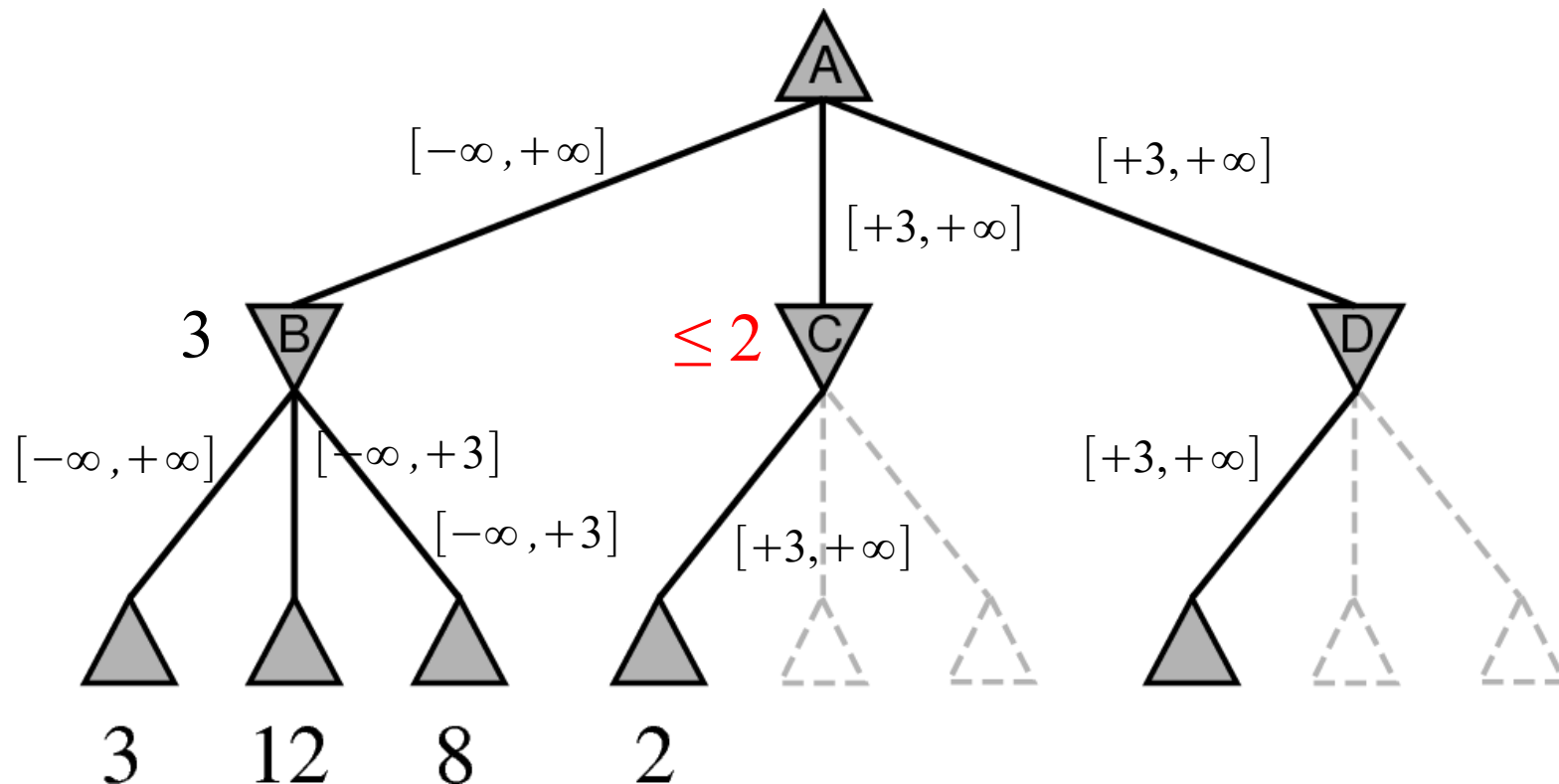
# Example: Alpha-Beta

- 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



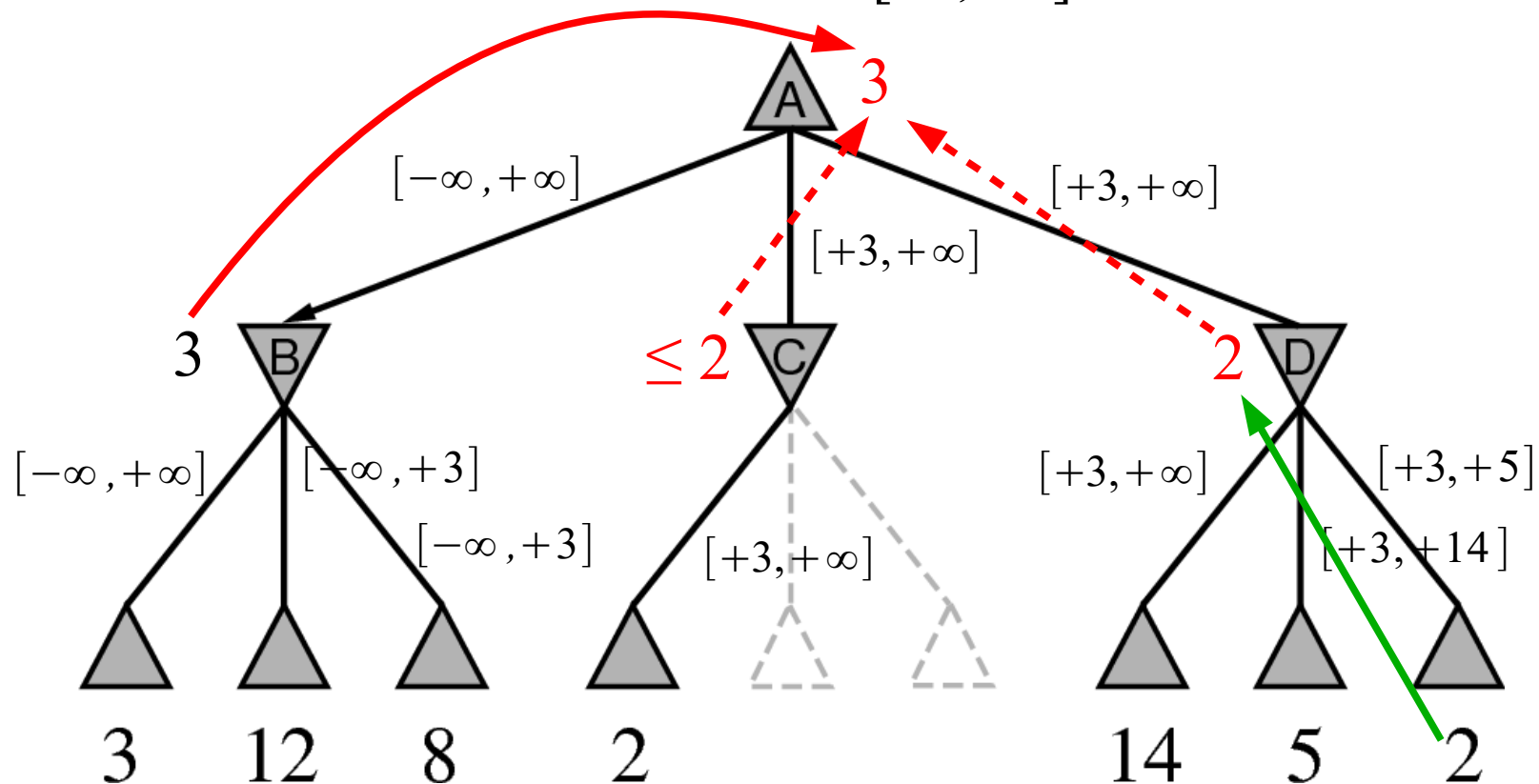
# Example: Alpha-Beta

- 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



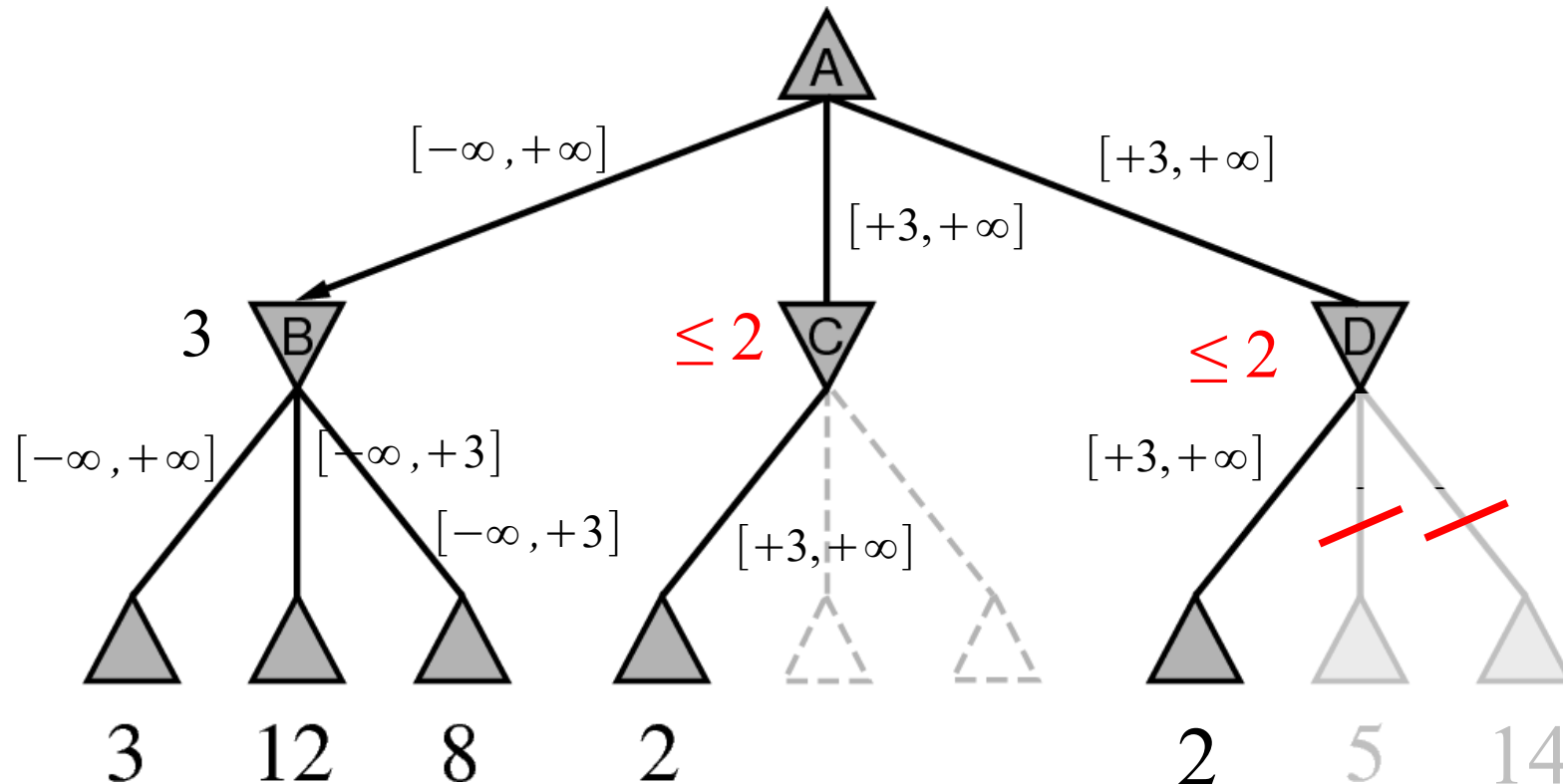
# Example: Alpha-Beta

- 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  $[+3, +5]$

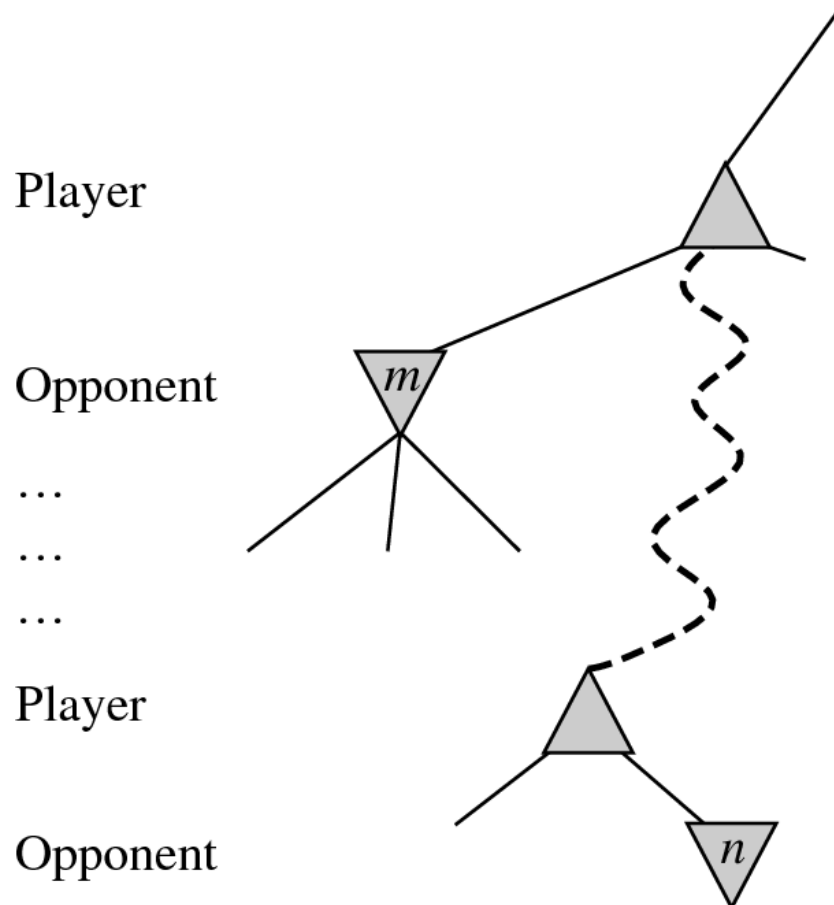


# Evaluation Order

- Note that the order of the evaluation of the nodes is crucial
  - e.g., if in node D, the node with evaluation 2 is searched first, another cutoff would have been possible
- **good move order** is crucial for **good performance**



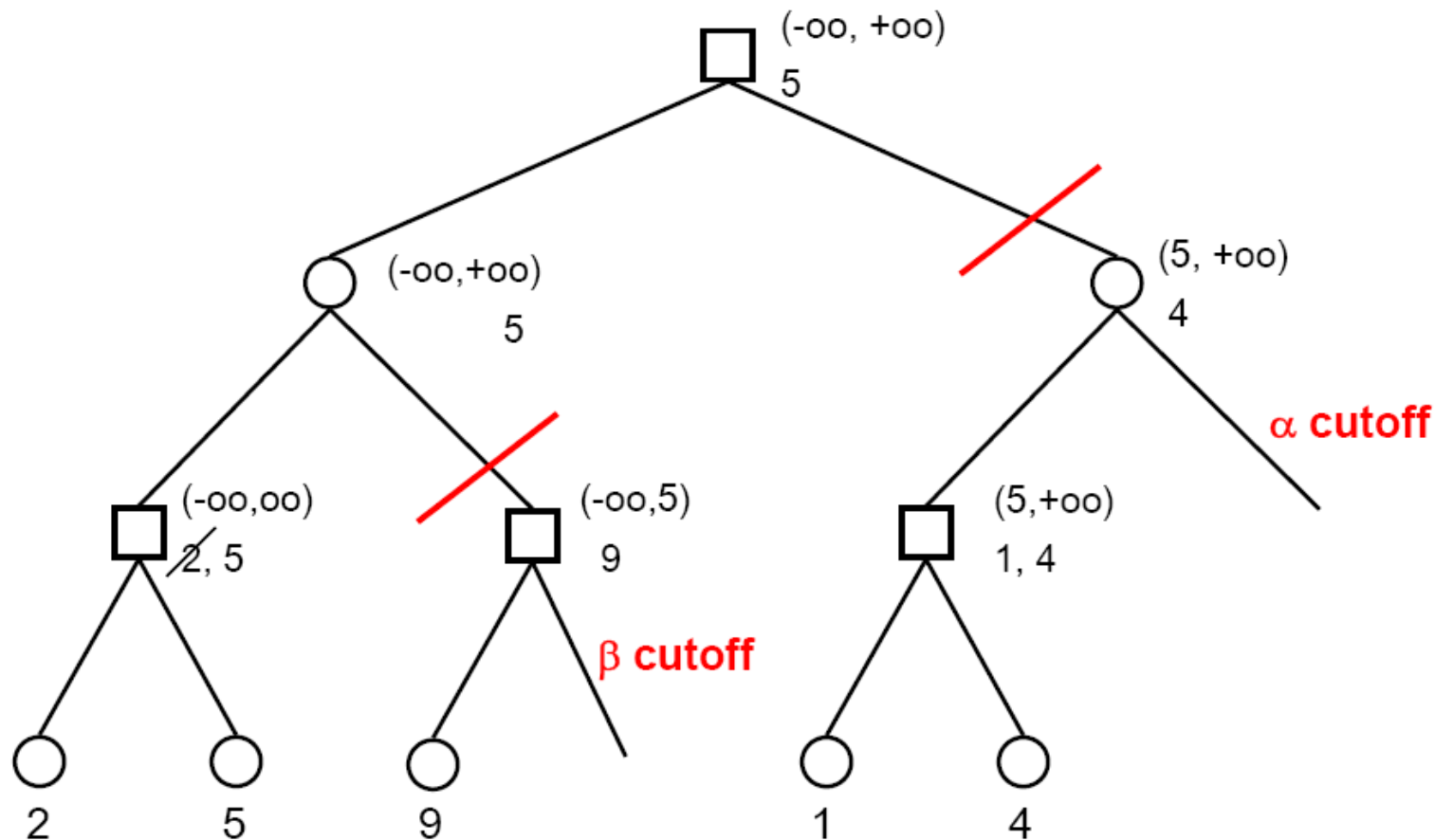
# General Alpha-Beta Pruning



- 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

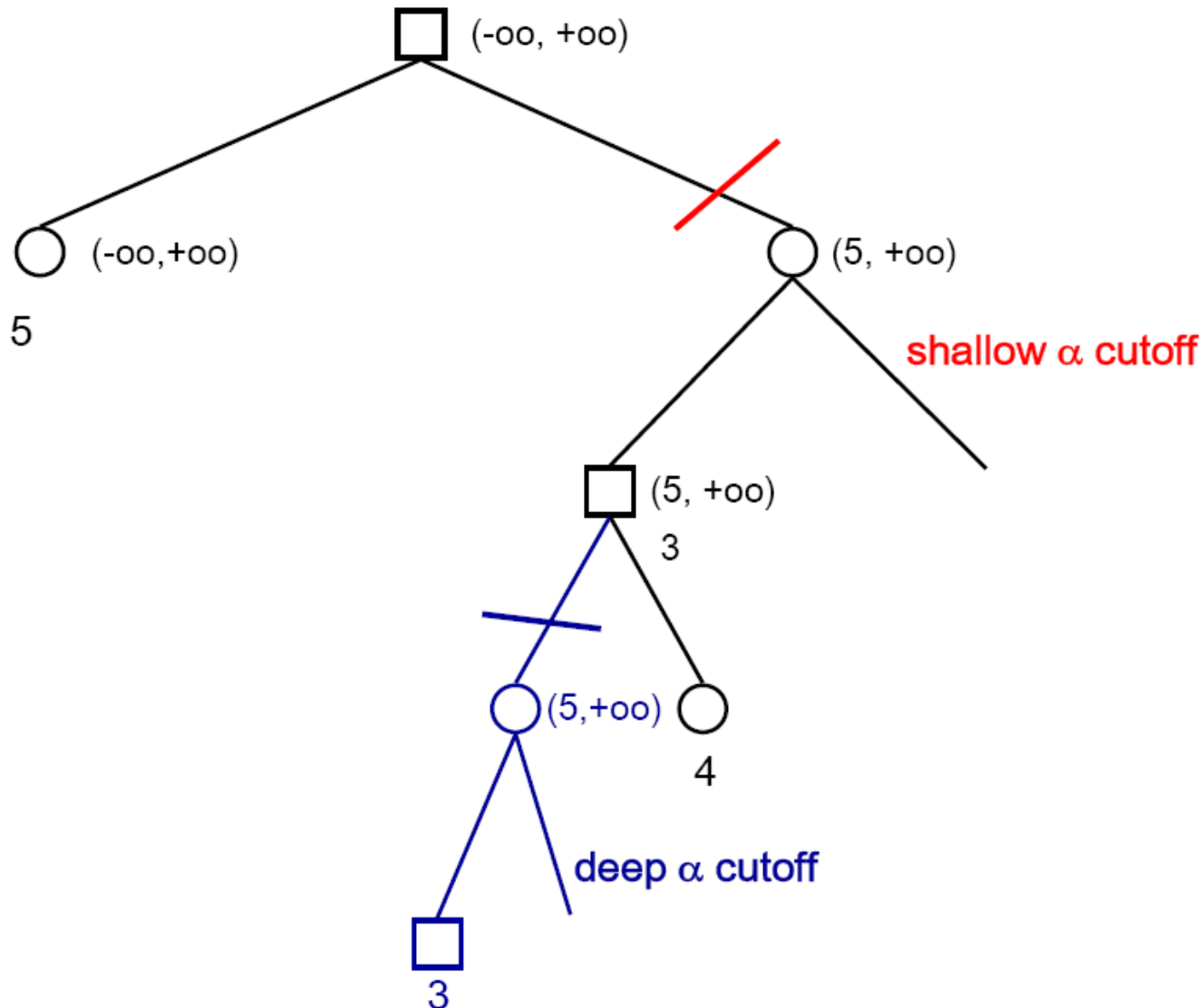
- Of course, cutoffs can also occur at MAX-nodes



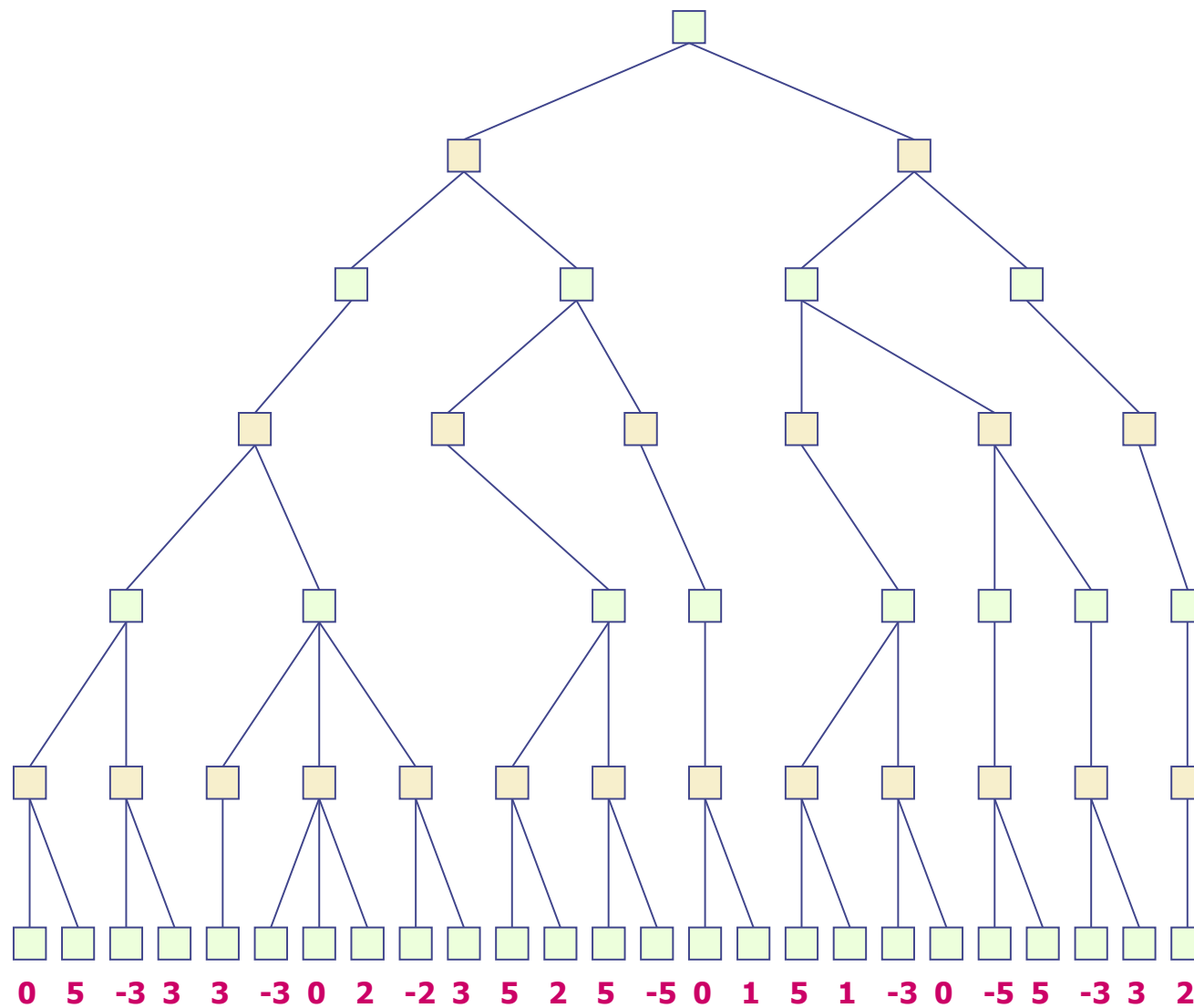


# Shallow vs. Deep Cutoffs

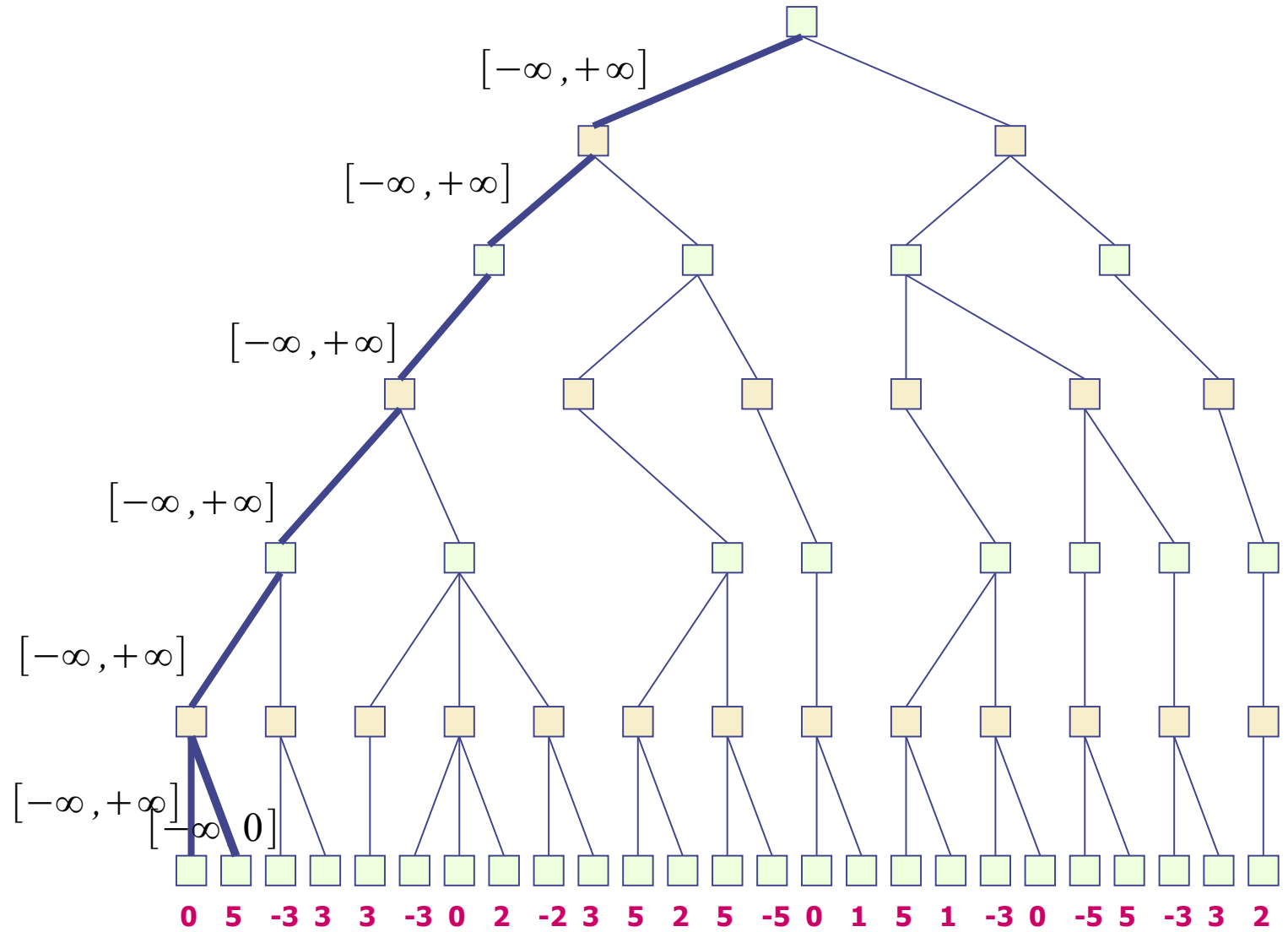
- Cutoffs may occur arbitrarily deep in (sub-)trees



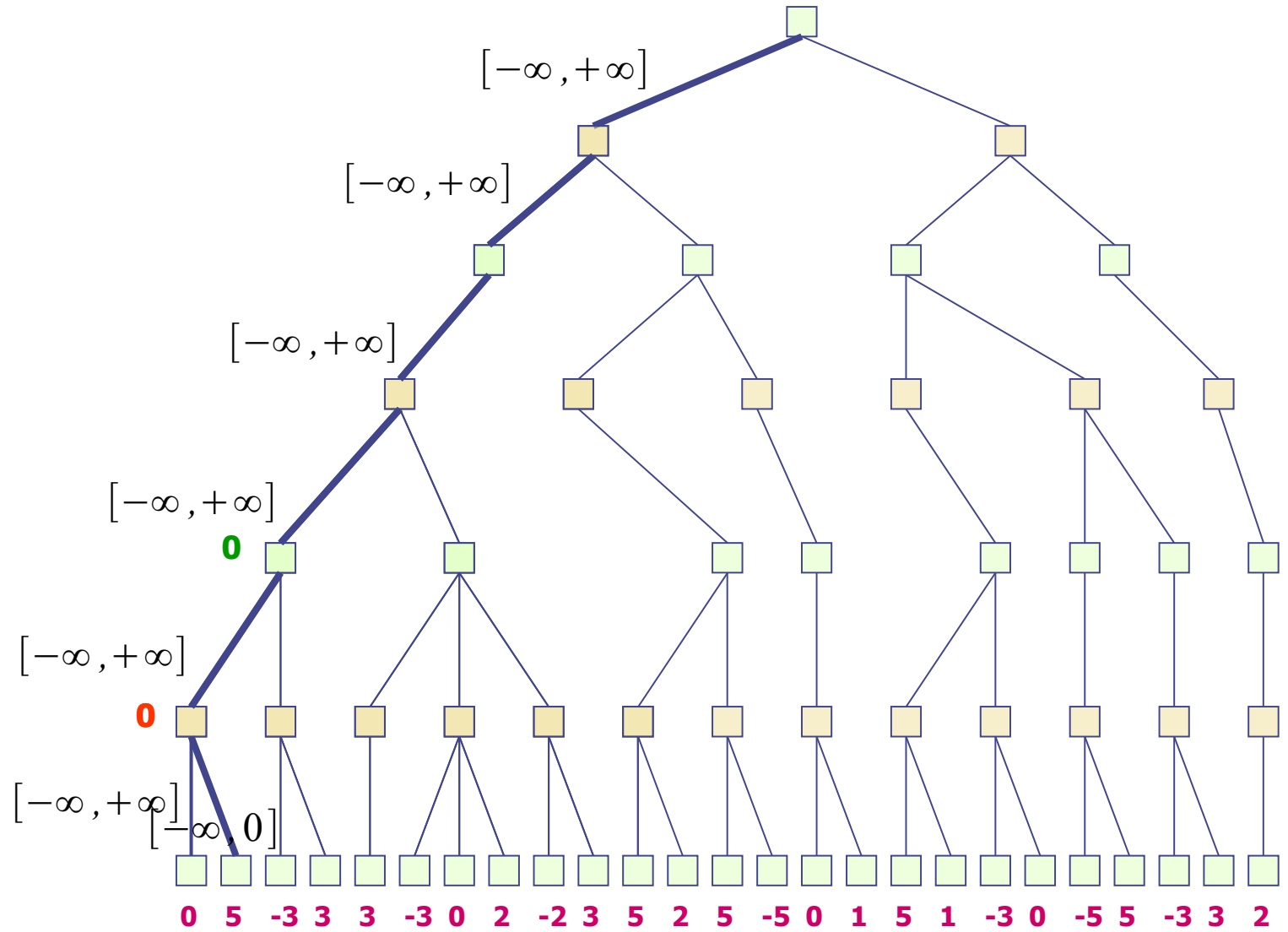
# Alpha-Beta Example



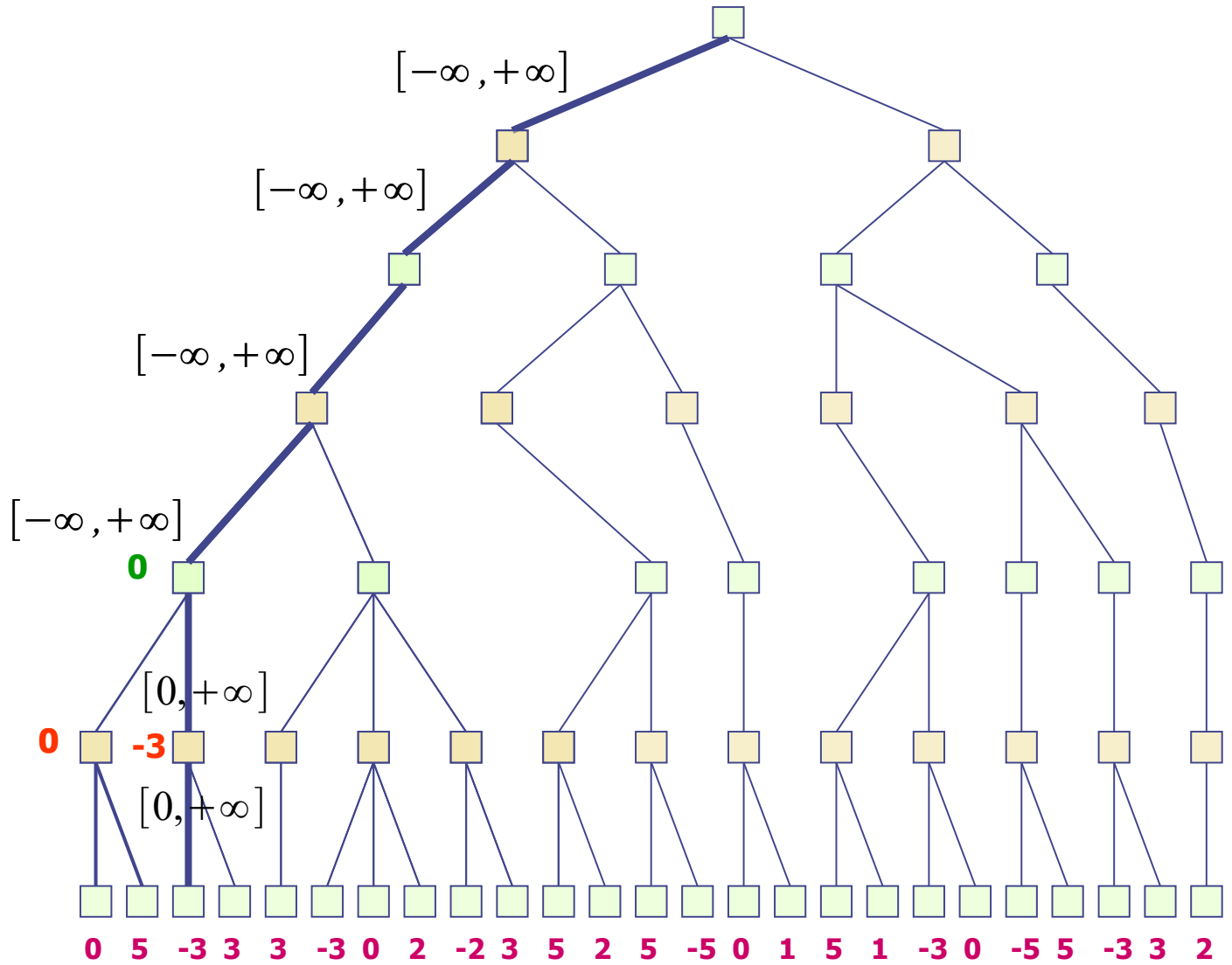
# Alpha-Beta Example



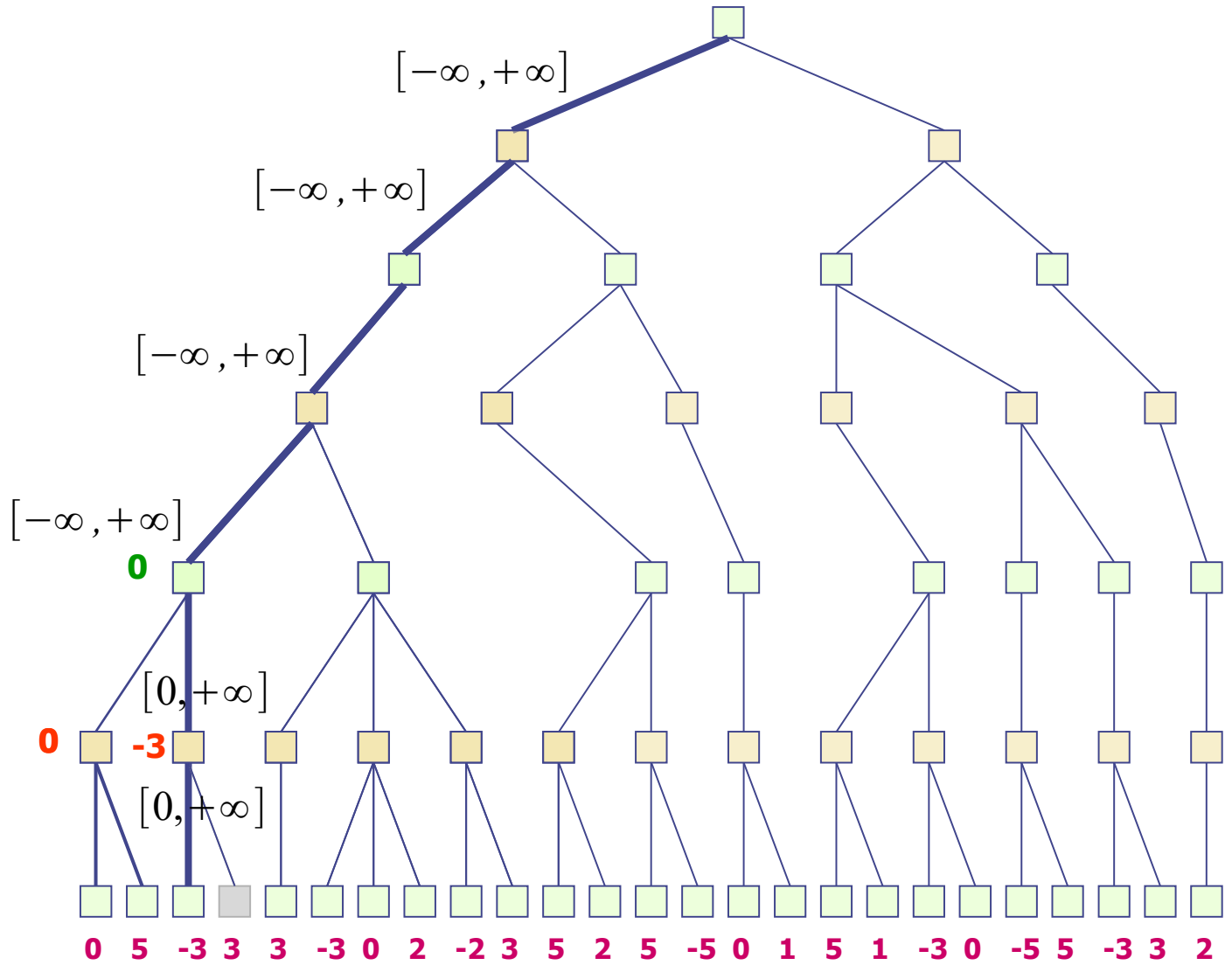
# Alpha-Beta Example



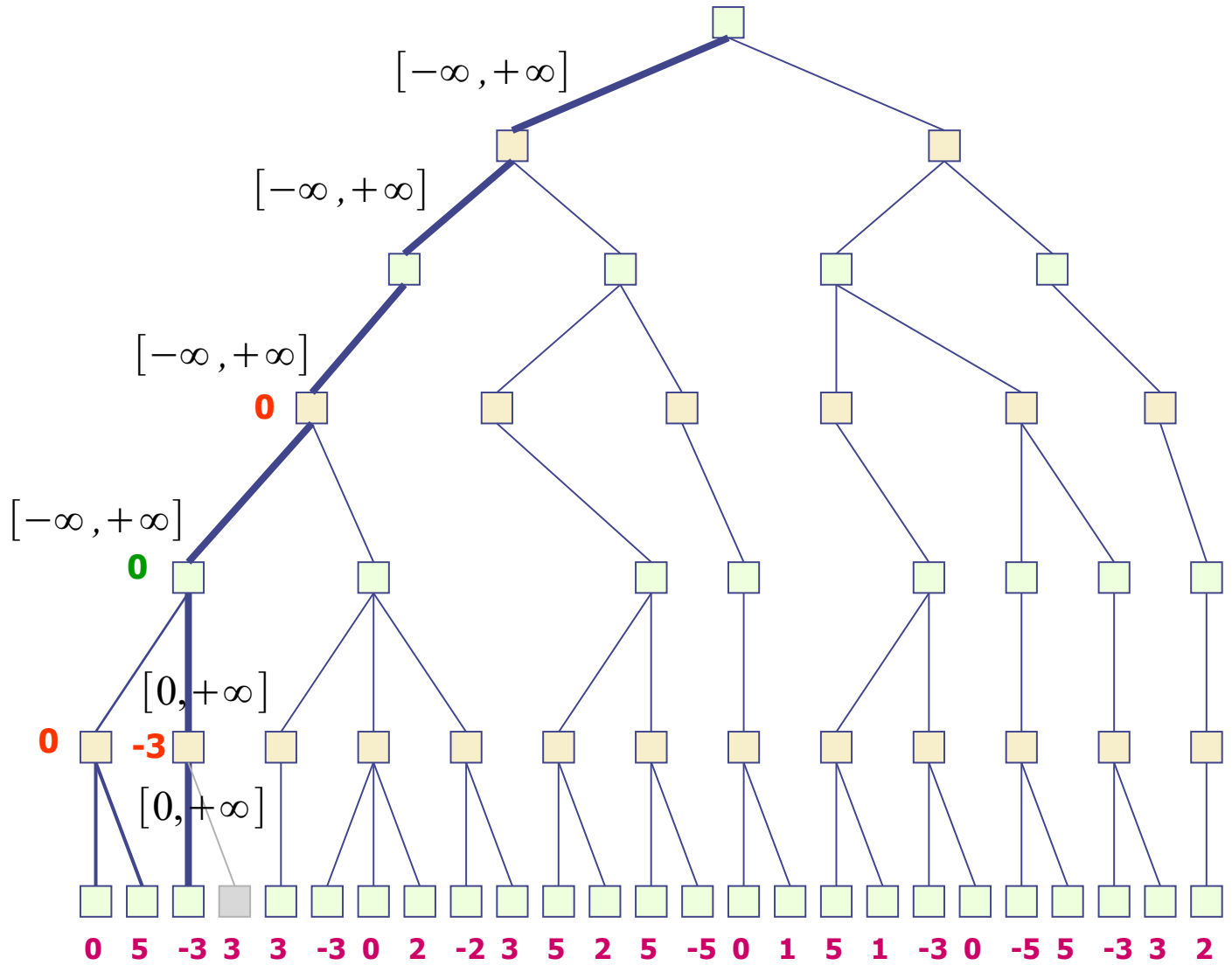
# Alpha-Beta Example



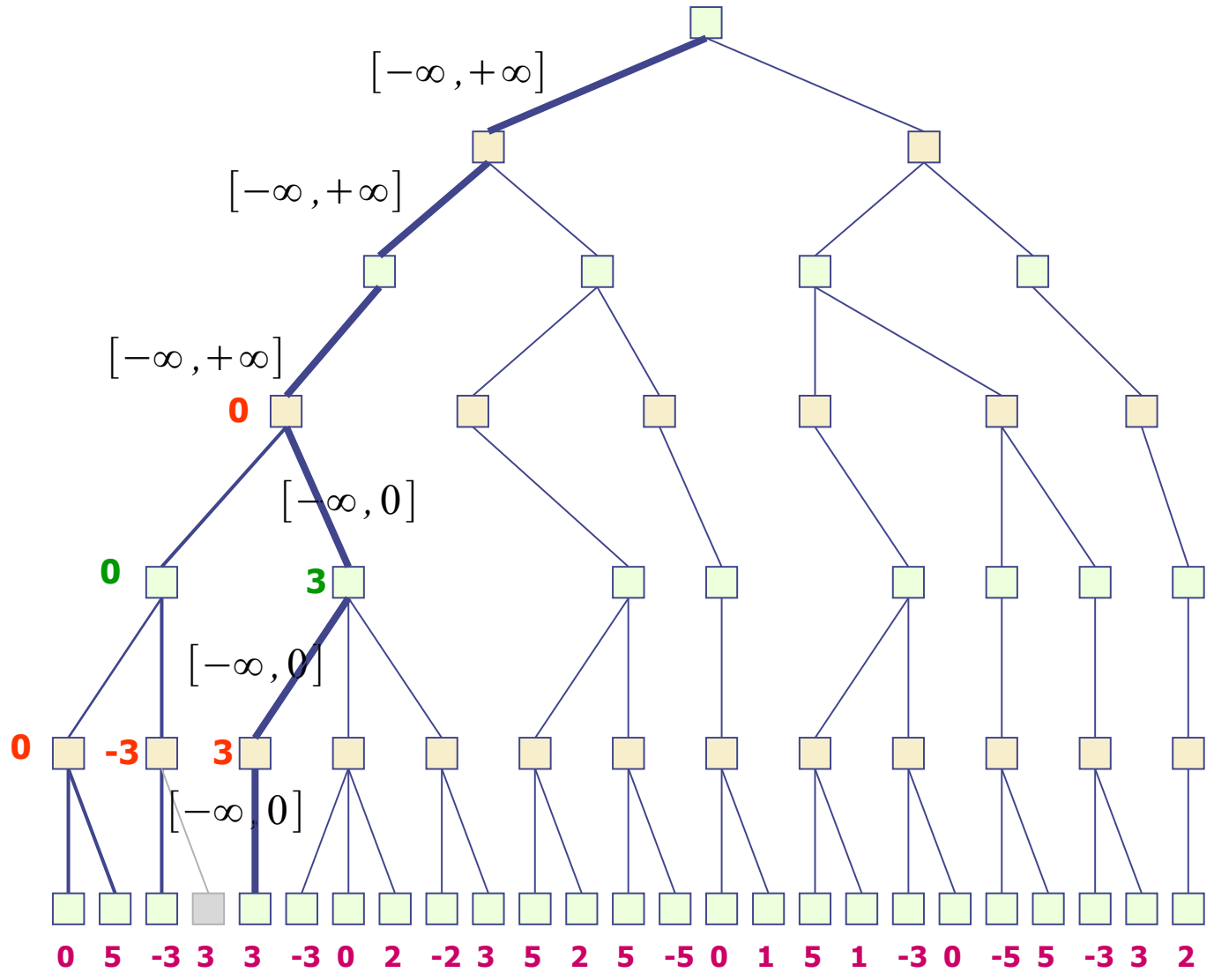
# Alpha-Beta Example



# Alpha-Beta Example

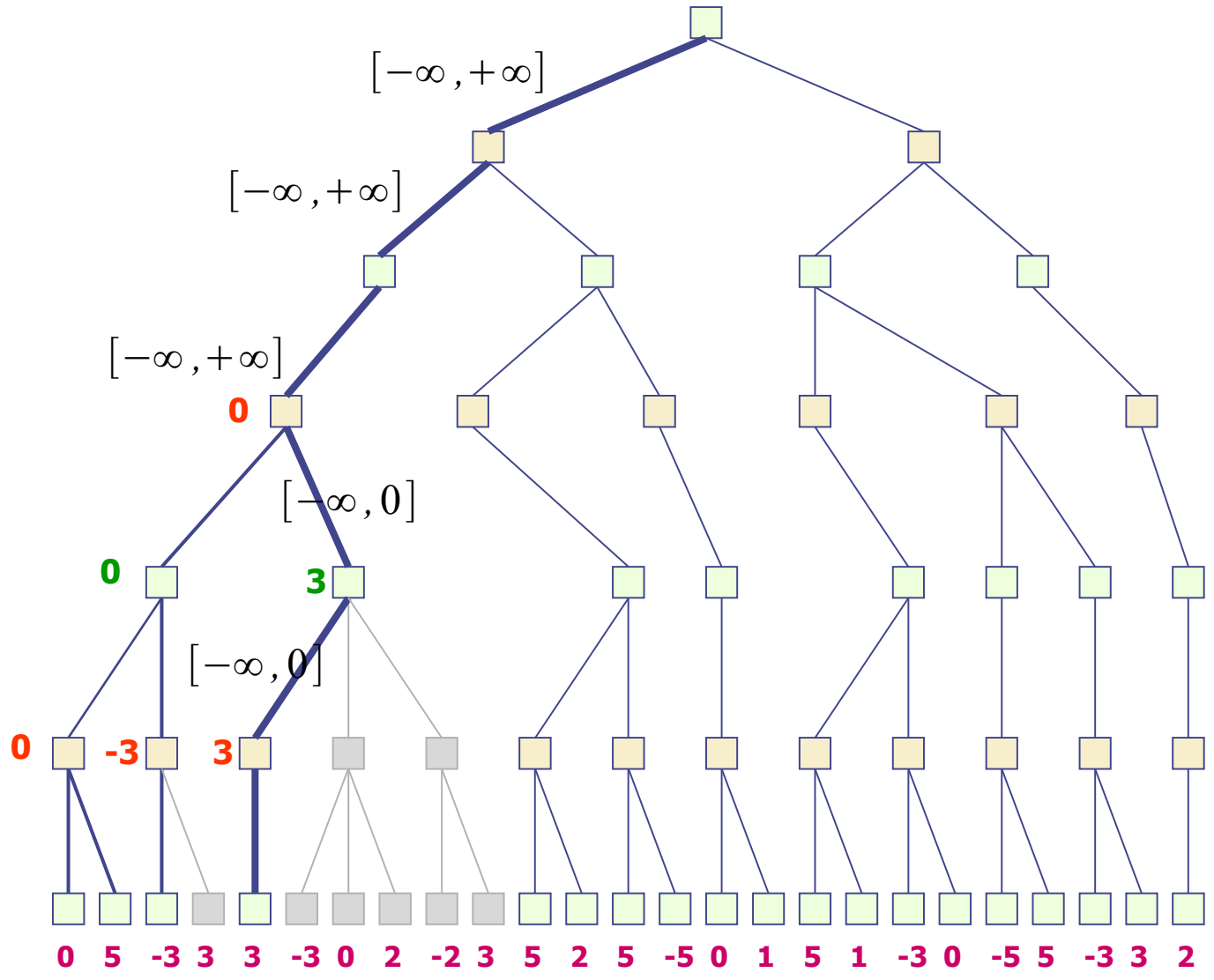


# Alpha-Beta Example

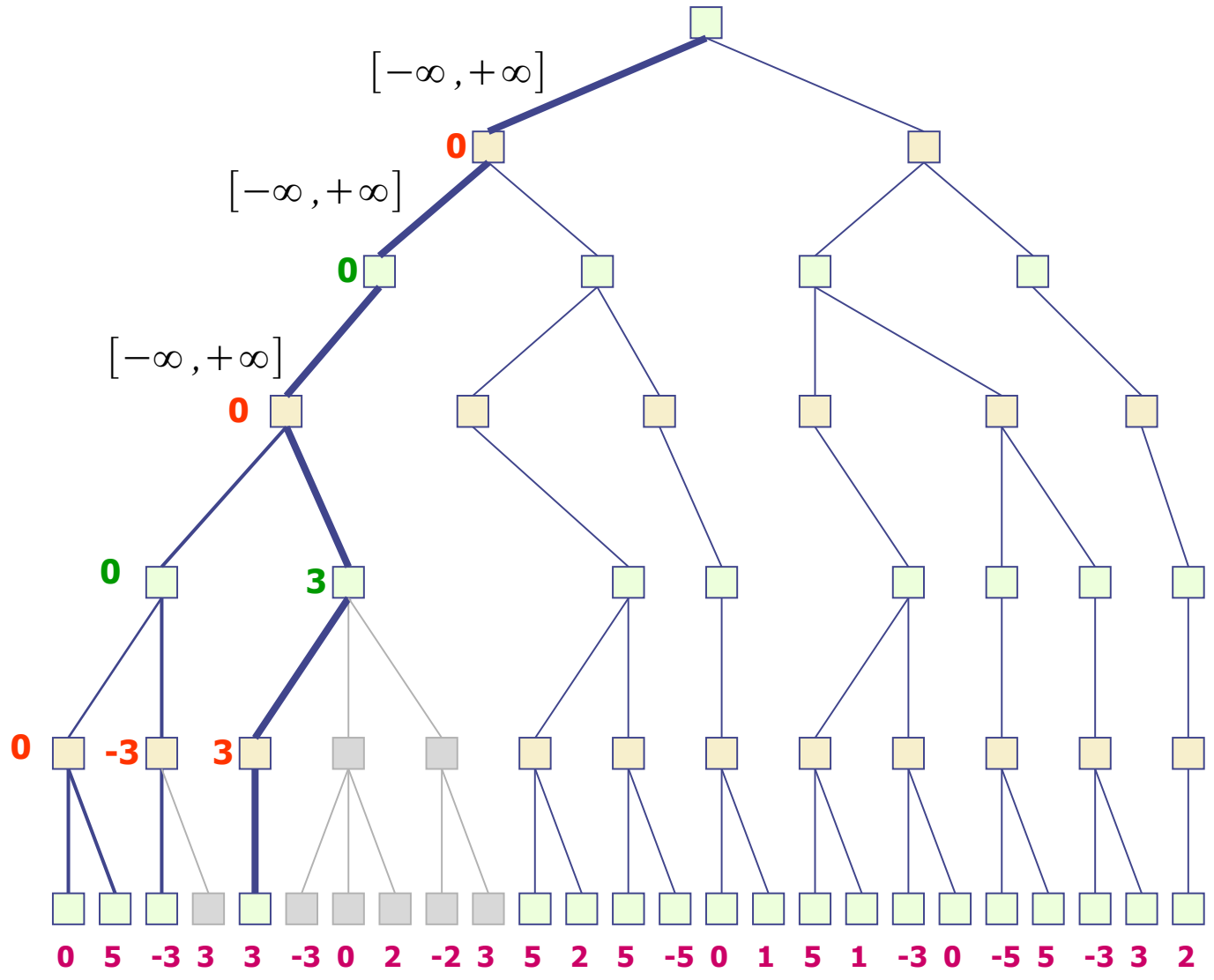




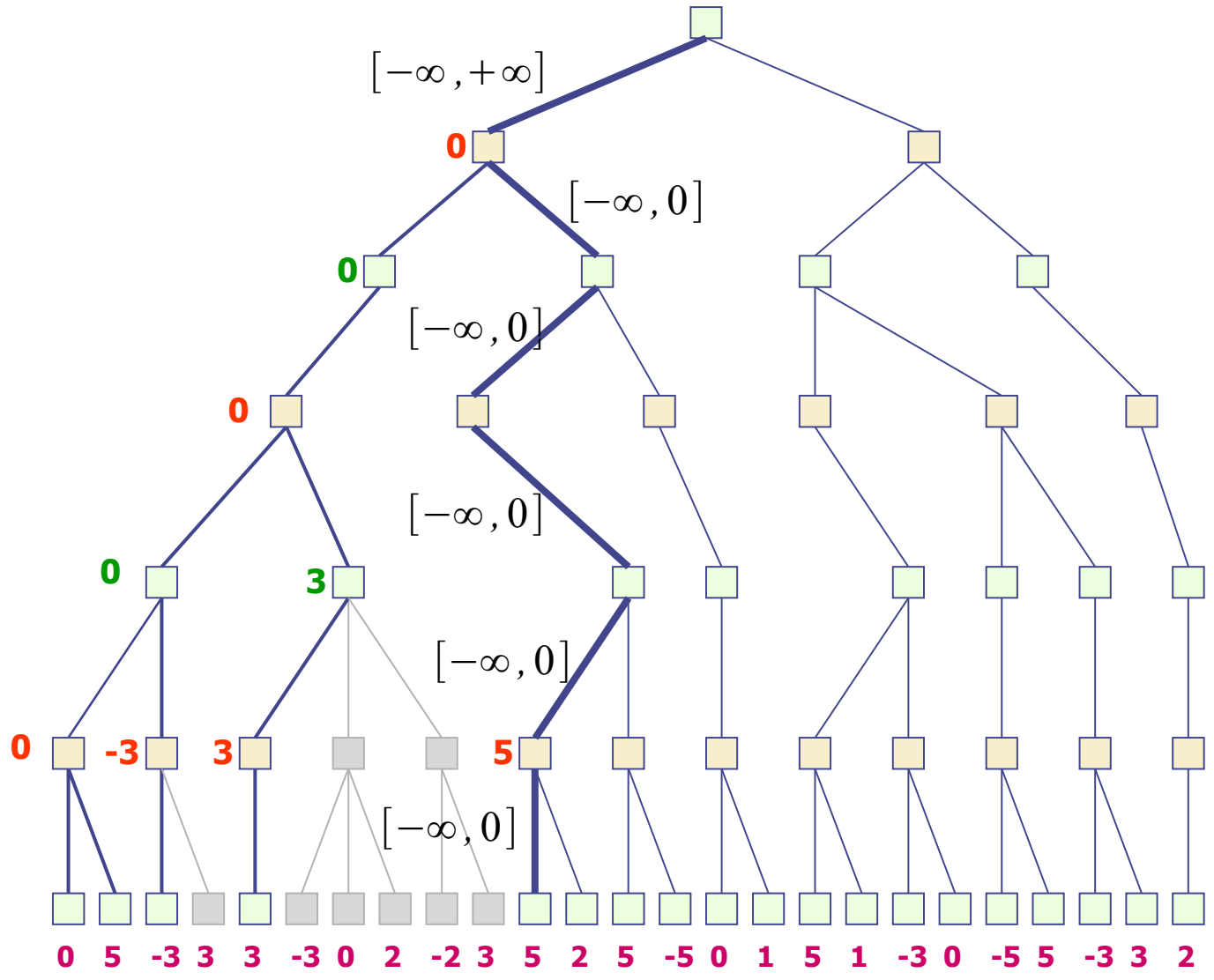
# Alpha-Beta Example



# Alpha-Beta Example



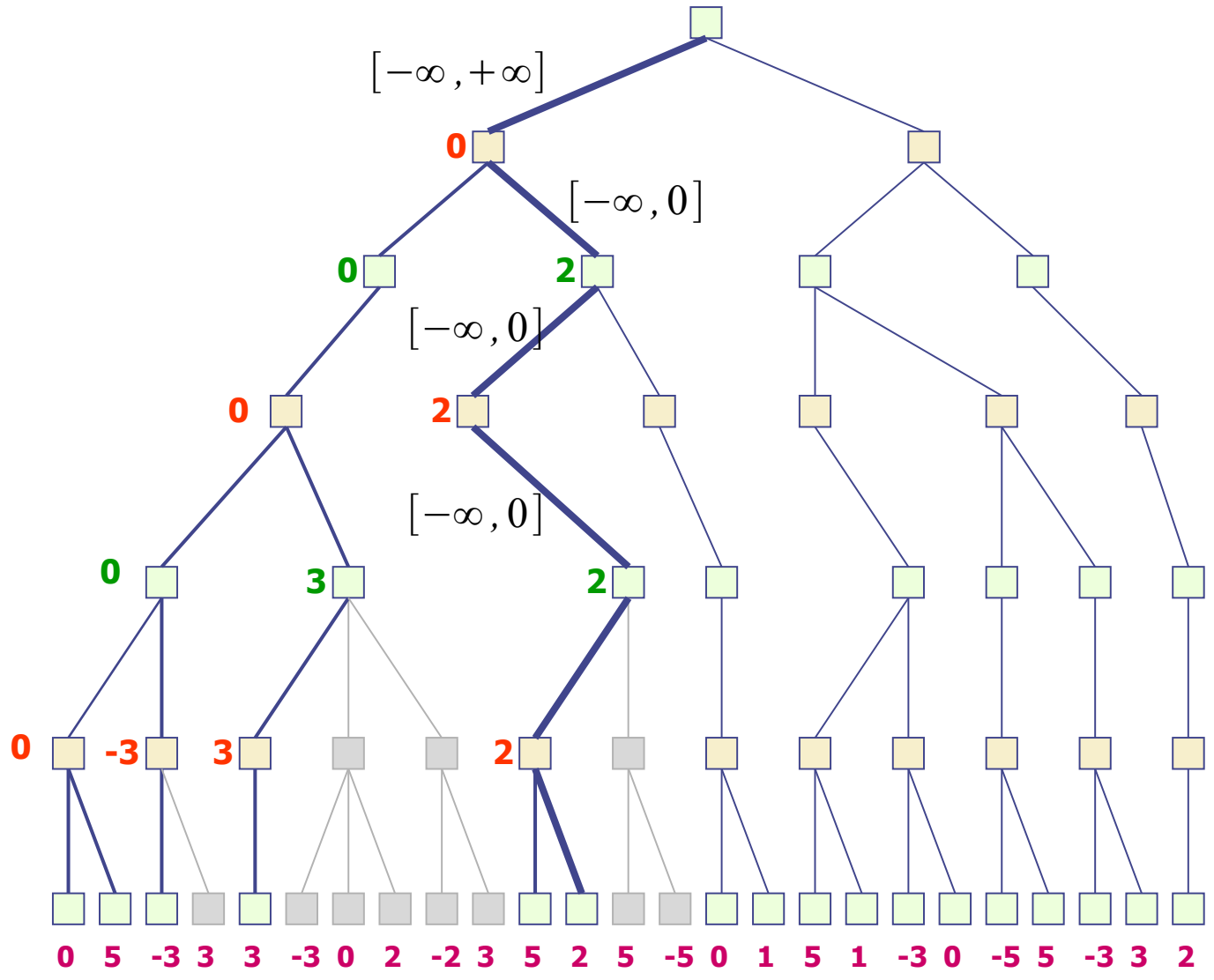
# Alpha-Beta Example



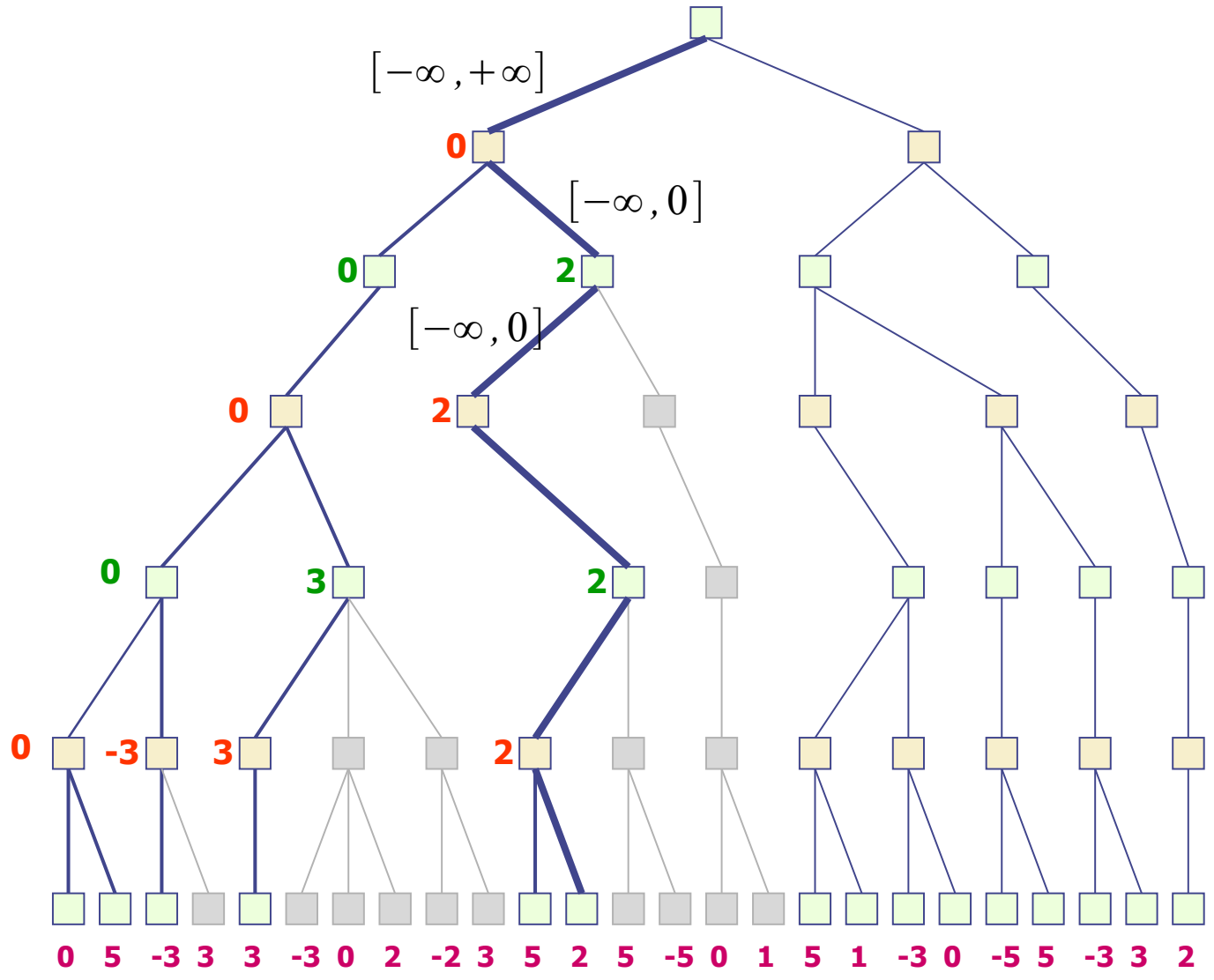




# Alpha-Beta Example



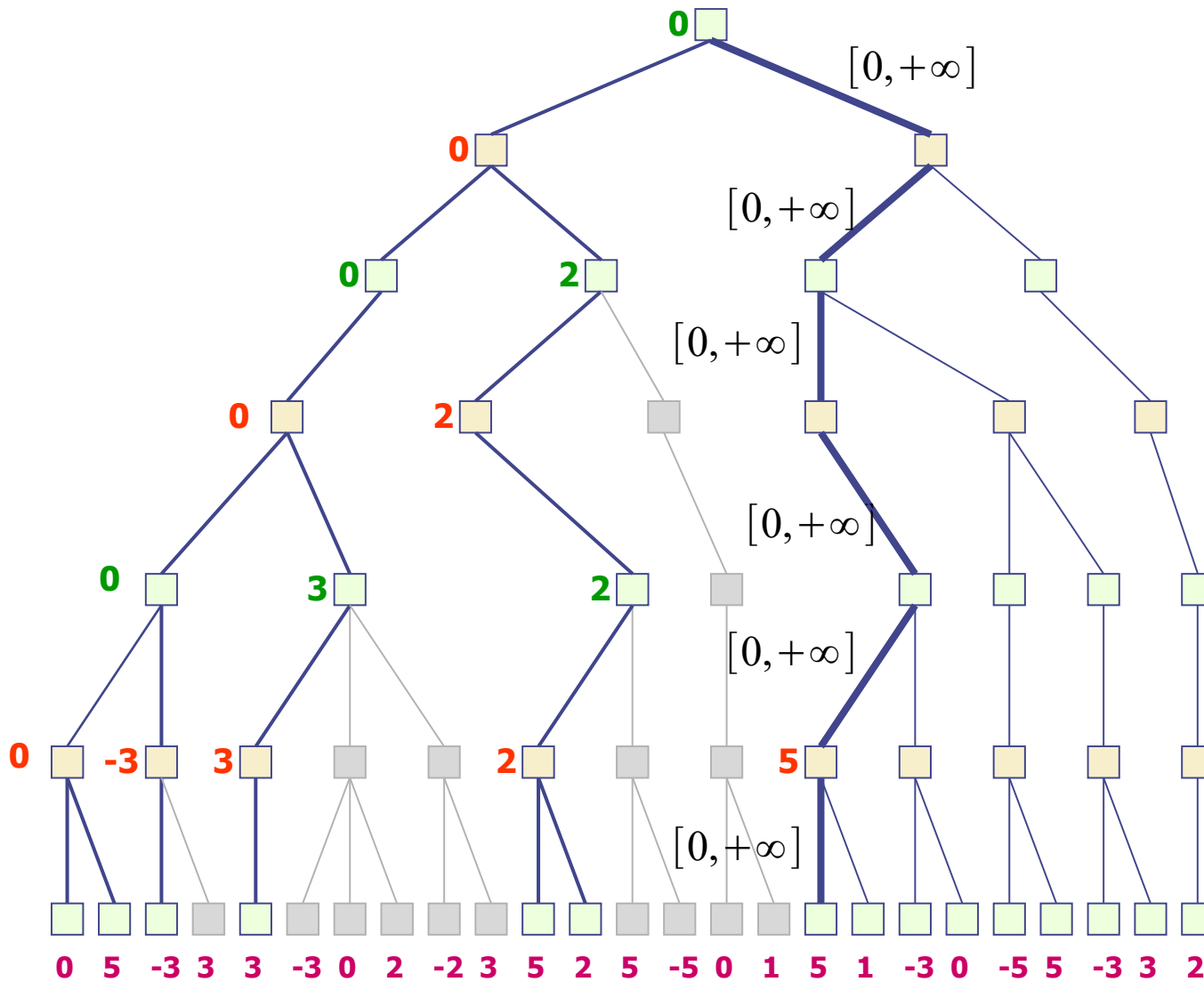
# Alpha-Beta Example



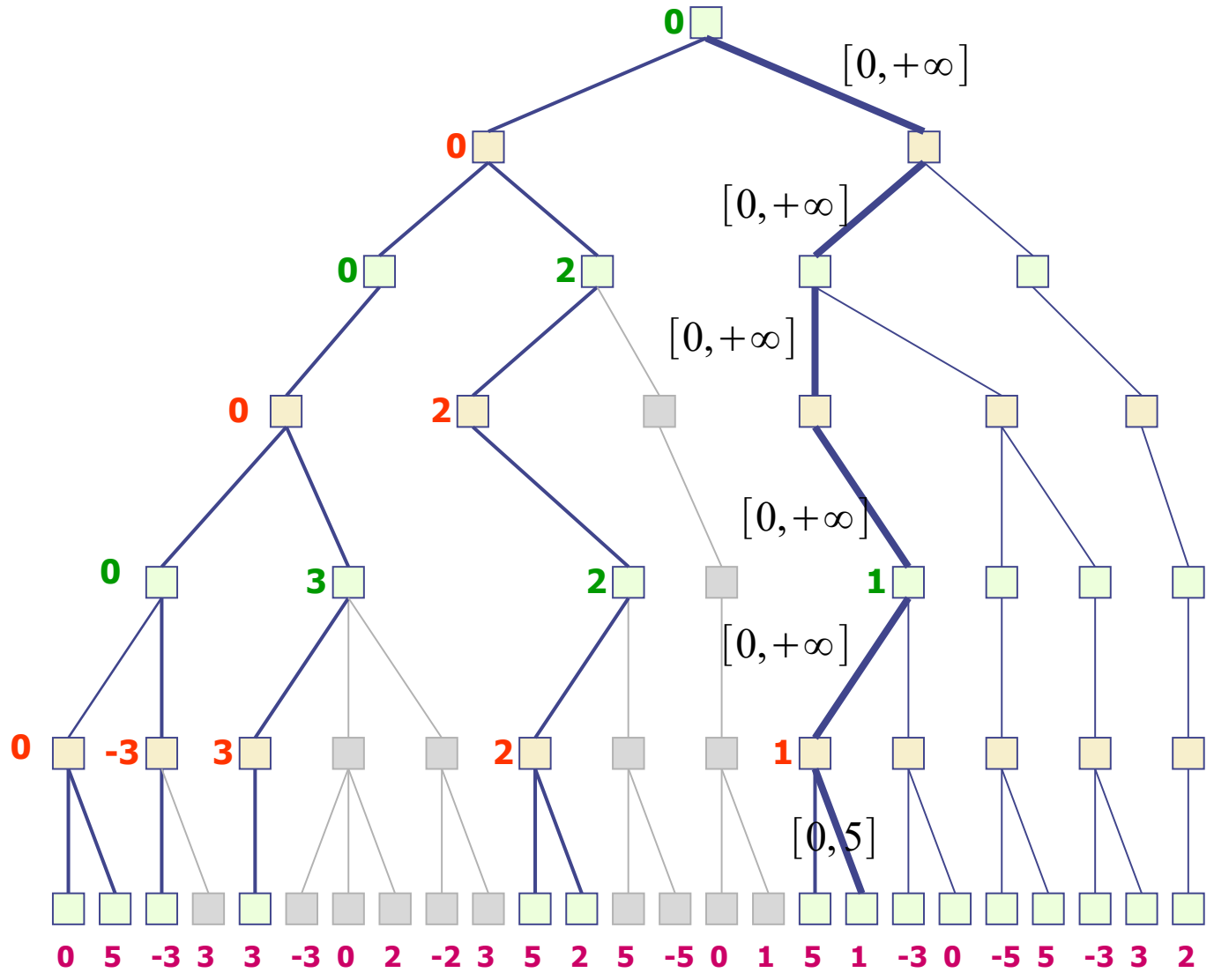




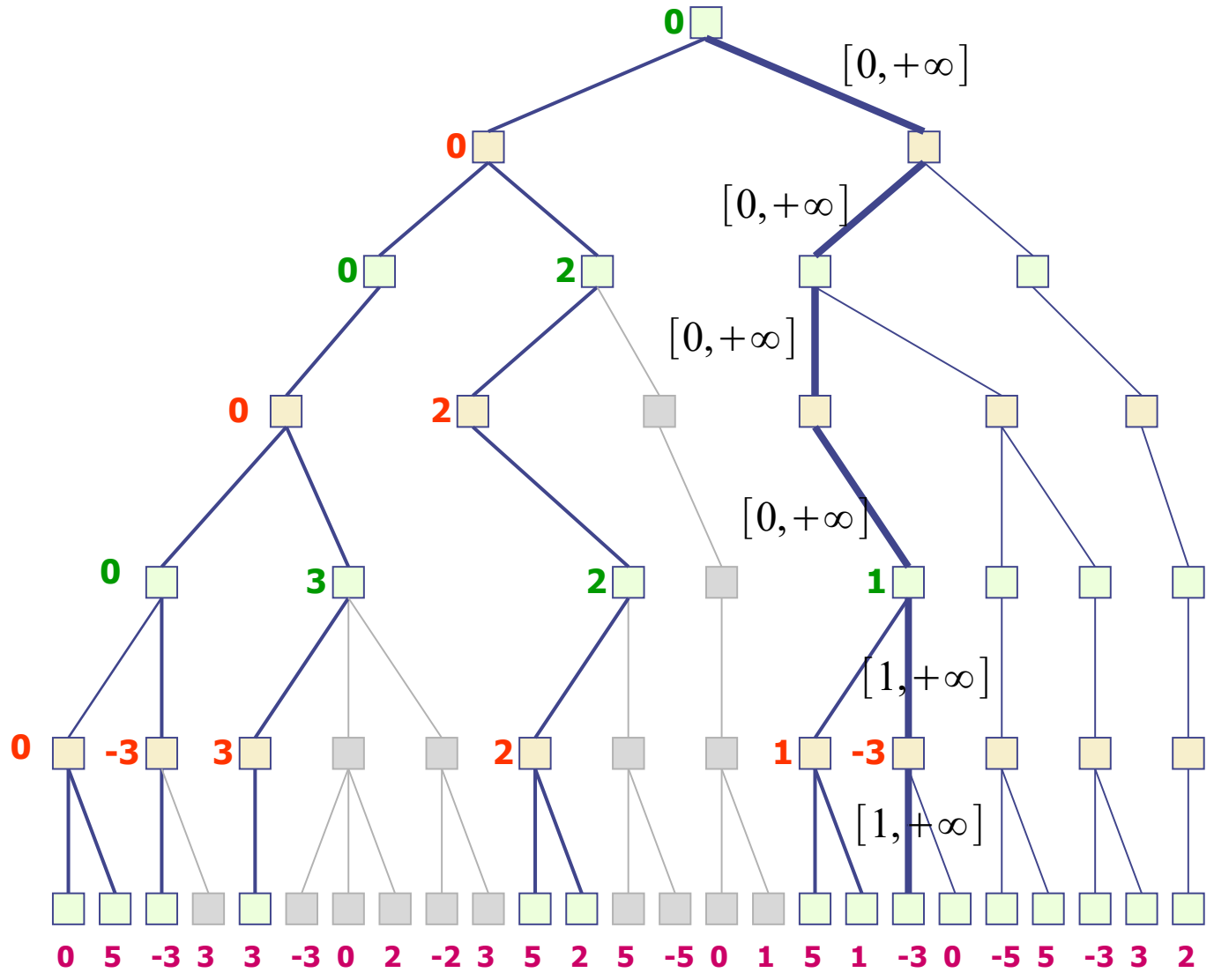
# Alpha-Beta Example



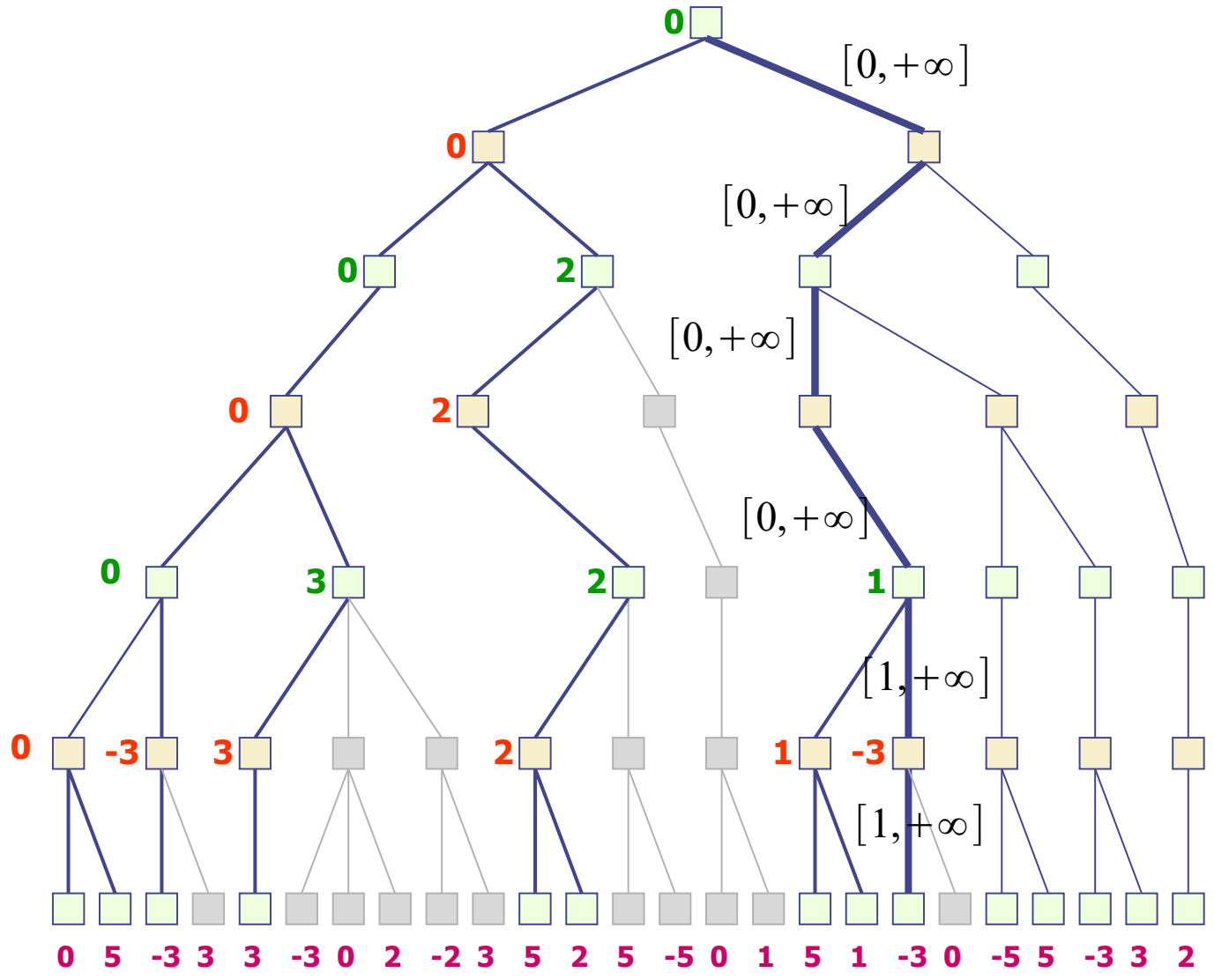
# Alpha-Beta Example



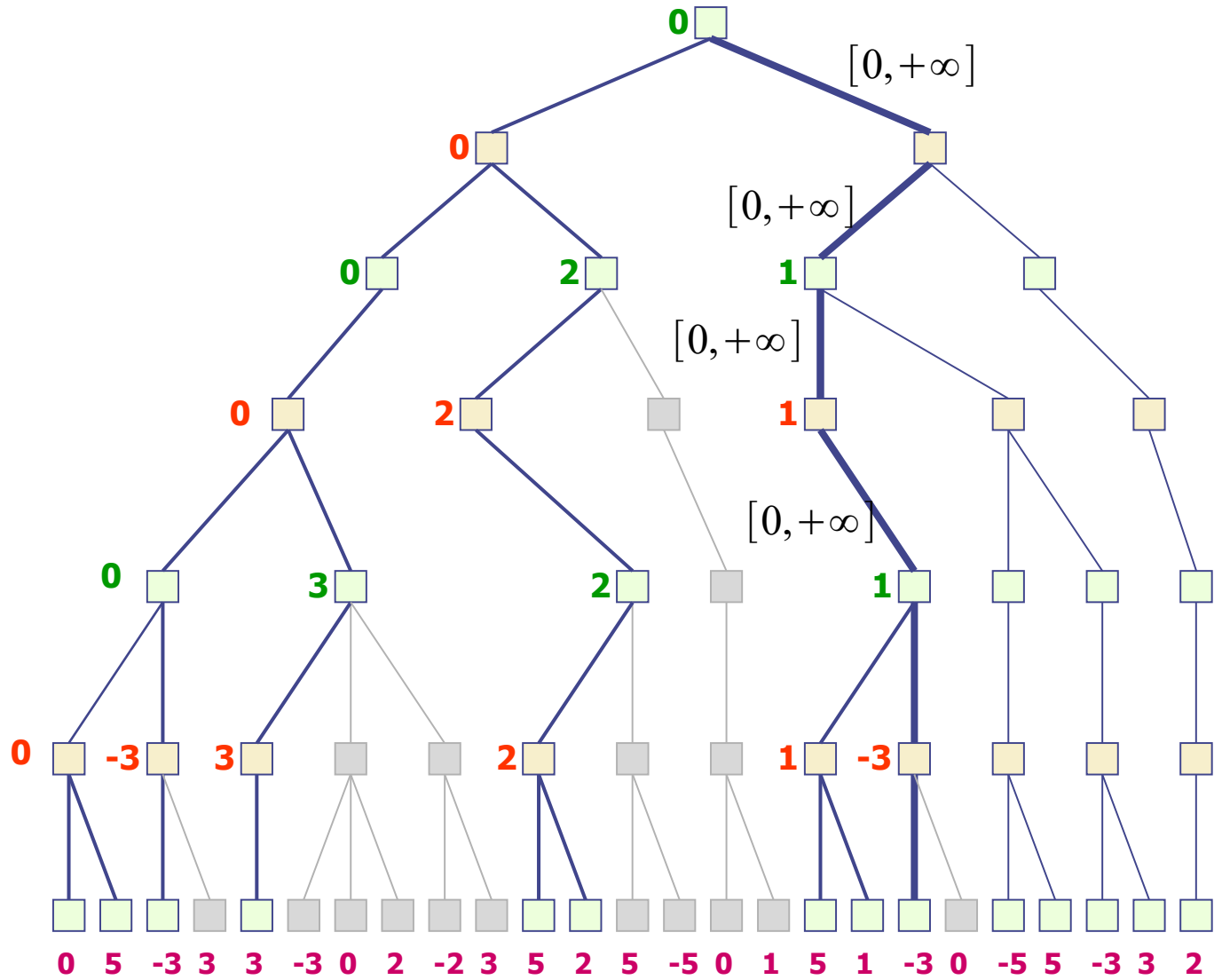
# Alpha-Beta Example



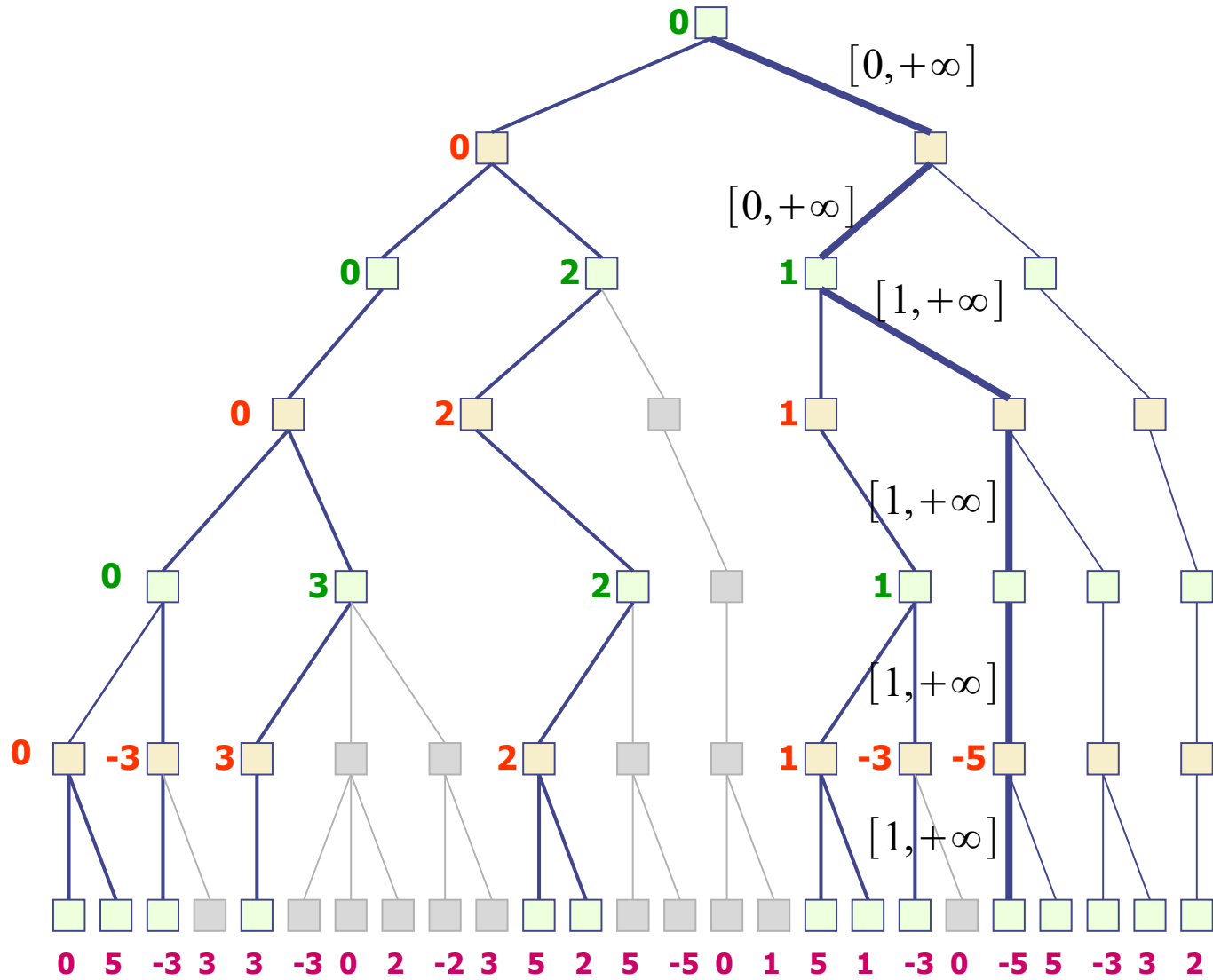
# Alpha-Beta Example



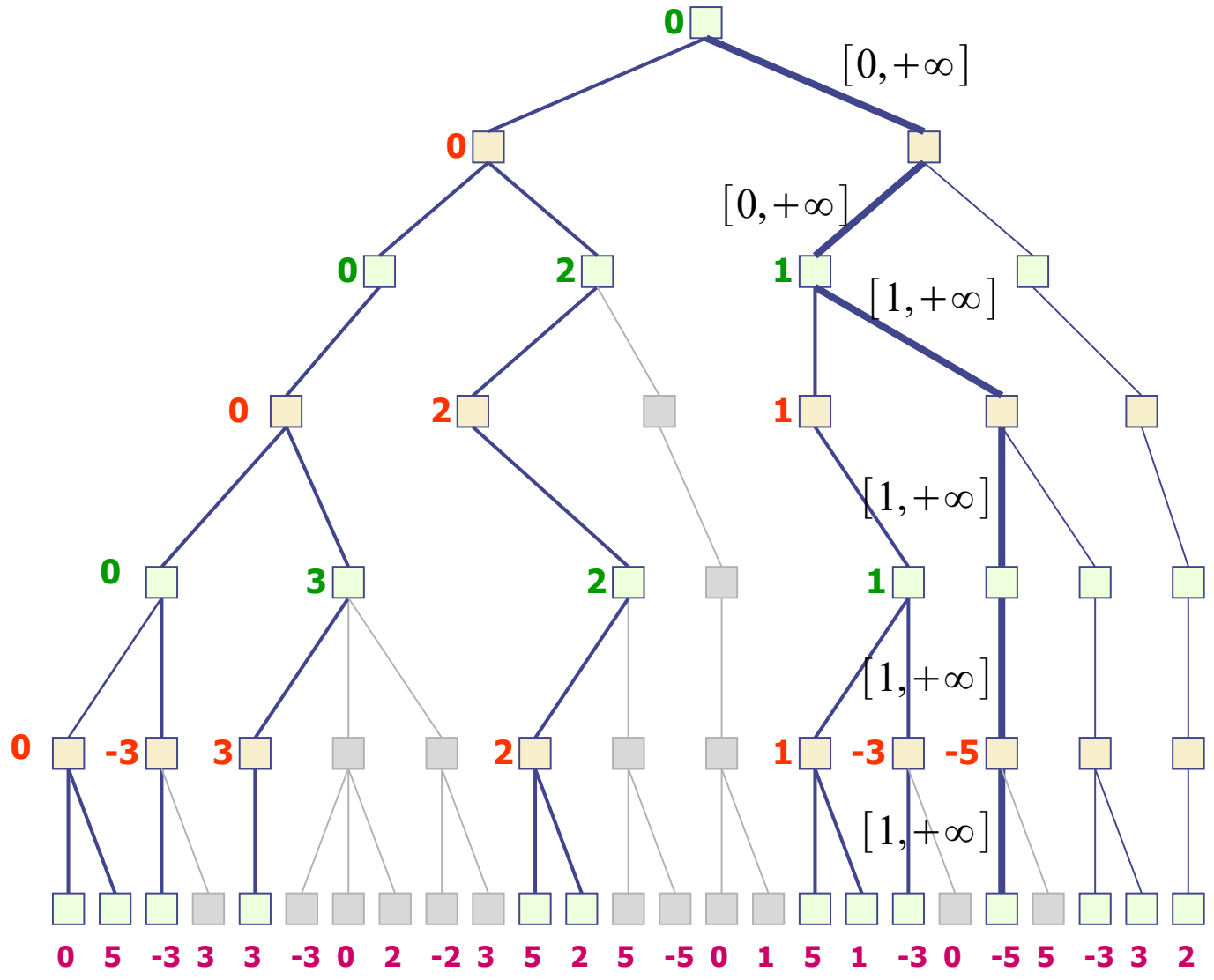
# Alpha-Beta Example



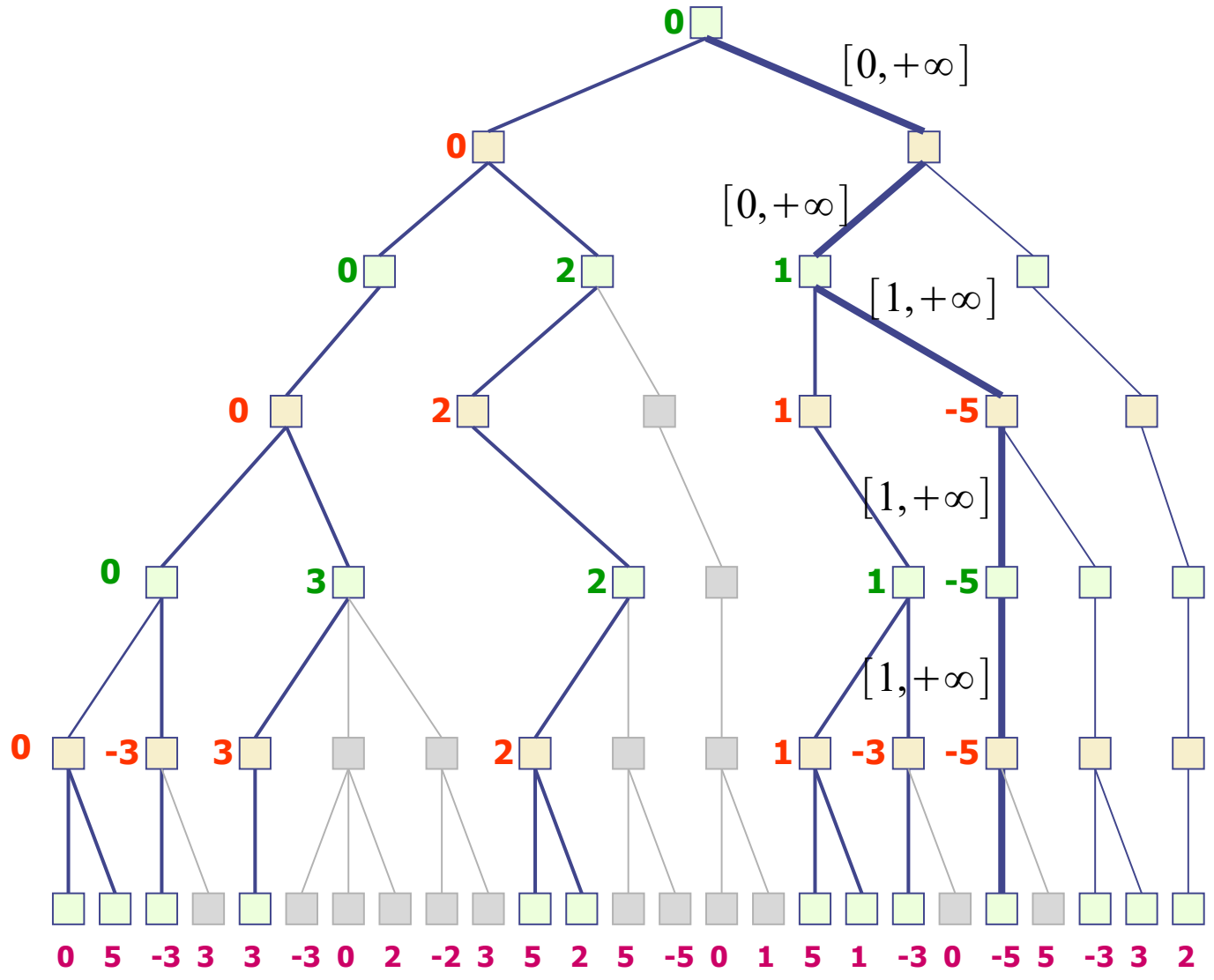
# Alpha-Beta Example



# Alpha-Beta Example

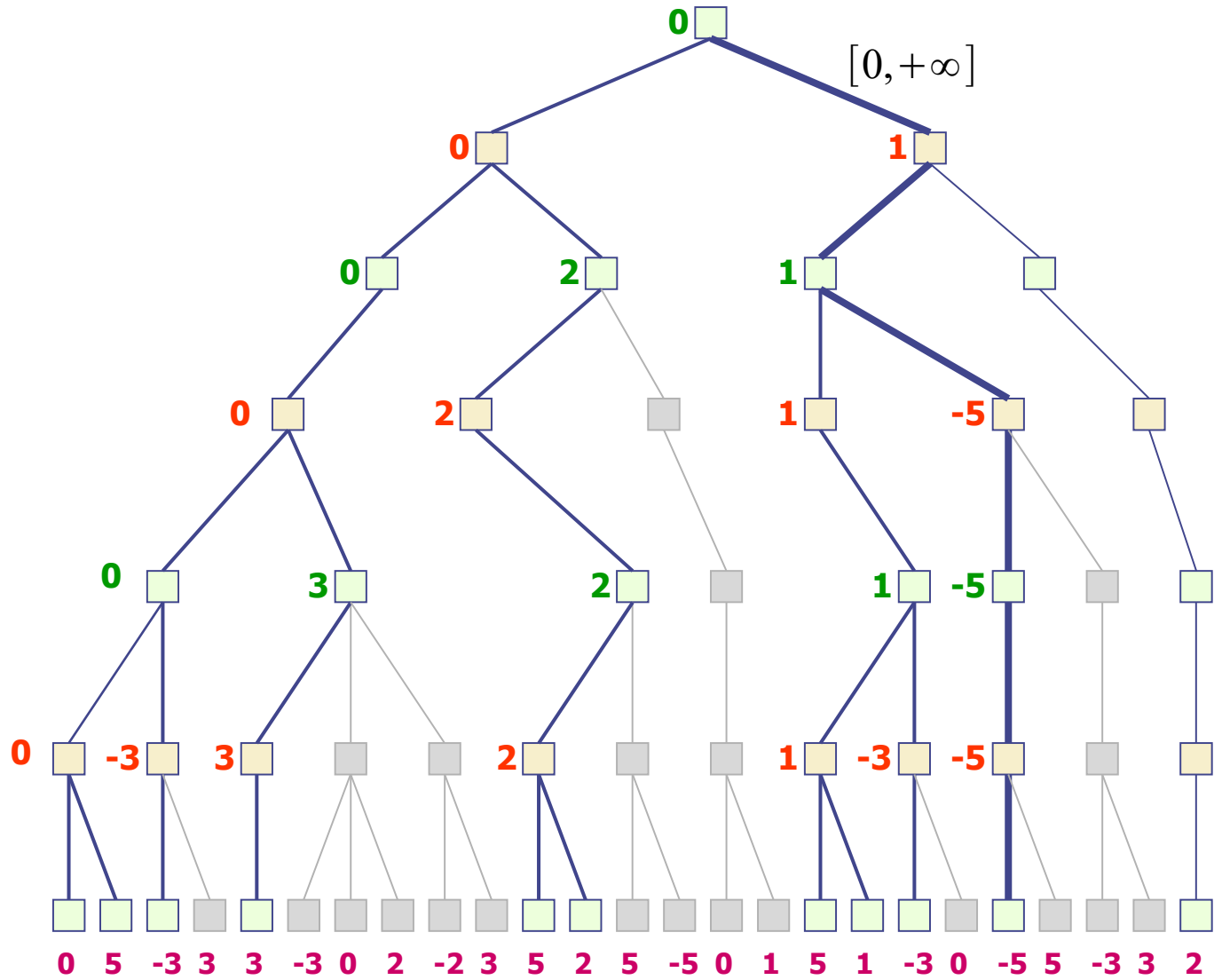


# Alpha-Beta Example

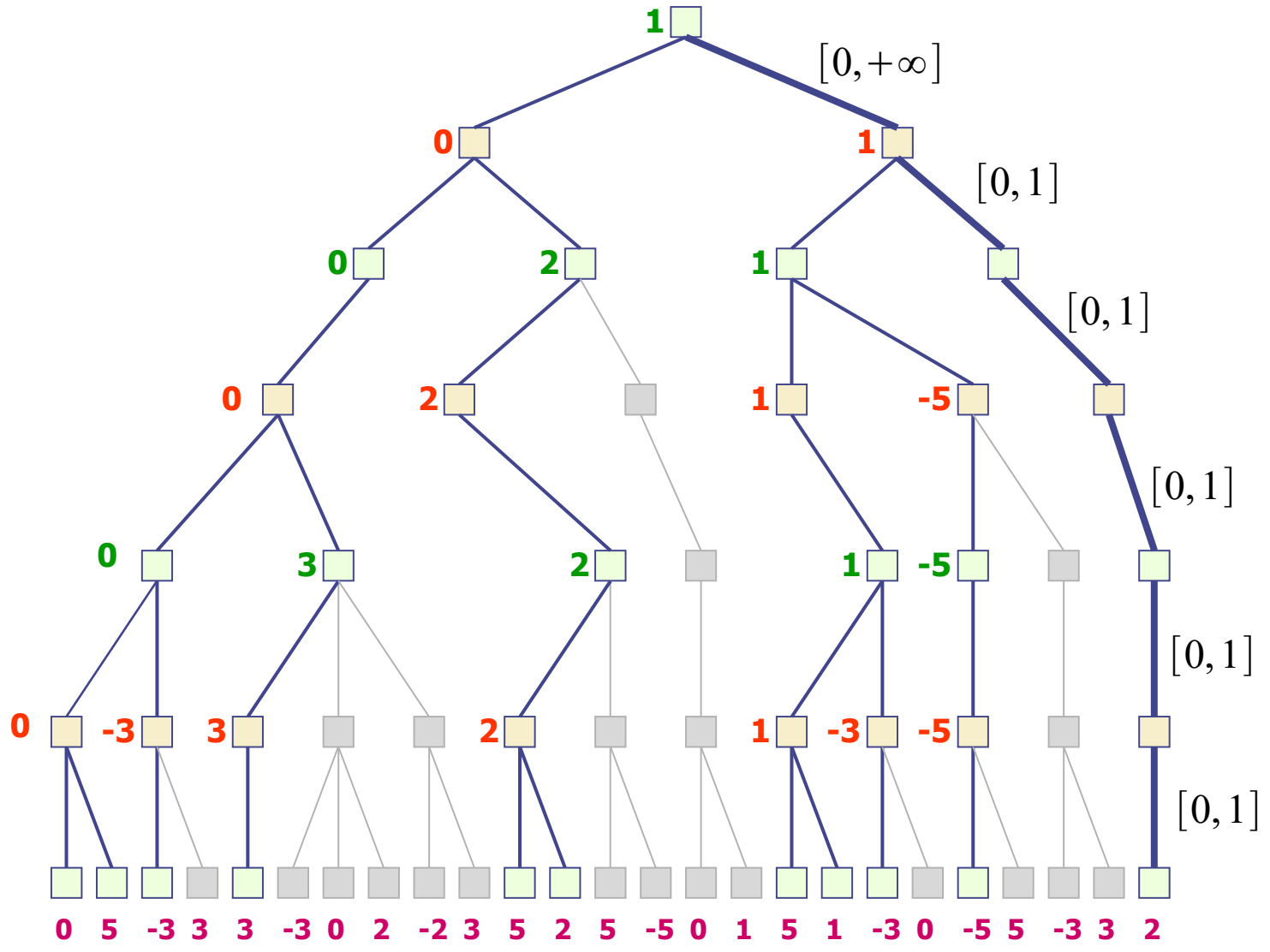




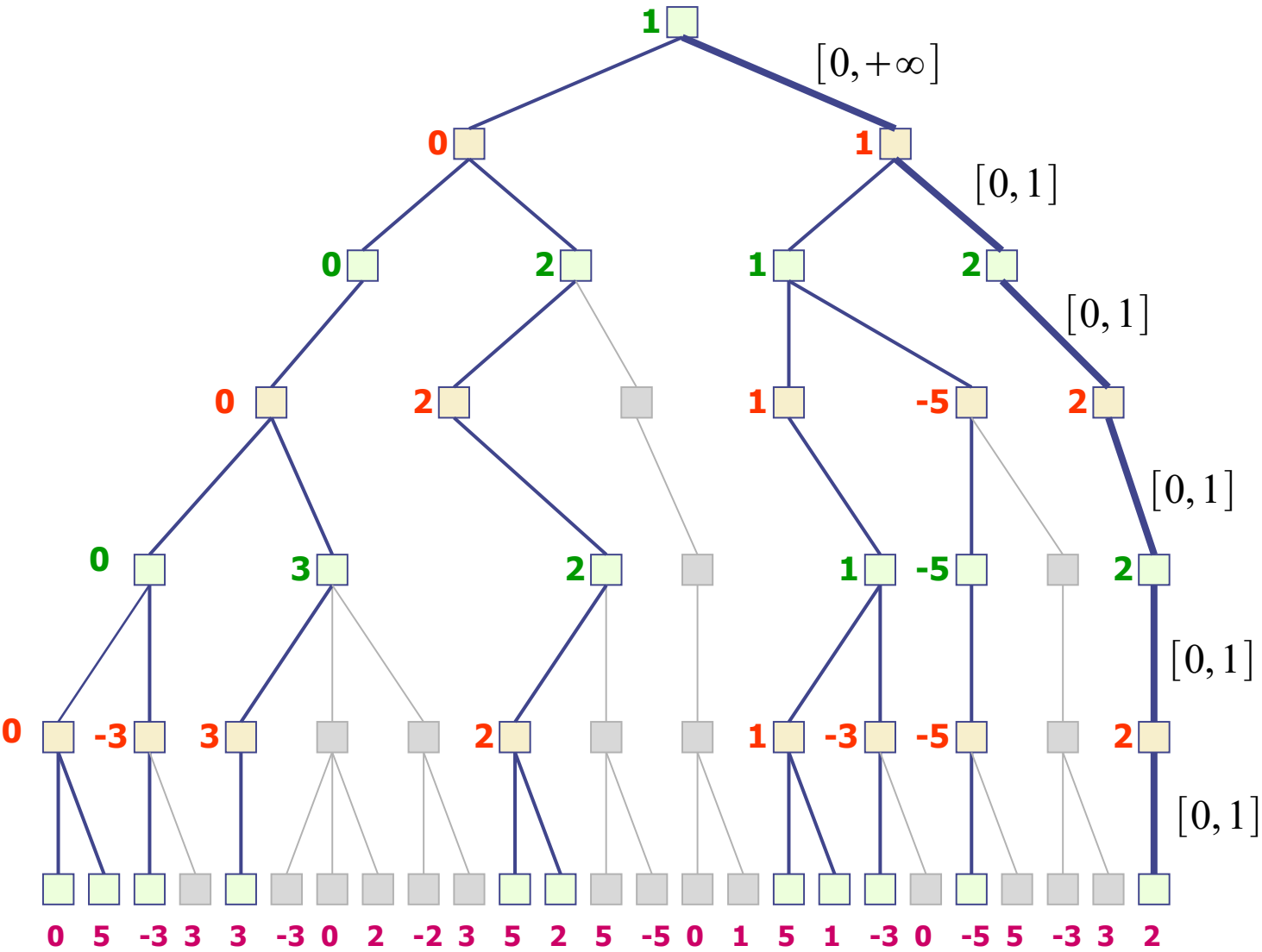
# Alpha-Beta Example



# Alpha-Beta Example

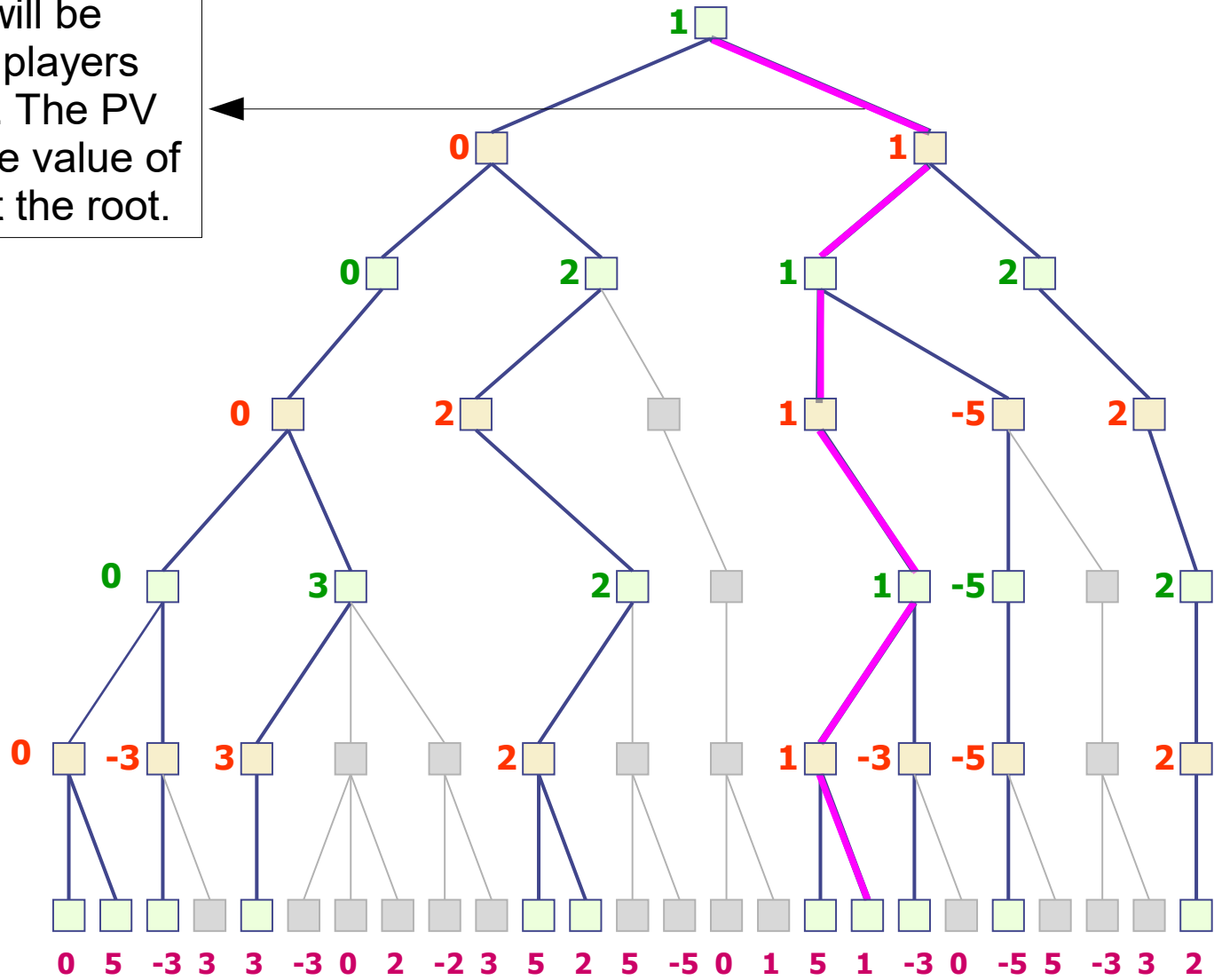


# Alpha-Beta Example



# Alpha-Beta Example

**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

# More Information

- Animated explanations and examples of Alpha-Beta at work (in German)
  - <http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo19.php>

# Alpha-Beta – NegaMax Formulation

```

int AlphaBeta ( position p; int  $\alpha$ ,  $\beta$  );
{
    /* compute minimax value of position p */

    int a, t, i;
    determine successors  $p_1, \dots, p_w$  of p;
    if ( w == 0 )
        return ( Evaluate(p) );                /* leaf node */


    a =  $\alpha$ ;

    for ( i = 1; i <= w; i++ ) {
        t = -AlphaBeta (  $p_i$ , -  $\beta$ , -a );
        a = max( a, t );
        if ( a >=  $\beta$  )
            return ( a );                    /* cut-off */

    }

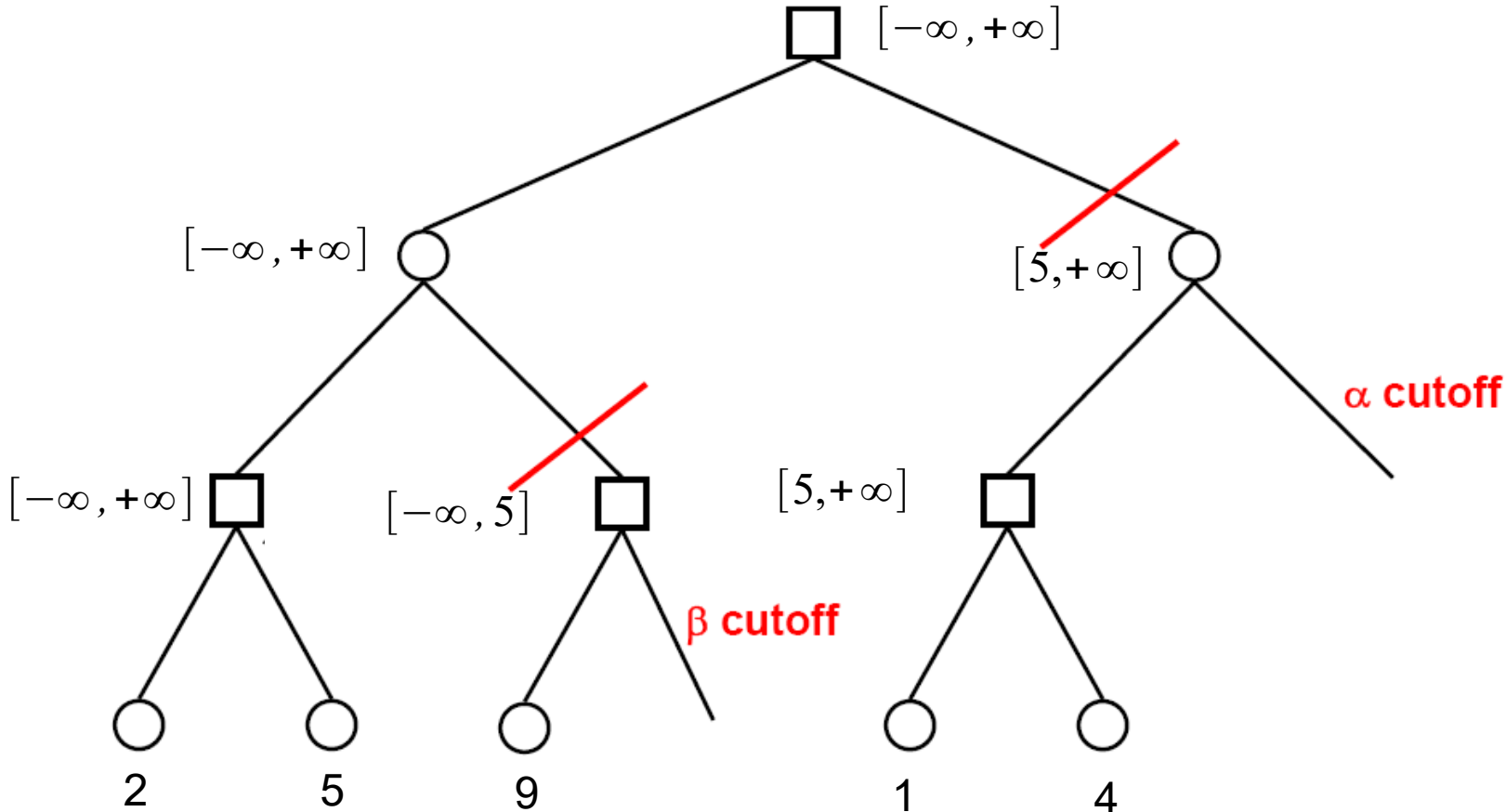
    return ( a );
}

```



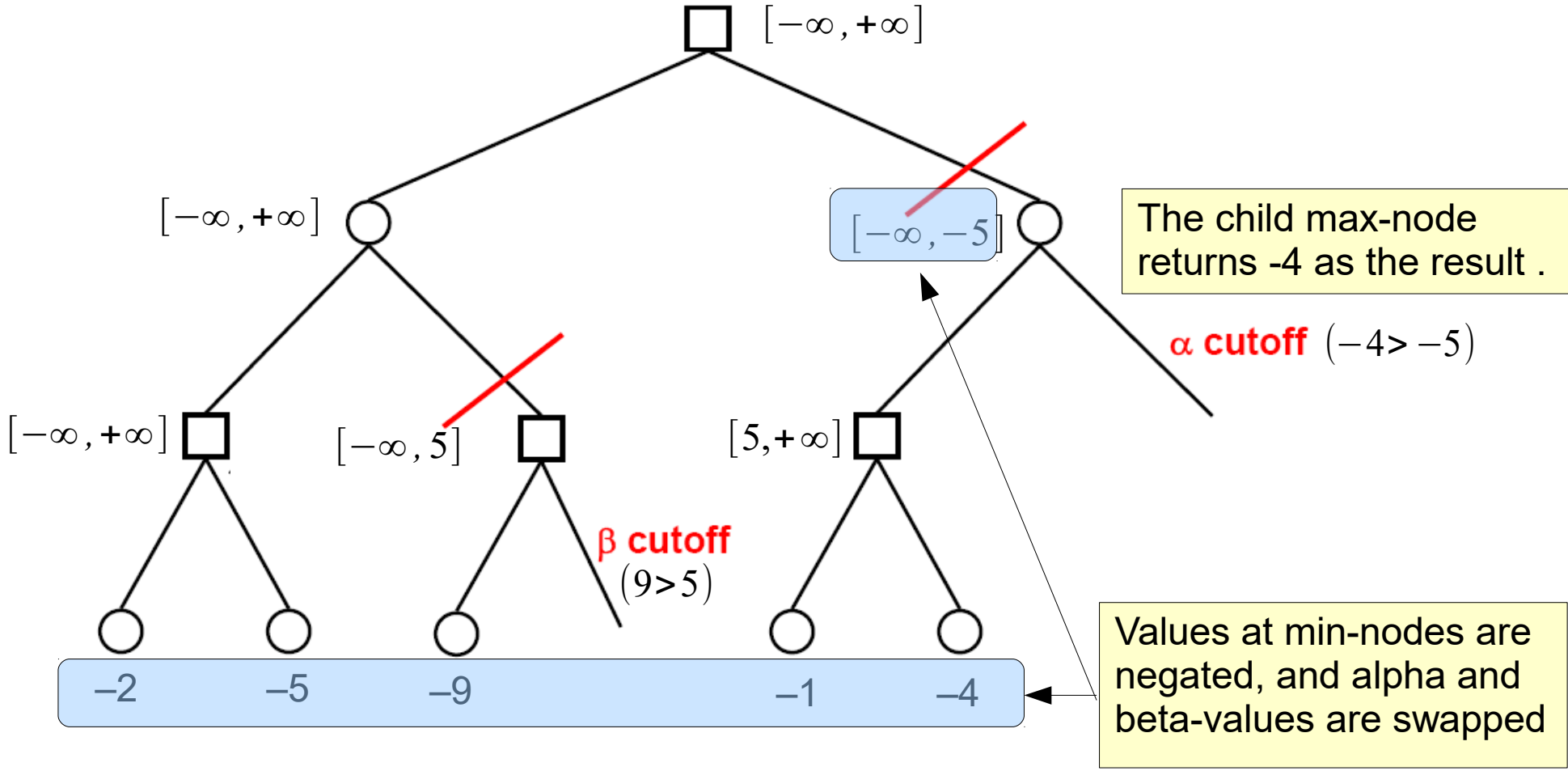
Recursive call with negated window  
 $[\alpha_{MIN} = -\beta_{MAX}, \beta_{MIN} = -\alpha_{MAX}]$   
 Note the negated return value!

# Alpha-Beta (Min-Max Formulation)





# Alpha-Beta (NegaMax Formulation)



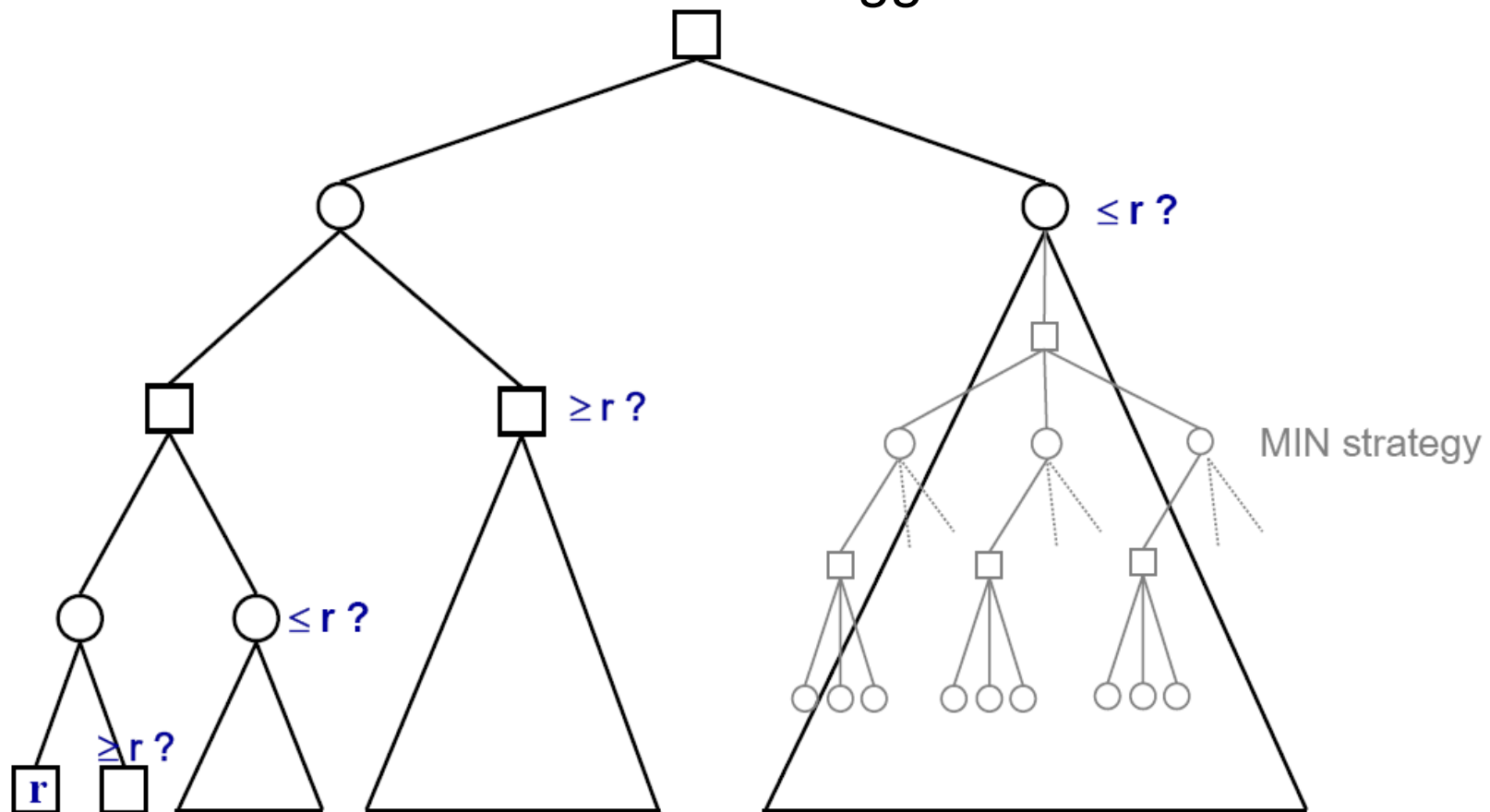
# Minimal Window Search

- 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**), the node is not interesting (a better node exists)
- If **FAIL-HIGH**, we know that this is better, but not how much
  - need to re-search the tree with a bigger window



# NegaScout (Reinefeld 1982)

```

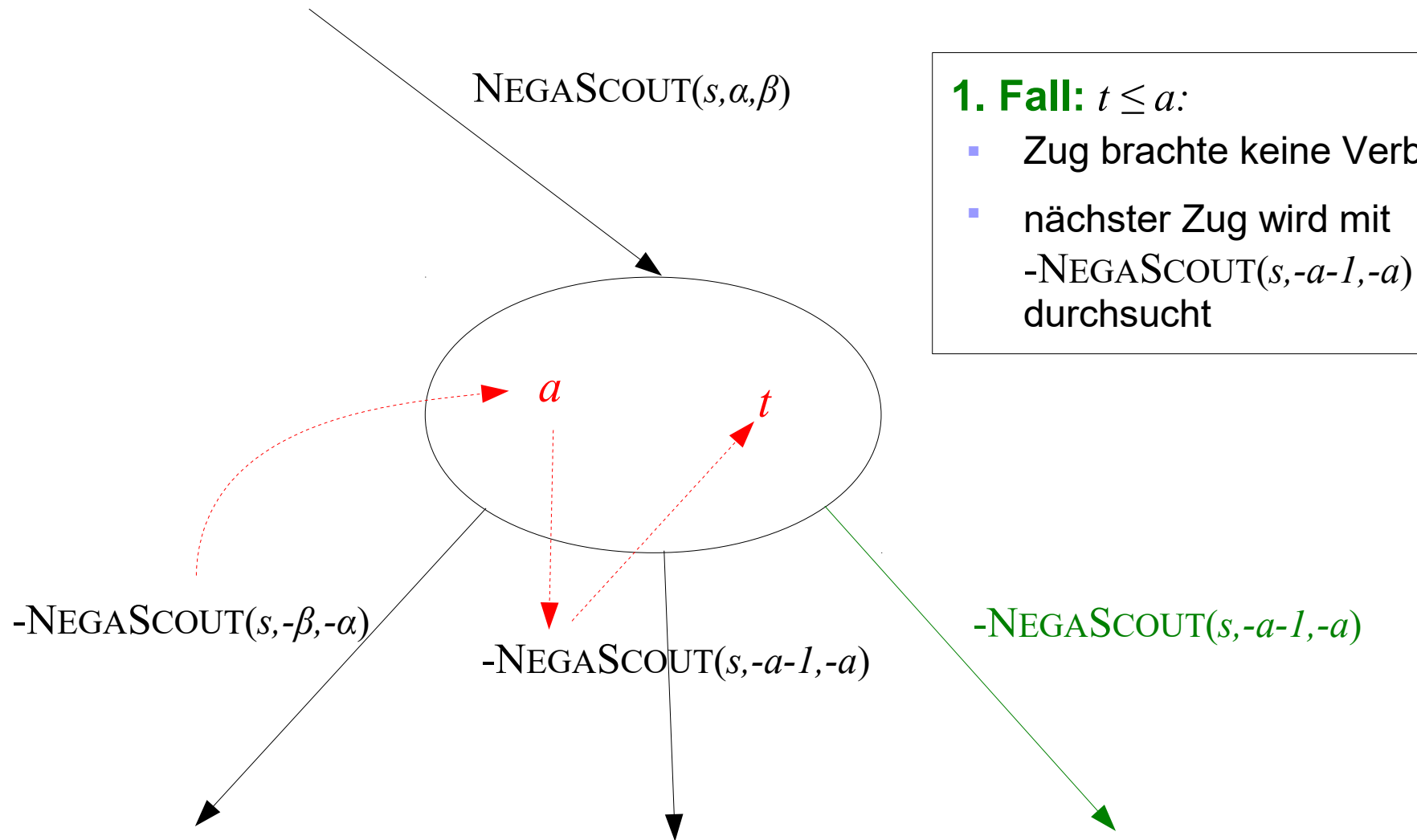
int NegaScout ( position p; int  $\alpha$ ,  $\beta$  );
{
    /* compute minimax value of position p */
    int a, b, t, i;
    determine successors  $p_1, \dots, p_w$  of p;
    if ( w == 0 )
        return ( Evaluate(p) );           /* leaf node */
    a =  $\alpha$ ;
    b =  $\beta$ ;
    for ( i = 1; i <= w; i++ ) {
        t = -NegaScout (  $p_i$ , -b, -a );
        if ( t > a && t <  $\beta$  && i > 1 && d < maxdepth-1 )
            a = -NegaScout (  $p_i$ , - $\beta$ , -t );    /* re-search */
        a = max( a, t );
        if ( a >=  $\beta$  )
            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)

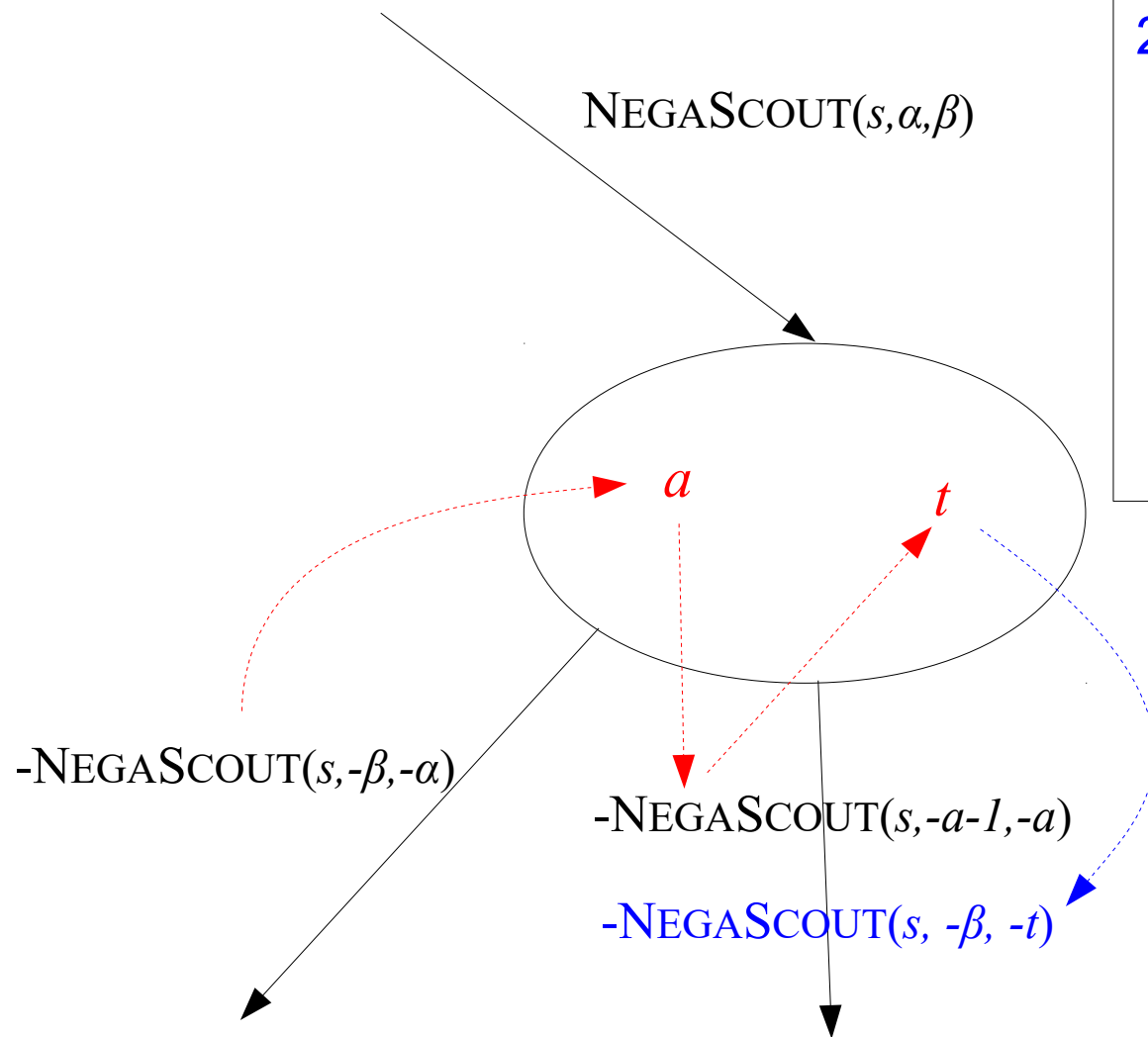
# Schematische Darstellung des Ablaufs in einem MAX-Knoten



## 1. Fall: $t \leq a$ :

- Zug brachte keine Verbesserung
- nächster Zug wird mit  $-\text{NEGASCOUT}(s, -a-1, -a)$  durchsucht

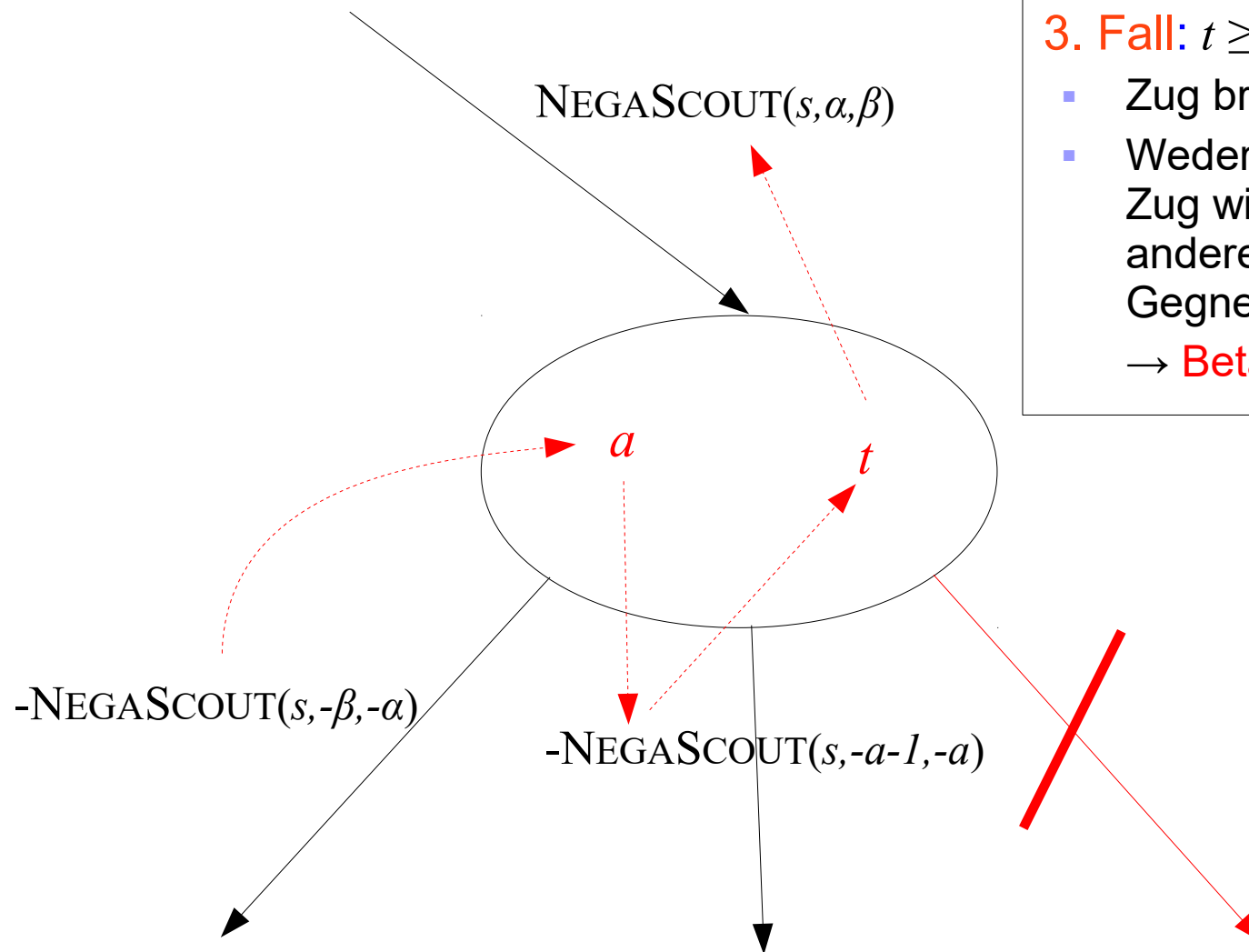
# Schematische Darstellung des Ablaufs in einem MAX-Knoten



## 2. Fall: $a < t < \beta$ :

- Zug bringt zumindest  $t$
- Genauer Wert läßt sich aber nicht bestimmen, da Zweige aufgrund des falschen  $\beta$ -Werts geprunt worden sein könnten
- 2. Zug muß nochmals durchsucht werden mit  $-\text{NEGASCOUT}(s, -\beta, -t)$

# Schematische Darstellung des Ablaufs in einem MAX-Knoten

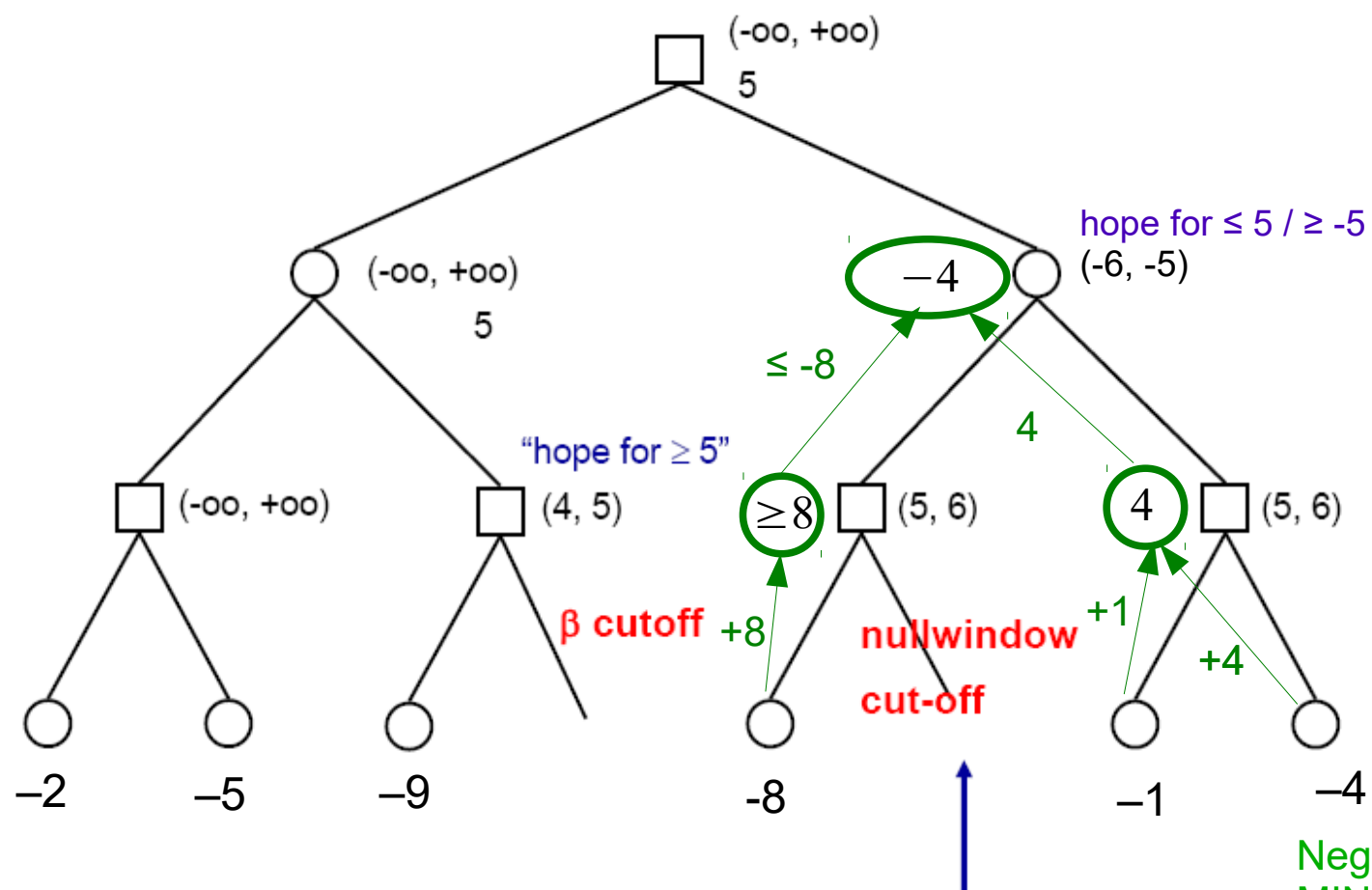


### 3. Fall: $t \geq \beta$ :

- Zug bringt mehr als  $\beta$
- Weder dieser noch ein anderer Zug wird gespielt werden, da ein anderer Pfad im Baum dem Gegner mehr verspricht.

→ **Beta-Cutoff**

# NegaScout Example



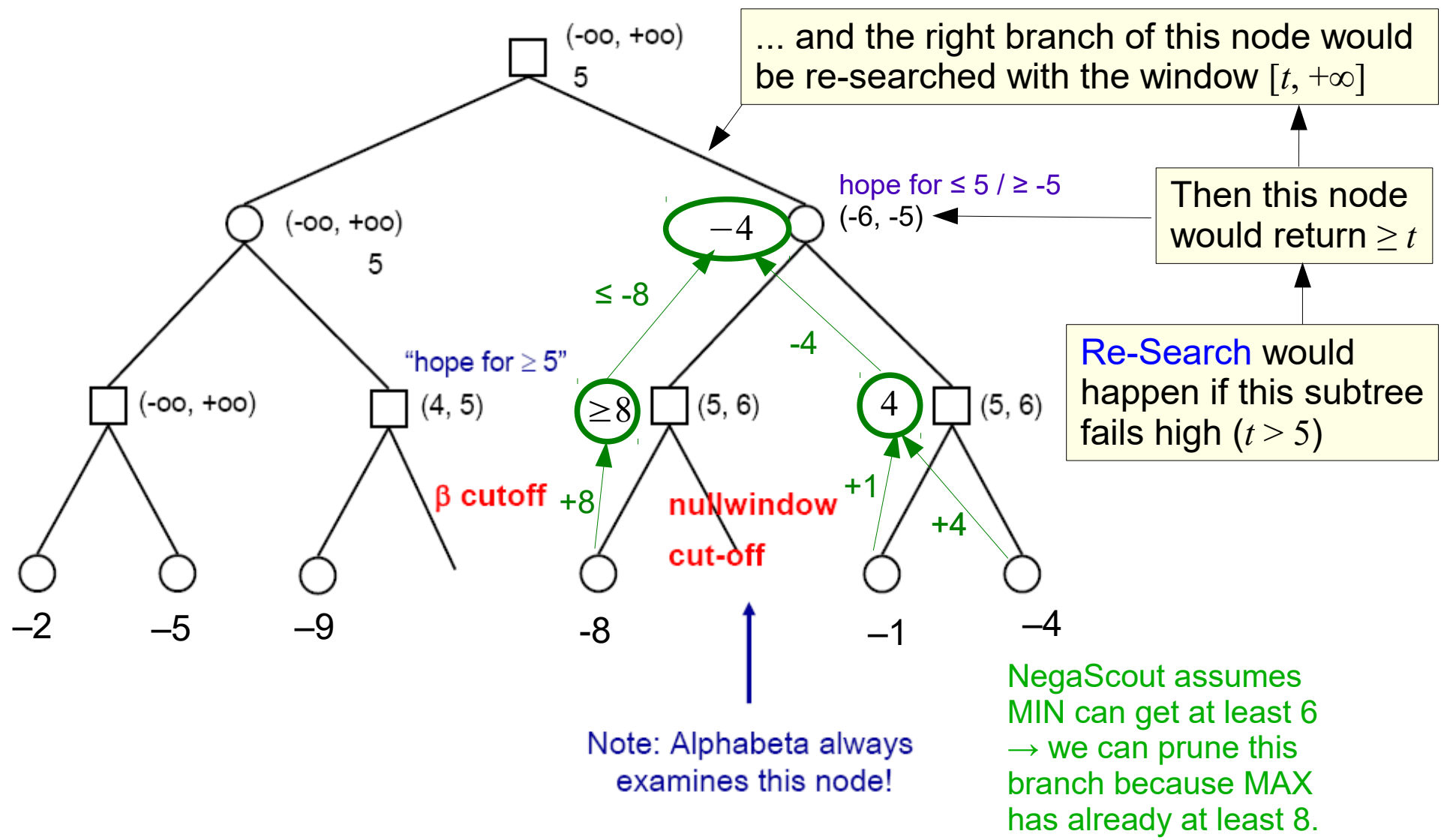
The assumption has turned out to be correct ( $4 \leq 5$ ), so the null Window cut-off was justified.

Note: Alphabeta always examines this node!

NegaScout assumes MIN can get at least 6 → we can prune this branch because MAX has already at least 8.



# 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

# 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  $2^d$ )

# Outline

- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - $\alpha$ - $\beta$  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

# 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:  $\sim 10^{40}$  nodes
  - Chess:  $\sim 10^{120}$  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.
  - $f(n) \gg 0$ : position  $n$  good for me and bad for you
  - $f(n) \ll 0$ : 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

- $$\text{EVAL}(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \dots + w_n \cdot f_n(s) = \sum_{i=1}^n w_i f_i(s)$$

- 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 ( $\rightarrow$  machine learning)
  - adding up the weighted features makes the assumption that each feature is independent of the other features

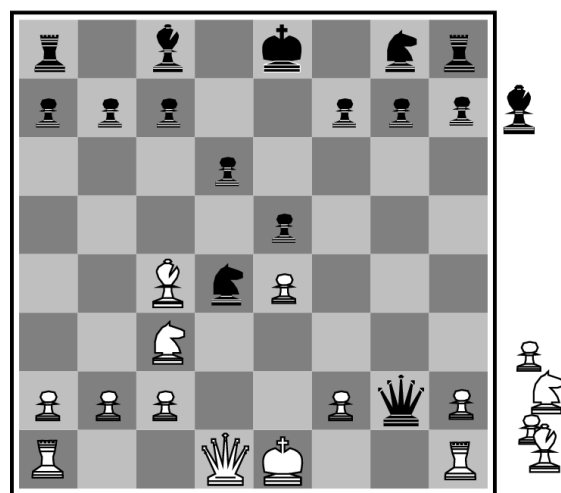
# Evaluation Function Examples

- Example of an evaluation function for Tic-Tac-Toe:
  - $f(n) = [\# \text{ 3-lengths open for me}] - [\# \text{ 3-lengths open for you}]$
  - 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

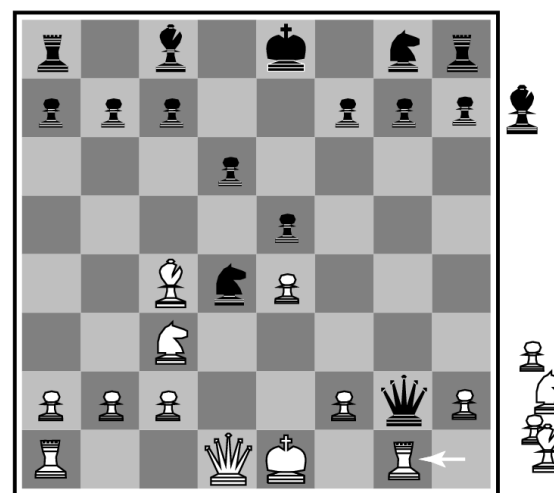
Evaluation is typically very brittle

- 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!

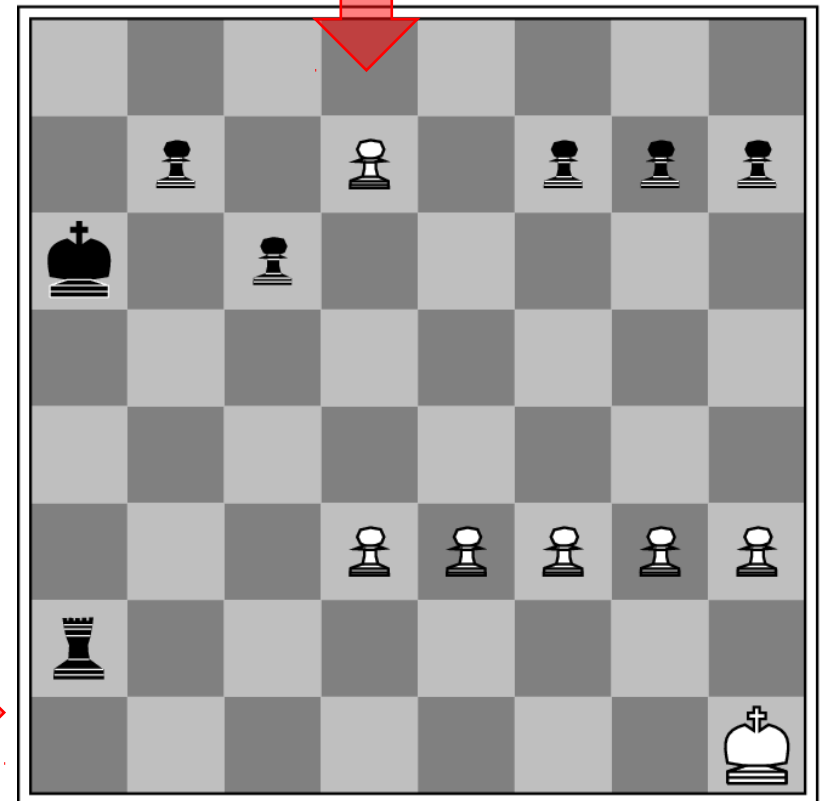
# Horizon Effect

- Problem with fixed-depth search:
  - if we only search  $n$  moves ahead, it may be possible that the catastrophe 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:

Black can give many consecutive checks before white escapes



Fixed depth search thinks it can avoid the queening move



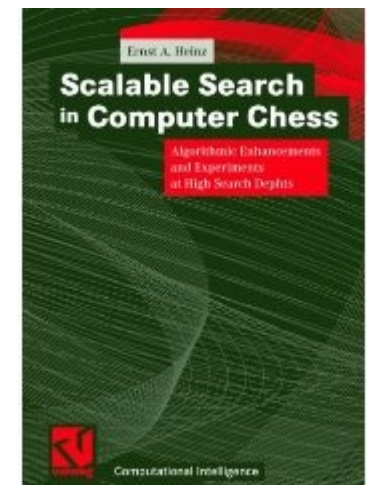
Black to move

# 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 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

# Iterative Deepening

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

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

Advantages:

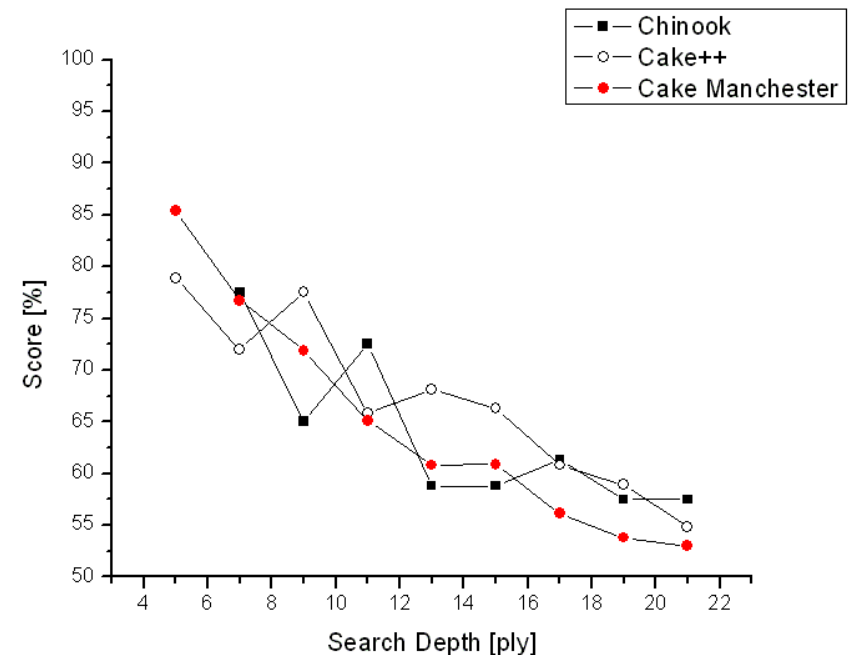
- works well with transposition tables
- improved dynamic move-ordering in alpha-beta
  - what worked well in the previous iteration is tried first in the next iteration
- simplifies time managements
  - if there is a fixed time limit per move, this can be handled flexibly by adjusting the number of 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 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 programs 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 an 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

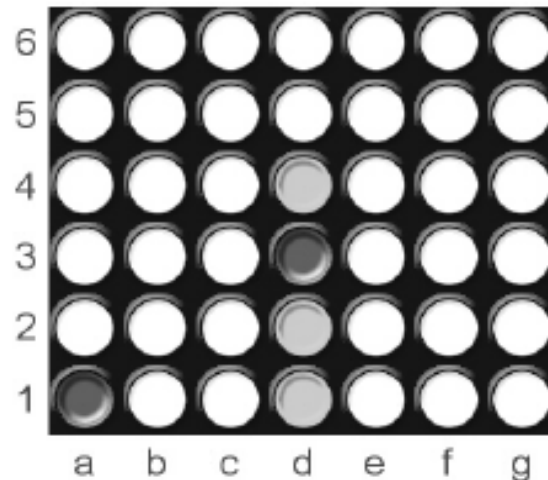
# Zobrist Keys for Connect-4

- Key Table:

piece-square	64-Bit-Zufallszahl
Weiß auf a1	1010100000101011011111101011101100111111010001100011000000100100
Schwarz auf a1	10101111111101111000010001000111111010100011101101110110110111011
Weiß auf a2	0111001010111111101100111011100010101000000100011101010110111001
Schwarz auf a2	0000001000011011110111100001011000011111011101001101000100011010
Weiß auf a3	1110100000110010011010100101101010011111100010101010000101111010
...	...
Weiß auf d1	0101101001010111011111000001100011000011110110011000001000010110
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:



```

1010111111110111100001000100011111010100011101101110110110111011 (Schwarz auf a1)
⊕ 0101101001010111011111000001100011000011110110011000001000010110 (Weiß auf d1)
⊕ 001000000110000011110001101100111000110111001010000000000001011 (Weiß auf d2)
⊕ 1100101101011110100001001000100100011001000010011100001111011011 (Schwarz auf d3)
⊕ 0100101001100010000110110011111001001011111101000000111101110111 (Weiß auf d4)
= 0101010011111100100101100101101111001000100110001010001100001010

```

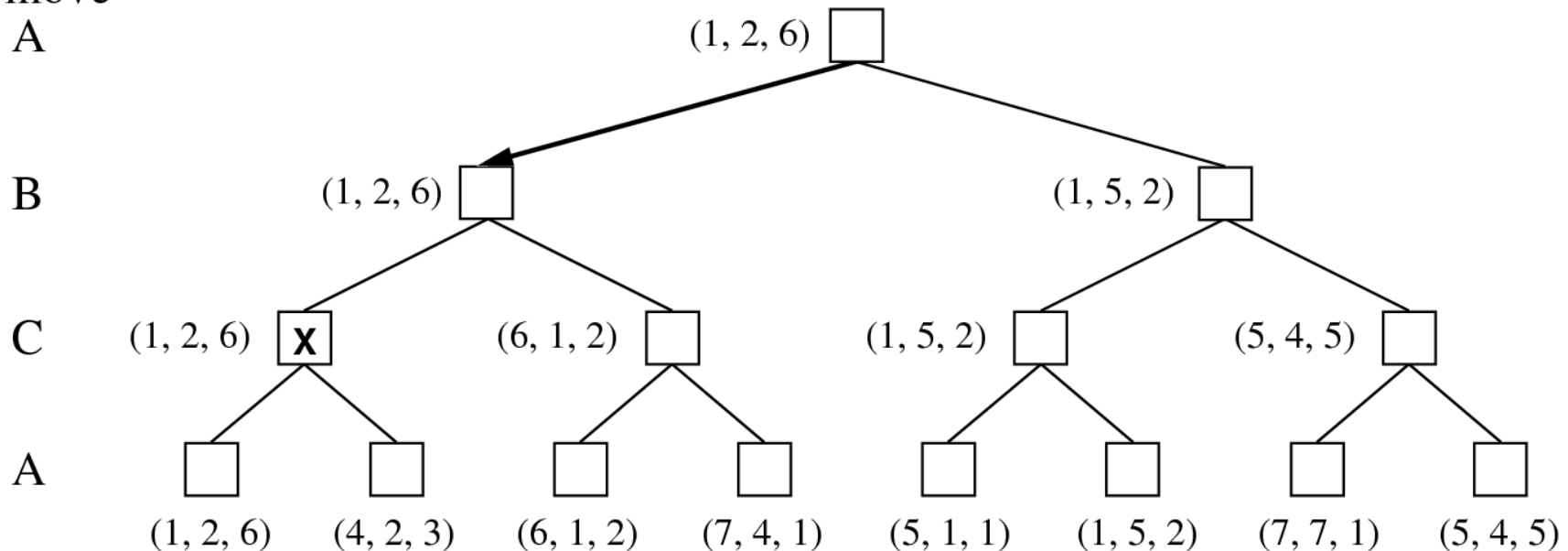
hash key for above position

# 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)  $\rightarrow f(n) = (f_A(n), f_B(n), f_C(n))$

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

to move





# 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)  $\rightarrow f(n) = (f_A(n), f_B(n), f_C(n))$

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

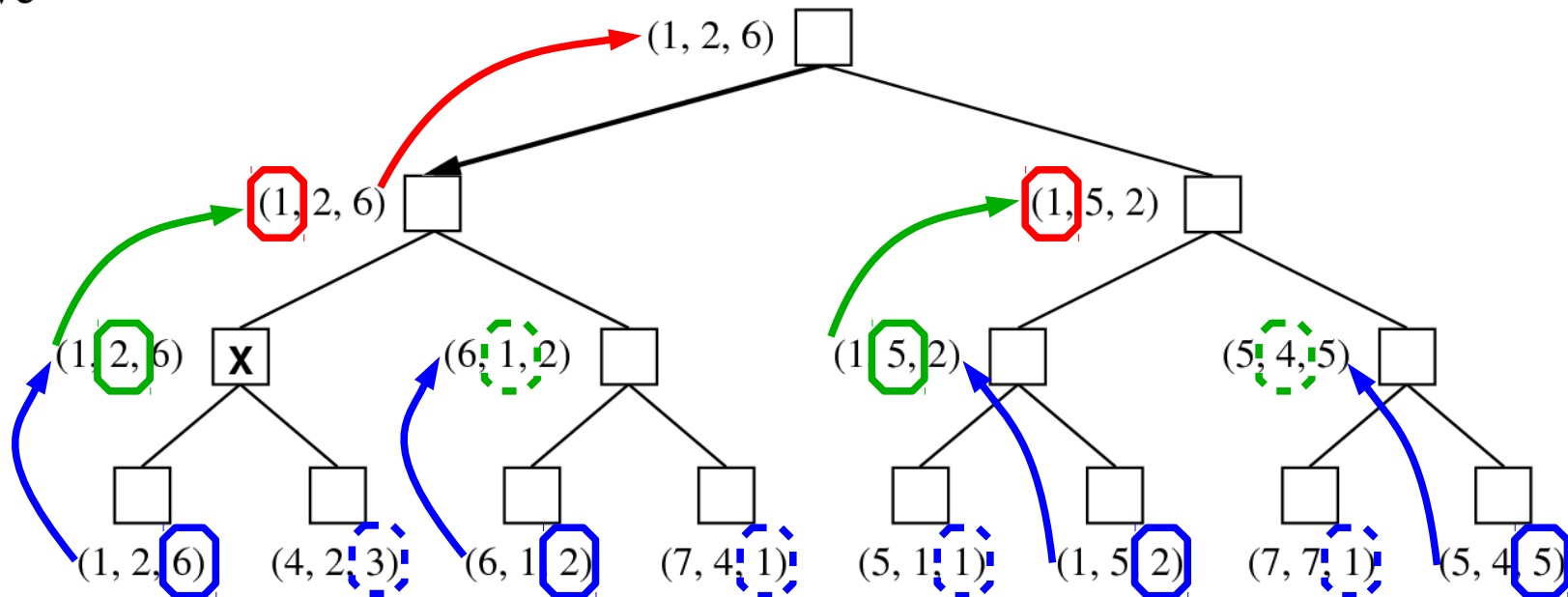
to move

A

B

C

A



# Outline

- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - $\alpha$ - $\beta$  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

# 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

MAX

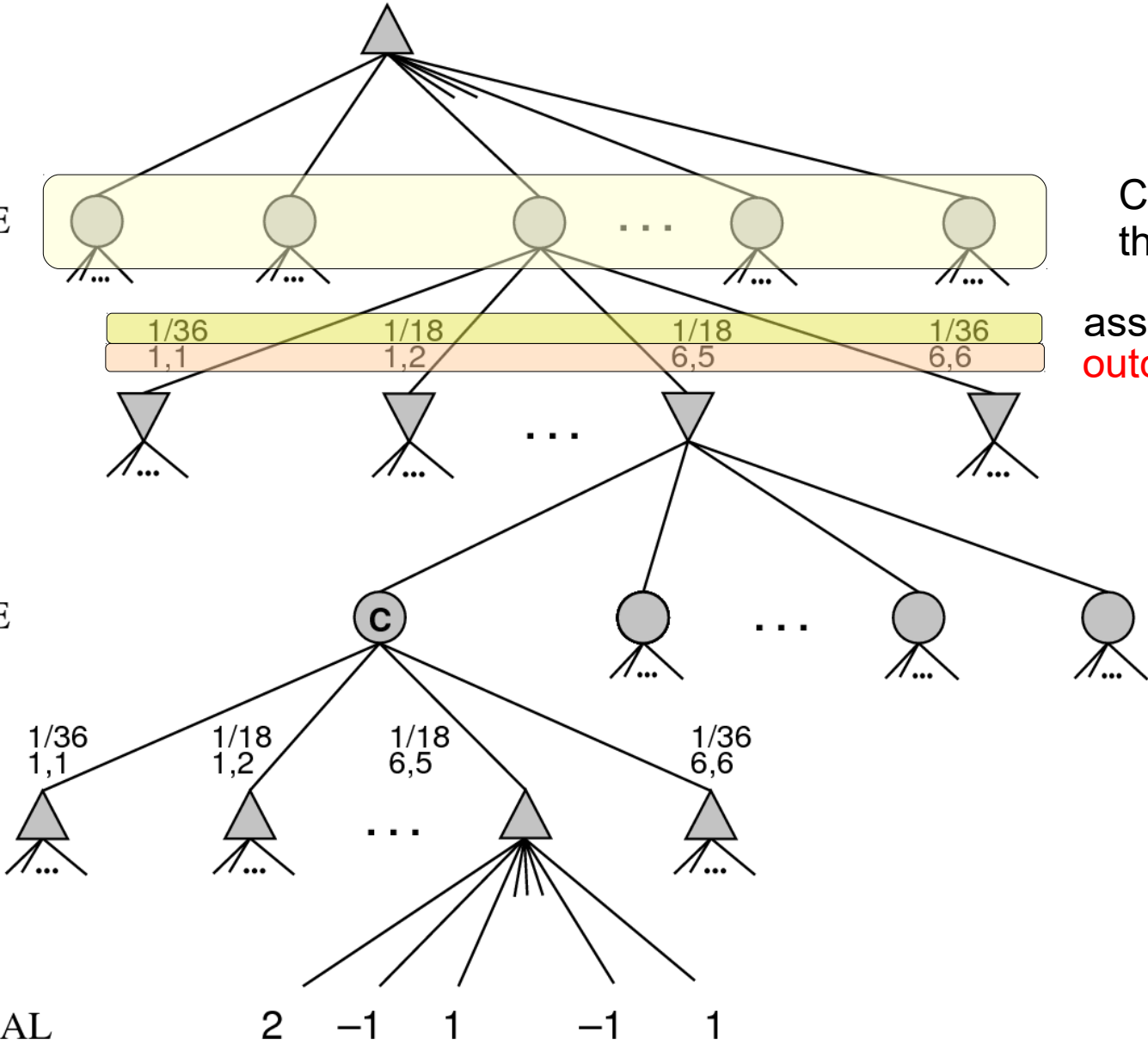
CHANCE

MIN

CHANCE

MAX

TERMINAL



Chance nodes for the roll of two dice

associated **probability** **outcome** of the dice roll

1/36	1/18	1/18	1/36
1,1	1,2	6,5	6,6

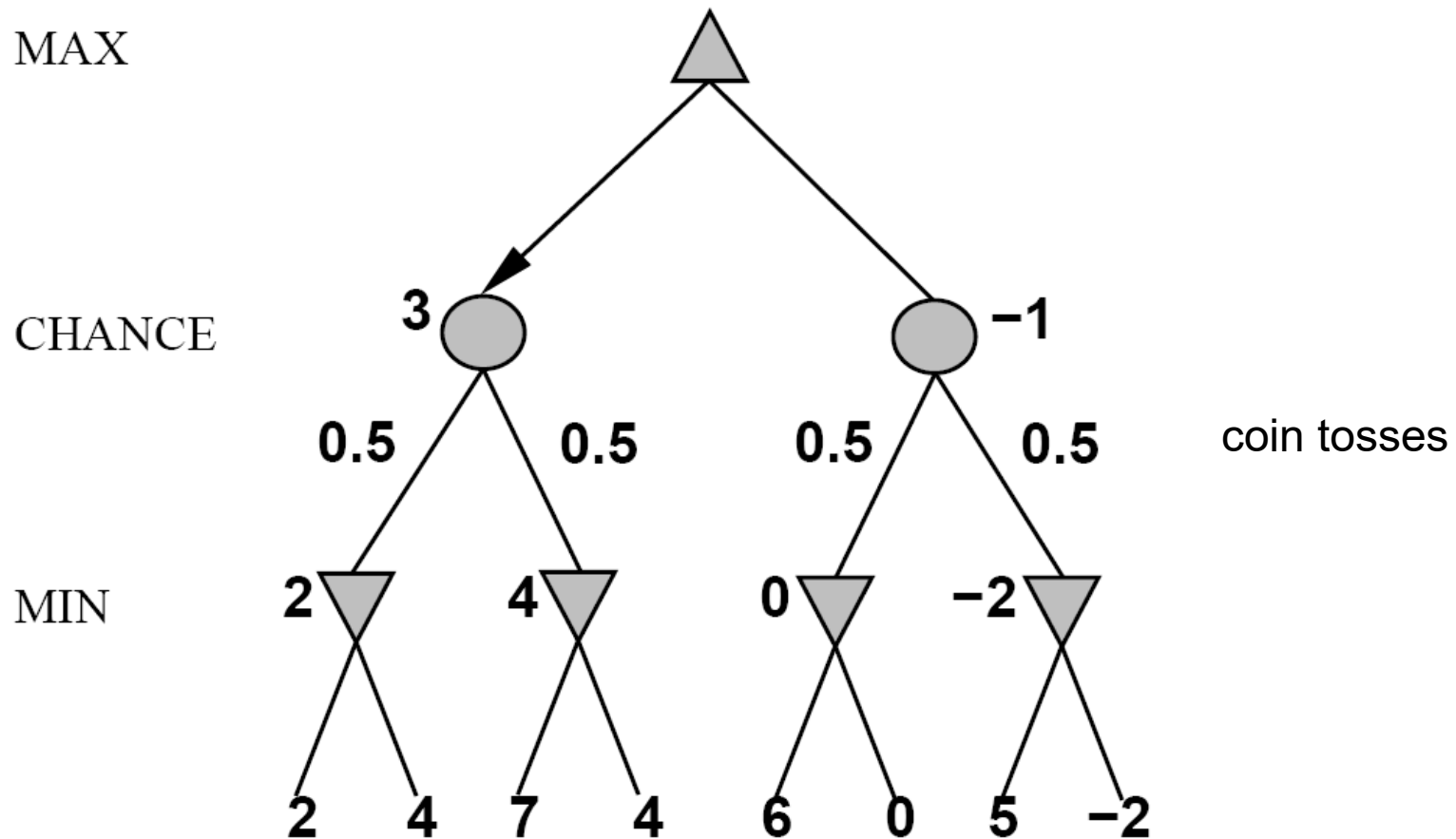
2 -1 1 -1 1

# Optimal Strategy with Chance Nodes

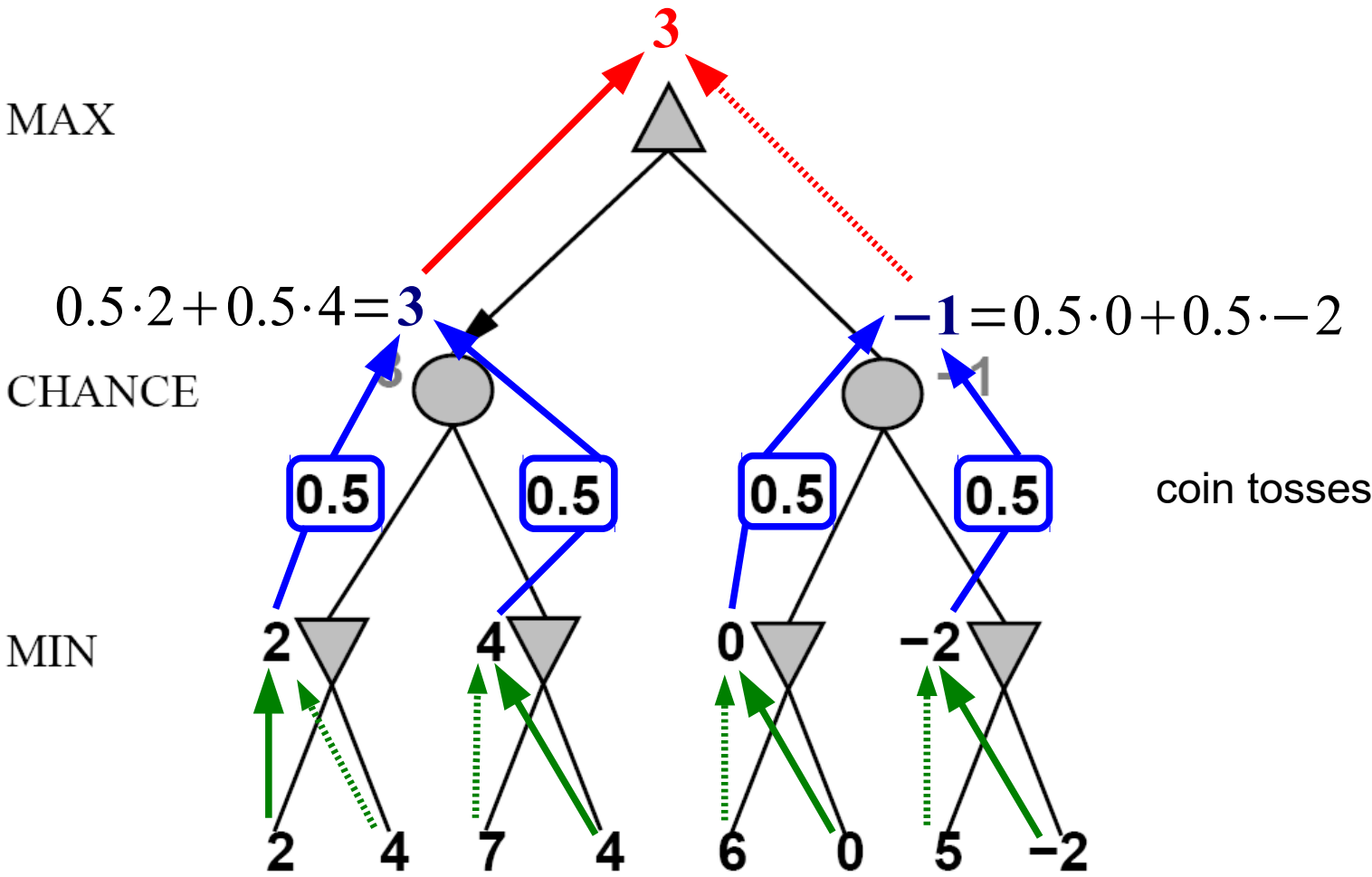
- MAX wants to play the move that maximizes his chances of winning
- Problem:
  - the exact outcome of a MAX-node cannot be computed because each MAX-node is followed by a chance node
  - analogously for MIN-nodes
- **Expected Minimax value**
  - compute the expected value of the outcome at each chance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

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

# Example



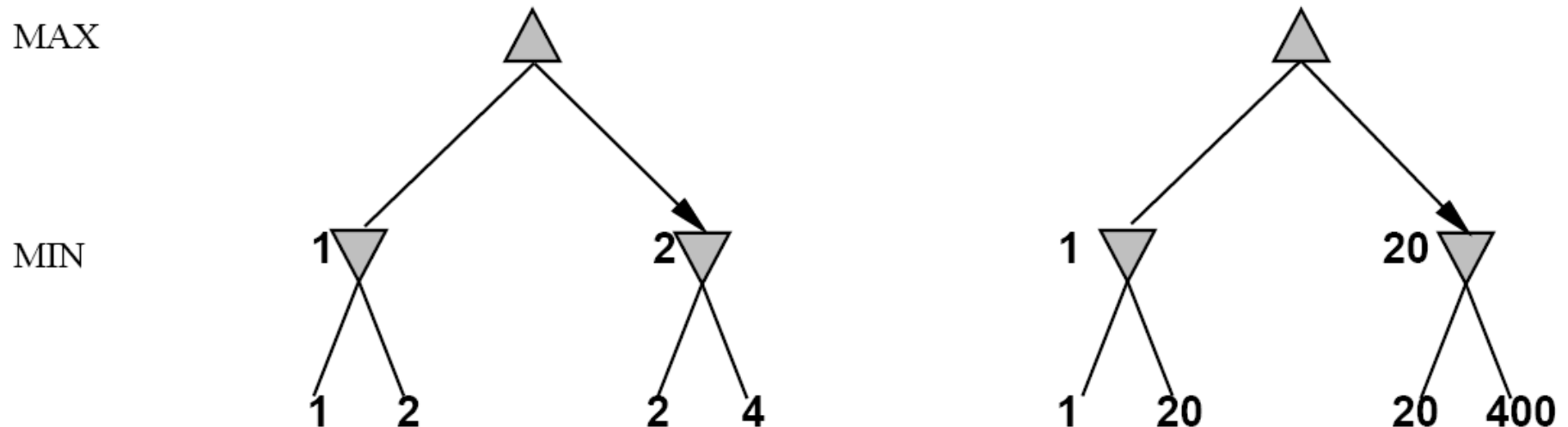
# Example



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)

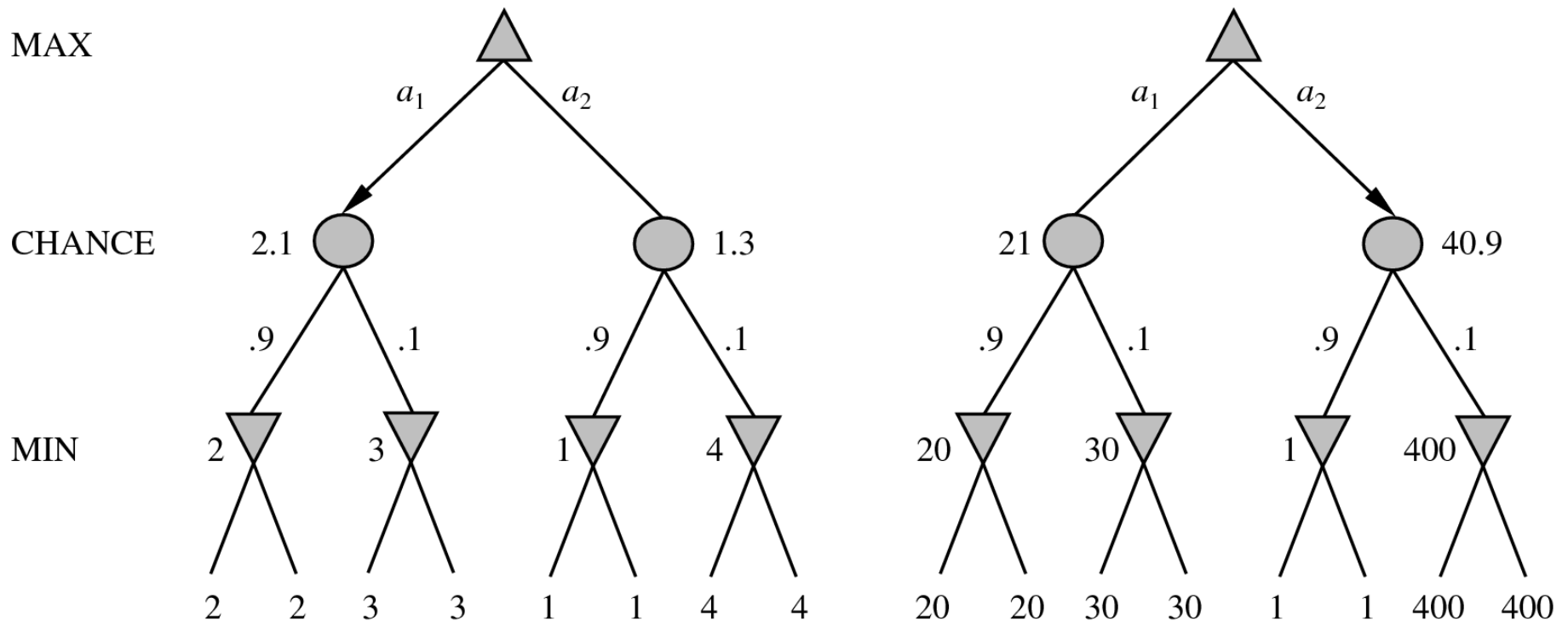


MAX plays the same move in both cases



# 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)

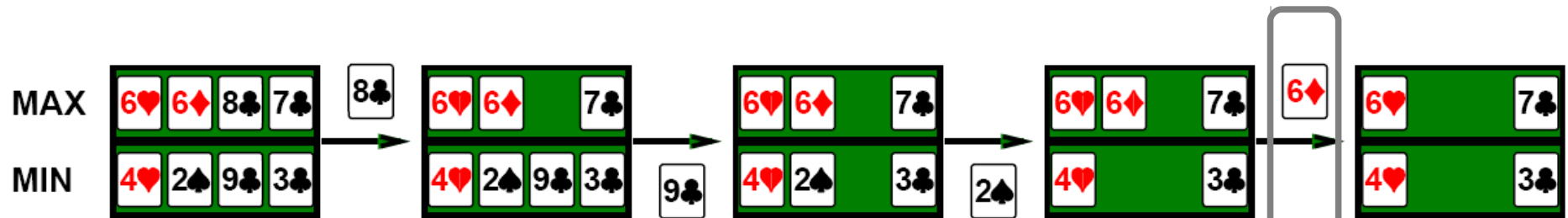
- at MIN and MAX nodes as usual
- at chance nodes, expected values can be bounded before all nodes have been searched if the value range is bounded

# 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** (a very good **Bridge program**) generates 100 deals consistent with bidding information (this also restricts!)
  - picks the move that wins the most tricks on average

# Imperfect Information - Example

*Scenario a)* MIN has 4♥



→ both players will make two tricks

MAX can play optimally if MAX knows MIN's cards

*Scenario b)* MIN has 4♦

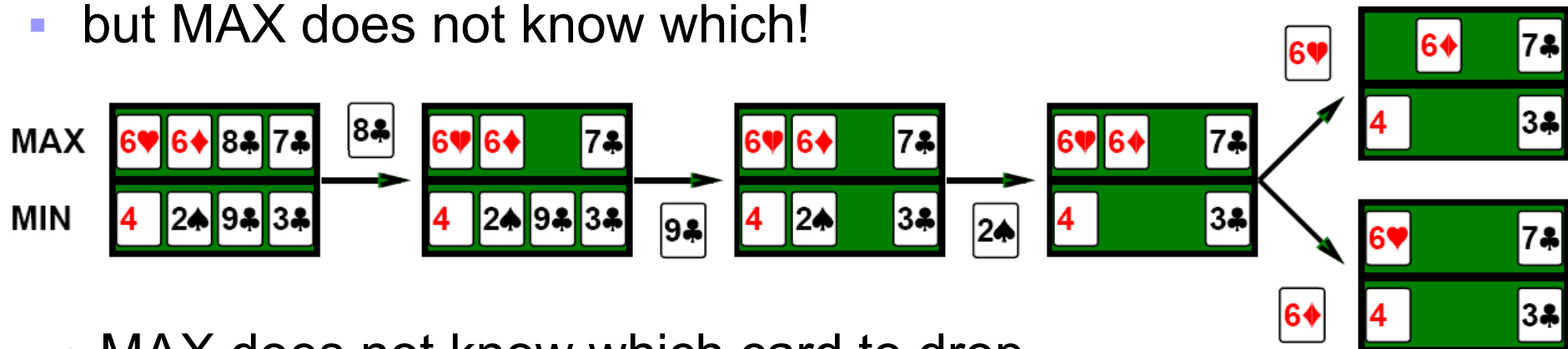


→ both players will make two tricks

# Imperfect Information - Example

*Scenario c)* MIN has either  $4♥$  or  $4♦$

- but MAX does not know which!

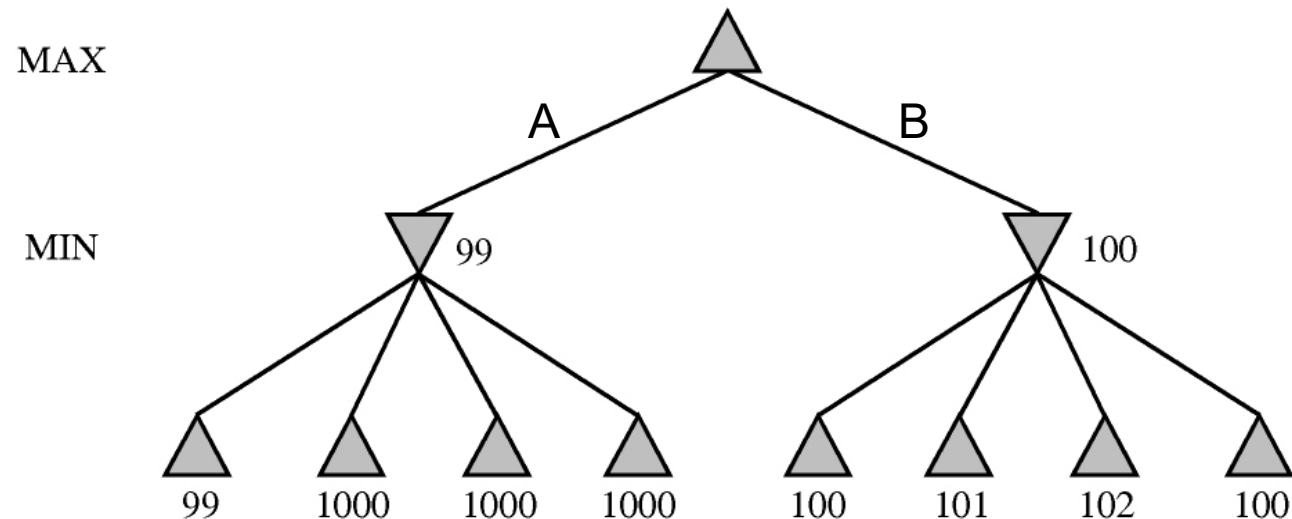


→ MAX does not know which card to drop  
and has a 50% chance of losing the game!

- 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)

# Belief States in Minimax Search

- Minimax always assumes that the opponent plays its best response (it is said to be *conservative*)
- 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!

# Opponent Modeling

Somewhat off-topic, but see also:  
<http://www.youtube.com/watch?v=3nxjztQKtY>

- 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 **part of the game**
  - e.g., bluffing and calling a bluff in Poker

**WORLD RPS SOCIETY**



# Outline

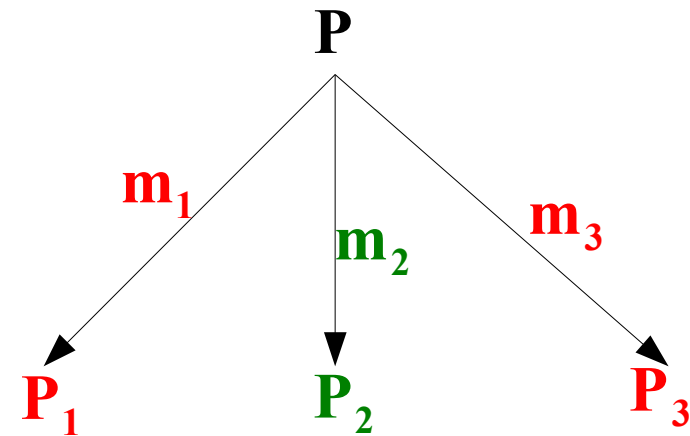
- Introduction
  - What are games?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - $\alpha$ - $\beta$  pruning
  - NegaScout
- Real-time Game-Tree Search
  - evaluation functions
  - practical enhancements
  - selective search
- Games of imperfect information and games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



# Game Playing

## Basic Setting

- The player  $P$  is faced with a choice of moves  $m_1$ ,  $m_2$ , and  $m_3$
- leading to positions  $P_1$ ,  $P_2$ , and  $P_3$
- the move that leads to the best position is preferred



Learning to make this choice may be:

classical approaches using reinforcement learning (e.g., TD-Gammon)

Utility-based:

$$u(P_1) = u_1$$

$$u(P_2) = u_2$$

$$u(P_3) = u_3$$

Preference-based:

$$P_2 \succ P_1$$

$$P_2 \succ P_3$$

current work in our group

# Multi-Armed Bandit Problems

(Robbins 1952)

## One-armed bandit:

- One action (pull the lever)
- Fixed long-term expected reward (or loss)



# Multi-Armed Bandit Problems

(Robbins 1952)

One-armed bandit:

- One action (pull the lever)
- Fixed long-term expected reward (or loss)

Multi-armed bandit



# Multi-Armed Bandit Problems

(Robbins 1952)

## One-armed bandit:

- One action (pull the lever)
- Fixed long-term expected reward (or loss)

## Multi-armed bandit

- $k$  actions (moves)  $\mathbf{m}_i$
- associated with different fixed but unknown long-term expected rewards / utilities  $u_i$



$u_1 = 150 \$$



$u_2 = 100 \$$



$u_3 = 300 \$$

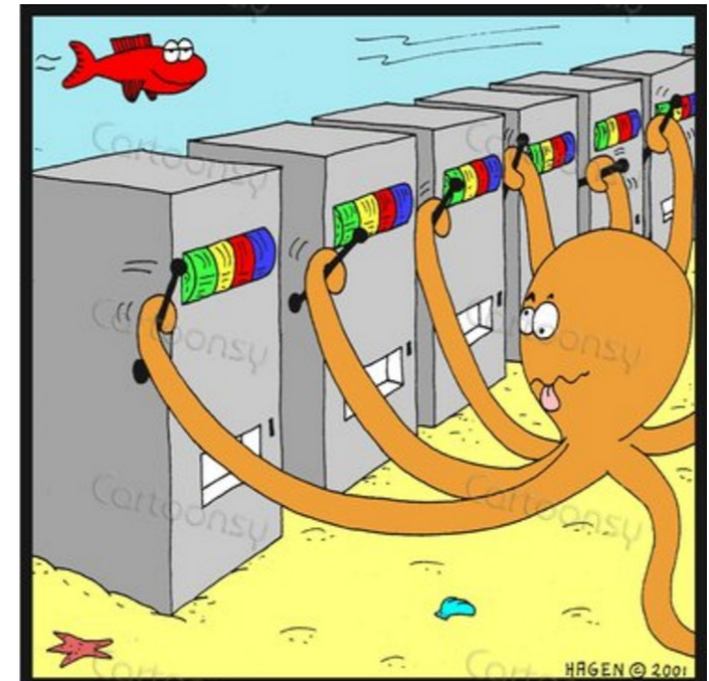


$u_4 = 200 \$$

# Learning Problem

**Task: Find a strategy (policy) to maximize your gain**

- Find the arm (move) with the highest long-term utility from observations of immediate utilities
- with as little regret as possible
- and then keep playing it



# Multi-Armed Bandits – Learning Problem

Which arm should I play next (at time  $t$ )?



$u_1 = ?$



$u_2 = ?$



$u_3 = ?$



$u_4 = ?$

Winning  $w_i^{(t)} = 50 \$$

And which arm should I play next (at time  $t + 1$ )?

# Upper Confidence Bound (UCB) Algorithm

(Auer et al. 2002)

## Problem: **Exploitation vs. Exploration**

- **Exploitation:** Pull the best arm in order to maximize your earnings
- **Exploration:** Try if other arms are more promising

## UCB algorithm

- Always Play the Arm with the highest Upper-Confidence Bound

$$UCB\ 1 = \frac{\sum_t w_i^{(t)}}{n_i} + \alpha \sqrt{\frac{\log(N)}{n_i}}$$

$$\begin{aligned} \sum_t w_i^{(t)} & \text{ total win for } \mathbf{m}_i \\ n_i & \text{ \#games with } \mathbf{m}_i \\ N = \sum_i n_i & \text{ total \#games} \end{aligned}$$

### Exploitation Term

Empirical estimate of  $r_i$   
from  $n_i$  observed earnings  $w_i^{(t)}$

### Exploration Term

Inverse percentage  
of trials of arm  $i$

# More realistic Games

Bandits are a very simple “game”

- only one game state
- fixed stochastic reward distribution independent of adversary

**More realistic games** require

- **Game States:** Most games have different states (positions) described with features, and possibly different move sets
  - utility will depend on the state and the features used for describing it
- **Delayed Reward:** Feedback will not be available right after the move played, but after a **sequence of moves** (often entire game)
- **Adversary:** Obtained feedback is not only for the own quality of play but also about the opponent’s play
- **Search:** Look-ahead is often more important than good utility estimates

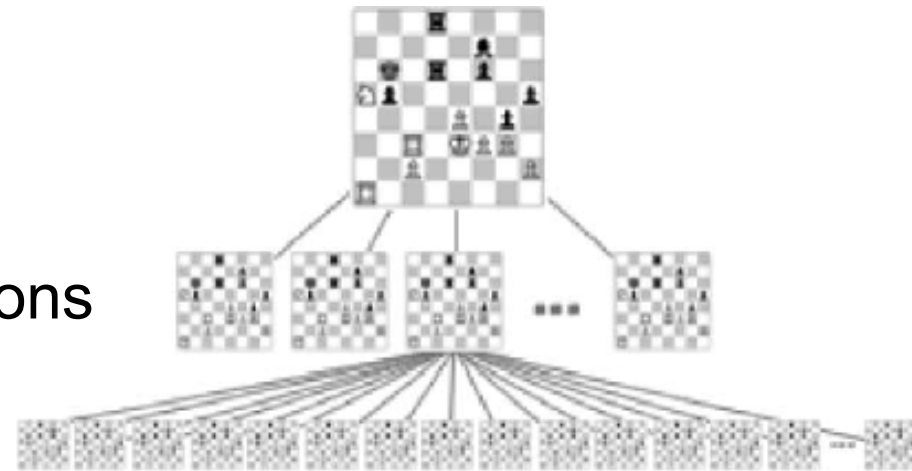


# Search-Based Game Playing

Conventional game playing programs combine

## Search:

- look ahead for a certain number of moves
- Minimax principle, search extensions



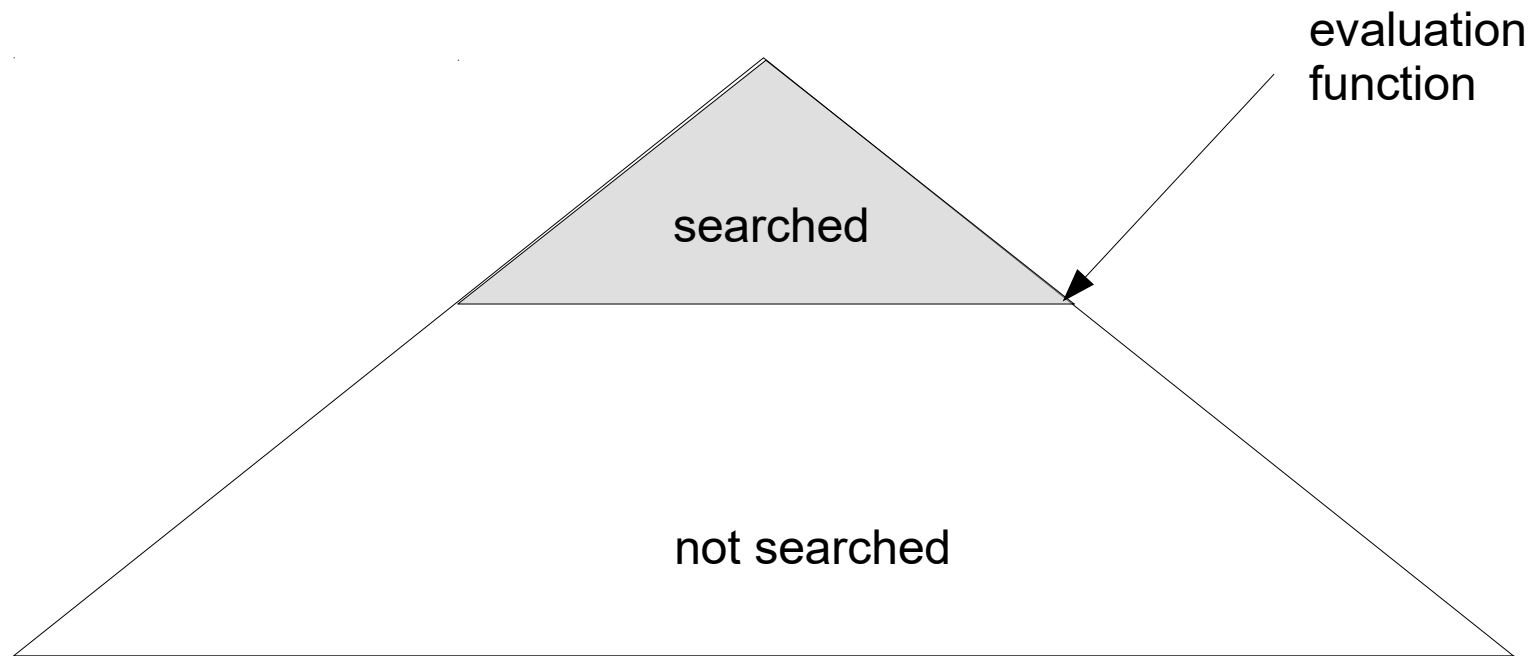
## Evaluation:

- evaluate the positions at the search horizon with an evaluation or utility function
- Two principle approaches for estimating utility of positions
  1. using a fixed, heuristic **evaluation** function
  2. approximation by (randomly) **sampling** a large number of games

**Learning in Games** → typically learning of an evaluation function

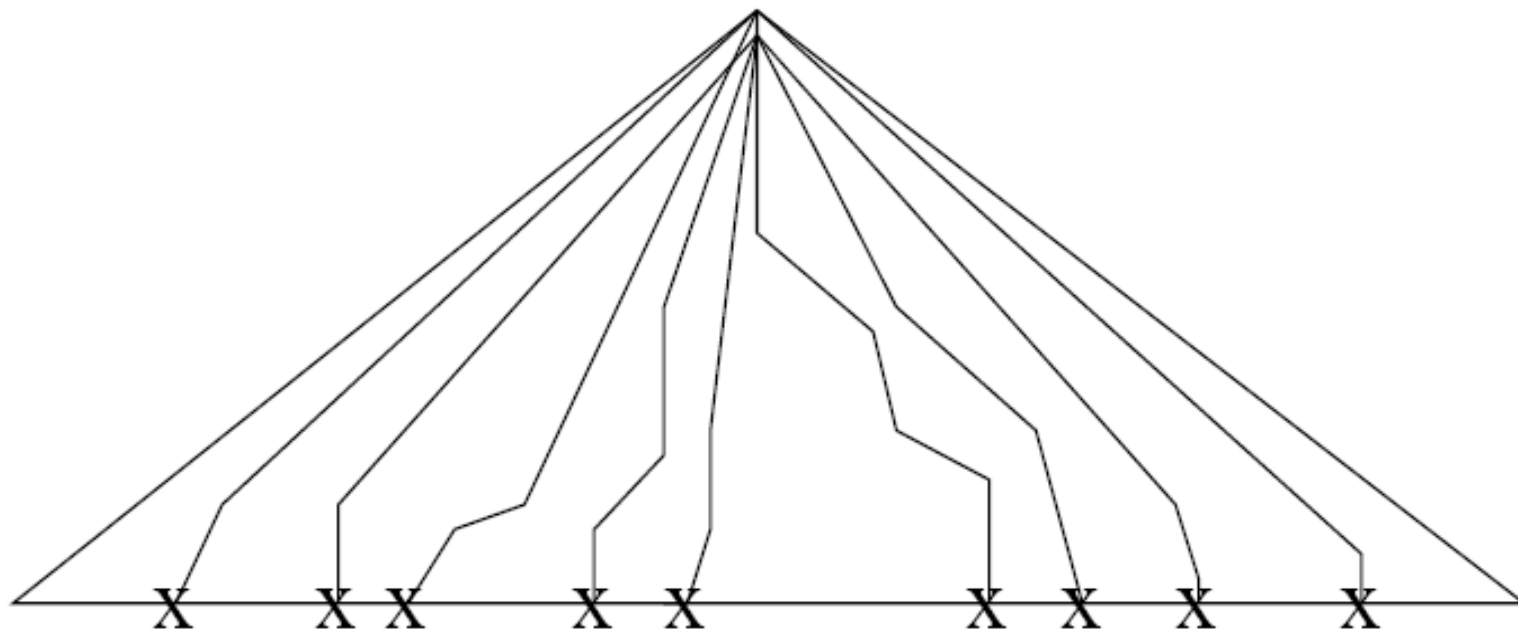
# Simulation Search – Key Idea

- The complete tree is not searchable
  - thus minimax/alpha-beta **limit the depth** of the search tree
    - **search all variations to a certain depth**



# 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

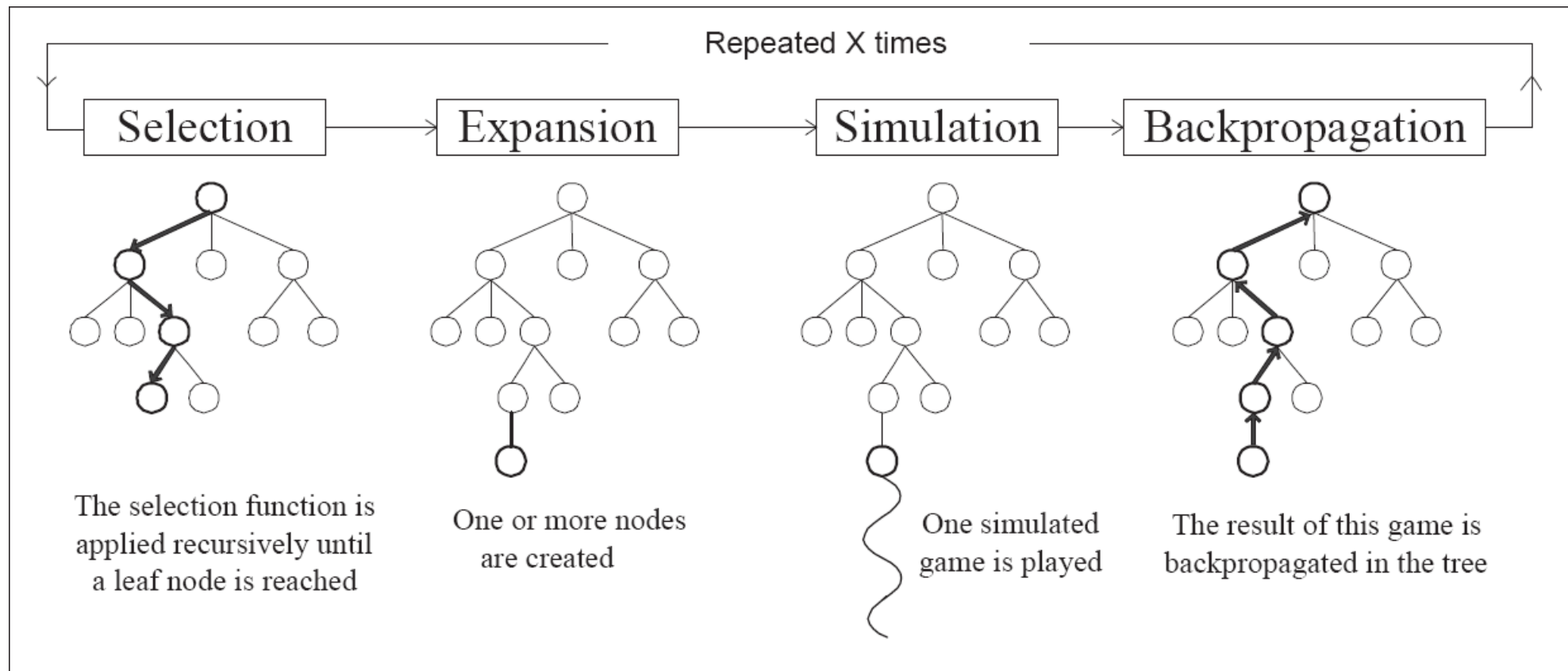
- **Algorithm Sketch:**
  - estimate the expected value of each move by counting the number of wins in a series of complete games
  - at each chance node select one of the options at random (according to the probabilities)
  - at MAX and MIN nodes make moves (e.g., guided by a fast evaluation function)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

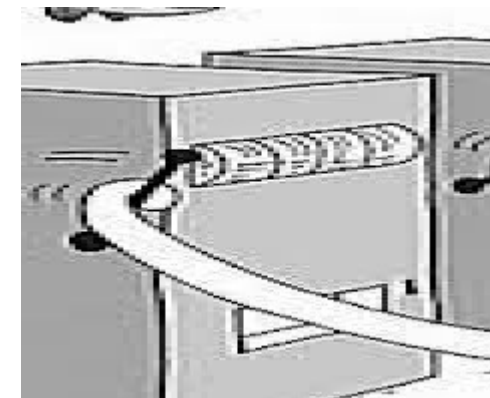
# UCT Search

(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  $value(n)$  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)

# 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

