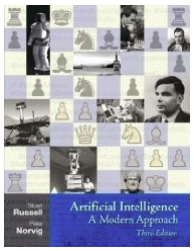
Outline

- Introduction
  - What are games and why are they interesting?
  - History and State-of-the-art in Game Playing
- Game-Tree Search
  - Minimax
  - NegaMax
  - α-β pruning
- Real-time Game-Tree Search
  - NegaScout
  - evaluation functions
  - practical enhancements
  - selective search
- Multiplayer Game Trees



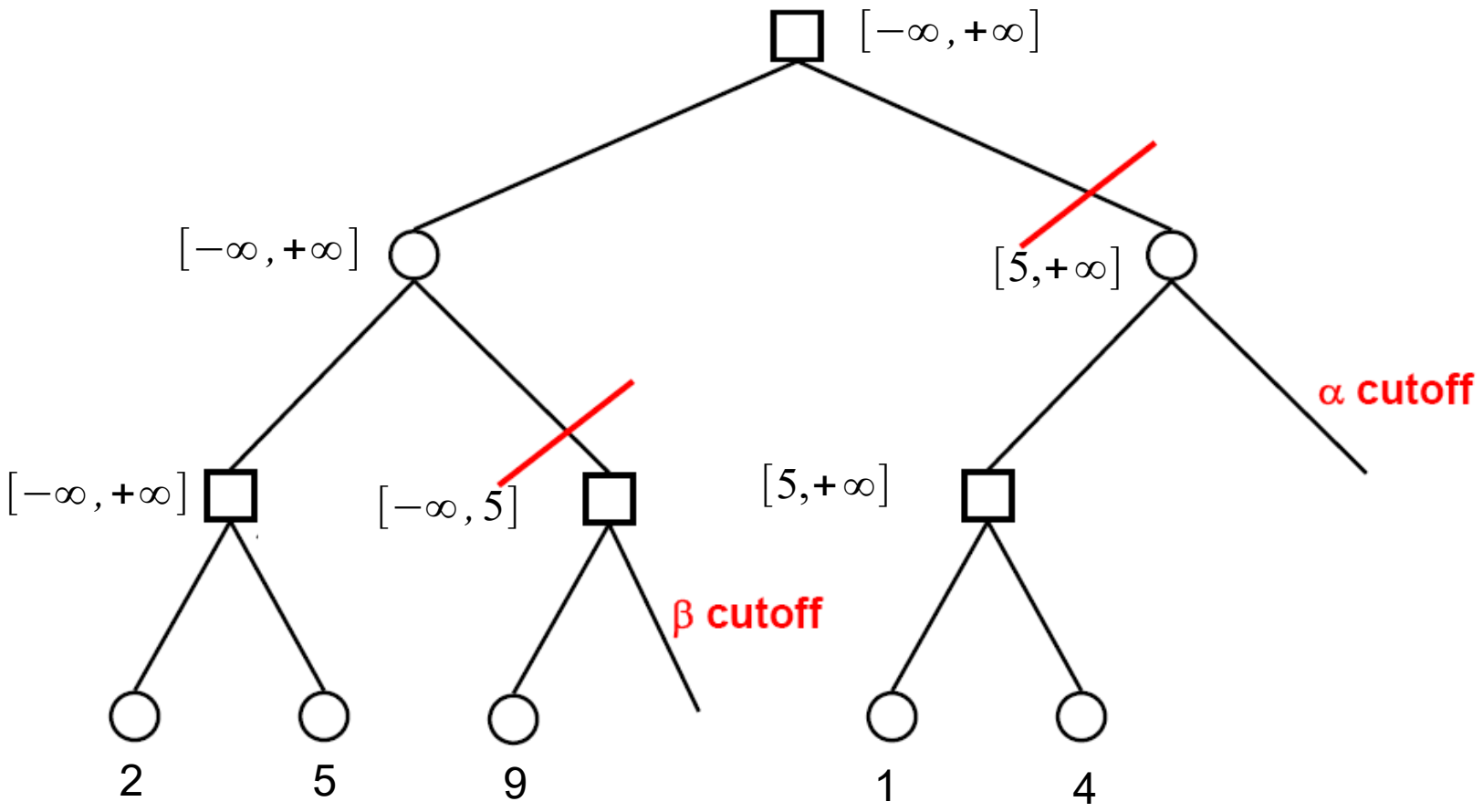
Many slides based on Russell & Norvig's slides Artificial Intelligence: A Modern Approach

Alpha-Beta – NegaMax Formulation

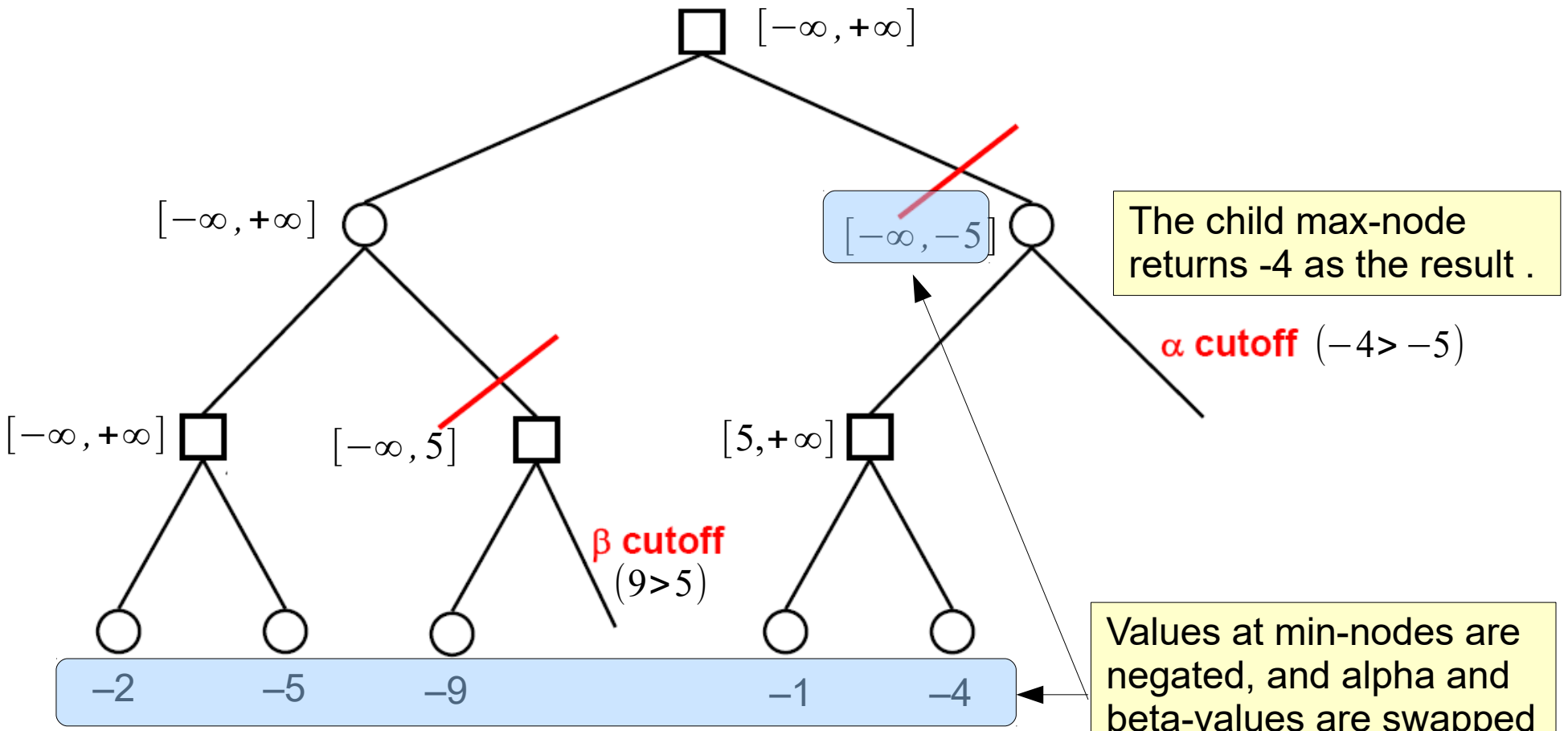
```
int AlphaBeta ( position p; int α, β );
{                                   /* compute minimax value of position p */
        int a, t, i;
        determine successors p_1,...,p_w of p;
        if ( w == 0 )
                return ( Evaluate(p) );                    /* leaf node */
        a = α;

        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!

Alpha-Beta (Min-Max Formulation)



Alpha-Beta (NegaMax Formulation)

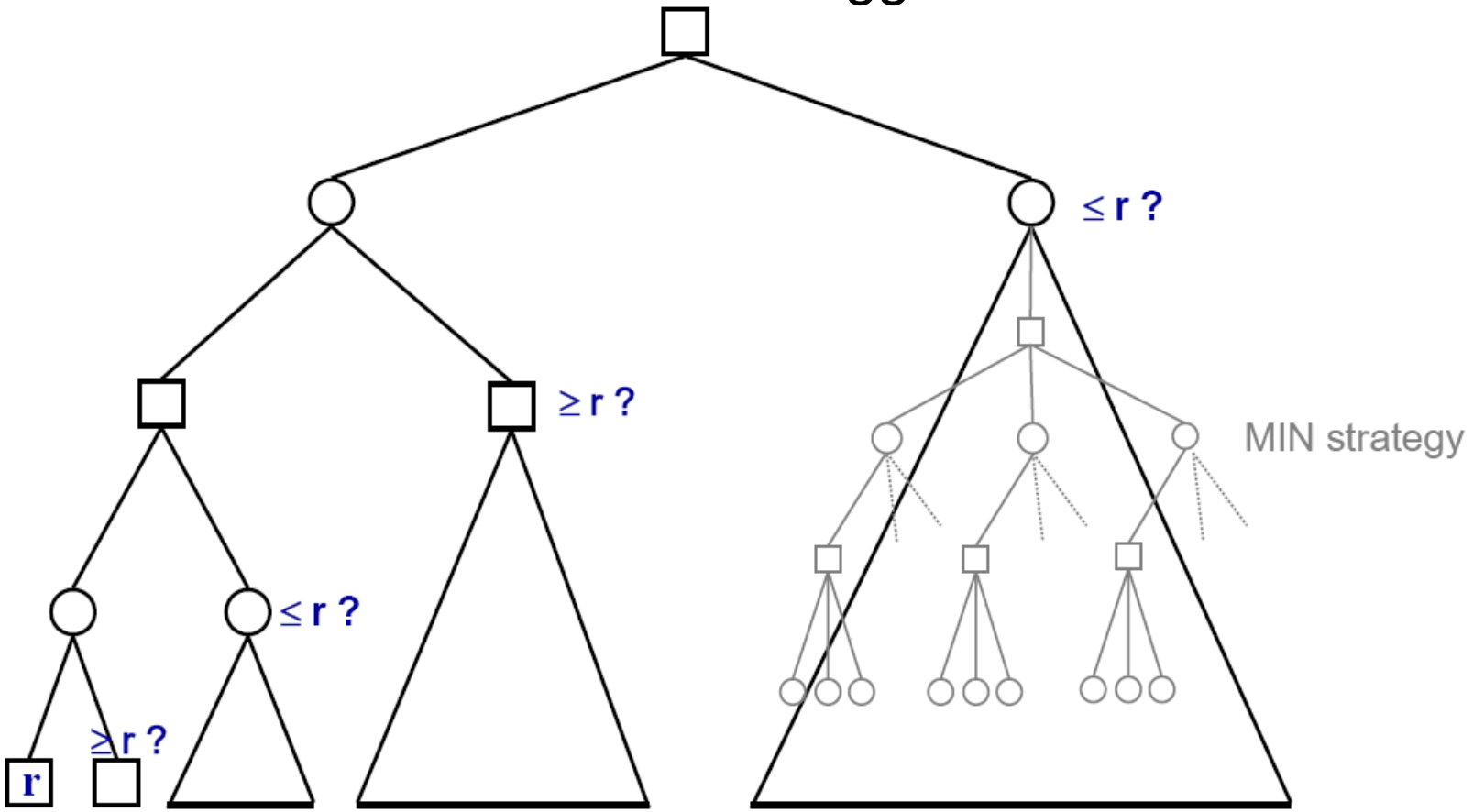


Minimal Window Search

- 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

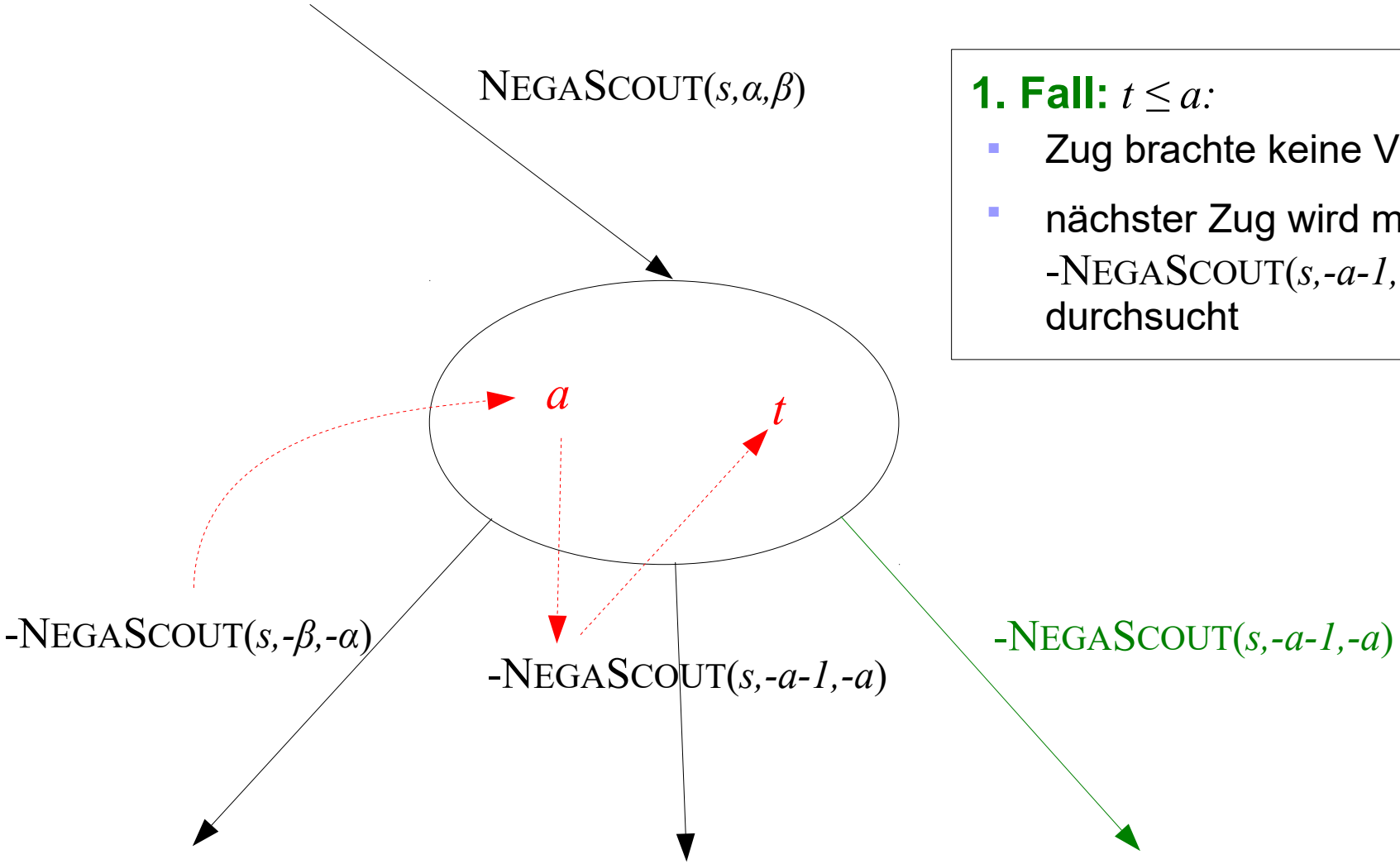


NegaScout (Reinefeld 1982)

```
int NegaScout ( position p; int α, β );
{                                   /* compute minimax value of position p */
        int a, b, t, i;
        determine successors p_1,...,p_w of p;
        if ( w == 0 )
                return ( Evaluate(p) );                    /* leaf node */
        a = α;
        b = β;
        for ( i = 1; i <= w; i++ ) {
                t = -NegaScout ( p_i, -b, -a );
                if ( t > a && t < β && i > 1 && d < maxdepth-1 )
                        a = -NegaScout ( p_i, -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)

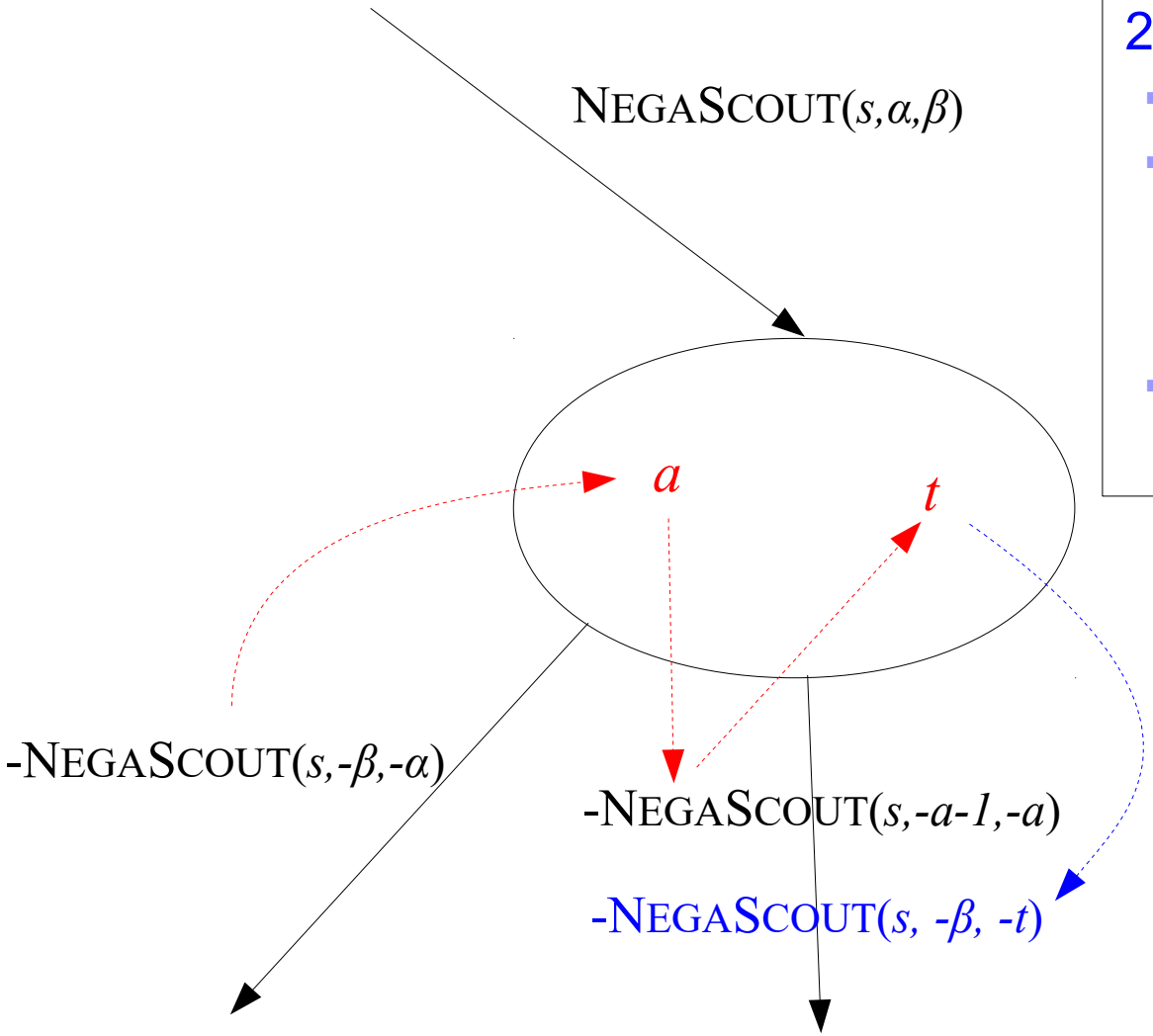
Schematische Darstellung des Ablaufs in einem MAX-Knoten



1. Fall: $t \leq a$:

- Zug brachte keine Verbesserung
- nächster Zug wird mit -NEGASCOUT(*s,-a-1,-a*) durchsucht

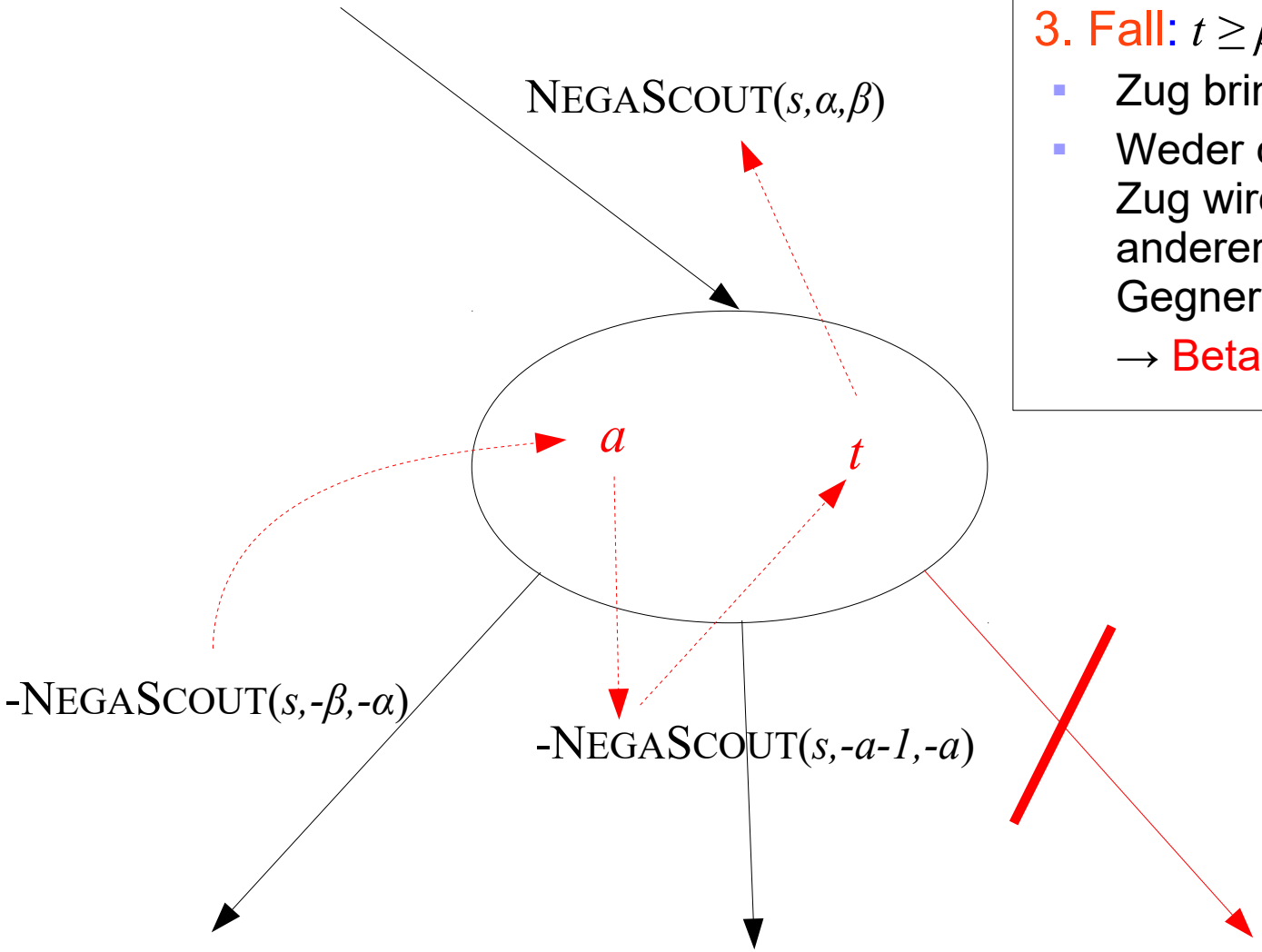
Schematische Darstellung des Ablaufs in einem MAX-Knoten



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

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

Schematische Darstellung des Ablaufs in einem MAX-Knoten

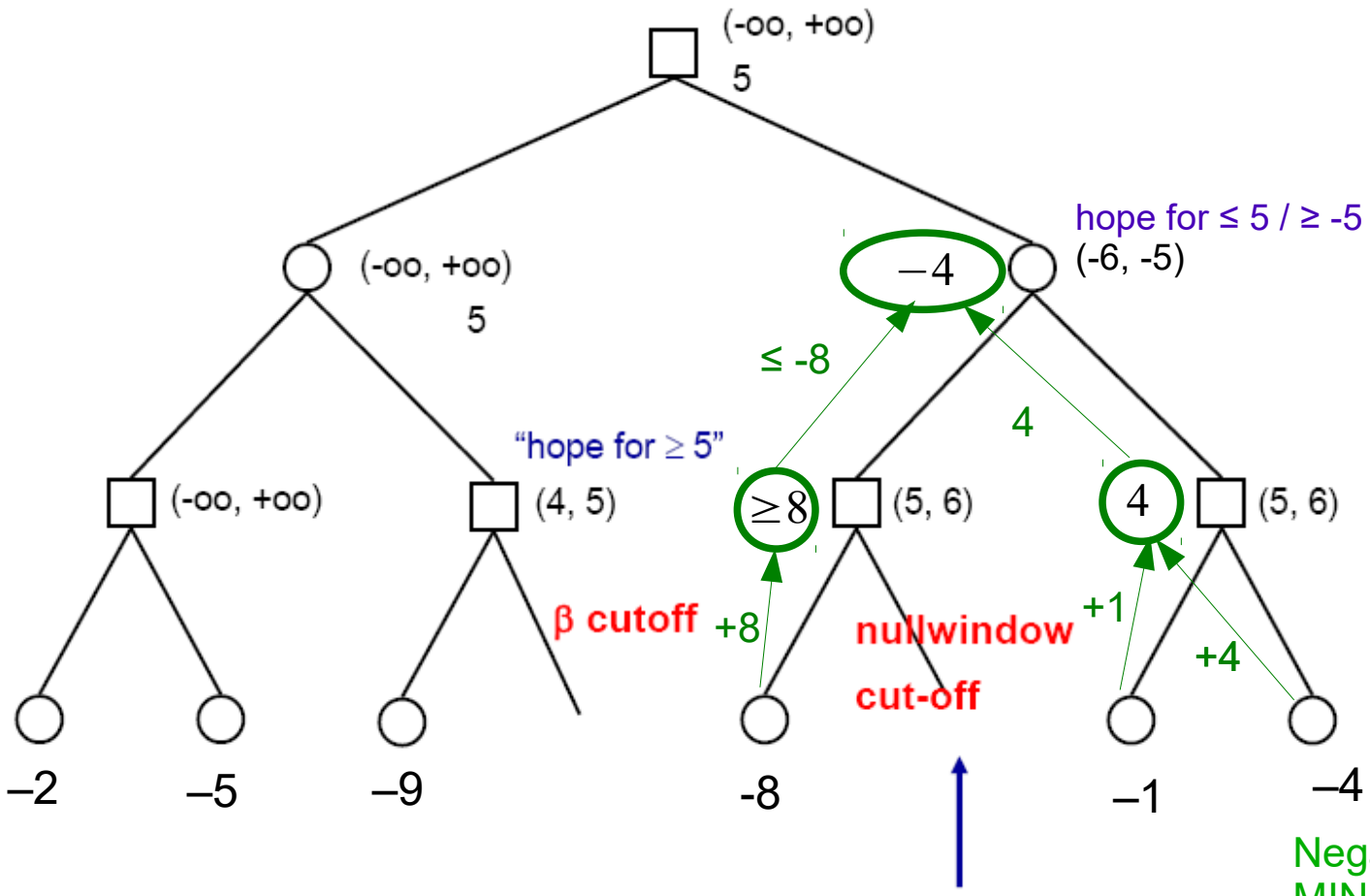


3. Fall: $t \geq \beta$:

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

  → Beta-Cutoff

NegaScout Example

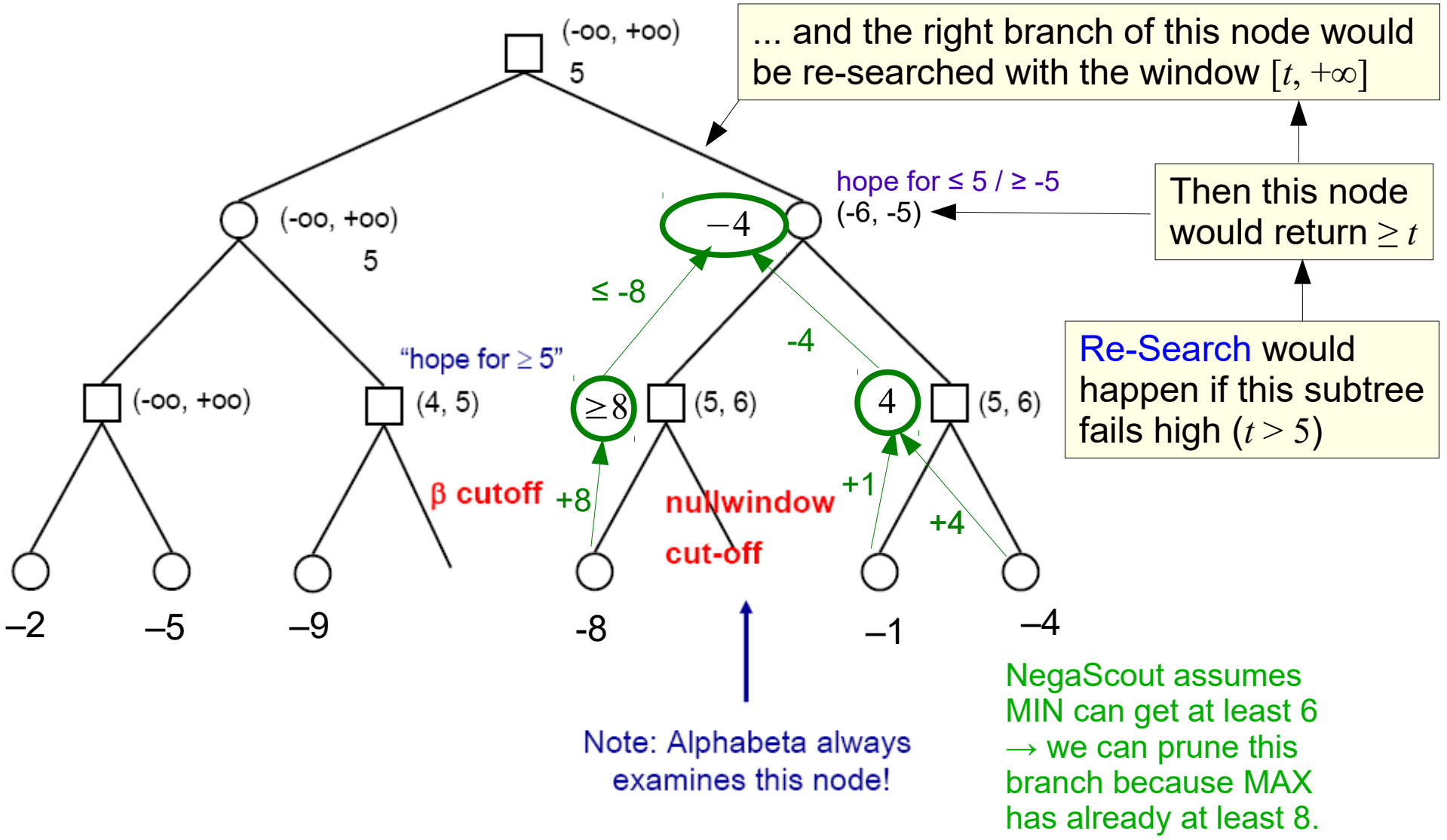


The assumption has turned out to be correct (4 ≤ 5), so the null Window cut-off was justified.

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

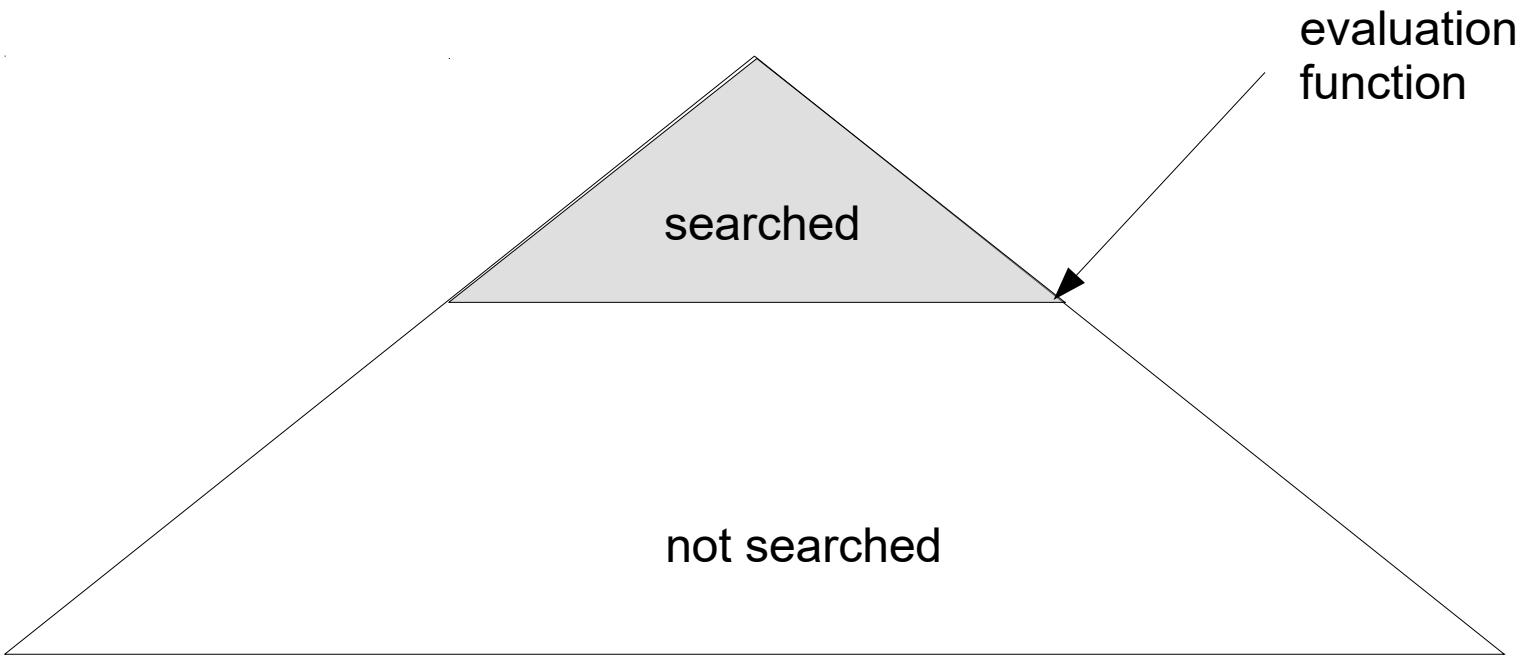
- The move ordering is crucial to the performance of alpha-beta search
- Domain-dependent heuristics:
  - capture moves first
    - ordered by value of capture
  - forward moves first
- Domain-independent heuristics:
  - Killer Heuristic
    - manage a list of moves that produced cutoffs at the current 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)

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

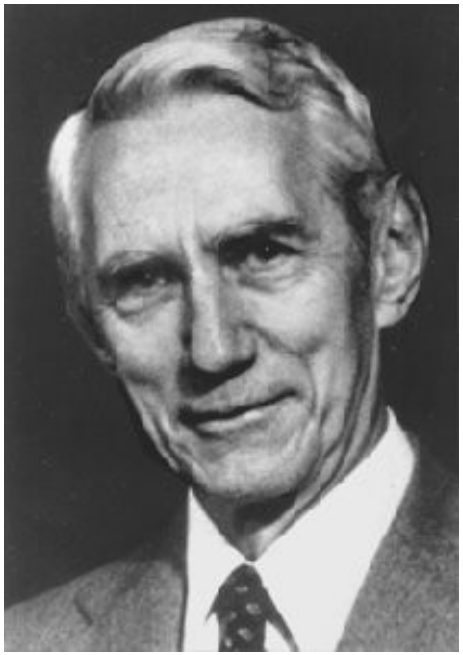
Using Evaluation Functions

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



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

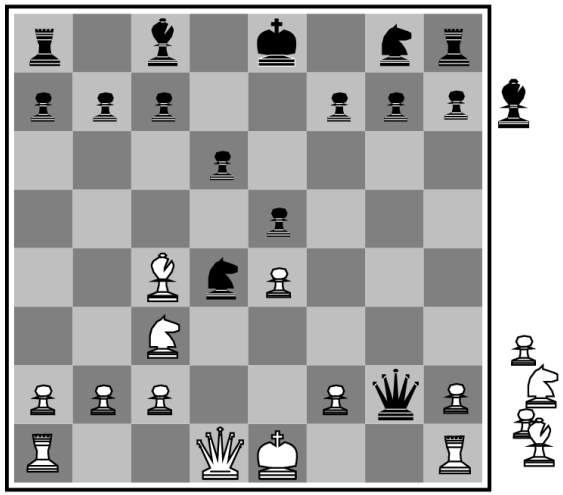
Evaluation Function Examples

- 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

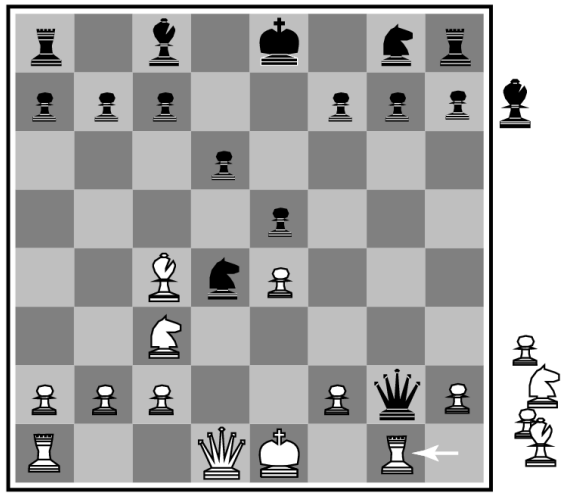
Evaluation is typically very brittle

- 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

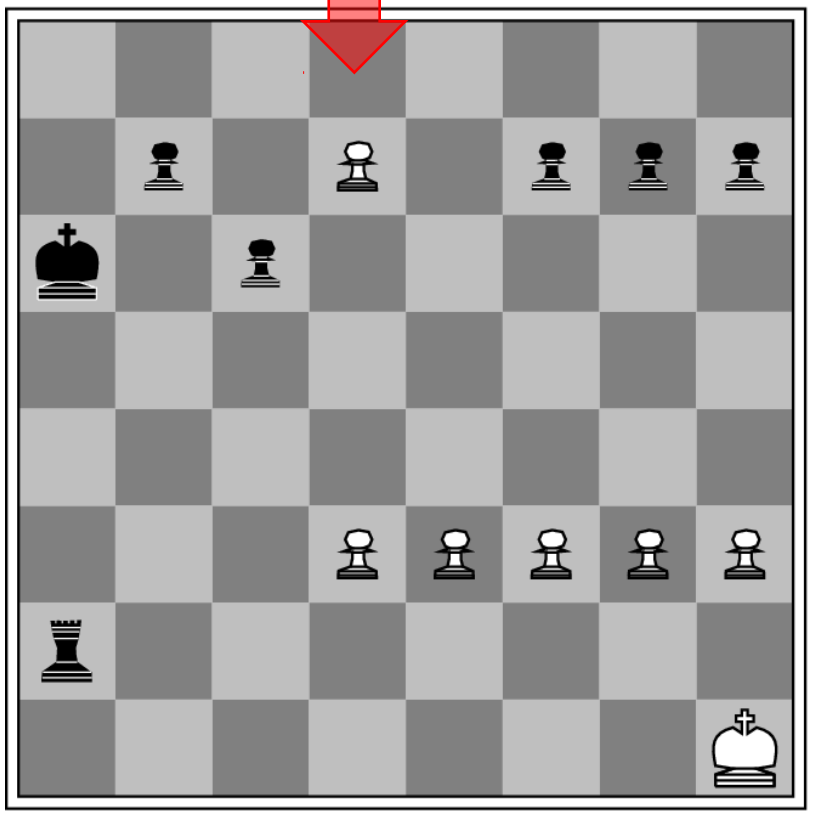
- Evaluation only useful for quiescent states
  - states w/o wild swings in value in near future
  - e.g.: states in the middle of an exchange are not quiet
- Algorithm
  - When search depth reached, compute quiescence state evaluation heuristic
  - If state quiescent, then proceed as usual; otherwise 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!

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

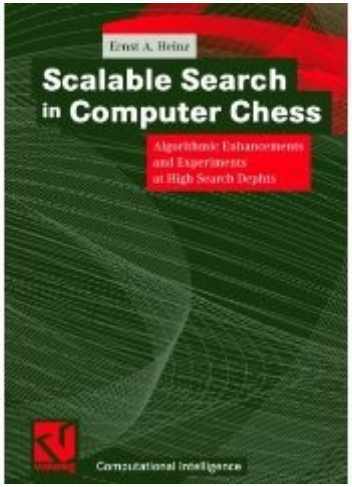
- game-playing programs sometimes extend the search depth
  - typically by skipping the step that increments the current search depth
    - increments with fractional values are also possible (multiple fractional extensions are needed for an extension by 1)
  - search is then continued as usual (until 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

Iterative Deepening

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

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

Advantages:

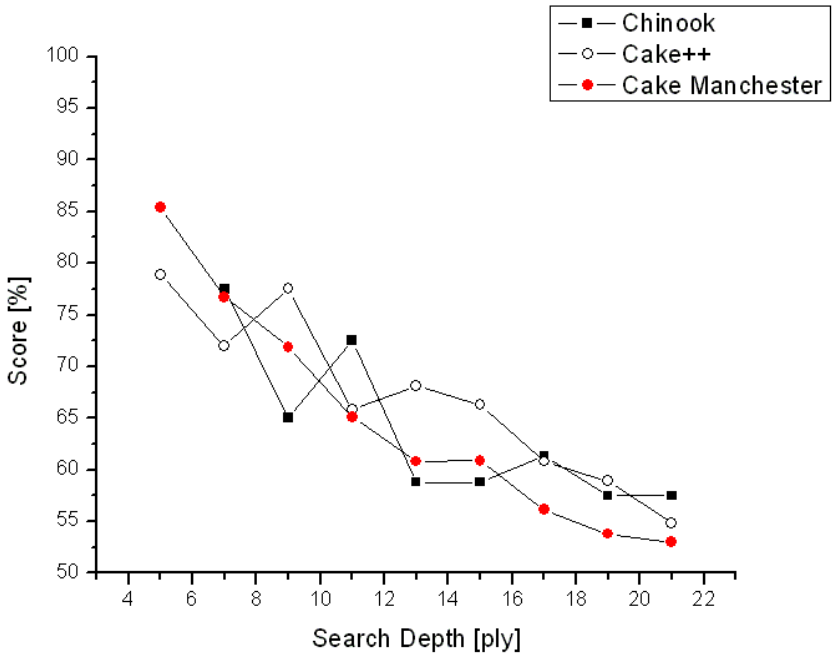
- works well with transposition tables
- improved dynamic move-ordering in alpha-beta
  - what worked well in the previous iteration is tried first in the next iteration
- simplifies time managements
  - if there is a fixed time limit per move, this can be handled flexibly by adjusting the number of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

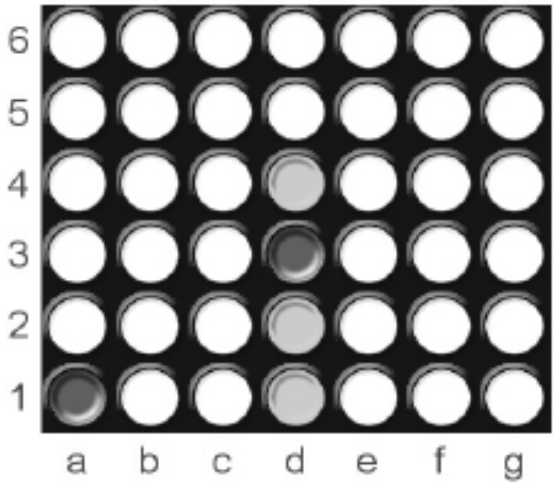
Zobrist Keys for Connect-4

- 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	0010000001100000111000110110011100011011100101000000000000001011
Schwarz auf d2	0000111101011001110011011111010011110011101010100100100001100100
Weiß auf d3	1000101100110111110001110110100000001001100001011000101001000000
Schwarz auf d3	1100101101011110100001001000100100011001000010011100001111011011
...	...
Schwarz auf g6	0110100110010001111100000001101100111010010111111000101001011 10

Zobrist Keys for Connect-4

- Computation of a position key:



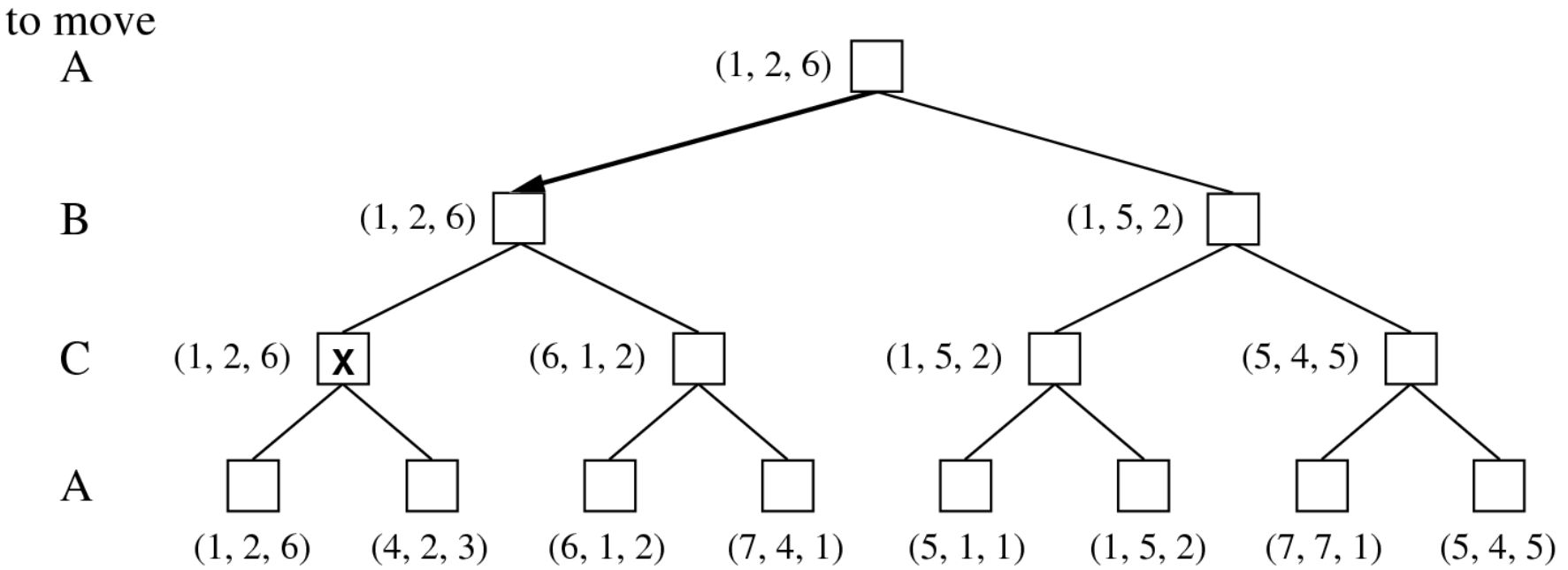
$$
\begin{array}{ll}
 & 10101111111101111000010001000111110101000111011011101101110111 \text{ (Schwarz auf a1)} \\
\oplus & 01011010010101110111110000011000110000111101100110000010000010110 \text{ (Weiß auf d1)} \\
\oplus & 00100000011000001111000110110011100011011100101000000000000001011 \text{ (Weiß auf d2)} \\
\oplus & 11001011010111101000010010001001000110010000100111000011111011011 \text{ (Schwarz auf d3)} \\
\oplus & 01001010011000100001101100111110010010111111010000001111011101111 \text{ (Weiß auf d4)} \\
= & 01010100111111001001011001011011110010001001100010100011000010 10
\end{array}
$$

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) → $f(n) = (f_A(n), f_B(n), f_C(n))$

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



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)