Programmieren 1

Die Reversi-Challenge

Human-Computer Interaction Group

Prof. Dr. Michael Rohs
michael.rohs@hci.uni-hannover.de

Leibniz Universität Hannover

Programmieren I Winter Challenge

```
 |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|_|_|_|_|
4|_|_|_|O|X|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

- Programmieren Sie einen Computer-Reversi-Spieler
- Reversi Turnier
  - alle eingereichten Spiele treten gegeneinander an
- Teilnahme freiwillig, zählt nicht als reguläres Übungsblatt
- zusätzliche Übungspunkte
  - 1 Punkt, beste 50%
  - 2 Punkte, beste 10%

Reversi Rules

Initial board:

```
 |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|_|_|_|_|
4|_|_|*|O|X|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

- Rules
  - X moves first
  - Pieces are reversed if surrounded (on two sides) by opponent pieces
  - A valid move is one in which at least one piece is reversed
  - If one player cannot make a move, play passes back to the other player
  - When neither player can move, the game ends, the player with the most pieces wins
- Details
  - https://en.wikipedia.org/wiki/Reversi

Reversi Rules

X places piece at C4:

```
 |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|*|_|_|_|
4|_|_|X|X|X|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

- Rules
  - X moves first
  - Pieces are reversed if surrounded (on two sides) by opponent pieces
  - A valid move is one in which at least one piece is reversed
  - If one player cannot make a move, play passes back to the other player
  - When neither player can move, the game ends, the player with the most pieces wins
- Details
  - https://en.wikipedia.org/wiki/Reversi

Reversi Rules

O places piece at E3:

```
 |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|O|_|_|_|
4|_|_|X|X|O|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

- Rules
  - X moves first
  - Pieces are reversed if surrounded (on two sides) by opponent pieces
  - A valid move is one in which at least one piece is reversed
  - If one player cannot make a move, play passes back to the other player
  - When neither player can move, the game ends, the player with the most pieces wins
- Details
  - https://en.wikipedia.org/wiki/Reversi

Programmieren I Winter Challenge

- Aufgabe
  - möglichst guten Reversi-Spieler programmieren
  - 1 sec pro Zug (sonst disqualifiziert, siehe Template)
  - C-Template & Details: WinterChallenge.{zip|pdf} auf Stud.IP
- Abgabe bis 15.1.
  - Quelltext (eine C-Datei)
- Preise
  - Urkunde und Buchpreis für den besten Computer-Reversi-Spieler
- Auswahl
  - Vorrunde: zufällige Paarung, spielen gegeneinander, sammeln Punkte
  - Top 16 (Achtelfinale) spielen gegeneinander beim Reversi-Turnier
  - Termin: letzte Semesterwoche (23.-27.1.), wird noch bekannt gegeben

Das Reversi-Programm darf nicht (z.B. mit einem Server) kommunizieren und darf nur selbst entwickelte Komponenten enthalten (außer der prog1lib und der C-Standardbibliothek).

Two-Player Finite Zero-Sum Strategy Games with Perfect Information

- Two-player: No coalitions possible
- Finite: Finite number of steps till the game is over
- Zero-sum: One player's gain equals another player's loss
- Strategy games: No element of chance beyond the players' moves
- Perfect information: Each player knows the full state of the game

- Basic assumption: Players act in a rational way and aim to maximize their win

- Other examples: Tic-Tac-Toe, Chess, Go

What should a computer player do?

```
 |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|O|_|_|_|
4|_|_|X|X|O|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

- What is the best move given the current game state and the list of possible moves?

- Evaluate quality of game states
- Find good moves: precompute many moves
  - If I do this move, what will my opponent's do?
  - How will can I react?
  - What will the opponent do then?

Estimating the Score (Payoff Value)

- Estimate the quality of the game state
  - Heuristics
- Different possible attributes
  - Number of my pieces minus number of opponent pieces?
    - Score from X's perspective: 4 – 1 = 3
    - Score from O's perspective: 1 – 4 = -3
  - Number of "safe" pieces of a color?
    - A piece in A1 cannot be reversed
  - Evaluate positions on the board, e.g., corners, edges, center
  - Mobility: Number of possible moves for player A or B
  - Number of flipped pieces in a move

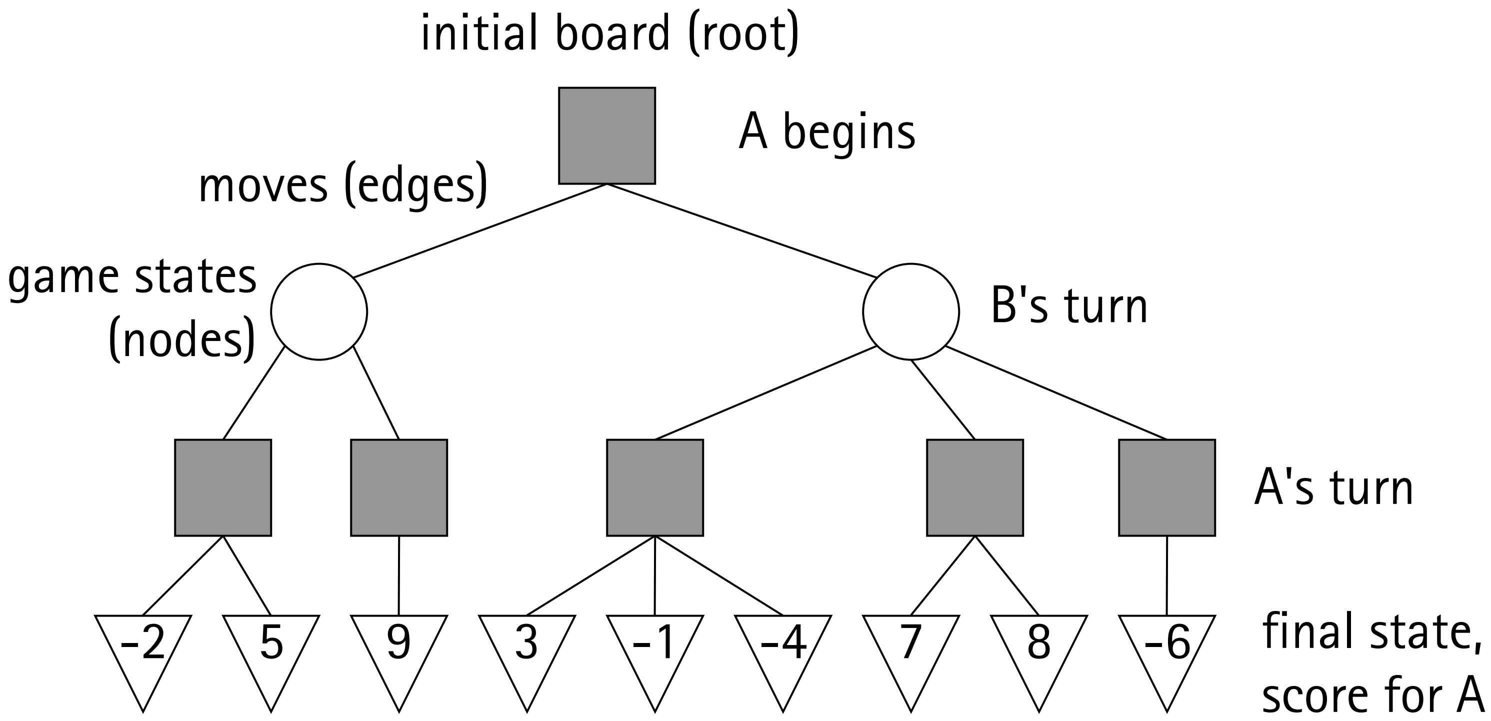
```
 |A|B|C|D|E|F|G|H|
1|_|_|_|_|_|_|_|_|
2|_|_|_|_|_|_|_|_|
3|_|_|_|_|_|_|_|_|
4|_|_|X|X|X|_|_|_|
5|_|_|_|X|O|_|_|_|
6|_|_|_|_|_|_|_|_|
7|_|_|_|_|_|_|_|_|
8|_|_|_|_|_|_|_|_|
```

Precomputing Moves

- Board games are simple, small worlds with clear rules
- Consequences of game moves can be precomputed
  - Precompute many moves and evaluate each one, choose "best" one
  - Little "intelligence", much computing power
  - Search space typically too large to compute every state
- Procedure
  - Given the current game state (e.g., pieces on the board)
  - Determine the set of possible moves for a player A
  - Estimate the benefits of each move (precompute, estimate)
  - Do the move with the best estimate for player A
    (= greatest loss for player B)

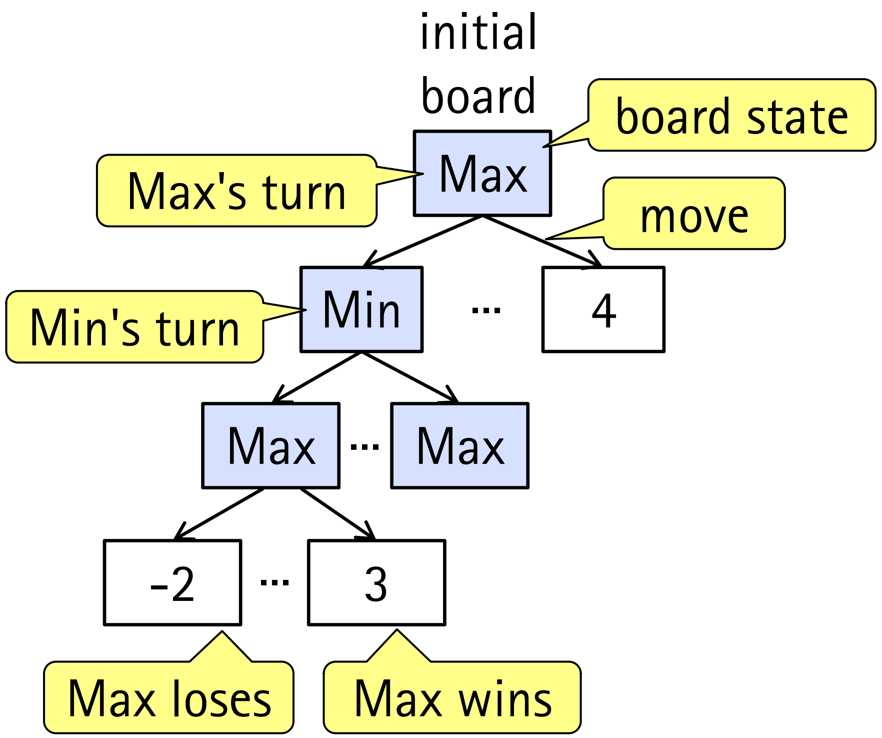
Game Trees

- Description of possible game states and moves
- For the analysis of games

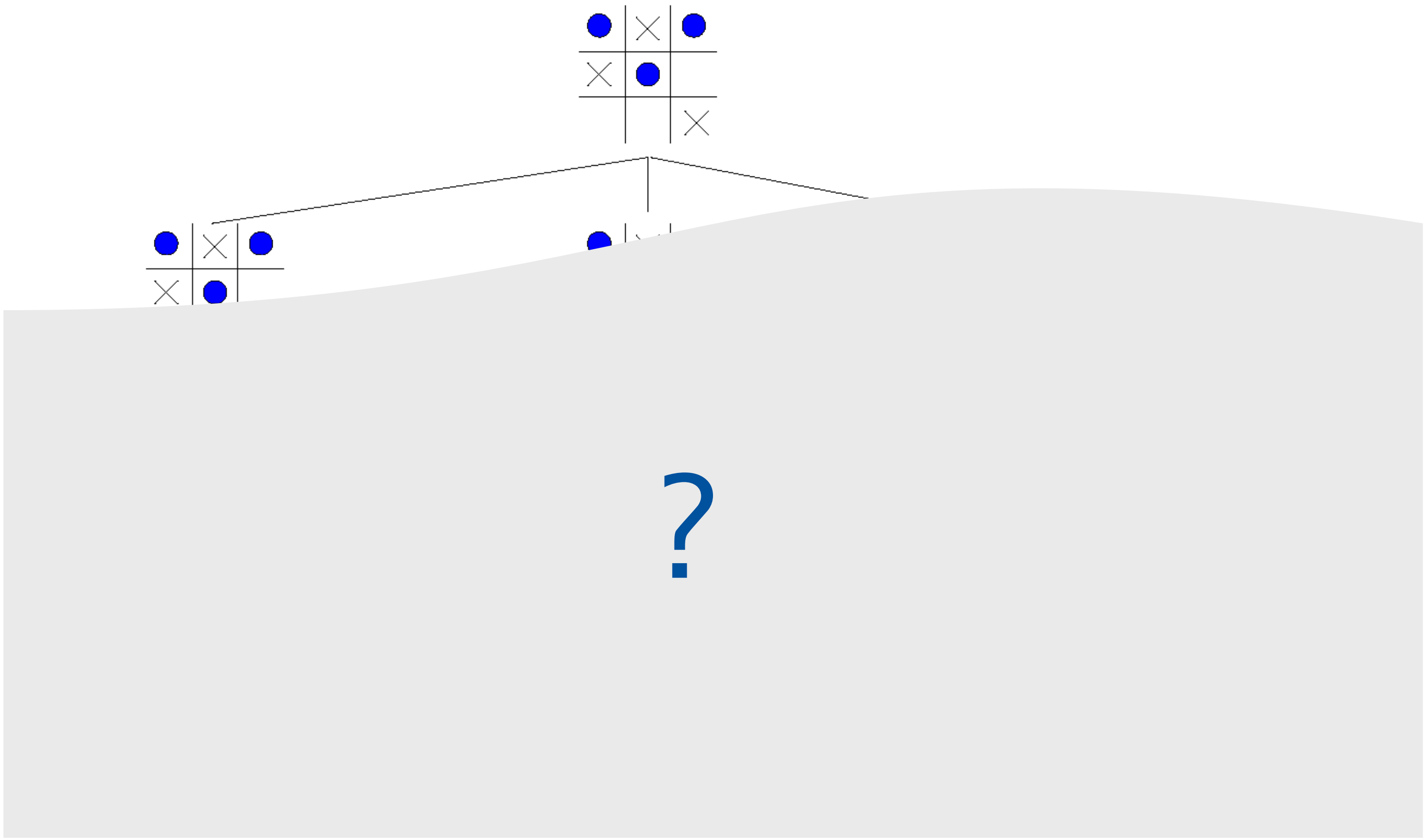


Game Trees

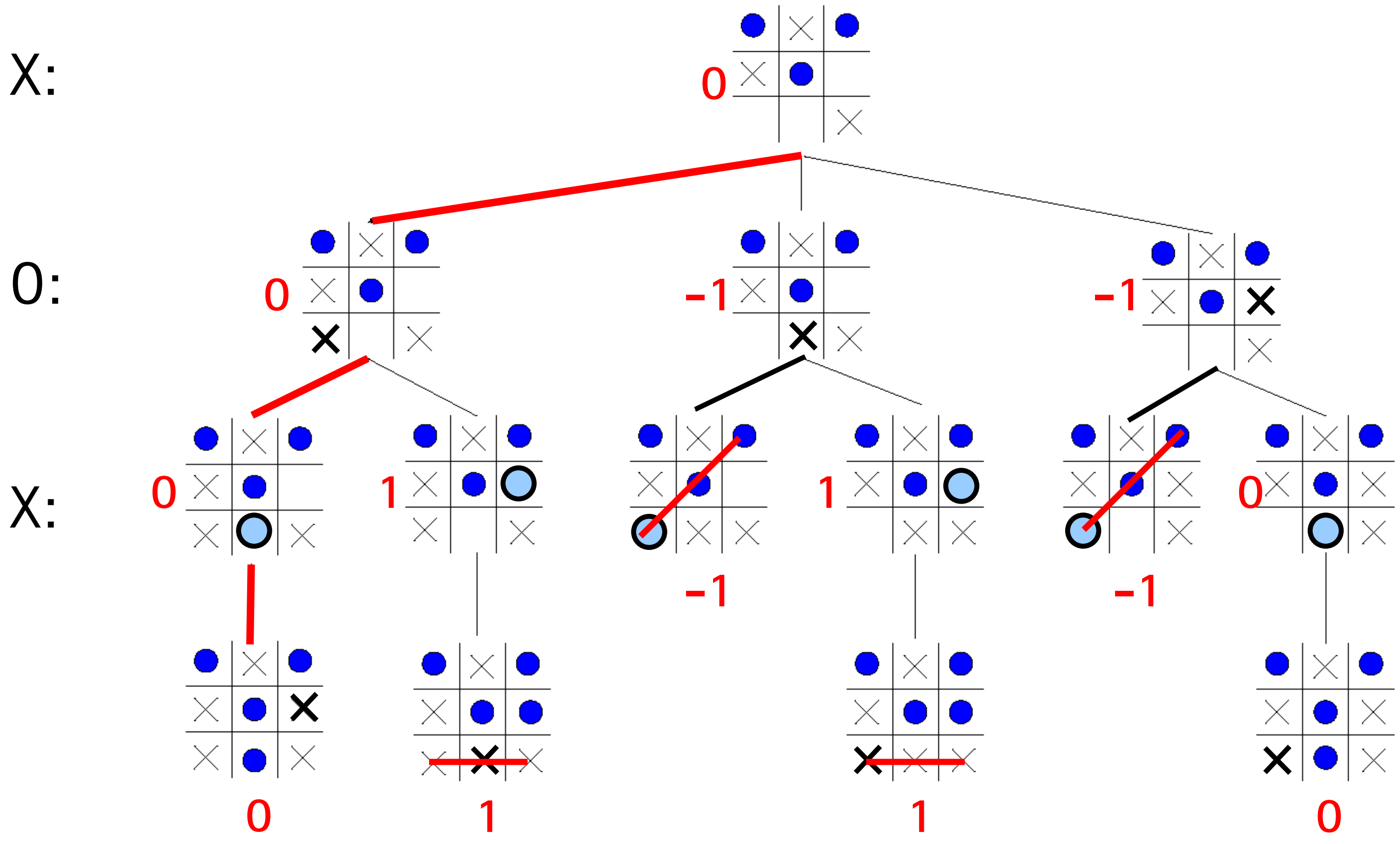
- Nodes represent possible states of the game board
- Edges represent possible moves
- Player "Max" and "Min" take turns
  - Convention: Max begins
- Leaves denote end situations
  - Contain score for Max
  - Max tries to reach a leaf with a high value
  - Min tries to reach a leaf with a low value
- Real game trees are very large
  - Chess: ~10^{120} nodes, Go: ~10^{761} nodes



Game Tree for Tic-Tac-Toe



Game Tree for Tic-Tac-Toe

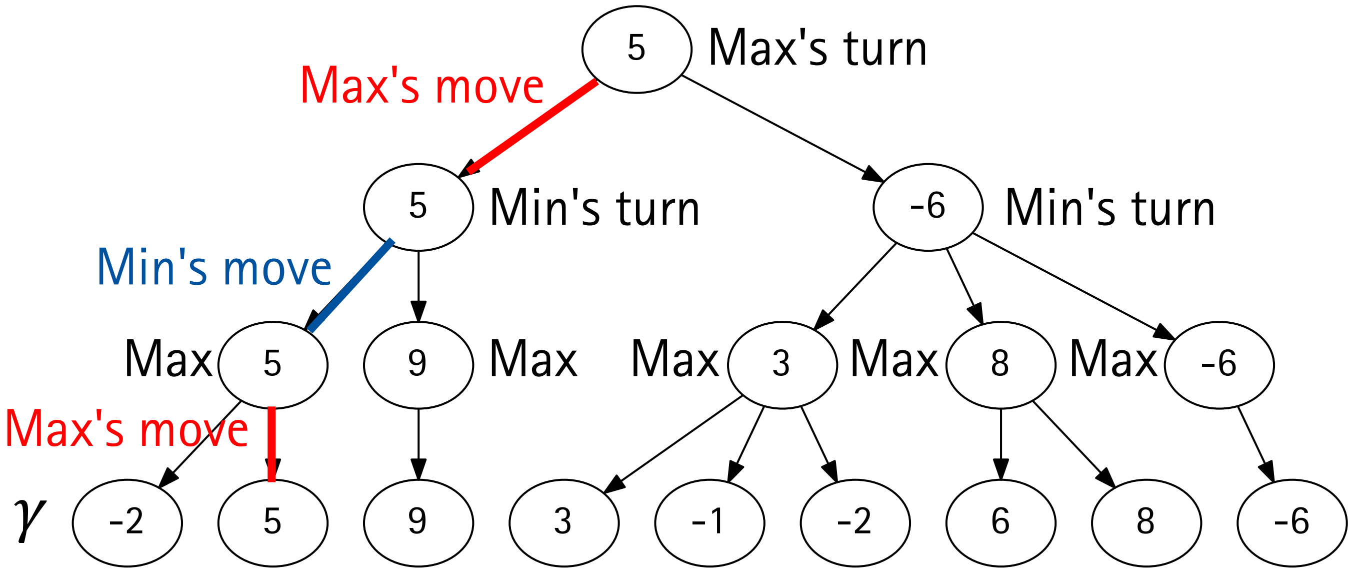


Minimax Algorithm

- Let γ(leaf) be Max's payoff function (scores for leaves)
- The minimax value v(n) of a node n is defined as:

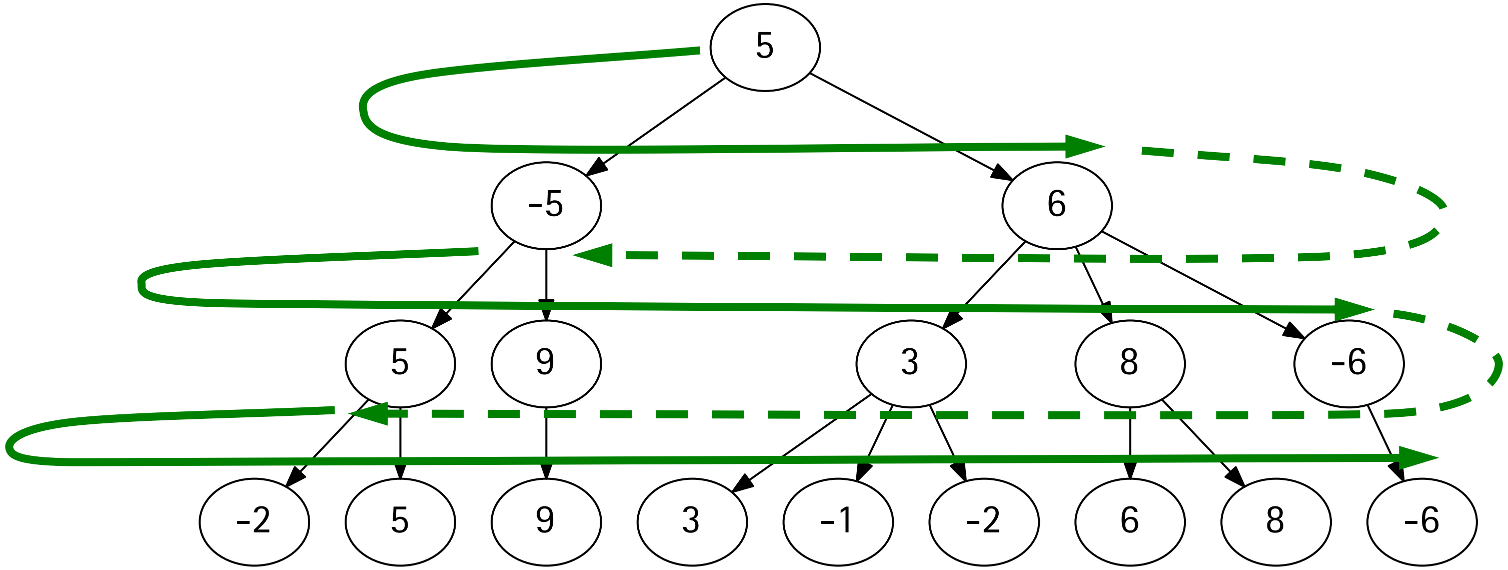
$$v(n) = \begin{cases} \gamma(n), \text{ if n is a leaf} \\ \max \{v(m) \mid \text{m is child of n}\}, \text{ if n is an inner Max node} \\ \min \{v(m) \mid \text{m is child of n}\}, \text{ if n is an inner Min node} \end{cases}$$

- Recursive definition
  → recursive algorithm



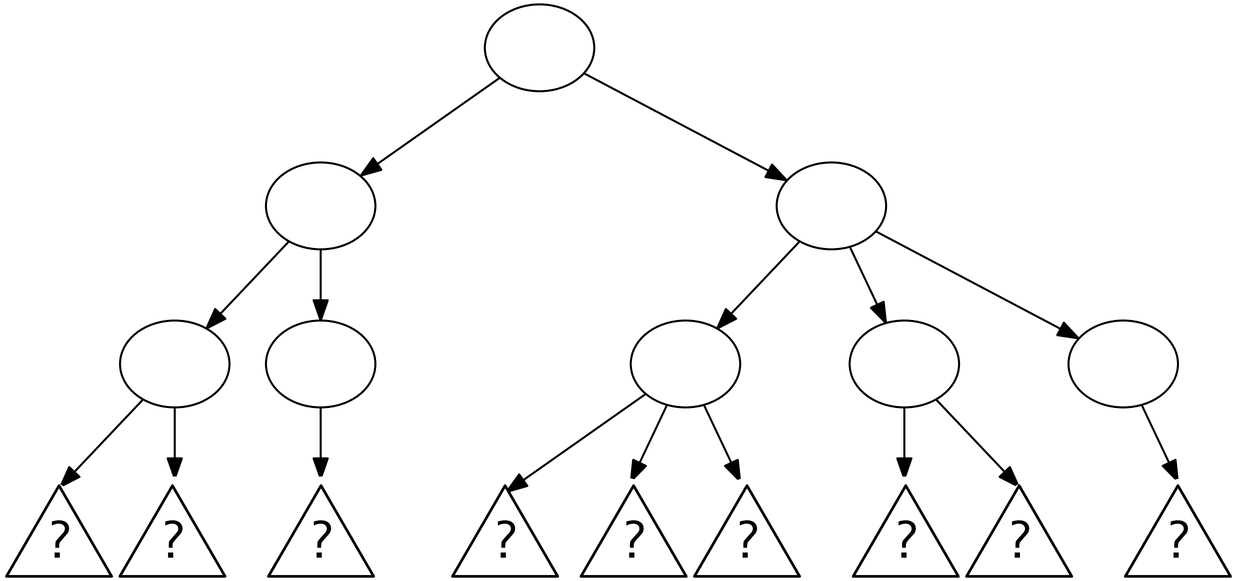
Building and Evaluating Game Trees

- Problem: Game trees are too large to build them completely
- Solution: Build them to level d, then evaluate
  - Maximum level d may depend on available time
  - Problem: Memory limitations for storing the tree levels
- Alternative: Only expand the most promising nodes



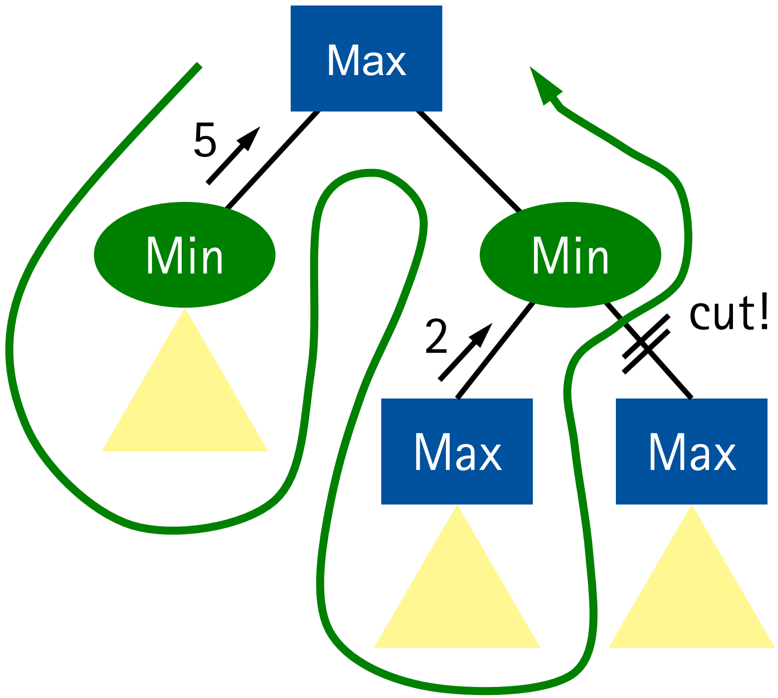
Building and Evaluating Game Trees

- Problem: Real scores are not known (if stopping at level d)
- Workaround: Estimate score from board state
- Estimation function: "Good" (close to real scores) and efficient



Tree Cuts

- May cut branches if they are worse than what has already been found
  - Cut means: Do not explore further



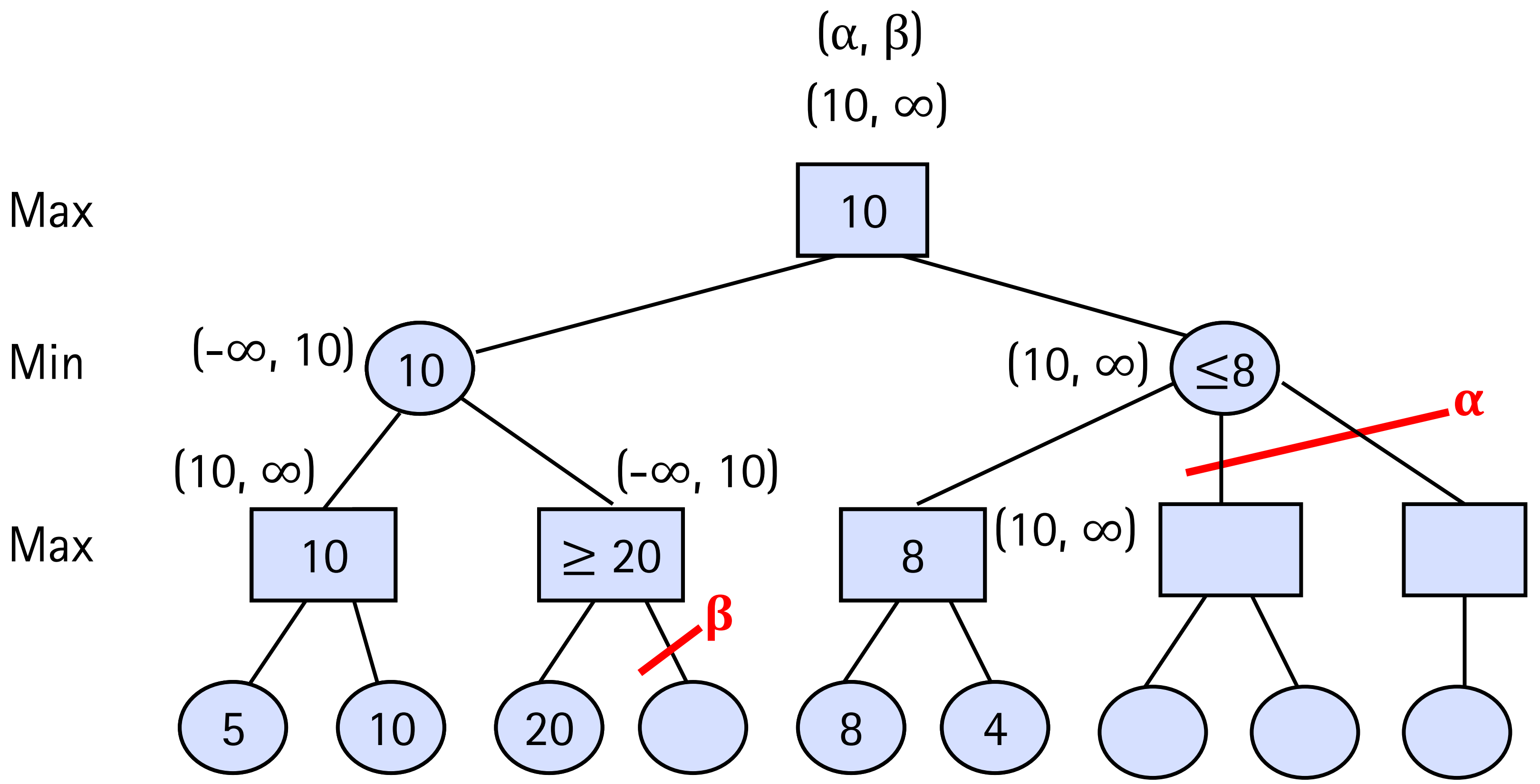
α–β Pruning

- Expand nodes only if they can possibly change the result
  - Yields the same moves as the original Minimax algorithm
  - Does not explore irrelevant branches of the game tree
- α-bound: The minimum (worst) score that Max can get, given the current exploration state of the game tree
  - α-bound (initially $-\infty$) increases during search of the game tree
- β-bound: The maximum (best) score that Max can get, given the current exploration state of the game tree
  - β-bound (initially ∞) decreases during search of the game tree
- Perform cut if $\alpha \geq \beta$

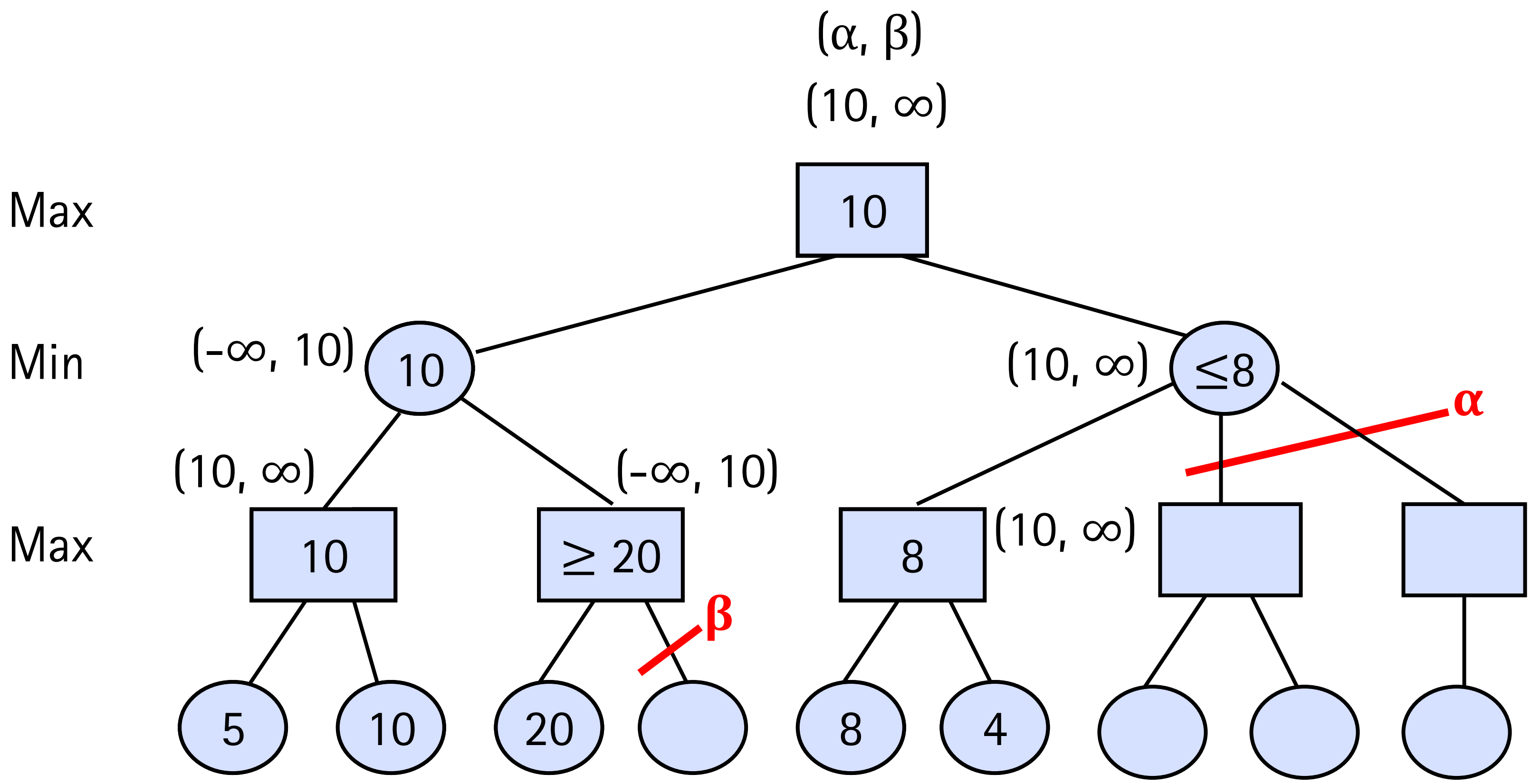
α–β Pruning Algorithm

```
int max_value(Node *g, int α, int β) {
    if (limit_reached(g)) return eval(g);
    for (Node *n = g->first; n != NULL; n = n->next) {
        α = max(α, min_value(n, α, β));
        if (α ≥ β) return β; // β cut
    }
    return α;
}
int min_value(Node *g, int α, int β) {
    if (limit_reached(g)) return eval(g);
    for (Node *n = g->first; n != NULL; n = n->next) {
        β = min(β, max_value(n, α, β));
        if (β ≤ α) return α; // α cut
    }
    return β;
}
```

α–β Pruning Example

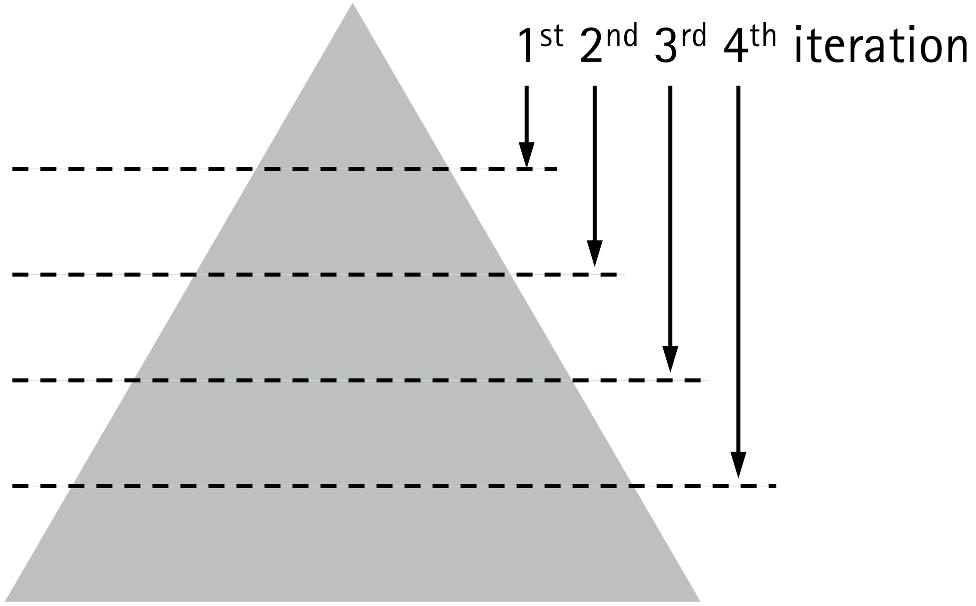


α–β Pruning Example



Iterative Deepening

- Build and evaluate game tree to level d, evaluate
- If still sufficient time, build and evaluate to level d+1, evaluate
- Evaluation to level d is cheap compared to evaluation to level d+1
- Memorize information from earlier iterations
    - E.g., use results of earlier iterations for reordering



Other Aspects

- Time management
  - Best use of the available time per move (and in total)
- Learning games
  - Can a game learn if it plays against itself?