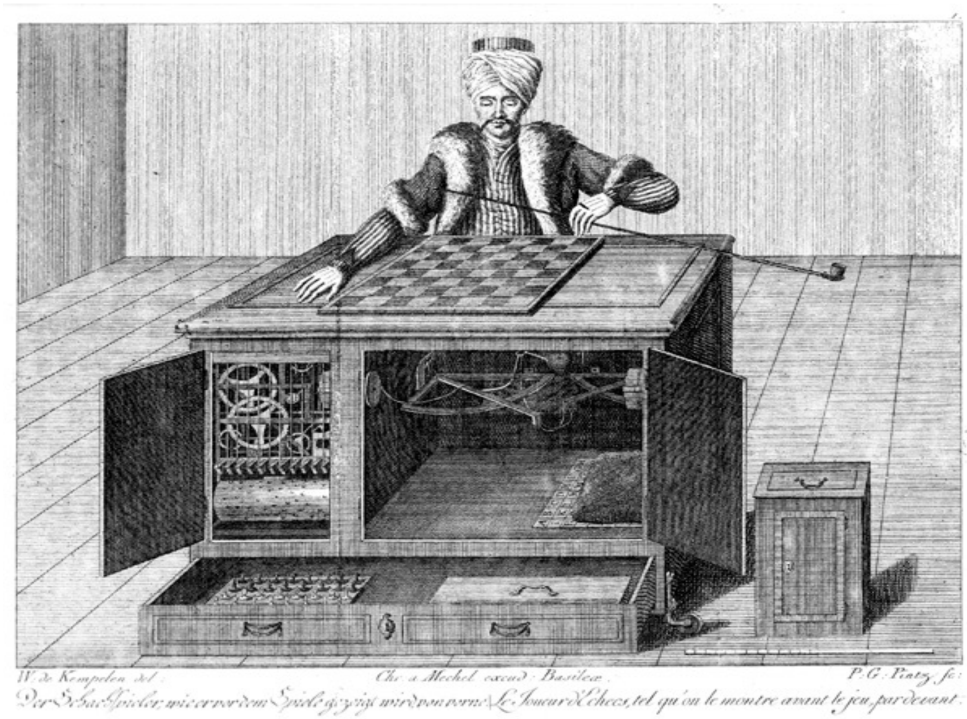
CS440/ECE448 Lecture 29: Two-Player Games

Mark Hasegawa-Johnson



By Karl Gottlieb von Windisch - Copper engraving from the book: Karl Gottlieb von Windisch, Briefe über den Schachspieler des Hrn. von Kempelen, nebst drei Kupferstichen die diese berühmte Maschine vorstellen. 1783.Original Uploader was Schaelss (talk) at 11:12, 7. Apr 2004., Public Domain, https://commons.wikimedia.org/w/index.php?curid=424092

Outline

- Alternating two-player zero-sum games
- Minimax search
- Evaluation functions
- Alpha-beta search
- Computational complexity of alpha-beta

Games vs. single-agent search

- We don't know how the opponent will act
- The solution is not a fixed sequence of actions from start state to goal state, but a ***strategy*** or ***policy***

- Definition of **<u>deterministic policy</u>** (today): a function $\pi: \mathcal{S} \rightarrow \mathcal{A}$ that maps from world states, $\mathcal{S}$, to actions, $\mathcal{A}$.
- Definition of **<u>stochastic policy</u>**: a function $\pi: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$ that maps from (state,action) pairs to probabilities.

Alternating two-player zero-sum games

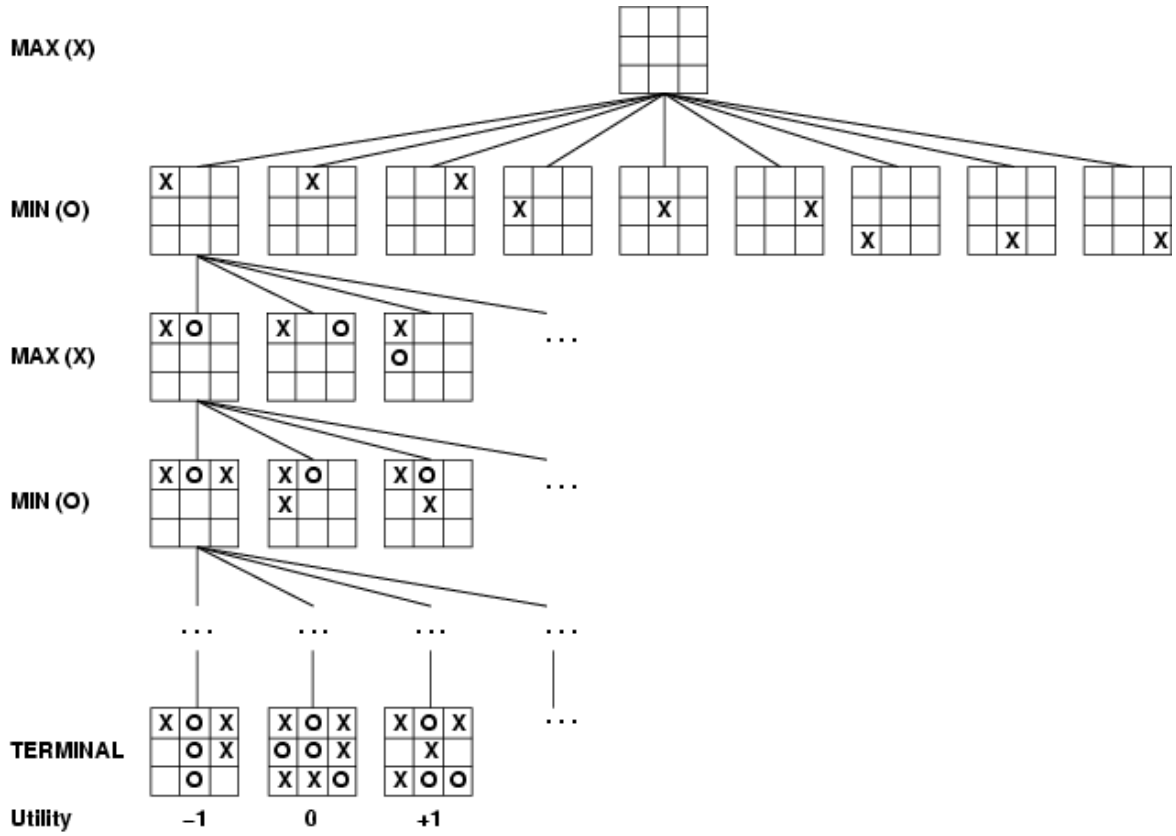
- Players take turns
- Each game outcome or **terminal state** has a **utility** for each player (e.g., 1 for win, 0 for tie, -1 for loss)
- The sum of both players' utilities is a constant, e.g.,

$$\text{Utility(player 0)} + \text{Utility(player 1)} = 0$$

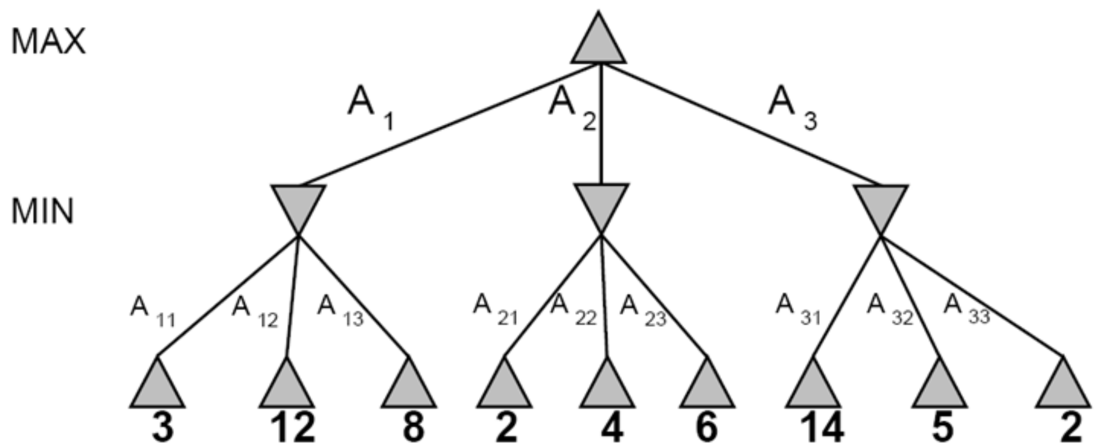
- Player 0 tries to maximize Utility(player 0). Let's call this player "Max"
- Player 1 tries to minimize Utility(player 0). Let's call this player "Min"

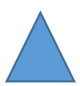
Game tree

A game of tic-tac-toe between two players, “max” and “min”



A more abstract game tree



 = game state from which MAX can play

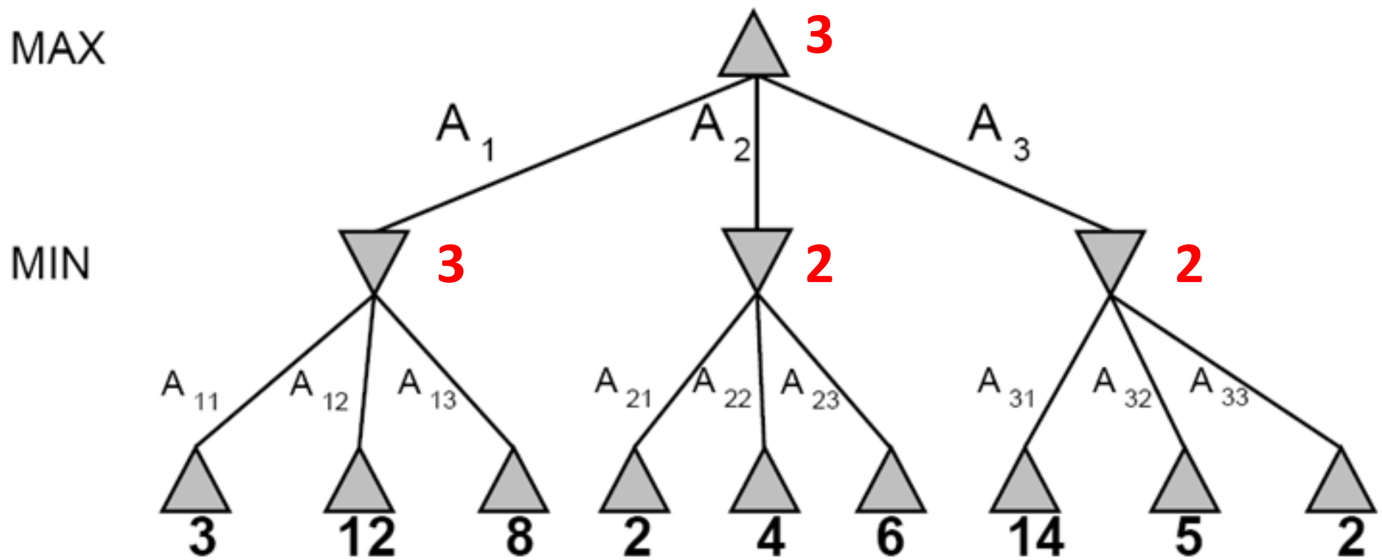
 = game state from which MIN can play

number = value of that game state for MAX

Outline

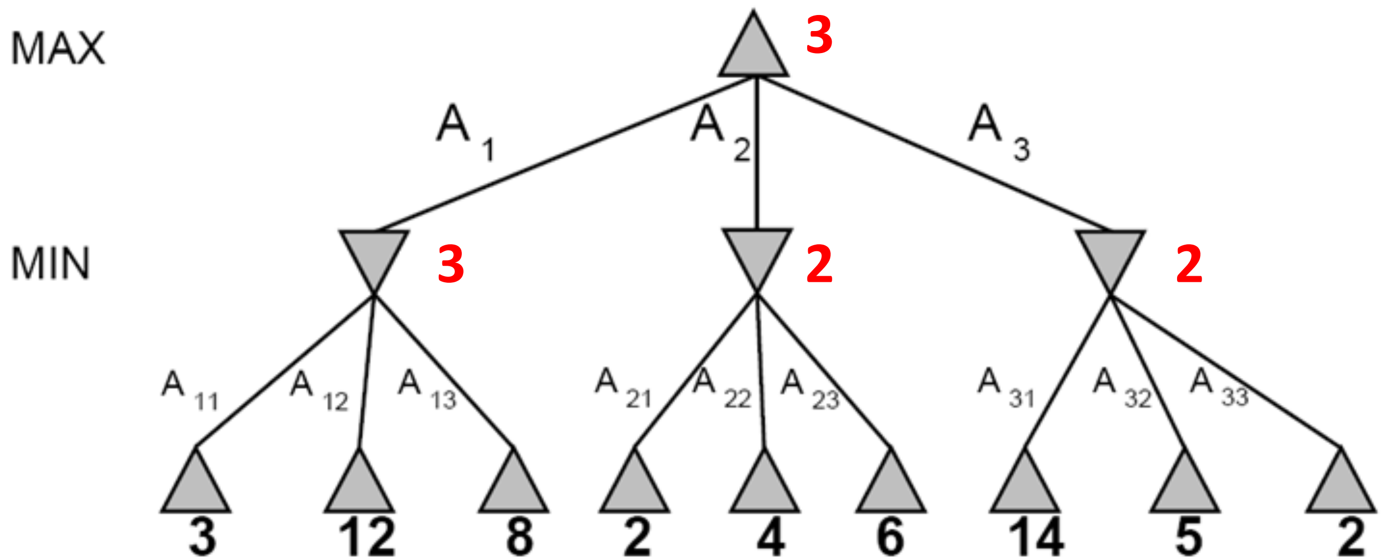
- Alternating two-player zero-sum games
- Minimax search
- Evaluation functions
- Alpha-beta search
- Computational complexity of alpha-beta

Game tree search



- **Minimax value of a node**: the utility (for MAX) of being in the corresponding state, assuming perfect play on both sides
- **Minimax strategy:** Choose the move that gives the best worst-case payoff

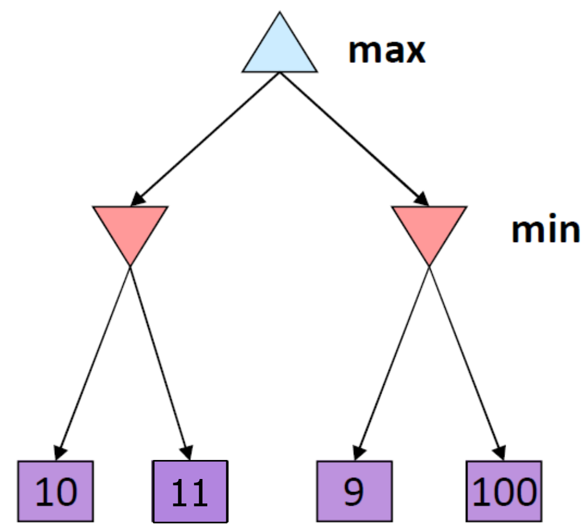
Computing the minimax value of a node



- **Minimax**(*node*) =
  - Utility(*node*) if *node* is terminal
  - $\max_{action}$ **Minimax**(Succ(*node, action*)) if *player* = MAX
  - $\min_{action}$ **Minimax**(Succ(*node, action*)) if *player* = MIN

Optimality of minimax

- The minimax strategy is optimal against an optimal opponent
- What if your opponent is suboptimal?
- If you play using the <u>**minimax-optimal**</u> sequence of moves, then the utility you earn will always be <u>**greater than or equal**</u> to the amount that you predict.

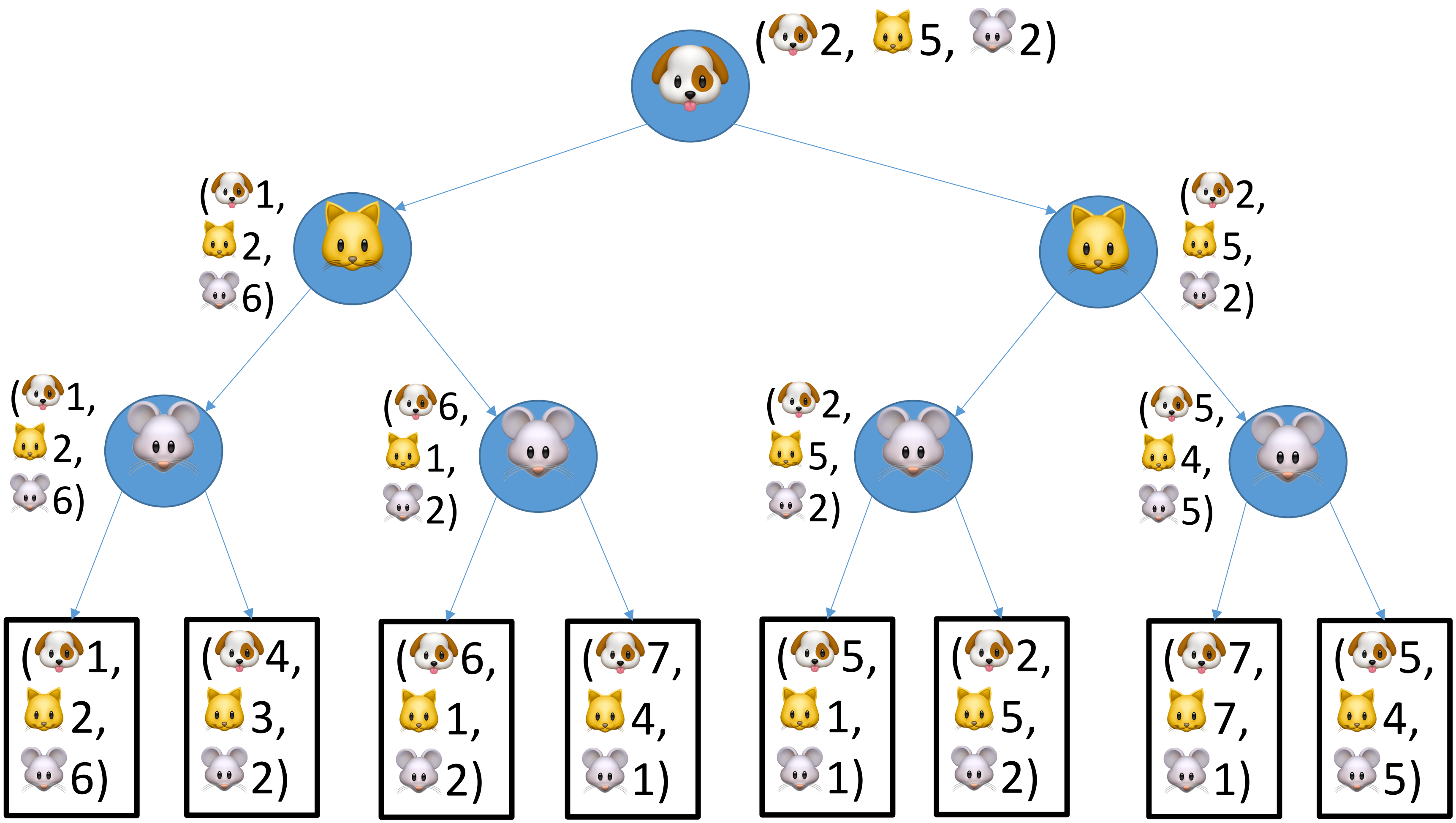


Example from D. Klein and P. Abbeel

Multi-player games; Non-zero-sum games

- More than two players. For example:
  - Dog (🐶) tries to maximize the number of doggie treats
  - Cat (🐱) tries to maximize the number of cat treats
  - Mouse (🐭) tries to maximize the number of mouse treats
- Non-zero-sum. We can't just assume that Min's score is the opposite of Max's. Instead, utilities are now tuples. For example:
  - (🐶5, 🐱8, 🐭2) = 5 doggie treats, 8 kitty treats, 2 mouse treats
- Each player maximizes their own utility at their node

Minimax in multi-player & non-zero-sum games



Outline

- Alternating two-player zero-sum games
- Minimax search
- Evaluation functions
- Alpha-beta search
- Computational complexity of alpha-beta

Limited-Horizon Search: limited computation

In a practical game, we compute minimax to a limited depth, because we have limited computational ability

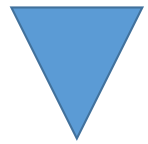
- Depth=1: evaluate every possible current move, look at the resulting game state, decide which resulting game state looks the best, and take that action.
  - Computational complexity to choose your next move: $\mathcal{O}\{b\}$, if there are b possible moves.
- Depth=2: evaluate every possible current move, and every move that your opponent might make in response, and then look at resulting game states.
  - Computational complexity to choose your next move: $\mathcal{O}\{b^2\}$.
- Depth=3: evaluate every possible sequence of three moves (mine, my opponent's, then mine), and look at the resulting game states.
  - Computational complexity to choose your next move: $\mathcal{O}\{b^3\}$.
- … Depth=∞? No way!!

Evaluation functions

It's impossible to search all the way to the end of the game. At some fixed depth, we need to stop and estimate the value of s using some cheap but reasonably accurate estimate of $v(s)$. It should have the following properties:

- $v(s)$ should be a reasonable estimate of the outcome of the game, but
- It must be possible to compute $v(s)$ quickly, i.e., typically we desire polynomial complexity.

Example: Depth 1 search, Chess



In chess, traditionally, the black player is MIN.

What move should MIN choose, from this board position?

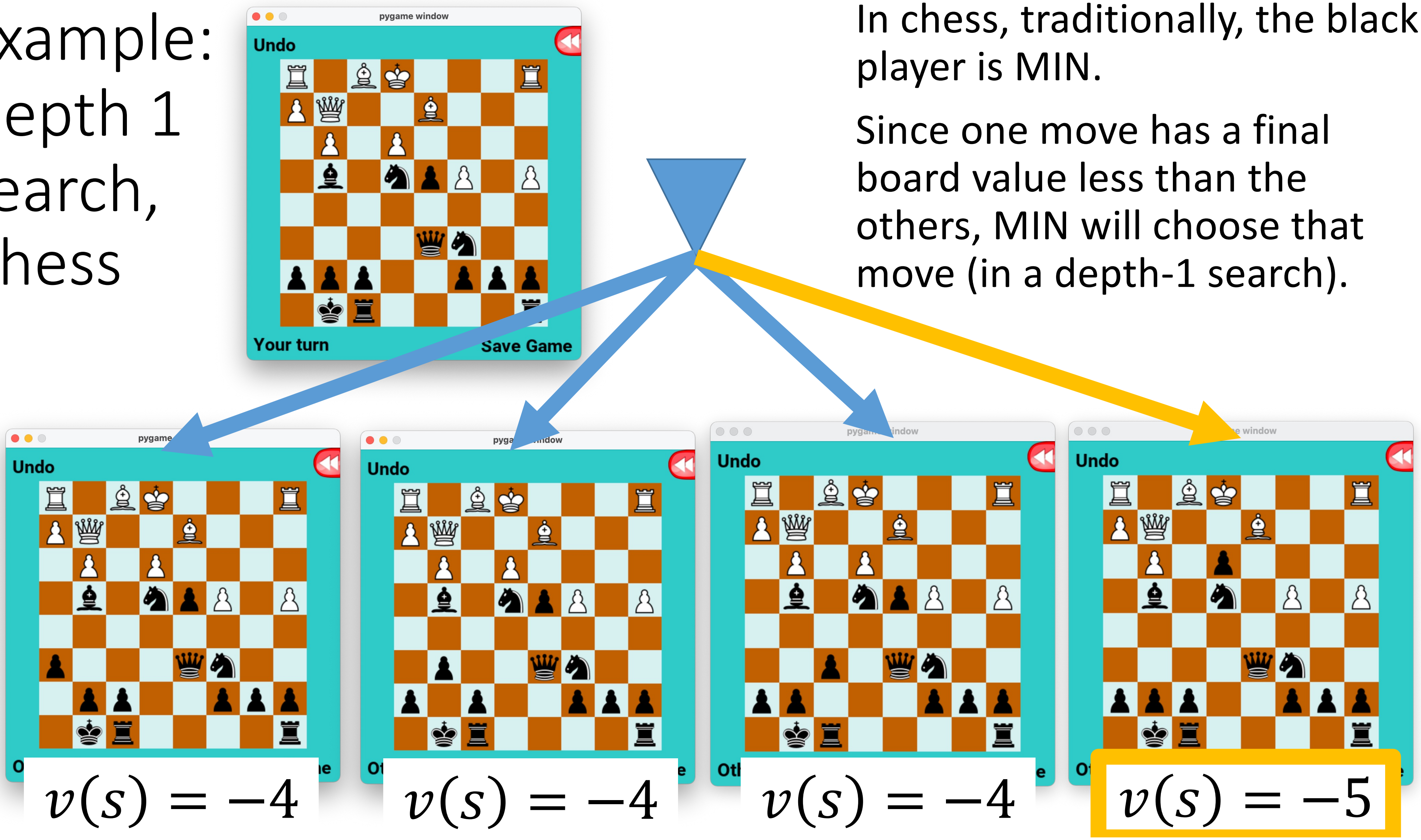
Graphics: created by the PyChess community.

Game board shown: game1.txt from the MP5 distribution.

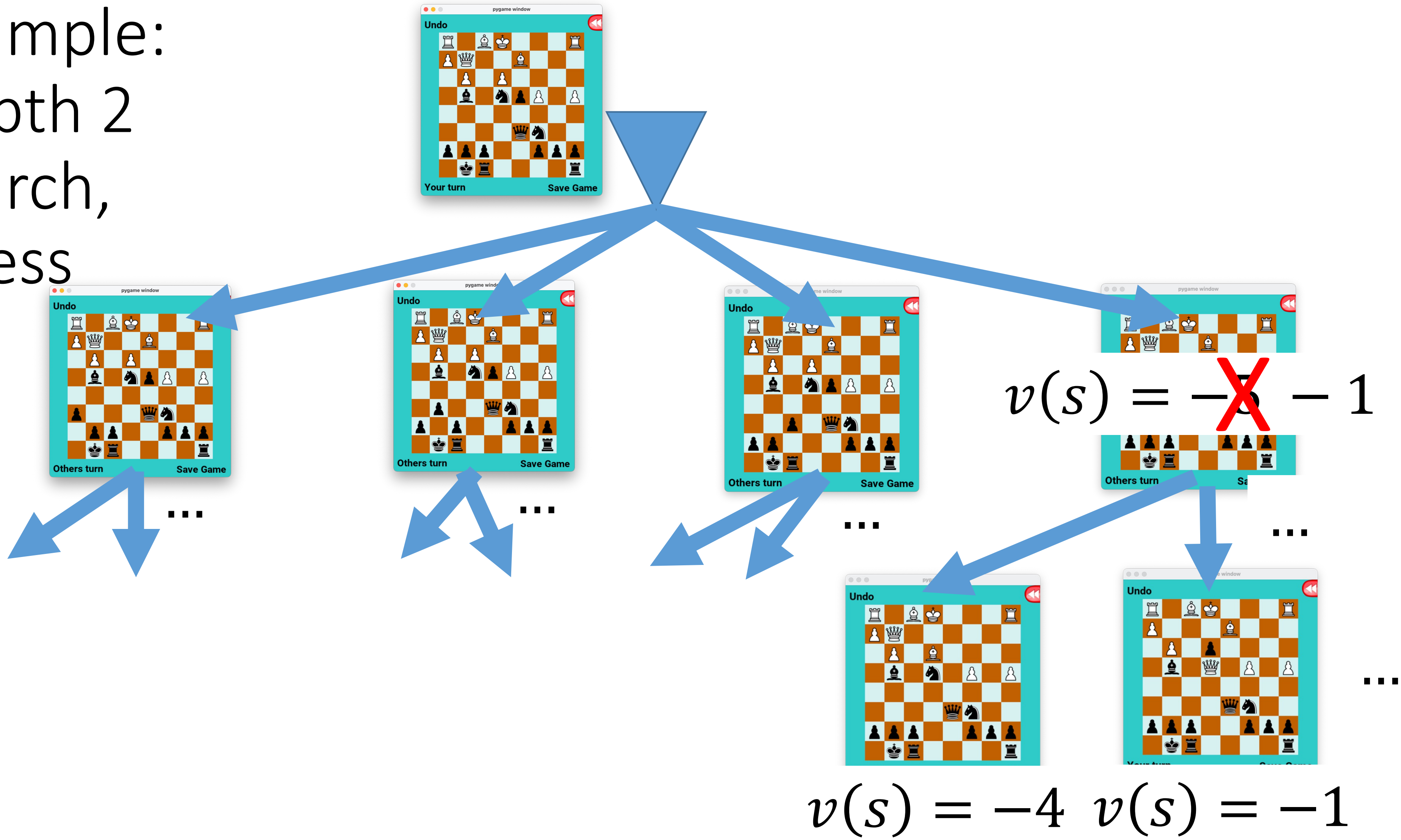
Example: Depth 1 search, Chess

In chess, traditionally, the black player is MIN.

Since one move has a final board value less than the others, MIN will choose that move (in a depth-1 search).

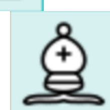
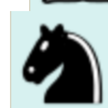
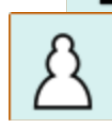


Example: Depth 2 search, Chess



Typical chess evaluation function

Each side receives:

- 9 points per remaining queen 
- 5 points per remaining rook 
- 3 points per remaining bishop 
- 3 points per remaining knight 
- 1 point per remaining pawn 

$v(s) =$ points for white - points for black

The PyChess evaluation function provides extra point depending on the location of each piece on the board.

Evaluation functions in general

Evaluation function must be reasonably accurate, but computationally simple. Often this means a linear evaluation function:

$$v(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots$$

- $f_1(s), f_2(s), \ldots$ are features of the game state s
- $w_1, w_2 \ldots$ are real-valued weights.

Notice: this is just a one-layer neural net, with input vector $f(s) = [f_1(s), f_2(s), \ldots]$ and weight vector $w = [w_1, w_2, \ldots]$.

Recently, deeper neural nets are also sometimes used.

Cutting off search

- **Horizon effect:** you may incorrectly estimate the value of a state by overlooking an event that is just beyond the depth limit
  - For example, a damaging move by the opponent that can be delayed but not avoided
- Remedies: search a small number of possible extensions to depth+1.
  - **Quiescence search:** extend only “unstable” moves, e.g., moves that capture a piece.
  - **Singular extension:** extend only very strong moves.
  - **Stochastic search**: randomly sample a small number of possible future paths.

Outline

- Alternating two-player zero-sum games
- Minimax search
- Evaluation functions
- Alpha-beta search
- Computational complexity of alpha-beta

Computational complexity of minimax

- Suppose that, at each game state, there are b possible moves
- Suppose we search to a depth of d
- Then the computational complexity is $O\{b^d\}$!

Basic idea of alpha-beta pruning

- Computational complexity of minimax is $O\{b^d\}$
- There is no known algorithm to make it polynomial time
- But… can we reduce the exponent? For example, could we make the complexity $O\{b^{d/2}\}$?
- If we could do that, then it would become possible to search twice as far, using the same amount of computation. This could be the difference between a beginner chess player vs. a grand master.

Basic idea of alpha-beta pruning

- The basic idea of alpha-beta pruning is to reduce the complexity of minimax from $O\{b^d\}$ to $O\{b^{d/2}\}$.
- We can do this by only evaluating half of the levels.
- How can we ”only evaluate half the levels” without losing accuracy?
- Why it works: It is possible to compute the exact minimax decision without expanding every node in the game tree

The pruning thresholds, alpha and beta

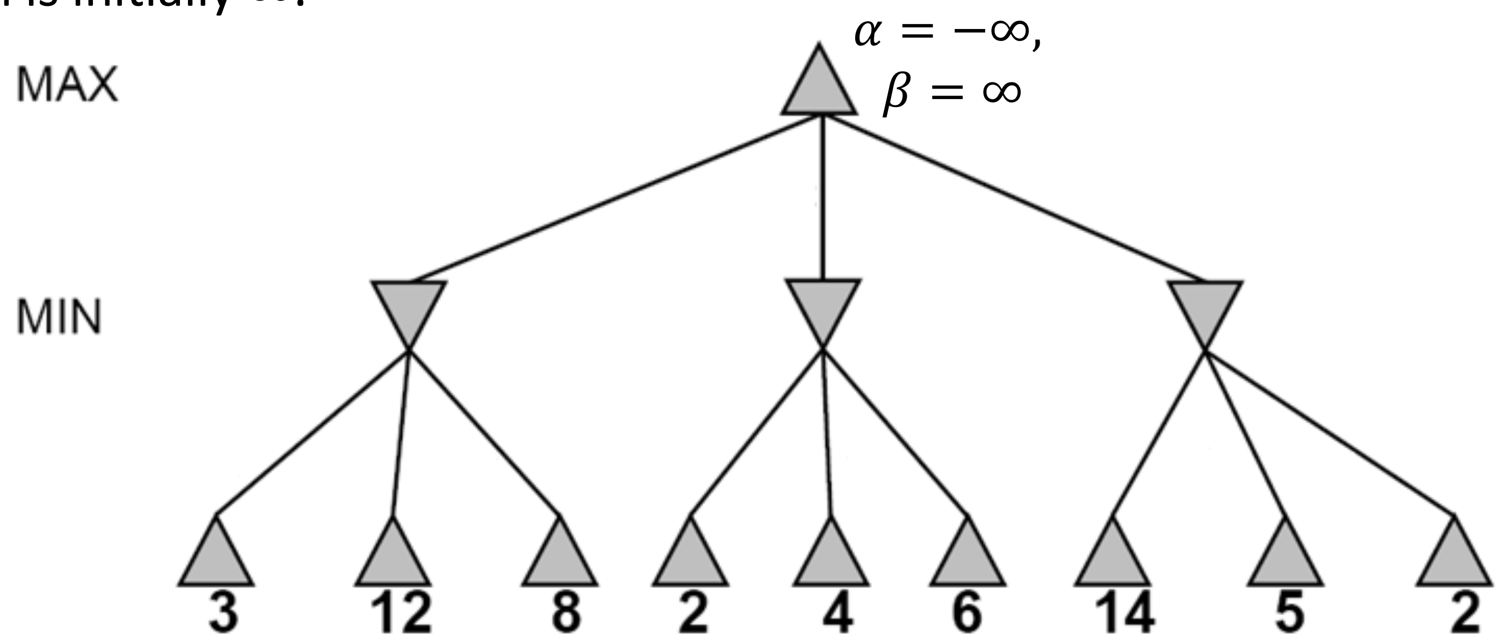
Alpha-beta pruning requires us to keep track of two pruning thresholds, alpha and beta.

- alpha (α) is the highest score that MAX knows how to force MIN to accept.
- beta (β) is the lowest score that MIN knows how to force MAX to accept.
- $\alpha \leq \beta$

Alpha-beta pruning

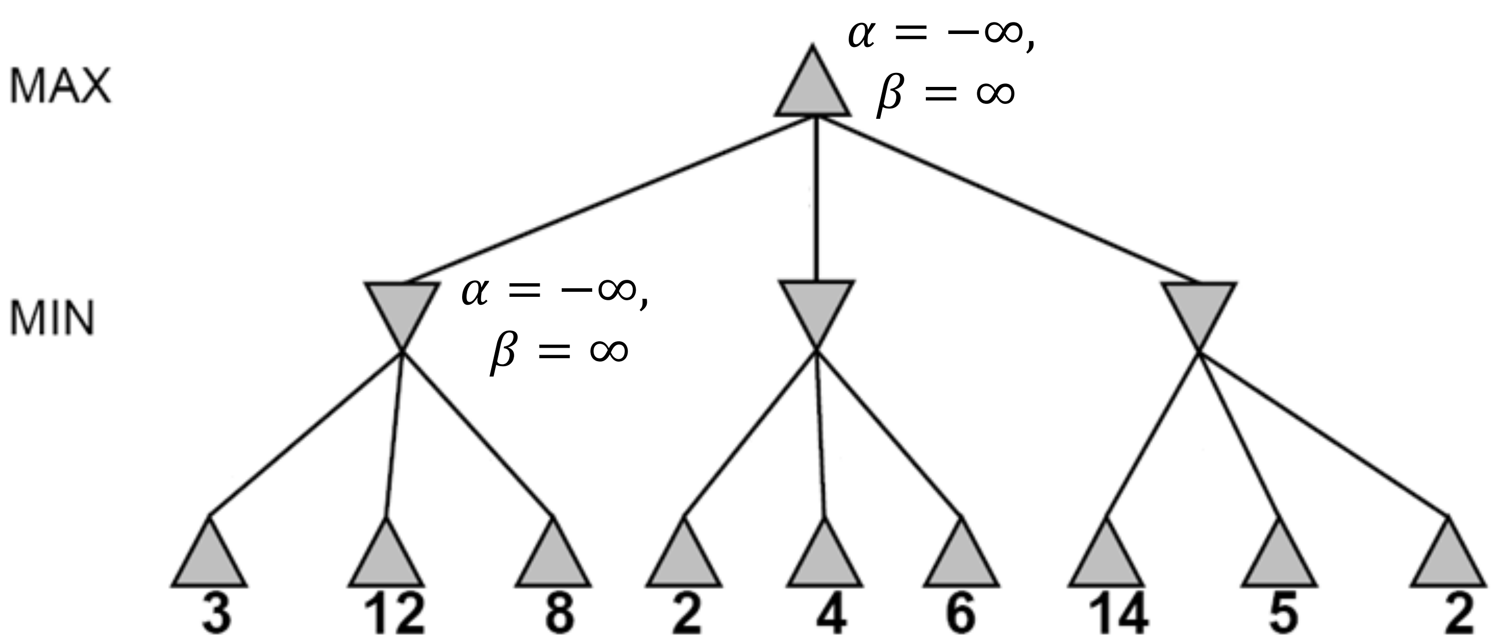
Initialize:

- alpha (α) is the highest score that MAX knows how to force MIN to accept, which is initially $-\infty$.
- beta (β) is the lowest score that MIN knows how to force MAX to accept, which is initially ∞.



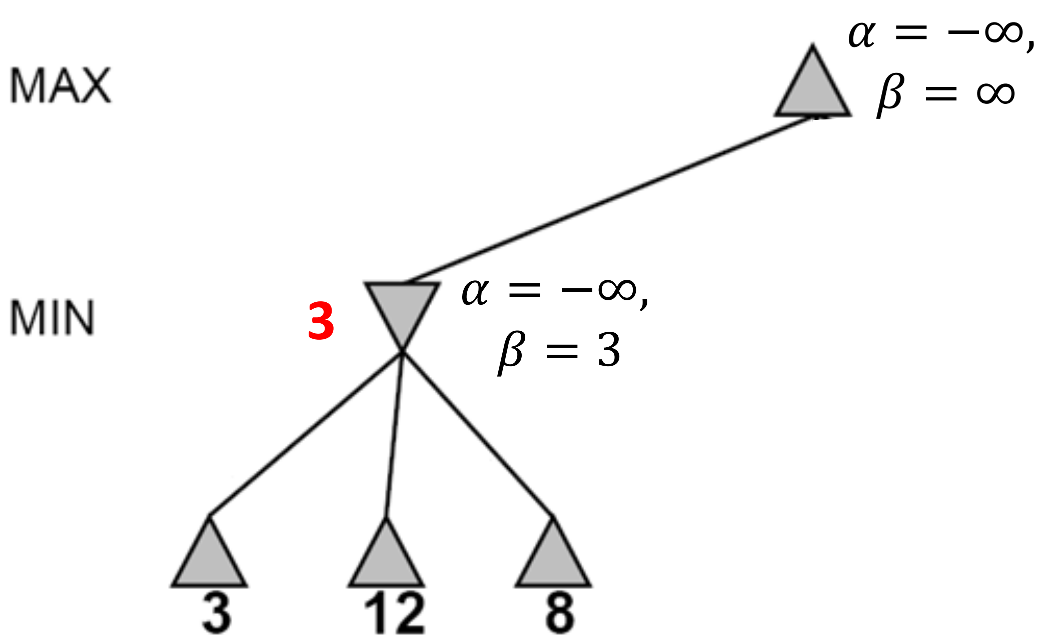
Alpha-beta pruning

Inheritance: Child inherits alpha and beta from its parent



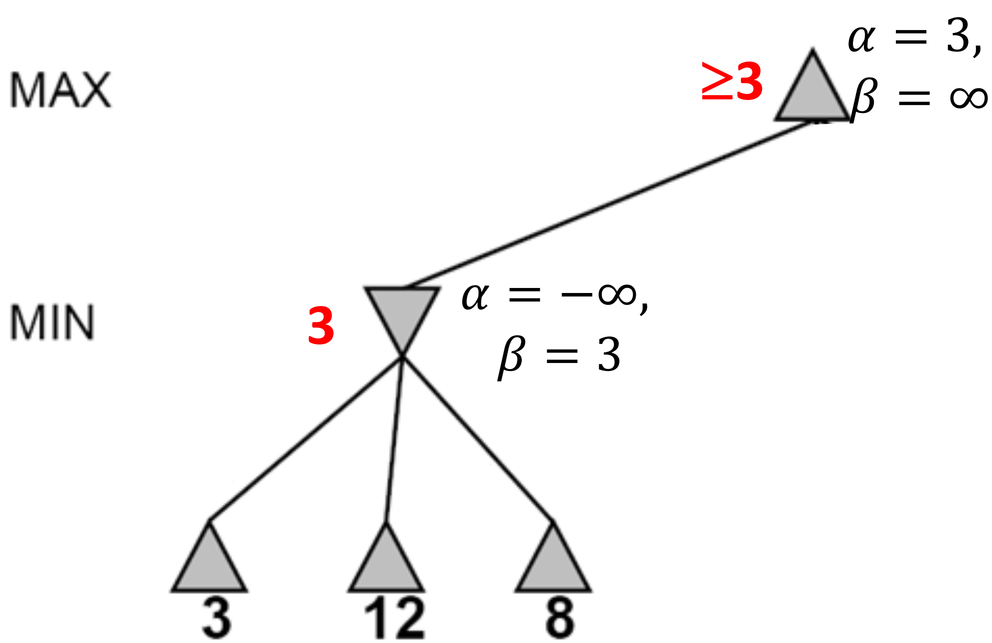
Alpha-beta pruning

Update: a **min** node can update beta. A max node can update alpha.



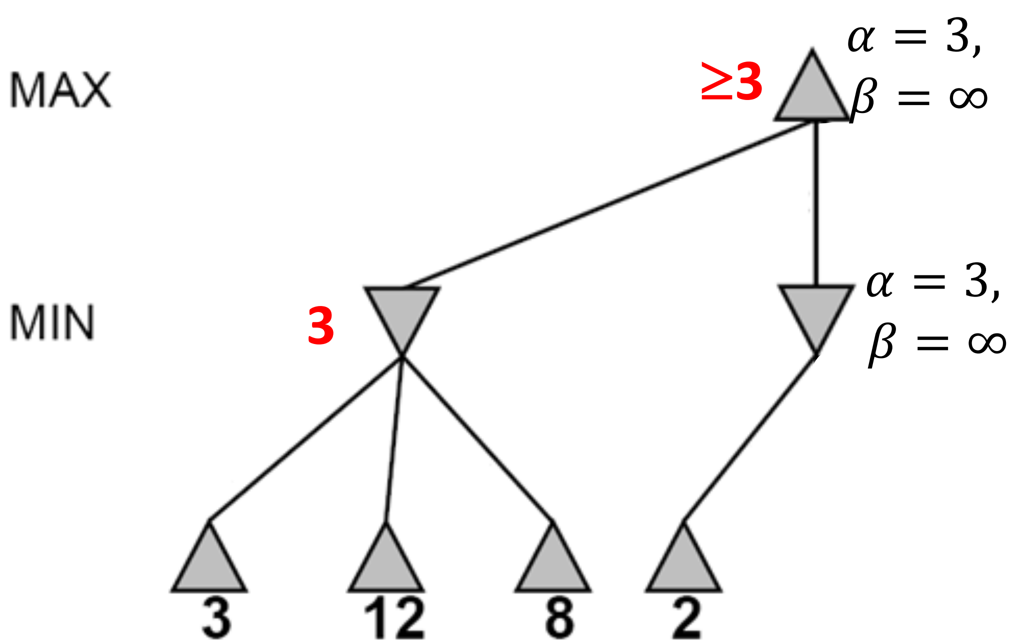
Alpha-beta pruning

Update: a min node can update beta. A **max** node can update alpha.



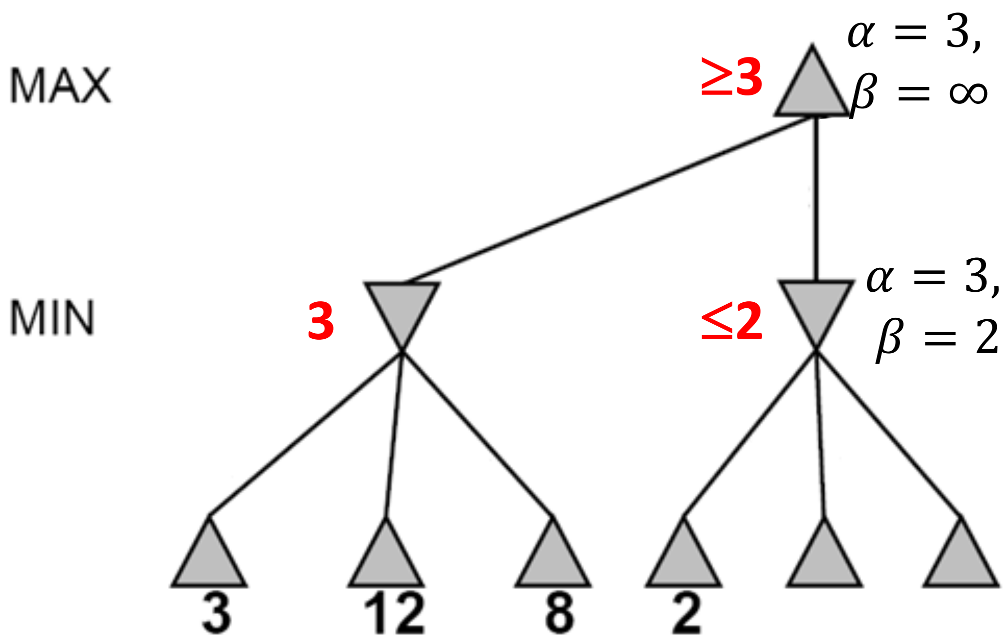
Alpha-beta pruning

Inheritance: Child inherits alpha and beta from its parent



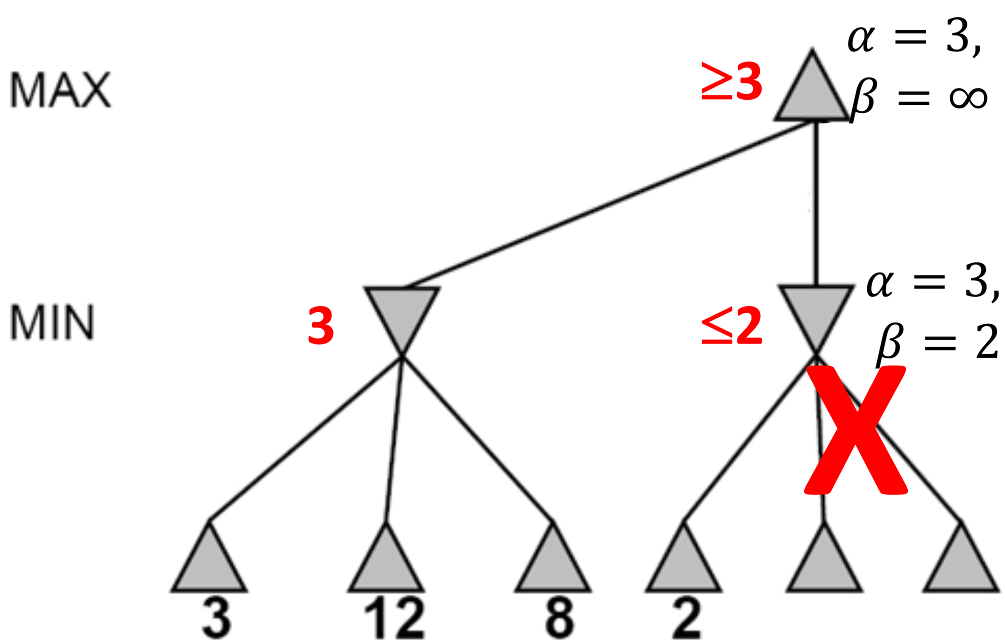
Alpha-beta pruning

Update: a min node can update beta. A max node can update alpha.



Alpha-beta pruning

Pruning: If beta ever falls below alpha, prune any remaining children, and return.

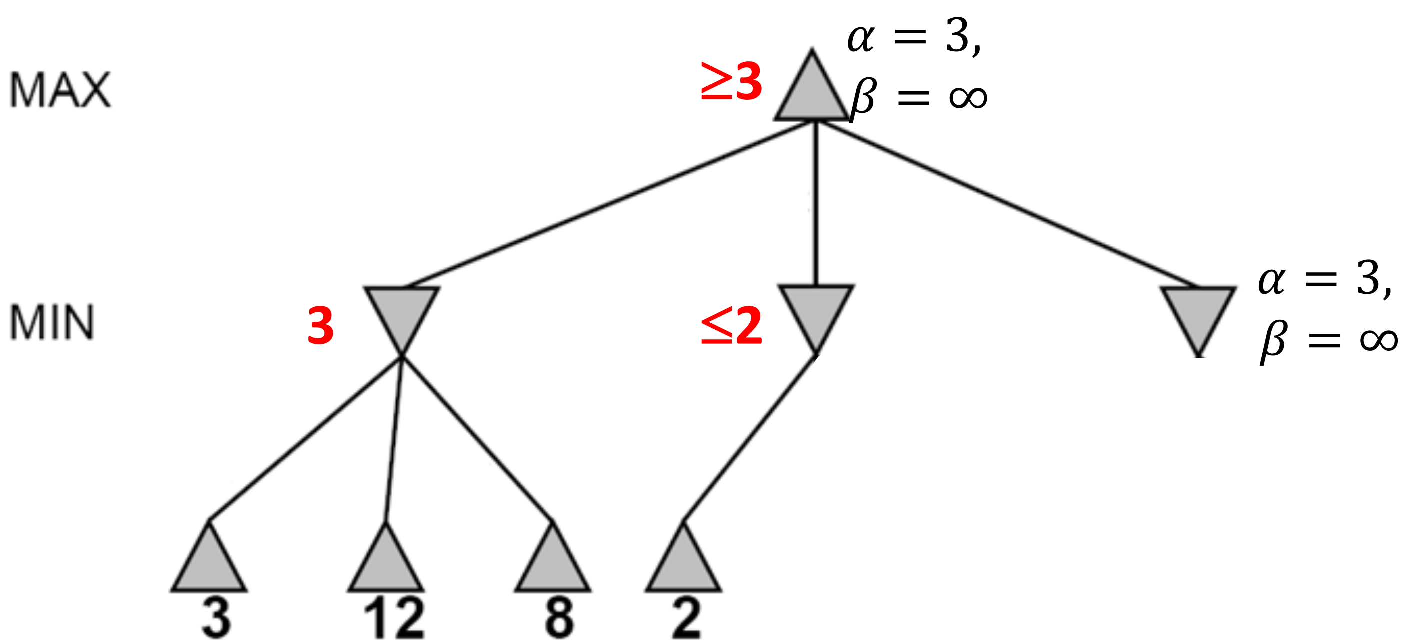


PRUNE!

- If MAX lets us get to this state, then MIN would achieve a final score <=2
- Therefore MAX will never let us get to this state!
- Therefore there's no need to score the remaining children of this node.

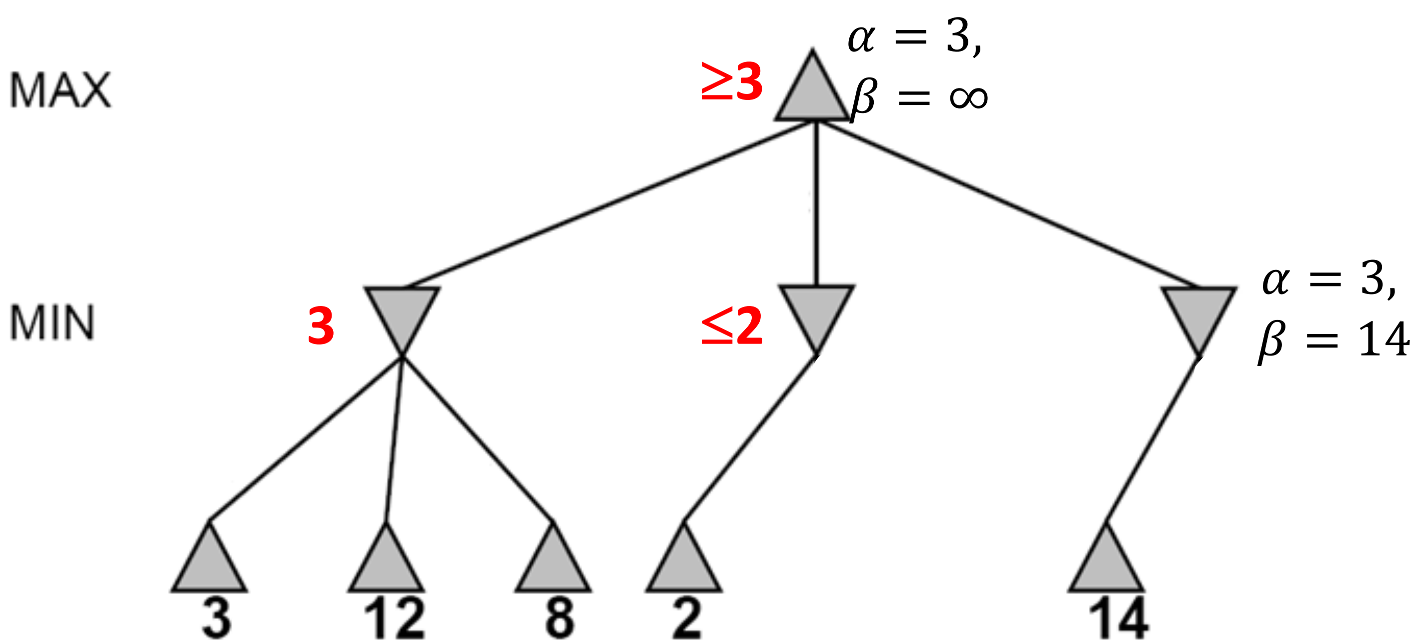
Alpha-beta pruning

Inheritance: Child inherits alpha and beta from its parent



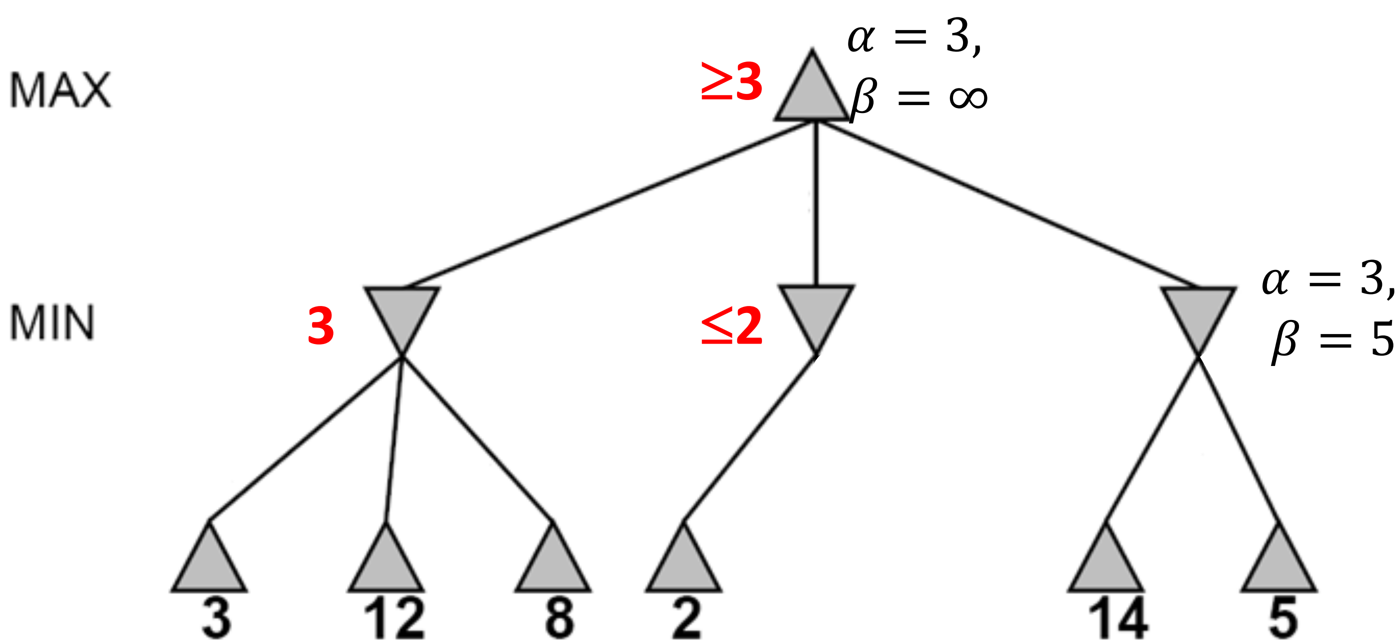
Alpha-beta pruning

Update: a min node can update beta. A max node can update alpha.



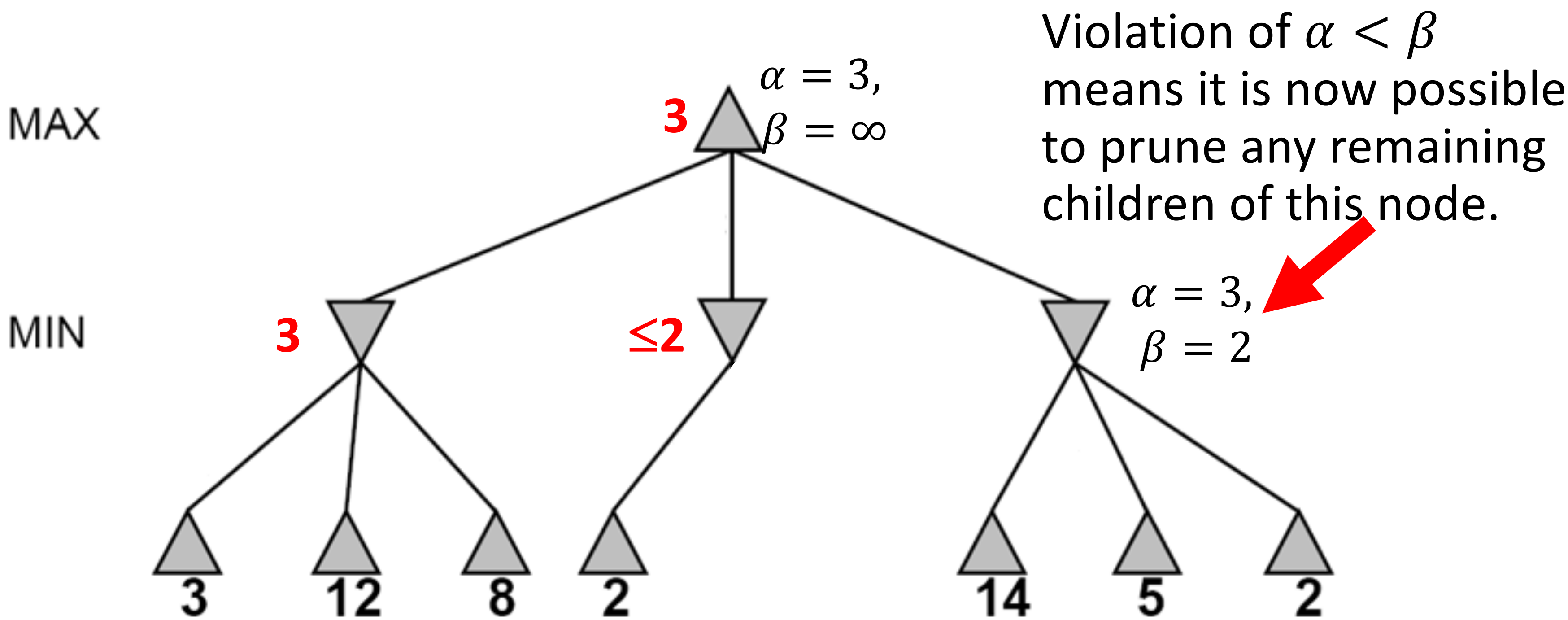
Alpha-beta pruning

Update: a min node can update beta. A max node can update alpha.



Alpha-beta pruning

Pruning: If beta ever falls below alpha, prune any remaining children, and return.



The alpha-beta algorithm

- Max inherits α, β from parents, sets $v = -\infty$, then for each child:
  - Set $v = \max(v, \text{child's } v)$
  - Set $\alpha = \max(\alpha, \text{child's } v)$
  - If $\alpha \geq \beta$, prune all remaining children

- Min inherits α, β from parents, sets $v = \infty$, then for each child:
  - Set $v = \min(v, \text{child's } v)$
  - Set $\beta = \min(\beta, \text{child's } v)$
  - If $\alpha \geq \beta$, prune all remaining children

Quiz

Try the quiz!

Outline

- Alternating two-player zero-sum games
- Minimax search
- Limited-horizon computation and heuristic evaluation functions
- Alpha-beta search
- Computational complexity of minimax and alpha-beta

Computational complexity of alpha-beta pruning

- The worst-case complexity of alpha-beta is the same as the complexity of minimax: $O\{b^d\}$
- The best-case complexity is $O\{b^{d/2}\}$
- It is often possible to achieve results close to the best-case by using a heuristic to sort the nodes before searching them

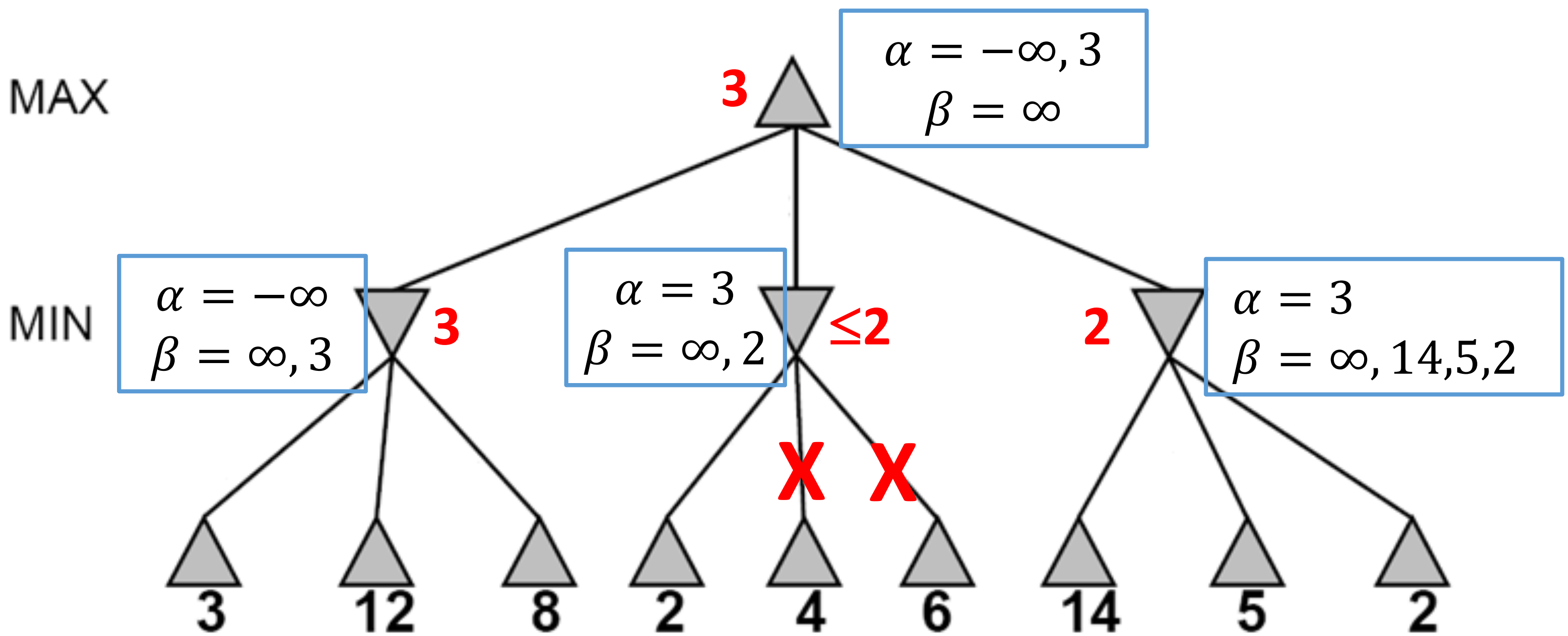
Optimal ordering

Minimum computational complexity ($O\{b^{d/2}\}$) is only achieved if:

- The children of a MAX node are evaluated, in order, starting with the highest-value child.
- The children of a MIN node are evaluated, in order, starting with the lowest-value child.

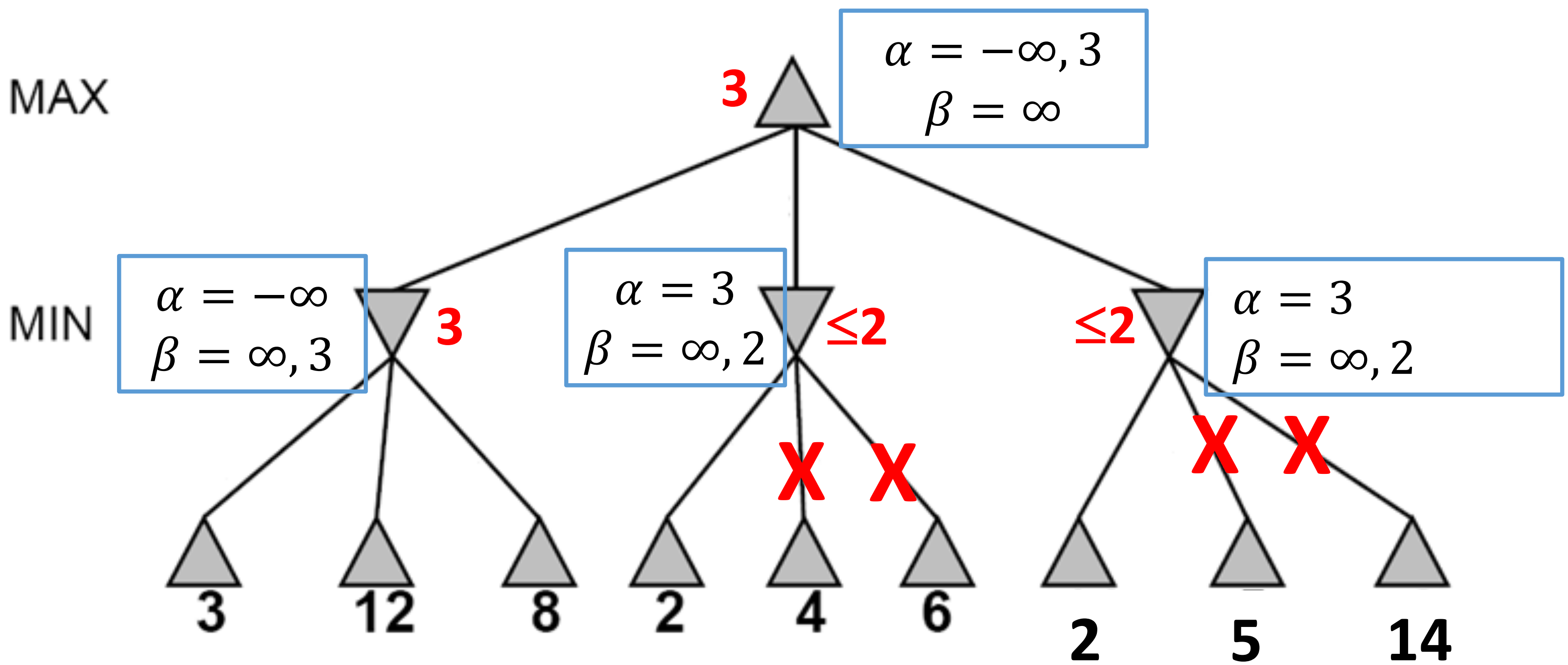
Non-optimal ordering

In this tree, the moves are not optimally ordered, so we were only able to prune two nodes.

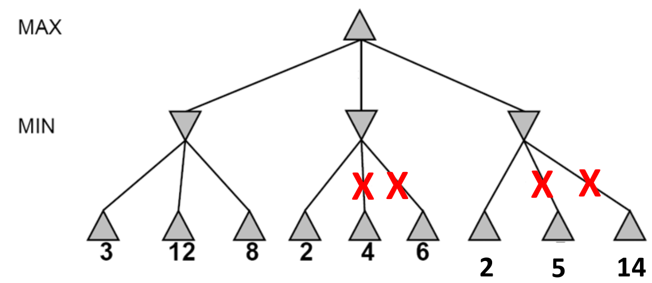


Optimal ordering

In this tree, the moves ARE optimally ordered, so we are able to prune four nodes (out of nine).



Computational Complexity

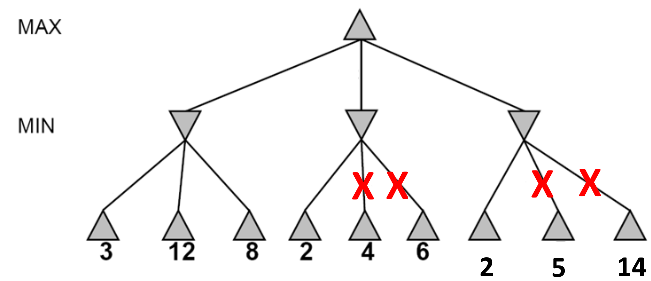


Consider a sequence of two levels, with b moves per level, and with optimal ordering.

- There are b^2 terminal nodes.
- Alpha-beta will evaluate all the children of the first child: b nodes.
- Alpha-beta will also evaluate the first child of each non-first child: $b-1$ nodes.
- In total, alpha-beta will evaluate $2b-1$ out of every b^2 nodes.
- For a tree of depth d, the number of nodes evaluated by alpha-beta is

$$(2b-1)^{d/2} = O\{b^{d/2}\}$$

Computational Complexity



…but wait… this means we need to know, IN ADVANCE, which move has the highest value, and which move has the lowest value!!

- Obviously, it is not possible to know the true value of a move without evaluating it.
- However, heuristics often are pretty good.
- We use the heuristic to decide which move to evaluate first.
- For games like chess, with good heuristics, complexity of alpha-beta is closer to $O\{b^{d/2}\}$ than to $O\{b^d\}$.

Conclusions

- Minimax search
  - Max node: $v(s) = \max_a v(\text{child}(s, a))$
  - Min node: $v(s) = \min_a v(\text{child}(s, a))$
- Limited-horizon computation and heuristic evaluation functions
  - It's impossible to search all the way to the end of the game!
  - Instead, search a fixed number of steps, then estimate $v(s)$ using the best approximation you can think of
- Alpha-beta search
  - Alpha is the highest score that Max knows how to force Min to accept
  - Beta is the lowest score that Min knows how to force Max to accept
  - If Beta ever falls below Alpha, prune the rest of the children
- Computational complexity of minimax and alpha-beta
  - Minimax is $O\{b^d\}$. With optimal move ordering, alpha-beta is $O\{b^{d/2}\}$.