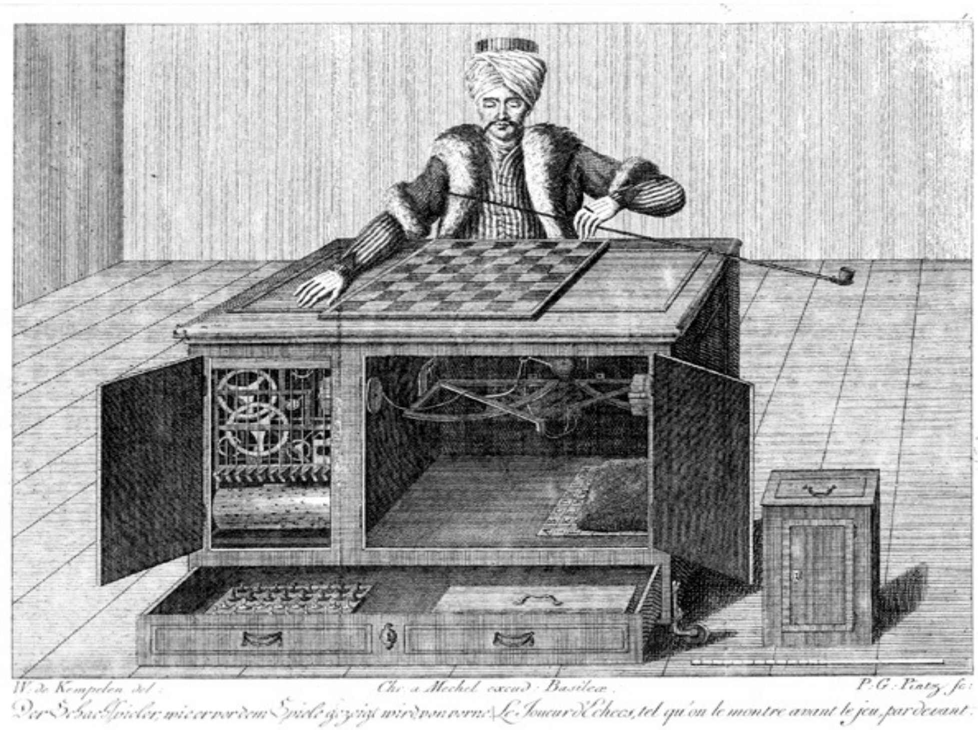
CS440/ECE448 Lecture 18: Two-Player Games

Slides by Mark Hasegawa-Johnson & Svetlana Lazebnik, 3/31/2021



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

Why study games?

- Games are a traditional hallmark of intelligence
- Games are easy to formalize
- Games can be a good model of real-world competitive or cooperative activities
  - Military confrontations, negotiation, auctions, etc.

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 **policy**: a policy is a function $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maps from world states, $s \in \mathcal{S}$, to actions, $\mathrm{a} \in \mathcal{A}$.

Game AI: Origins

- Minimax algorithm: Ernst Zermelo, 1912
- Chess playing with evaluation function, quiescence search, selective search: Claude Shannon, 1949 (paper)
- Alpha-beta search: John McCarthy, 1956
- Checkers program that learns its own evaluation function by playing against itself: Arthur Samuel, 1956

Types of game environments

	Deterministic	**Stochastic**
Perfect information (fully observable)	Chess, Checkers, Go	Backgammon, Monopoly
Imperfect information (partially observable)	Battleship	Scrabble, Poker, Bridge

Zero-sum Games

Alternating two-player zero-sum games

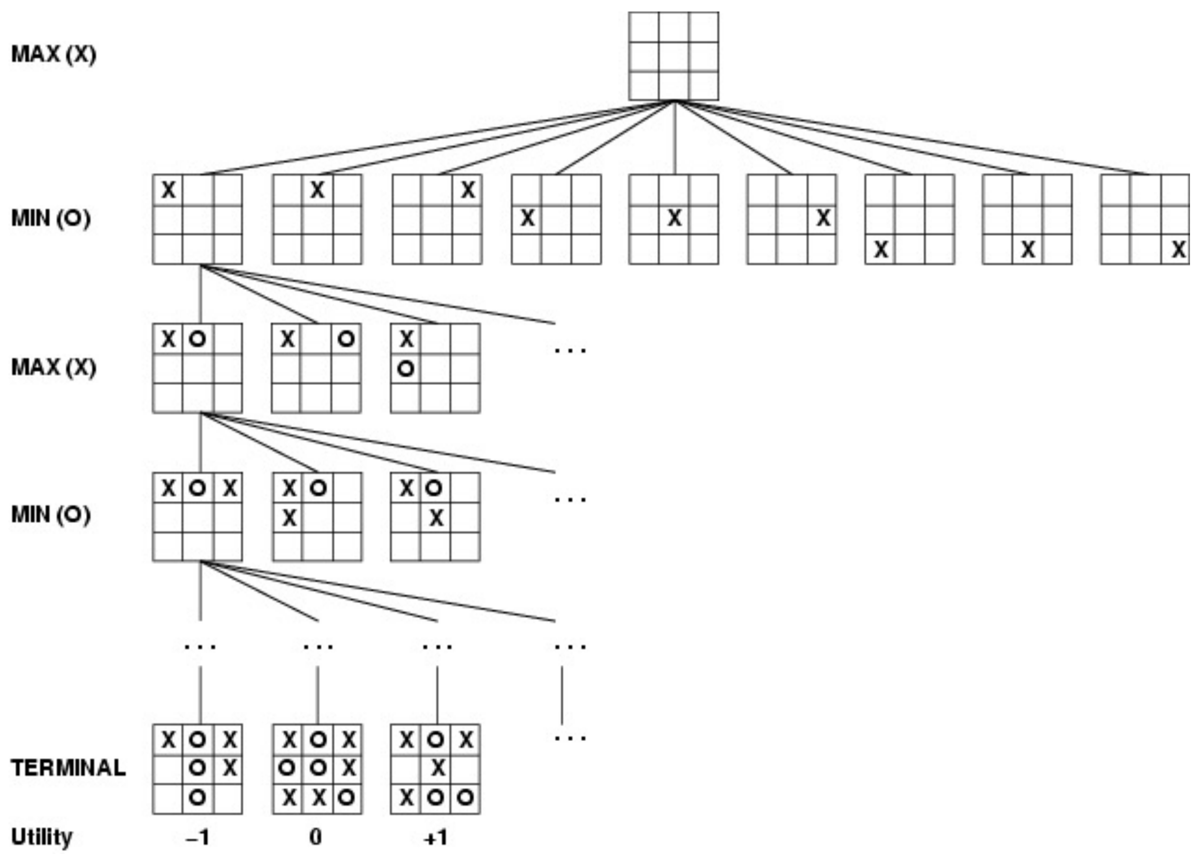
- Players take turns
- Each game outcome or **terminal state** has a **utility** for each player (e.g., 1 for win, 0 for tie, -1 for loss)
- The sum of both players' utilities is a constant, e.g.,

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

- 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”

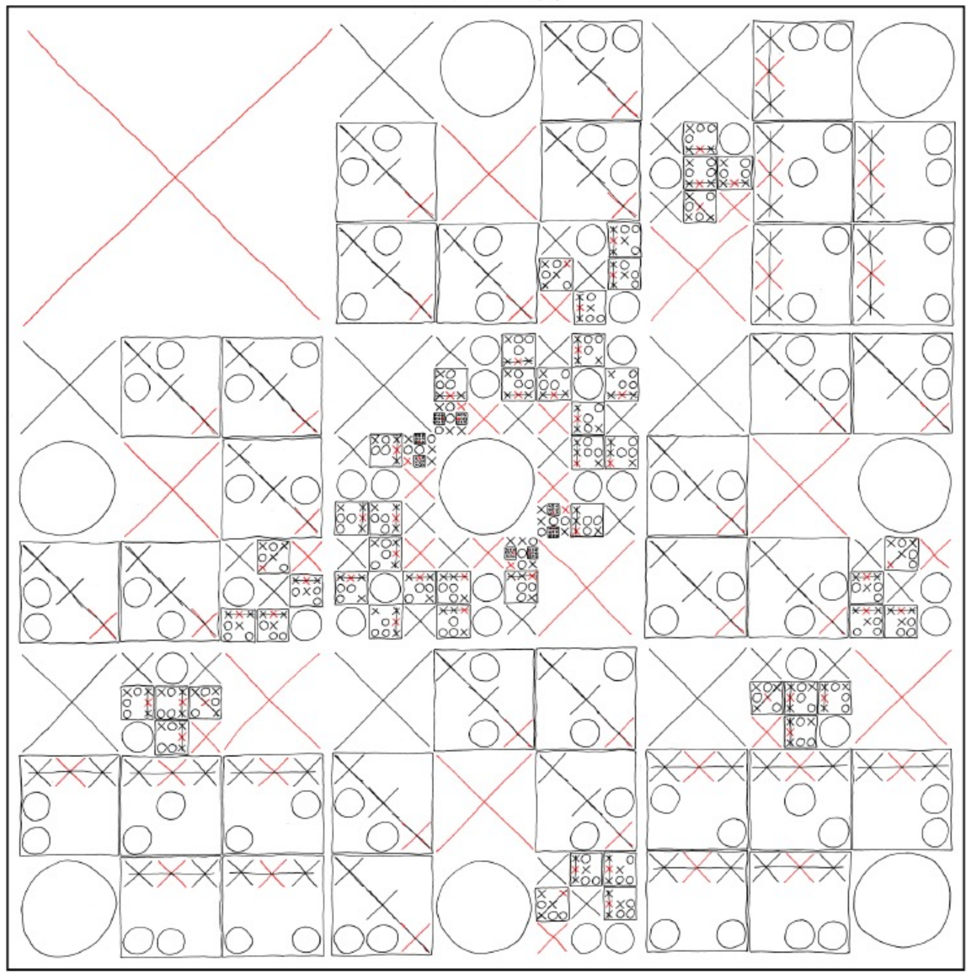


COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES

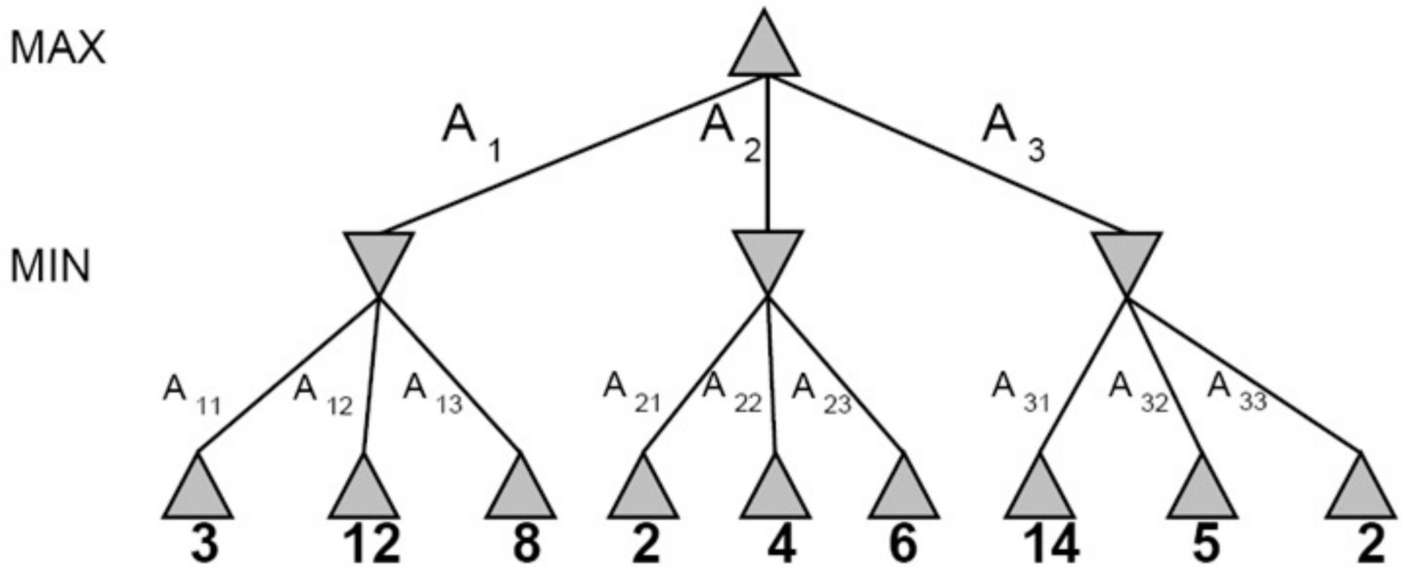
YOUR MOVE IS GIVEN BY THE POSITION OF THE LARGEST RED SYMBOL ON THE GRID. WHEN YOUR OPPONENT PICKS A MOVE, ZOOM IN ON THE REGION OF THE GRID WHERE THEY WENT. REPEAT.

http://xkcd.com/832/

MAP FOR X:



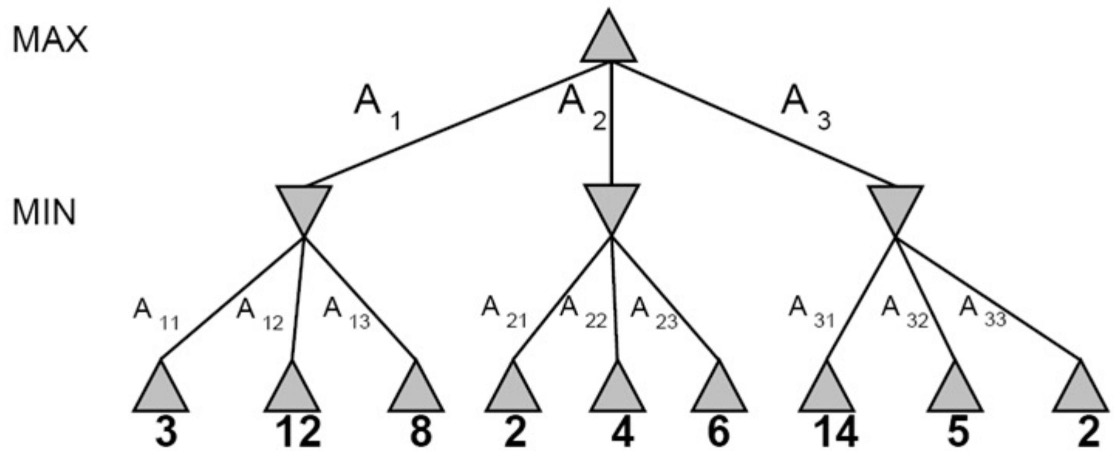
A more abstract game tree

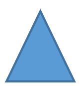


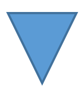
Terminal utilities (for MAX)

A *depth-two* game

Standard notation for game trees



 = game state from which MAX can play

 = game state from which MIN can play

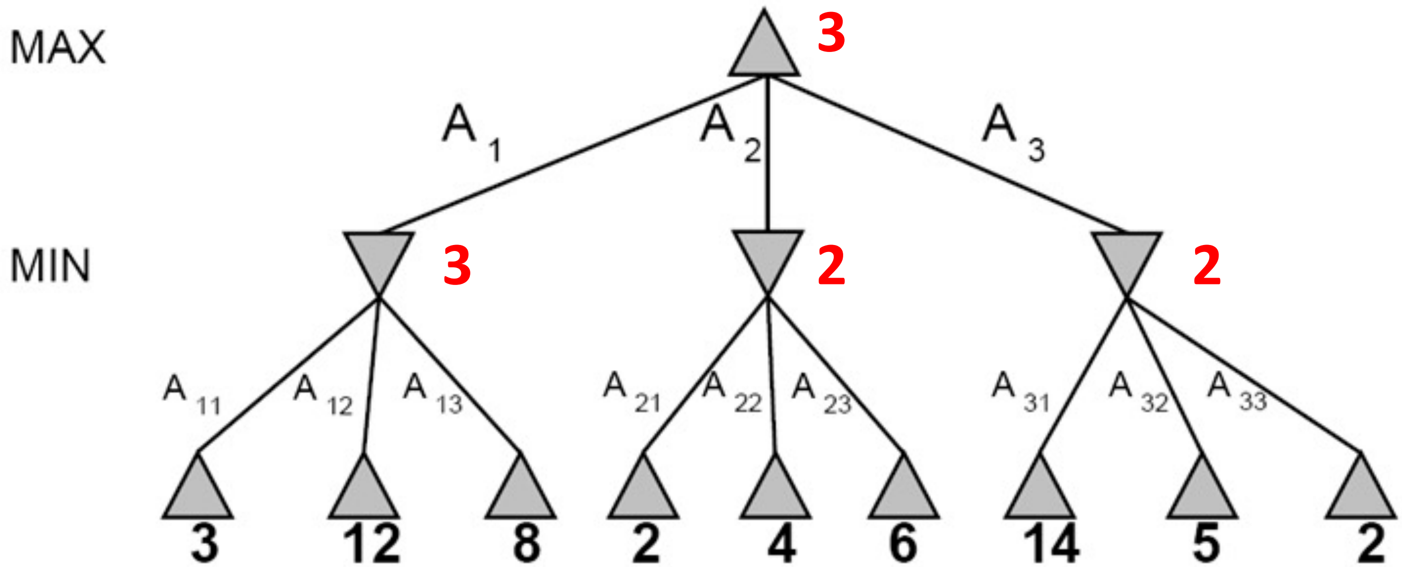
number = value of that game state for MAX

Minimax Search

The rules of every game

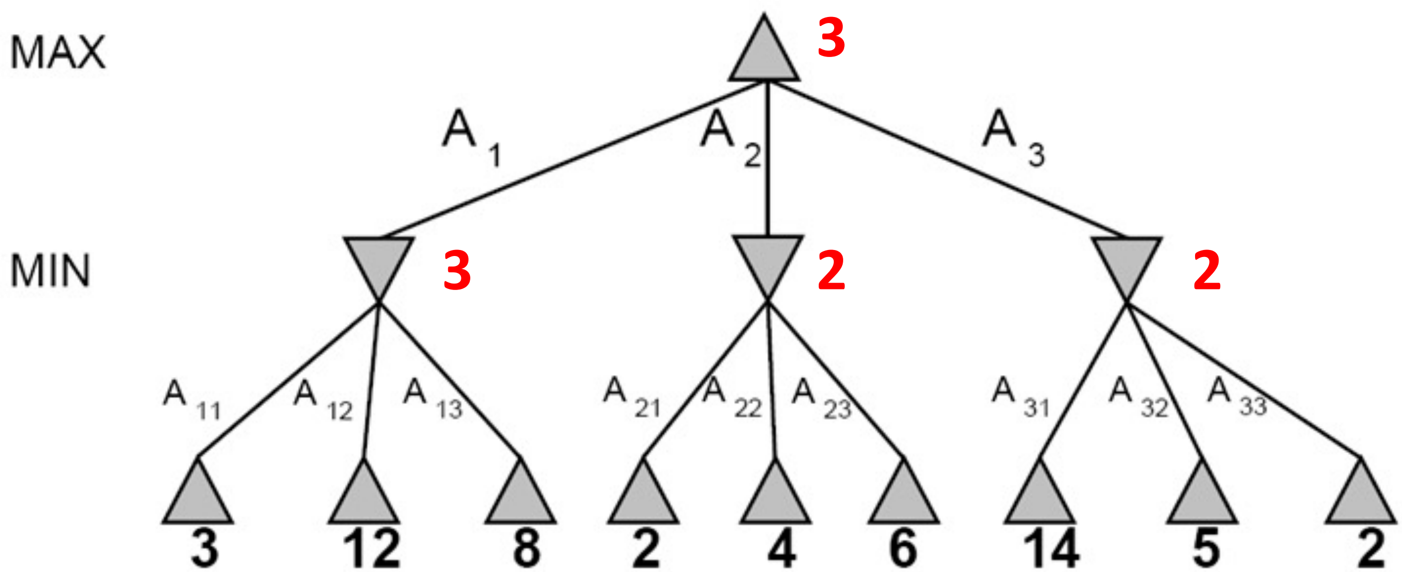
- Every possible outcome has a value (or "utility") for me.
- Zero-sum game: if the value to me is +V, then the value to my opponent is –V.
- Phrased another way:
  - My rational action, on each move, is to choose a move that will maximize the value of the outcome
  - My opponent's rational action is to choose a move that will minimize the value of the outcome
- Call me "Max"
- Call my opponent "Min"

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

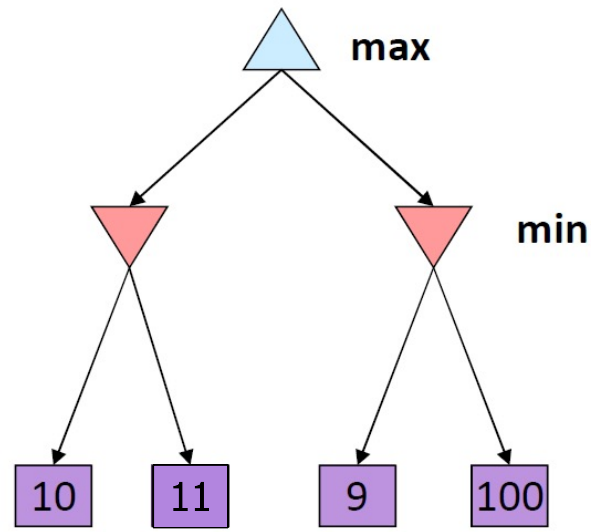
Computing the minimax value of a node



- **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 **minimax-optimal** sequence of moves, then the utility you earn will always be **greater than or equal** to the amount that you predict.

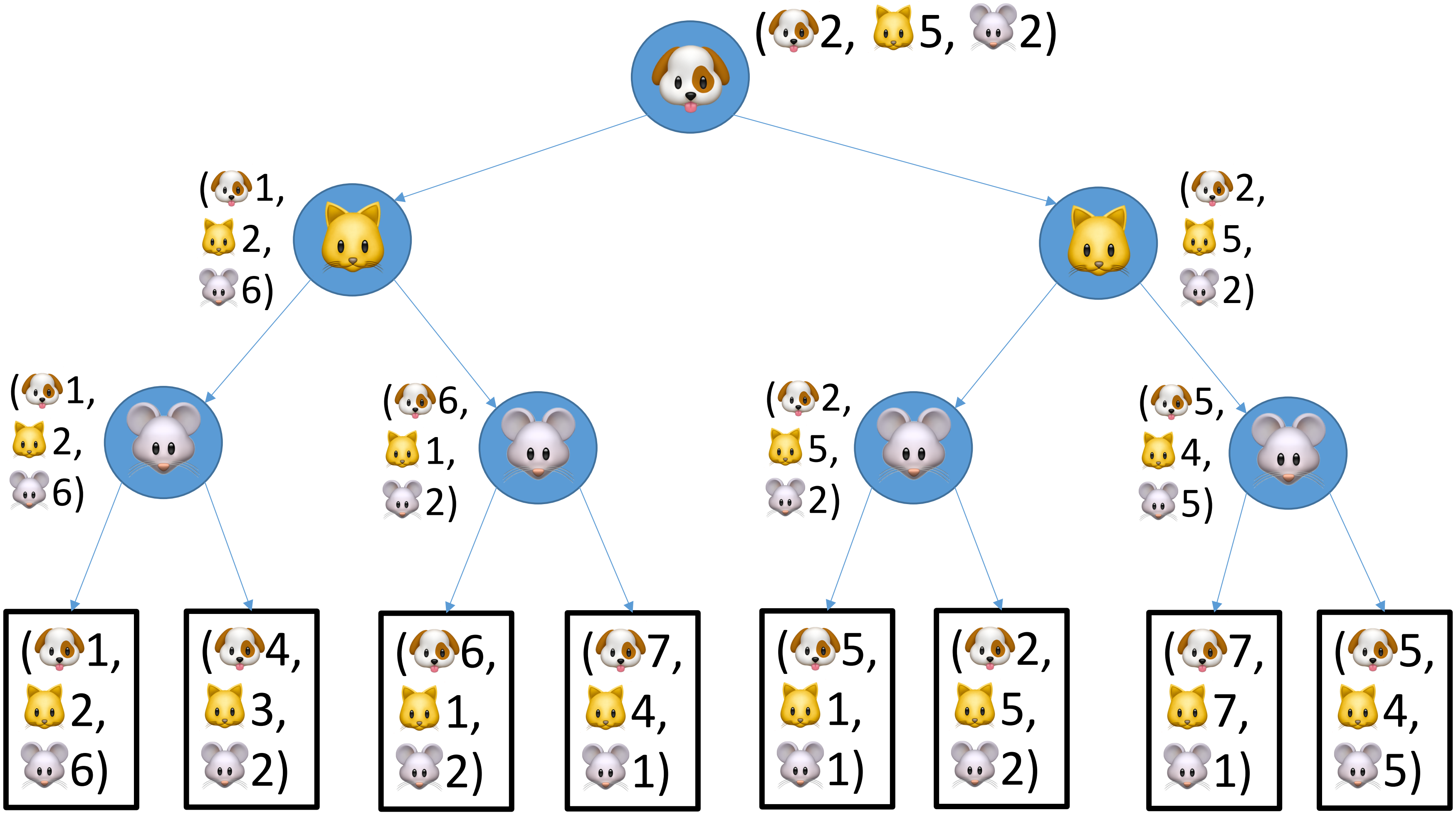


Example from D. Klein and P. Abbeel

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



Limited-Horizon Computation

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*** (a mapping from state to best move in that state)

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*** (a mapping from state to best move in that state)
- Efficiency is critical to playing well
  - The time to make a move is limited
  - The branching factor, search depth, and number of terminal configurations are huge
    - In chess, branching factor ≈ 35 and depth ≈ 100, giving a search tree of 10^{154} nodes
      - Number of atoms in the observable universe ≈ 10^{80}
  - This rules out searching all the way to the end of the game

Limited-Horizon Search

In a practical game, we compute minimax to a limited depth

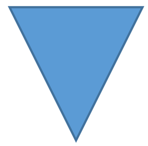
- Depth=1: evaluate every possible current move, look at the resulting game state, decide which resulting game state looks the best, and take that action.
  - Computational complexity to choose your next move: $\mathcal{O}\{N\}$, if there are N possible moves.
- Depth=2: evaluate every possible current 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: $\mathcal{O}\{N^2\}$.
- 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: $\mathcal{O}\{N^3\}$.

Evaluation functions

In order to evaluate the quality of a game state $s \in \mathcal{S}$, we need to design an evaluation function $v(s)$. 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 that its computational complexity is no more than $\mathcal{O}\{N\}$. If its complexity was higher, then we might get better results by using a cheaper evaluation function in a deeper minimax search.

Example: Depth 1 search, Chess



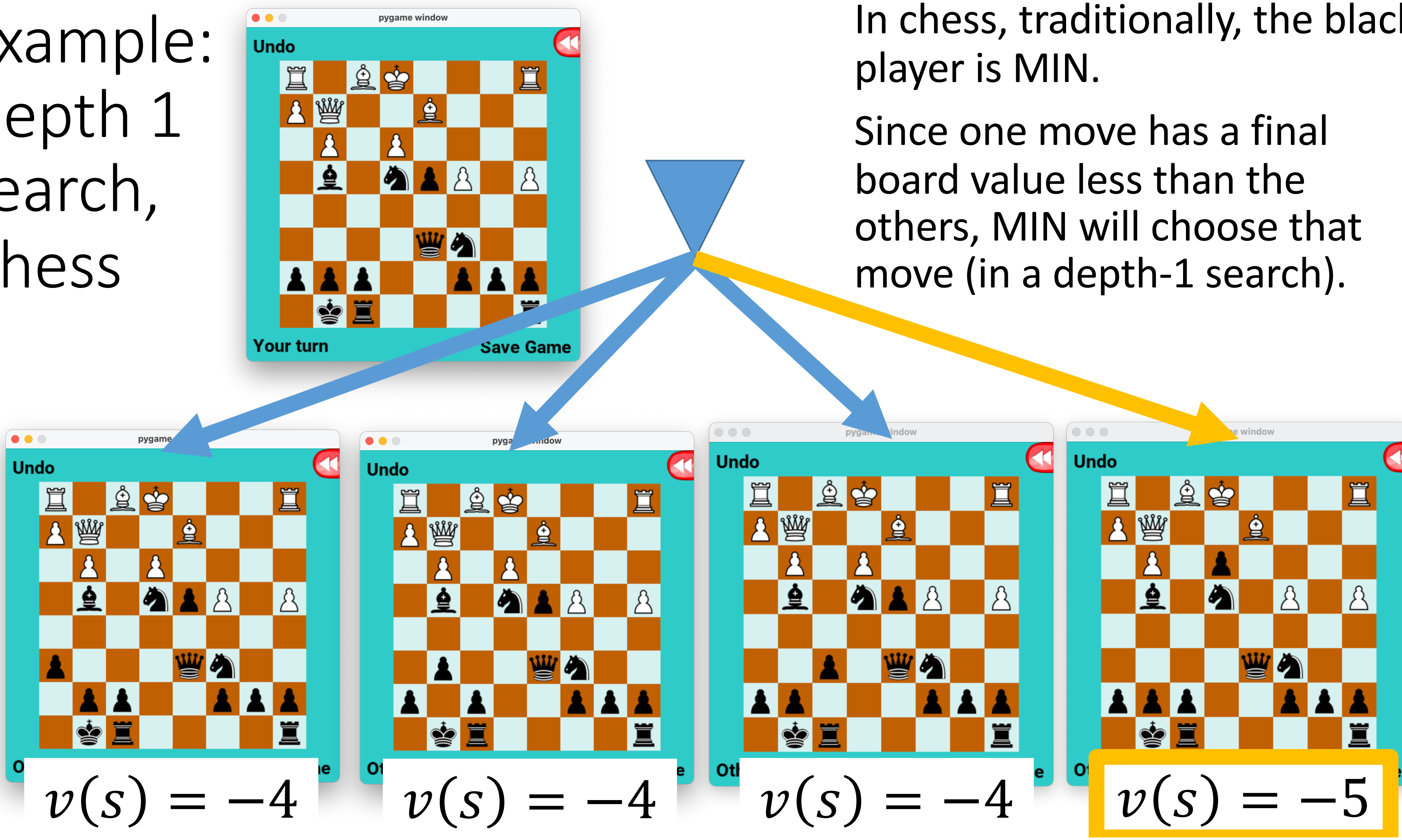
In chess, traditionally, the black player is MIN.

What move should MIN choose, from this board position?

Graphics: created by the PyChess community.

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

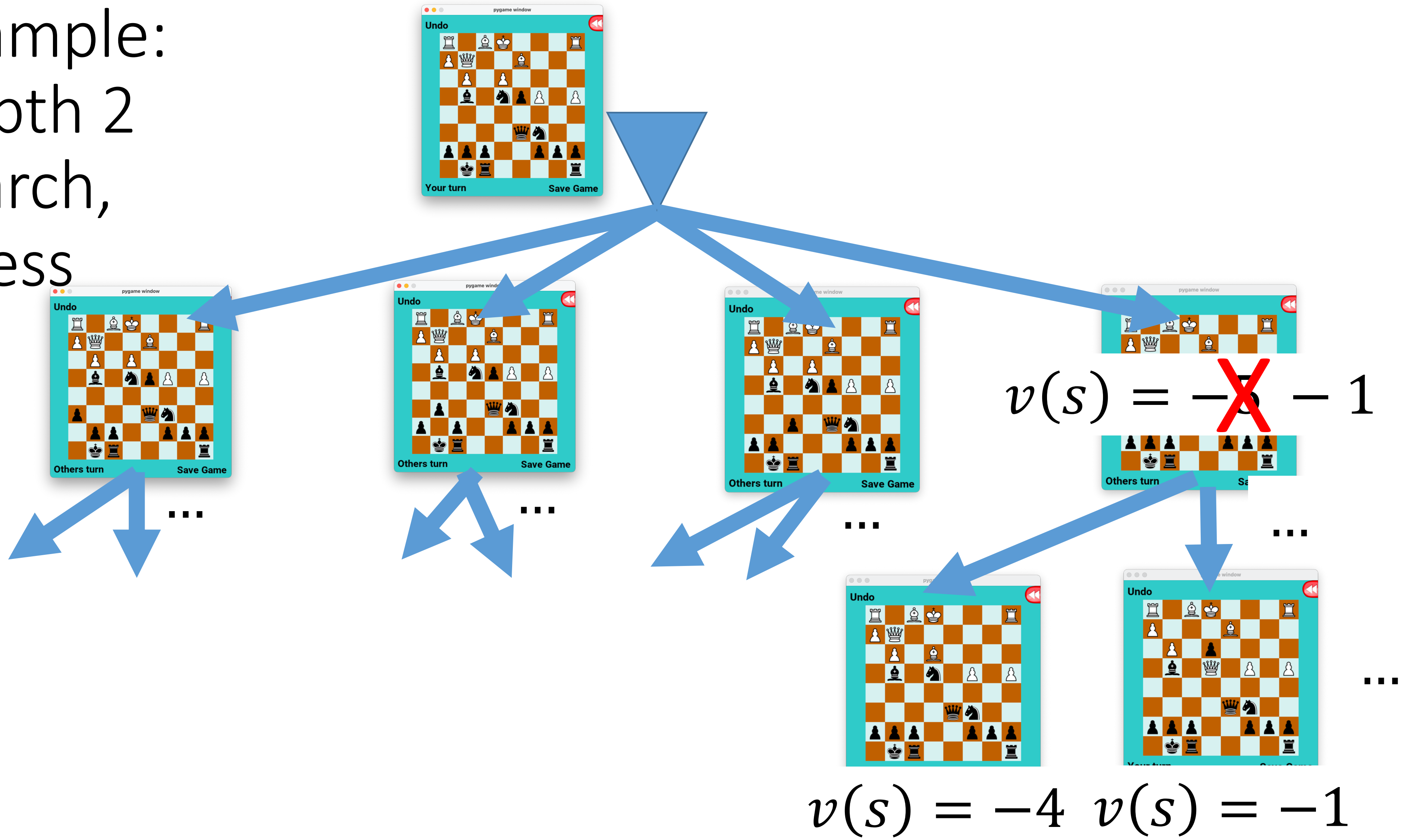
Example: Depth 1 search, Chess



In chess, traditionally, the black player is MIN.

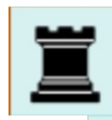
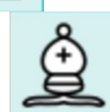
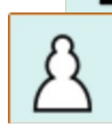
Since one move has a final board value less than the others, MIN will choose that move (in a depth-1 search).

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.

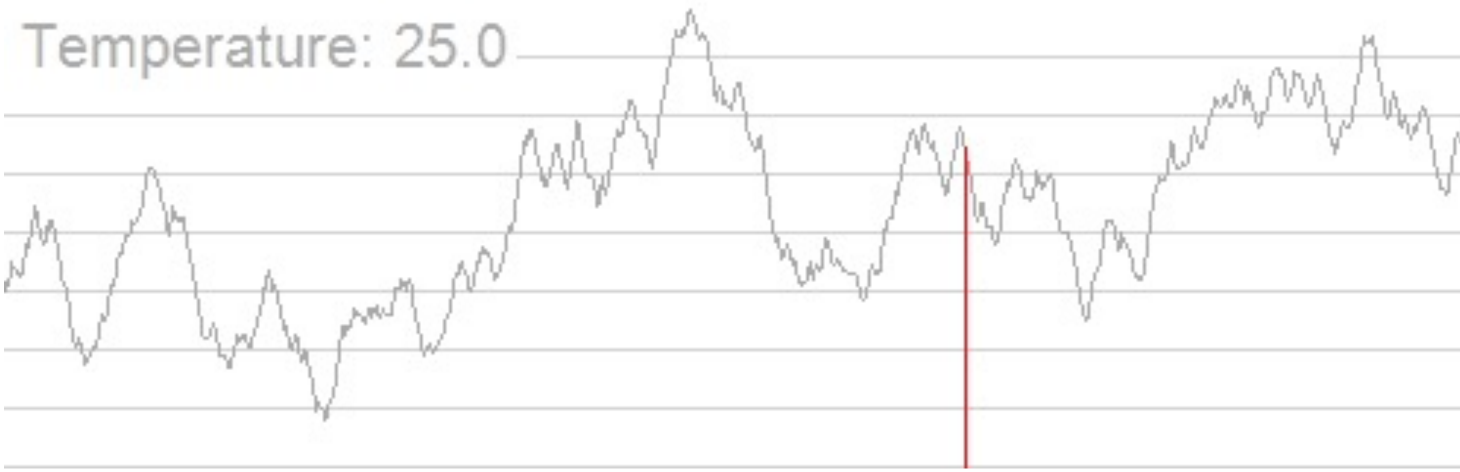
Notice: this is just a one-layer neural net, with input vector $f(s) = [f_1(s), f_2(s), \ldots]$ and weight vector $w = [w_1, w_2, \ldots]$.

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:** consider only “unstable” moves, e.g., moves that capture a piece.
  - **Singular extension:** consider only very strong moves.
  - **Stochastic search**: randomly sample a small number of possible future paths.

Stochastic search



Stochastic search

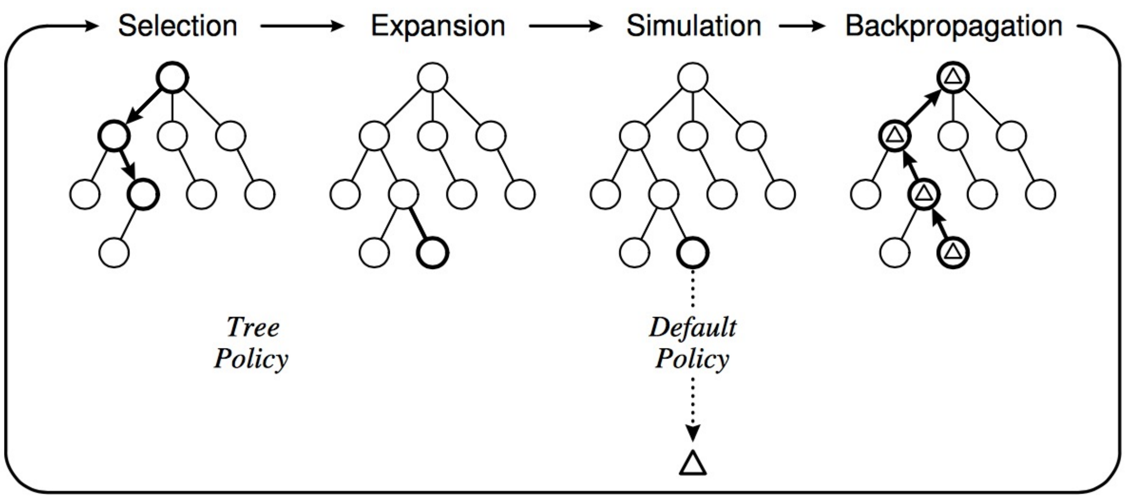
- An approximate solution: stochastic search

$$v(s) \approx \frac{1}{n}\sum_{i=1}^{n} v(i^{th} \text{ random game starting from } s)$$

- Asymptotically optimal: as $n \to \infty$, the approximation gets better.
- Controlled computational complexity: choose n to match the amount of computation you can afford.

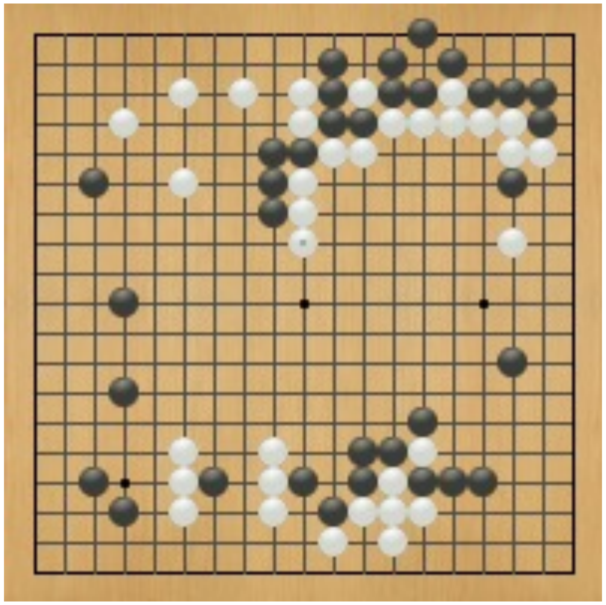
Stochastic search

- Instead of depth-limited search with an evaluation function, use randomized simulations
- Starting at the current state (root of search tree), iterate:
  - Select a leaf node for expansion using some type of random move selection policy
  - Continue until desired depth
  - For any given move, average the value of the final game states to determine the value of the move.



C. Browne et al., [A survey of Monte Carlo Tree Search Methods](), 2012

Case study: AlphaGo

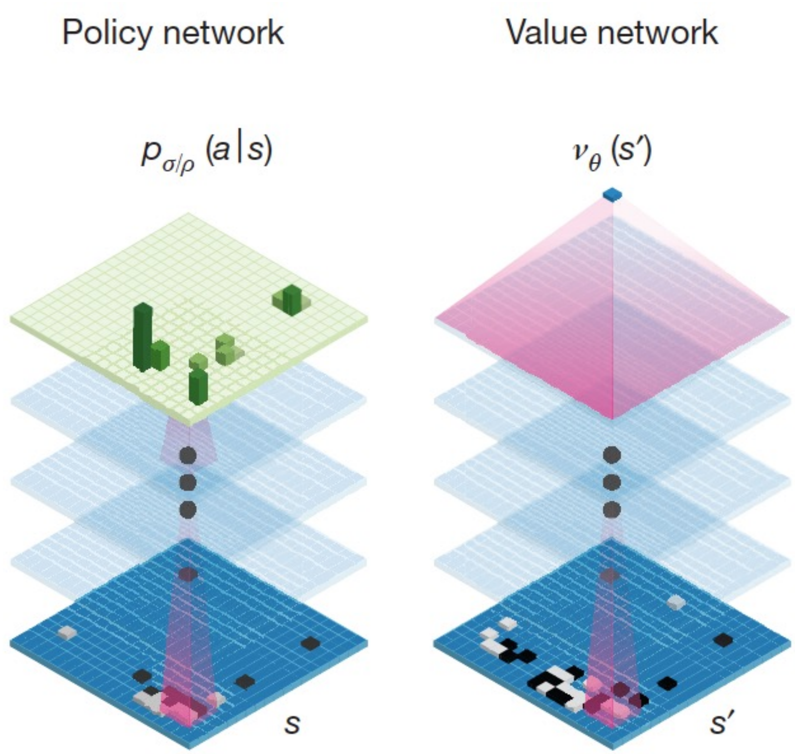


- **"Gentlemen should not waste their time on trivial games -- they should play Go."**
- ***-- Confucius,***
- ***The Analects***
- ***ca. 500 B. C. E.***

Anton Ninno, Roy Laird, Ph.D.
antonninno@yahoo.com
roylaird@gmail.com

special thanks to Kiseido Publications

AlphaGo

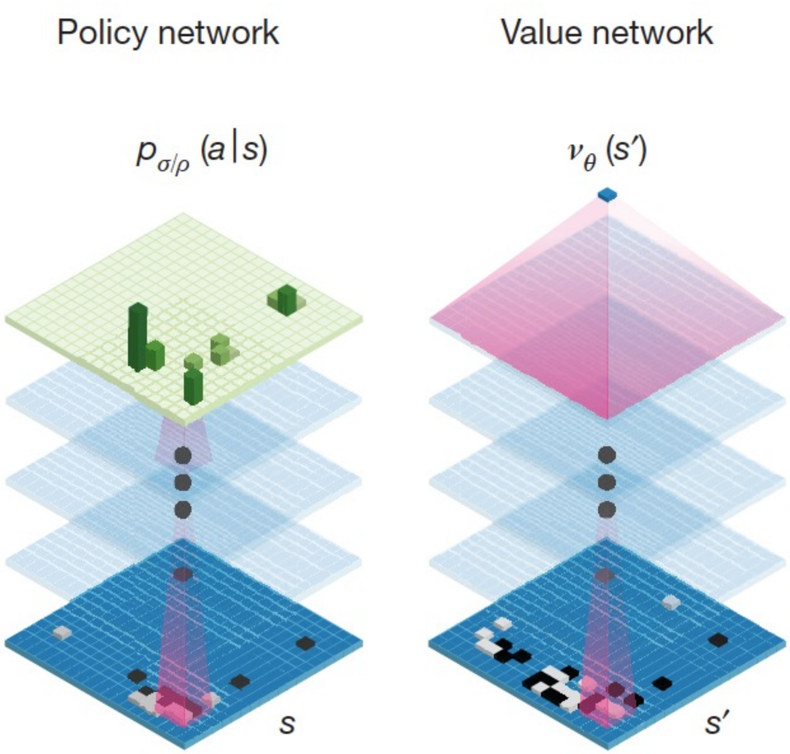


Deep convolutional neural networks

- Treat the Go board as an image
- Can be trained to predict distribution over possible moves (*policy*) or expected *value* of position

D. Silver et al., Mastering the Game of Go with Deep Neural Networks and Tree Search, Nature 529, January 2016

AlphaGo



- Policy network: Given a game state, s, predict what would be the best next move.
  - Input: game board as an image, s.
  - Output: $p(a|s)$, probability that action a is best.
- Value network: Given a game state, s, compute the expected value of the board for player 0 (MAX).
  - Input: game board as an image, s.
  - Output: $v(s)$, value of the game state.

D. Silver et al., [Mastering the Game of Go with Deep Neural Networks and Tree Search](), Nature 529, January 2016

Stochastic Search in AlphaGo

- Each edge in the search tree has
  - Probabilities $p(a|s)$ computed by the policy network
  - State+Move values $Q(s,a)$ computed by the value network
  - Counts $N(s,a)$ specifying how many times that move has been tried
- Tree traversal policy selects actions randomly according to some combination of $p(a|s)$, $Q(s,a)$, and $N(s,a)$
- At the end of each simulation, values of the final boards are averaged in order to re-estimate the value of the initial move.

Stochastic Search in AlphaGo

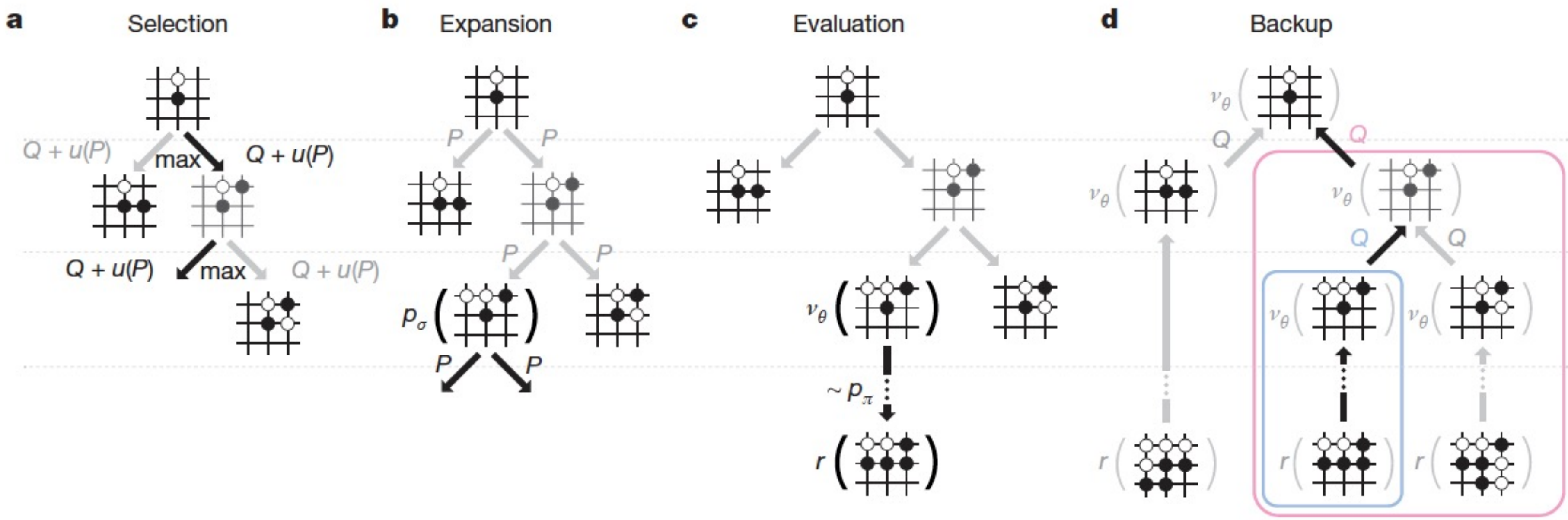


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

D. Silver et al., [Mastering the Game of Go with Deep Neural Networks and Tree Search](), Nature 529, January 2016

Summary

- A zero-sum game can be expressed as a minimax tree.
- Limited-horizon search is always necessary (you can't search to the end of the game), and always suboptimal.
- Evaluation function: a relatively low-complexity function that estimates the value of the board (maybe linear, maybe a neural net)
- Stochastic search: randomly choose moves, out to some pre-determined depth, then average the final board positions to estimate the value of the initial move