Artificial Intelligence

11. Adversarial Search

What To Do When Your "Solution" is Somebody Else's Failure

Jana Koehler Álvaro Torralba

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Summer Term 2019

Thanks to Prof. Hoffmann for slide sources

Agenda

1. Introduction
2. Minimax Search
3. Evaluation Functions
4. Alpha-Beta Search
5. Monte-Carlo Tree Search (MCTS)
6. Conclusion

The Problem

→ "Adversarial search" = Game playing against an opponent.

Why AI Game Playing?

Many good reasons:

- Playing a game well clearly requires a form of "intelligence".
- Games capture a pure form of competition between opponents.
- Games are abstract and precisely defined, thus very easy to formalize.

→ Game playing is one of the oldest sub-areas of AI (ca. 1950).

→ The dream of a machine that plays Chess is, indeed, *much* older than AI! (von Kempelen's "Schachtürke" (1769), Torres y Quevedo's "El Ajedrecista" (1912))

"Game" Playing? *Which* Games?

... sorry, we're not gonna do football here.

Restrictions:

- Game states discrete, number of game states finite.
- Finite number of possible moves.
- The game state is fully observable.
- The outcome of each move is deterministic.

- Two players: Max and Min.
- Turn-taking: It's each player's turn alternatingly. Max begins.

- Terminal game states have a utility u. Max tries to maximize u, Min tries to minimize u.
- In that sense, the utility for Min is the exact opposite of the utility for Max ("zero-sum").
- There are no infinite runs of the game (no matter what moves are chosen, a terminal state is reached after a finite number of steps).

An Example Game

- Game states: Positions of figures.
- Moves: Given by rules.
- Players: White (Max), Black (Min).
- Terminal states: Checkmate.
- Utility of terminal states, e.g.:
  - $+100$ if Black is checkmated.
  - 0 if stalemate.
  - -100 if White is checkmated.

"Game" Playing? Which Games *Not*?

. . . football.

Important types of games that we don't tackle here:

- Chance. (E.g., backgammon)
- More than two players. (E.g., halma)
- Hidden information. (E.g., most card games)
- Simultaneous moves. (E.g., football)
- Not zero-sum, i.e., outcomes may be beneficial (or detrimental) for both players. ($\rightarrow$ Game theory: Auctions, elections, economy, politics, . . .)

$\rightarrow$ Many of these more general game types can be handled by similar/extended algorithms.

(A Brief Note On) Formalization

Definition (Game State Space). *A game state space is a 6-tuple* $\Theta = (S, A, T, I, S^T, u)$ *where:*

- S, A, T, I: *States, actions, deterministic transition relation, initial state. As in classical search problems, except:*
  - S *is the disjoint union of* S^{Max}, S^{Min}, *and* S^T.
  - A *is the disjoint union of* A^{Max} *and* A^{Min}.
  - *For* $a \in A^{Max}$, *if* $s \xrightarrow{a} s'$ *then* $s \in S^{Max}$ *and* $s' \in S^{Min} \cup S^T$.
  - *For* $a \in A^{Min}$, *if* $s \xrightarrow{a} s'$ *then* $s \in S^{Min}$ *and* $s' \in S^{Max} \cup S^T$.
- S^T *is the set of terminal states.*
- $u : S^T \mapsto \mathbb{R}$ *is the utility function.*

Commonly used terminology: state=position, terminal state=end state, action=move.

(A round of the game – one move Max, one move Min – is often referred to as a "move", and individual actions as "half-moves". We *don't* do that here.)

Why Games are Hard to Solve

Our Agenda for This Chapter

- **Minimax Search:** How to compute an optimal strategy?

  → Minimax is the canonical (and easiest to understand) algorithm for *solving* games, i.e., computing an optimal strategy.

- **Evaluation Functions:** But what if we don't have the time/memory to solve the entire game?

  → Given limited time, the best we can do is look ahead as far as possible. Evaluation functions tell us how to evaluate the leaf states at the cut-off.

- **Alpha-Beta Search:** How to prune unnecessary parts of the tree?

  → An essential improvement over Minimax.

- **Monte-Carlo Tree Search (MCTS):** An alternative form of game search, based on sampling rather than exhaustive enumeration.

  → The main alternative to Alpha-Beta Search.

→ Alpha-Beta = state of the art in Chess, MCTS = state of the art in Go.

Questionnaire

Question!

When was the first game-playing computer built?

(A): 1941 (B): 1950

(C): 1958 (D): 1965

Question!

Does the video game industry attempt to make the computer opponents as intelligent as possible?

(A): Yes (B): No

"Minimax"?

$\rightarrow$ We want to compute an optimal move for player "Max". In other words: "We are Max, and our opponent is Min."

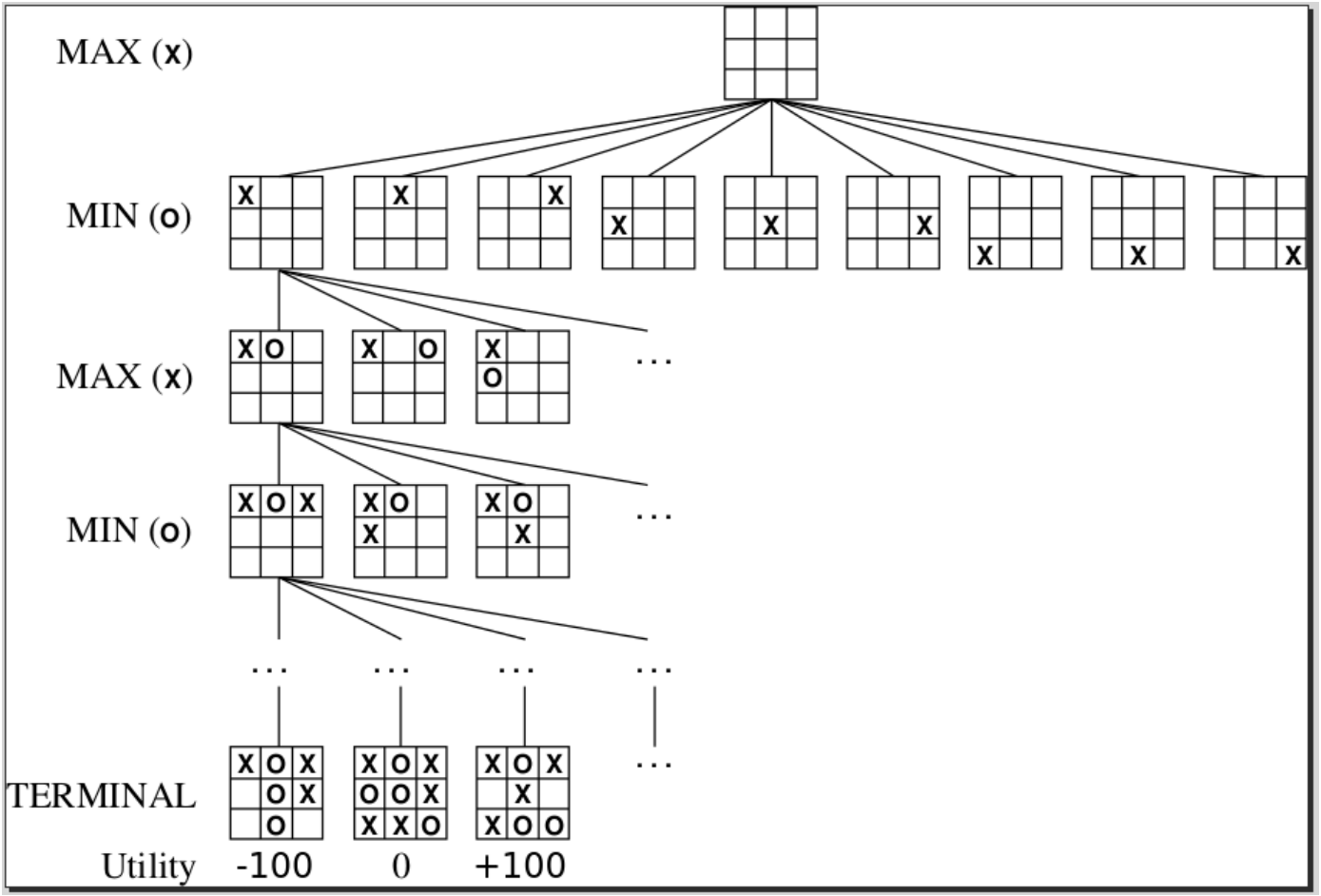
Remember:

- Max attempts to *maximize* the utility $u(s)$ of the terminal state that will be reached during play.
- Min attempts to *minimize* $u(s)$.

So what?

- The computation alternates between minimization and maximization $\Longrightarrow$ hence "Minimax".

Example Tic-Tac-Toe



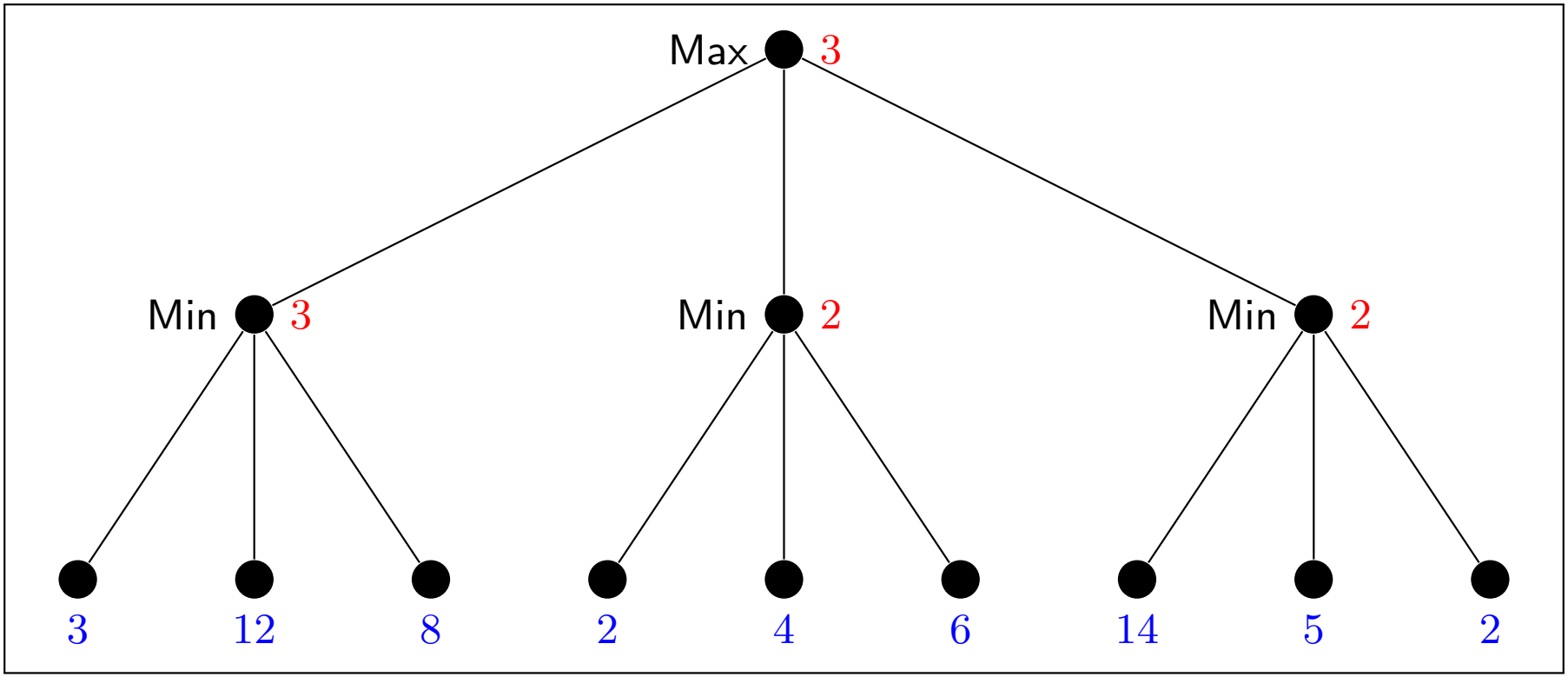
- Game tree, current player marked on the left.
- Last row: terminal positions with their utility.

Minimax: Outline

We max, we min, we max, we min . . .

1. Depth-first search in game tree, with Max in the root.
2. Apply utility function to terminal positions.
3. Bottom-up for each inner node n in the tree, compute the utility $u(n)$ of n as follows:
   - If it's Max's turn: Set $u(n)$ to the maximum of the utilities of n's successor nodes.
   - If it's Min's turn: Set $u(n)$ to the minimum of the utilities of n's successor nodes.
4. Selecting a move for Max at the root: Choose one move that leads to a successor node with maximal utility.

Minimax: Example



- Blue numbers: Utility function u applied to terminal positions.
- Red numbers: Utilities of inner nodes, as computed by Minimax.

Minimax: Pseudo-Code

Input: State $s \in S^{Max}$, in which Max is to move.

```
function Minimax-Decision(s) returns an action
  v ← Max-Value(s)
  return an action a ∈ Actions(s) yielding value v

function Max-Value(s) returns a utility value
  if Terminal-Test(s) then return u(s)
  v ← −∞
  for each a ∈ Actions(s) do
     v ← max(v, Min-Value(ChildState(s, a)))
  return v

function Min-Value(s) returns a utility value
  if Terminal-Test(s) then return u(s)
  v ← +∞
  for each a ∈ Actions(s) do
     v ← min(v, Max-Value(ChildState(s, a)))
  return v
```

Minimax: Example, Now in Detail

Minimax, Pro and Contra

Pro:

- Minimax is the simplest possible (reasonable) game search algorithm.

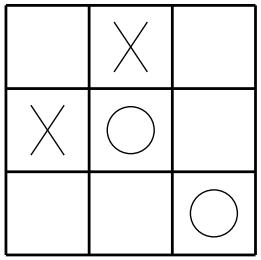
  If any of you sat down, prior to this lecture, to implement a Tic-Tac-Toe player, chances are you invented this in the process (or looked it up on Wikipedia).
- Returns an optimal action, assuming perfect opponent play.

Contra: Completely infeasible (search tree way too large).

Remedies:

- Limit search depth, apply evaluation function to the cut-off states.
- Use alpha-beta pruning to reduce search.
- Don't search exhaustively; sample instead: MCTS.

Questionnaire



- Tic Tac Toe.
- Max = x, Min = o.
- Max wins: $u = 100$; Min wins: $u = -100$; stalemate: $u = 0$.

Question!

What's the Minimax value for the state shown above? (Note: Max to move)

(A): 100 (B): -100

Question!

What's the Minimax value for the initial game state?

(A): 100 (B): -100

Evaluation Functions

Problem: Minimax game tree too big.

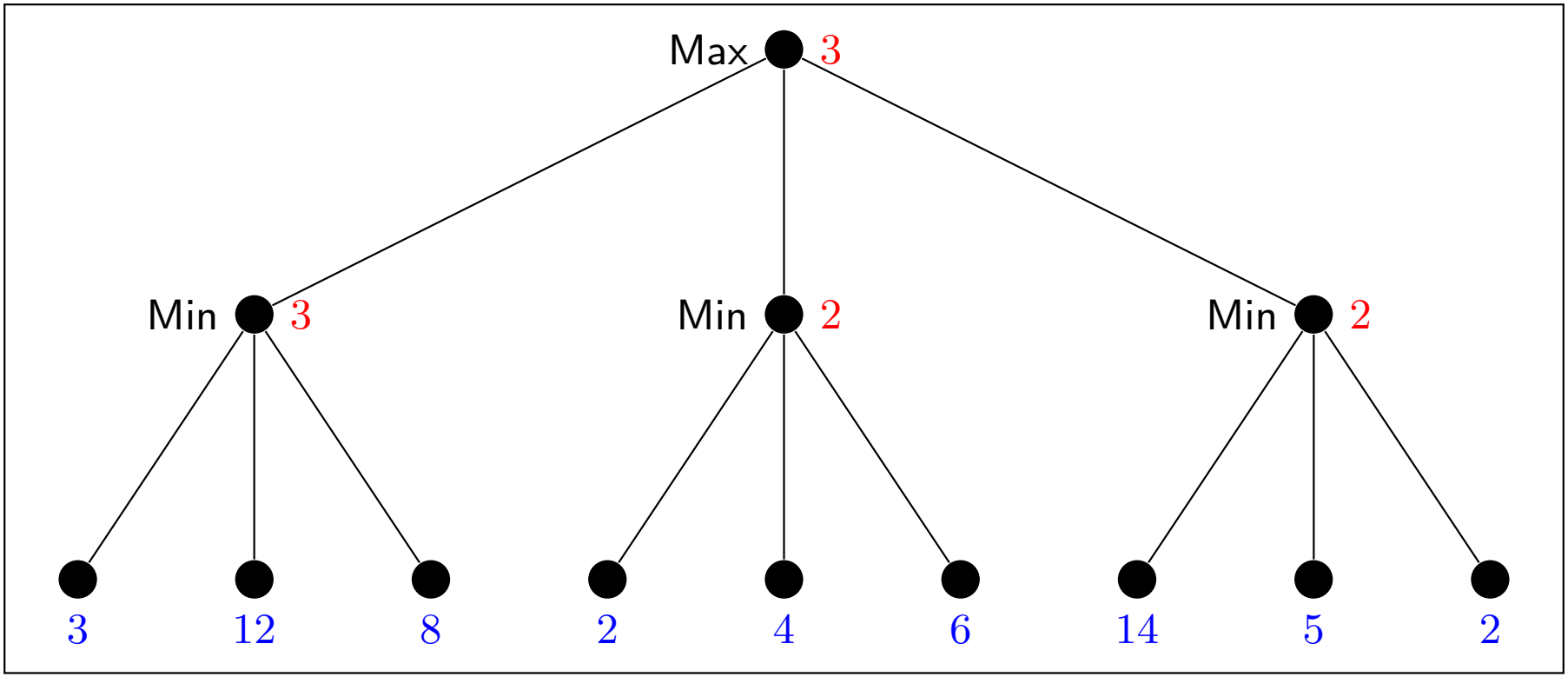
Solution: Impose a search depth limit ("horizon") d, and apply an evaluation function to the non-terminal cut-off states.

An evaluation function f maps game states to numbers:

- $f(s)$ is an estimate of the actual value of s (as would be computed by unlimited-depth Minimax for s).
  $\rightarrow$ If cut-off state is terminal: Use actual utility u instead of f.
- Analogy to heuristic functions (cf. **Chapter 4**): We want f to be both (a) accurate and (b) fast.
- Another analogy: (a) and (b) are in contradiction ... need to trade-off accuracy against overhead.

  $\rightarrow$ Most games (e.g. Chess): f inaccurate but very fast. AlphaGo: f accurate but slow.

Our Example, Revisited: Minimax With Depth Limit $d = 2$



- Blue: Evaluation function f, applied to the cut-off states at $d = 2$.
- Red: Utilities of inner nodes, as computed by Minimax using d, f.

Example Chess

Evaluation function in Chess:

Linear Evaluation Functions, Search Depth

Fast simple f: weighted linear function

$$w_1 f_1 + w_2 f_2 + \cdots + w_n f_n$$

where the w_i are the weights, and the f_i are the features.

How to obtain such functions?

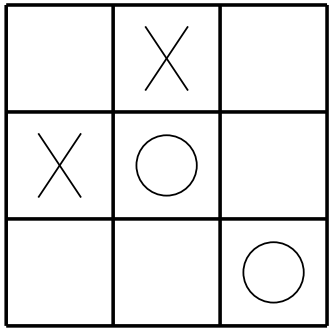
- Weights w_i can be learned automatically.
- The features f_i have to be designed by human experts.

And how deeply to search?

- Iterative deepening until time for move is up.
- Better: quiescence search, dynamically adapt depth limit, search deeper in "unquiet" positions (e.g. Chess piece exchange situations).

Questionnaire

Questionnaire, ctd.



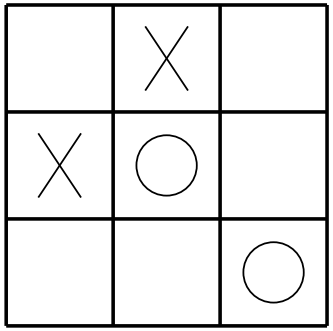
- Tic-Tac-Toe. Max = x, Min = o.
- Evaluation function $f_1(s)$: Number of rows, columns, and diagonals that contain AT LEAST ONE "x".
- (d: depth limit; I: initial state)

Question!

With $d = 3$ i.e. considering the moves Max-Min-Max, and using f_1, which moves may Minimax choose for Max in the initial state I?

(A): Middle. (B): Corner.

Questionnaire, ctd.



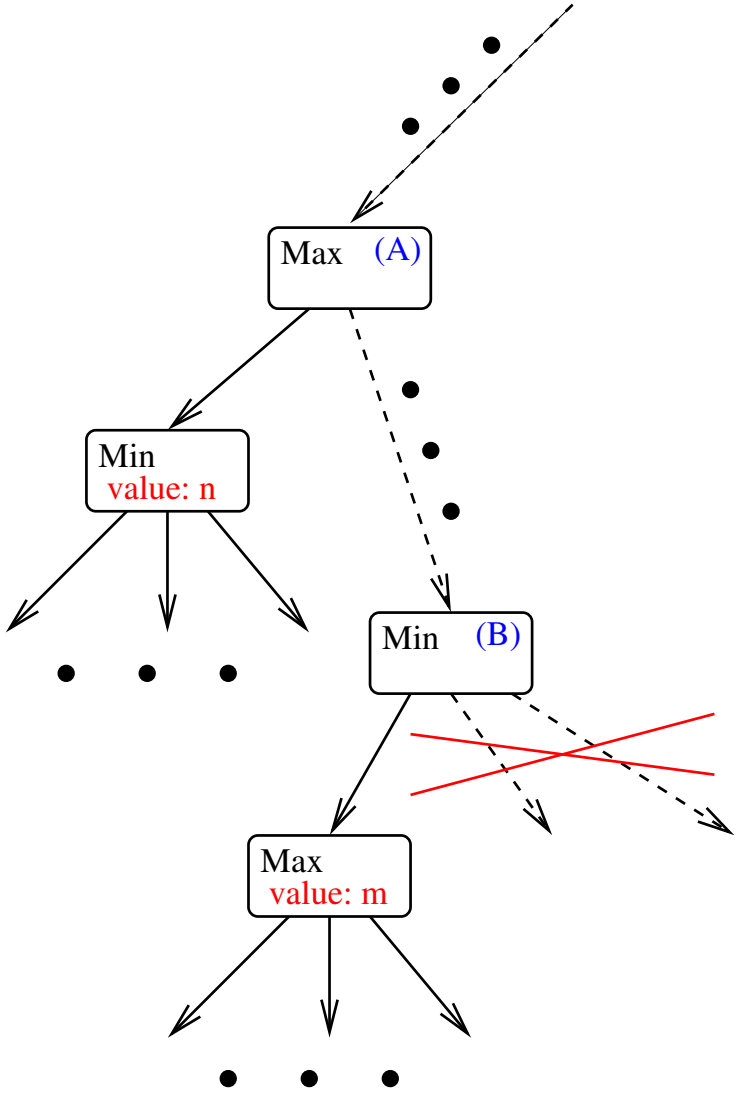
- Tic-Tac-Toe. Max = x, Min = o.
- Evaluation function $f_2(s)$: Number of rows, columns, and diagonals that contain AT LEAST TWO "x".
- (d: depth limit; I: initial state)

Question!

With $d = 3$ i.e. considering the moves Max-Min-Max, and using f_2, which moves may Minimax choose for Max in the initial state I?

(A): Middle. (B): Corner.

Alpha Pruning: Idea

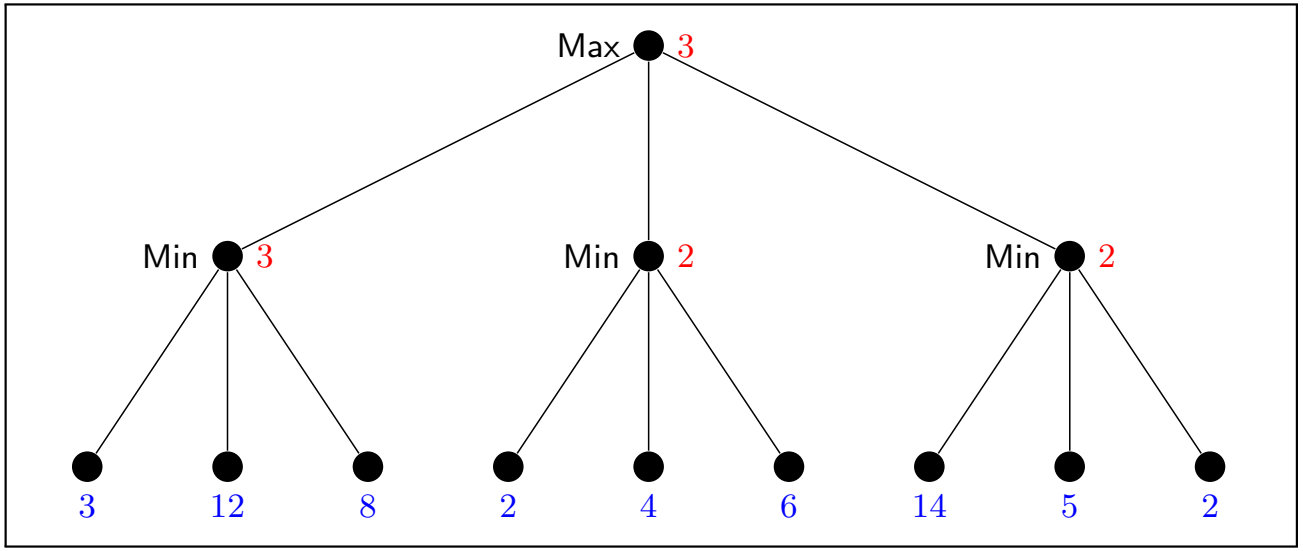


Say $n > m$**.**

Alpha Pruning: The Idea in Our Example

Question:

Can we save some work here?



Alpha Pruning

Alpha-Beta Pruning

Reminder:

- **What is α:** For each search node n, the highest Max-node utility that search has found already on its path to n.
- **How to use α:** In a Min node n, if one of the successors already has utility $\leq \alpha$, then stop considering n. (Pruning out its remaining successors.)

We can use a dual method for Min:

- **What is β:** For each search node n, the lowest Min-node utility that search has found already on its path to n.
- **How to use β:** In a Max node n, if one of the successors already has utility $\geq \beta$, then stop considering n. (Pruning out its remaining successors.)

... and of course we can use both together.

Alpha-Beta Search: Pseudo-Code

```
function Alpha-Beta-Search(s) returns an action
  v ← Max-Value(s, −∞, +∞)
  return an action a ∈ Actions(s) yielding value v

function Max-Value(s, α, β) returns a utility value
  if Terminal-Test(s) then return u(s)
  v ← −∞
  for each a ∈ Actions(s) do
     v ← max(v, Min-Value(ChildState(s, a), α, β))
     α ← max(α, v)
     if v ≥ β then return v /* Here: v ≥ β ⇔ α ≥ β */
  return v

function Min-Value(s, α, β) returns a utility value
  if Terminal-Test(s) then return u(s)
  v ← +∞
  for each a ∈ Actions(s) do
     v ← min(v, Max-Value(ChildState(s, a), α, β))
     β ← min(β, v)
     if v ≤ α then return v /* Here: v ≤ α ⇔ α ≥ β */
  return v
```

$=$ Minimax (slide 18) $+$ α/β book-keeping and pruning.

Alpha-Beta Search: Example

Alpha-Beta Search: Modified Example

How Much Pruning Do We Get?

→ Choosing best moves first yields most pruning in alpha-beta search.

With branching factor b and depth limit d:

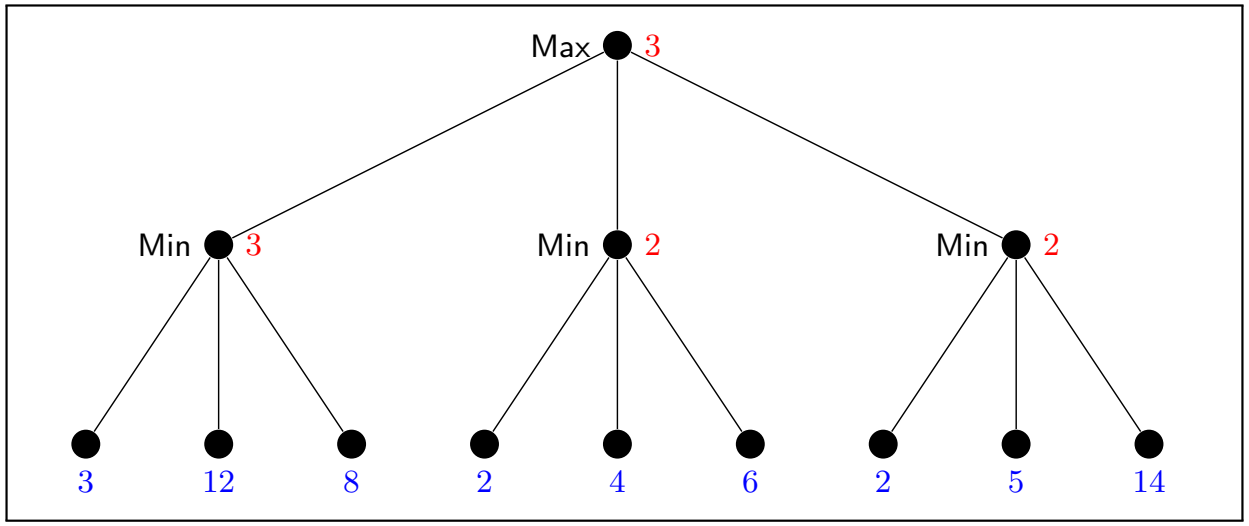
- Minimax: b^d nodes.
- Best case: Best moves first $\Rightarrow b^{d/2}$ nodes! Double the lookahead!
- Practice: Often possible to get close to best case.

Example Chess:

- Move ordering: Try captures first, then threats, then forward moves, then backward moves.
- Double lookahead: E.g. with time for 10^9 nodes, Minimax 3 rounds (white move, black move), alpha-beta 6 rounds.

Computer Chess State of the Art

Questionnaire



Question!

How many nodes does alpha-beta prune out here?

(A): 0 (B): 2

(C): 4 (D): 6

And now . . .

AlphaGo = Monte-Carlo tree search + neural networks

Limitations of Alpha Beta Search

Alpha Beta search is a strong algorithm but it has two issues (e.g. in Go):

Monte-Carlo Sampling

→ When deciding which action to take on game state s:

Imagine that each of the available actions is a slot machine that on average gives you an unknown reward:

- Explotation: play in the machine that returns the best reward
- Exploration: play machines that have not been tried a lot yet

Upper Confidence Bound (UCB): formula that automatically balances exploration and exploitation to maximize total gains

Monte-Carlo Sampling: Illustration

Monte-Carlo Tree Search

→ When deciding which action to take on game state s:

Monte-Carlo Sampling: Evaluate actions through sampling.

while time not up **do**
  select a transition $s \xrightarrow{a} s'$
  run a random sample from s' until terminal state t
  update, for a, average $u(t)$ and #expansions
return an a for s with maximal average $u(t)$

Monte-Carlo Tree Search: Maintain a search tree T.

while time not up **do**
  apply actions within T up to a state s' and $s' \xrightarrow{a'} s''$ s.t. $s'' \notin T$
  run random sample from s'' until terminal state t
  add s'' to T
  update, from a' up to root, averages $u(t)$ and #expansions
return an a for s with maximal average $u(t)$
When executing a, keep the part of T below a

→ Compared to alpha-beta search: no exhaustive enumeration. Pro: runtime & memory. Contra: need good guidance how to "select" and "sample".

Monte-Carlo Tree Search: Illustration

How to Guide the Search?

How to "sample"? What exactly is "random"?

- Exploitation: Prefer moves that have high average already (interesting regions of state space).
- Exploration: Prefer moves that have not been tried a lot yet (don't overlook other, possibly better, options).

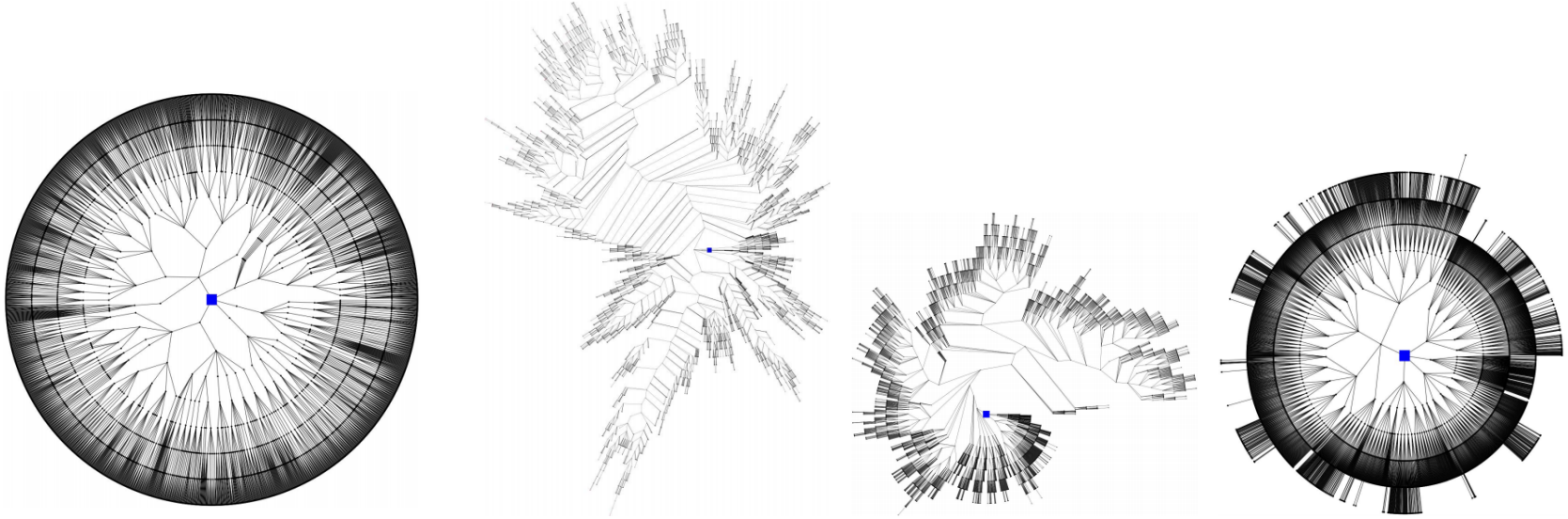
→ Classical formulation: balance exploitation vs. exploration.

UCT:

- "Upper Confidence bounds applied to Trees" [Kocsis and Szepesvári (2006)]. Inspired by Multi-Armed Bandit (as in: Casino) problems.
- Basically a formula defining the balance. Very popular (buzzword).
- Recent critics (e.g. [Feldman and Domshlak (2014)]): "Exploitation" in search is very different from the Casino, as the "accumulated rewards" are fictitious (we're merely thinking about the game, not actually playing and winning/losing all the time).

Alpha-beta versus UCT

Illustration from Ramanujan and Selman (2011) that visualizes the search space of Alpha Beta and three variants of UCT (more exploration or exploitation):



Alpha Beta UCT (from more exploitation to more exploration)

AlphaGo: Overview

Neural Networks:

- Policy networks: Given a state s, output a probability distibution over the actions applicable in s.
- Value networks: Given a state s, outpout a number estimating the game value of s.

Combination with MCTS:

- Policy networks bias the action choices within the MCTS tree (and hence the leaf-state selection), and bias the random samples.
- Value networks are an additional source of state values in the MCTS tree, along with the random samples.

$\rightarrow$ And now in a little more detail:

Summary

- Games (2-player turn-taking zero-sum discrete and finite games) can be understood as a simple extension of classical search problems.
- Each player tries to reach a terminal state with the best possible utility (maximal vs. minimal).
- Minimax searches the game depth-first, max'ing and min'ing at the respective turns of each player. It yields perfect play, but takes time $O(b^d)$ where b is the branching factor and d the search depth.
- Except in trivial games (Tic-Tac-Toe), Minimax needs a depth limit and apply an evaluation function to estimate the value of the cut-off states.
- Alpha-beta search remembers the best values achieved for each player elsewhere in the tree already, and prunes out sub-trees that won't be reached in the game.
- Monte-Carlo tree search (MCTS) samples game branches, and averages the findings. AlphaGo controls this using neural networks: evaluation function ("value network"), and action filter ("policy network").

Reading

- *Chapter 5: Adversarial Search*, Sections 5.1 – 5.4 [Russell and Norvig (2010)].

  Content: Section 5.1 corresponds to my "Introduction", Section 5.2 corresponds to my "Minimax Search", Section 5.3 corresponds to my "Alpha-Beta Search". I have tried to add some additional clarifying illustrations. RN gives many complementary explanations, nice as additional background reading.

  Section 5.4 corresponds to my "Evaluation Functions", but discusses additional aspects relating to narrowing the search and look-up from opening/termination databases. Nice as additional background reading.

  I suppose a discussion of MCTS and AlphaGo will be added to the next edition ...

References I

Zohar Feldman and Carmel Domshlak. Simple regret optimization in online planning for markov decision processes. *Journal of Artificial Intelligence Research*, 51:165–205, 2014.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer-Verlag, 2006.

Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS'11)*. AAAI Press, 2011.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Third Edition)*. Prentice-Hall, Englewood Cliffs, NJ, 2010.