Fondamenti di Intelligenza Artificiale

06. Adversarial Search

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

Prof Sara Bernardini
bernardini@diag.uniroma1.it
www.sara-bernardini.com

SAPIENZA
UNIVERSITÀ DI ROMA

Spring Term

Agenda

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 Mechanical Turk (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. They are 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, end state=terminal 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

Why Games are Hard to Solve

How To Describe a Game State Space?

→ Like for classical search problems, there are three possible ways to describe a game: blackbox/API description, declarative description, explicit game state space.

Which ones do humans use?

- Explicit ≈ Hand over a book with all 10^{40} game positions in Chess.
- Blackbox ≈ Give possible Chess moves on demand but don't say how they are generated.
- Declarative! With "game description language" = natural language.

Specialized vs. General Game Playing

And which game descriptions do computers use?

- Explicit: Only in illustrations.
- Blackbox/API: Assumed description in **This Chapter**.
  - Method of choice for almost all those game players out there in the market (Chess computers, video game opponents, you name it).
  - Programs designed for, and specialized to, a particular game.
  - Human knowledge is (was?) key: evaluation functions (see later), opening databases (Chess!), end databases.
- Declarative: General Game Playing, active area of research in AI.
  - Generic Game Description Language (GDL), based on logic.
  - Solvers are given only "the rules of the game", no other knowledge/input whatsoever (cf. **Chapter 2**).
  - Regular academic competitions since 2005.

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 (state of the art in Chess).

- **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 (state of the art in Go).

- **Stochastic Games:** What if is there an element of chance in the game?

  → How to adapt Minimax to stochastic games.

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."

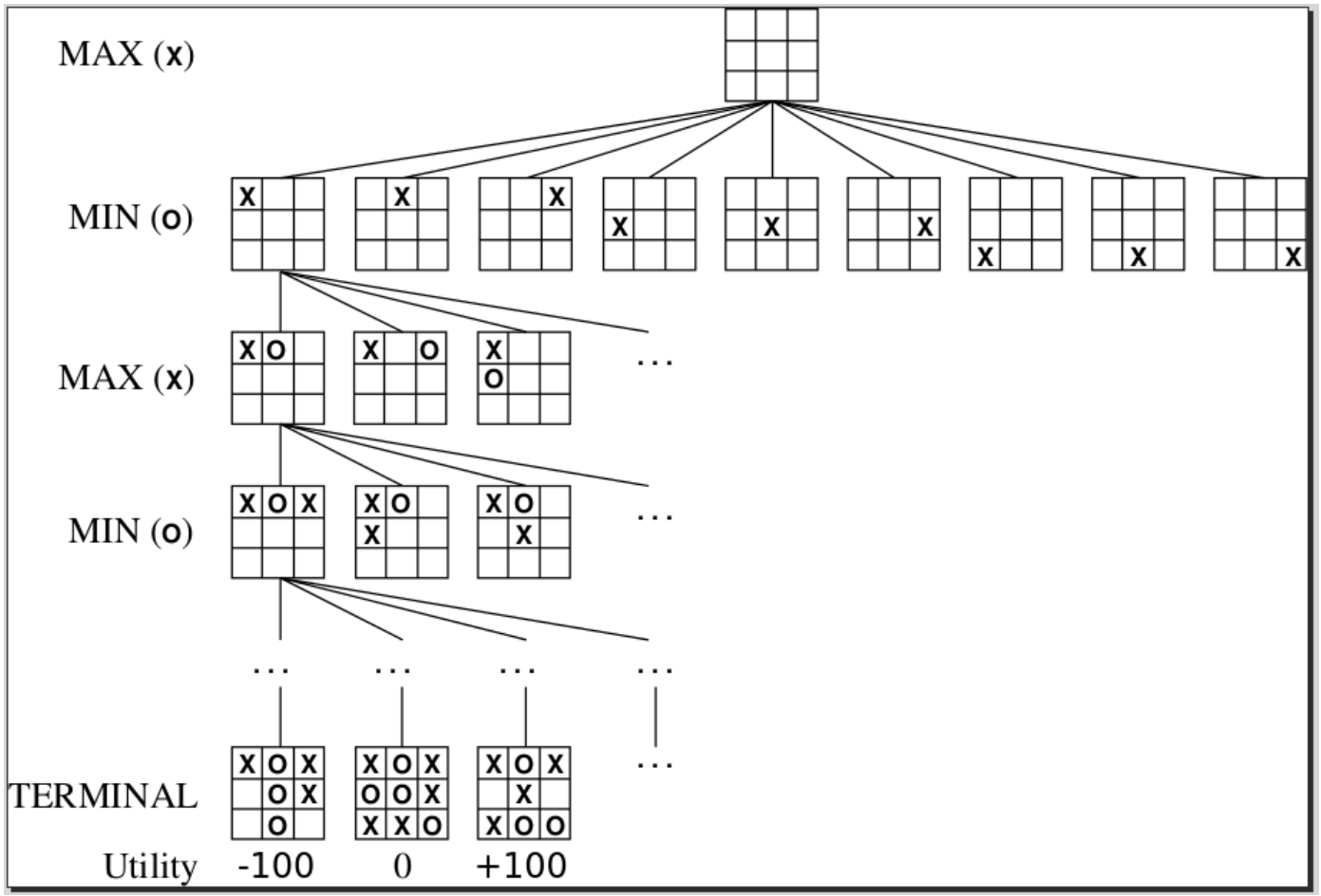
Remember:

- 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



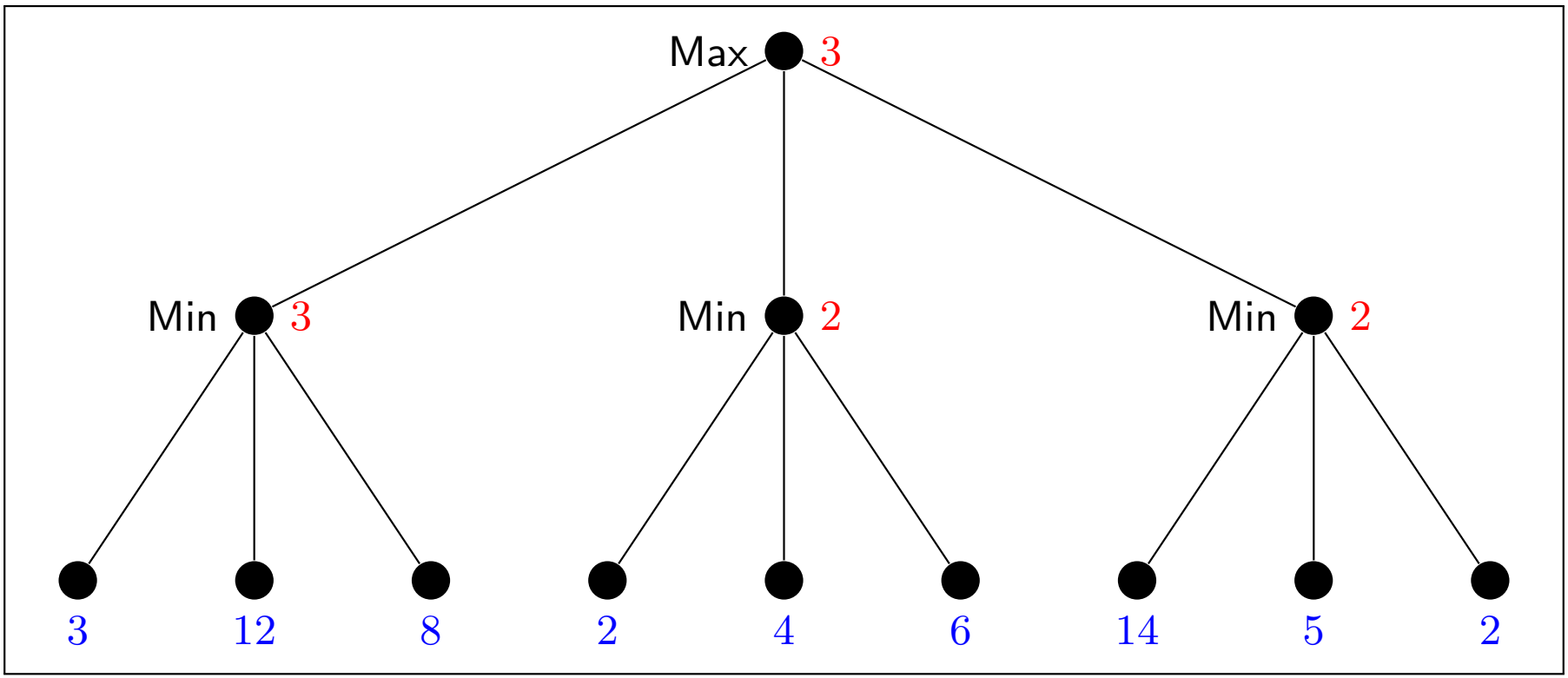
- Game tree, current player marked on the left.
- Last row: terminal positions with their utility.
- Upper bound on size of this tree?

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
  $v \leftarrow$ *Max-Value*(s)
  return an *action* $a \in$ *Actions(*s*)* yielding value v

function Max-Value(s) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow -\infty$
  for each $a \in$ *Actions(*s*)* **do**
    $v \leftarrow \max(v,$*Min-Value(ChildState*$(s,a)))$
  return v

function Min-Value(s) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow +\infty$
  for each $a \in$ *Actions(*s*)* **do**
    $v \leftarrow \min(v,$*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:

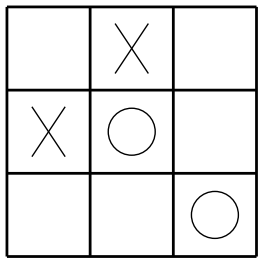
- Limit search depth, apply evaluation function to the cut-off states.
- Use alpha-beta pruning to reduce search.
- Don't search exhaustively; sample instead: MCTS.

Properties of Minimax

- Complete: Yes, if tree is finite (chess has specific rules for this).
- Optimal: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$ with b branching factor and m depth of the solution.
- Space complexity: $O(bm)$ (depth-first exploration).

For chess, $b \approx 35$, $m \approx 80$ for "reasonable" games. Considering duplicate states, the state space is 35^{80}.
$\rightarrow$ Exact solution completely infeasible!

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.

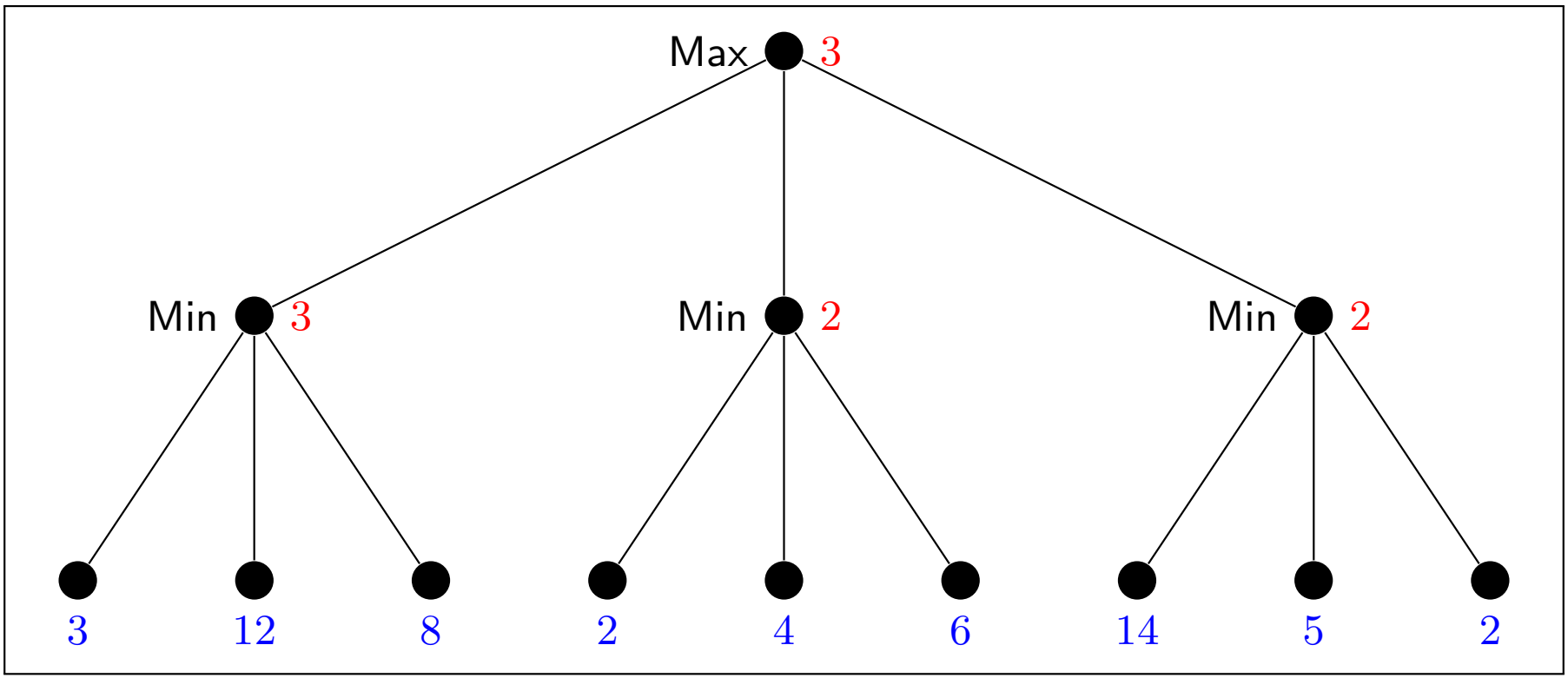
Solution: Impose a search depth limit ("horizon") d, and apply an evaluation function to the non-terminal cut-off states.

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

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

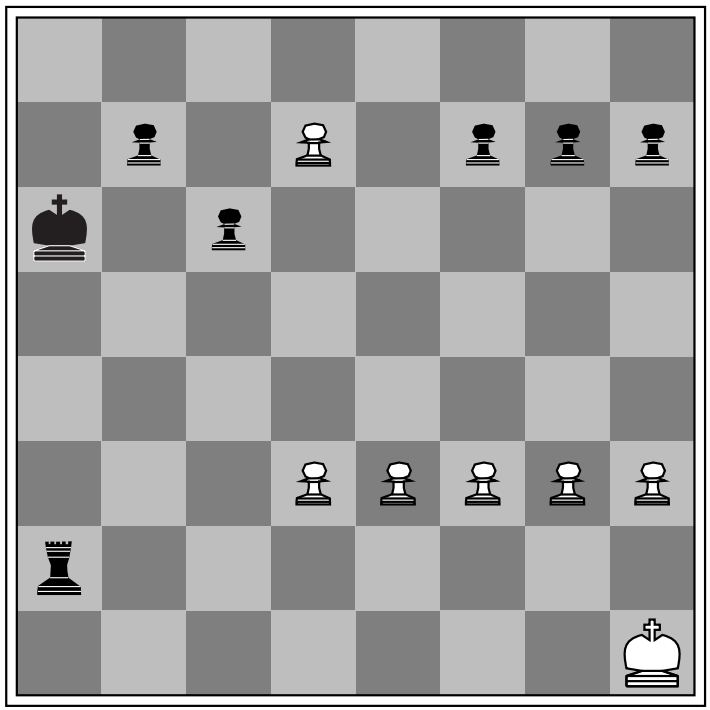
where the w_i are the weights, and the f_i are the features.

How to obtain such functions?

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

The Horizon Problem

Problem: Critical aspects of the game can be cut-off by the horizon.



Black to move

Who's gonna win here?

How Deeply to Search?

Want: In given time, search as deeply as possible.

Problem: Very difficult to predict search runtime.

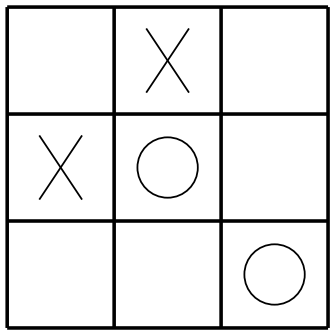
→ Solution? Iterative deepening.

- Search with depth limit $d = 1, 2, 3, \ldots$
- Time's up: Return result of deepest completed search.

Better solution: Quiescence search

- Dynamically adapted d.
- Search more deeply in "unquiet" positions.→ Value of evaluation function fluctuates quickly.
- Example Chess: Piece exchange situations ("you take mine, I take yours") are very unquiet ... $\Longrightarrow$ Keep searching until the end of the piece exchange is reached.

Questionnaire, ctd.



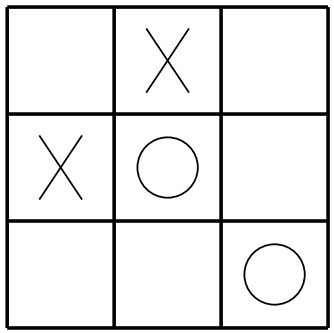
- Tic-Tac-Toe. Max = x, Min = o.
- Evaluation function $f_1(s)$: Number of rows, columns, and diagonals that contain AT LEAST ONE "x".
- (d: depth limit; I: initial state)

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.



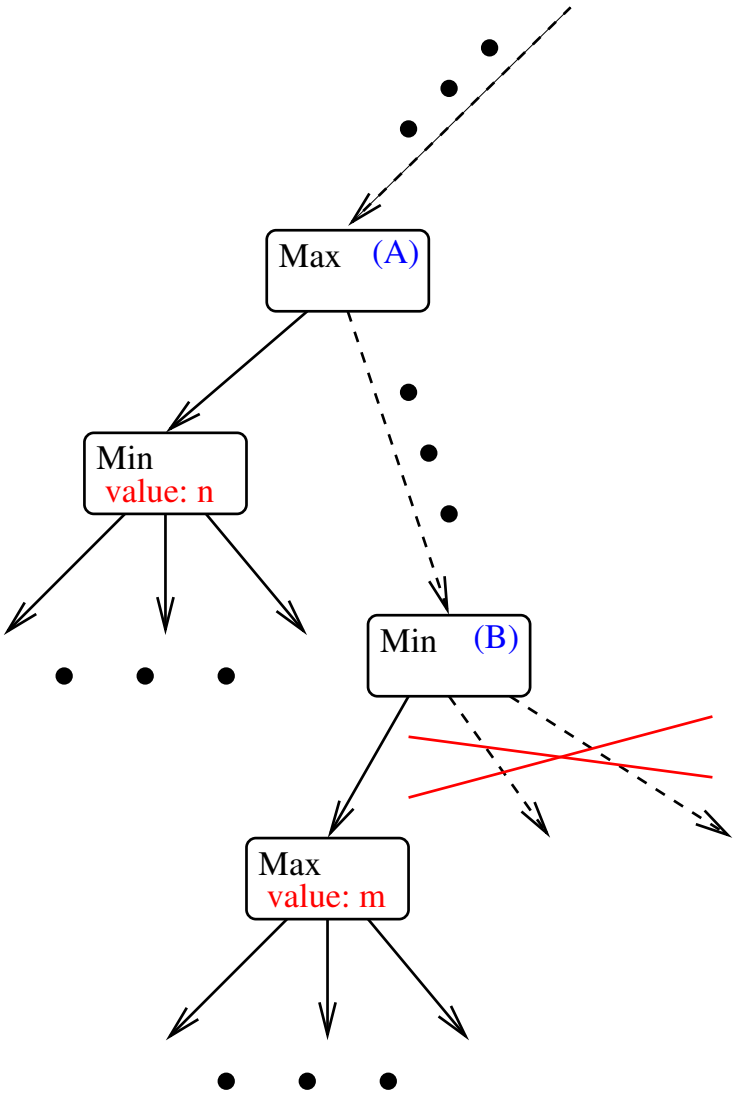
- Tic-Tac-Toe. Max = x, Min = o.
- Evaluation function $f_2(s)$: Number of rows, columns, and diagonals that contain AT LEAST TWO "x".
- (d: depth limit; I: initial state)

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

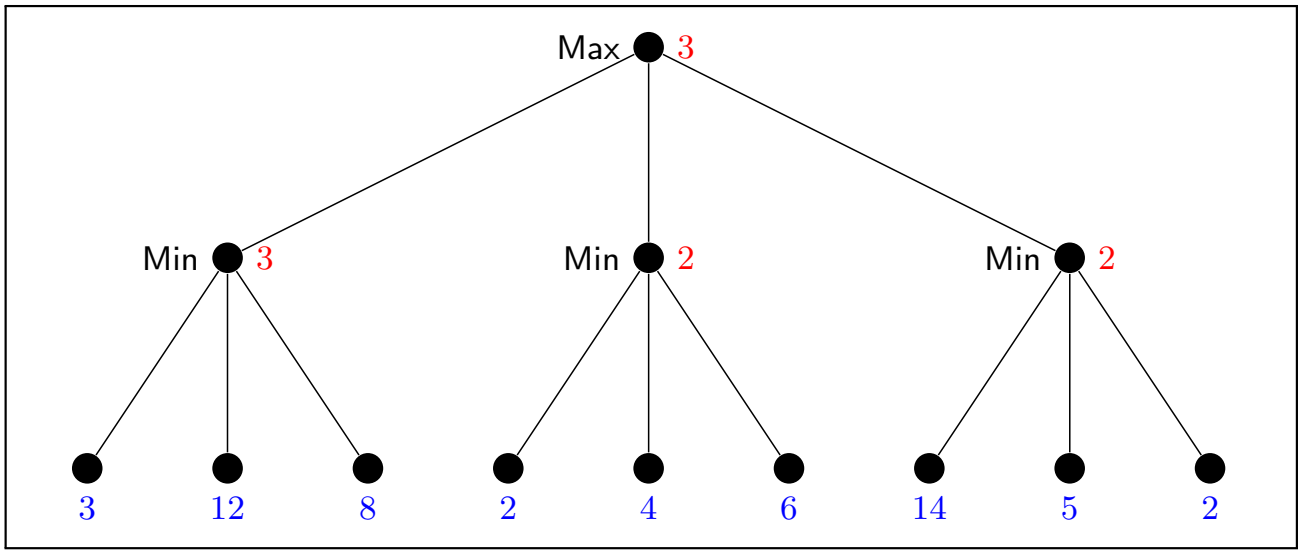


Say $n > m$.

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, α is the highest utility that search has already found for Max on its path to n. Think: "at least".
- **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, β is the lowest utility that search has already found for Min on its path to n. Think: "at most".
- **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
  $v \leftarrow$ *Max-Value*$(s, -\infty, +\infty)$
  return an *action* $a \in$ *Actions(s)* yielding value v

function Max-Value(s, α, β) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow -\infty$
  for each $a \in$ *Actions(s)* **do**
    $v \leftarrow \max(v,$*Min-Value*$($*ChildState*$(s, a), \alpha, \beta))$
    $\alpha \leftarrow \max(\alpha, v)$
    if $v \geq \beta$ **then return** v /* Here: $v \geq \beta \Leftrightarrow \alpha \geq \beta$ */
  return v

function Min-Value(s, α, β) **returns** a utility value
  if *Terminal-Test*(s) **then return** $u(s)$
  $v \leftarrow +\infty$
  for each $a \in$ *Actions(s)* **do**
    $v \leftarrow \min(v,$*Max-Value*$($*ChildState*$(s, a), \alpha, \beta))$
    $\beta \leftarrow \min(\beta, v)$
    if $v \leq \alpha$ **then return** v /* Here: $v \leq \alpha \Leftrightarrow \alpha \geq \beta$ */
  return v

$=$ Minimax (slide 21) $+$ 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 m:

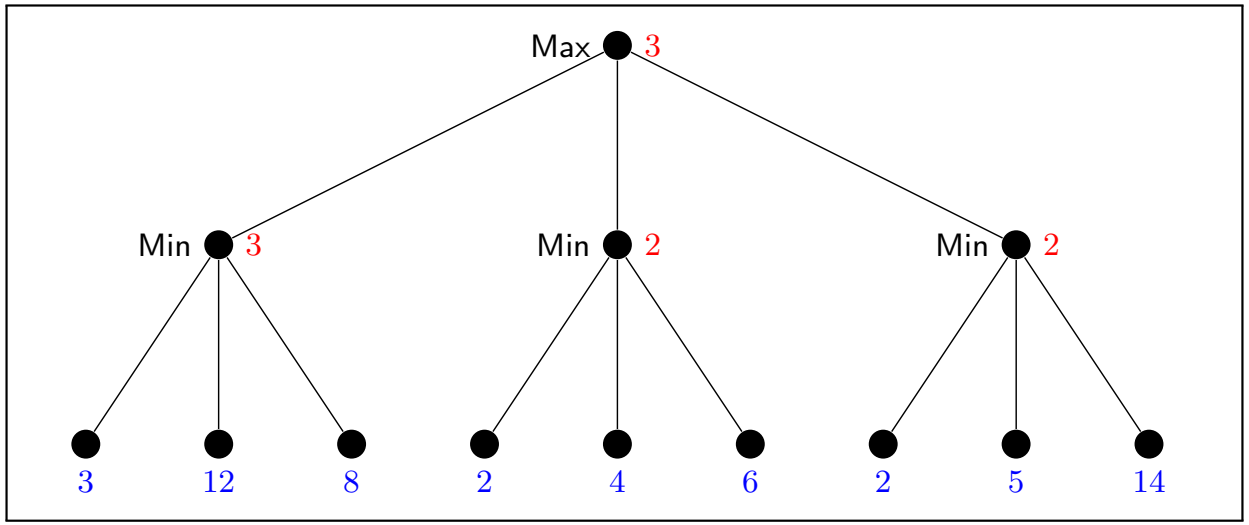
- Minimax: b^m nodes.
- Best case: Best moves first $\Rightarrow b^{m/2}$ nodes! Double the lookahead!
- Practice: Often possible to get close to best case.

Example Chess:

- Move ordering: Try captures and checks 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 Tree Search: Underlying Principles

- How to estimate the value of a state? Average utility over a number of simulations of complete games starting from the state (playout).
- How do we choose what moves to make during the playout? A **playout policy** needs to bias the moves towards good ones.
  → For Go and other games, playout policies have been successfully learned from self-play by using neural networks.
- From what positions do we start the playouts, and how many playouts do we allocate to each position?
  → Pure Monte Carlo Search does simulations starting from the current state of the game and tracks which of the possible moves has the highest win percentage.
  → **Selection policy** focuses the computation on important parts of the game tree, balancing exploration and exploitation.

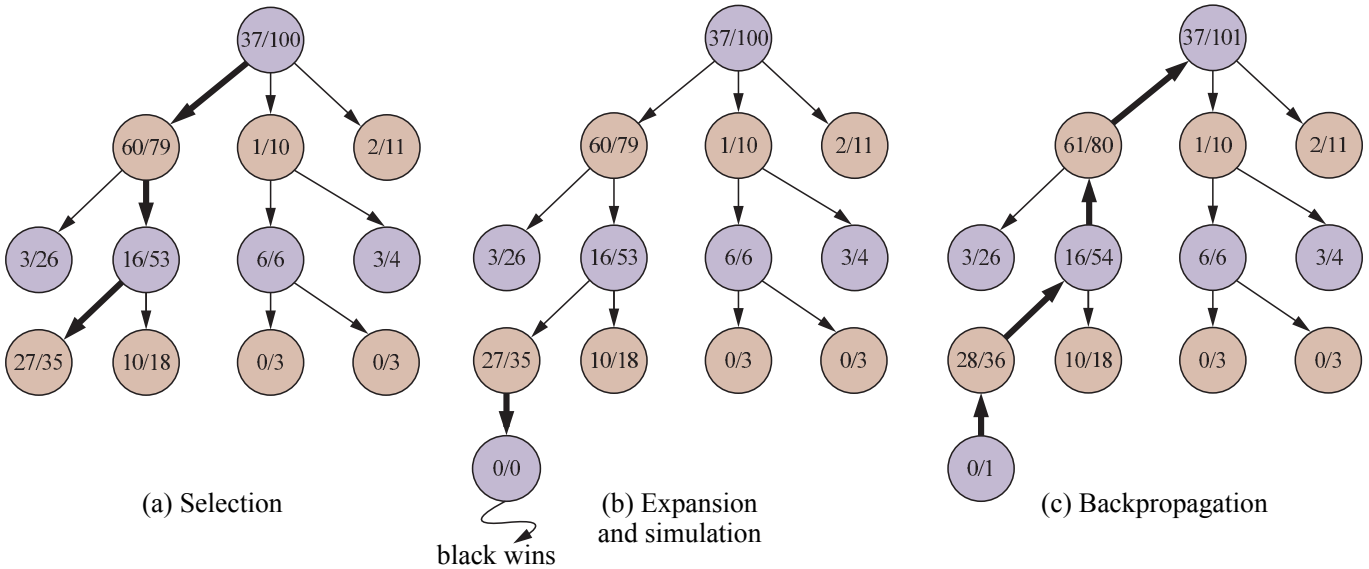
Monte-Carlo Tree Search: 4 Steps

Monte-Carlo Tree Search repeatedly follows the following steps:

1. Selection: traverse the tree starting at the root, applying the selection policy to choose successors until reaching a leaf node that has not been fully expanded.
2. Expansion: Grow tree by generating a new child of the leaf node by applying a new action.
3. Simulation: From the newly generated child node, perform a run of the game, selecting moves for both players according to the playout policy, to obtain the final reward.
4. Back-propagation: Use the result of the simulation to update all the search tree nodes going up to the root.

When time to decide is over, choose the "best" move and play it.

→There are multiple versions of MCTS depending on what strategies are followed in each of the steps.



Root represents a state where white has just moved (white has won 37 out of the 100 playouts done so far.)

(a) We select moves, all the way down the tree, ending at the leaf node marked 27/35 (for 27 wins for black out of 35 playouts).

(b) We expand the selected node and do a simulation (playout), which ends in a win for black.

(c) The results of the simulation are back-propagated up the tree.

How to Guide the Search?: Selection Rule

How to "sample"? Selection Rule

- 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.

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:

Monte-Carlo Sampling: Evaluate actions through sampling.

- Exploitation: 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.

Upper Confidence bounds applied to Trees

- UCT [Kocsis and Szepesvári (2006)]. Inspired by Multi-Armed Bandit problems.
- A formula defining the balance. Very popular (buzzword).

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{logN(PARENT(n))}{N(n)}}$$

$U(n) =$ total utility of all playouts that went through node n
$N(n) =$ number of playouts through node n
$PARENT(n)$ is the parent node of n in the tree

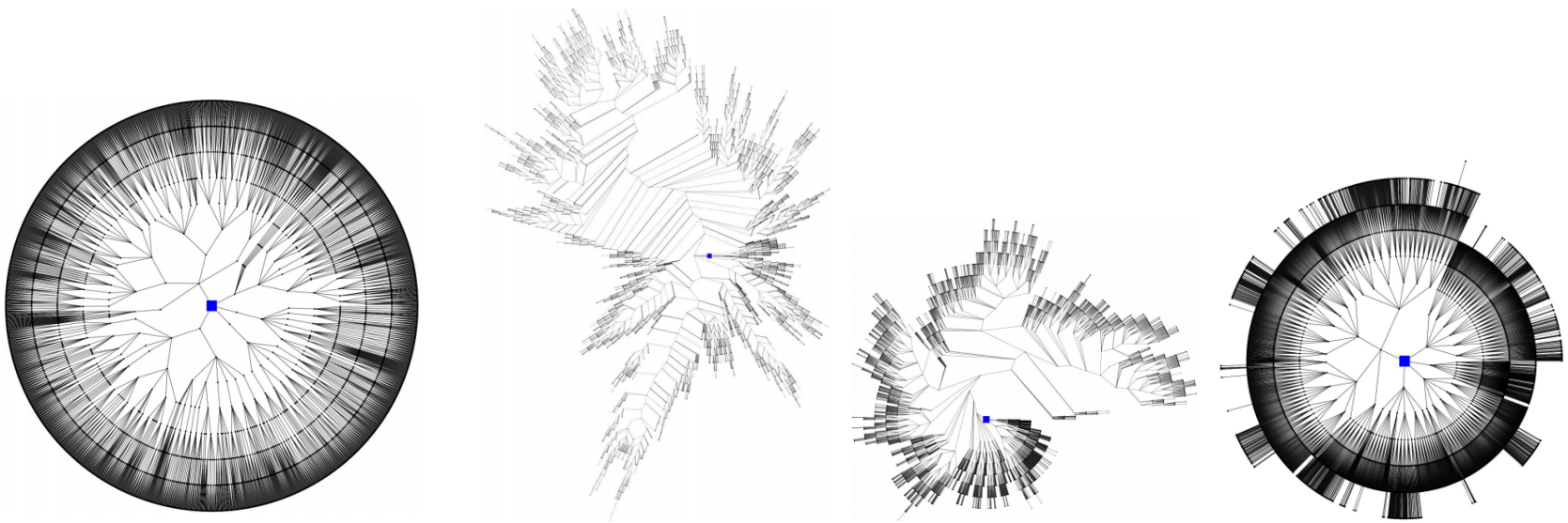
Monte-Carlo Tree Search: Algorithm

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

- A game tree (*tree*) is initialized.
- Cycle SELECT / EXPAND / SIMULATE / BACK-PROPAGATE is repeated until the algorithm runs out of time.
- Move that led to node with highest number of playouts is returned.
- UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage.

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):

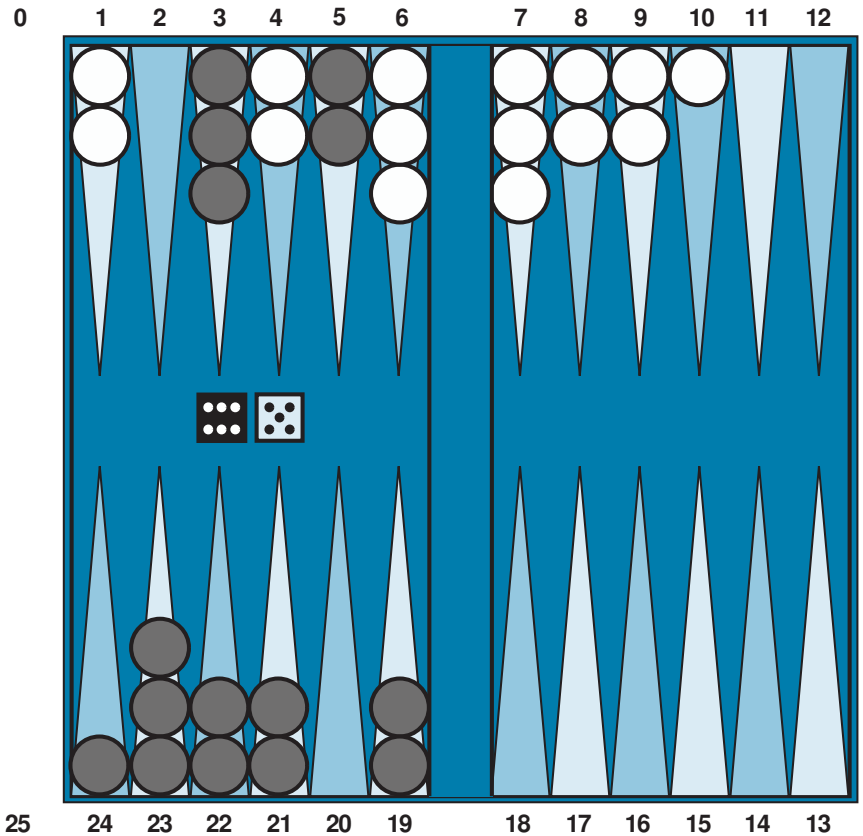


Alpha Beta UCT (from more exploitation to more exploration)

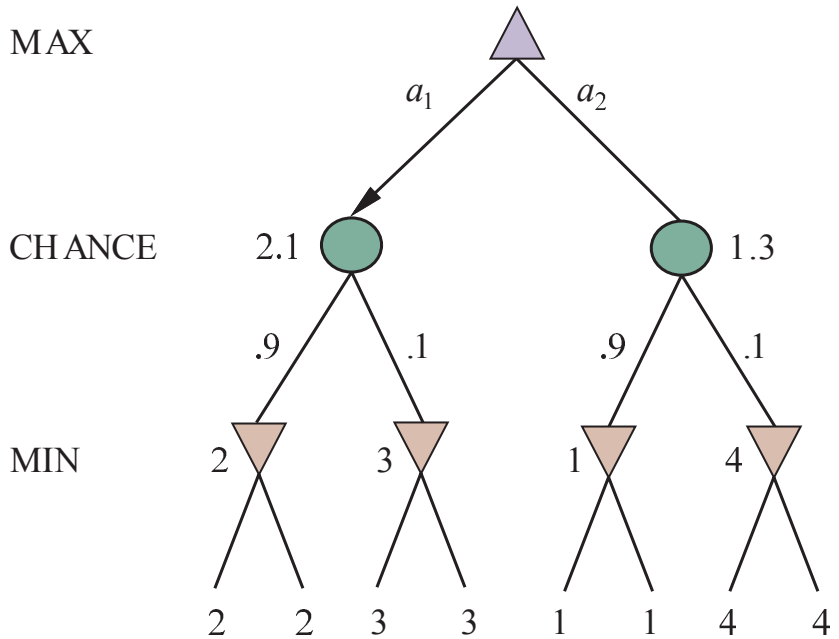
Monte-Carlo Tree Search: Conclusions

- MTCS is chosen when branching factor becomes very high and the evaluation function is difficult.
- MTCS is more robust than Alpha-Beta search (it is also possible to combine MCTS and evaluation functions).
- MTCS can be applied to unknown games.
- MTCS is a form of reinforcement learning.

Backgammon



Stochastic Games includes a random element, e.g. throwing of dice. They combine luck and skill.



- A game tree in backgammon must include chance nodes in addition to MAX and MIN nodes (circles).
- We can only calculate the expected value of a position: the average over all possible outcomes of the chance nodes.
- We can generalize the minimax value for deterministic game, expectiminimax value.

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.
- 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").
- For stochastic games, we can generalize minimax to expectiminimax.

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.

  Last edition of the RN (4th edition) contains a description of MCTS in Chapter 5.4.

References I

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.