

Foundations of Artificial Intelligence

Chapter 06: **Adversarial Search and Games**

Roberto Sebastiani

DISI, Università di Trento, Italy – `roberto.sebastiani@unitn.it`

`https://disi.unitn.it/rseba/DIDATTICA/fai\_2026/`

Teaching assistant: **Nicola Debole**, `nicola.debole@unitn.it`,

M.S. Course “Artificial Intelligence Systems”, academic year 2026-2027

Last update: Monday 7th September, 2026, 15:23

Copyright notice: Most examples and images displayed in the slides of this course are taken from [Russell & Norvig, “Artificial Intelligence, a Modern Approach”, 3rd and 4th ed., Pearson], including explicitly figures from the above-mentioned books, so that their copyright is detained by the authors. A few other material (text, figures, examples) is authored by (in alphabetical order): Pieter Abbeel, Bonnie J. Dorr, Anca Dragan, Dan Klein, Nikita Kitaev, Tom Lenaerts, Michela Milano, Dana Nau, Maria Simi, who detain its copyright.

These slides cannot be displayed in public without the permission of the author.

- 1 Games
- 2 Optimal Decisions in Games
 - Min-Max Search
 - Alpha-Beta Pruning
- 3 Adversarial Search with Resource Limits
- 4 Stochastic Games

- 1 Games
- 2 Optimal Decisions in Games
 - Min-Max Search
 - Alpha-Beta Pruning
- 3 Adversarial Search with Resource Limits
- 4 Stochastic Games

Games and AI

- Games are a form of **multi-agent environment**
 - Q.: **What do other agents do and how do they affect our success?**
 - recall: cooperative vs. competitive multi-agent environments
 - competitive multi-agent environments give rise to **adversarial problems** (aka **games**)
- Q.: **Why study games in AI?**
 - lots of fun, historically entertaining
 - **easy to represent**: agents restricted to **small number of actions** with **precise rules**
 - interesting also because **computationally very hard**
(ex: chess has $b \approx 35$, $\#nodes \approx 10^{40}$)
 - metaphor for important application domains
(e.g. competitive markets, life sciences, sport, politics, warfare, ...)

Games and AI

- Games are a form of **multi-agent environment**
 - Q.: **What do other agents do and how do they affect our success?**
 - recall: cooperative vs. competitive multi-agent environments
 - competitive multi-agent environments give rise to **adversarial problems** (aka **games**)
- Q.: **Why study games in AI?**
 - lots of fun, historically entertaining
 - **easy to represent**: agents restricted to **small number of actions** with **precise rules**
 - interesting also because **computationally very hard**
(ex: **chess has $b \approx 35$, $\#nodes \approx 10^{40}$**)
 - metaphor for important application domains
(e.g. competitive markets, life sciences, sport, politics, warfare, ...)

Search and Games

- Search (with no adversary)

- solution is a (heuristic) method for finding a goal
- heuristics techniques can find optimal solutions
- evaluation function: estimate of cost from start to goal through given node
- examples: path planning, scheduling activities, ...

- Games (with adversary), aka adversarial search

- solution is a strategy: specifies a move for every possible opponent reply
- evaluation function (utility): evaluate “goodness” of game position
- examples: tic-tac-toe, chess, checkers, Othello, backgammon, ...
- often computationally very hard $\implies$ time limits force an approximate solution

Search and Games

- Search (with no adversary)
 - solution is a (heuristic) method for finding a goal
 - heuristics techniques can find optimal solutions
 - evaluation function: estimate of cost from start to goal through given node
 - examples: path planning, scheduling activities, ...
- Games (with adversary), aka adversarial search
 - solution is a strategy: specifies a move for every possible opponent reply
 - evaluation function (utility): evaluate “goodness” of game position
 - examples: tic-tac-toe, chess, checkers, Othello, backgammon, ...
 - often computationally very hard $\implies$ time limits force an approximate solution

Types of Games

- Many different kinds of games
- Relevant features:
 - deterministic vs. stochastic (with chance)
 - one, two, or more players
 - zero-sum vs. general games
 - perfect information (can you see the state?) vs. imperfect
- Most common: deterministic, turn-taking, two-player, zero-sum games, perfect information
- Want algorithms for calculating a strategy (aka policy):
 - recommends a move from each state: $policy : S \mapsto A$

(*) "blind tictactoe": a version of tic-tac-toe where the players don't get to see each others' moves.

Types of Games

- Many different kinds of games
- Relevant features:
 - deterministic vs. stochastic (with chance)
 - one, two, or more players
 - zero-sum vs. general games
 - perfect information (can you see the state?) vs. imperfect
- Most common: deterministic, turn-taking, two-player, zero-sum games, perfect information
- Want algorithms for calculating a strategy (aka policy):
 - recommends a move from each state: $policy : S \mapsto A$

(*) "blind tictactoe": a version of tic-tac-toe where the players don't get to see each others' moves.

Types of Games

- Many different kinds of games
- Relevant features:
 - deterministic vs. stochastic (with chance)
 - one, two, or more players
 - zero-sum vs. general games
 - perfect information (can you see the state?) vs. imperfect
- Most common: deterministic, turn-taking, two-player, zero-sum games, perfect information
- Want algorithms for calculating a strategy (aka policy):
 - recommends a move from each state: $policy : S \mapsto A$

(*) "blind tictactoe": a version of tic-tac-toe where the players don't get to see each others' moves.

Types of Games

- Many different kinds of games
- Relevant features:
 - deterministic vs. stochastic (with chance)
 - one, two, or more players
 - zero-sum vs. general games
 - perfect information (can you see the state?) vs. imperfect
- Most common: deterministic, turn-taking, two-player, zero-sum games, perfect information
- Want algorithms for calculating a strategy (aka policy):
 - recommends a move from each state: $policy : S \mapsto A$

(*) "blind tictactoe": a version of tic-tac-toe where the players don't get to see each others' moves.

Types of Games

- Many different kinds of games
- Relevant features:
 - deterministic vs. stochastic (with chance)
 - one, two, or more players
 - zero-sum vs. general games
 - perfect information (can you see the state?) vs. imperfect
- Most common: deterministic, turn-taking, two-player, zero-sum games, perfect information
- Want algorithms for calculating a strategy (aka policy):
 - recommends a move from each state: $policy : S \mapsto A$

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

(*) "blind tictactoe": a version of tic-tac-toe where the players don't get to see each others' moves.

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- A game is a kind of search problem:
 - Initial state S_0 : specifies how the game is set up at the start
 - $Player(s)$: defines which player has the move in a state
 - $Actions(s)$: returns the set of legal moves in a state
 - $Result(s, a)$: the transition model, defines the result of a move
 - $TerminalTest(s)$: true iff the game is over (if so, s terminal state)
 - $Utility(s, p)$: (aka objective function or payoff function):
defines the final numeric value for a game ending in state s for player p
 - ex: chess: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: tic-tac-toe: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the game tree
 - nodes are states, arcs are actions
 - ex: tic-tac-toe: $\approx 10^5$ nodes, chess: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - Initial state S_0 : specifies how the game is set up at the start
 - $Player(s)$: defines which player has the move in a state
 - $Actions(s)$: returns the set of legal moves in a state
 - $Result(s, a)$: the **transition model**, defines the result of a move
 - $TerminalTest(s)$: true iff the game is over (if so, s **terminal state**)
 - $Utility(s, p)$: (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: chess: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: tic-tac-toe: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: tic-tac-toe: $\approx 10^5$ nodes, chess: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - **Initial state S_0** : specifies how the game is set up at the start
 - *Player(s)*: defines which player has the move in a state
 - *Actions(s)*: returns the set of legal moves in a state
 - *Result(s, a)*: the **transition model**, defines the result of a move
 - *TerminalTest(s)*: true iff the game is over (if so, s **terminal state**)
 - *Utility(s, p)*: (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: chess: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: tic-tac-toe: 1 (win), -1 (loss), 0 (draw)
- S_0 , *Actions(s)* and *Result(s, a)* recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: tic-tac-toe: $\approx 10^5$ nodes, chess: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - **Initial state S_0** : specifies how the game is set up at the start
 - **$Player(s)$** : defines which player has the move in a state
 - **$Actions(s)$** : returns the set of legal moves in a state
 - **$Result(s, a)$** : the **transition model**, defines the result of a move
 - **$TerminalTest(s)$** : true iff the game is over (if so, s **terminal state**)
 - **$Utility(s, p)$** : (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: chess: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: tic-tac-toe: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: tic-tac-toe: $\approx 10^5$ nodes, chess: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - **Initial state S_0** : specifies how the game is set up at the start
 - **$Player(s)$** : defines which player has the move in a state
 - **$Actions(s)$** : returns the set of legal moves in a state
 - **$Result(s, a)$** : the **transition model**, defines the result of a move
 - **$TerminalTest(s)$** : true iff the game is over (if so, s **terminal state**)
 - **$Utility(s, p)$** : (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: chess: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: tic-tac-toe: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: tic-tac-toe: $\approx 10^5$ nodes, chess: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - **Initial state S_0** : specifies how the game is set up at the start
 - **$Player(s)$** : defines which player has the move in a state
 - **$Actions(s)$** : returns the set of legal moves in a state
 - **$Result(s, a)$** : the **transition model**, defines the result of a move
 - **$TerminalTest(s)$** : true iff the game is over (if so, s **terminal state**)
 - **$Utility(s, p)$** : (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: chess: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: tic-tac-toe: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: tic-tac-toe: $\approx 10^5$ nodes, chess: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - **Initial state S_0** : specifies how the game is set up at the start
 - **$Player(s)$** : defines which player has the move in a state
 - **$Actions(s)$** : returns the set of legal moves in a state
 - **$Result(s, a)$** : the **transition model**, defines the result of a move
 - **$TerminalTest(s)$** : true iff the game is over (if so, s **terminal state**)
 - **$Utility(s, p)$** : (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: chess: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: tic-tac-toe: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: tic-tac-toe: $\approx 10^5$ nodes, chess: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

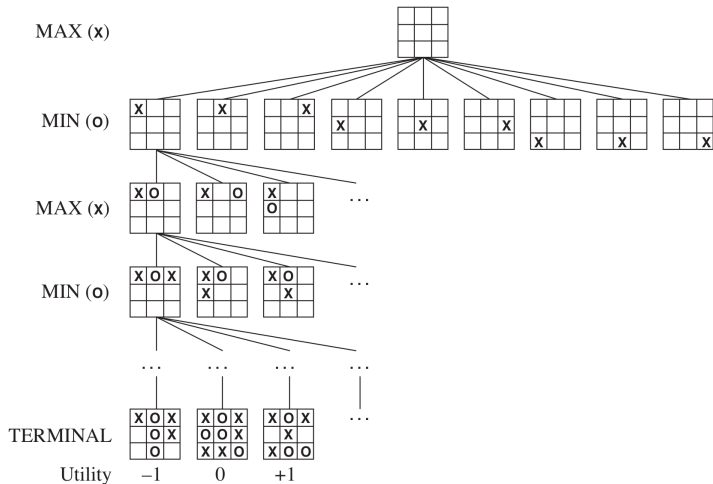
- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - **Initial state S_0** : specifies how the game is set up at the start
 - **$Player(s)$** : defines which player has the move in a state
 - **$Actions(s)$** : returns the set of legal moves in a state
 - **$Result(s, a)$** : the **transition model**, defines the result of a move
 - **$TerminalTest(s)$** : true iff the game is over (if so, s **terminal state**)
 - **$Utility(s, p)$** : (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: **chess**: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: **tic-tac-toe**: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: **tic-tac-toe**: $\approx 10^5$ nodes, **chess**: $\approx 10^{40}$ nodes, ...

Games: Main Concepts

- We first consider games with **two players**: “MAX” and “MIN”
 - MAX moves first;
 - they take turns moving until the game is over
 - at the end of the game, points are awarded to the winner and penalties are given to the loser
- **A game is a kind of search problem:**
 - **Initial state S_0** : specifies how the game is set up at the start
 - **$Player(s)$** : defines which player has the move in a state
 - **$Actions(s)$** : returns the set of legal moves in a state
 - **$Result(s, a)$** : the **transition model**, defines the result of a move
 - **$TerminalTest(s)$** : true iff the game is over (if so, s **terminal state**)
 - **$Utility(s, p)$** : (aka **objective function** or **payoff function**):
defines the final numeric value for a game ending in state s for player p
 - ex: **chess**: 1 (win), 0 (loss), $\frac{1}{2}$ (draw)
 - ex: **tic-tac-toe**: 1 (win), -1 (loss), 0 (draw)
- S_0 , $Actions(s)$ and $Result(s, a)$ recursively define the **game tree**
 - nodes are states, arcs are actions
 - ex: **tic-tac-toe**: $\approx 10^5$ nodes, **chess**: $\approx 10^{40}$ nodes, ...

Game Tree: Example

Partial game tree for tic-tac-toe (2-player, deterministic, turn-taking)



Zero-Sum Games vs. General Games

- General Games

- agents have independent utilities
- cooperation, indifference, competition, and more are all possible

- Zero-Sum Games: the total payoff to all players is the same for each game instance

- adversarial, pure competition
- agents have opposite utilities (values on outcomes)

⇒ Idea: With two-player zero-sum games, we can use one single utility value

- one agent maximizes it, the other minimizes it

⇒ optimal adversarial search as min-max search

Zero-Sum Games vs. General Games

- General Games

- agents have independent utilities
- cooperation, indifference, competition, and more are all possible

- Zero-Sum Games: the total payoff to all players is the same for each game instance

- adversarial, pure competition
- agents have opposite utilities (values on outcomes)

⇒ Idea: With two-player zero-sum games, we can use one single utility value

- one agent maximizes it, the other minimizes it

⇒ optimal adversarial search as min-max search

Zero-Sum Games vs. General Games

- General Games

- agents have independent utilities
- cooperation, indifference, competition, and more are all possible

- Zero-Sum Games: the total payoff to all players is the same for each game instance

- adversarial, pure competition
- agents have opposite utilities (values on outcomes)

⇒ Idea: With two-player zero-sum games, we can use one single utility value

- one agent maximizes it, the other minimizes it

⇒ optimal adversarial search as min-max search

Outline

- 1 Games
- 2 Optimal Decisions in Games**
 - Min-Max Search
 - Alpha-Beta Pruning
- 3 Adversarial Search with Resource Limits
- 4 Stochastic Games

Outline

- 1 Games
- 2 Optimal Decisions in Games
 - Min-Max Search
 - Alpha-Beta Pruning
- 3 Adversarial Search with Resource Limits
- 4 Stochastic Games

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $Minimax(s)$ returns the highest value

$$Minimax(s) \stackrel{\text{def}}{=} \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ \max_{a \in Actions(s)} Minimax(Result(s, a)) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} Minimax(Result(s, a)) & \text{if } Player(s) = MIN \end{cases}$$

Adversarial Search as Min-Max Search

- Assume MAX and MIN are very smart and always play optimally
- MAX must find a contingent strategy specifying:
 - MAX's move in the initial state
 - MAX's moves in the states resulting from every possible response by MIN,
 - MAX's moves in the states resulting from every possible response by MIN to those moves,
 - ...

(a single-agent move is called half-move or ply)

- Analogous to the AND-OR search algorithm
 - MAX playing the role of OR
 - MIN playing the role of AND
- Optimal strategy: for which $\text{Minimax}(s)$ returns the highest value

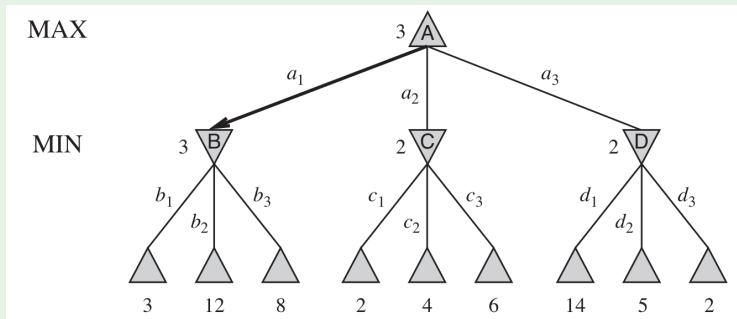
$$\text{Minimax}(s) \stackrel{\text{def}}{=} \begin{cases} \text{Utility}(s) & \text{if } \text{TerminalTest}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s, a)) & \text{if } \text{Player}(s) = \text{MIN} \end{cases}$$

Min-Max Search: Example

A two-ply game tree

- Δ nodes are “MAX nodes”, ∇ nodes are “MIN nodes”,
 - terminal nodes show the utility values for MAX
 - the other nodes are labeled with their minimax value
- Minimax maximizes the worst-case outcome for MAX

⇒ MAX's root best move is a_1

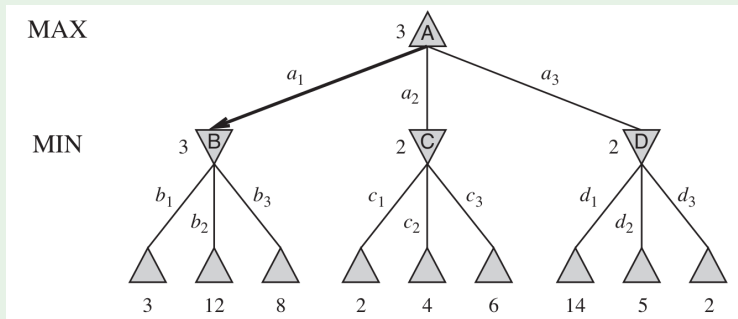


Min-Max Search: Example

A two-ply game tree

- Δ nodes are “MAX nodes”, ∇ nodes are “MIN nodes”,
 - terminal nodes show the utility values for MAX
 - the other nodes are labeled with their minimax value
- Minimax maximizes the worst-case outcome for MAX

⇒ MAX's root best move is a_1

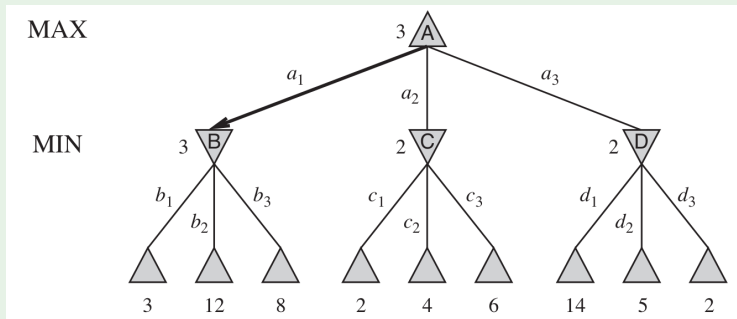


Min-Max Search: Example

A two-ply game tree

- Δ nodes are “MAX nodes”, ∇ nodes are “MIN nodes”,
 - terminal nodes show the utility values for MAX
 - the other nodes are labeled with their minimax value
- Minimax maximizes the worst-case outcome for MAX

⇒ MAX's root best move is a_1



The Minimax Algorithm

Depth-First Search Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action  
  return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$ 
```

```
function MAX-VALUE(state) returns a utility value  
  if TERMINAL-TEST(state) then return UTILITY(state)  
   $v \leftarrow -\infty$   
  for each a in ACTIONS(state) do  
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$   
  return v
```

```
function MIN-VALUE(state) returns a utility value  
  if TERMINAL-TEST(state) then return UTILITY(state)  
   $v \leftarrow \infty$   
  for each a in ACTIONS(state) do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$   
  return v
```


Remark

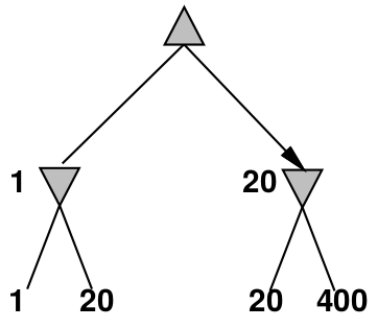
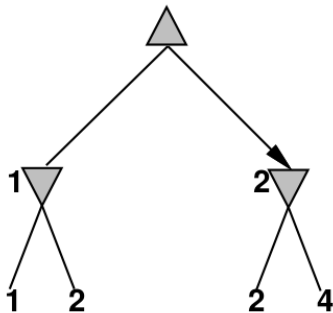
Exact values don't matter!

Behaviour preserved **under any monotonic transformation** of $Eval()$

- Only the order matters!

MAX

MIN



(© S. Russell & P. Norwig, AIMA)

Remark

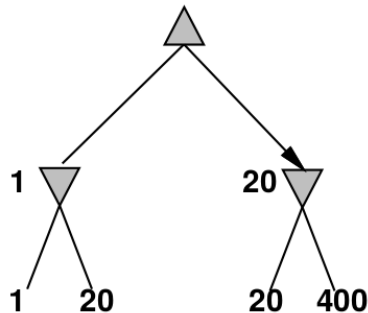
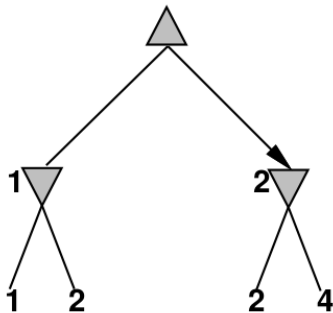
Exact values don't matter!

Behaviour preserved under any monotonic transformation of $Eval()$

- Only the order matters!

MAX

MIN



(© S. Russell & P. Norwig, AIMA)

Multi-Player Games: Optimal Decisions

- Replace the single value for each node with a **vector of values**
 - each value represent score from each player's viewpoint
 - terminal states: utility for each agent
 - agents, in turn, choose the action with best value for themselves
- Alliances are possible!
 - e.g., if one agent is in dominant position, the other can ally

Multi-Player Games: Optimal Decisions

- Replace the single value for each node with a **vector of values**
 - each value represent score from each player's viewpoint
 - terminal states: utility for each agent
 - agents, in turn, choose the action with best value for themselves
- Alliances are possible!
 - e.g., if one agent is in dominant position, the other can ally

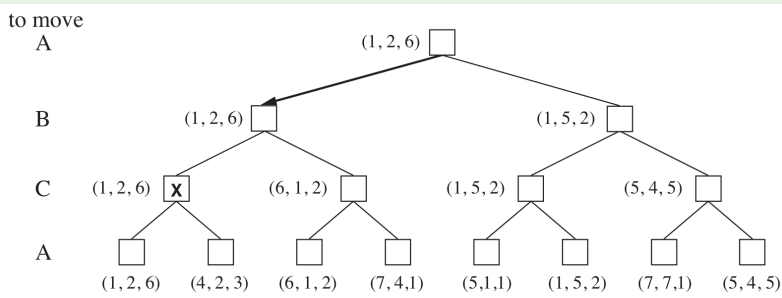
Multi-Player Games: Optimal Decisions

- Replace the single value for each node with a **vector of values**
 - each value represent score from each player's viewpoint
 - terminal states: utility for each agent
 - agents, in turn, choose the action with best value for themselves
- Alliances are possible!
 - e.g., if one agent is in dominant position, the other can ally

Multiplayer Min-Max Search: Example

The first three plies of a game tree with three players (A, B, C)

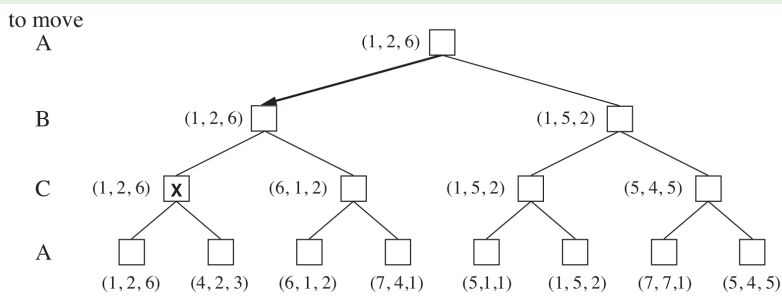
- Each node labeled with values from each player's viewpoint
- Agents choose the action with best value for themselves
 - ⇒ A chooses the left move (1, 2, 6) (bad for A and B, good for C), or
A chooses the left move (1, 5, 2) (equivalently bad for A, good for B, bad for C)
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2)
 - ⇒ **benefit for both**



Multiplayer Min-Max Search: Example

The first three plies of a game tree with three players (A, B, C)

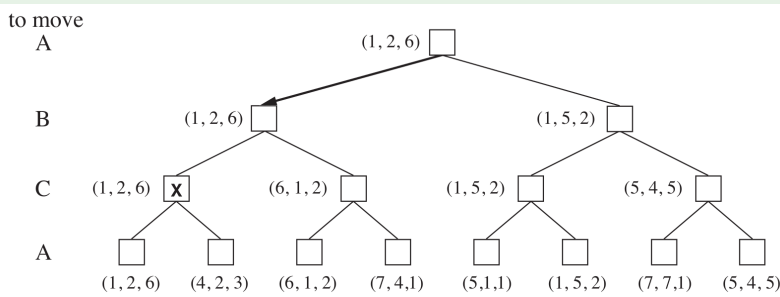
- Each node labeled with values from each player's viewpoint
- Agents choose the action with best value for themselves
 - ⇒ A chooses the left move (1, 2, 6) (bad for A and B, good for C), or
 - A chooses the left move (1, 5, 2) (equivalently bad for A, good for B, bad for C)
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2)
 - ⇒ **benefit for both**



Multiplayer Min-Max Search: Example

The first three plies of a game tree with three players (A, B, C)

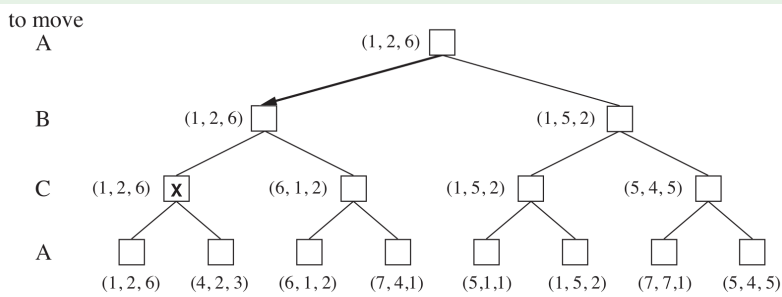
- Each node labeled with values from each player's viewpoint
- Agents choose the action with best value for themselves
 - ⇒ A chooses the left move (1, 2, 6) (bad for A and B, good for C), or
A chooses the left move (1, 5, 2) (equivalently bad for A, good for B, bad for C)
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2)
 - ⇒ **benefit for both**



Multiplayer Min-Max Search: Example

The first three plies of a game tree with three players (A, B, C)

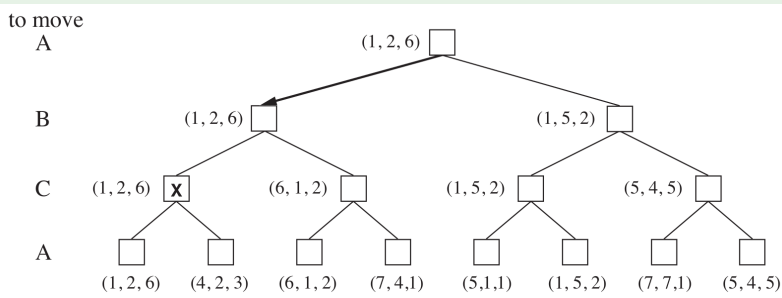
- Each node labeled with values from each player's viewpoint
- Agents choose the action with best value for themselves
 - ⇒ A chooses the left move (1, 2, 6) (bad for A and B, good for C), or
A chooses the left move (1, 5, 2) (equivalently bad for A, good for B, bad for C)
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2)
 - ⇒ benefit for both



Multiplayer Min-Max Search: Example

The first three plies of a game tree with three players (A, B, C)

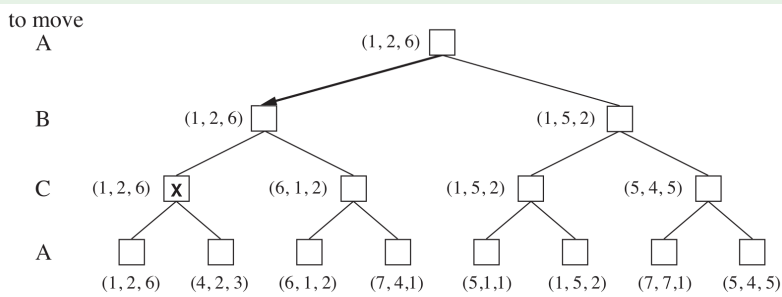
- Each node labeled with values from each player's viewpoint
- Agents choose the action with best value for themselves
 - ⇒ A chooses the left move (1, 2, 6) (bad for A and B, good for C), or
A chooses the left move (1, 5, 2) (equivalently bad for A, good for B, bad for C)
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2)
 - ⇒ benefit for both



Multiplayer Min-Max Search: Example

The first three plies of a game tree with three players (A, B, C)

- Each node labeled with values from each player's viewpoint
- Agents choose the action with best value for themselves
 - ⇒ A chooses the left move (1, 2, 6) (bad for A and B, good for C), or
A chooses the left move (1, 5, 2) (equivalently bad for A, good for B, bad for C)
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2)
 - ⇒ **benefit for both**



Exercise

- Consider the Multiplayer Min-Max Search example of previous slide
 - Redo it with choice order A-C-B
 - Redo it with choice order C-A-B
 - Redo it with choice order C-B-A
 - Redo it with choice order B-A-C
 - Redo it with choice order B-C-A
- Do they have all the same outcome?
- For each case, try to define the best moves in case of alliance between the top two players

Exercise

- Consider the Multiplayer Min-Max Search example of previous slide
 - Redo it with choice order A-C-B
 - Redo it with choice order C-A-B
 - Redo it with choice order C-B-A
 - Redo it with choice order B-A-C
 - Redo it with choice order B-C-A
- Do they have all the same outcome?
 - For each case, try to define the best moves in case of alliance between the top two players

Exercise

- Consider the Multiplayer Min-Max Search example of previous slide
 - Redo it with choice order A-C-B
 - Redo it with choice order C-A-B
 - Redo it with choice order C-B-A
 - Redo it with choice order B-A-C
 - Redo it with choice order B-C-A
- Do they have all the same outcome?
- For each case, try to define the best moves in case of alliance between the top two players

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
 $\implies$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \implies 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
 $\implies$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \implies 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
 $\Rightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \Rightarrow 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \implies 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \implies 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \Rightarrow 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \Rightarrow 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \Rightarrow 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \Rightarrow 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \Rightarrow 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \Rightarrow 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

For chess, $b \approx 35$, $m \approx 100 \implies 35^{100} = 10^{154}$ (!)

We need to prune the tree!

The Minimax Algorithm: Properties

- Complete? Yes, if tree is finite
- Optimal? Yes, against an optimal opponent
 - What about non-optimal opponent?
⇒ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

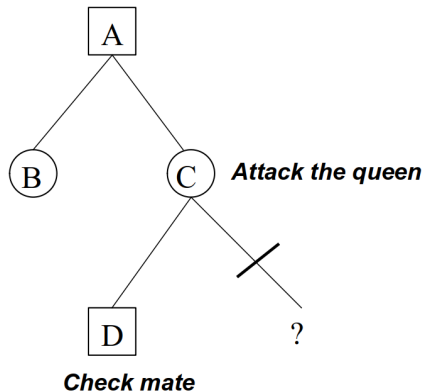
For chess, $b \approx 35$, $m \approx 100 \implies 35^{100} = 10^{154}$ (!)

We need to prune the tree!

Outline

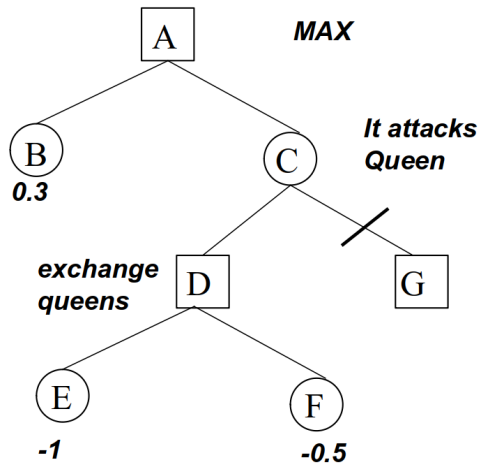
- 1 Games
- 2 Optimal Decisions in Games
 - Min-Max Search
 - Alpha-Beta Pruning
- 3 Adversarial Search with Resource Limits
- 4 Stochastic Games

Example: Chess (1)



- No matter which is the evaluation of the other children of C (I realize that I should never move to C).

Example: Chess (2)



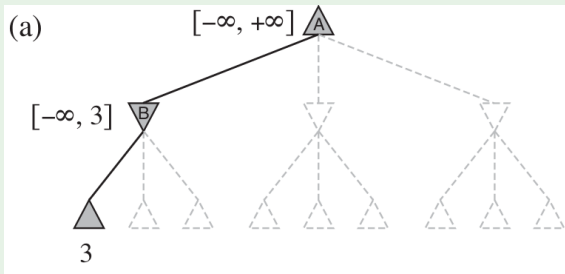
- Max in A avoids C because B is better. At most max gets from C a -0.5 so 0.3 is better
- The subtree in G can be cut as soon as I receive the value of D.
Indeed: $C = \min(-0.5, G)$;
 $A = \max(0.3, \min(-0.5, G)) = 0.3$

Since A is independent of G, the tree under G can be cut.

Pruning Min-Max Search: Example

- Consider the Min-Max example, let $[\alpha, \beta]$ track the currently-known bounds:
(α (resp β): best value for MAX (resp MIN) so far at any choice point along the path)
- (a): B labeled with $[-\infty, 3]$ (MIN will not choose values ≥ 3 for B)
- (c): B labeled with $[3, 3]$ (MIN cannot find values ≤ 3 for B) $\Rightarrow$ A labeled with $[3, +\infty]$
- (d): Is it necessary to evaluate the remaining leaves of C?
NO! They cannot produce an upper bound ≥ 2
 $\Rightarrow$ MAX cannot update the $\alpha = 3$ bound due to C
- (e): MAX updates the upper bound to 14 (D is last subtree)
- (f): D labeled $[2, 2] \Rightarrow$ MAX updates the upper bound to 3

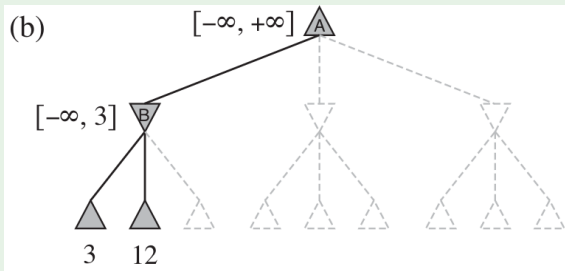
$\Rightarrow$ final value: 3



Pruning Min-Max Search: Example

- Consider the Min-Max example, let $[\alpha, \beta]$ track the currently-known bounds:
(α (resp β): best value for MAX (resp MIN) so far at any choice point along the path)
- (a): B labeled with $[-\infty, 3]$ (MIN will not choose values ≥ 3 for B)
- (c): B labeled with $[3, 3]$ (MIN cannot find values ≤ 3 for B) $\Rightarrow$ A labeled with $[3, +\infty]$
- (d): Is it necessary to evaluate the remaining leaves of C?
NO! They cannot produce an upper bound ≥ 2
 $\Rightarrow$ MAX cannot update the $\alpha = 3$ bound due to C
- (e): MAX updates the upper bound to 14 (D is last subtree)
- (f): D labeled $[2, 2] \Rightarrow$ MAX updates the upper bound to 3

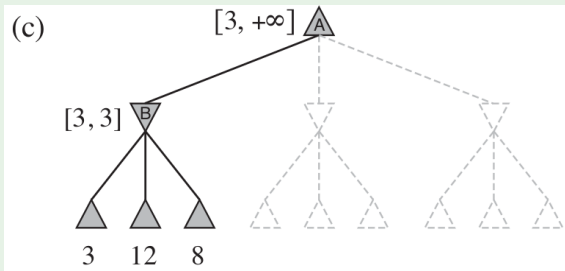
$\Rightarrow$ final value: 3



Pruning Min-Max Search: Example

- Consider the Min-Max example, let $[\alpha, \beta]$ track the currently-known bounds:
(α (resp β): best value for MAX (resp MIN) so far at any choice point along the path)
- (a): B labeled with $[-\infty, 3]$ (MIN will not choose values ≥ 3 for B)
- (c): B labeled with $[3, 3]$ (MIN cannot find values ≤ 3 for B) $\Rightarrow$ A labeled with $[3, +\infty]$
- (d): Is it necessary to evaluate the remaining leaves of C?
NO! They cannot produce an upper bound ≥ 2
 $\Rightarrow$ MAX cannot update the $\alpha = 3$ bound due to C
- (e): MAX updates the upper bound to 14 (D is last subtree)
- (f): D labeled $[2, 2] \Rightarrow$ MAX updates the upper bound to 3

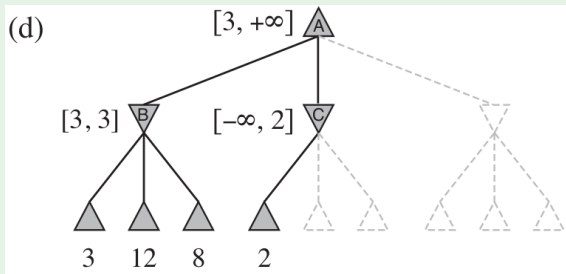
$\Rightarrow$ final value: 3



Pruning Min-Max Search: Example

- Consider the Min-Max example, let $[\alpha, \beta]$ track the currently-known bounds:
(α (resp β): best value for MAX (resp MIN) so far at any choice point along the path)
- (a): B labeled with $[-\infty, 3]$ (MIN will not choose values ≥ 3 for B)
- (c): B labeled with $[3, 3]$ (MIN cannot find values ≤ 3 for B) $\Rightarrow$ A labeled with $[3, +\infty]$
- (d): Is it necessary to evaluate the remaining leaves of C?
NO! They cannot produce an upper bound ≥ 2
 $\Rightarrow$ MAX cannot update the $\alpha = 3$ bound due to C
- (e): MAX updates the upper bound to 14 (D is last subtree)
- (f): D labeled $[2, 2] \Rightarrow$ MAX updates the upper bound to 3

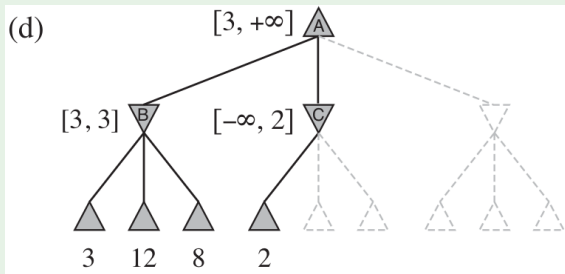
$\Rightarrow$ final value: 3



Pruning Min-Max Search: Example

- Consider the Min-Max example, let $[\alpha, \beta]$ track the currently-known bounds:
(α (resp β): best value for MAX (resp MIN) so far at any choice point along the path)
- (a): B labeled with $[-\infty, 3]$ (MIN will not choose values ≥ 3 for B)
- (c): B labeled with $[3, 3]$ (MIN cannot find values ≤ 3 for B) $\Rightarrow$ A labeled with $[3, +\infty]$
- (d): Is it necessary to evaluate the remaining leaves of C?
NO! They cannot produce an upper bound ≥ 2
 $\Rightarrow$ MAX cannot update the $\alpha = 3$ bound due to C
- (e): MAX updates the upper bound to 14 (D is last subtree)
- (f): D labeled $[2, 2] \Rightarrow$ MAX updates the upper bound to 3

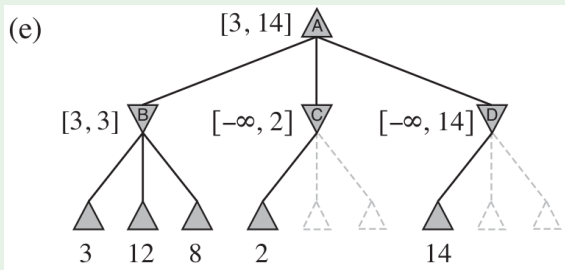
$\Rightarrow$ final value: 3



Pruning Min-Max Search: Example

- Consider the Min-Max example, let $[\alpha, \beta]$ track the currently-known bounds:
(α (resp β): best value for MAX (resp MIN) so far at any choice point along the path)
- (a): B labeled with $[-\infty, 3]$ (MIN will not choose values ≥ 3 for B)
- (c): B labeled with $[3, 3]$ (MIN cannot find values ≤ 3 for B) $\Rightarrow$ A labeled with $[3, +\infty]$
- (d): Is it necessary to evaluate the remaining leaves of C?
NO! They cannot produce an upper bound ≥ 2
 $\Rightarrow$ MAX cannot update the $\alpha = 3$ bound due to C
- (e): MAX updates the upper bound to 14 (D is last subtree)
- (f): D labeled $[2, 2] \Rightarrow$ MAX updates the upper bound to 3

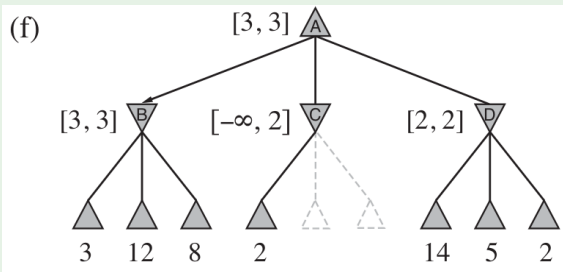
$\Rightarrow$ final value: 3



Pruning Min-Max Search: Example

- Consider the Min-Max example, let $[\alpha, \beta]$ track the currently-known bounds:
(α (resp β): best value for MAX (resp MIN) so far at any choice point along the path)
- (a): B labeled with $[-\infty, 3]$ (MIN will not choose values ≥ 3 for B)
- (c): B labeled with $[3, 3]$ (MIN cannot find values ≤ 3 for B) $\Rightarrow$ A labeled with $[3, +\infty]$
- (d): Is it necessary to evaluate the remaining leaves of C?
NO! They cannot produce an upper bound ≥ 2
 $\Rightarrow$ MAX cannot update the $\alpha = 3$ bound due to C
- (e): MAX updates the upper bound to 14 (D is last subtree)
- (f): D labeled $[2, 2] \Rightarrow$ MAX updates the upper bound to 3

$\Rightarrow$ final value: 3

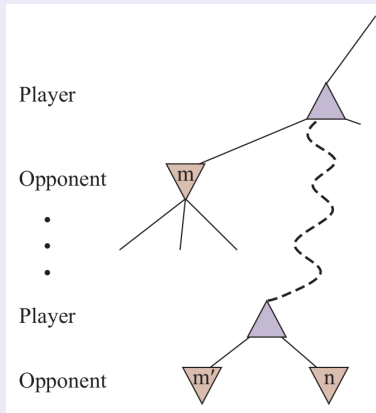


Alpha-Beta Pruning Technique for Min-Max Search

- Idea: consider a node n (terminal or intermediate) and its **current** value
 - If player has a better choice at the same level of n (m') or at any point higher up in the tree (m), then **n will never be reached in actual play**

⇒ as soon as we know enough of n to draw this conclusion, **we can prune n**
- Alpha-Beta Pruning: nodes labeled with $[\alpha, \beta]$ s.t.:
 - α : best value for MAX (highest) so far at any choice point along the path
 - ⇒ lower bound for future values
 - β : best value for MIN (lowest) so far at any choice point along the path
 - ⇒ upper bound for future values

⇒ **Prune n if its value is worse (lower) than the current α value for MAX (dual for β , MIN)**

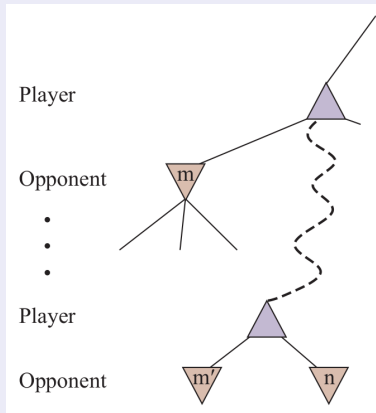


Alpha-Beta Pruning Technique for Min-Max Search

- Idea: consider a node n (terminal or intermediate) and its **current** value
 - If player has a better choice at the same level of n (m') or at any point higher up in the tree (m), then **n will never be reached in actual play**

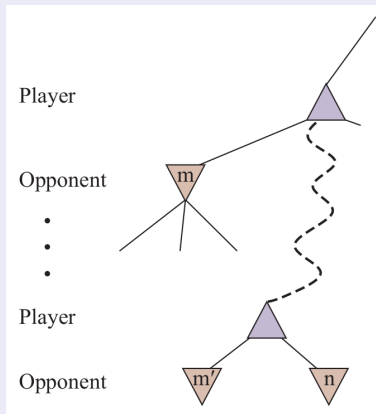
⇒ as soon as we know enough of n to draw this conclusion, **we can prune n**
- **Alpha-Beta Pruning**: nodes labeled with $[\alpha, \beta]$ s.t.:
 - α : **best value for MAX (highest) so far at any choice point along the path**
⇒ lower bound for future values
 - β : **best value for MIN (lowest) so far at any choice point along the path**
⇒ upper bound for future values

⇒ **Prune n if its value is worse (lower) than the current α value for MAX (dual for β , MIN)**



Alpha-Beta Pruning Technique for Min-Max Search

- Idea: consider a node n (terminal or intermediate) and its **current** value
 - If player has a better choice at the same level of n (m') or at any point higher up in the tree (m), then **n will never be reached in actual play**
 - ⇒ as soon as we know enough of n to draw this conclusion, **we can prune n**
 - **Alpha-Beta Pruning**: nodes labeled with $[\alpha, \beta]$ s.t.:
 - α : **best value for MAX (highest) so far at any choice point along the path**
 - ⇒ lower bound for future values
 - β : **best value for MIN (lowest) so far at any choice point along the path**
 - ⇒ upper bound for future values
- ⇒ **Prune n if its value is worse (lower) than the current α value for MAX (dual for β , MIN)**



The Alpha-Beta Search Algorithm

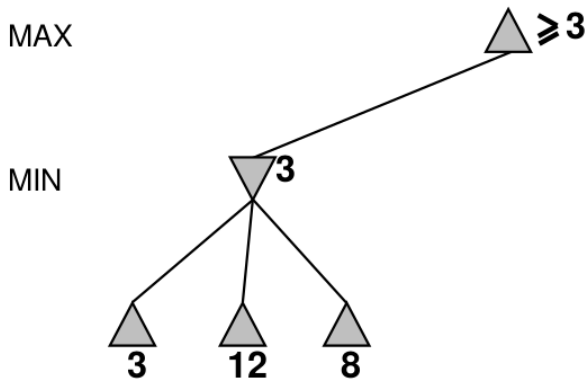
```
function ALPHA-BETA-SEARCH(state) returns an action  
   $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$   
  return the action in  $\text{ACTIONS}(\text{state})$  with value  $v$ 
```

```
function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value  
  if  $\text{TERMINAL-TEST}(\text{state})$  then return  $\text{UTILITY}(\text{state})$   
   $v \leftarrow -\infty$   
  for each  $a$  in  $\text{ACTIONS}(\text{state})$  do  
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
    if  $v \geq \beta$  then return  $v$  // MIN will never choose a bigger value  
     $\alpha \leftarrow \text{MAX}(\alpha, v)$   
  return  $v$ 
```

```
function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value  
  if  $\text{TERMINAL-TEST}(\text{state})$  then return  $\text{UTILITY}(\text{state})$   
   $v \leftarrow +\infty$   
  for each  $a$  in  $\text{ACTIONS}(\text{state})$  do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
    if  $v \leq \alpha$  then return  $v$  // MAX will never choose a smaller value  
     $\beta \leftarrow \text{MIN}(\beta, v)$   
  return  $v$ 
```

Example revisited: Alpha-Beta Cuts

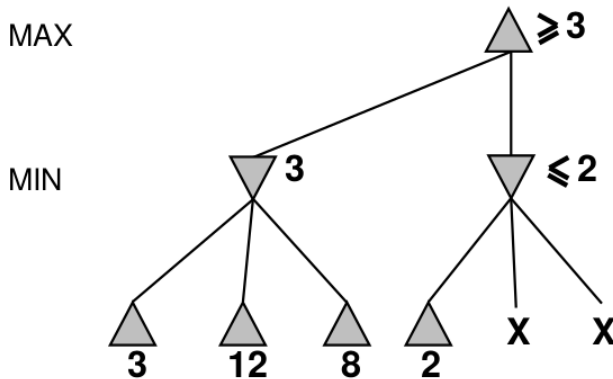
- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

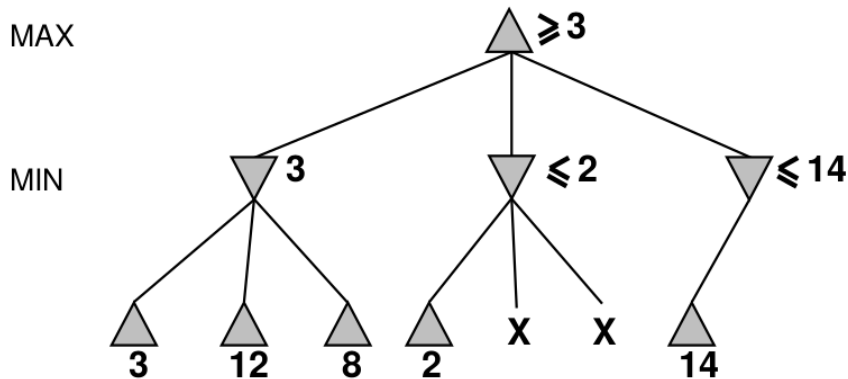
Example revisited: Alpha-Beta Cuts

- Notation: $\geq \alpha$; $\leq \beta$;



Example revisited: Alpha-Beta Cuts

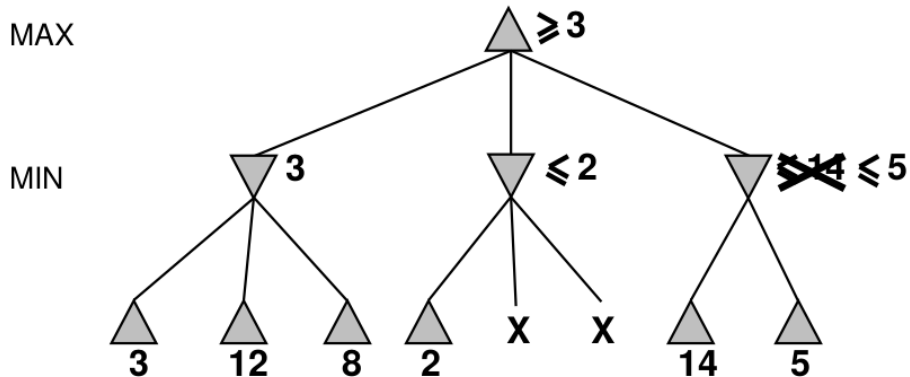
- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

Example revisited: Alpha-Beta Cuts

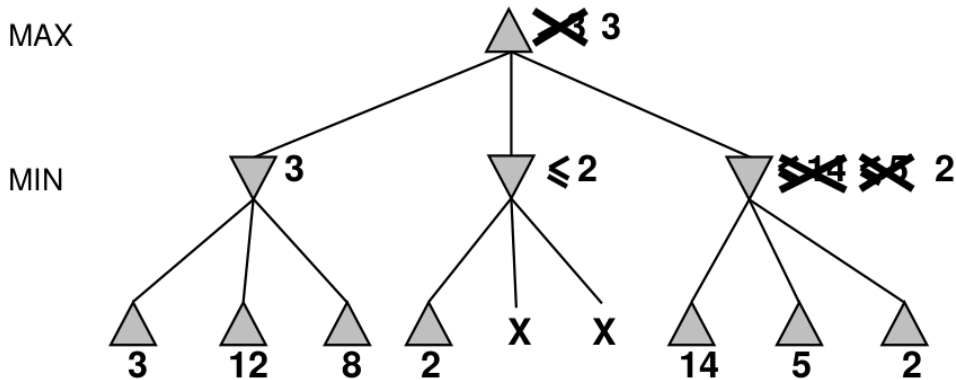
- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

Example revisited: Alpha-Beta Cuts

- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

Properties of Alpha-Beta Search

- Pruning does not affect the final result $\implies$ **correctness preserved**
- Good move ordering improves effectiveness of pruning
 - Ex: if MIN expands 3rd child of D first (see 2nd last example), then the others are pruned
 $\implies$ try to examine first the successors that are likely to be best
- With “perfect” ordering, time complexity reduces to $O(b^{m/2})$
 - aka “killer-move heuristic”
 $\implies$ doubles solvable depth!
- With “random” ordering, time complexity reduces to $O(b^{3m/4})$
- “Graph-based” version further improves performances
 - track explored states via hash table

Properties of Alpha-Beta Search

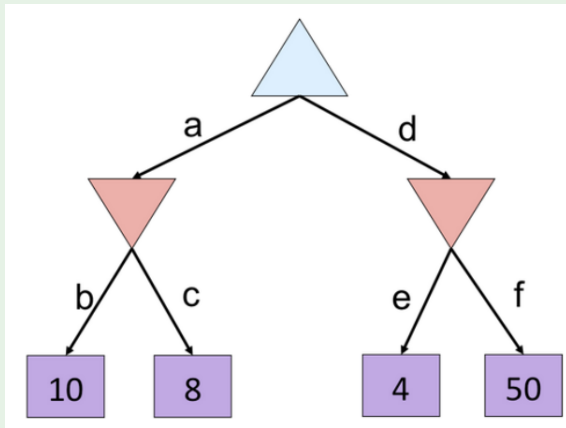
- Pruning does not affect the final result $\implies$ **correctness preserved**
- Good move ordering improves effectiveness of pruning
 - Ex: if MIN expands 3rd child of D first (see 2nd last example), then the others are pruned
 $\implies$ try to examine first the successors that are likely to be best
- With “perfect” ordering, time complexity reduces to $O(b^{m/2})$
 - aka “killer-move heuristic”
 $\implies$ doubles solvable depth!
- With “random” ordering, time complexity reduces to $O(b^{3m/4})$
- “Graph-based” version further improves performances
 - track explored states via hash table

Properties of Alpha-Beta Search

- Pruning does not affect the final result $\implies$ **correctness preserved**
- Good move ordering improves effectiveness of pruning
 - Ex: if MIN expands 3rd child of D first (see 2nd last example), then the others are pruned $\implies$ try to examine first the successors that are likely to be best
- With “perfect” ordering, time complexity reduces to $O(b^{m/2})$
 - aka “killer-move heuristic” $\implies$ doubles solvable depth!
- With “random” ordering, time complexity reduces to $O(b^{3m/4})$
- “Graph-based” version further improves performances
 - track explored states via hash table

Exercise I

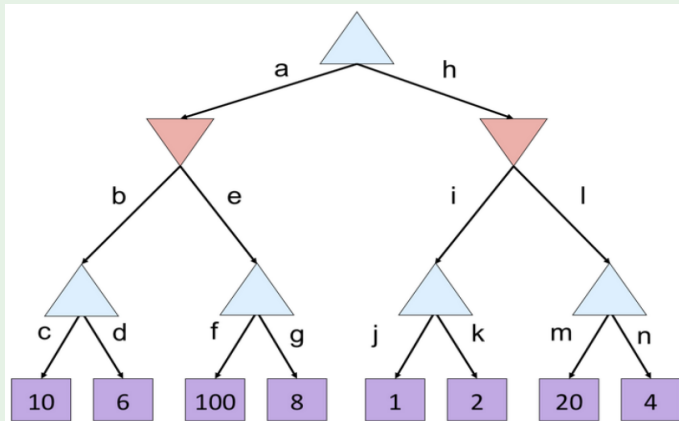
Apply alpha-beta search to the following tree



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

Exercise II

Apply alpha-beta search to the following tree



(© D. Klein, P. Abbeel, S. Levine, S. Russell, U. Berkeley)

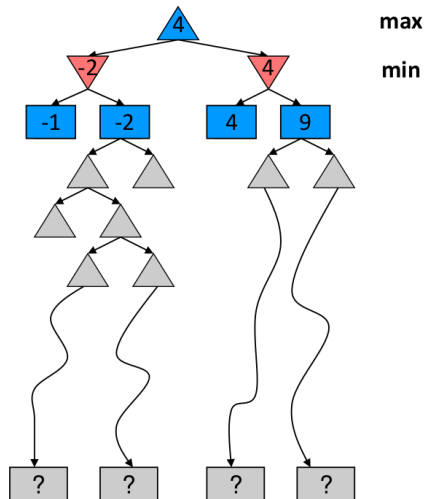
Outline

- 1 Games
- 2 Optimal Decisions in Games
 - Min-Max Search
 - Alpha-Beta Pruning
- 3 Adversarial Search with Resource Limits**
- 4 Stochastic Games

Adversarial Search with Resource Limits

Problem: In realistic games, full search is impractical!

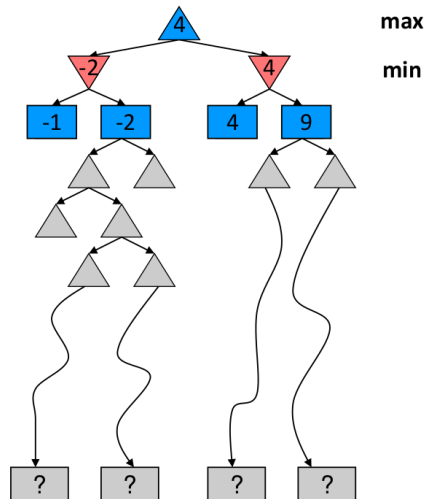
- Complexity: b^d (ex. chess: $\approx 35^{100}$)
- Idea [Shannon, 1949]: Depth-limited search
 - cut off minimax search earlier, after limited depth
 - replace terminal utility function with evaluation for non-terminal nodes
- Ex (chess): depth $d = 8$ (decent)
 $\implies \alpha\text{-}\beta$: $35^{8/2} \approx 10^5$ (feasible)



Adversarial Search with Resource Limits

Problem: In realistic games, full search is impractical!

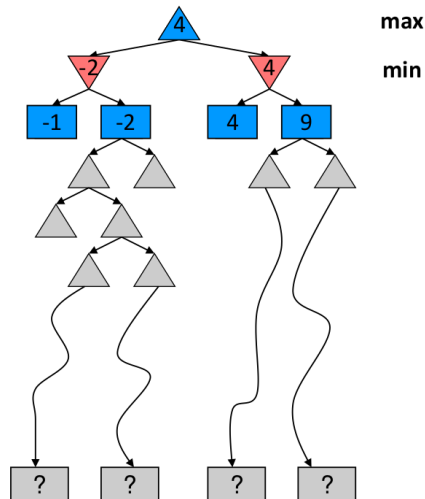
- Complexity: b^d (ex. chess: $\approx 35^{100}$)
- Idea [Shannon, 1949]: **Depth-limited search**
 - cut off minimax search earlier, after limited depth
 - replace **terminal utility function** with **evaluation for non-terminal nodes**
- Ex (chess): depth $d = 8$ (decent)
 $\implies \alpha\text{-}\beta: 35^{8/2} \approx 10^5$ (feasible)



Adversarial Search with Resource Limits

Problem: In realistic games, full search is impractical!

- Complexity: b^d (ex. chess: $\approx 35^{100}$)
- Idea [Shannon, 1949]: Depth-limited search
 - cut off minimax search earlier, after limited depth
 - replace terminal utility function with evaluation for non-terminal nodes
- Ex (chess): depth $d = 8$ (decent)
 $\implies \alpha\text{-}\beta$: $35^{8/2} \approx 10^5$ (feasible)



Adversarial Search with Resource Limits [cont.]

- Idea:

- cut off the search earlier, at limited depths
- apply a heuristic evaluation function to states in the search

⇒ effectively turning nonterminal nodes into terminal leaves

- Modify *Minimax()* or Alpha-Beta search in two ways:

- replace the utility function *Utility(s)* by a heuristic evaluation function *Eval(s)*, which estimates the position's utility
- replace the terminal test *TerminalTest(s)* by a cutoff test *CutOffTest(s, d)*, that decides when to apply *Eval()*
- plus some bookkeeping to increase depth *d* at each recursive call

⇒ Heuristic variant of *Minimax()*:

$$H\text{-Minimax}(s, d) \stackrel{\text{def}}{=} \begin{cases} Eval(s) & \text{if } CutOffTest(s, d) \\ \max_{a \in Actions(s)} H\text{-Minimax}(Result(s, a), d + 1) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} H\text{-Minimax}(Result(s, a), d + 1) & \text{if } Player(s) = MIN \end{cases}$$

⇒ Heuristic variant of alpha-beta: substitute the terminal test with

If *CutOffTest(s)* then return *Eval(s)*

Adversarial Search with Resource Limits [cont.]

- Idea:

- cut off the search earlier, at limited depths
- apply a heuristic evaluation function to states in the search

⇒ effectively turning nonterminal nodes into terminal leaves

- Modify *Minimax()* or Alpha-Beta search in two ways:

- replace the utility function *Utility(s)* by a heuristic evaluation function *Eval(s)*, which estimates the position's utility
- replace the terminal test *TerminalTest(s)* by a cutoff test *CutOffTest(s, d)*, that decides when to apply *Eval()*
- plus some bookkeeping to increase depth *d* at each recursive call

⇒ Heuristic variant of *Minimax()*:

$$H\text{-Minimax}(s, d) \stackrel{\text{def}}{=} \begin{cases} Eval(s) & \text{if } CutOffTest(s, d) \\ \max_{a \in Actions(s)} H\text{-Minimax}(Result(s, a), d + 1) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} H\text{-Minimax}(Result(s, a), d + 1) & \text{if } Player(s) = MIN \end{cases}$$

⇒ Heuristic variant of alpha-beta: substitute the terminal test with

If *CutOffTest(s)* then return *Eval(s)*

Adversarial Search with Resource Limits [cont.]

- Idea:

- cut off the search earlier, at limited depths
- apply a heuristic evaluation function to states in the search

⇒ effectively turning nonterminal nodes into terminal leaves

- Modify *Minimax()* or Alpha-Beta search in two ways:

- replace the utility function *Utility(s)* by a heuristic evaluation function *Eval(s)*, which estimates the position's utility
- replace the terminal test *TerminalTest(s)* by a cutoff test *CutOffTest(s, d)*, that decides when to apply *Eval()*
- plus some bookkeeping to increase depth *d* at each recursive call

⇒ Heuristic variant of *Minimax()*:

$$H\text{-Minimax}(s, d) \stackrel{\text{def}}{=} \begin{cases} Eval(s) & \text{if } CutOffTest(s, d) \\ \max_{a \in Actions(s)} H\text{-Minimax}(Result(s, a), d + 1) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} H\text{-Minimax}(Result(s, a), d + 1) & \text{if } Player(s) = MIN \end{cases}$$

⇒ Heuristic variant of alpha-beta: substitute the terminal test with

If *CutOffTest(s)* then return *Eval(s)*

Evaluation Functions

$Eval(s)$

- Should be relatively cheap to compute
- Returns an estimate of the expected utility from a given position
 - Ideal function: returns the actual minimax value of the position
- Should order terminal states the same way as the utility function
 - e.g., wins > draws > losses
- For nonterminal states, should be strongly correlated with the actual chances of winning
- Defines equivalence classes of positions (same $Eval(s)$ value)
 - e.g. returns a value reflecting the % of states with each outcome
- Typically weighted linear sum of features:
$$Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \dots + w_n \cdot f_n(s)$$
 - ex (chess): $f_{pawns}(s) = \#white\ pawns - \#black\ pawns$,
 $w_{pawns} = 1$; $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth
 - ex: knights more valuable with low depths, rooks more valuable with high depths
- May be very inaccurate for some positions

Evaluation Functions

$Eval(s)$

- Should be relatively cheap to compute
- Returns an estimate of the expected utility from a given position
 - Ideal function: returns the actual minimax value of the position
- Should order terminal states the same way as the utility function
 - e.g., wins > draws > losses
- For nonterminal states, should be strongly correlated with the actual chances of winning
- Defines equivalence classes of positions (same $Eval(s)$ value)
 - e.g. returns a value reflecting the % of states with each outcome
- Typically weighted linear sum of features:
$$Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \dots + w_n \cdot f_n(s)$$
 - ex (chess): $f_{pawns}(s) = \#white\ pawns - \#black\ pawns$,
 $w_{pawns} = 1$; $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth
 - ex: knights more valuable with low depths, rooks more valuable with high depths
- May be very inaccurate for some positions

Evaluation Functions

$Eval(s)$

- Should be relatively cheap to compute
- Returns an estimate of the expected utility from a given position
 - Ideal function: returns the actual minimax value of the position
- Should order terminal states the same way as the utility function
 - e.g., wins > draws > losses
- For nonterminal states, should be strongly correlated with the actual chances of winning
- Defines equivalence classes of positions (same $Eval(s)$ value)
 - e.g. returns a value reflecting the % of states with each outcome
- Typically weighted linear sum of features:
$$Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \dots + w_n \cdot f_n(s)$$
 - ex (chess): $f_{pawns}(s) = \#white\ pawns - \#black\ pawns$,
 $w_{pawns} = 1$; $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth
 - ex: knights more valuable with low depths, rooks more valuable with high depths
- May be very inaccurate for some positions

Evaluation Functions

$Eval(s)$

- Should be relatively cheap to compute
- Returns an estimate of the expected utility from a given position
 - Ideal function: returns the actual minimax value of the position
- Should order terminal states the same way as the utility function
 - e.g., wins > draws > losses
- For nonterminal states, should be strongly correlated with the actual chances of winning
- Defines equivalence classes of positions (same $Eval(s)$ value)
 - e.g. returns a value reflecting the % of states with each outcome
- Typically weighted linear sum of features:
$$Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \dots + w_n \cdot f_n(s)$$
 - ex (chess): $f_{pawns}(s) = \#white\ pawns - \#black\ pawns$,
 $w_{pawns} = 1$; $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth
 - ex: knights more valuable with low depths, rooks more valuable with high depths
- May be very inaccurate for some positions

Evaluation Functions

$Eval(s)$

- Should be relatively cheap to compute
- Returns an estimate of the expected utility from a given position
 - Ideal function: returns the actual minimax value of the position
- Should order terminal states the same way as the utility function
 - e.g., wins > draws > losses
- For nonterminal states, should be strongly correlated with the actual chances of winning
- Defines equivalence classes of positions (same $Eval(s)$ value)
 - e.g. returns a value reflecting the % of states with each outcome
- Typically weighted linear sum of features:
$$Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \dots + w_n \cdot f_n(s)$$
 - ex (chess): $f_{pawns}(s) = \#white\ pawns - \#black\ pawns$,
 $w_{pawns} = 1$; $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth
 - ex: knights more valuable with low depths, rooks more valuable with high depths
- May be very inaccurate for some positions

Evaluation Functions

$Eval(s)$

- Should be relatively cheap to compute
- Returns an estimate of the expected utility from a given position
 - Ideal function: returns the actual minimax value of the position
- Should order terminal states the same way as the utility function
 - e.g., wins > draws > losses
- For nonterminal states, should be strongly correlated with the actual chances of winning
- Defines equivalence classes of positions (same $Eval(s)$ value)
 - e.g. returns a value reflecting the % of states with each outcome
- Typically weighted linear sum of features:
$$Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + \dots + w_n \cdot f_n(s)$$
 - ex (chess): $f_{pawns}(s) = \#white\ pawns - \#black\ pawns$,
 $w_{pawns} = 1$; $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth
 - ex: knights more valuable with low depths, rooks more valuable with high depths
- May be very inaccurate for some positions

Example

- Two same-score positions (White: -8, Black: -3), white to move

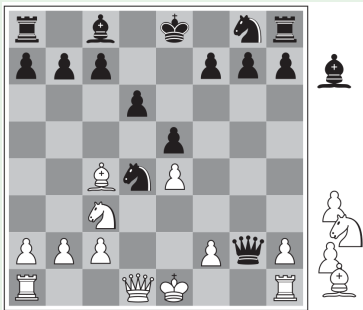
(a) Black has an advantage of a knight and two pawns,

⇒ should be enough to win the game

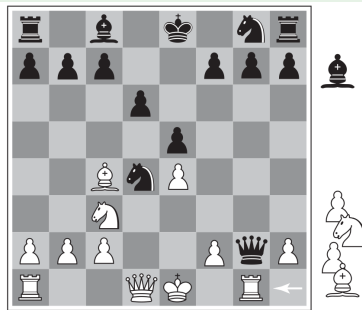
(b) White will capture the queen,

⇒ give it an advantage that should be strong enough to win

(Personal note: only a very-stupid black player would get into (b) scenario)



(a) White to move



(b) White to move

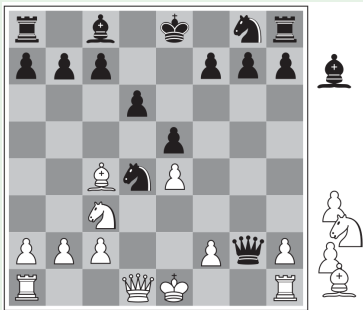
Example

- Two same-score positions (White: -8, Black: -3), white to move

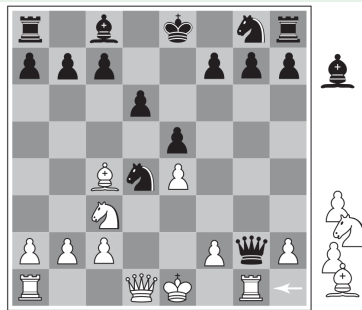
(a) Black has an advantage of a knight and two pawns,
⇒ should be enough to win the game

(b) White will capture the queen,
⇒ give it an advantage that should be strong enough to win

(Personal note: only a very-stupid black player would get into (b) scenario)



(a) White to move



(b) White to move

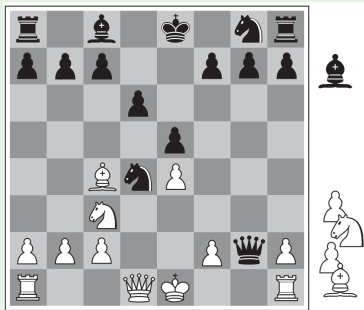
Example

- Two same-score positions (White: -8, Black: -3), white to move

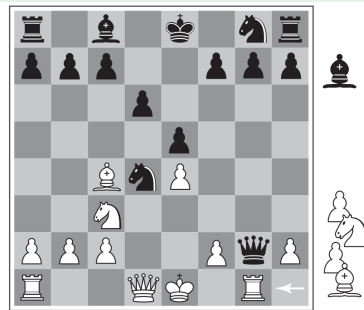
(a) Black has an advantage of a knight and two pawns,
⇒ should be enough to win the game

(b) White will capture the queen,
⇒ give it an advantage that should be strong enough to win

(Personal note: only a very-stupid black player would get into (b) scenario)



(a) White to move



(b) White to move

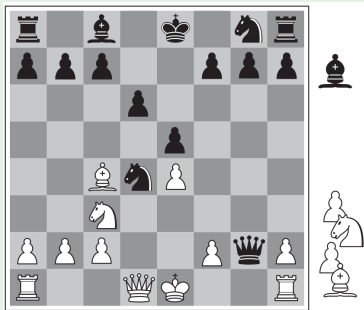
Example

- Two same-score positions (White: -8, Black: -3), white to move

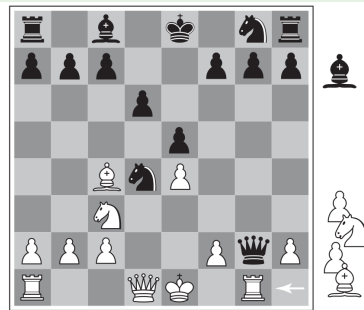
(a) Black has an advantage of a knight and two pawns,
⇒ should be enough to win the game

(b) White will capture the queen,
⇒ give it an advantage that should be strong enough to win

(Personal note: only a very-stupid black player would get into (b) scenario)



(a) White to move



(b) White to move

Cutting-off the Search

CutOffTest(state, depth)

- Most straightforward approach: **set a fixed depth limit**
 - d chosen s.t. a move is selected within the allocated time
 - sometimes may produce very inaccurate outcomes (see previous example)
- More robust approach: **apply Iterative Deepening**
- More sophisticated: apply *Eval()* only to **quiescent** states
 - **quiescent**: unlikely to exhibit wild swings in value in the near future
 - e.g. positions with direct favorable captures are not quiescent (previous example (b))

⇒ **further expand non-quiescent states until quiescence is reached**

Cutting-off the Search

CutOffTest(state, depth)

- Most straightforward approach: **set a fixed depth limit**
 - d chosen s.t. a move is selected within the allocated time
 - sometimes may produce very inaccurate outcomes (see previous example)
- More robust approach: **apply Iterative Deepening**
- More sophisticated: apply *Eval()* only to **quiescent** states
 - **quiescent**: unlikely to exhibit wild swings in value in the near future
 - e.g. positions with direct favorable captures are not quiescent (previous example (b))

⇒ further expand non-quiescent states until quiescence is reached

Cutting-off the Search

CutOffTest(state, depth)

- Most straightforward approach: **set a fixed depth limit**
 - d chosen s.t. a move is selected within the allocated time
 - sometimes may produce very inaccurate outcomes (see previous example)
- More robust approach: **apply Iterative Deepening**
- More sophisticated: apply *Eval()* only to **quiescent** states
 - **quiescent**: unlikely to exhibit wild swings in value in the near future
 - e.g. positions with direct favorable captures are not quiescent
(previous example (b))

⇒ further expand non-quiescent states until quiescence is reached

Cutting-off the Search

CutOffTest(state, depth)

- Most straightforward approach: **set a fixed depth limit**
 - d chosen s.t. a move is selected within the allocated time
 - sometimes may produce very inaccurate outcomes (see previous example)
- More robust approach: **apply Iterative Deepening**
- More sophisticated: apply *Eval()* only to **quiescent** states
 - **quiescent**: unlikely to exhibit wild swings in value in the near future
 - e.g. positions with direct favorable captures are not quiescent
(previous example (b))

⇒ **further expand non-quiescent states until quiescence is reached**

Deterministic Games in Practice

- **Checkers:** (1994) **Chinook** ended 40-year-reign of world champion **Marion Tinsley**
 - used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board
 - a total of 443,748,401,247 positions
- **Chess:** (1997) **Deep Blue** defeated world champion **Gary Kasparov** in a six-game match
 - searches 200 million positions per second
 - uses very sophisticated evaluation, and undisclosed methods
- **Othello:**
 - Human champions refuse to compete against computers, which are too good
- **Go:** (2016) **AlphaGo** beats world champion **Lee Sedol**
 - number of possible positions > number of atoms in the universe

Deterministic Games in Practice

- **Checkers:** (1994) **Chinook** ended 40-year-reign of world champion **Marion Tinsley**
 - used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board
 - a total of 443,748,401,247 positions
- **Chess:** (1997) **Deep Blue** defeated world champion **Gary Kasparov** in a six-game match
 - searches 200 million positions per second
 - uses very sophisticated evaluation, and undisclosed methods
- **Othello:**
 - Human champions refuse to compete against computers, which are too good
- **Go:** (2016) **AlphaGo** beats world champion **Lee Sedol**
 - number of possible positions > number of atoms in the universe

Deterministic Games in Practice

- **Checkers:** (1994) **Chinook** ended 40-year-reign of world champion **Marion Tinsley**
 - used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board
 - a total of 443,748,401,247 positions
- **Chess:** (1997) **Deep Blue** defeated world champion **Gary Kasparov** in a six-game match
 - searches 200 million positions per second
 - uses very sophisticated evaluation, and undisclosed methods
- **Othello:**
 - Human champions refuse to compete against computers, which are too good
- **Go:** (2016) **AlphaGo** beats world champion **Lee Sedol**
 - number of possible positions > number of atoms in the universe

Deterministic Games in Practice

- **Checkers:** (1994) **Chinook** ended 40-year-reign of world champion **Marion Tinsley**
 - used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board
 - a total of 443,748,401,247 positions
- **Chess:** (1997) **Deep Blue** defeated world champion **Gary Kasparov** in a six-game match
 - searches 200 million positions per second
 - uses very sophisticated evaluation, and undisclosed methods
- **Othello:**
 - Human champions refuse to compete against computers, which are too good
- **Go:** (2016) **AlphaGo** beats world champion **Lee Sedol**
 - number of possible positions $>$ number of atoms in the universe

AlphaGo beats GO world champion, Lee Sedol (2016)



Outline

- 1 Games
- 2 Optimal Decisions in Games
 - Min-Max Search
 - Alpha-Beta Pruning
- 3 Adversarial Search with Resource Limits
- 4 Stochastic Games

Stochastic Games: Generalities

- In real life, **unpredictable external events** may occur
- **Stochastic Games** mirror unpredictability by **random steps**:
 - e.g. dice throwing, card-shuffling, coin flipping, tile extraction, ...
- Ex: Backgammon
- Cannot calculate definite minimax value, only **expected values**
- Uncertain outcomes controlled by **chance**, not an adversary!
 - adversarial $\implies$ **worst case**
 - chance $\implies$ **average case**
- Ex: if chance is 0.5 each (coin):
 - minimax: 10
 - average: $(100+9)/2=54.5$

Stochastic Games: Generalities

- In real life, **unpredictable external events** may occur
- **Stochastic Games** mirror unpredictability by **random steps**:
 - e.g. **dice throwing**, **card-shuffling**, **coin flipping**, **tile extraction**, ...
- Ex: **Backgammon**
- Cannot calculate definite minimax value, only **expected values**
- Uncertain outcomes controlled by **chance**, not an adversary!
 - adversarial $\implies$ **worst case**
 - chance $\implies$ **average case**
- Ex: if chance is 0.5 each (coin):
 - minimax: 10
 - average: $(100+9)/2=54.5$

Stochastic Games: Generalities

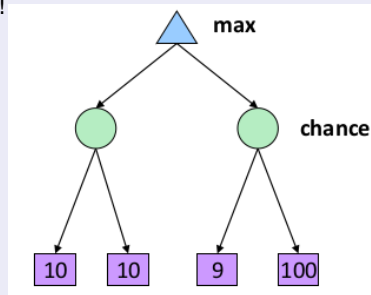
- In real life, **unpredictable external events** may occur
- **Stochastic Games** mirror unpredictability by **random steps**:
 - e.g. **dice throwing**, **card-shuffling**, **coin flipping**, **tile extraction**, ...
- Ex: **Backgammon**
- Cannot calculate definite minimax value, only **expected values**
- Uncertain outcomes controlled by **chance**, not an adversary!
 - adversarial $\implies$ **worst case**
 - chance $\implies$ **average case**
- Ex: if chance is 0.5 each (coin):
 - minimax: 10
 - average: $(100+9)/2=54.5$

Stochastic Games: Generalities

- In real life, **unpredictable external events** may occur
- **Stochastic Games** mirror unpredictability by **random steps**:
 - e.g. **dice throwing**, **card-shuffling**, **coin flipping**, **tile extraction**, ...
- Ex: **Backgammon**
- Cannot calculate definite minimax value, only **expected values**
- Uncertain outcomes controlled by **chance**, not an adversary!
 - adversarial $\implies$ **worst case**
 - chance $\implies$ **average case**
- Ex: if chance is 0.5 each (coin):
 - minimax: 10
 - average: $(100+9)/2=54.5$

Stochastic Games: Generalities

- In real life, **unpredictable external events** may occur
- **Stochastic Games** mirror unpredictability by **random steps**:
 - e.g. **dice throwing**, **card-shuffling**, **coin flipping**, **tile extraction**, ...
- Ex: **Backgammon**
- Cannot calculate definite minimax value, only **expected values**
- Uncertain outcomes controlled by **chance**, not an adversary!
 - adversarial $\Rightarrow$ **worst case**
 - chance $\Rightarrow$ **average case**
- Ex: if chance is 0.5 each (coin):
 - minimax: 10
 - average: $(100+9)/2=54.5$



An Example: Backgammon

- Rules

- 15 pieces each
- white moves clockwise to 25, black moves counterclockwise to 0
- a piece can move to a position unless ≥ 2 opponent pieces there
- if there is one opponent, it is captured and must start over
- termination: all whites in 25 or all blacks in 0

- Ex: Possible white moves (dice: 6,5):

(5-10,5-11)

(5-11,19-24)

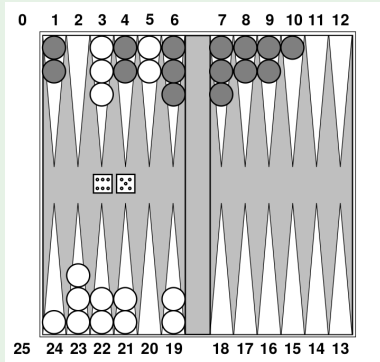
(5-10,10-16)

(5-11,11-16)

- Combines strategy with luck

⇒ stochastic component (dice)

- double rolls (1-1),..., (6-6)
have 1/36 probability each
- other 15 distinct rolls
have a 1/18 probability each



(© S. Russell & P. Norwig, AIMA)

An Example: Backgammon

- Rules

- 15 pieces each
- white moves clockwise to 25, black moves counterclockwise to 0
- a piece can move to a position unless ≥ 2 opponent pieces there
- if there is one opponent, it is captured and must start over
- termination: all whites in 25 or all blacks in 0

- Ex: Possible white moves (dice: 6,5):

(5-10,5-11)

(5-11,19-24)

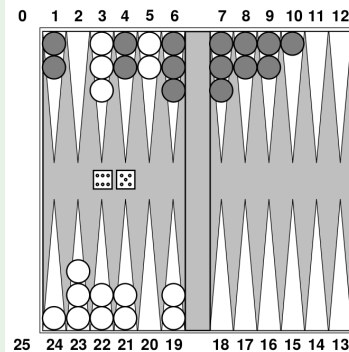
(5-10,10-16)

(5-11,11-16)

- Combines strategy with luck

⇒ stochastic component (dice)

- double rolls (1-1),..., (6-6)
have 1/36 probability each
- other 15 distinct rolls
have a 1/18 probability each



(© S. Russell & P. Norwig, AIMA)

An Example: Backgammon

- Rules

- 15 pieces each
- white moves clockwise to 25, black moves counterclockwise to 0
- a piece can move to a position unless ≥ 2 opponent pieces there
- if there is one opponent, it is captured and must start over
- termination: all whites in 25 or all blacks in 0

- Ex: Possible white moves (dice: 6,5):

(5-10,5-11)

(5-11,19-24)

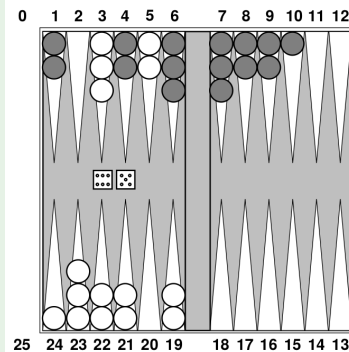
(5-10,10-16)

(5-11,11-16)

- Combines strategy with **luck**

⇒ **stochastic component** (dice)

- double rolls (1-1),..., (6-6)
have 1/36 probability each
- other 15 distinct rolls
have a 1/18 probability each

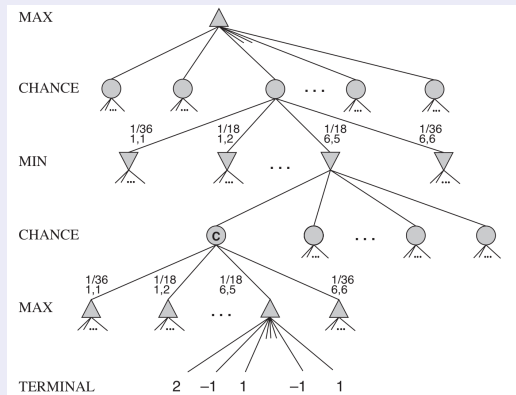


(© S. Russell & P. Norwig, AIMA)

Stochastic Games Trees

Idea:

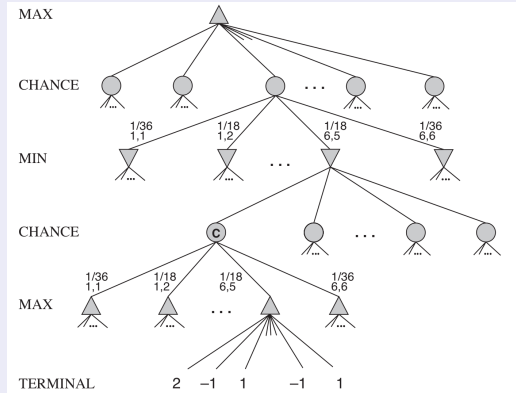
- A tree for a stochastic game includes **chance nodes** in addition to MAX and MIN nodes.
 - chance nodes above agent represent **stochastic events** for agent (e.g. dice roll)
 - outgoing arcs represent **stochastic event outcomes**
 - labeled with **stochastic event** and relative **probability**



Stochastic Games Trees

Idea:

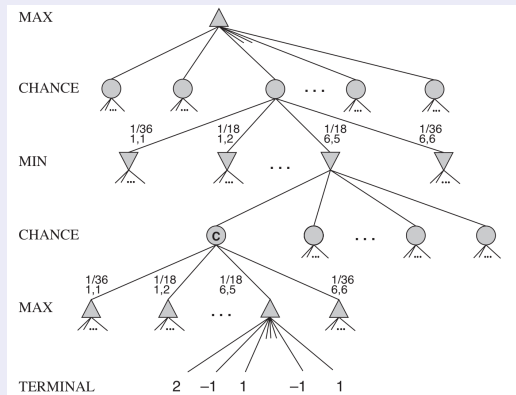
- A tree for a stochastic game includes **chance nodes** in addition to **MAX** and **MIN** nodes.
 - chance nodes above agent represent **stochastic events** for agent (e.g. dice roll)
 - outgoing arcs represent **stochastic event outcomes**
 - labeled with **stochastic event** and relative **probability**



Stochastic Games Trees

Idea:

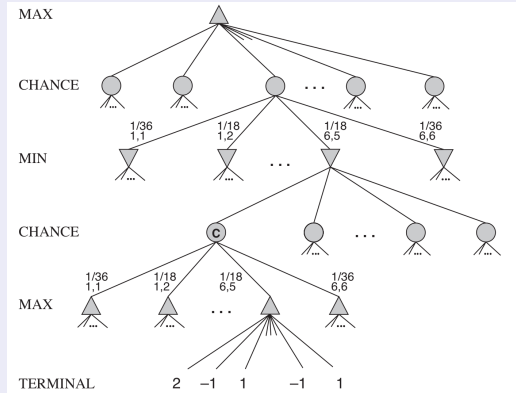
- A tree for a stochastic game includes chance nodes in addition to MAX and MIN nodes.
 - chance nodes above agent represent stochastic events for agent (e.g. dice roll)
 - outgoing arcs represent stochastic event outcomes
 - labeled with stochastic event and relative probability



Stochastic Games Trees

Idea:

- A tree for a stochastic game includes chance nodes in addition to MAX and MIN nodes.
 - chance nodes above agent represent stochastic events for agent (e.g. dice roll)
 - outgoing arcs represent stochastic event outcomes
 - labeled with stochastic event and relative probability



Algorithm for Stochastic Games: *ExpectMinimax()*

- Extension of *Minimax()*, handling also chance nodes:

$$ExpectMinimax(s) \stackrel{\text{def}}{=} \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ \max_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MIN \\ \sum_r P(r) \cdot ExpectMinimax(Result(s, r)) & \text{if } Player(s) = Chance \end{cases}$$

- $P(r)$: probability of stochastic event outcome r
- chance seen as an actor ("Chance")
- stochastic event outcomes r (e.g., dice values) seen as actions

⇒ Returns the weighted average of the minimax outcomes (recall that $\sum_r P(r) = 1$)

Algorithm for Stochastic Games: *ExpectMinimax()*

- Extension of *Minimax()*, handling also chance nodes:

$$ExpectMinimax(s) \stackrel{\text{def}}{=} \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ \max_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MIN \\ \sum_r P(r) \cdot ExpectMinimax(Result(s, r)) & \text{if } Player(s) = Chance \end{cases}$$

- $P(r)$: probability of stochastic event outcome r
- **chance seen as an actor** ("Chance")
- **stochastic event outcomes r** (e.g., dice values) **seen as actions**

⇒ Returns the weighted average of the minimax outcomes (recall that $\sum_r P(r) = 1$)

Algorithm for Stochastic Games: *ExpectMinimax()*

- Extension of *Minimax()*, handling also chance nodes:

$$ExpectMinimax(s) \stackrel{\text{def}}{=} \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ \max_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MIN \\ \sum_r P(r) \cdot ExpectMinimax(Result(s, r)) & \text{if } Player(s) = Chance \end{cases}$$

- $P(r)$: probability of stochastic event outcome r
- chance seen as an actor ("Chance")
- stochastic event outcomes r (e.g., dice values) seen as actions

⇒ Returns the weighted average of the minimax outcomes (recall that $\sum_r P(r) = 1$)

Algorithm for Stochastic Games: *ExpectMinimax()*

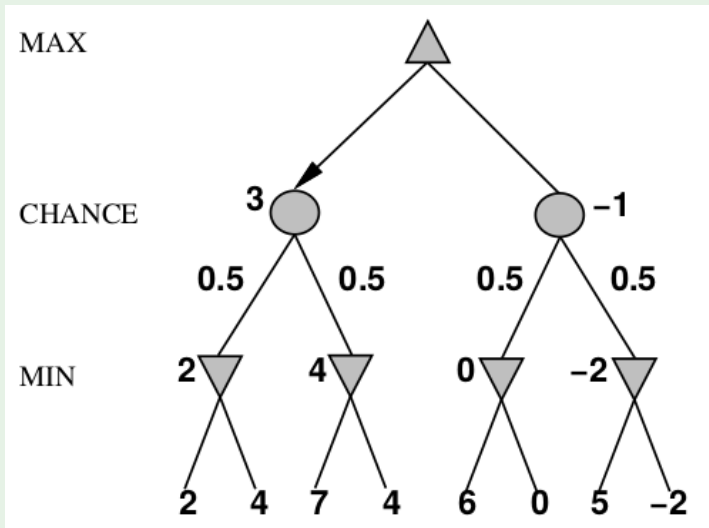
- Extension of *Minimax()*, handling also chance nodes:

$$ExpectMinimax(s) \stackrel{\text{def}}{=} \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ \max_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MAX \\ \min_{a \in Actions(s)} ExpectMinimax(Result(s, a)) & \text{if } Player(s) = MIN \\ \sum_r P(r) \cdot ExpectMinimax(Result(s, r)) & \text{if } Player(s) = Chance \end{cases}$$

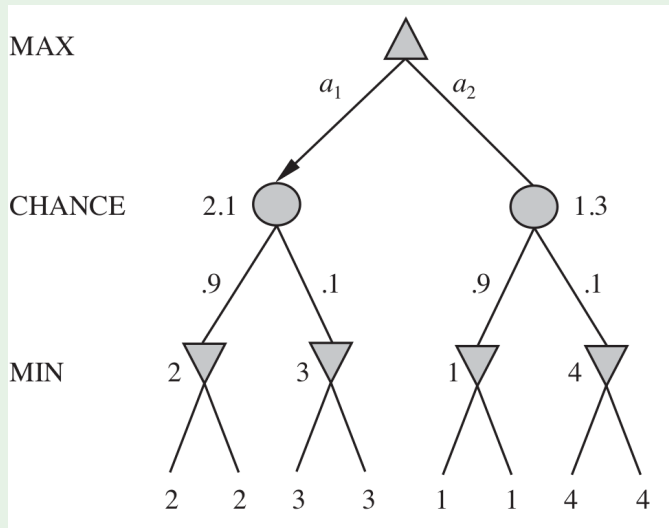
- $P(r)$: probability of stochastic event outcome r
- chance seen as an actor ("Chance")
- stochastic event outcomes r (e.g., dice values) seen as actions

⇒ Returns the weighted average of the minimax outcomes (recall that $\sum_r P(r) = 1$)

Simple Example with Coin-Flipping



Example (Non-uniform Probabilities)



Remark (compare with deterministic case)

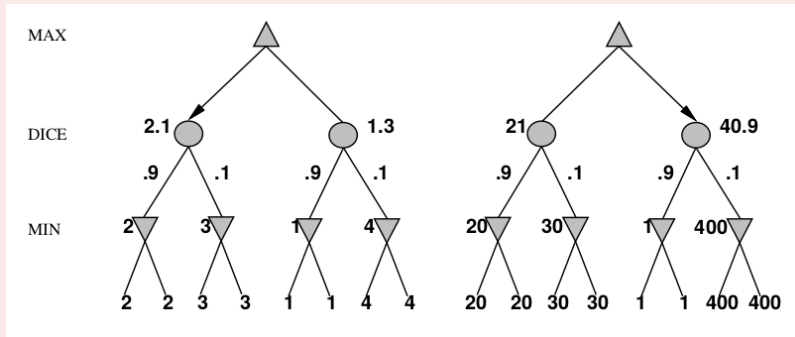
Exact values do matter!

Behaviour **not** preserved under monotonic transformations of $Utility()$

- preserved only by positive linear transformation of $Utility()$

- hint: $p_1 v_1 \geq p_2 v_2 \implies p_1 (av_1 + b) \geq p_2 (av_2 + b)$ if $a \geq 0$

$\implies Utility()$ should be proportional to the expected payoff



Remark (compare with deterministic case)

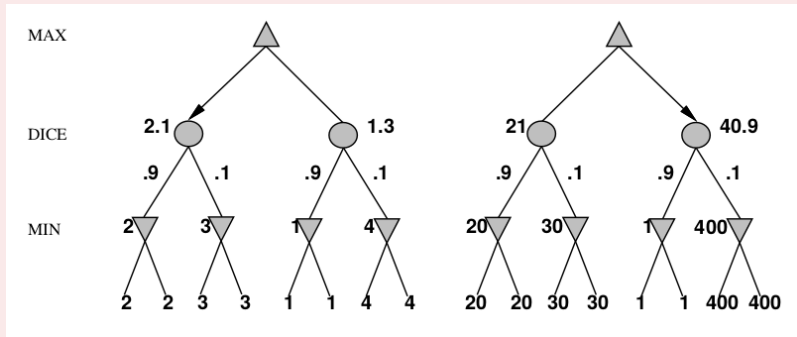
Exact values do matter!

Behaviour **not** preserved under monotonic transformations of $Utility()$

- preserved only by **positive linear transformation** of $Utility()$

- hint: $p_1 v_1 \geq p_2 v_2 \implies p_1 (av_1 + b) \geq p_2 (av_2 + b)$ if $a \geq 0$

$\implies Utility()$ should be proportional to the expected payoff



Remark (compare with deterministic case)

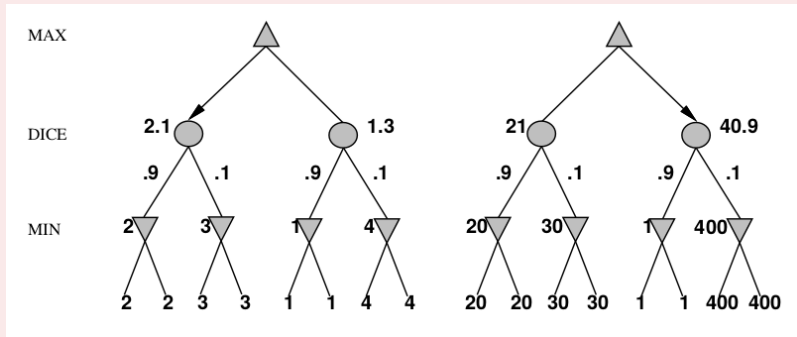
Exact values do matter!

Behaviour **not** preserved under monotonic transformations of $Utility()$

- preserved only by **positive linear transformation** of $Utility()$

- hint: $p_1 v_1 \geq p_2 v_2 \implies p_1 (av_1 + b) \geq p_2 (av_2 + b)$ if $a \geq 0$

$\implies Utility()$ should be proportional to the expected payoff



Example

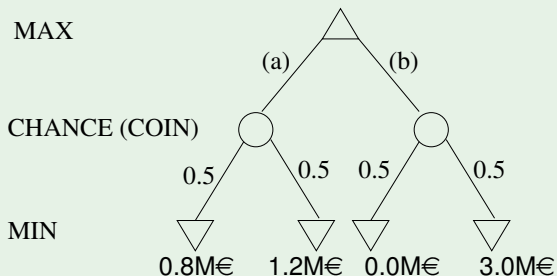
Is money a good utility function?

- Ex: choose between two alternatives in a coin-toss tree:
 - (a) gain 0.8M€ (heads) vs. gain 1.2M€ (tails)
 - (b) gain 0.0M€ (heads) vs. gain 3.0M€ (tails).
- Which one will you choose? Why?
- ExpectMinimax() chooses (b) ($1.5\text{M€} > 1.0\text{M€}$)
- If you choose (a), what is wrong with applying ExpectMinimax() here?

Example

Is money a good utility function?

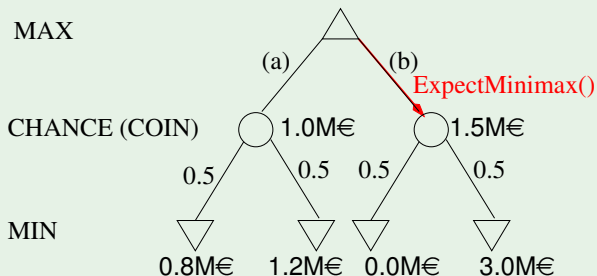
- Ex: choose between two alternatives in a coin-toss tree:
 - (a) gain 0.8M€ (heads) vs. gain 1.2M€ (tails)
 - (b) gain 0.0M€ (heads) vs. gain 3.0M€ (tails).
- Which one will you choose? Why?
- ExpectMinimax() chooses (b) ($1.5\text{M€} > 1.0\text{M€}$)
- If you choose (a), what is wrong with applying ExpectMinimax() here?



Example

Is money a good utility function?

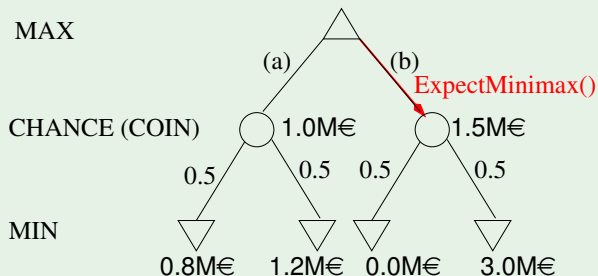
- Ex: choose between two alternatives in a coin-toss tree:
 - (a) gain 0.8M€ (heads) vs. gain 1.2M€ (tails)
 - (b) gain 0.0M€ (heads) vs. gain 3.0M€ (tails).
- Which one will you choose? Why?
- **ExpectMinimax() chooses (b)** ($1.5\text{M€} > 1.0\text{M€}$)
- If you choose (a), what is wrong with applying ExpectMinimax() here?



Example

Is money a good utility function?

- Ex: choose between two alternatives in a coin-toss tree:
 - (a) gain 0.8M€ (heads) vs. gain 1.2M€ (tails)
 - (b) gain 0.0M€ (heads) vs. gain 3.0M€ (tails).
- Which one will you choose? Why?
- ExpectMinimax() chooses (b) ($1.5\text{M€} > 1.0\text{M€}$)
- If you choose (a), what is wrong with applying ExpectMinimax() here?



Stochastic Games in Practice

- Dice rolls increase b : 21 possible rolls with 2 dice

⇒ $O(b^m \cdot n^m)$, n being the number of distinct roll

- Ex: Backgammon has ≈ 20 moves

⇒ depth 4: $20 \cdot (21 \cdot 20)^3 \approx 10^9$ (!)

- Alpha-beta pruning much less effective than with deterministic games

⇒ Unrealistic to consider high depths in most stochastic games

- Heuristic variants of *ExpectMinimax()* effective, low cutoff depths

- Ex: TD-GGAMMON uses depth-2 search + very-good *Eval()*

- *Eval()* “learned” by running million training games
- competitive with world champions

Stochastic Games in Practice

- Dice rolls increase b : 21 possible rolls with 2 dice

⇒ $O(b^m \cdot n^m)$, n being the number of distinct roll

- Ex: Backgammon has ≈ 20 moves

⇒ depth 4: $20 \cdot (21 \cdot 20)^3 \approx 10^9$ (!)

- Alpha-beta pruning much less effective than with deterministic games

⇒ Unrealistic to consider high depths in most stochastic games

- Heuristic variants of *ExpectMinimax()* effective, low cutoff depths

- Ex: TD-GGAMMON uses depth-2 search + very-good *Eval()*

- *Eval()* “learned” by running million training games
- competitive with world champions

Stochastic Games in Practice

- Dice rolls increase b : 21 possible rolls with 2 dice
⇒ $O(b^m \cdot n^m)$, n being the number of distinct roll
- Ex: Backgammon has ≈ 20 moves
⇒ depth 4: $20 \cdot (21 \cdot 20)^3 \approx 10^9$ (!)
- Alpha-beta pruning much less effective than with deterministic games

⇒ Unrealistic to consider high depths in most stochastic games

- Heuristic variants of *ExpectMinimax()* effective, low cutoff depths
- Ex: TD-GGAMMON uses depth-2 search + very-good *Eval()*
 - *Eval()* “learned” by running million training games
 - competitive with world champions

Stochastic Games in Practice

- Dice rolls increase b : 21 possible rolls with 2 dice
⇒ $O(b^m \cdot n^m)$, n being the number of distinct roll
 - Ex: Backgammon has ≈ 20 moves
⇒ depth 4: $20 \cdot (21 \cdot 20)^3 \approx 10^9$ (!)
 - Alpha-beta pruning much less effective than with deterministic games
- ⇒ **Unrealistic to consider high depths in most stochastic games**
- Heuristic variants of *ExpectMinimax()* effective, low cutoff depths
 - Ex: TD-GGAMMON uses depth-2 search + very-good *Eval()*
 - *Eval()* “learned” by running million training games
 - competitive with world champions

Stochastic Games in Practice

- Dice rolls increase b : 21 possible rolls with 2 dice
⇒ $O(b^m \cdot n^m)$, n being the number of distinct roll
- Ex: Backgammon has ≈ 20 moves
⇒ depth 4: $20 \cdot (21 \cdot 20)^3 \approx 10^9$ (!)
- Alpha-beta pruning much less effective than with deterministic games
⇒ **Unrealistic to consider high depths in most stochastic games**
- Heuristic variants of *ExpectMinimax()* effective, low cutoff depths
- Ex: TD-GGAMMON uses depth-2 search + very-good *Eval()*
 - *Eval()* “learned” by running million training games
 - competitive with world champions

Stochastic Games in Practice

- Dice rolls increase b : 21 possible rolls with 2 dice
⇒ $O(b^m \cdot n^m)$, n being the number of distinct roll
- Ex: Backgammon has ≈ 20 moves
⇒ depth 4: $20 \cdot (21 \cdot 20)^3 \approx 10^9$ (!)
- Alpha-beta pruning much less effective than with deterministic games
⇒ **Unrealistic to consider high depths in most stochastic games**
- Heuristic variants of *ExpectMinimax()* effective, low cutoff depths
- Ex: **TD-GGAMMON** uses **depth-2 search + very-good *Eval()***
 - *Eval()* “learned” by running million training games
 - competitive with world champions