Fundamentals of Artificial Intelligence
Chapter 05: **Adversarial Search and Games**

Roberto Sebastiani

DISI, Università di Trento, Italy – roberto.sebastiani@unitn.it
http://disi.unitn.it/rseba/DIDATTICA/fai_2021/

Teaching assistant: Mauro Dragoni – dragoni@fbk.eu
http://www.maurodragoni.com/teaching/fai/

M.S. Course “Artificial Intelligence Systems”, academic year 2021-2022

Last update: Sunday 24th October, 2021, 10:21

Outline

Outline

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 $\Longrightarrow$ 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 $\Longrightarrow$ 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)
- 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)
- 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)
- 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)
- 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)
- 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)
- 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)
- 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

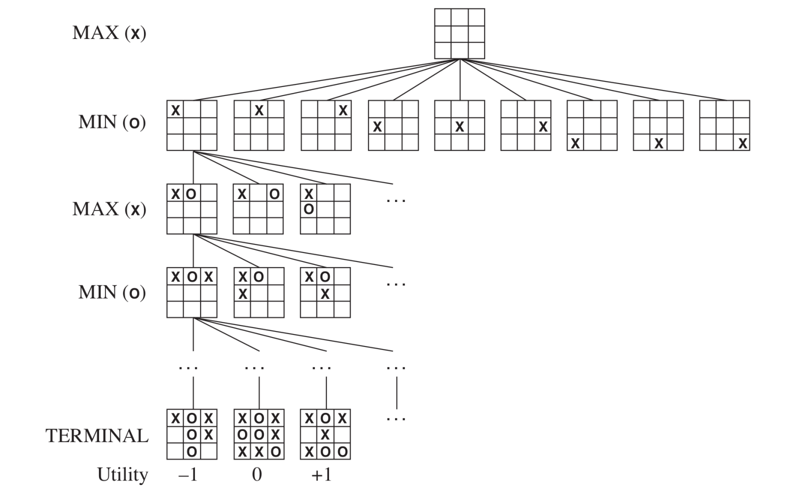
- We first consider games with two players: "MAX" and "MIN"
  - MAX moves first;
  - they take turns moving until the game is over
  - at the end of the game, points are awarded to the winner and penalties are given to the loser
- A game is a kind of search problem:
  - Initial state S_0: specifies how 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)
- 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)
- 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)



(© S. Russell & P. Norwig, AIMA)

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)

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

- one agent maximizes it, the other minimizes it

$\Longrightarrow$ 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)

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

- one agent maximizes it, the other minimizes it

$\Longrightarrow$ 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)

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

- one agent maximizes it, the other minimizes it

$\Longrightarrow$ 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)

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

- one agent maximizes it, the other minimizes it

$\Longrightarrow$ optimal adversarial search as min-max search

Outline

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 $Minimax(s)$ 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 $Minimax(s)$ 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 $Minimax(s)$ 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 $Minimax(s)$ 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 $Minimax(s)$ returns the highest value

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

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 $Minimax(s)$ 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 $Minimax(s)$ 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 $Minimax(s)$ 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 $Minimax(s)$ returns the highest value

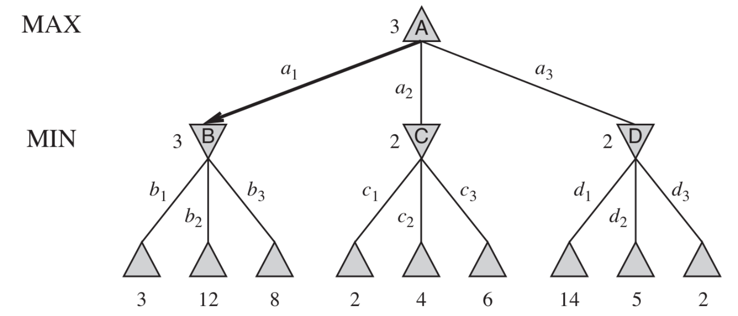
$$Minimax(s) \stackrel{\text{def}}{=} \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ max_{a \in Actions(s)} Minimax(Result(s,a)) & \text{if } Player(s) = MAX \\ min_{a \in Actions(s)} Minimax(Result(s,a)) & \text{if } Player(s) = MIN \end{cases}$$

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



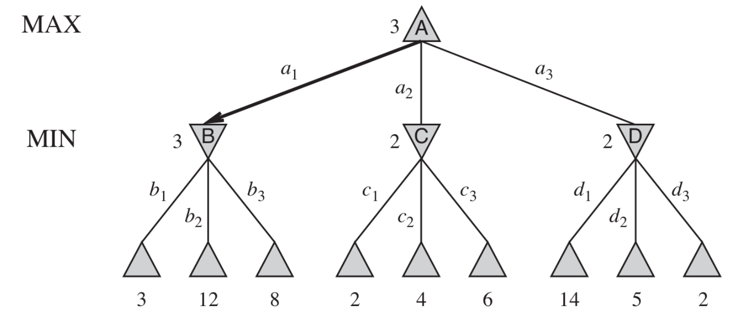
(© S. Russell & P. Norwig, AIMA)

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

$\Longrightarrow$ MAX's root best move is a_1



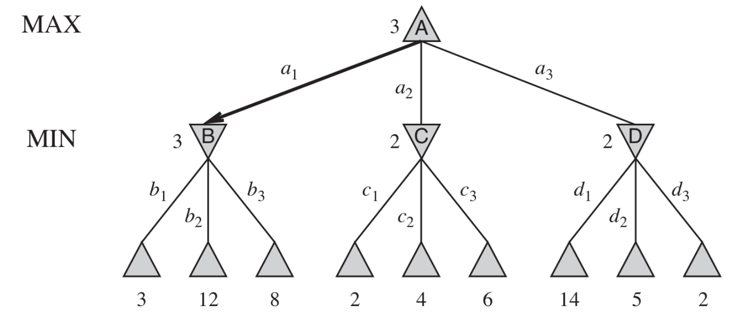
(© S. Russell & P. Norwig, AIMA)

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



(© S. Russell & P. Norwig, AIMA)

The Minimax Algorithm

Depth-First Search Minimax Algorithm

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

Multi-Player Games: Optimal Decisions

- Replace the single value for each node with a vector of values
  - 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
  - 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

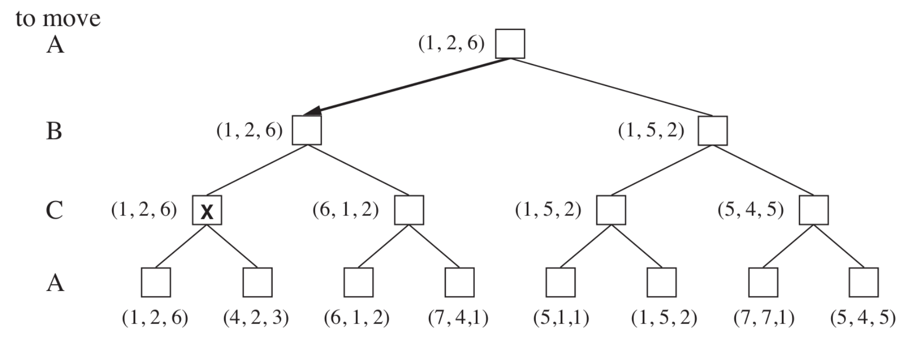
Multi-Player Games: Optimal Decisions

- Replace the single value for each node with a vector of values
  - 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
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2) $\Longrightarrow$ benefit for both

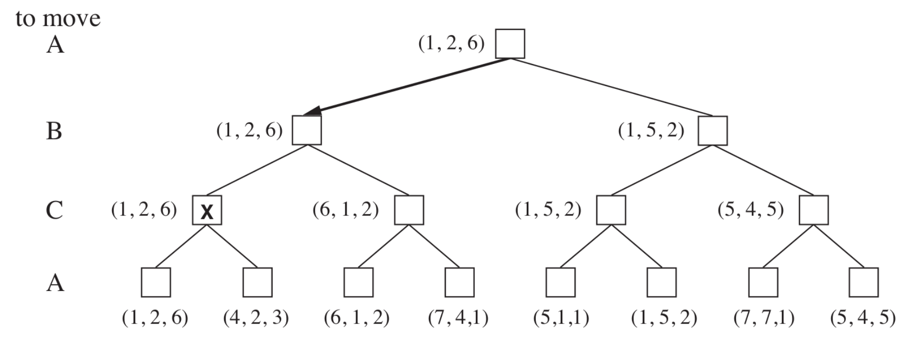


(© S. Russell & P. Norwig, AIMA)

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
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2) $\Longrightarrow$ benefit for both

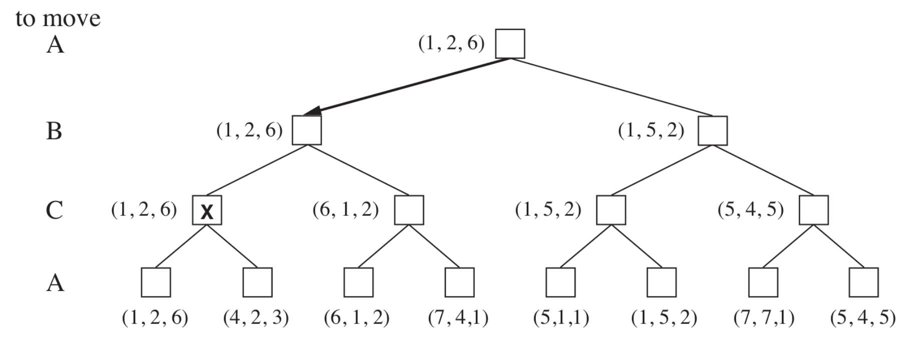


(© S. Russell & P. Norwig, AIMA)

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
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2) $\Longrightarrow$ benefit for both

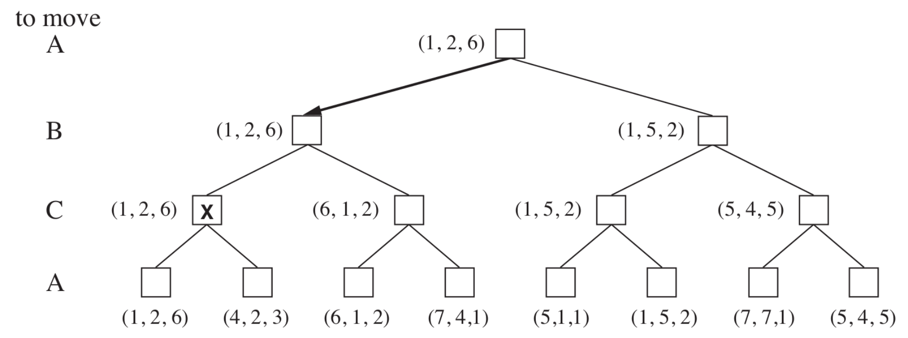


(© S. Russell & P. Norwig, AIMA)

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
- If A and B are allied, then they may agree that B and then A choose (5,4,5) instead of (1,5,2) $\Longrightarrow$ benefit for both



(© S. Russell & P. Norwig, AIMA)

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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

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?

  $\Longrightarrow$ 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?

  $\Longrightarrow$ 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?

  $\Longrightarrow$ 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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

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?

  $\Longrightarrow$ 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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

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?

  $\Longrightarrow$ 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?

  $\Longrightarrow$ even better, but non optimal in this case
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (DFS)

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

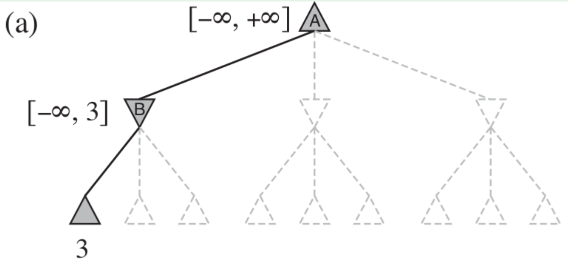
We need to prune the tree!

Outline

Pruning Min-Max Search: Example

- Consider the previous execution of the Minimax algorithm
- Let [*min*, *max*] track the currently-known bounds for the search
  - (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)
  - (d): Is it necessary to evaluate the remaining leaves of *C*?
    NO! They cannot produce an upper bound ≥ 2
    $\Longrightarrow$ MAX cannot update the $min = 3$ bound due to C
  - (e): MAX updates the upper bound to 14 (D is last subtree)
  - (f): D labeled $[2, 2]$ $\Longrightarrow$ MAX updates the upper bound to 3

$\Longrightarrow$ 3 final value

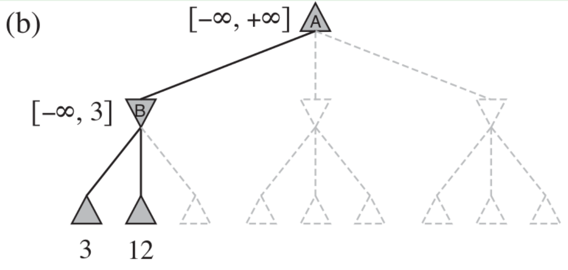


(© S. Russell & P. Norwig, AIMA)

Pruning Min-Max Search: Example

- Consider the previous execution of the Minimax algorithm
- Let [*min*, *max*] track the currently-known bounds for the search
  - (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)
  - (d): Is it necessary to evaluate the remaining leaves of *C*?
    NO! They cannot produce an upper bound ≥ 2
    $\Longrightarrow$ MAX cannot update the $min = 3$ bound due to C
  - (e): MAX updates the upper bound to 14 (D is last subtree)
  - (f): D labeled $[2, 2] \Longrightarrow$ MAX updates the upper bound to 3

$\Longrightarrow$ 3 final value

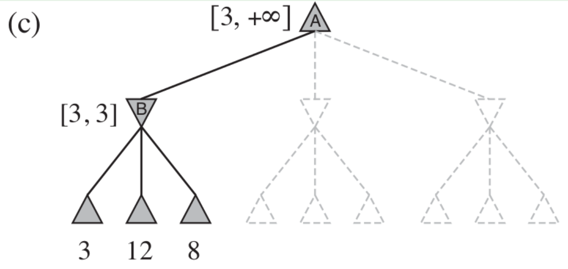


(© S. Russell & P. Norwig, AIMA)

Pruning Min-Max Search: Example

- Consider the previous execution of the Minimax algorithm
- Let [*min*, *max*] track the currently-known bounds for the search
  - (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)
  - (d): Is it necessary to evaluate the remaining leaves of C?
    NO! They cannot produce an upper bound ≥ 2
    $\Longrightarrow$ MAX cannot update the $min = 3$ bound due to C
  - (e): MAX updates the upper bound to 14 (D is last subtree)
  - (f): D labeled $[2, 2]$ $\Longrightarrow$ MAX updates the upper bound to 3

$\Longrightarrow$ 3 final value

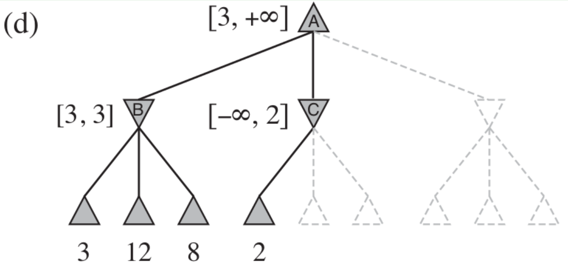


(© S. Russell & P. Norwig, AIMA)

Pruning Min-Max Search: Example

- Consider the previous execution of the Minimax algorithm
- Let [*min*, *max*] track the currently-known bounds for the search
  - (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)
  - (d): Is it necessary to evaluate the remaining leaves of *C*?
    NO! They cannot produce an upper bound ≥ 2
    $\Longrightarrow$ MAX cannot update the *min* $= 3$ bound due to C
  - (e): MAX updates the upper bound to 14 (D is last subtree)
  - (f): D labeled $[2, 2] \Longrightarrow$ MAX updates the upper bound to 3

$\Longrightarrow$ 3 final value

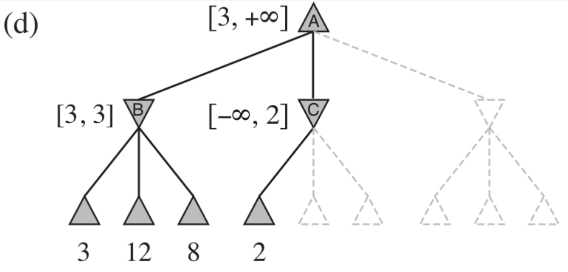


(© S. Russell & P. Norwig, AIMA)

Pruning Min-Max Search: Example

- Consider the previous execution of the Minimax algorithm
- Let [*min*, *max*] track the currently-known bounds for the search
  - (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)
  - (d): Is it necessary to evaluate the remaining leaves of C?
    NO! They cannot produce an upper bound ≥ 2
    $\Longrightarrow$ MAX cannot update the $min = 3$ bound due to C
  - (e): MAX updates the upper bound to 14 (D is last subtree)
  - (f): D labeled $[2, 2]$ $\Longrightarrow$ MAX updates the upper bound to 3

$\Longrightarrow$ 3 final value

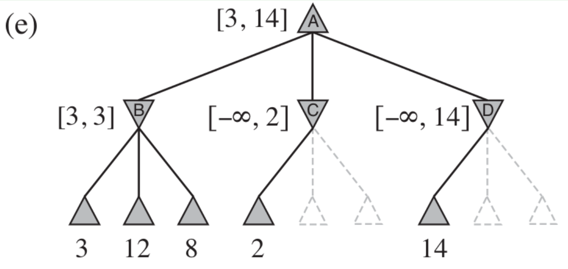


(© S. Russell & P. Norwig, AIMA)

Pruning Min-Max Search: Example

- Consider the previous execution of the Minimax algorithm
- Let [*min*, *max*] track the currently-known bounds for the search
  - (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)
  - (d): Is it necessary to evaluate the remaining leaves of *C*?
    NO! They cannot produce an upper bound ≥ 2
    $\Longrightarrow$ MAX cannot update the $min = 3$ bound due to C
  - (e): MAX updates the upper bound to 14 (D is last subtree)
  - (f): D labeled $[2, 2]$ $\Longrightarrow$ MAX updates the upper bound to 3

$\Longrightarrow$ 3 final value

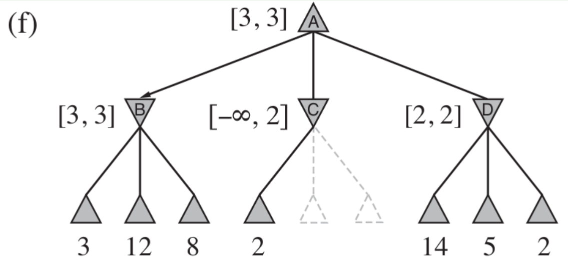


(© S. Russell & P. Norwig, AIMA)

Pruning Min-Max Search: Example

- Consider the previous execution of the Minimax algorithm
- Let [*min*, *max*] track the currently-known bounds for the search
  - (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)
  - (d): Is it necessary to evaluate the remaining leaves of *C*?
    NO! They cannot produce an upper bound ≥ 2
    $\Longrightarrow$ MAX cannot update the *min* $= 3$ bound due to C
  - (e): MAX updates the upper bound to 14 (D is last subtree)
  - (f): D labeled $[2, 2]$ $\Longrightarrow$ MAX updates the upper bound to 3

$\Longrightarrow$ 3 final value



(© S. Russell & P. Norwig, AIMA)

Alpha-Beta Pruning Technique for Min-Max Search

- Idea: consider a node n (terminal or intermediate)
  - If player has a better choice m at the parent node of n or at any choice point further up, n will never be reached in actual play

  $\Longrightarrow$ if we know enough of n to draw this conclusion, we can prune n
- Alpha-Beta Pruning: nodes labeled with $[\alpha, \beta]$ s.t.:

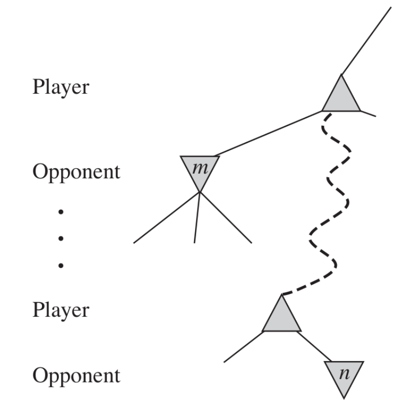
  α : best value for MAX (highest) so far off the current path

  $\Longrightarrow$ lower bound for future values

  β : best value for MIN (lowest) so far off the current path

  $\Longrightarrow$ upper bound for future values

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



(© S. Russell & P. Norwig, AIMA)

Alpha-Beta Pruning Technique for Min-Max Search

- Idea: consider a node n (terminal or intermediate)
  - If player has a better choice m at the parent node of n or at any choice point further up, n will never be reached in actual play

  $\Longrightarrow$ if we know enough of n to draw this conclusion, we can prune n
- Alpha-Beta Pruning: nodes labeled with $[\alpha, \beta]$ s.t.:

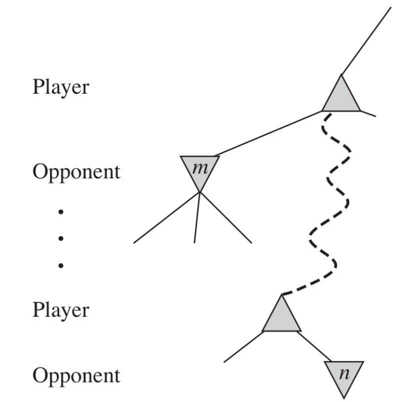
  α : best value for MAX (highest) so far off the current path

  $\Longrightarrow$ lower bound for future values

  β : best value for MIN (lowest) so far off the current path

  $\Longrightarrow$ upper bound for future values

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



(© S. Russell & P. Norwig, AIMA)

Alpha-Beta Pruning Technique for Min-Max Search

- Idea: consider a node n (terminal or intermediate)
  - If player has a better choice m at the parent node of n or at any choice point further up, n will never be reached in actual play

  $\Longrightarrow$ if we know enough of n to draw this conclusion, we can prune n
- Alpha-Beta Pruning: nodes labeled with $[\alpha, \beta]$ s.t.:

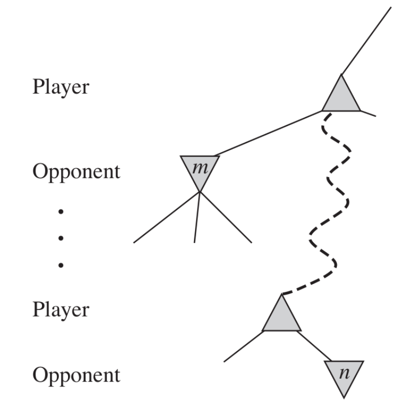
  α : best value for MAX (highest) so far off the current path

  $\Longrightarrow$ lower bound for future values

  β : best value for MIN (lowest) so far off the current path

  $\Longrightarrow$ upper bound for future values

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



(© S. Russell & P. Norwig, AIMA)

The Alpha-Beta Search Algorithm

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

---

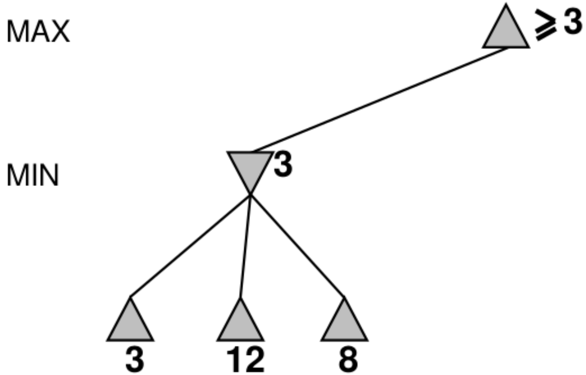
function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Example revisited: Alpha-Beta Cuts

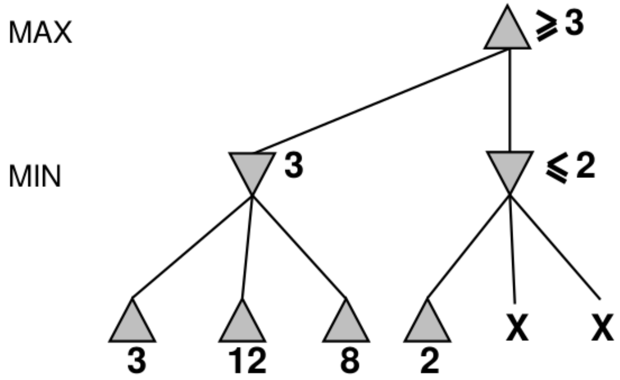
- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

Example revisited: Alpha-Beta Cuts

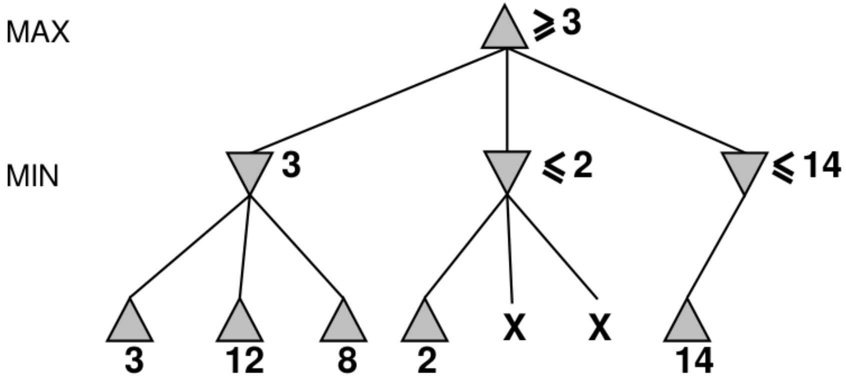
- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

Example revisited: Alpha-Beta Cuts

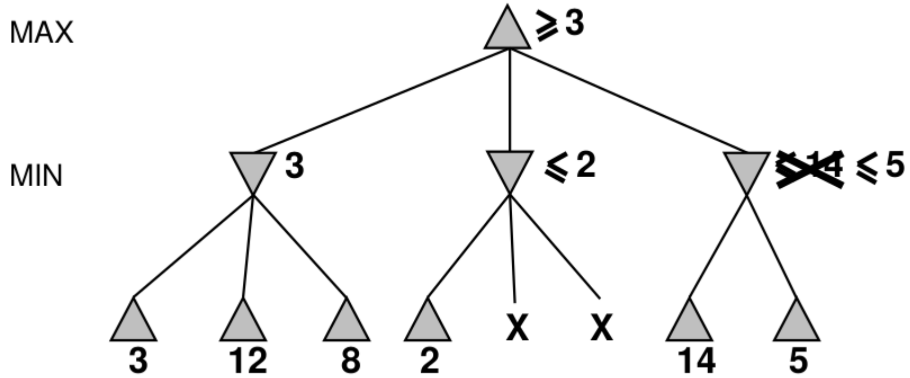
- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

Example revisited: Alpha-Beta Cuts

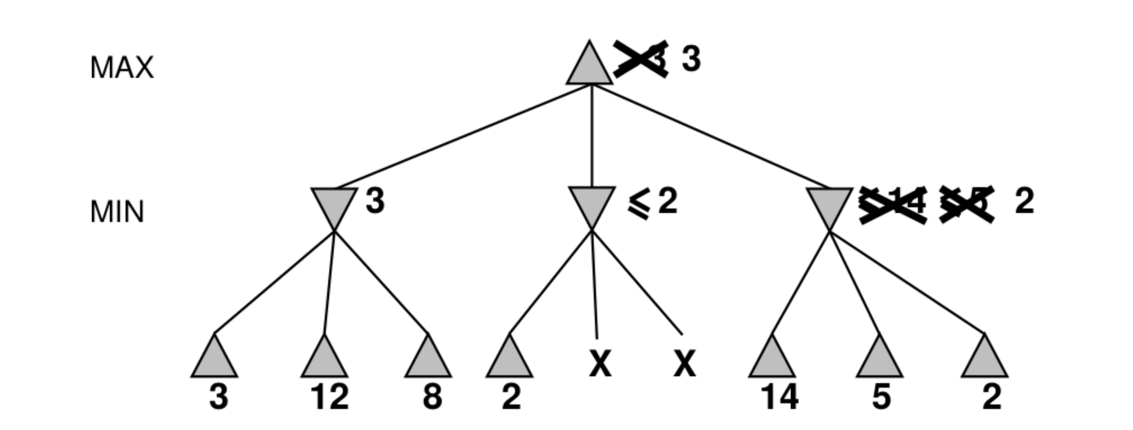
- Notation: $\geq \alpha$; $\leq \beta$;



(© S. Russell & P. Norwig, AIMA)

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 $\Longrightarrow$ correctness preserved
- Good move ordering improves effectiveness of pruning
  - Ex: if MIN expands 3^{rd} child of D first, the others are pruned
  - 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"

  $\Longrightarrow$ 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 $\Longrightarrow$ correctness preserved
- Good move ordering improves effectiveness of pruning
  - Ex: if MIN expands 3rd child of D first, the others are pruned
  - 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"

  $\Longrightarrow$ 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 $\Longrightarrow$ correctness preserved
- Good move ordering improves effectiveness of pruning
  - Ex: if MIN expands 3^{rd} child of D first, the others are pruned
  - 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"

  $\Longrightarrow$ 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 $\Longrightarrow$ correctness preserved
- Good move ordering improves effectiveness of pruning
  - Ex: if MIN expands 3rd child of D first, the others are pruned
  - 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”

  $\Longrightarrow$ 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 $\Longrightarrow$ correctness preserved
- Good move ordering improves effectiveness of pruning
  - Ex: if MIN expands 3^{rd} child of D first, the others are pruned
  - 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"

  $\Longrightarrow$ 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

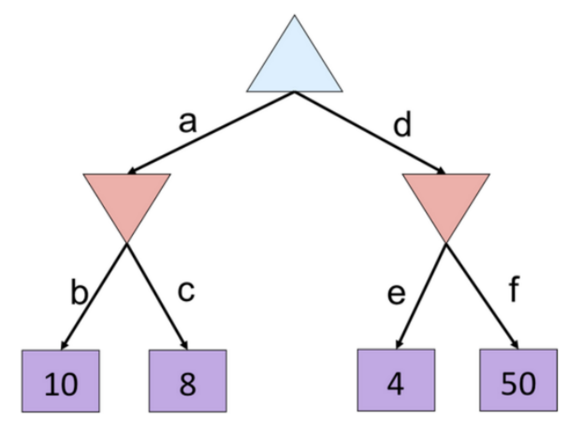
Properties of Alpha-Beta Search

- Pruning does not affect the final result $\Longrightarrow$ correctness preserved
- Good move ordering improves effectiveness of pruning
  - Ex: if MIN expands 3^{rd} child of D first, the others are pruned
  - 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”

  $\Longrightarrow$ 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

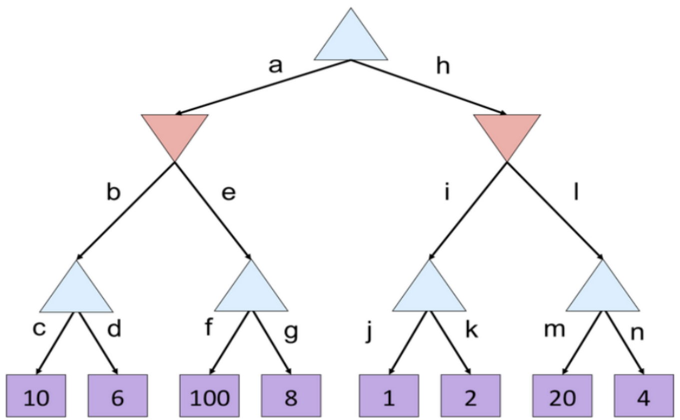
Exercise I

Apply alpha-beta search to the following tree



Exercise II

Apply alpha-beta search to the following tree

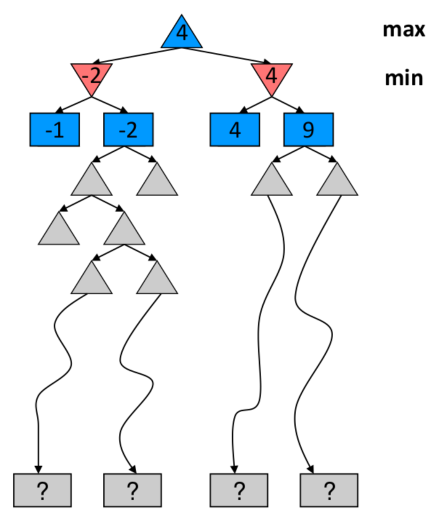


Outline

Adversarial Search with Resource Limits

Problem: In realistic games, full search is impractical!

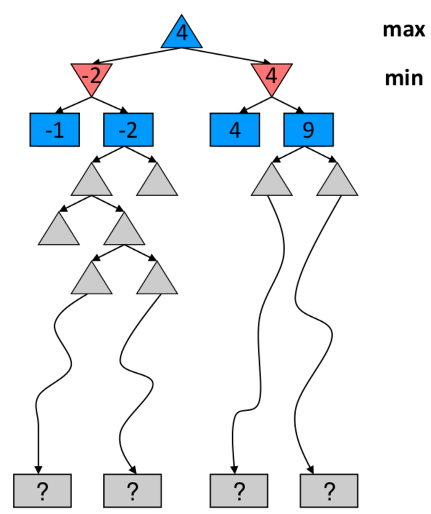
- Complexity: b^d (ex. chess: $\approx 35^{100}$)
- Idea [Shannon, 1949]: Depth-limited search
  - cut off minimax search earlier, after limited depth
  - replace terminal utility function with evaluation for non-terminal nodes
- Ex (chess): depth $d = 8$ (decent) $\Longrightarrow$ α-β: $35^{8/2} = 10^5$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



Adversarial Search with Resource Limits

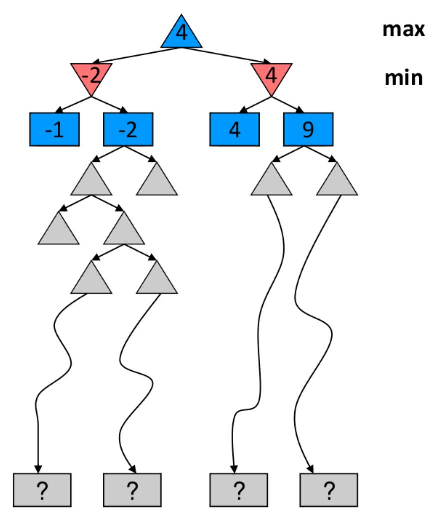
Problem: In realistic games, full search is impractical!

- 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)
  $\Longrightarrow \alpha$-β: $35^{8/2} = 10^5$ (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)
  $\Longrightarrow \alpha$-β: $35^{8/2} = 10^5$ (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

  $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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

  $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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
  - $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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

  $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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
  - $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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

  $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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

  $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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

  $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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

  $\Longrightarrow$ 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

$\Longrightarrow$ Heuristic variant of *Minimax*():

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

$\Longrightarrow$ 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$, $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- May be very inaccurate for some positions

Evaluation Functions

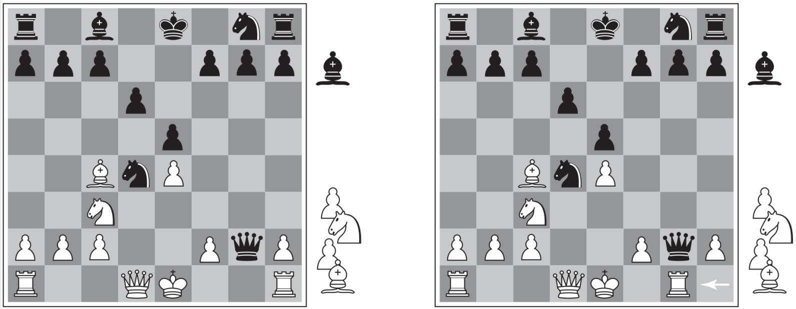
$Eval(s)$

- 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:
  $Eval(s) = w_1 \cdot f_1(s) + w_2 \cdot f_2(s) + ... + w_n \cdot f_n(s)$
  - ex (chess): $f_{queens}(s) = \#white\ queens - \#black\ queens$,
    $w_{pawns} = 1$: $w_{bishops} = w_{knights} = 3$, $w_{rooks} = 5$, $w_{queens} = 9$
- May depend on depth (ex: knights vs. rooks)
- May be very inaccurate for some positions

Example

- Two same-score positions (White: -8, Black: -3)
  - (a) Black has an advantage of a knight and two pawns,
    $\Longrightarrow$ should be enough to win the game
  - (b) White will capture the queen,
    $\Longrightarrow$ give it an advantage that should be strong enough to win

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

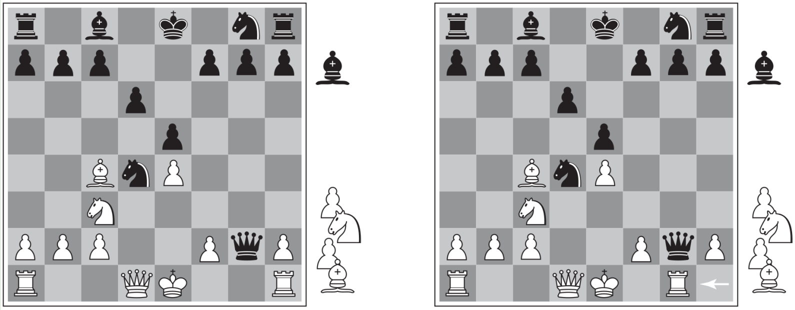


(a) White to move (b) White to move

Example

- Two same-score positions (White: -8, Black: -3)
  (a) Black has an advantage of a knight and two pawns,
  $\Longrightarrow$ should be enough to win the game
  (b) White will capture the queen,
  $\Longrightarrow$ give it an advantage that should be strong enough to win

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

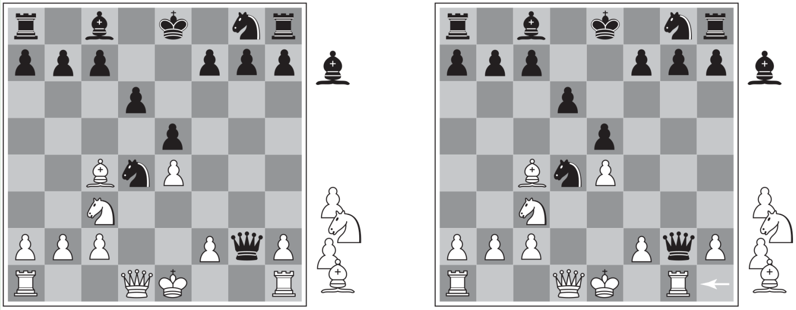


(a) White to move (b) White to move

Example

- Two same-score positions (White: -8, Black: -3)
  - (a) Black has an advantage of a knight and two pawns,
    $\Longrightarrow$ should be enough to win the game
  - (b) White will capture the queen,
    $\Longrightarrow$ give it an advantage that should be strong enough to win

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



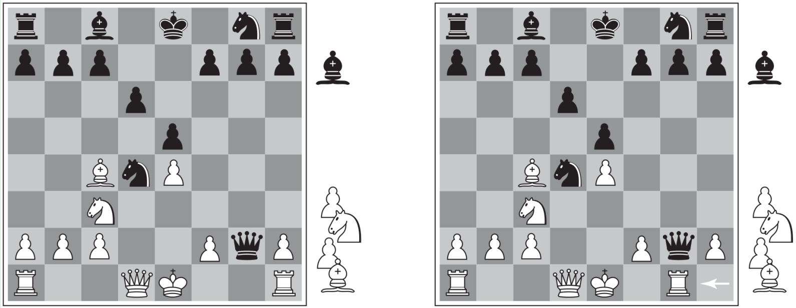
(a) White to move (b) White to move

(© S. Russell & P. Norwig, AIMA)

Example

- Two same-score positions (White: -8, Black: -3)
  - (a) Black has an advantage of a knight and two pawns, $\Longrightarrow$ should be enough to win the game
  - (b) White will capture the queen, $\Longrightarrow$ give it an advantage that should be strong enough to win

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



(a) White to move (b) White to move

(© S. Russell & P. Norwig, AIMA)

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 sophisticate: 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))

$\Longrightarrow$ 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 sophisticate: 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))

$\Longrightarrow$ 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 sophisticate: 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))

$\Longrightarrow$ 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 sophisticate: 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))

$\Longrightarrow$ 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 sophisticate: 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))

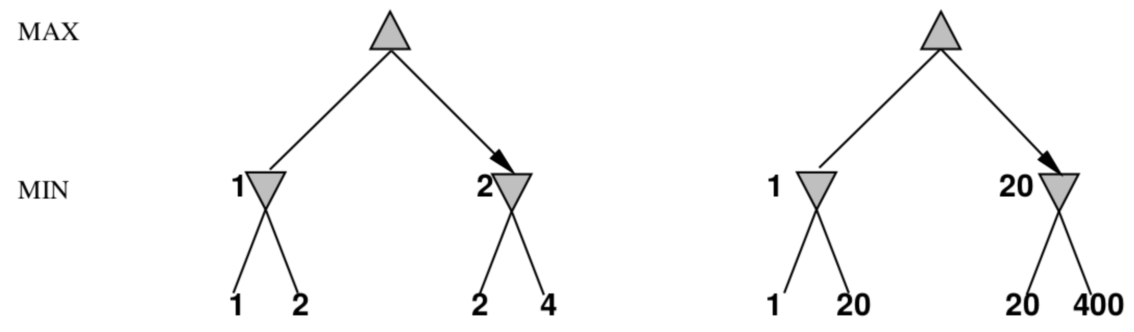
$\Longrightarrow$ further expand non-quiescent states until quiescence is reached

Remark

Exact values don't matter!

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

- Only the order matters!
- payoff in deterministic games acts as an ordinal utility function



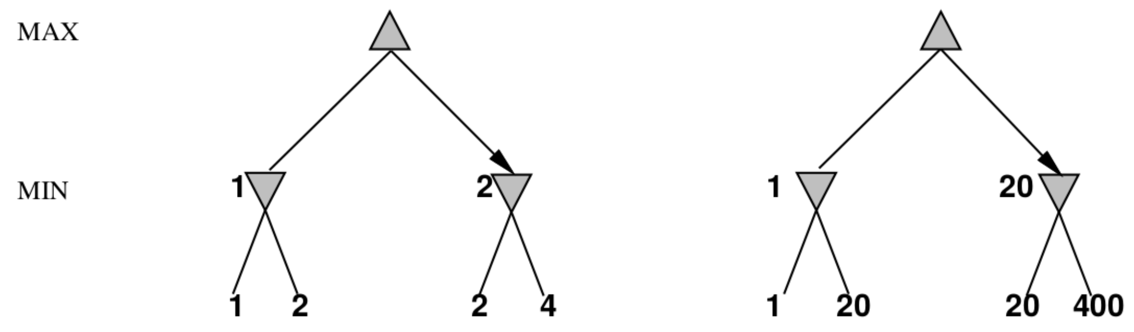
(© S. Russell & P. Norwig, AIMA)

Remark

Exact values don't matter!

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

- Only the order matters!
- payoff in deterministic games acts as an ordinal utility function



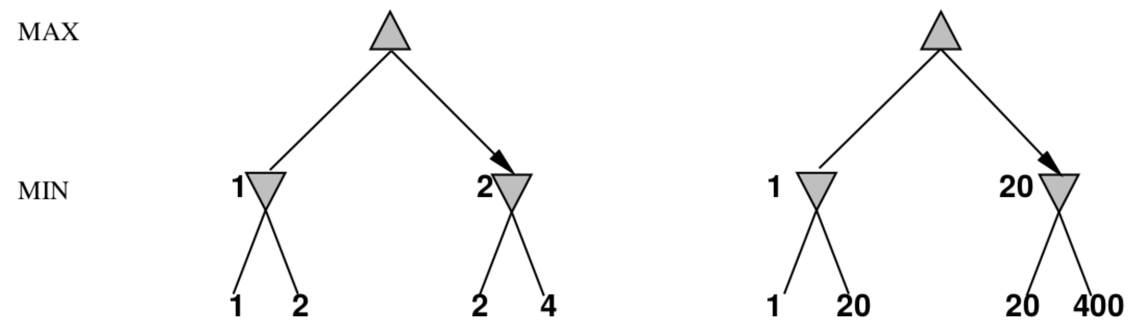
(© S. Russell & P. Norwig, AIMA)

Remark

Exact values don't matter!

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

- Only the order matters!
- payoff in deterministic games acts as an ordinal utility function



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

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

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 $\Longrightarrow$ worst case
  - chance $\Longrightarrow$ 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 $\Longrightarrow$ worst case
  - chance $\Longrightarrow$ 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 $\Longrightarrow$ worst case
  - chance $\Longrightarrow$ average case
- Ex: if chance is 0.5 each (coin):
  - minimax: 10
  - average: (100+9)/2=54.5

Stochastic Games: Generalities

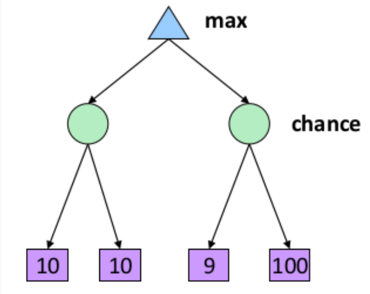
- In real life, unpredictable external events may occur
- Stochastic Games mirror unpredictability by random steps:
  - e.g. dice throwing, card-shuffling, coin flipping, tile extraction, ...
- Ex: Backgammon
- Cannot calculate definite minimax value, only expected values
- Uncertain outcomes controlled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

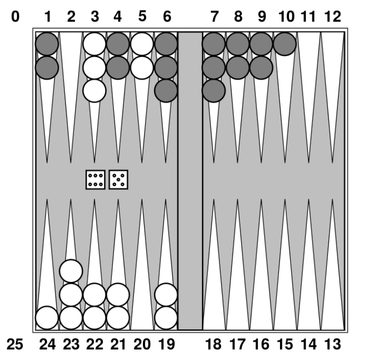
Stochastic Games: Generalities

- 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 $\Longrightarrow$ worst case
  - chance $\Longrightarrow$ 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
  $\Longrightarrow$ stochastic component (dice)
  - double rolls (1-1),...,(6-6)
    have 1/36 probability each
  - other 15 distinct rolls
    have a 1/18 probability each

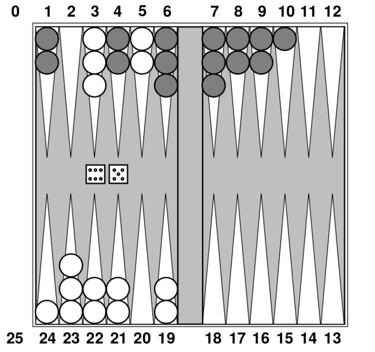


(© S. Russell & P. Norwig, AIMA)

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
  $\Longrightarrow$ stochastic component (dice)
  - double rolls (1-1),...,(6-6) have 1/36 probability each
  - other 15 distinct rolls have a 1/18 probability each

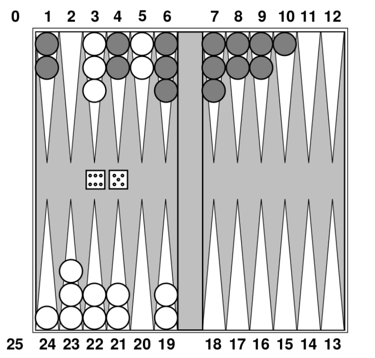


(© S. Russell & P. Norwig, AIMA)

An Example: Backgammon

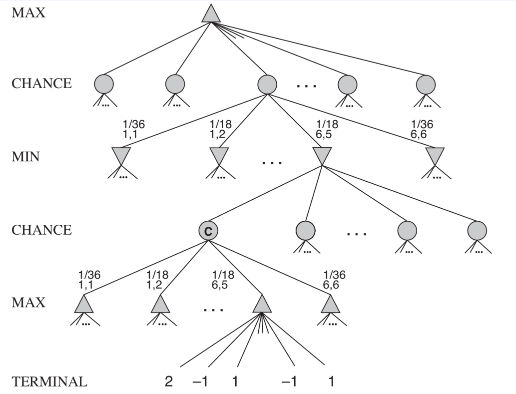
- Rules
  - 15 pieces each
  - white moves clockwise to 25, black moves counterclockwise to 0
  - a piece can move to a position unless ≥ 2 opponent pieces there
  - if there is one opponent, it is captured and must start over
  - termination: all whites in 25 or all blacks in 0
- Ex: Possible white moves (dice: 6,5):

  (5-10,5-11)
  (5-11,19-24)
  (5-10,10-16)
  (5-11,11-16)
- Combines strategy with luck
  $\Longrightarrow$ stochastic component (dice)
  - double rolls (1-1),...,(6-6) have 1/36 probability each
  - other 15 distinct rolls have a 1/18 probability each



Stochastic Games Trees

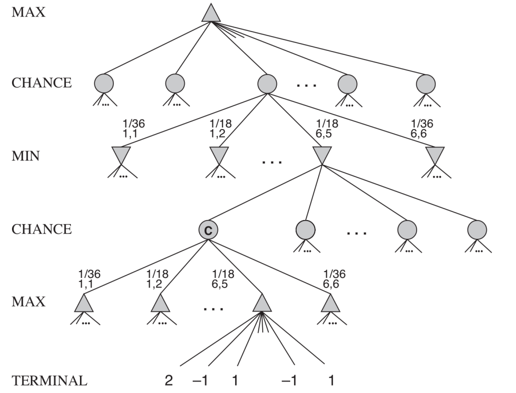
- Idea: A game tree for a stochastic game includes chance nodes in addition to MAX and MIN nodes.
  - chance nodes above agent represent stochastic events for agent (e.g. dice roll)
  - outcoming arcs represent stochastic event outcomes
  - labeled with stochastic event and relative probability



(© S. Russell & P. Norwig, AIMA)

Stochastic Games Trees

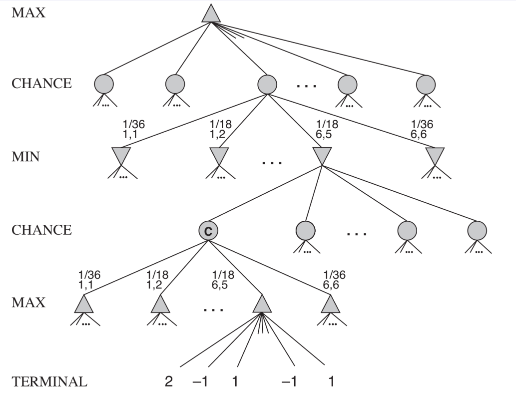
- Idea: A game tree for a stochastic game includes chance nodes in addition to MAX and MIN nodes.
  - chance nodes above agent represent stochastic events for agent (e.g. dice roll)
  - outcoming arcs represent stochastic event outcomes
  - labeled with stochastic event and relative probability



(© S. Russell & P. Norwig, AIMA)

Stochastic Games Trees

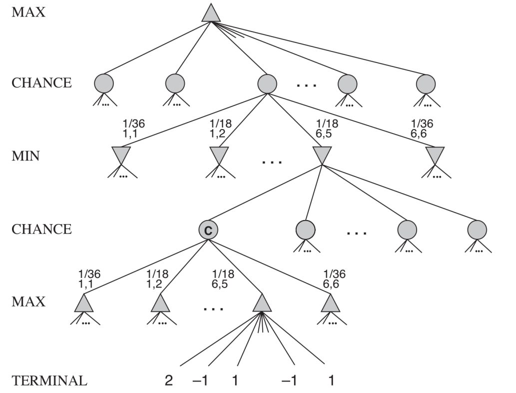
- Idea: A game tree for a stochastic game includes chance nodes in addition to MAX and MIN nodes.
  - chance nodes above agent represent stochastic events for agent (e.g. dice roll)
  - outcoming arcs represent stochastic event outcomes
  - labeled with stochastic event and relative probability



(© S. Russell & P. Norwig, AIMA)

Stochastic Games Trees

- Idea: A game 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)
  - outcoming arcs represent stochastic event outcomes
  - labeled with stochastic event and relative probability



(© S. Russell & P. Norwig, AIMA)

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

$\Longrightarrow$ 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

$\Longrightarrow$ 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

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

Algorithm for Stochastic Games: *ExpectMinimax*()

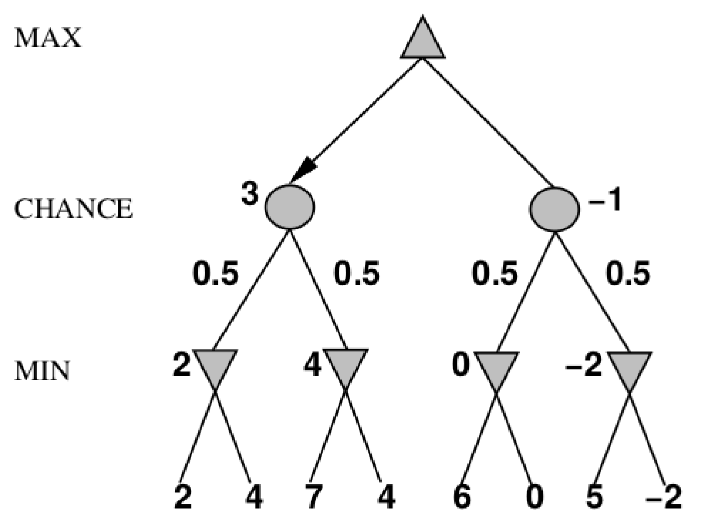
- Extension of *Minimax*(), handling also chance nodes:

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

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

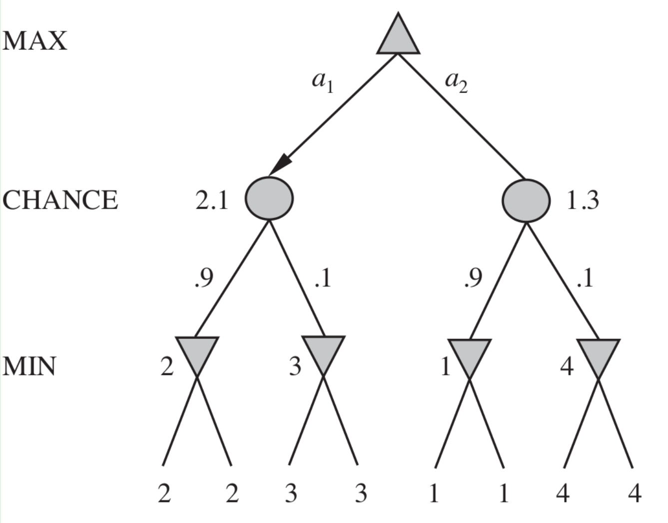
$\Longrightarrow$ Returns the weighted average of the minimax outcomes (recall that $\sum_r P(r) = 1$))

Simple Example with Coin-Flipping



(© S. Russell & P. Norwig, AIMA)

Example (Non-uniform Probabilities)



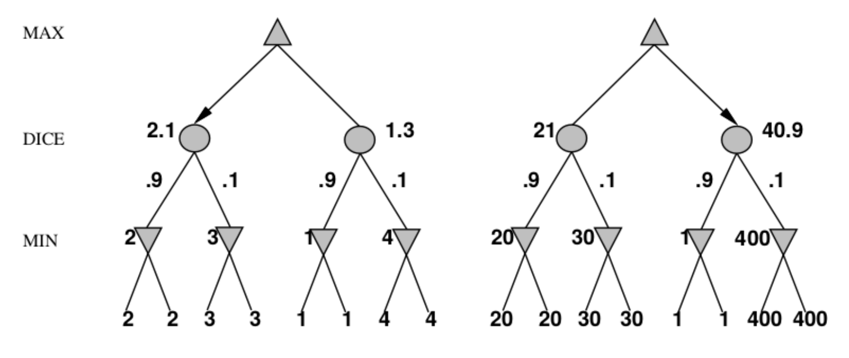
Remark (compare with deterministic case)

Exact values do matter!

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

- preserved only by positive linear transformation of *Utility*()
  - hint: $p_1 v_1 \geq p_2 v_2 \Longrightarrow p_1(av_1 + b) \geq p_2(av_2 + b)$ if $a \geq 0$

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



(© S. Russell & P. Norwig, AIMA)

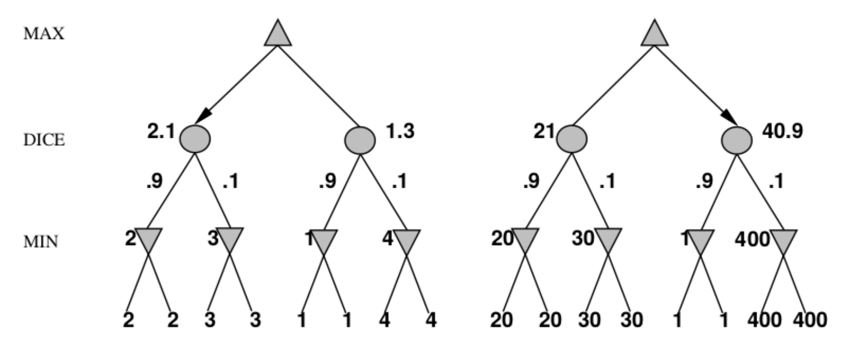
Remark (compare with deterministic case)

Exact values do matter!

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

- preserved only by positive linear transformation of *Utility*()
  - hint: $p_1 v_1 \geq p_2 v_2 \Longrightarrow p_1(av_1 + b) \geq p_2(av_2 + b)$ if $a \geq 0$

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



(© S. Russell & P. Norwig, AIMA)

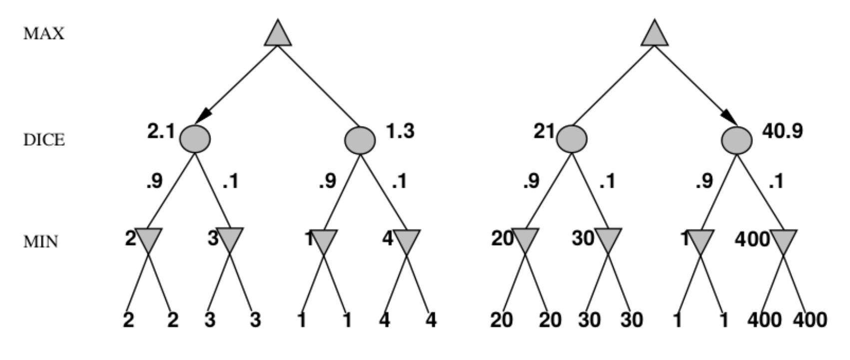
Remark (compare with deterministic case)

Exact values do matter!

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

- preserved only by positive linear transformation of $Utility()$
  - hint: $p_1 v_1 \geq p_2 v_2 \Longrightarrow p_1(av_1 + b) \geq p_2(av_2 + b)$ if $a \geq 0$

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



(© S. Russell & P. Norwig, AIMA)

Stochastic Games in Practice

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

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

$\Longrightarrow$ Unrealistic to consider high depths in most stochastic games

- Heuristic variants of *ExpectMinimax*() effective, low cutoff depths
- Ex: TD-GAMMON 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
  $\Longrightarrow$ $O(b^m \cdot n^m)$, n being the number of distinct roll
- Ex: Backgammon has ≈ 20 moves
  $\Longrightarrow$ depth 4: $20 \cdot (21 \times 20)^3 \approx 10^9$ (!)
- Alpha-beta pruning much less effective than with deterministic games

$\Longrightarrow$ Unrealistic to consider high depths in most stochastic games

- Heuristic variants of *ExpectMinimax*() effective, low cutoff depths
- Ex: TD-GAMMON 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
  $\Longrightarrow$ $O(b^m \cdot n^m)$, n being the number of distinct roll
- Ex: Backgammon has ≈ 20 moves
  $\Longrightarrow$ depth 4: $20 \cdot (21 \times 20)^3 \approx 10^9$ (!)
- Alpha-beta pruning much less effective than with deterministic games

$\Longrightarrow$ Unrealistic to consider high depths in most stochastic games

- Heuristic variants of *ExpectMinimax*() effective, low cutoff depths
- Ex: TD-GAMMON 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
  $\Longrightarrow$ $O(b^m \cdot n^m)$, n being the number of distinct roll
- Ex: Backgammon has $\approx$ 20 moves
  $\Longrightarrow$ depth 4: $20 \cdot (21 \times 20)^3 \approx 10^9$ (!)
- Alpha-beta pruning much less effective than with deterministic games

$\Longrightarrow$ Unrealistic to consider high depths in most stochastic games

- Heuristic variants of *ExpectMinimax*() effective, low cutoff depths
- Ex: TD-GAMMON 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

  $\Longrightarrow$ $O(b^m \cdot n^m)$, n being the number of distinct roll
- Ex: Backgammon has $\approx$ 20 moves

  $\Longrightarrow$ depth 4: $20 \cdot (21 \times 20)^3 \approx 10^9$ (!)
- Alpha-beta pruning much less effective than with deterministic games

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

$\Longrightarrow$ Unrealistic to consider high depths in most stochastic games

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