Game Playing (Adverserial Search)

Games vs. Search Problems

- "Unpredictable" opponent → solution is a strategy specifying a move for every possible opponent reply
- Time limits → unlikely to find goal, must approximate
- Plan of attack:
  - Computer considers possible lines of play (Babbage, 1846)
  - Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
  - Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
  - First chess program (Turing, 1951)
  - Machine learning to improve evaluation accuracy (Samuel, 1952-57)
  - Pruning to allow deeper search (McCarthy, 1956)

Games

- In game theory (economics), any multi-agent environment (either cooperative or competitive) is a game provided that the impact of each agent on the other is significant,
- AI games are a specialized kind - deterministic, turn taking, two-player, zero sum games of perfect information
  - A zero-sum game is a mathematical representation of a situation in which a participant's gain (or loss) of utility is exactly balanced by the losses (or gains) of the utility of other participant(s)
- In our terminology – deterministic, fully observable environments with two agents whose actions alternate and the utility values at the end of the game are always equal and opposite (+1 and –1)
  - If a player wins a game of chess (+1), the other player necessarily loses (-1)
- Environments with very many agents are best viewed as economies rather than games

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello, rock−paper−scissors	backgammon monopoly
imperfect information	battleships, kriegspiel, stratego	bridge, poker, scrabble nuclear war

What Kind of Games?

- Abstraction: To describe a game we must capture every relevant aspect of the game. Such as :
  - Chess
  - Tic-tac-toe
  - …
- Accessible environments: Such games are characterized by perfect information
- Search: game-playing then consists of a search through possible game positions
- Unpredictable opponent: introduces uncertainty thus game-playing must deal with contingency problems

Games

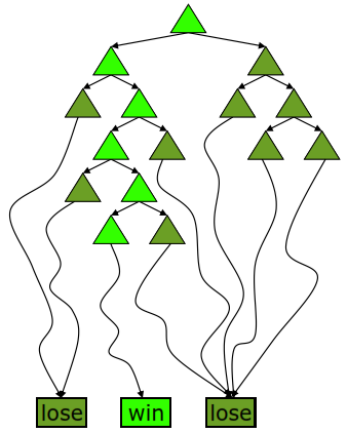
- Multi agent environments : any given agent will need to consider the actions of other agents and how they affect its own welfare.
- The unpredictability of these other agents can introduce many possible contingencies
- There could be competitive or cooperative environments
- Competitive environments, in which the agent's goals are in conflict require adversarial search – these problems are called as games

Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA →S
  - Terminal Test: S → {t,f}
  - Terminal Utilities: SxP → R
- Solution for a player is a policy: S → A

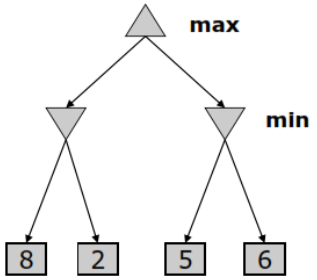
Deterministic Single-Player?

- Deterministic, single player, perfect information:
  - Know the rules
  - Know what actions do
  - Know when you win
  - E.g. Freecell, 8-Puzzle, Rubik's cube
- **… it's just search!**
- Slight reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children (the max value)
  - Note that we don't have path sums as before (utilities at end)
- After search, can pick move that leads to best node



Deterministic Two-Player

- E.g. tic-tac-toe, chess, checkers
- Zero-sum games
  - One player maximizes result
  - The other minimizes result
- Minimax search
  - A state-space search tree
  - Players alternate
  - Each layer, or ply, consists of a round of moves
  - Choose move to position with highest minimax value = best achievable utility against best play



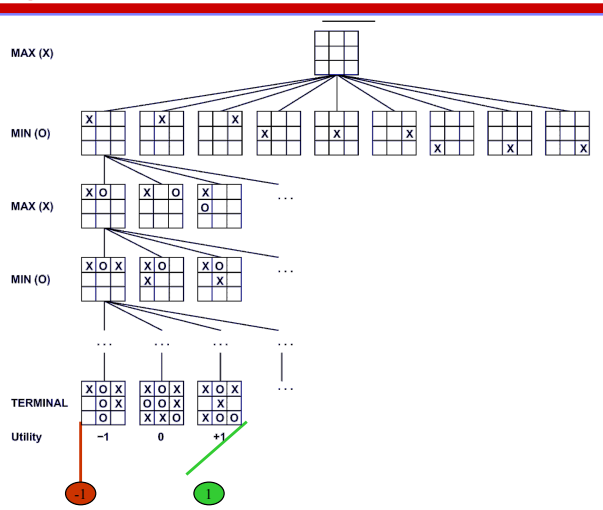
Searching for the Next Move

- **Complexity:** many games have a huge search space
  - **Chess:** *b = 35, m=100* ⇒ *nodes = 100* ^{35} if each node takes about 1 ns to explore then each move will take about ***10*** ^{50} **millenniums** to calculate.
- **Resource (e.g., time, memory) limit:** optimal solution not feasible/possible, thus must approximate
- **Pruning:** makes the search more efficient by discarding portions of the search tree that *obviously* cannot improve quality of result.
- **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search.

Two-player Games

- A game formulated as a search problem:
  - Initial state: board position and turn
  - Operators: definition of legal moves
  - Terminal state: conditions for when game is over
  - Utility function: a numeric value that describes the outcome of the game. e.g., -1, 0, 1 for loss, draw, win.

Example: Tic-Tac-Toe



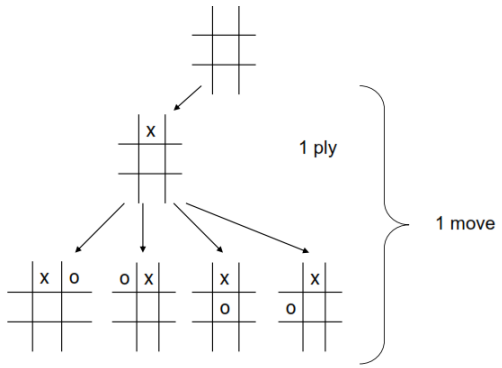
The minimax algorithm

- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the three by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use minimax decision to select the move with the max (of the min) utility value
- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

Mini-Max Algorithm

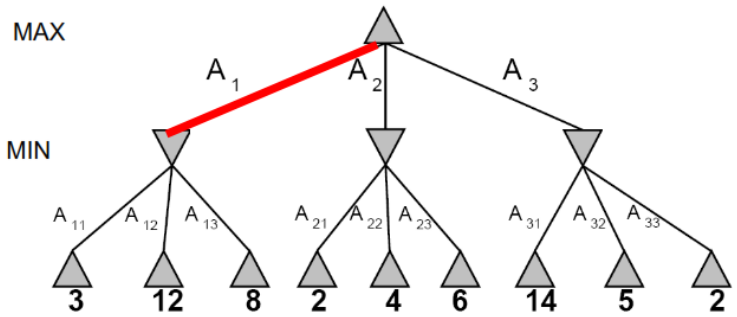
- Generate search tree of ALL possible games
- Score leaves (assume higher better for 'you')
  - E.g., +1 you win, -1 you lose, 0 if a draw (i.e., a tie)
- Assume opponent scores 'board configurations' the same way you do (more on this later)
- Propagate info from leaves to root of search tree, then choose best move
  - Your turn: choose MAX score
  - Opponent's turn: choose MIN score (which is best for opponent)
- Only choose NEXT move; when your turn again, repeat
  - Opponent might not have done what you expected

Generate Game Tree



The Minimax Algorithm

- e.g., 2-ply game:



Minimax value

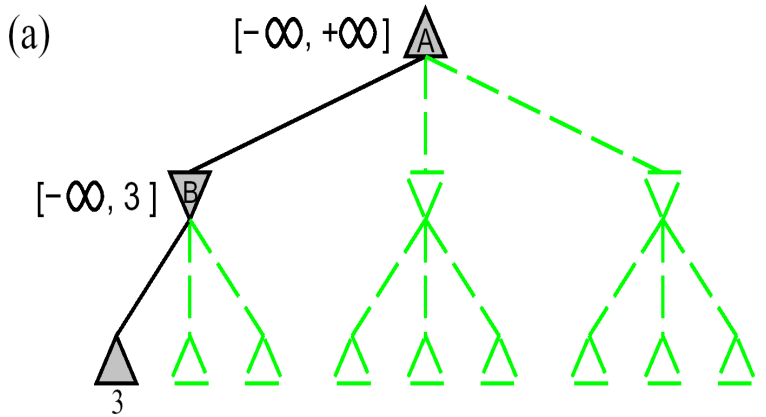
- Given a game tree, the optimal strategy can be determined by examining the minimax value of each node (MINIMAX-VALUE(n))
- The minimax value of a node is the utility of being in the corresponding state, assuming that both players play optimally from there to the end of the game
- Given a choice, MAX prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value

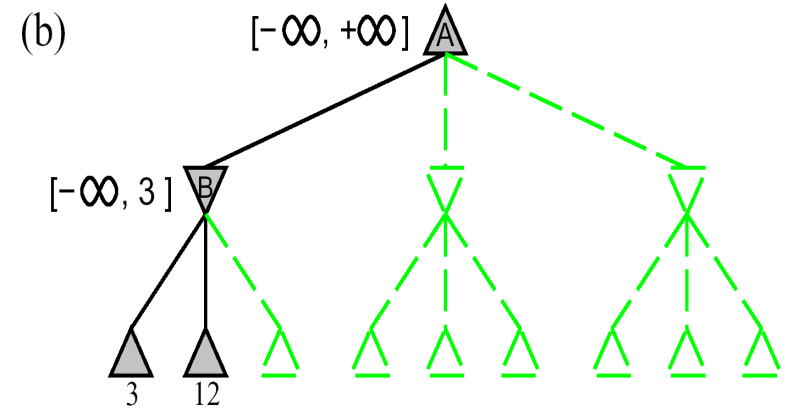
Minimax Algorithm

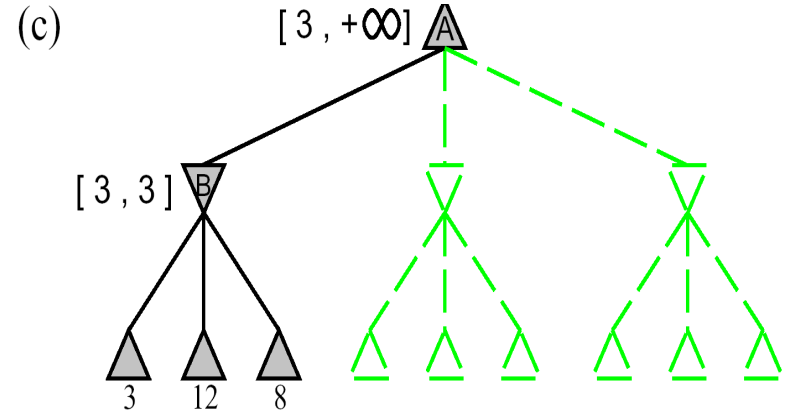
```
function MINIMAX-DECISION(state) returns an action
  inputs: state, current state in game
  return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
  return v
```

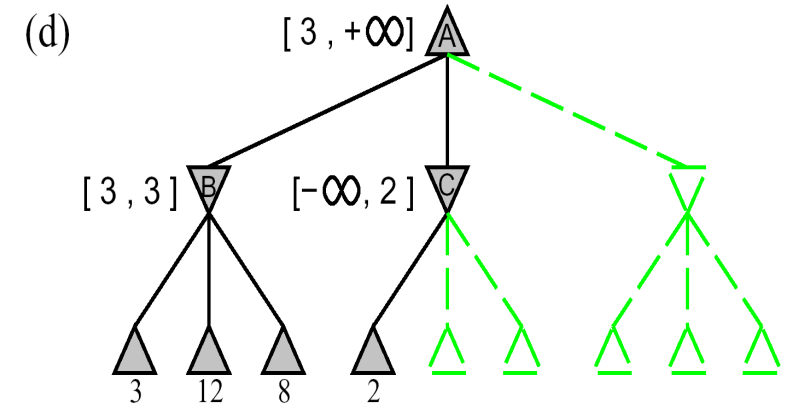

Properties of Minimax

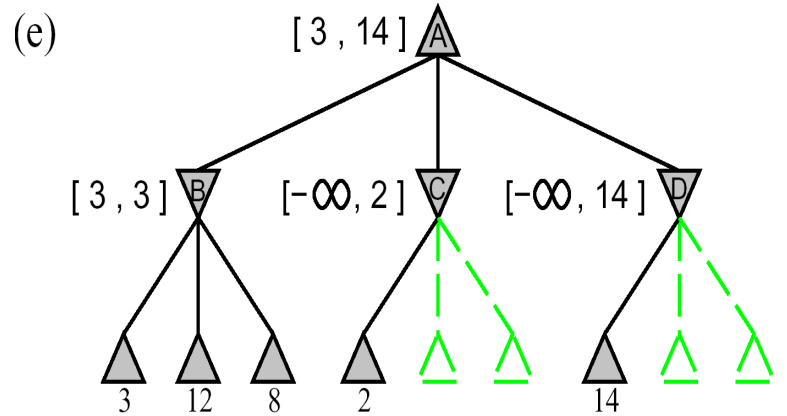
- Completeness: if tree is finite (chess has specfiic rules for this)
- Optimality: Yes, against an optimal opponent.
- Time complexity: $O(b^m)$
- Space complexity: O(bm) (depth-first exploration)
- For chess, b ≈35, m ≈ 100 for “reasonable" games
  ∴exact solution completely infeasible
- But do we need to explore every path?

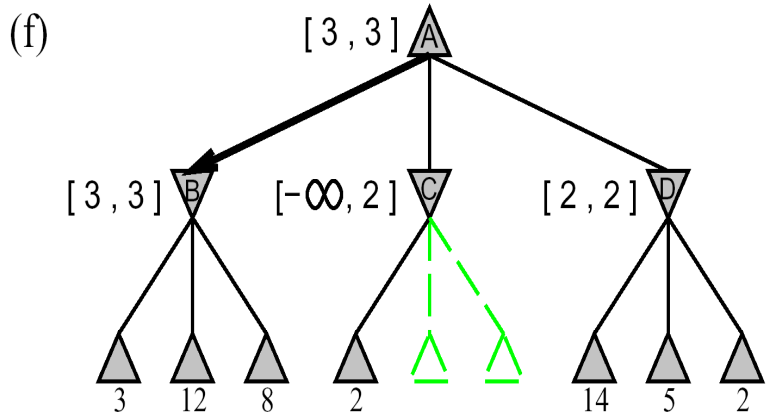



(b)
[−∞, +∞]
A
[−∞, 3]
B
3
12


(c)
[3 , +∞]
A
[3 , 3]
B
3
12
8


(d)
[3 , +∞]
A
[3 , 3]
B
[−∞, 2]
C
3
12
8
2


(e)
[3 , 14]
A
[3 , 3]
B
[−∞, 2]
C
[−∞, 14]
D
3
12
8
2
14



α-β Pruning: Search Cutoff

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node
- **α-β pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.
- Does it work? Yes, it roughly cuts the branching factor from b to √b resulting in as double as far look-ahead than pure minimax

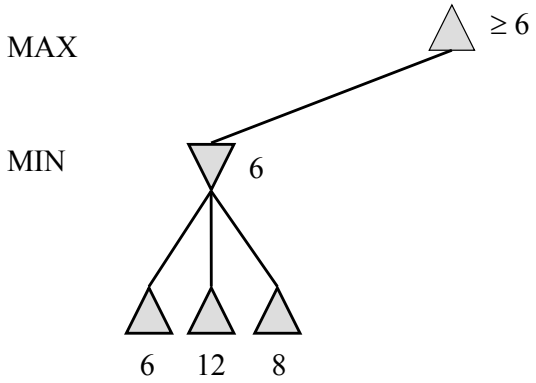
Alpha-beta Search

function ALPHA-BETA-DECISION(*state*) **returns** an action
  return the *a* in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(*a*, *state*))

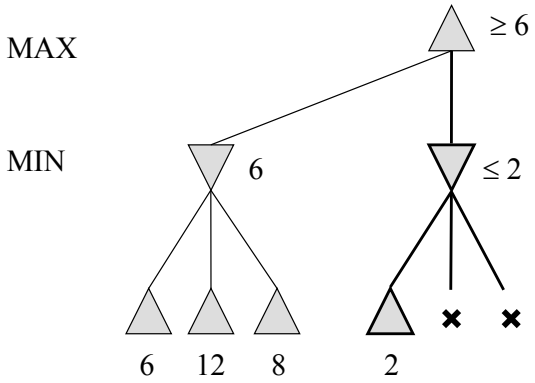
function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for *a*, *s* in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  same as MAX-VALUE but with roles of α, β reversed

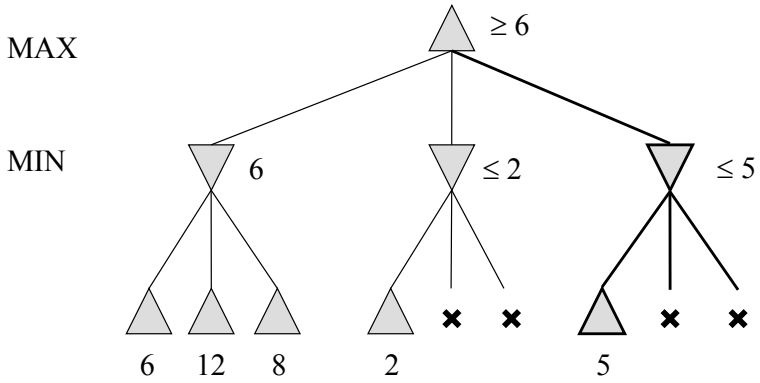
α-β Pruning: Example



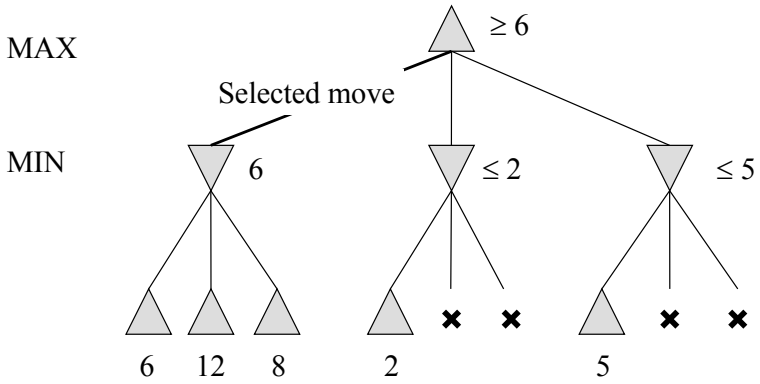
α-β Pruning: Example



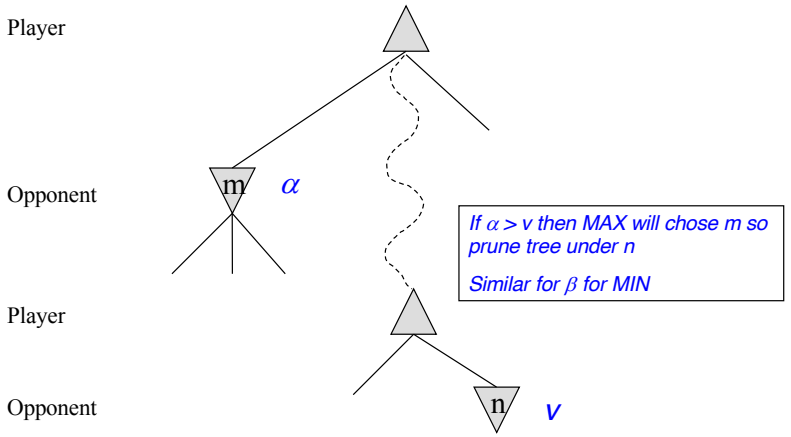
α-β Pruning: Example



α-β Pruning: Example



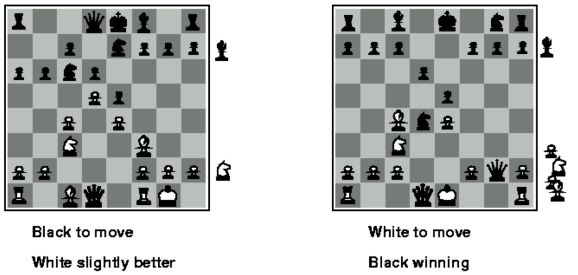
α-β Pruning: General Principle



Move Evaluation Without Complete Search

- Complete search is too complex and impractical
- Evaluation function: evaluates value of state using heuristics and cuts off search
- New MINIMAX:
  - CUTOFF-TEST: cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
  - EVAL: evaluation function to replace utility function (e.g., number of chess pieces taken)

Evaluation Functions



Black to move
White slightly better

White to move
Black winning

- **Weighted linear evaluation function:** to combine *n* heuristics
  $f = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$
- e.g., *w's* could be the values of pieces (1 for pawn, 3 for bishop etc.) *f's* could be the number of type of pieces on the board

Evaluation Functions

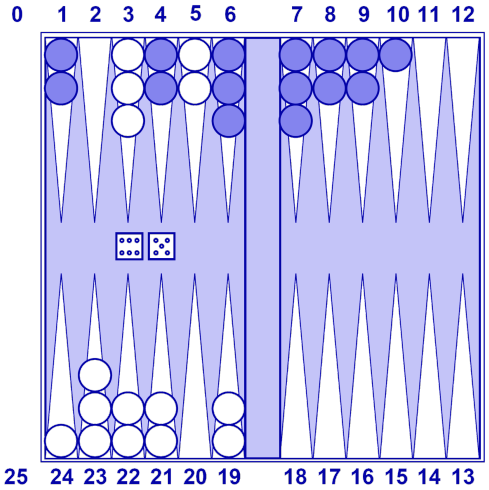
- Material value for each piece in chess
  - Pawn: 1
  - Knight: 3
  - Bishop: 3
  - Rook: 5
  - Queen: 9

  This can be used as weights and the number of each kind can be used as features
- Other features
  - Good pawn structure
  - King safety
- These features and weights are not part of the rules of chess, they come from playing experience

Cutting off search

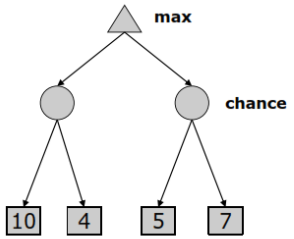
- MinimaxCutoff is identical to MinimaxValue except
  - Terminal? is replaced by Cutoff?
  - Utility is replaced by Eval
- Does it work in practice?
- bm = 106, b=35 → m=4
- 4-ply lookahead is a hopeless chess player!
  - 4-ply ≈ human novice
  - 8-ply ≈ typical PC, human master
  - 12-ply ≈ Deep Blue, Kasparov

Nondeterministic Games



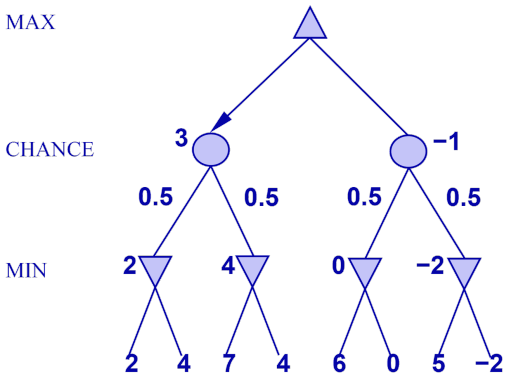
Expectimax Search Trees

- What if we don't know what the result of an action will be? E.g.,
  - In solitaire, next card is unknown
  - In minesweeper, mine locations
  - In pacman, the ghosts act randomly
  - Games that include chance
- Can do **expectimax search**
  - Chance nodes, like min nodes, except the outcome is uncertain
  - Calculate expected utilities
  - Max nodes as in minimax search
  - Chance nodes take average (expectation) of value of children

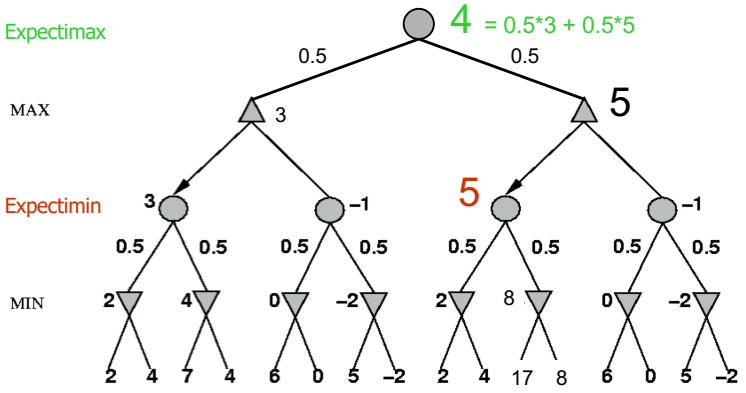


Nondeterministic Games: The Element of Chance

In nondeterministic games, chance introduced by dice, card-shuffling
Simplified example with coin-flipping

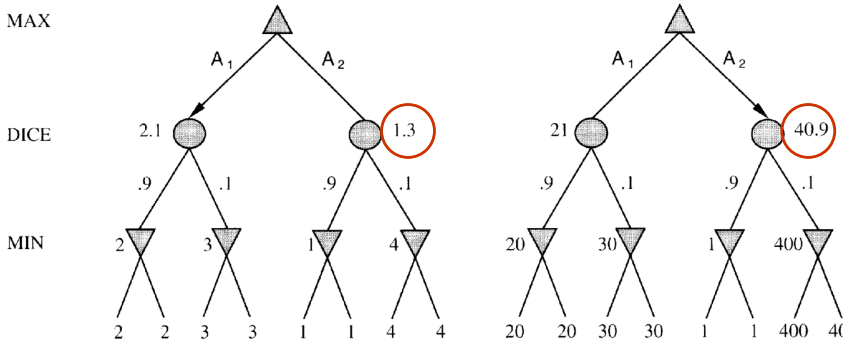


Nondeterministic Games: The Element Of Chance



Evaluation functions

Order-preserving transformation do not necessarily behave the same



State-of-the-art For Nondeterministic Games

- Dice rolls increase b: 21 possible rolls with 2 dice
- Backgammon ≈ 20 legal moves (can be 6,000 with 1-1 roll)
- depth 4 = 20 x(21x20)^3 ≈1.2x10^9
- As depth increases, probability of reaching a given node shrinks
  - value of look-ahead is diminished
- α–β pruning is much less effective
- TDGammon uses depth-2 search + very good Eval
  - world-champion level

Games of Imperfect Information

- e.g., card games, where opponent's initial cards are unknown
- Typically we can calculate a probability for each possible deal
- Seems just like having one big dice roll at the beginning of the game
- Idea: compute the minimax value of each action in each deal, then choose the action with highest expected value over all deals
- Special case: if an action is optimal for all deals, it's optimal.
- GIB, current best bridge program, approximates this idea by
  - 1) generating 100 deals consistent with bidding information
  - 2) picking the action that wins most tricks on average

State-of-the-art For Deterministic Games

- Chess: Deep Blue defeated human world champion Gary Kasparov in a six- game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Now reinforcement learning based programs learn to play from scratch and beat the top chess program.
- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. 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. Schaeffer kept on building on his database of endgames, and in 2007 “solved” checkers
- Othello: Human champions refuse to compete against computers, who are too good.
- Go: Human champions used to refuse to compete against computers, which were too bad Alpha Go changed everything using a reinforcement learning approach..

Top Chess Engines (2018)

- **1. Stockfish 9 – Elo 3438:** Stockfish is the strongest free chess engine. Stockfish 9 is well beyond the skill of any grandmaster.
- **2. Komodo 11.3.1 – Elo 3404:** The chess engine is a commercial one, but older versions (8 and older) are free for non-commercial use.
- **3. Houdini 6 – Elo 3400:** Earlier versions are free for non-commercial use (up to version 1.5a). Later versions – 2.0 and onwards – are commercial.
- **4. Fire 7.1 – Elo 3325:** It was originally started out as an open-source project, but later on, the code became closed. However, there is a fork off of fire 2.2, which has been renamed Firenzina, which is still open-sourced.
- **5. Deep Shredder 13 – Elo 3286:** Shredder is definitely to be considered one of the best chess engines of the world since it has already won seventeen titles as World Computer Chess Champion and is accepted as one of the best chess engines of the world.
- **6. Allie+Stein,** the new neural network based engine