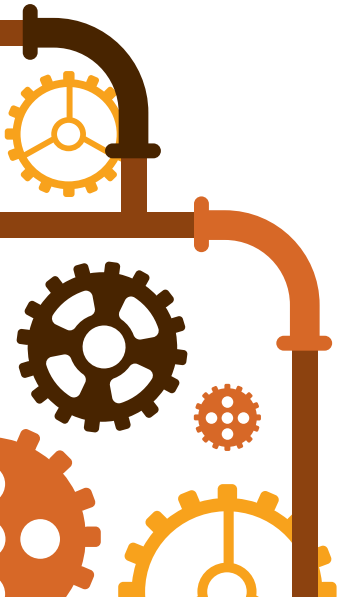
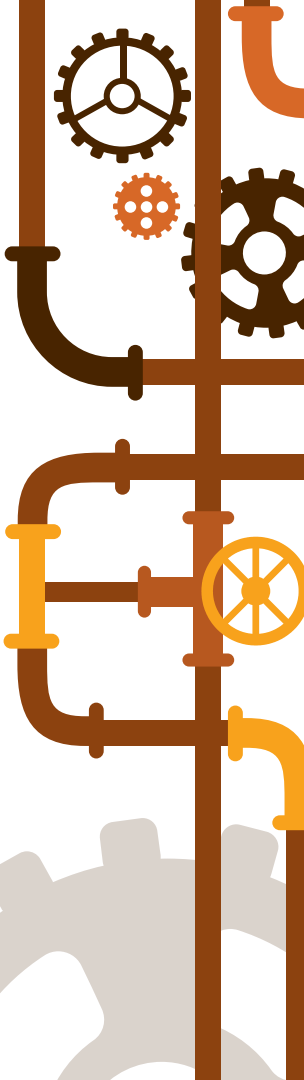



CS 482/682 AI - Search in Games

Fall 2021

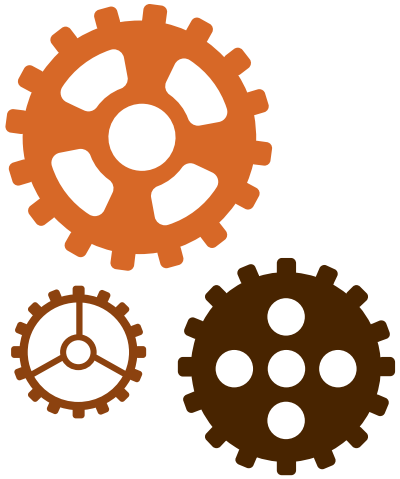
A* Search

Combine Greedy search with Uniform Cost Search

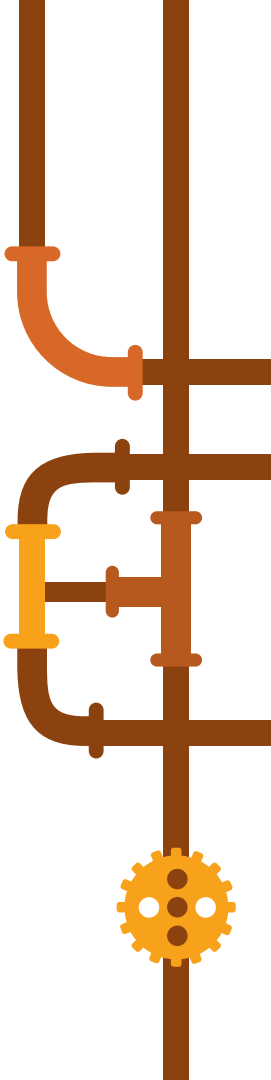
Minimize the total path cost:

(f) = actual path so far (g) + estimate of future path to goal (h)

City	Distance to PHX
Boston	2299
Chicago	1447
Nashville	1444
Key West	1927
Austin	870
San Francisco	658



What if you can't control the path taken through the search tree?



Chapter 5

- We cover competitive environments
- Two of more agents have conflicting goals → adversarial search problems
- Rather than dealing with real-world skirmishes, we concentrate on chess and poker, etc.
- The simplified nature of these games make it easier to design an algorithm

Types of Games

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

Games vs. Search Problems

- “Unpredictable” opponent → solution is a contingency plan
- Time limits → unlikely to find goal, just approximate
- Plan of attach:
    - Algorithm for perfect plan (Von Neumann, 1944)
    - Finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952-57)
    - Pruning to reduce costs (McCarthy, 1956)

Two-player Zero-sum Games

- Two-player, turn-taking, perfect information, zero-sum games
- Perfect information: fully observable
- Zero-sum: what is good for one player is just as bad for the other
- We will call two players MIN and MAX
- MAX moves first and then the players take turns moving until the game is over

Some Notations

S_0: The **initial state**, which specifies how the game is set up at the start.

To-Move(s): The player whose turn it is to move in state s.

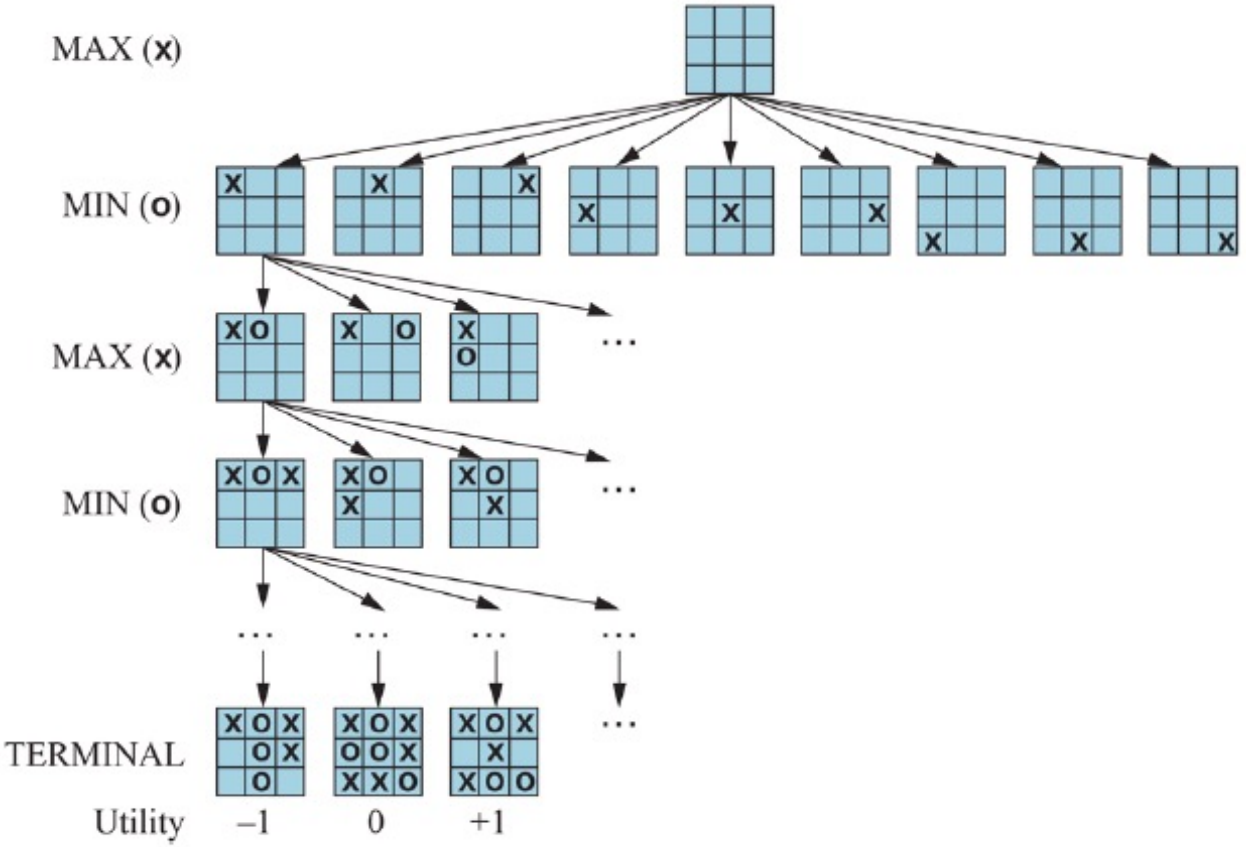
Actions(s): The set of legal moves in state s.

Result(s, a): The **transition model**, which defines the state resulting from taking action a in state s.

Is-Terminal(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.

Utility(s, p): A **utility function** (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s

Game Tree



Game Tree

- For tic-tac-toe, the game tree is relatively small
  - fewer than 9! = 362,880 terminal nodes
  - Less than 5,478 states
- For chess there are over 10^{40} nodes
  - The game tree is best thought of as a theoretical construct that we cannot realize in the physical world

Optimal Decisions in Games

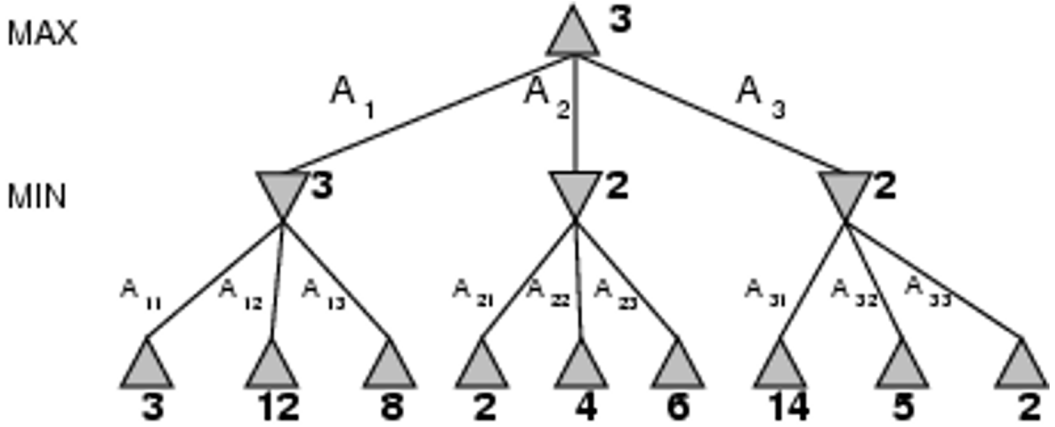
- MAX wants to find a sequence of actions leading to a win, but
- MIN also seeks to prevent MAX from winning at every move
- The strategy must be a conditional plan
    - A contingent strategy specifying a response to each of MIN's possible moves
    - The desirable outcome must be guaranteed no matter what MIN does

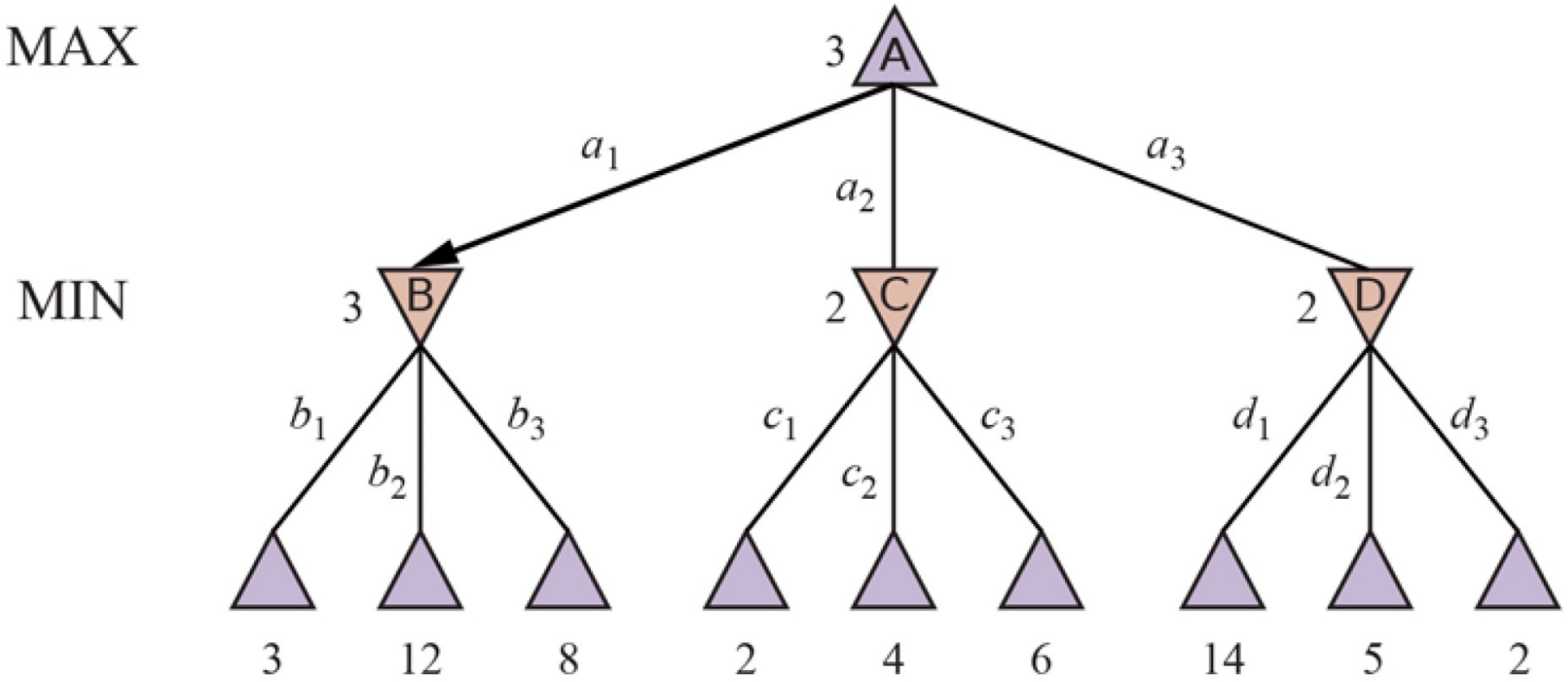
The Minimax Algorithm

1. Generate the entire game tree
2. Apply the utility function to each terminal node (high values are good for your side)
3. Filter values from the terminal nodes up through the tree:
    a. At nodes controlled by your opponent, choose the minimum value of the children
    b. At nodes controlled by you, choose the maximum value of the children
4. When you reach the top of the tree, you have an optimal solution

Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value = best achievable payoff against best play
- E.g., 2-ply game:





A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

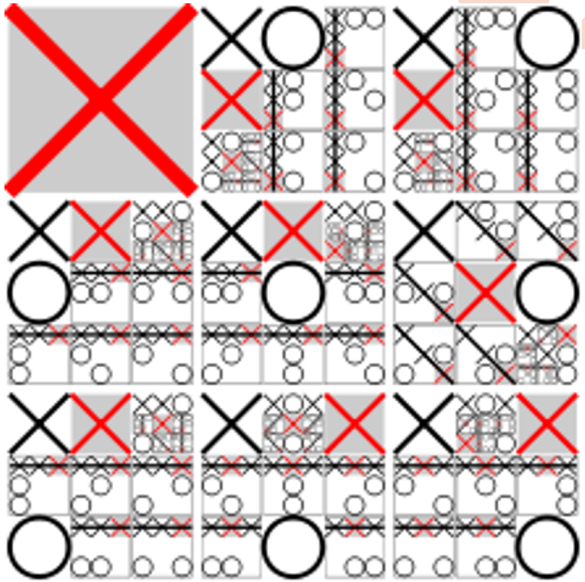
Minimax algorithm

```
function MINIMAX-DECISION(game) returns an operator
    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]
```

```
function MINIMAX-VALUE(state, game) returns a utility value
    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Properties of minimax

- Complete? Yes (if tree is finite)
- Optimal? 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



Is it practical to construct a complete search tree?

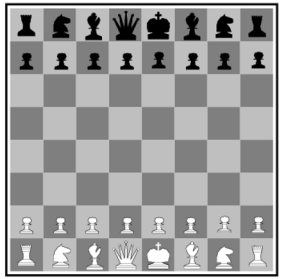
Typical chess program using minimax:

- Evaluate 1000 positions per second
- Tournament chess is 150 seconds per move
- Total of 150,000 positions
- Branching factor for chess is ~35
- Evaluate only 3-4 ply
- Average human player can make plans 6-8 ply ahead

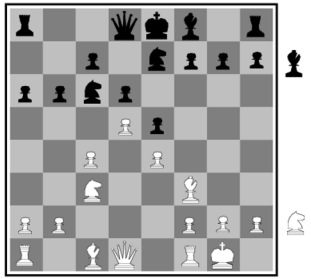
Imperfect Decisions

- What if you don't have time/space to build the entire search tree?
  - Use a heuristic and limit the depth!
    - In game playing, the heuristic function is often called an evaluation function
    - *Cutoff test* and *evaluation function*
  - As always, the quality of the heuristic function can make an enormous impact

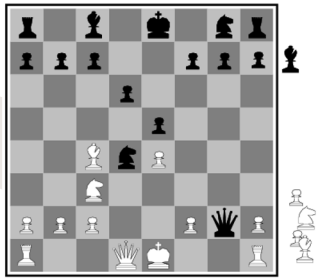
What do these Evaluation Functions Look Like?



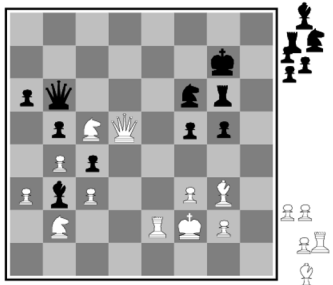
(a) White to move
Fairly even



(b) Black to move
White slightly better



(c) White to move
Black winning

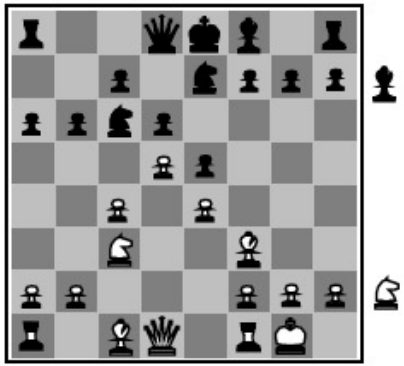


(d) Black to move
White about to lose

f(state)⟶real

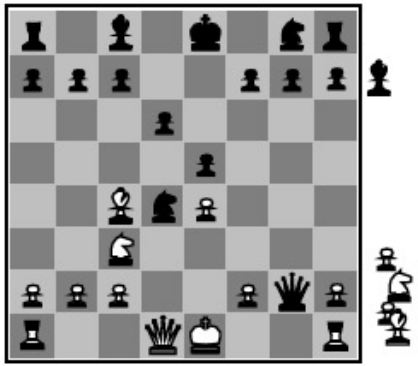
- These heuristics are critical for complex games, like chess
- Account for:
  - Piece count
  - Whose turn it is
  - Board positioning
- The relative ordering of values matters, not the values themselves

Evaluation Functions



Black to move

White slightly better



White to move

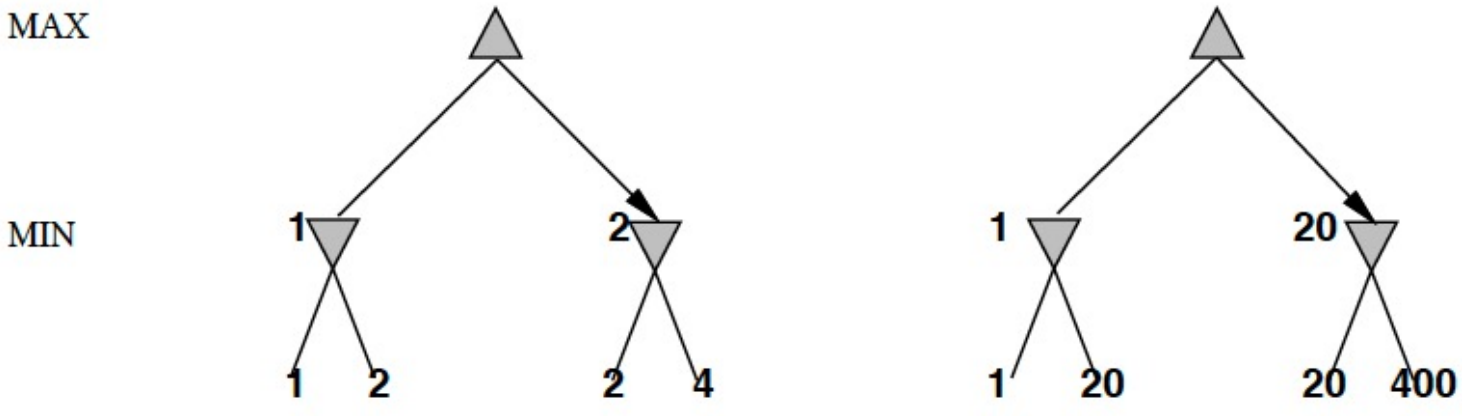
Black winning

For chess, typically *linear* weighted sum of features

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens)

Exact Values don't Matter



Behaviour is preserved under any *monotonic* transformation of EVAL

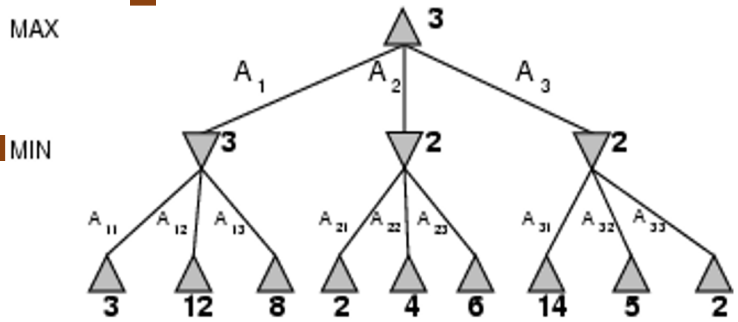
Only the order matters:

payoff in deterministic games acts as an *ordinal utility* function

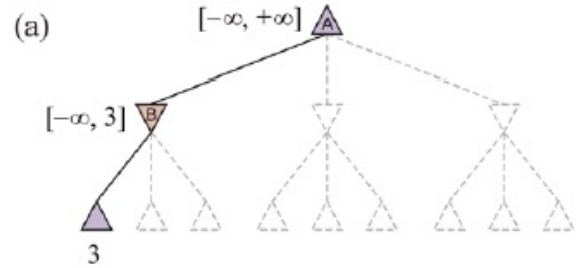
How to Improve Opponent Search: Pruning

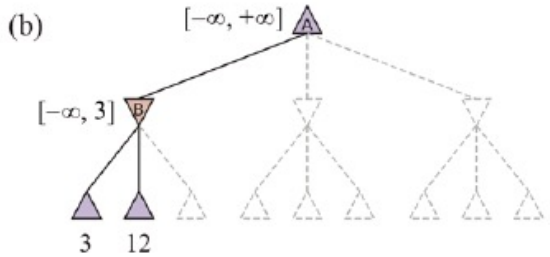
- Don’t evaluate all the parts of the tree
- Pruning techniques eliminate parts of the search tree without looking at them
- Today, we will look at one simple but effective form of pruning: alpha-beta

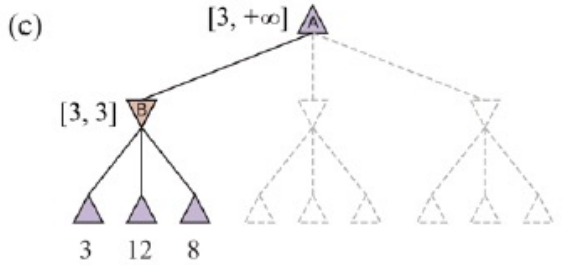
The alpha-beta principle
If you have an idea that is surely bad,
don’t take time to see how truly awful it is

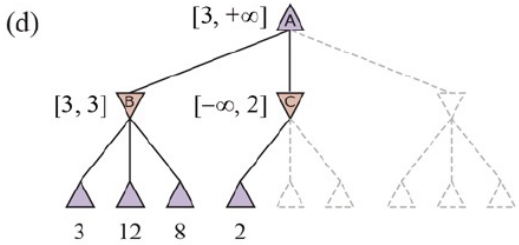


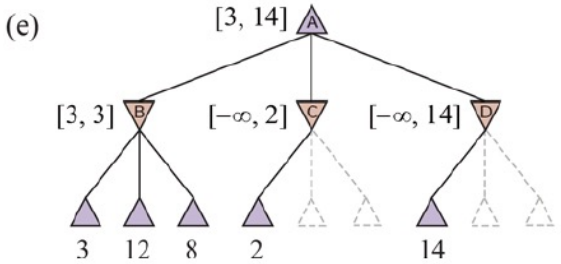
Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second

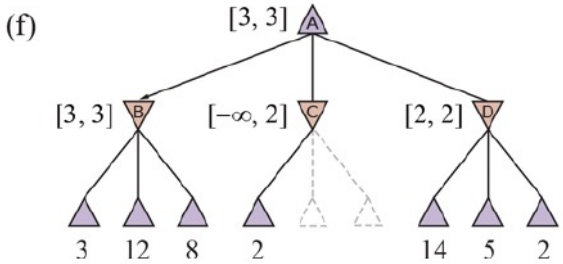


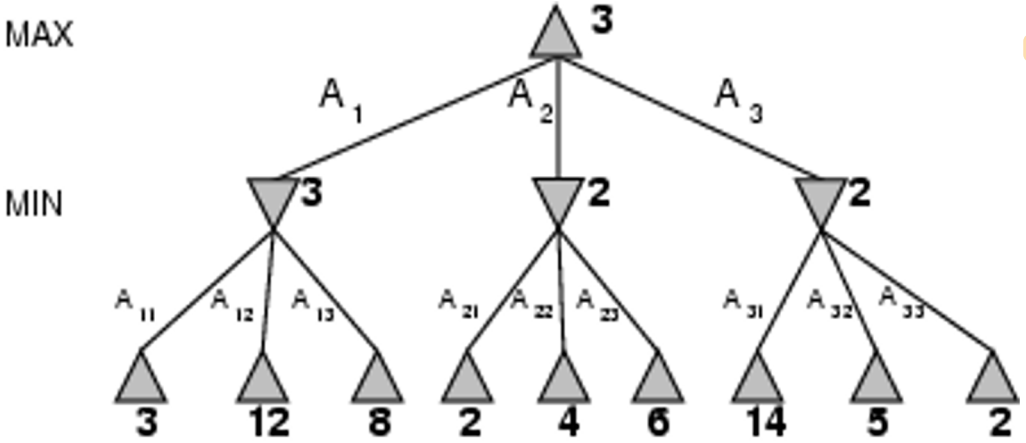




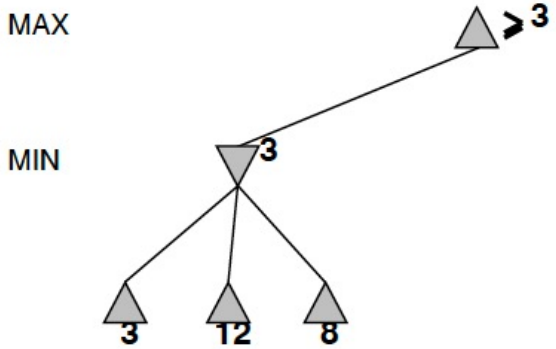


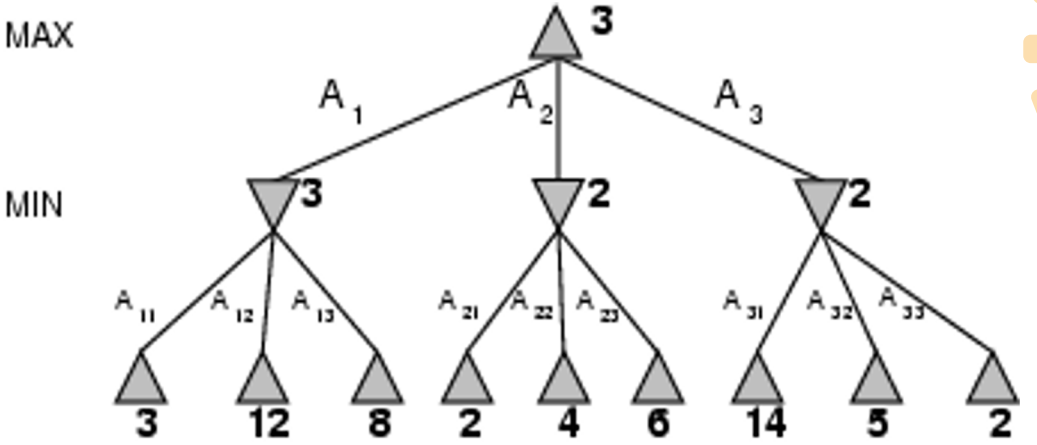




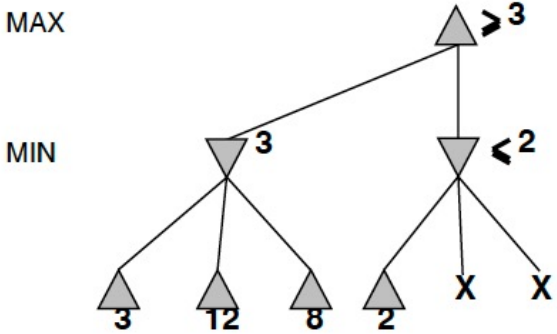


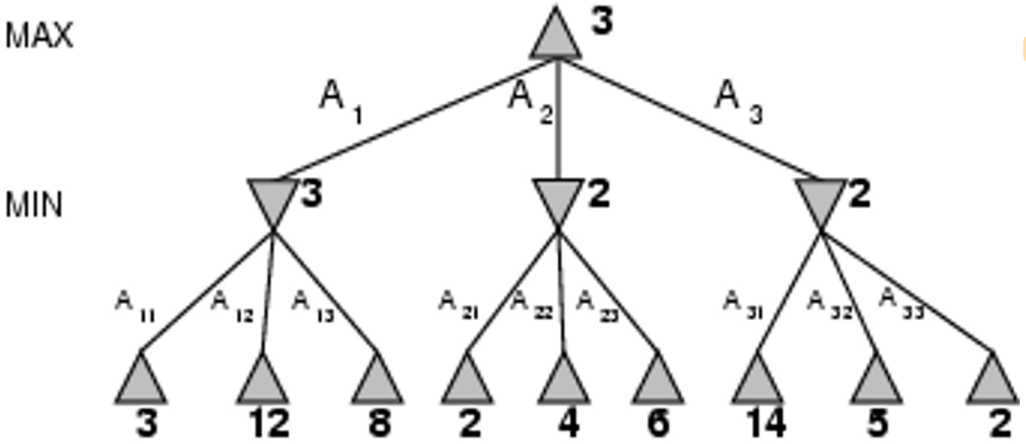
α–β **pruning example**



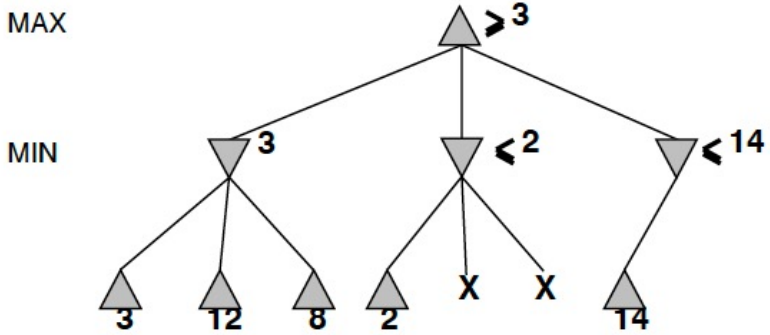


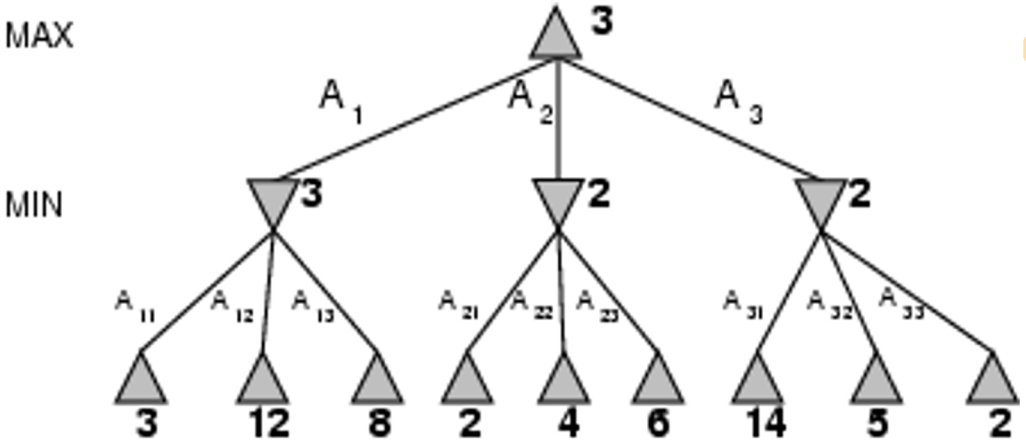
α–β pruning example



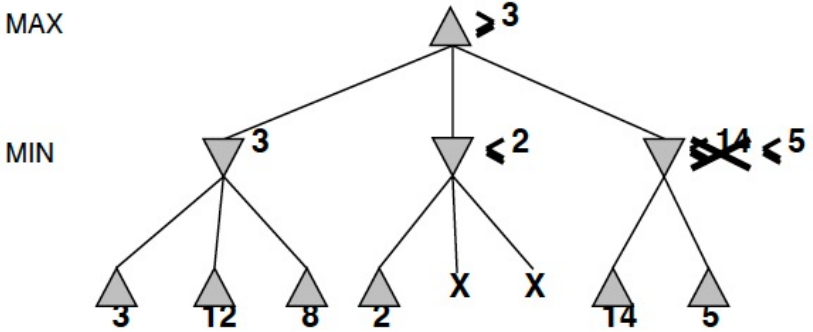


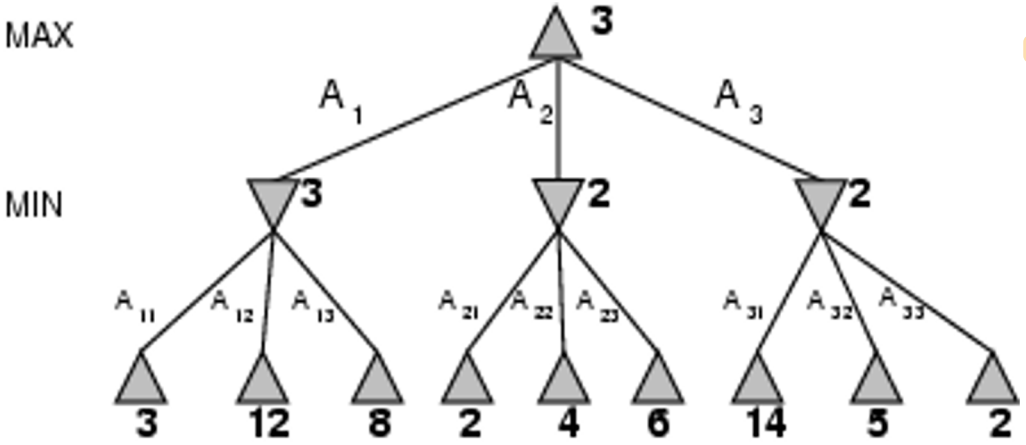
α–β pruning example



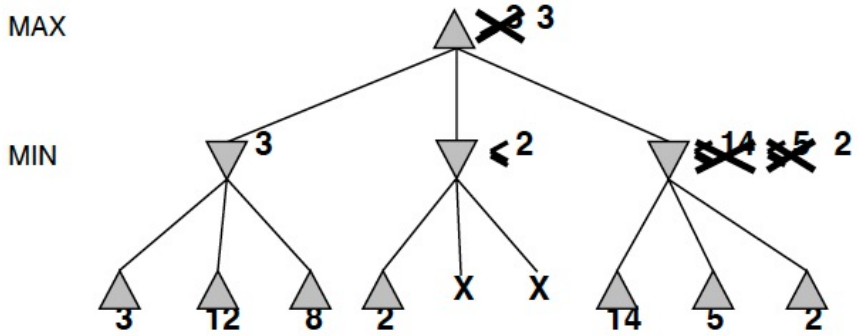


α–β **pruning example**





α–β pruning example

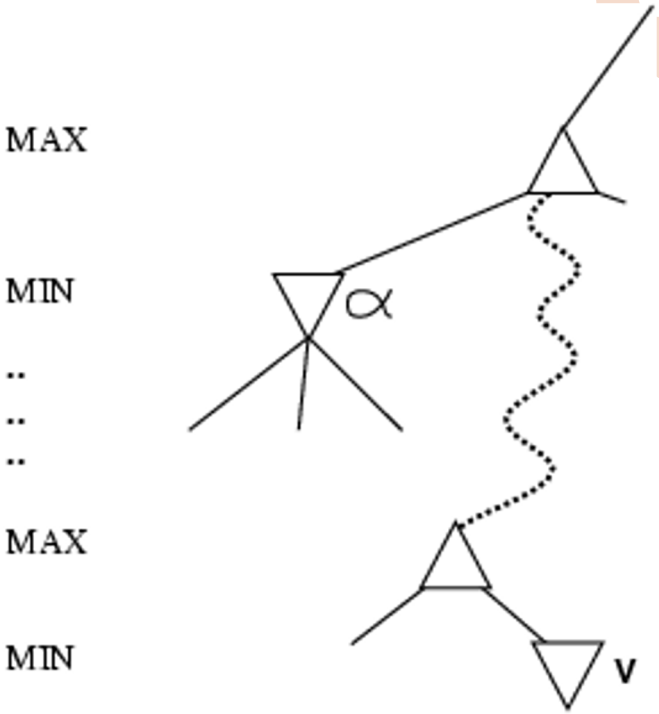


Properties of α-β

- Pruning does not affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity = $O(b^{d/2})$
- $\rightarrow$ doubles depth of search
- A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)

Why is it called α-β?

- α is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for max
- If v is worse than α, max will avoid it
    - prune that branch
- Define β similarly for min



The α–β algorithm

Basically MINIMAX + keep track of α, β + prune

function MAX-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state*
    β, the best score for MIN along the path to *state*

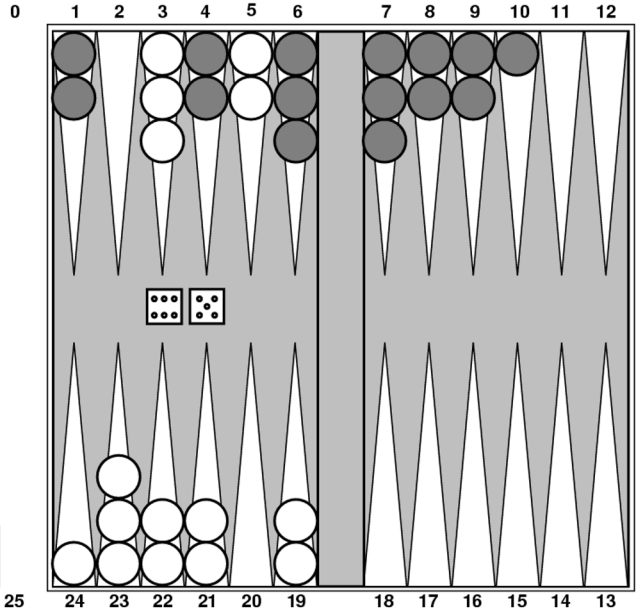
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\alpha \leftarrow$ MAX(α, MIN-VALUE(*s*, *game*, α, β))
    if $\alpha \geq \beta$ **then return** β
  end
  return α

---

function MIN-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*

  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(*s*, *game*, α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

Games that Include Chance



How can we model games that involve some random chance?

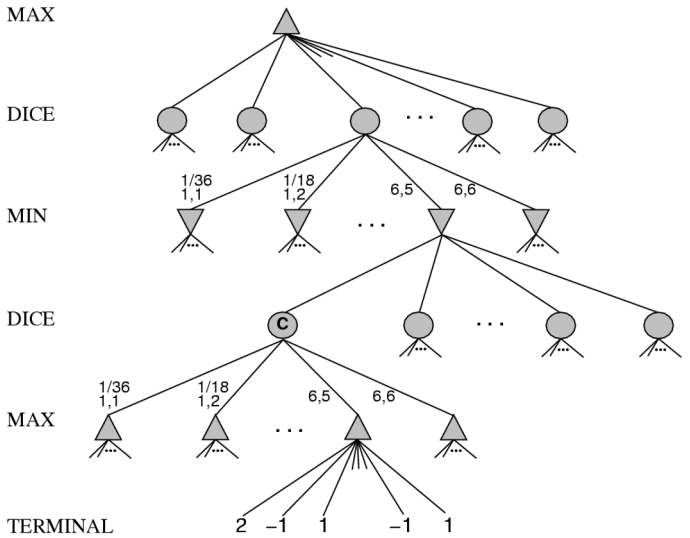
For example, dice rolls in backgammon determine the available moves

Solution: treat the randomization as another player

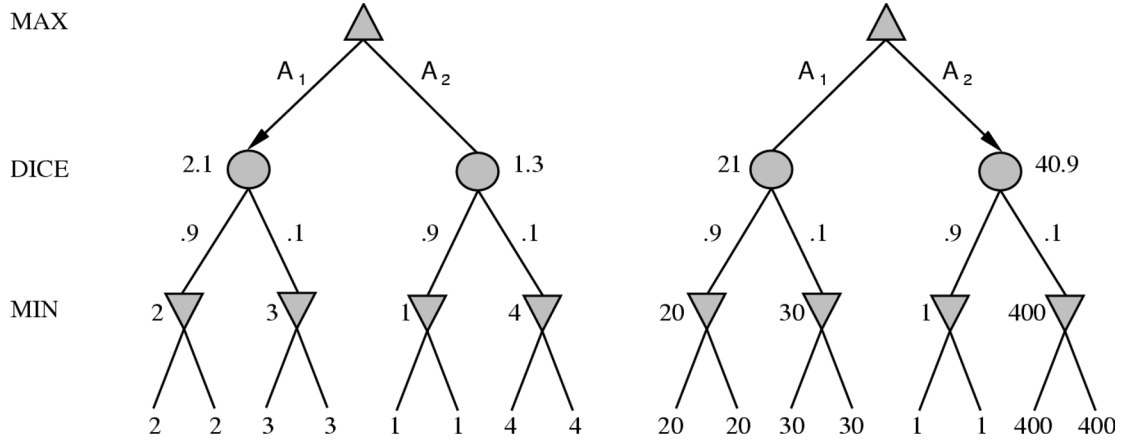
Expectiminimax

When you encounter nodes that are determined by chance, compute the expected value based on the probability distribution

Must be careful about the evaluation functions... now these values have exact meaning rather than just ordering properties

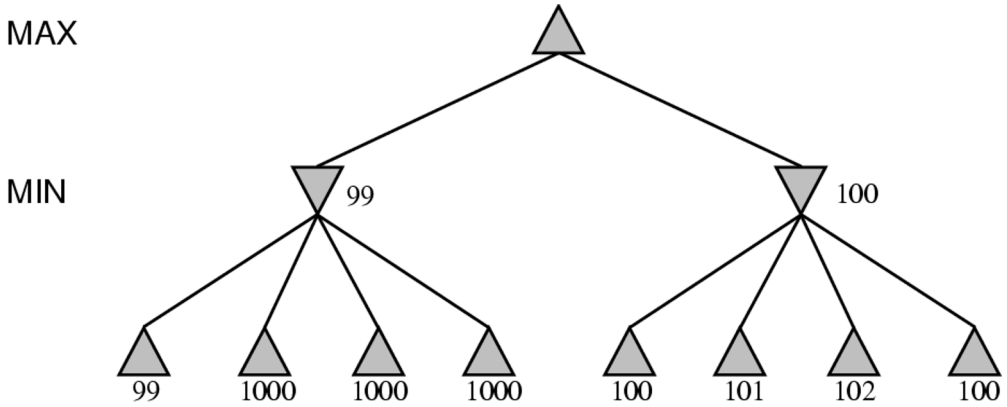


Evaluation Functions and Expectiminimax

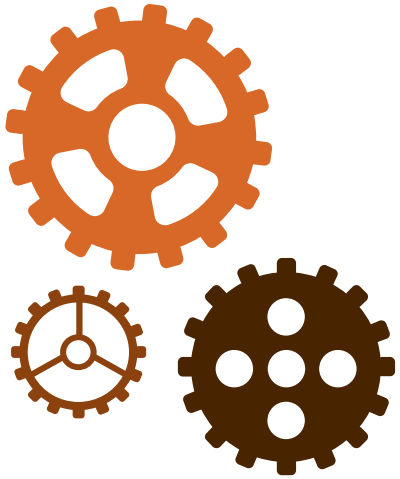


By changing the evaluation function values, we change the outcome.
This would not occur under normal minimax

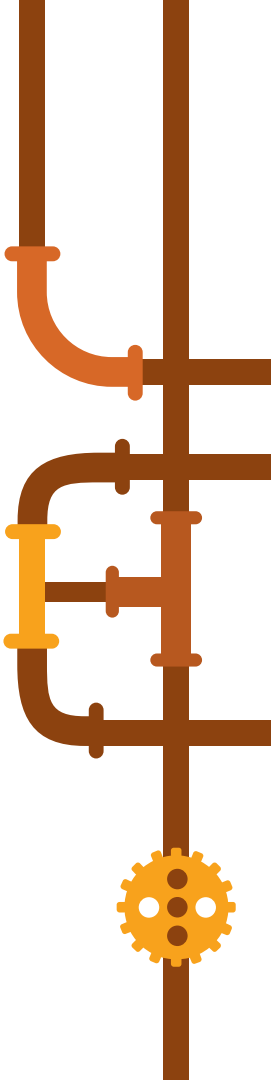
Is Minimax Always a Good Idea?



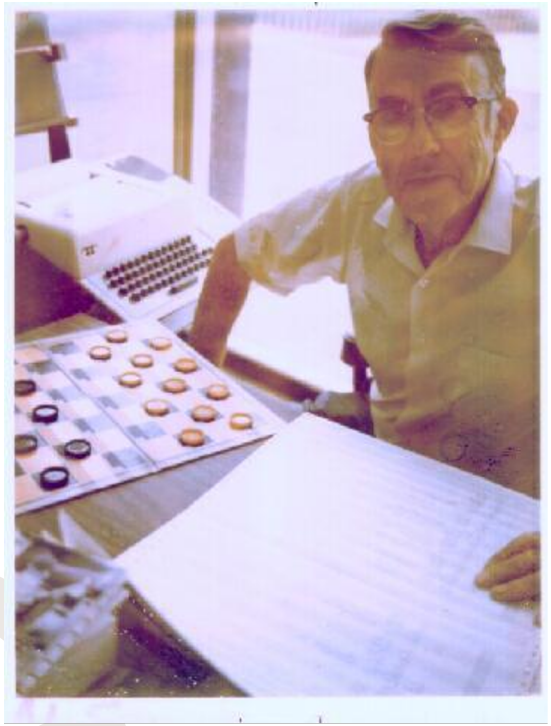
Minimax makes the assumption that your opponent acts exactly as you would (and can look no further ahead)
In cases like the tree above, this may be a poor assumption



How has this worked in the real world?



Samuel's Checkers Player



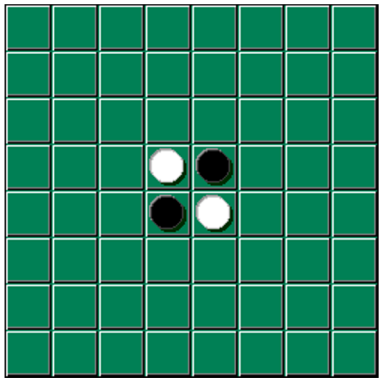
Written in 1952
Minimax search with alpha-beta pruning
Evaluation function was learned by playing games against itself
Played competitively after a few days of training
Hardware:

- 10,000 words of memory
- Magnetic tape storage
- .000001 GHz processor

Chinook

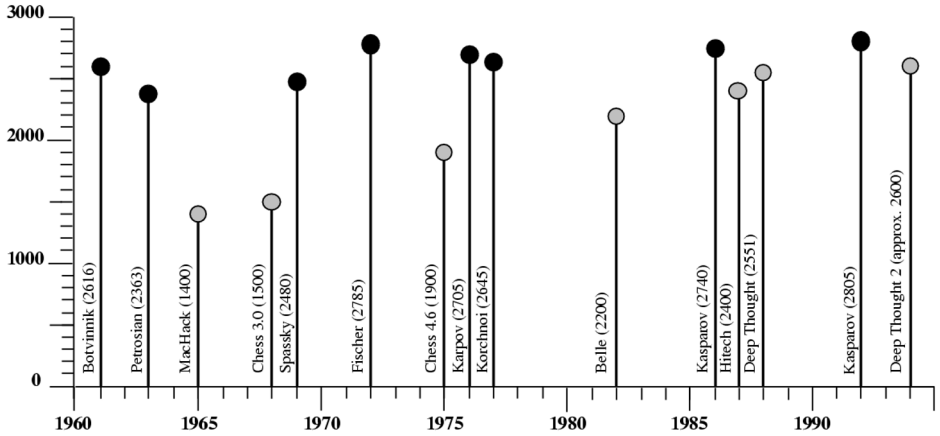
In 1994, Chinook defeated Dr. Marion Tinsley, the world checkers champion, who withdrew from the match for health reasons
Tinsley had held his title for 40 years, and only lost 3 matches.
First machine to claim a human world championship title
Incorporated end-game databases for all board positions containing 8 or fewer pieces

Othello



Smaller search space than chess (usually 5-15 legal moves)
Evaluation functions are difficult to craft
Most programs are better than human players
In 1997, the Logistello program defeated the human world champion six games to none.

Historical Look at Chess-Playing Algorithms



10^{120} possible board positions
Branching factor of ~35
Computer chess players were increasing at roughly the rate of processor speed

Deep Blue

May, 1997 defeated Garry Kasparov, the world chess champion

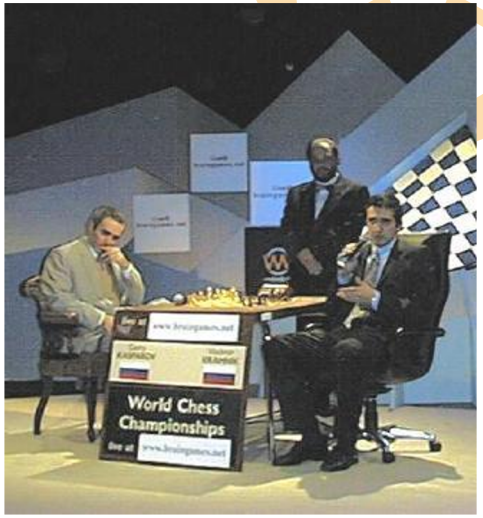
Special Hardware:

- 32-node IBM RS/6000 SP high-performance computer
- Each node contains 8 dedicated VLSI chess processors

~200 billion evaluations within three minutes, which is the time allotted to each player's move in classical chess (Kasparov can evaluate ~3 boards per second)

Finely crafted evaluation function

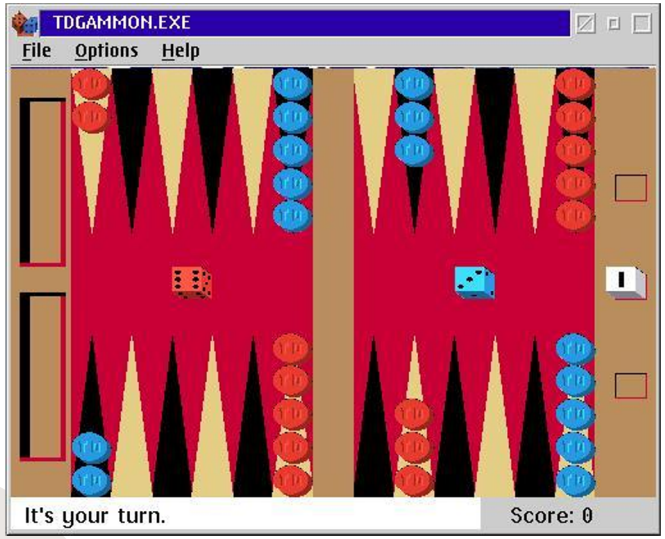
Not “AI” according to its creators



Deep Fritz

- Challenged Vladimir Kramnik (reigning world champion) in 2002.
  - (Kramnik took Kasparov's title from him in 2000)
- Eight game match ended in a draw
- Important piece:
  - FRITZ was running on an ordinary PC, not a supercomputer.

TD-Gammon (Gerry Tesauro)



In 1998, played 100 games against world champion Malcolm Davis

Davis won, but by a narrow margin, and mostly due to one large blunder

Neural net evaluation function

- 300 input values
- 160 hidden units
- ~50,000 weights

1,500,000 training matches

Go

Very difficult to form an evaluation function

Some successful algorithms:

- Go4++
- Many Faces of Go

Table 1: A Comparison the Features of Chess and Go

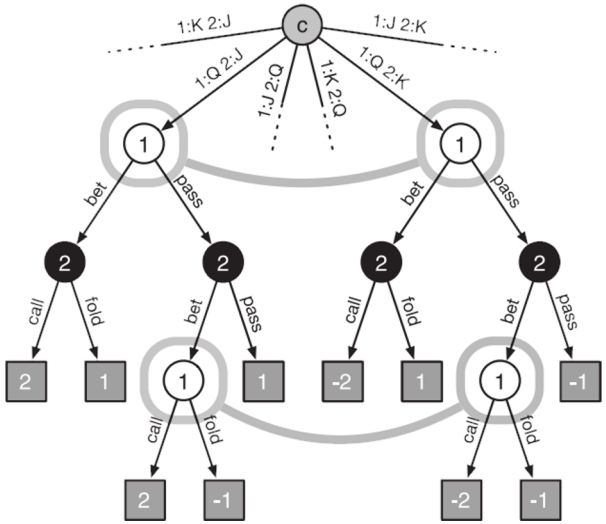
	Features	Chess	Go
1	board size	8 x 8 squares	19 x 19 grid
2	# moves per game	~80	~300
3	branching factor	small (~35)	large (~200)
4	end of game and scoring	checkmate (simple definition - quick to identify)	counting territory (consensus by players - hard to identify)
5	long range effects	pieces can move long distances (e.g., queens, rooks, bishops)	stones do not move, patterns of stones have long range effects (e.g., ladders; life & death)
6	state of board	changes rapidly as pieces move	mostly changes incrementally (except for captures)
7	evaluation of board positions	good correlation with number and quality of pieces on board	poor correlation with number of stones on board or territory surrounded
8	programming approaches used	amenable to tree searches with good evaluation criteria	too many branches for brute force search, pruning is difficult due to lack of good evaluation measures
9	human lookahead	typically up to 10 moves	even beginners read up to 60 moves (deep but narrow searches e.g., ladders)
10	horizon effect	grandmaster level	beginner level (e.g., ladders)
11	human grouping processes	hierarchical grouping (Chase & Simon, 1973)	stones belong to many groups simultaneously (Reitman, 1976)
12	handicap system	none	good handicap system

AI Beats Ken Jennings (2011)

AlphaGo (Google) beats Ke Jie (2017)

AI and Poker

Polaris
University of Alberta
Texas Hold ‘em Annual matches against poker pros (such as Phil Laak and Ali Eslami)
500-hand duplicate matches



Coming Up Next...

- Logical Reasoning
- First-order logic
- Knowledge representation

Read Chapter 7