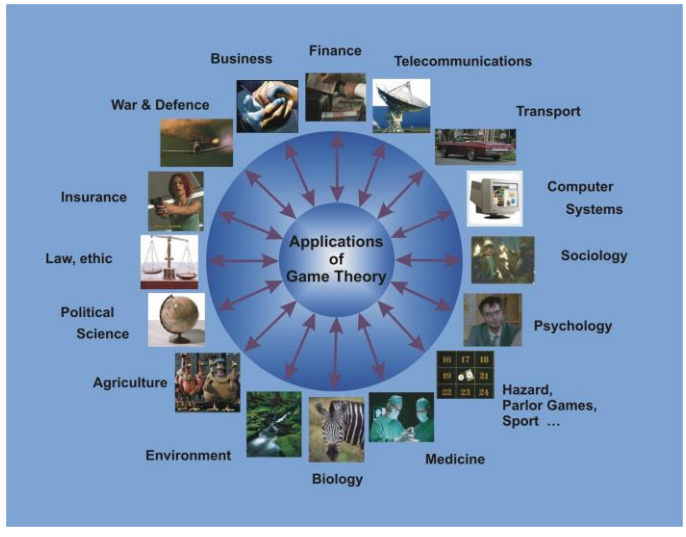
Games: Adversarial Search minimax and expectimax

- Game Theory
- Optimal Decisions in Games
  - minimax decisions
  - α–β pruning
  - Expectimax
- Resource limits and approximate evaluation
- Games of imperfect information

Game Theory in Artificial Intelligence by Pier Paolo



Game Theory: Focus Here

- Two players
  - Max-min
  - Taking turns, fully observable
- Moves: Action
- Position: state
- Zero sum:
  - good for one player, bad for another
  - No win-win outcome.

Types of Games

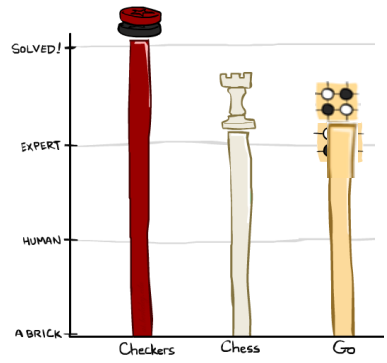
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
    - » We don't make AI to act in isolation, it should a) work around people and b) help people
    - » That means that every AI agent needs to solve a game
- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Can reduce payoff to a single value that one player maximizes and the other minimizes
  - Adversarial, pure competition

Types of games

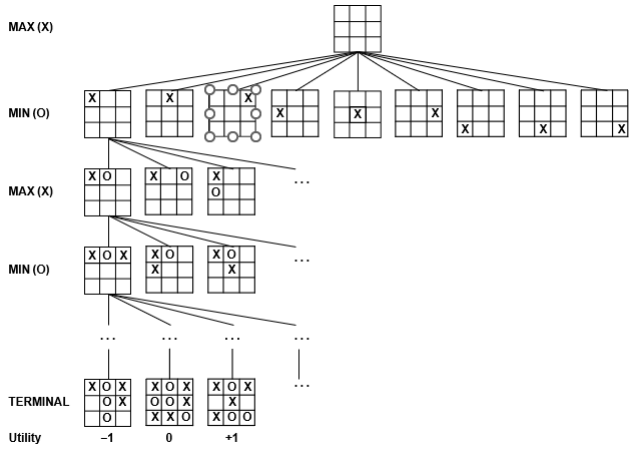
	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships,	bridge, poker, scrabble

Zero-Sum Game Games ☺

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Go :2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.**



Game tree (2-player, deterministic, turns)

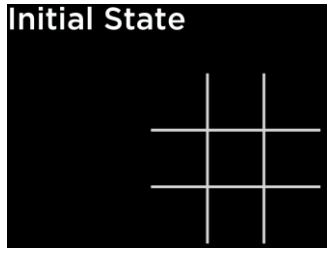


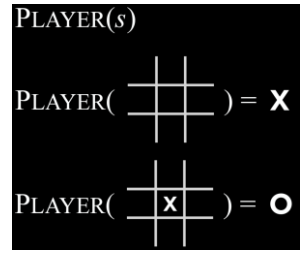
Deterministic Games with Terminal Utilities

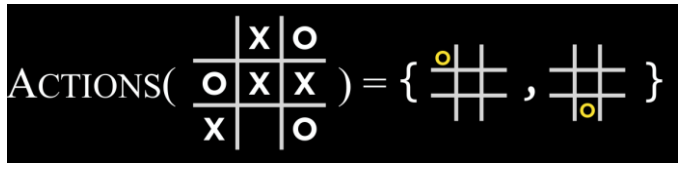
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R

- Solution for a player is a policy: S $\rightarrow$ A

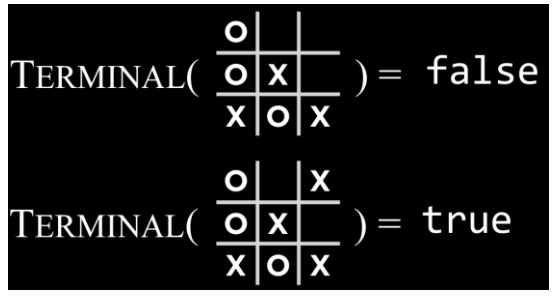
Tic Tac Toe

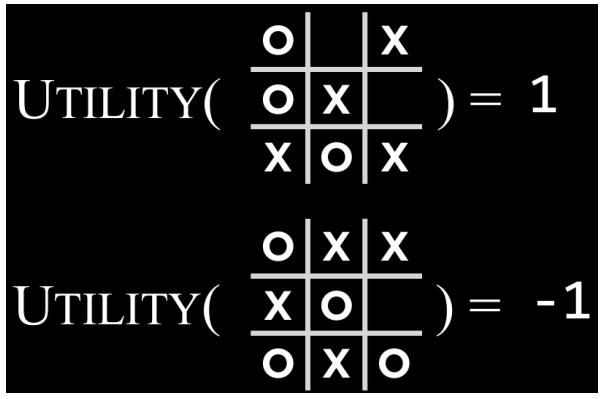




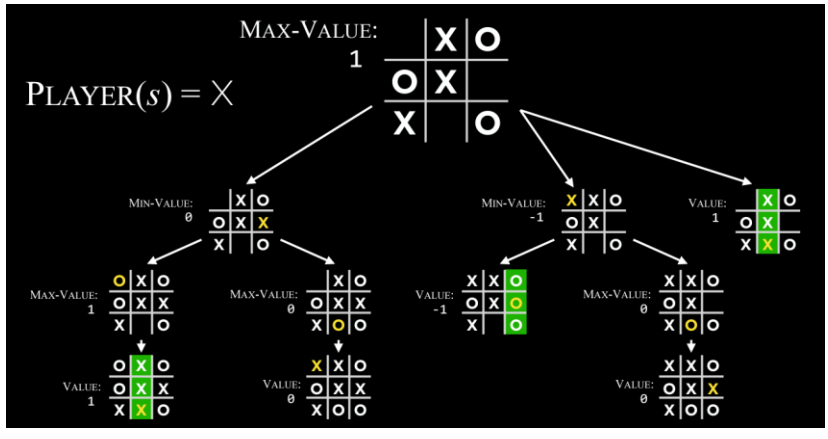


Tic Tac Toe





Players: Max, Min



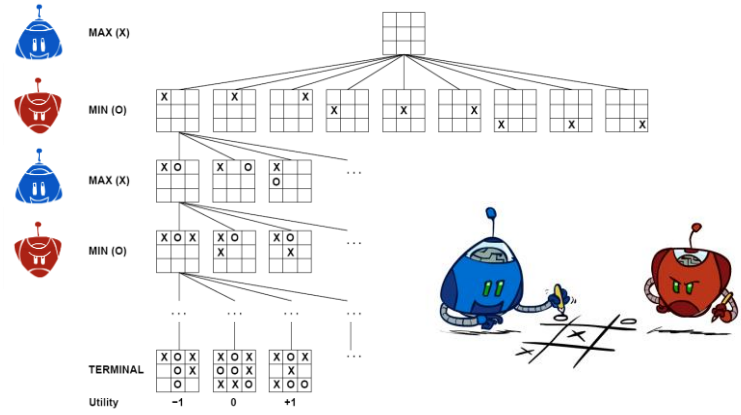
New: Cost -> Utility!

- no longer minimizing cost!
- agent now wants to maximize its score/utility!

Maximizing *utility* not minimizing *cost*

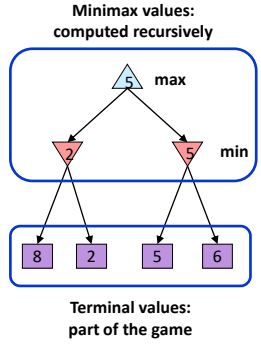
- Utilities are functions from outcomes (states of the world) to real numbers that describe an agent's preferences
- Where do utilities come from?
  - In a game, may be simple (+1/-1)
  - Utilities summarize the agent's goals
  - Theorem: any "rational" preferences can be summarized as a utility function
- We hard-wire utilities and let behaviors emerge
  - Why don't we let agents pick utilities?
  - Why don't we prescribe behaviors?

Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

⟺

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s') \qquad V(s') = \min_{s \in \text{successors}(s')} V(s)$$

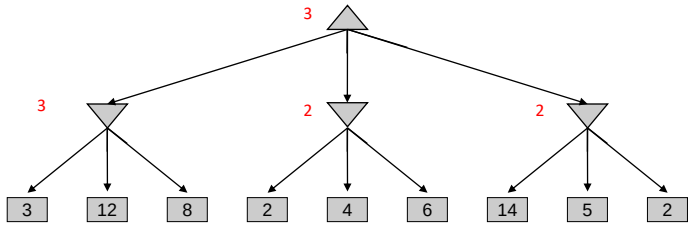
Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

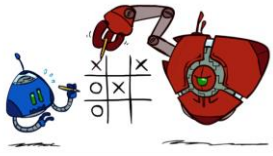
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

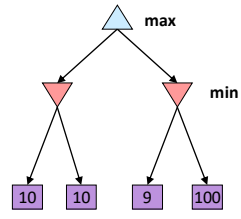
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

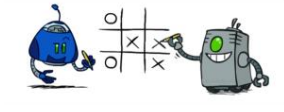
Minimax Example: Trace Code



Minimax Properties



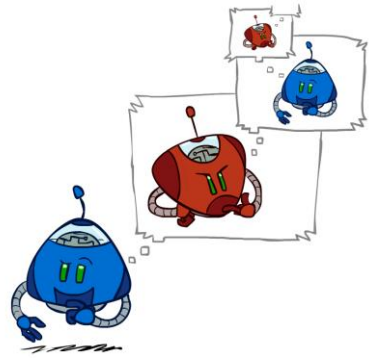




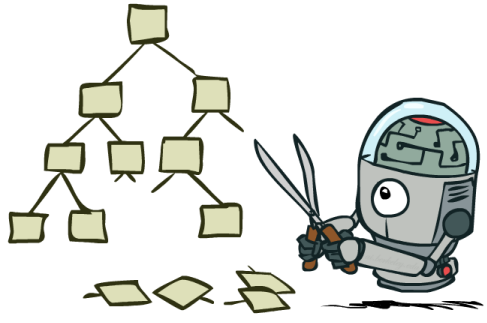
Optimal against a perfect player. Otherwise?

Minimax Efficiency

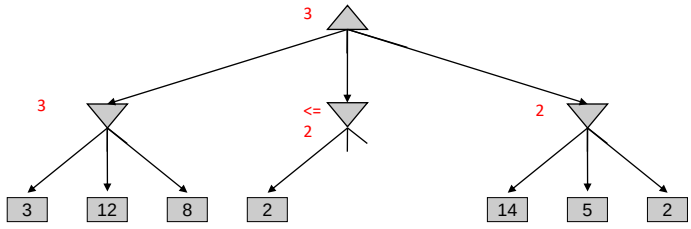
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: O(bm)
- Example: For chess, b ≈ 35, m ≈ 100
  - **Exact solution is completely infeasible**
  - **But, do we need to explore the whole tree?**



Game Tree Pruning

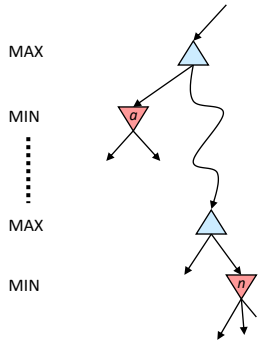


Minimax Example



Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



- MAX version is symmetric

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```


Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

```
def value(state , α, β):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state , α, β)
    if the next agent is MIN: return min-value(state , α, β)
```

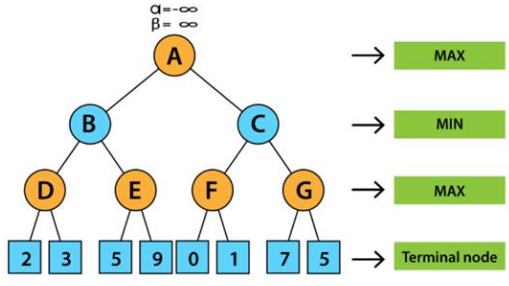
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Prune

α-β Pruning: Detailed Example -1

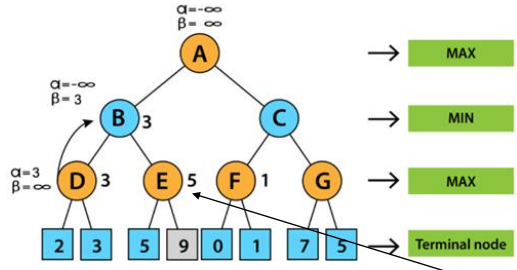
https://people.cs.pitt.edu/~litman/courses/cs2710/lectures/pruningReview.pdf




```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

α-β Pruning: Detailed Example - 2

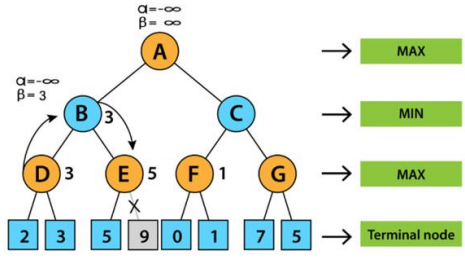



```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

α: MAX's best option on path to root
β: MIN's best option on path to root

α-β Pruning: Detailed Example - 3

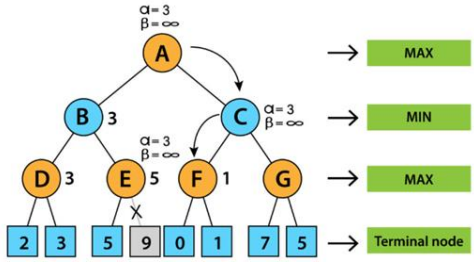



```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

α: MAX's best option on path to root
β: MIN's best option on path to root

α-β Pruning: Detailed Example - 4

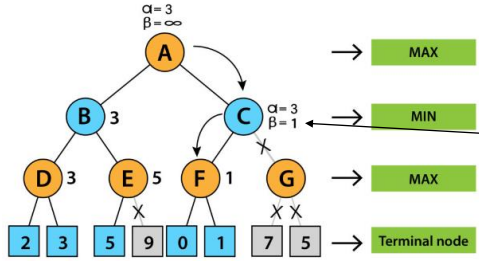



```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

α: MAX's best option on path to root
β: MIN's best option on path to root

α-β Pruning: Detailed Example - 5

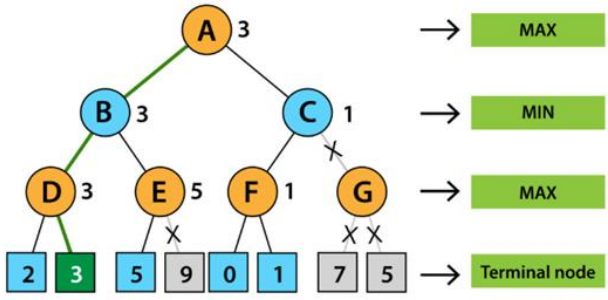



```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v

def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

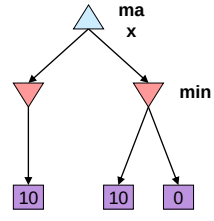
α: MAX's best option on path to root
β: MIN's best option on path to root

α-β Pruning: Detailed Example - 6

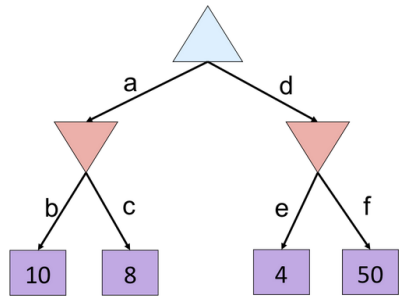


Alpha-Beta Pruning Properties

- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of metareasoning (computing about what to compute)

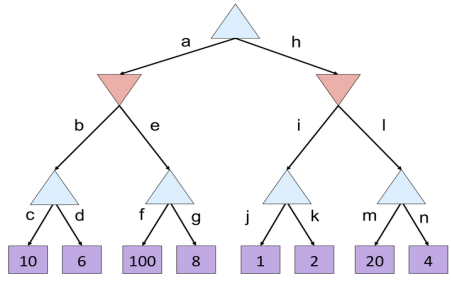


Alpha-Beta Quiz



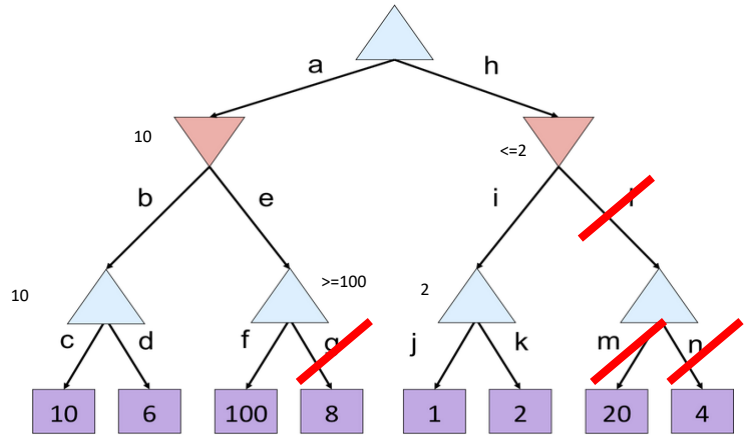
α: MAX's best option on path to root
β: MIN's best option on path to root

Alpha-Beta Quiz 2



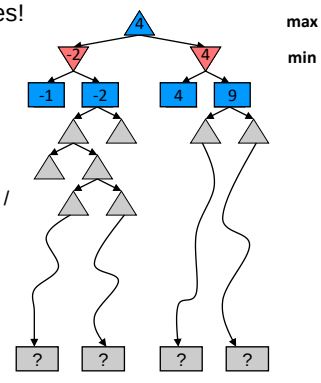
α: MAX's best option on path to root
β: MIN's best option on path to root

Alpha-Beta Quiz 2



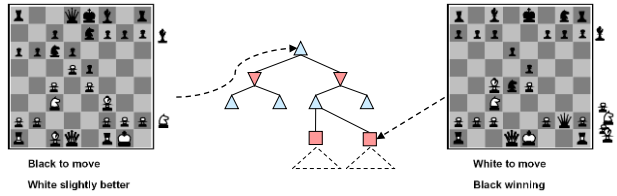
Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search



- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

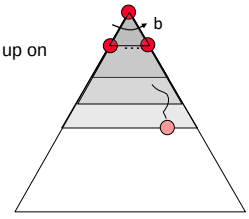
[Demo: depth limited (L6D4, L6D5)]

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If "1" failed, do a DFS which only searches paths of length 2 or less.
3. If "2" failed, do a DFS which only searches paths of length 3 or less.

….and so on.



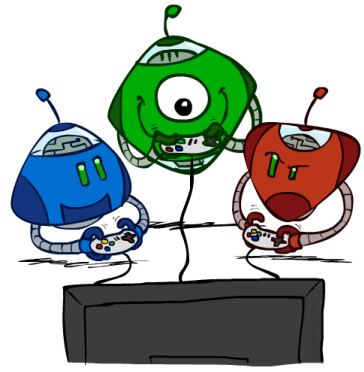
Why do we want to do this for multiplayer games?

Note: wrongness of eval functions matters less and less the deeper the search goes!

Summary

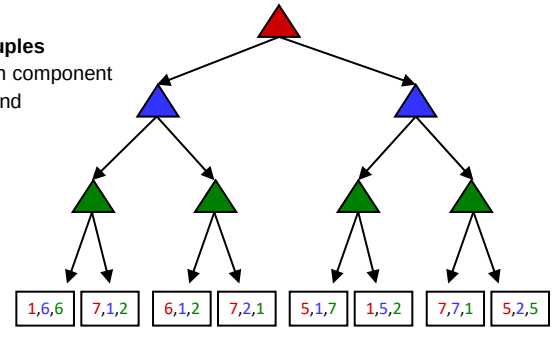
- Game playing is best modeled as a search problem
- Game trees represent alternate computer/opponent moves
- Minimax is a procedure which chooses moves by assuming that the opponent will always choose the move which is best for them
  - Basically, it avoids all worst-case outcomes for Max, to find the best.
  - If the opponent makes an error, Minimax will take optimal advantage of that error and make the best possible play that exploits the error.
- Cutting off search
  - In general, it is infeasible to search the entire game tree.
  - In practice, Cutoff-Test decides when to stop searching further.
- Static heuristic evaluation function
  - Estimate quality of a given board configuration for the Max player.
  - Called when search is cut off, to determine value of position found.

Other Game Types

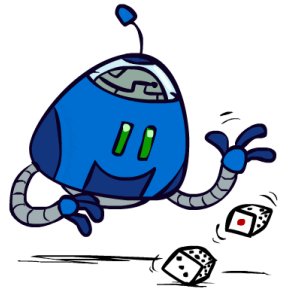


Multi-Agent

- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have **utility tuples**
  - Node values are also **utility tuples**
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…



Uncertain Outcomes



What are Probabilities?

- **Objectivist / frequentist** answer:
  - Averages over repeated experiments
  - E.g. empirically estimating P(rain) from historical observation
  - Assertion about how future experiments will go (in the limit)
  - Makes one think of inherently random events, like rolling dice
- **Subjectivist / Bayesian** answer:
  - Degrees of belief about unobserved variables
  - E.g. an agent's belief that it's raining, given the temperature
  - E.g. player_1's belief that player_2 will take an action, given the state
  - Often learn probabilities from past experiences (more later)
  - New evidence updates beliefs (more later)

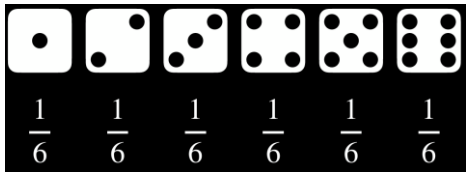
Probability: Basic Concepts

- Uncertainty can be represented as set of events and the likelihood, or probability, of each of them happening.
  - Event can be thought of as a possible state of a world or aspect of the world
- Axioms: Let S be the **sample space (all possible outcomes)**
  - **$0 \leq P(x) \leq 1$**: every probability value must range between 0 and 1
  - $\sum_{x \in S} P(x) = 1$ the sum of the probabilities over the sample space = 1
  - Random Variable: ***F*** : S $\rightarrow$ Possible Values in Real Numbers
  - Expected Value: E(***F***) = $\sum_{values\ that\ F\ takes\ on} F(x) * P(F(x))$

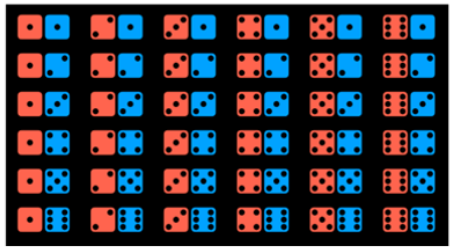
Example: Rolling a single die

- **Fair Die** = Probability of each outcome is 1/6
- Can think of the value as a very simple random variable
- Expected value of rolling a single fair die

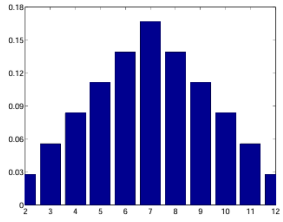
$= 1 \bullet \frac{1}{6} + 2 \bullet \frac{1}{6} + 3 \bullet \frac{1}{6} + 4 \bullet \frac{1}{6} + 5 \bullet \frac{1}{6} + 6 \bullet \frac{1}{6} = \frac{21}{6} = 3\frac{1}{2}$



Example: Rolling a pair of dice: red and blue

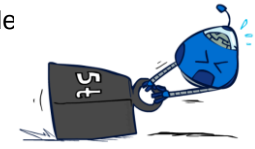


- Probability of each event, e.g. (2_R, 3_B) = 1/36
- Random variable: ***Sum*** of values of the pair
- Probability distribution of ***Sum***



Reminder: Expectations

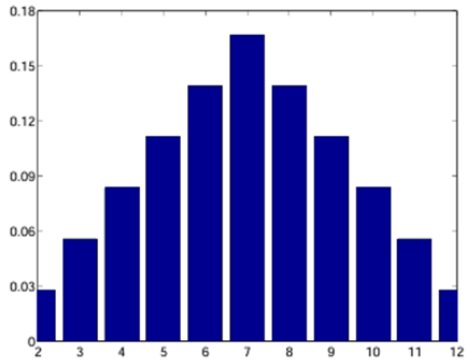
- The expected value of a function of a random variable the average, weighted by the probability distribution over outcomes
- Example: How long to get to the airport?



Time:	20 min	+	30 min	+	60 min	→	35 min
	x		x		x		
Probability:	0.25		0.50		0.25		

Quiz: What is the expected value of the sum from rolling two dice

- Make a rough guess (to check logic as go along)
  - remember E(X) is a *weighted (by probability) average*
- ???? Write the expression
- Do the calculation



Summary: Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

- Example: Traffic on freeway
  - Random variable: T = whether there's traffic
  - Outcomes: T in {none, light, heavy}
  - Distribution: P(T=none) = 0.25, P(T=light) = 0.50, P(T=heavy) = 0.25

- Some laws of probability (more later):
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one

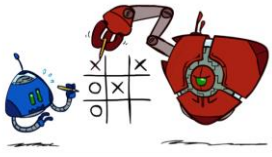
- As we get more evidence, probabilities may change:
  - P(T=heavy) = 0.25, P(T=heavy | Hour=8am) = 0.60
  - We'll talk about methods for reasoning and updating probabilities later

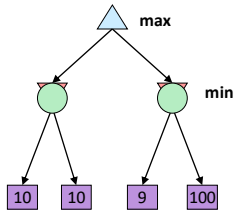
The Dangers of Optimism and Pessimism

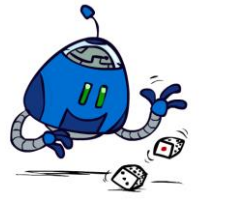
Dangerous Optimism
Assuming chance when the world is adversarial

Dangerous Pessimism
Assuming the worst case when it's not likely

Worst-Case vs. Average Case



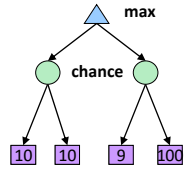




- Idea: Uncertain outcomes controlled by adversary vs chance!
- Can be a combination of the two!!

Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: humans are not perfect
  - Actions can fail: when moving a robot, wheels might slip
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes
- Expectimax search: compute the average score under optimal play
  - **Max nodes** as in minimax search
  - **Chance nodes** are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



[Demo: min vs exp (L7D1,2)]

Expectimax Pseudocode

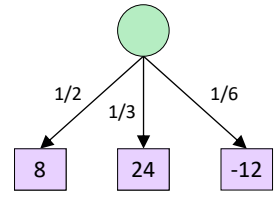
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Expectimax Pseudocode

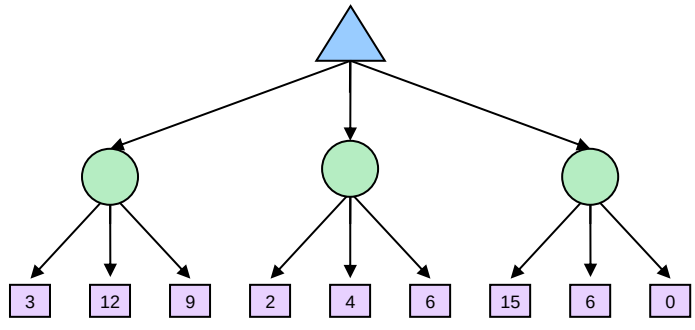
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



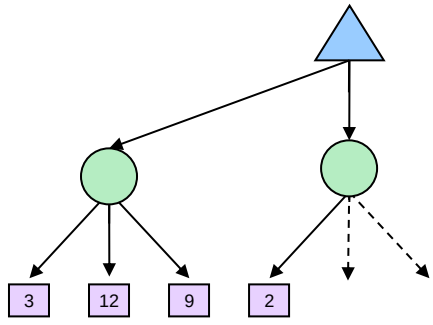
v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

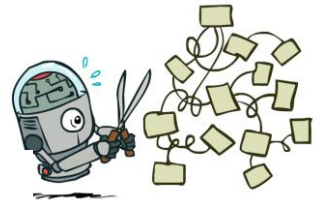
Expectimax Example:

Again –**a postorder traversal**

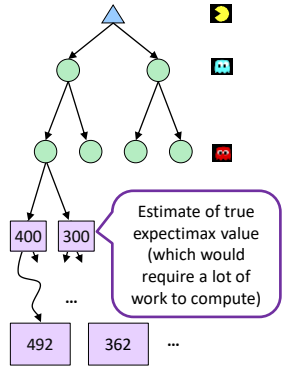


Expectimax Pruning?





Depth-Limited Expectimax



Quiz: Informed Probabilities

- Let's say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise
- Question: What tree search should you use?

- Answer: Expectimax!
  - To figure out EACH chance node's probabilities, you have to run a simulation of your opponent
  - This kind of thing gets very slow very quickly
  - Even worse if you have to simulate your opponent simulating you…

This is basically how you would model a human, except for their utility: their utility might be the same as yours (i.e. you try to help them, but they are depth 2 and noisy), or they might have a slightly different utility (like another person navigating in the office)