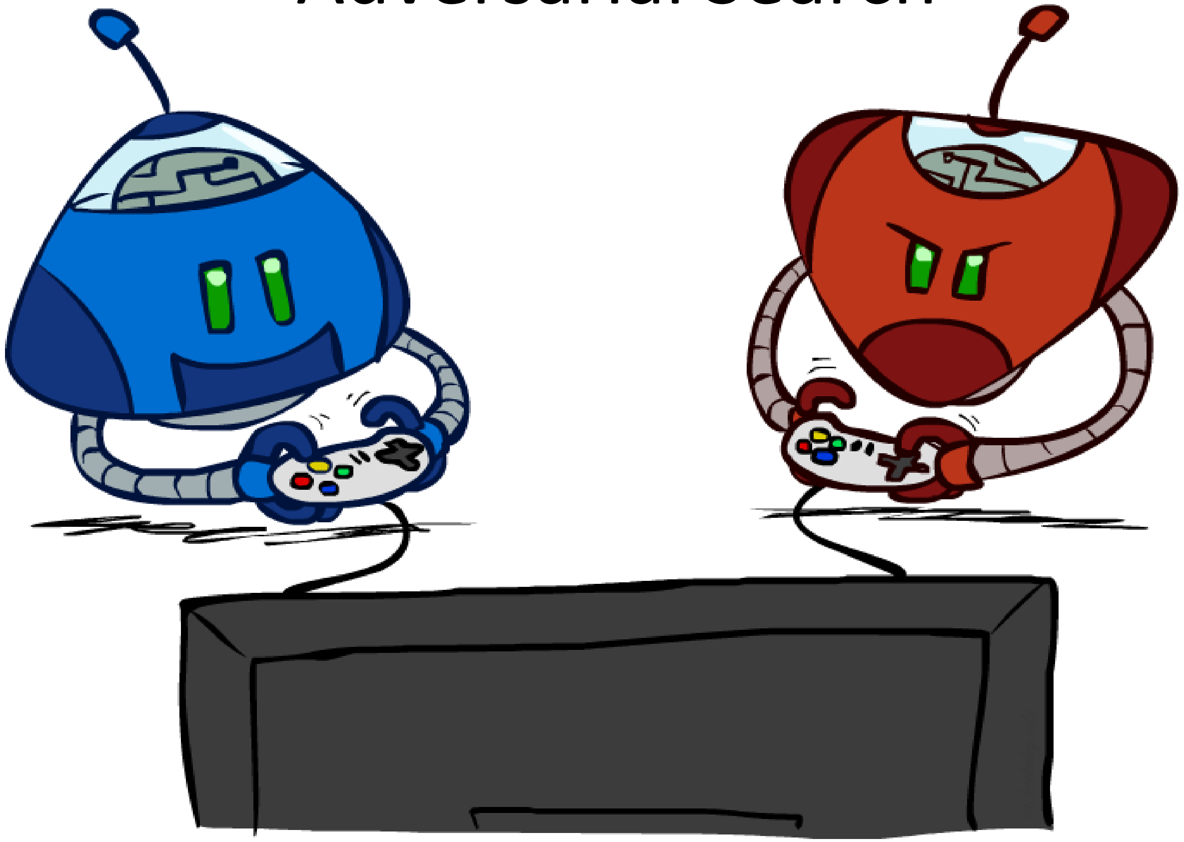
COE 4213564
Introduction to Artificial Intelligence

Adversarial Search



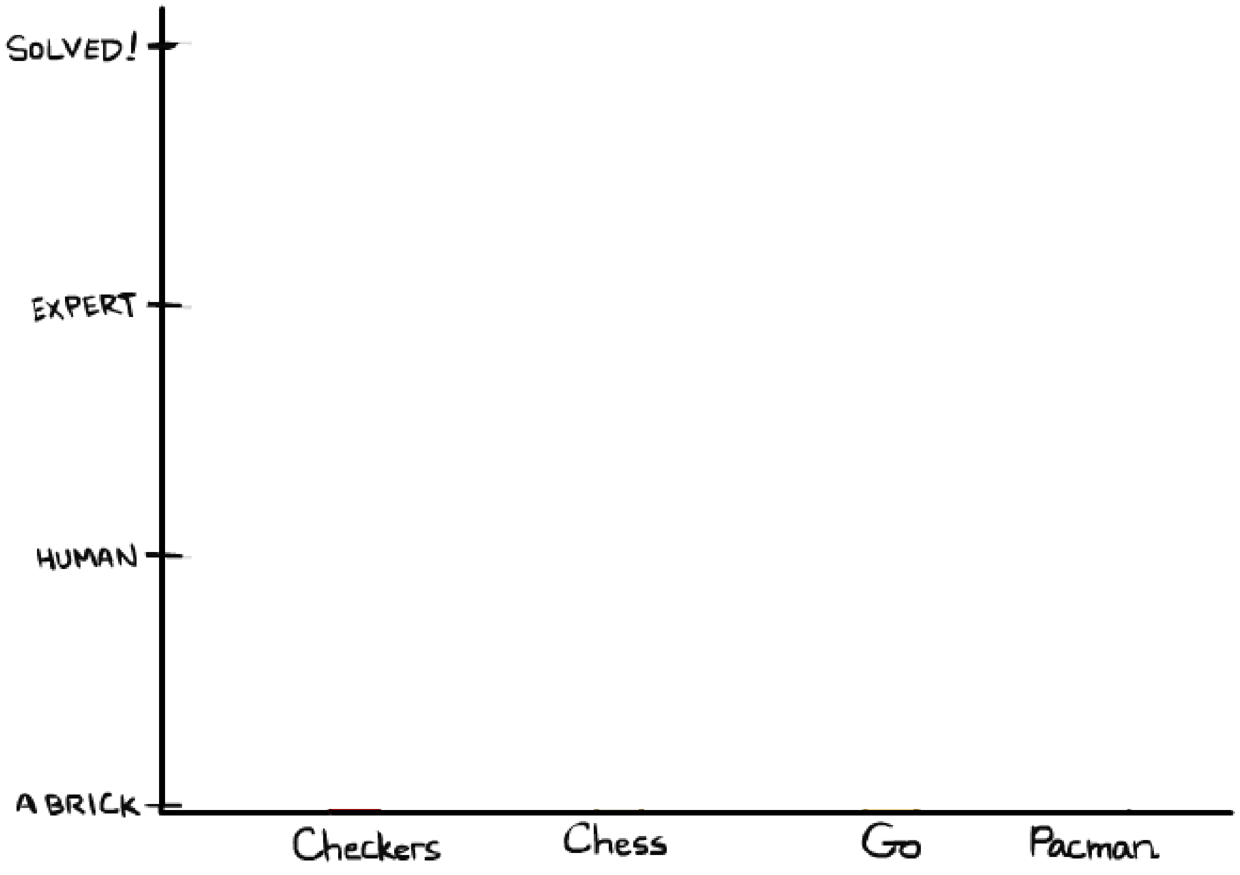
Many slides are adapted from CS 188 (http://ai.berkeley.edu), CS 322, CIS 521, CS 221, CS182, CS4420.

Game Playing State-of-the-Art

- **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!
  https://en.wikipedia.org/wiki/Checkers
  - Solved means you can force to win or draw if you play optimally.

- **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.
  https://en.wikipedia.org/wiki/Chess

- **Go:** Human champions are now starting to be challenged by machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.
  https://en.wikipedia.org/wiki/Go_(game)



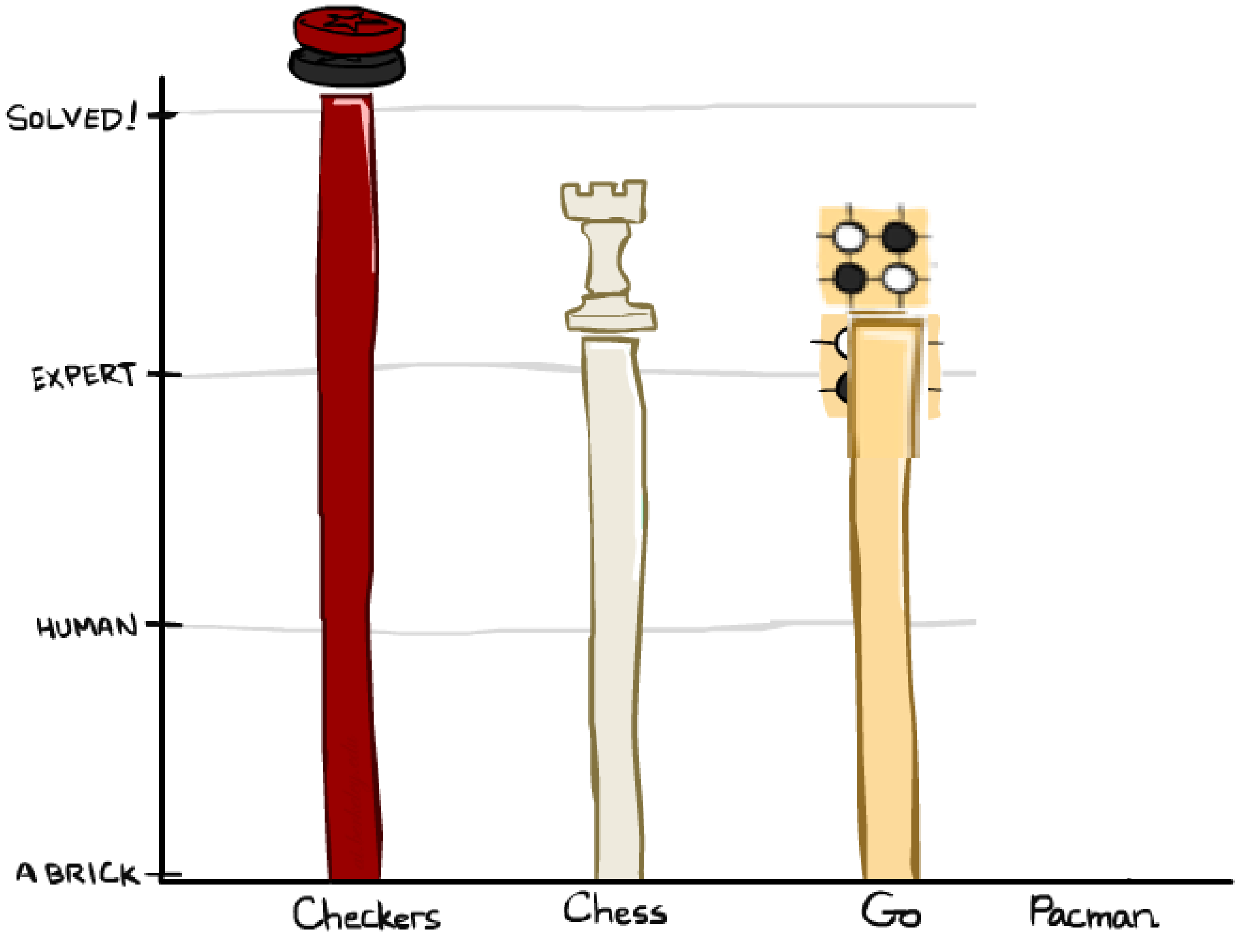
Game Playing State-of-the-Art

- **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!

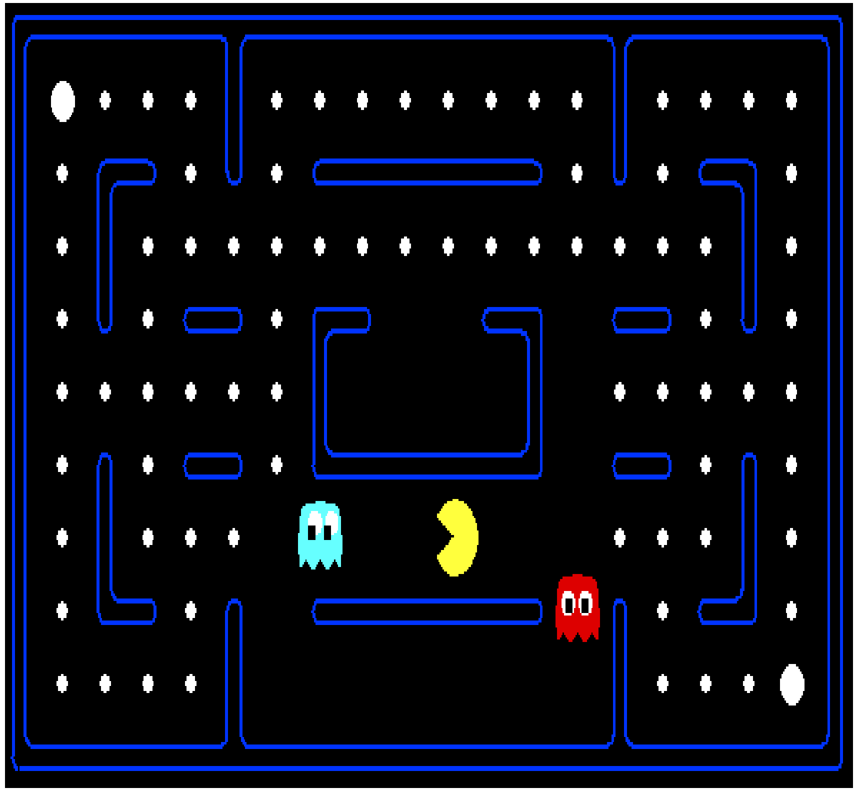
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go: 2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.**

- **Pacman**



Behavior from Computation



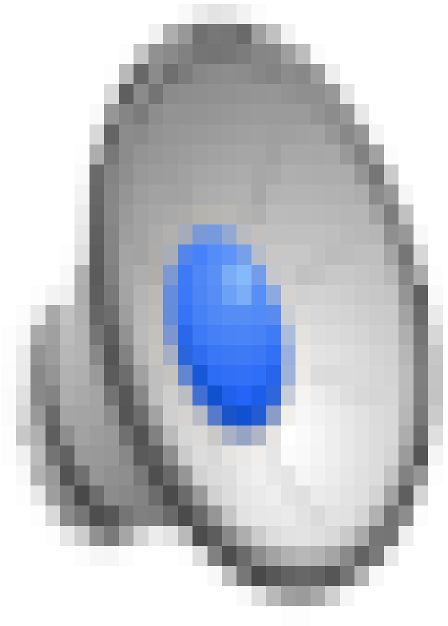
- Pac-man — The Protagonist
- Inky and Clyde — The Antagonists
- Pellet — The food source of our hungry friend
- Power Pellet — the object that renders Pac-man's adversaries edible.

Blinky, **Pinky**, **Inky** and **Clyde**, collectively known as the **Ghost Gang**, are a quartet of characters from the *Pac-Man* video game franchise. Created by Toru Iwatani, they first appear in the 1980 arcade game *Pac-Man* as the main antagonists.

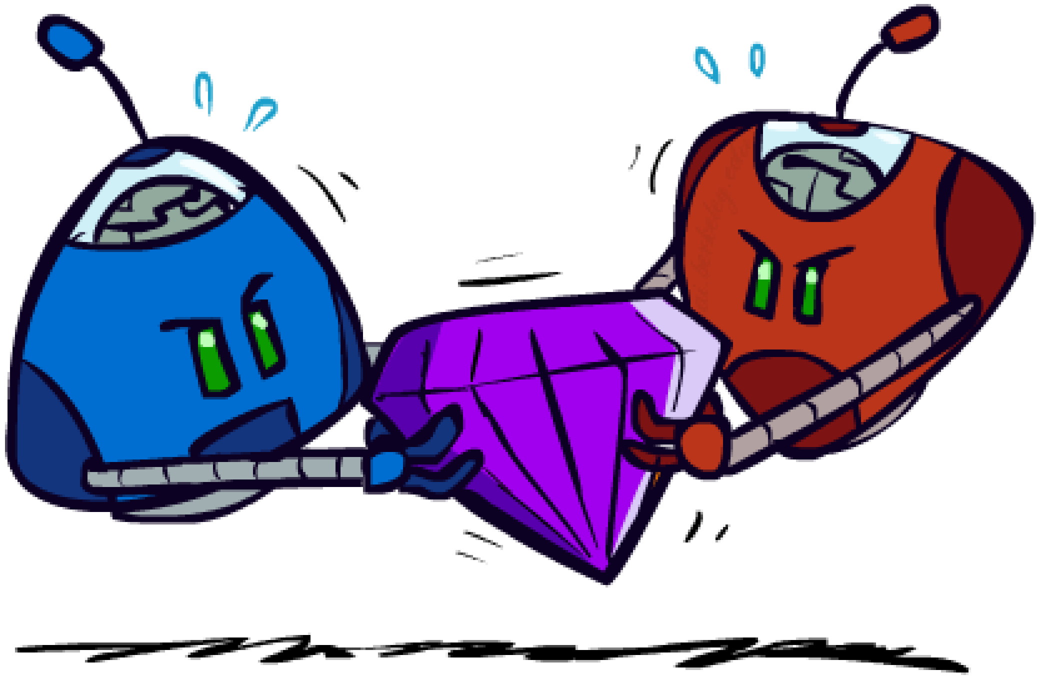
- Demo: Pacman eating food pellets, avoiding ghosts, eating power pellets and then eating ghosts and getting extra score.

[Demo: mystery pacman (L6D1)]

Video of Demo Mystery Pacman



Adversarial Games



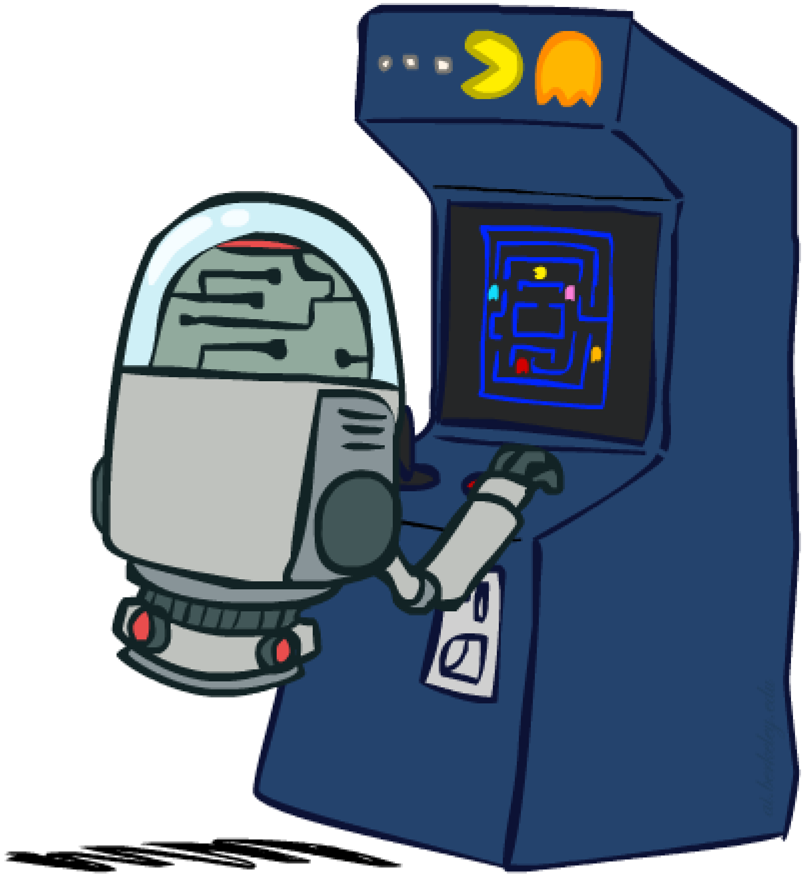
Types of Games

- Many different kinds of games!
  - How to categorize?
- Axes:
  - Deterministic or stochastic?
    Deteministic ex: Checkers, Chess
    Stocastic ex: backgammon (throw a dice)
  - One, two, or more players?
  - Zero sum? (All playing against each other)
  - Perfect information (can you see the state)?
    Do you know everything about the current situation of the game? Chess: Yes; Poker: No (don't know other player's cards.)

- Want algorithms for calculating a strategy (policy) which recommends a move from each state
- By considering an opponent that we don't control

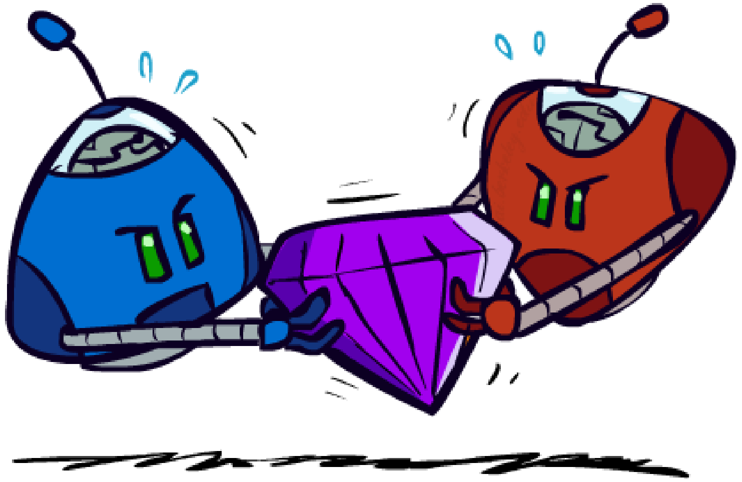
Deterministic Games

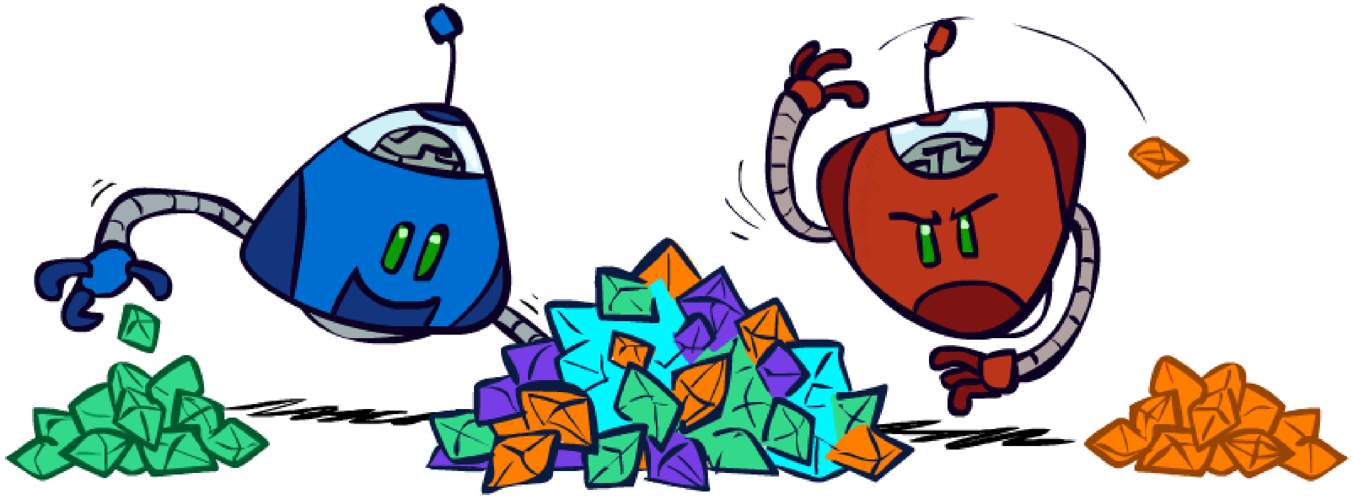
- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R
    (Every outcome of the game will be
    scored like win, lose, draw, amount of money,
    numerical score)



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

Zero-Sum Games

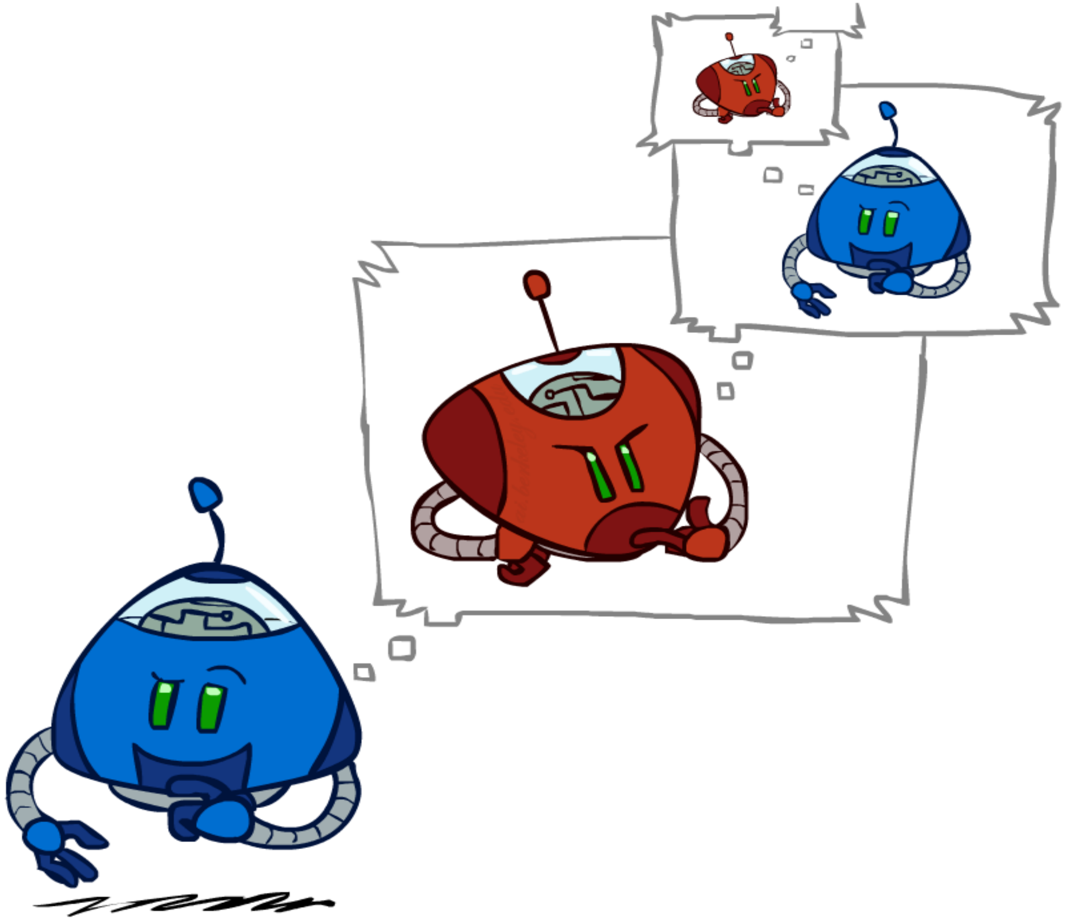




- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
    - one agent gets it other one doesn't get it
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

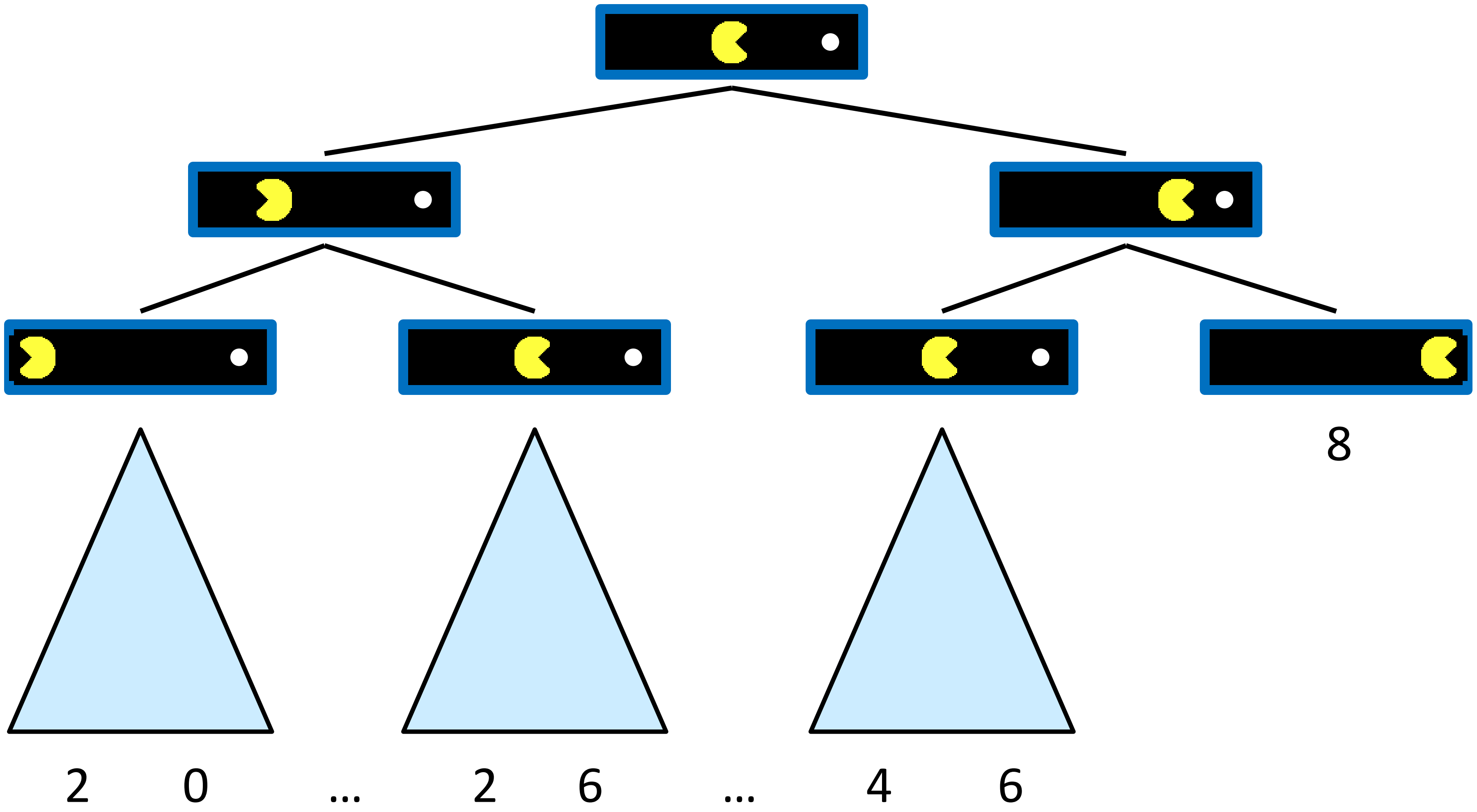
- General Games
  - Agents have independent utilities (values on outcomes)
  - Ex: Blue agent collect green jewels and red one collect orange jewels by helping each other.
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

Adversarial Search



- For zero-sum games, we use approach adverserial search.
- Competitive environments, in which two or more agents have conflicting goals, giving rise to adversarial search problems

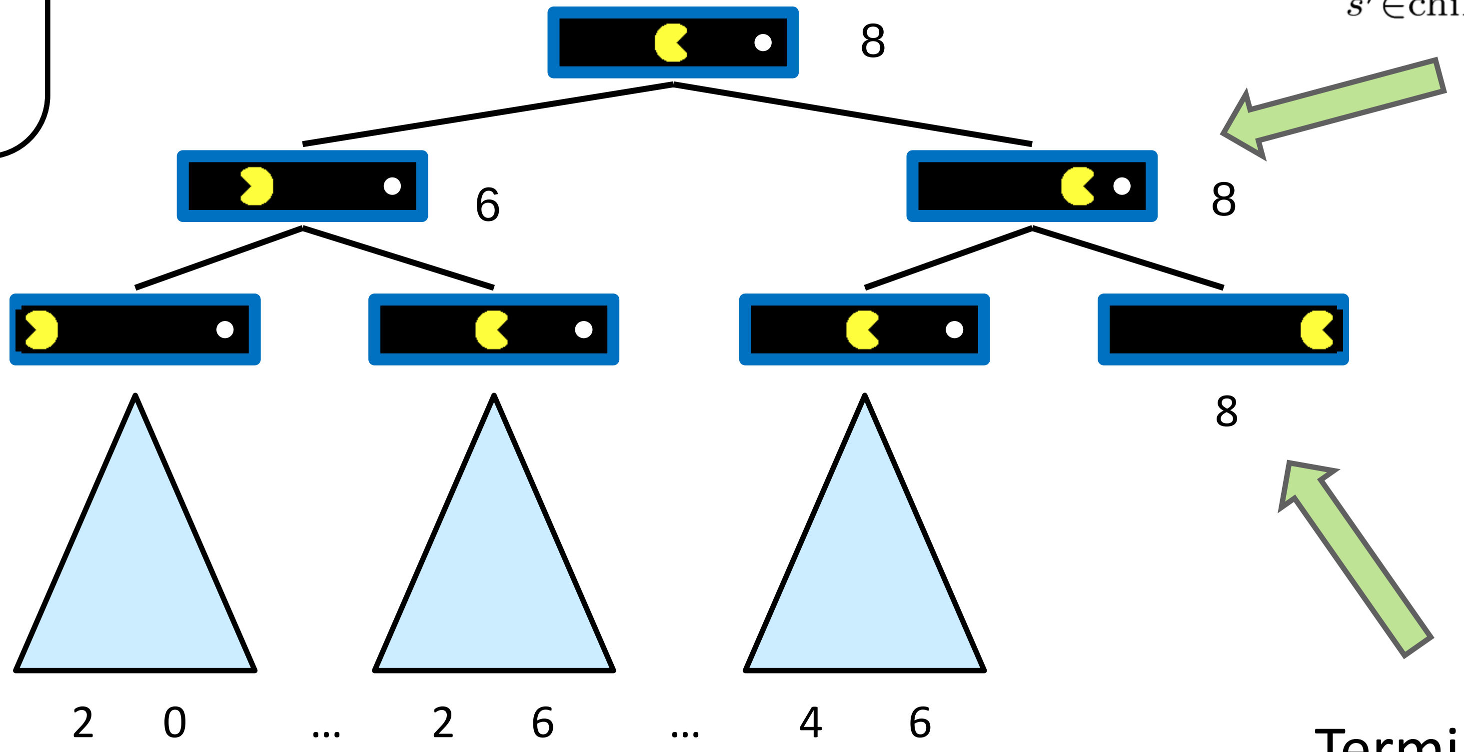
Single-Agent Trees



- Let’s look at first to single-agent trees and generalize it to two-agent trees. - Pacman trying to eat food pellets; actions : east and west
- Utility function: -1 for every steps taken; +10 for every pellets eaten

Value of a State

Value of a state:
The best achievable outcome (utility) from that state



Non-Terminal States:

$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

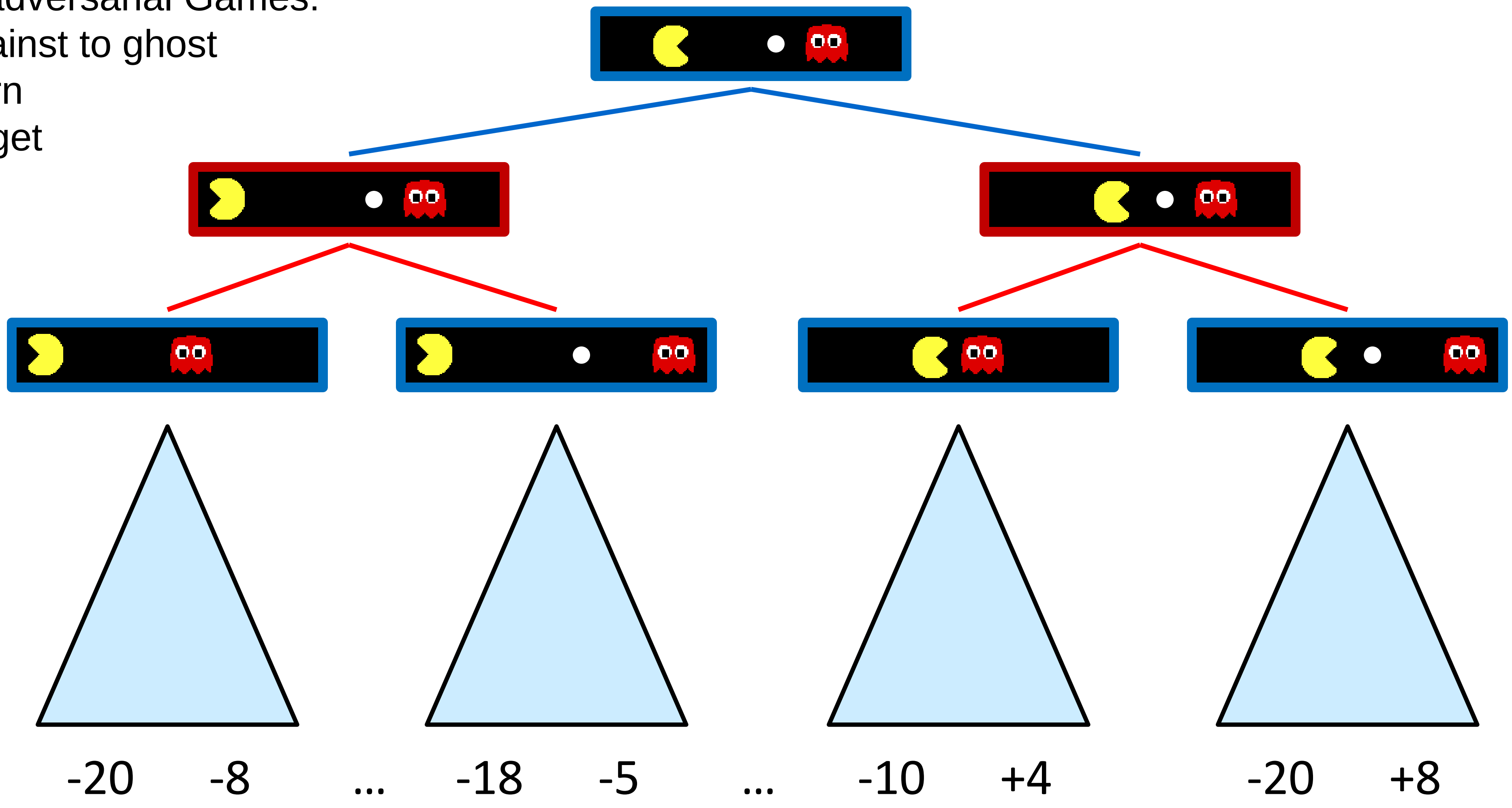
Terminal States:

$$V(s) = \text{known}$$

Adversarial Game Trees

Generalize to adversarial Games:

- Pacman against to ghost
- Moves in turn
- At the end, get
  a high score
  a low score



Minimax Values

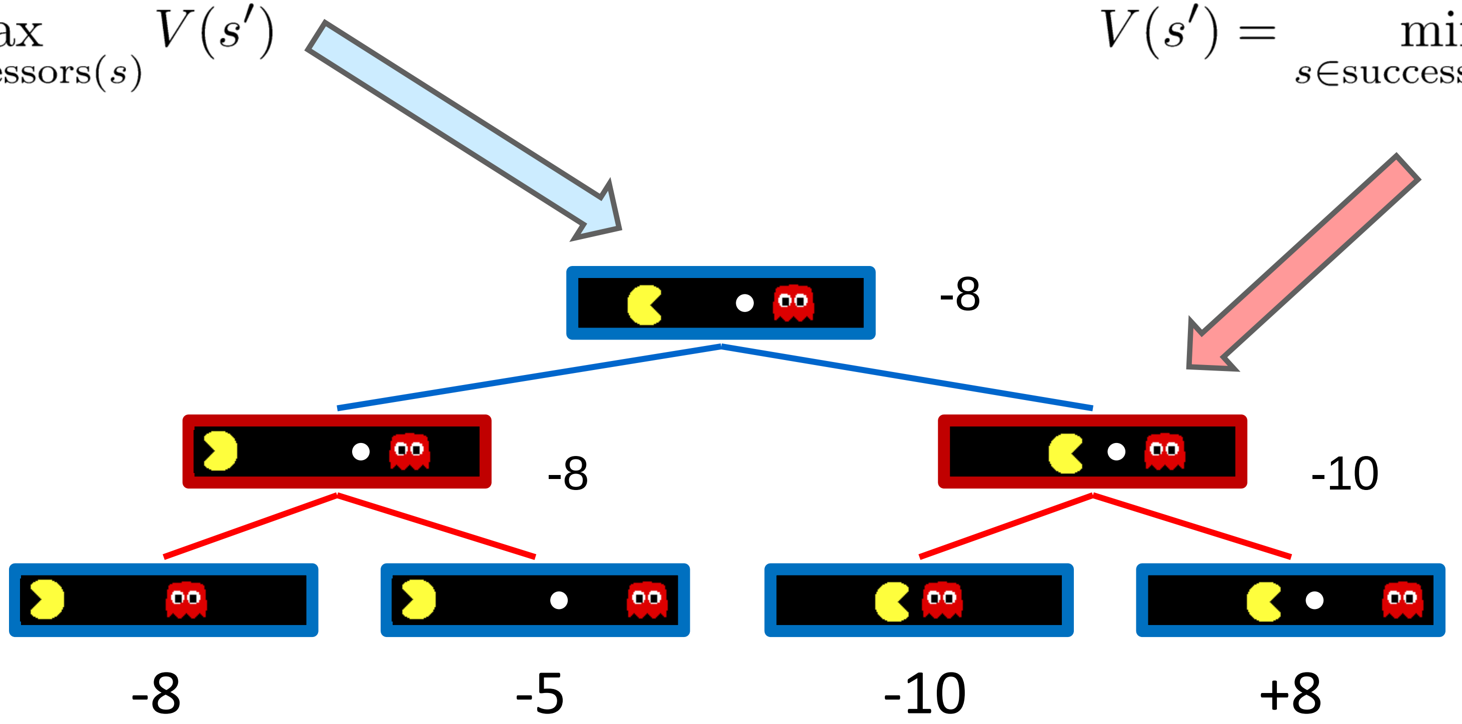
States Under Agent's Control:

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

States Under Opponent's Control:

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

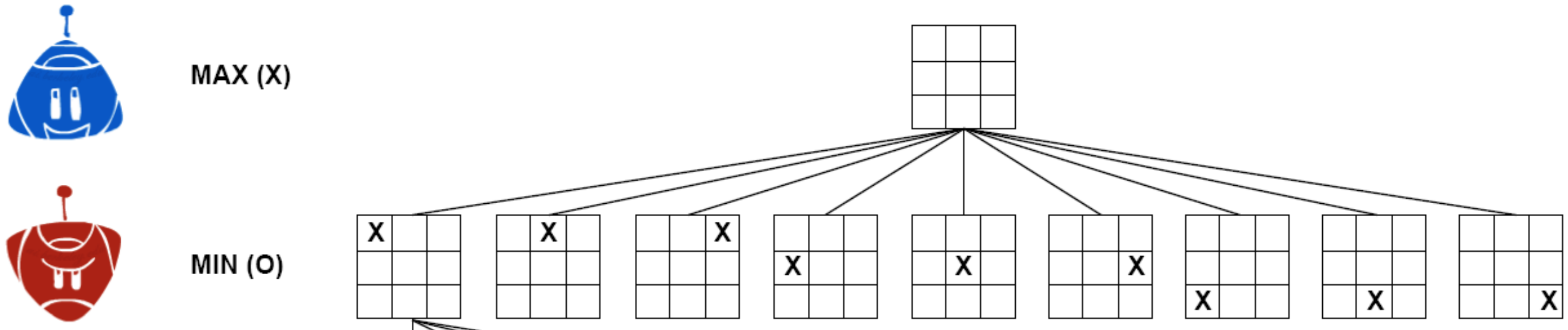
- Assume that game is over after each player makes a move.
- Packman tries to maximize while ghost tries to minimize utility scores

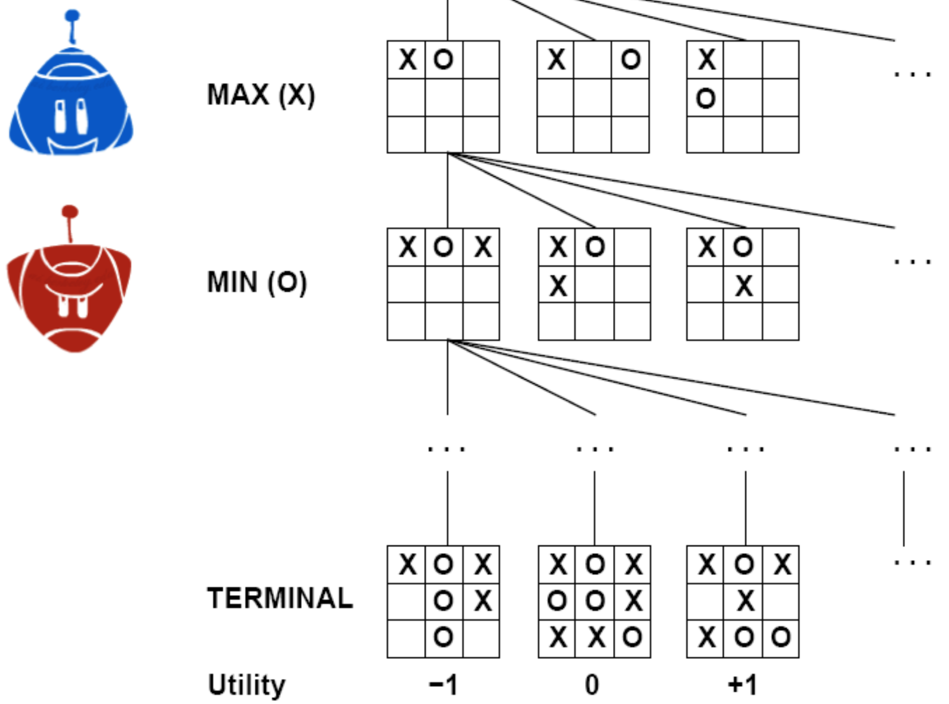


Terminal States:

$$V(s) = \text{known}$$

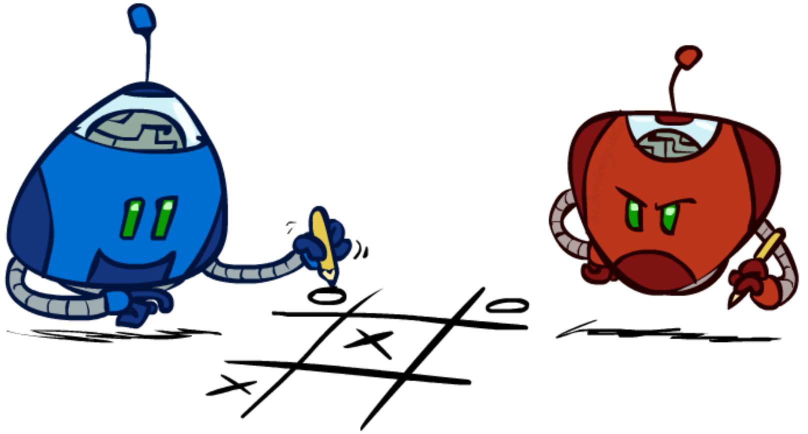
Tic-Tac-Toe Game Tree





The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

For tic-tac-toe the game tree is relatively small—fewer than 9!=362;880 terminal nodes (with only 5,478 distinct states). But for chess there are over 10^40 nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.

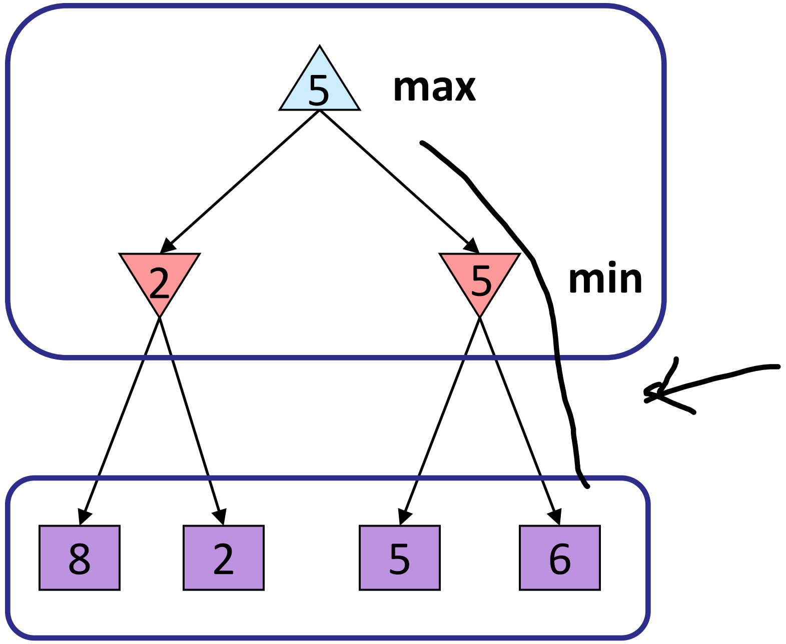


Adversarial Search (Minimax)

- 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 values: computed recursively



Terminal values: part of the game

Consider the trivial game in Figure 6.2. The possible moves for MAX at the root node are labeled a_1, a_2, and a_3. The possible replies to a_1 for MIN are b_1, b_2, b_3, and so on. This particular game ends after one move each by MAX and MIN. (Note: In some games, the word "move" means that both players have taken an action; therefore the word **ply** is used to unambiguously mean one move by one player, bringing us one level deeper in the game tree.) The utilities of the terminal states in this game range from 2 to 14.

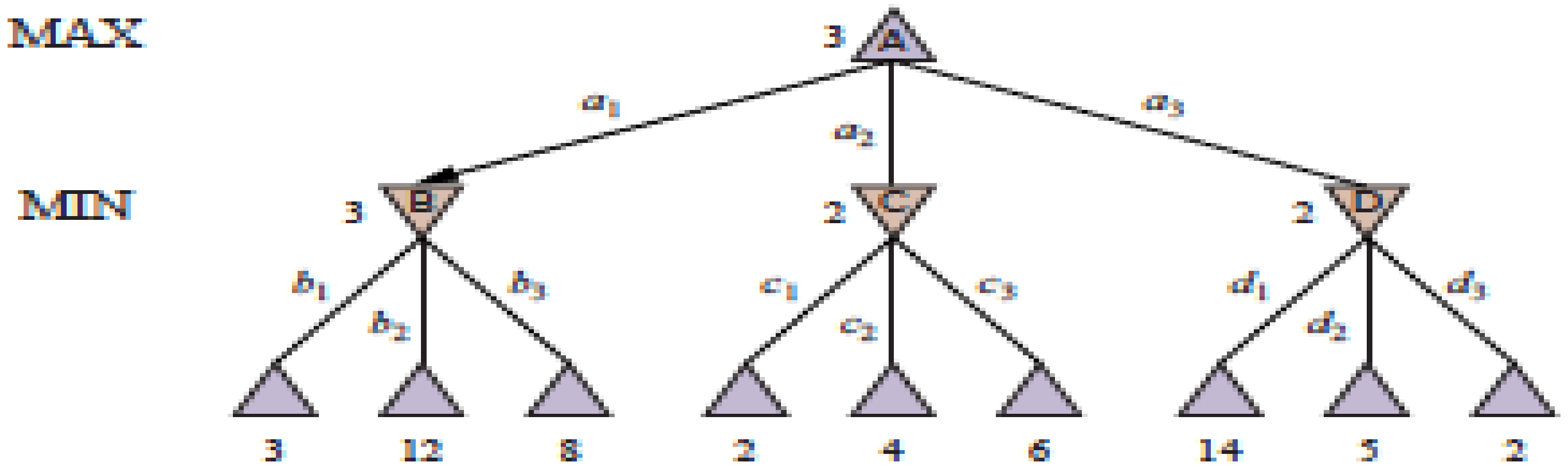


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state 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 Implementation

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

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

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

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

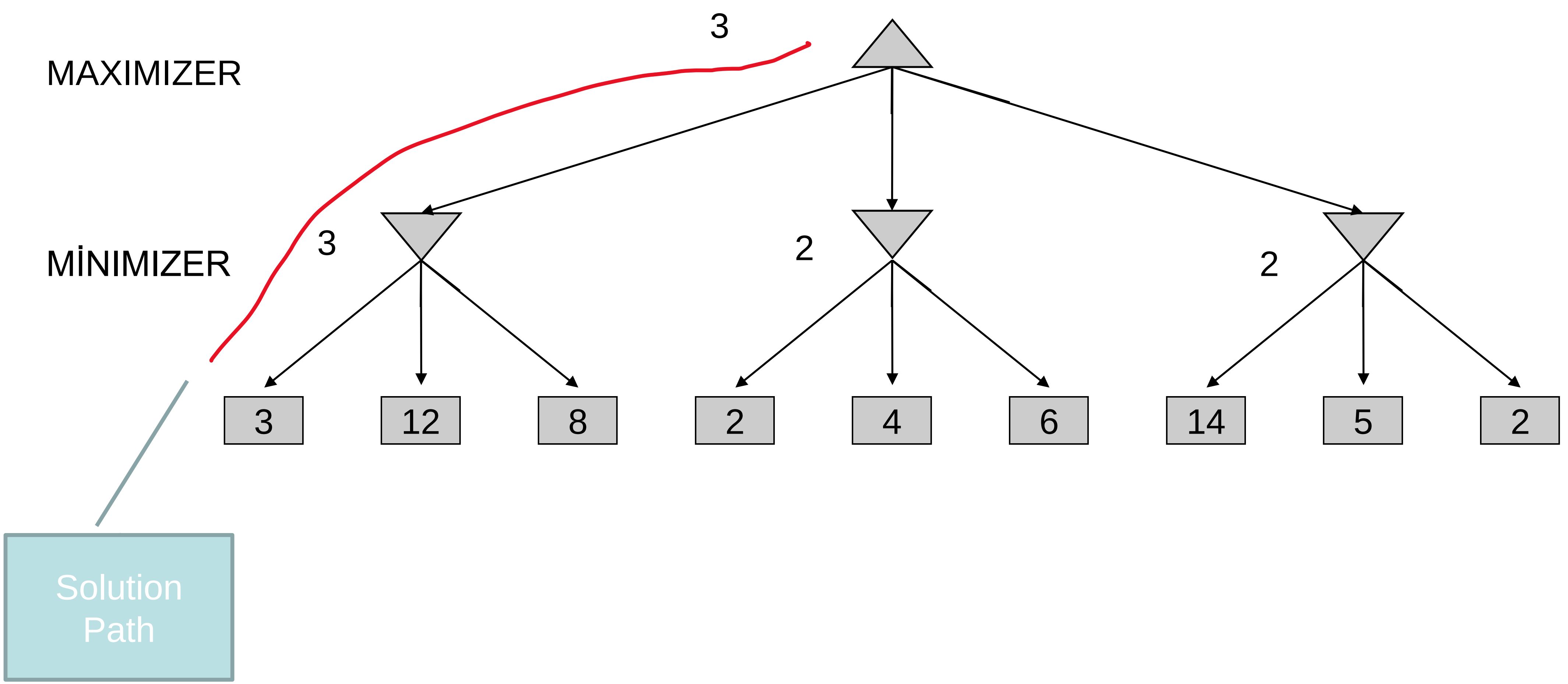
Minimax Implementation (Dispatch)

```
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)
```

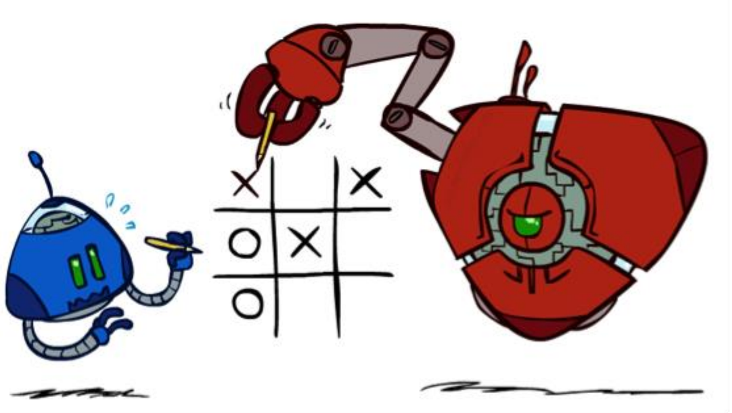
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

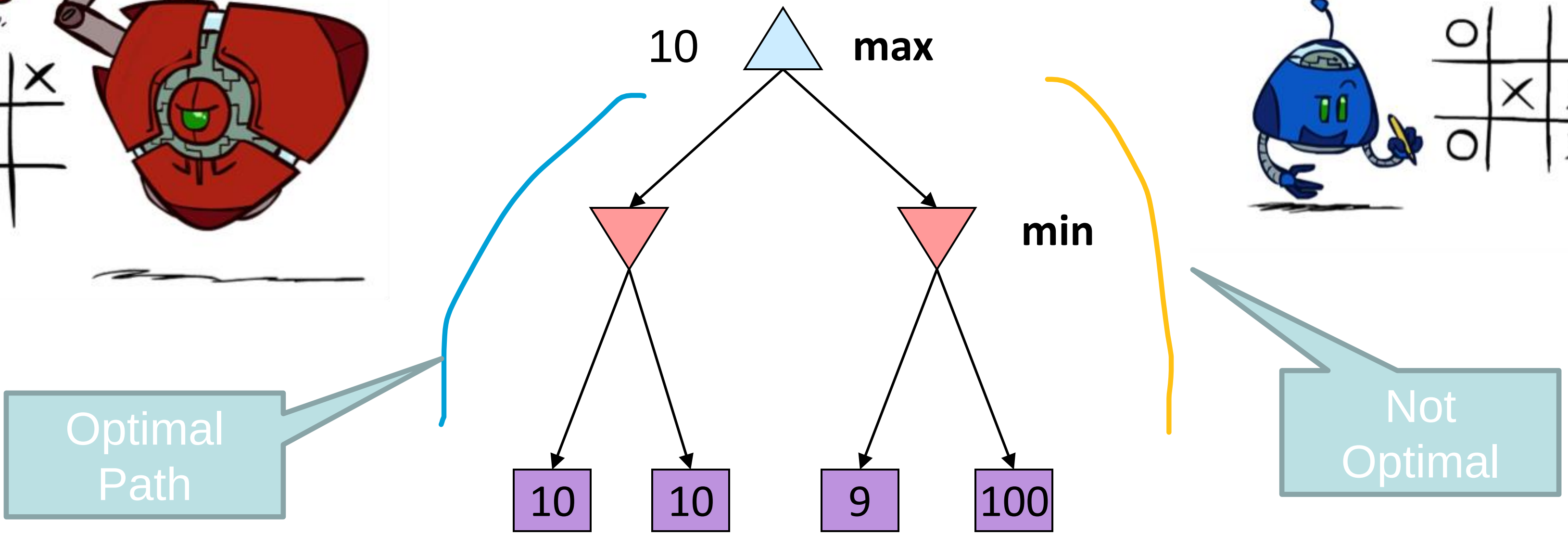
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

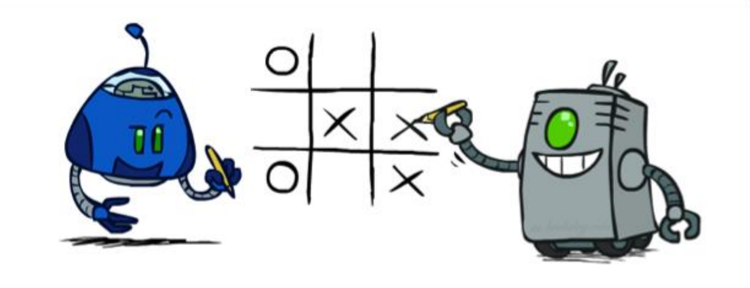
Minimax Example



Minimax Properties







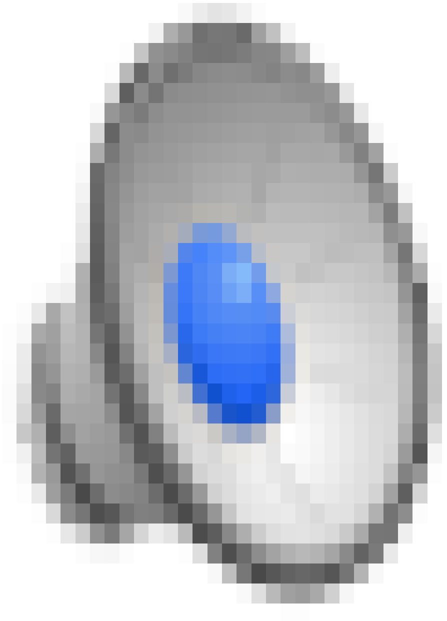
Optimal against a perfect player. Otherwise?

Player may mistakes. Not optimal, then select a different path stochastically.

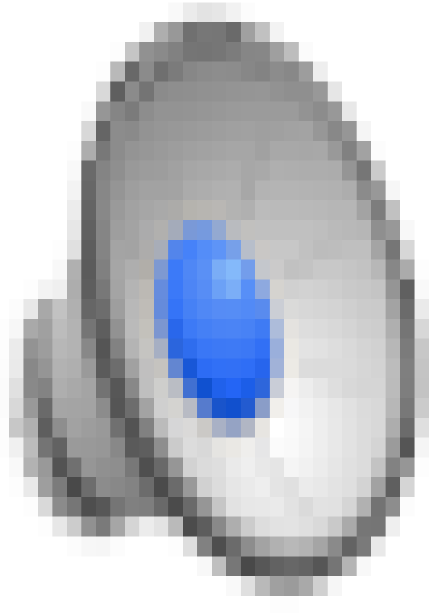
Demo 1: Min score by eaten by ghost in one step. (Minmax solution)
Demo 2: Not smart, taking random actions and takes risks. It might give better solutions at different runs.

[Demo: min vs exp (L6D2, L6D3)]

Video of Demo Min vs. Exp (Min)



Video of Demo Min vs. Exp (Exp)



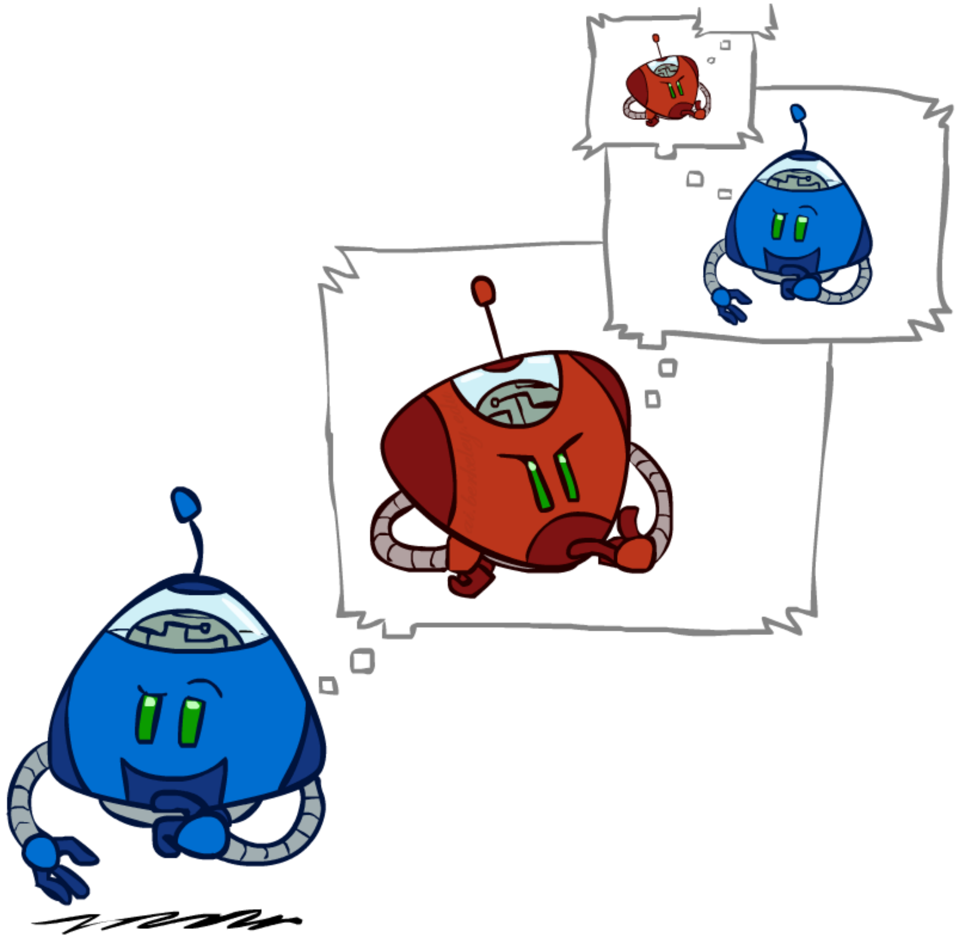
Minimax Efficiency

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all actions at once, or $O(m)$ for an algorithm that generates actions one at a time (see page 98). The exponential complexity makes MINIMAX impractical for complex games; for example, chess has a branching factor of about 35 and the average game has depth of about 80 ply, and it is not feasible to search $35^{80} \approx 10^{123}$ states. MINIMAX does, however, serve as a basis for the mathematical analysis of games. By approximating the minimax analysis in various ways, we can derive more practical algorithms.

Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

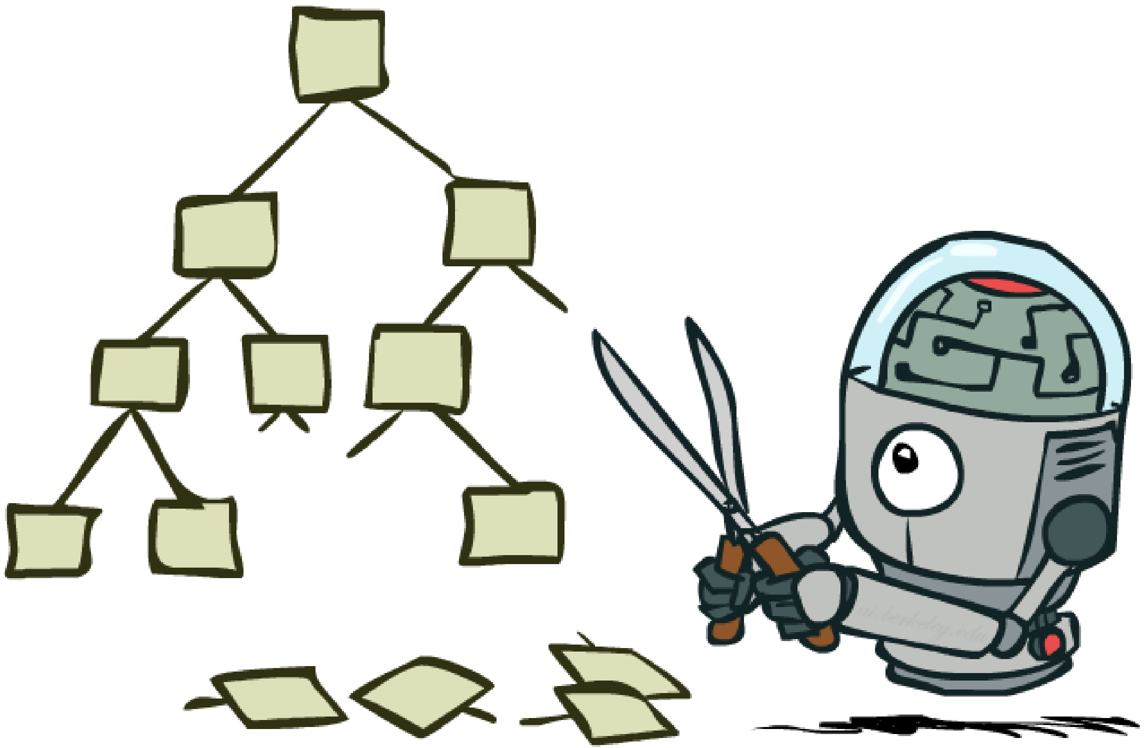
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?



Resource Limits

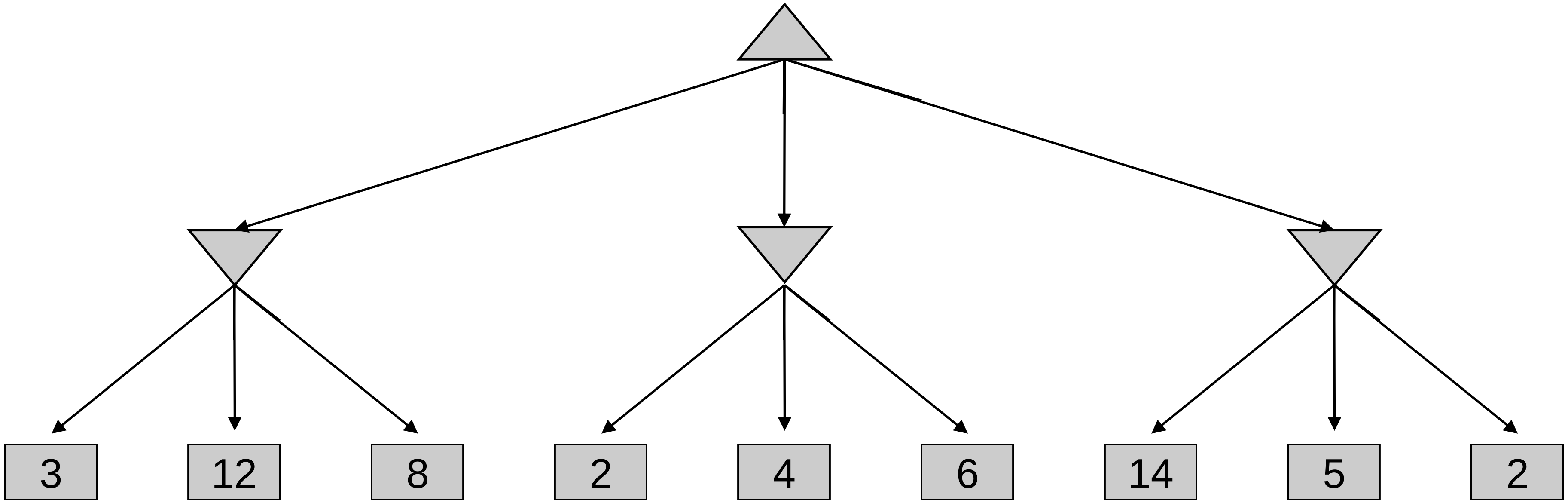
We can not explore the entire game tree because of finite computing power

Game Tree Pruning

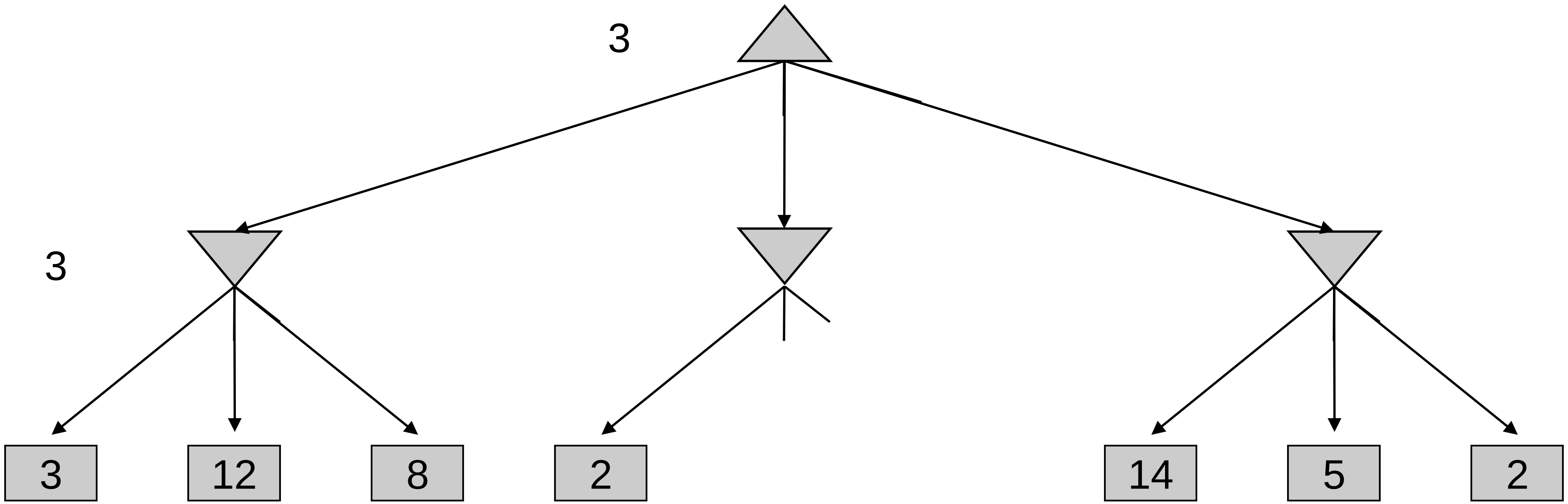


*The number of game states is exponential in the depth of the tree. No algorithm cancompletely eliminate the exponent, but we can sometimes cut it in half, computing the correct minimax decision without examining every state by pruning large parts of the tree that make no difference to the outcome. The particular technique we examine is called **alpha–beta pruning.***

Minimax Example



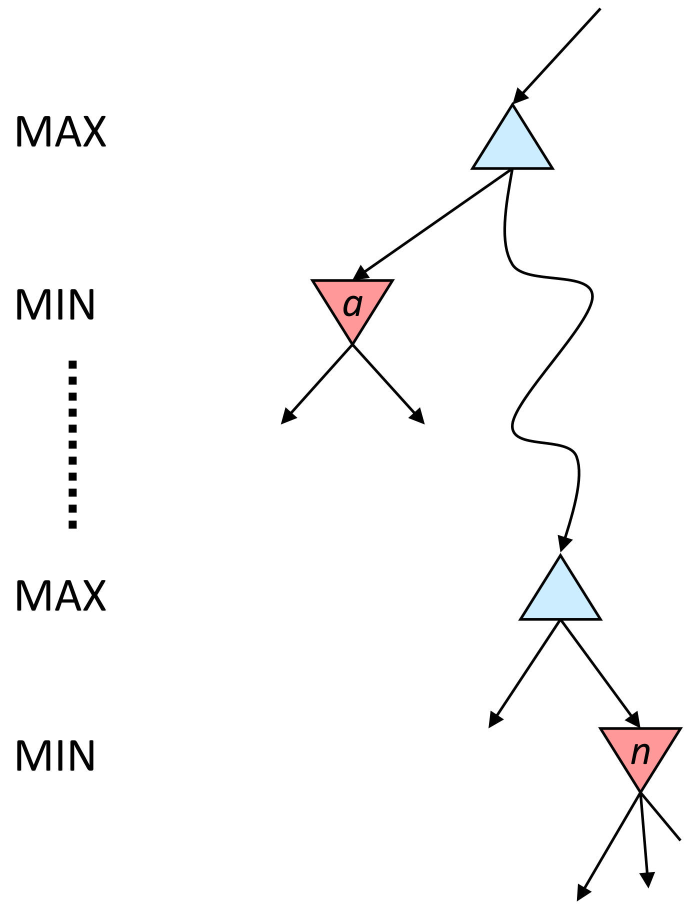
Minimax Pruning



Skip the nodes that has values less than 3

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 Pruning

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the two extra parameters in MAX-VALUE(*state*, α, β) (see Figure 6.7) that describe bounds on the backed-up values that appear anywhere along the path:

$\alpha =$ the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: $\alpha =$ "at least."

$\beta =$ the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: $\beta =$ "at most."

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively. The complete algorithm is given in Figure 6.7. Figure 6.5 traces the progress of the algorithm on a game tree.

Alpha-Beta Pruning

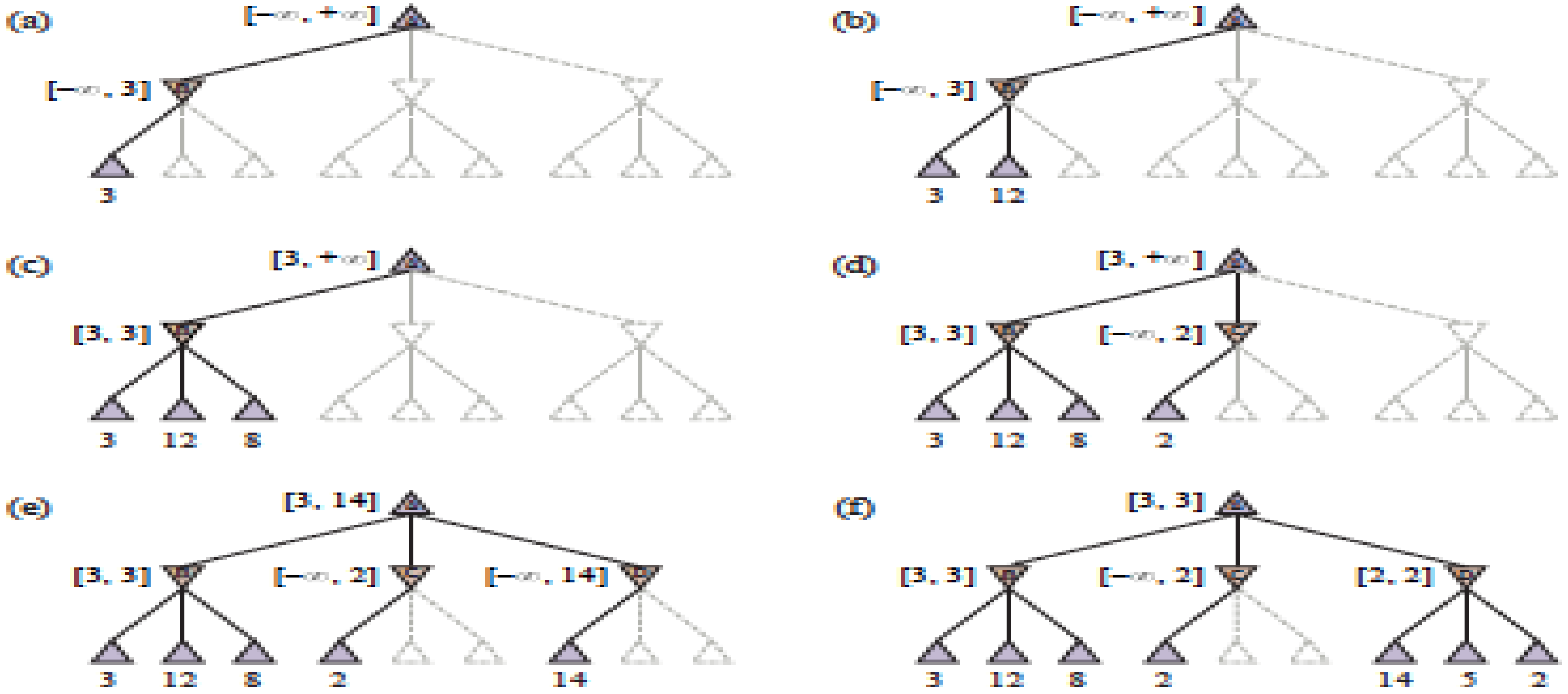


Figure 6.5 Stages in the calculation of the optimal decision for the game tree in Figure 6.2.

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
```

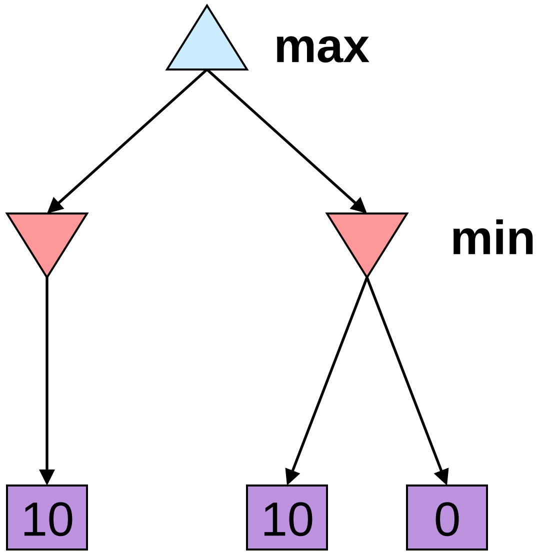
Alpha-Beta Implementation

```
function ALPHA-BETA-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state, −∞, +∞)
    return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
        if v2 > v then
            v, move ← v2, a
            α ← MAX(α, v)
        if v ≥ β then return v, move
    return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
        if v2 < v then
            v, move ← v2, a
            β ← MIN(β, v)
        if v ≤ α then return v, move
    return v, move
```

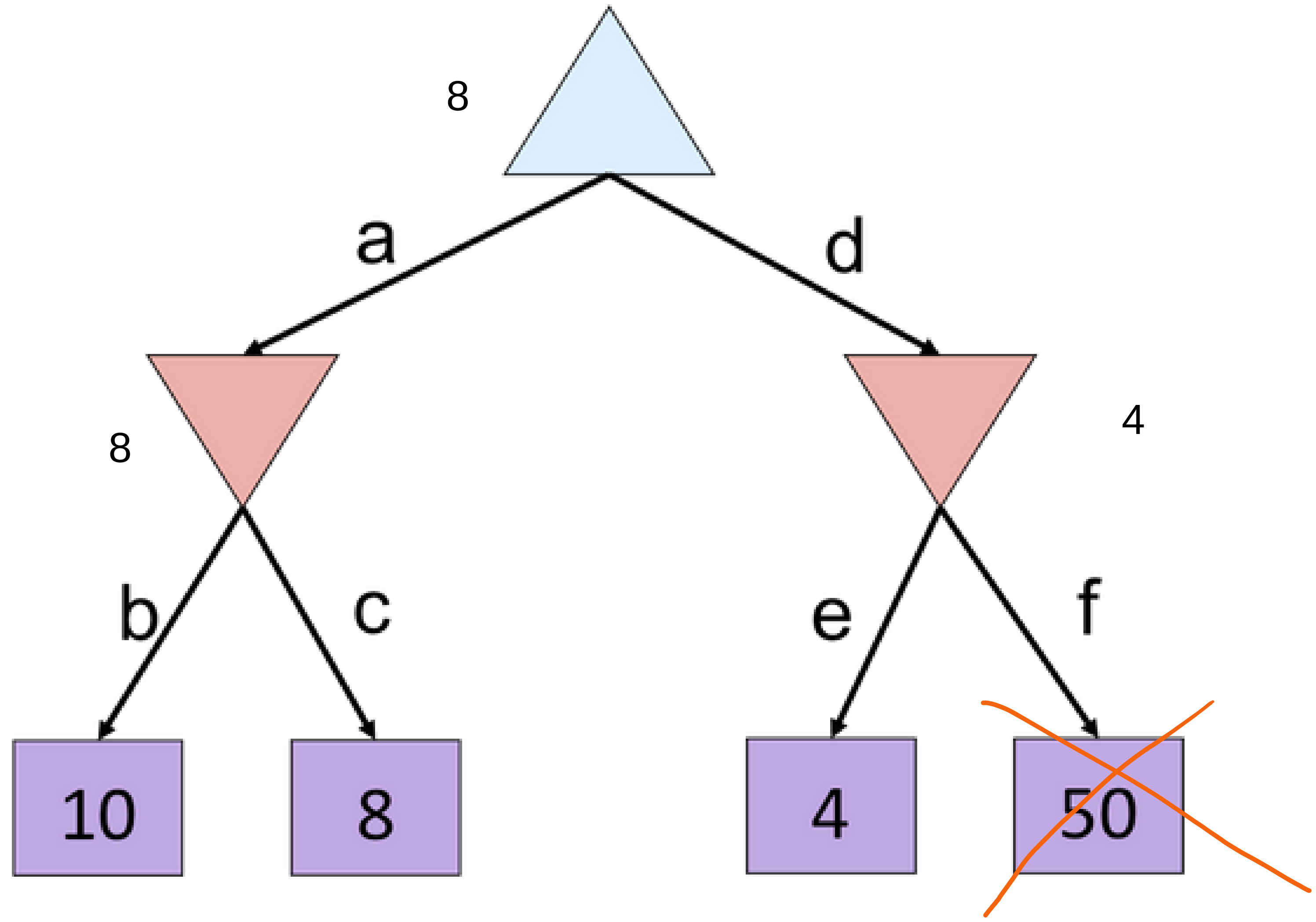
Figure 6.7 The alpha–beta search algorithm. Notice that these functions are the same as the MINIMAX-SEARCH functions in Figure 6.3, except that we maintain bounds in the variables α and β, and use them to cut off search when a value is outside the bounds.

Alpha-Beta Pruning Properties

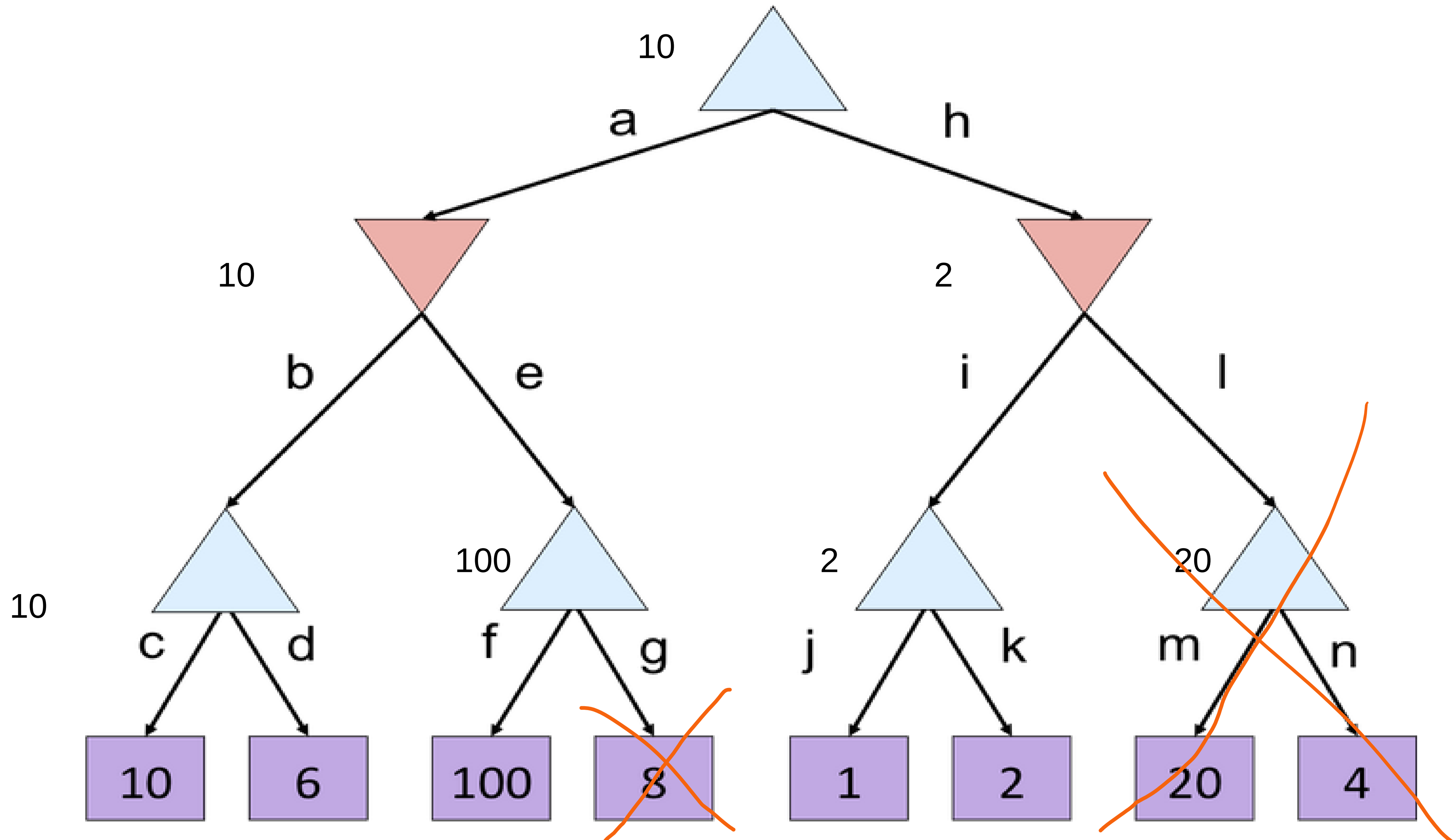
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O((2b)^{m/2})$ or better $O\left(\left(\sqrt{b}+0.5\right)^{m+1}\right)$
  - Doubles solvable depth!
  - Full search of, e.g. Chess, is still hopeless…
- With random ordering:
  - The total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b.
- This is a simple example of meta-reasoning (computing about what to compute)



Alpha-Beta Quiz



Alpha-Beta Quiz 2

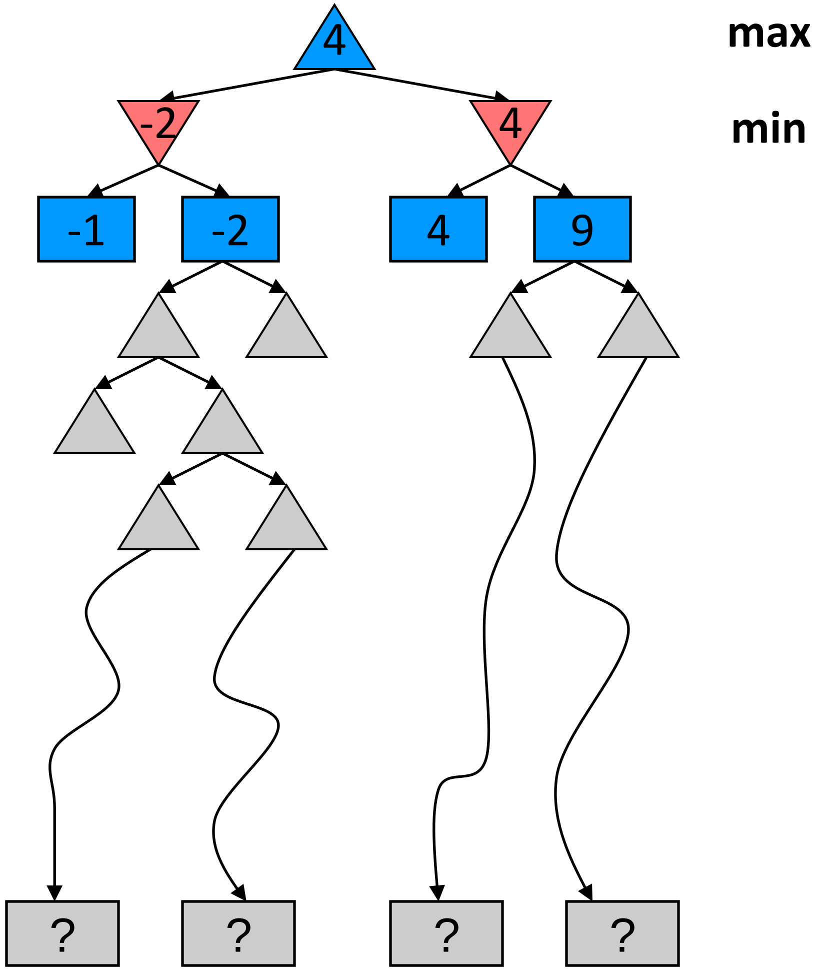


Resource Limits

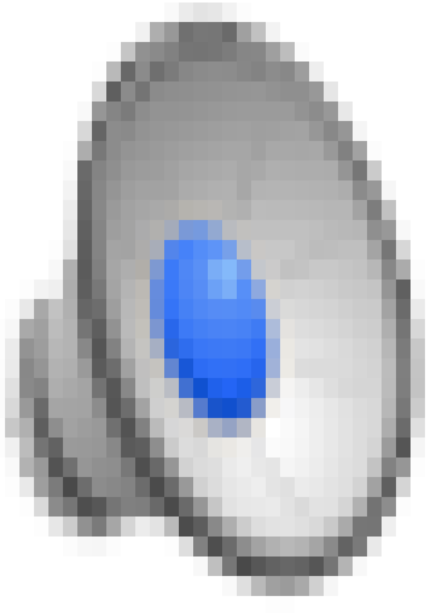
We can not explore the entire game tree because of finite computing power.

Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Another 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
- Another method: Use iterative deepening for an anytime algorithm where you go level by level depeding on your computing power.

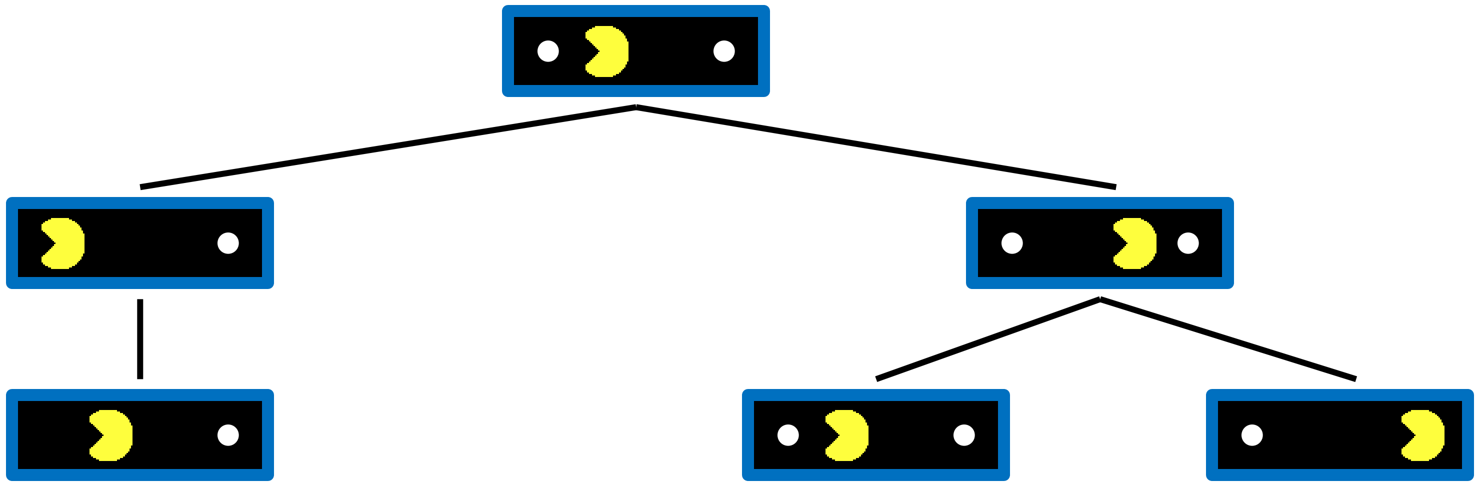


Video of Demo Thrashing (depth d=2)



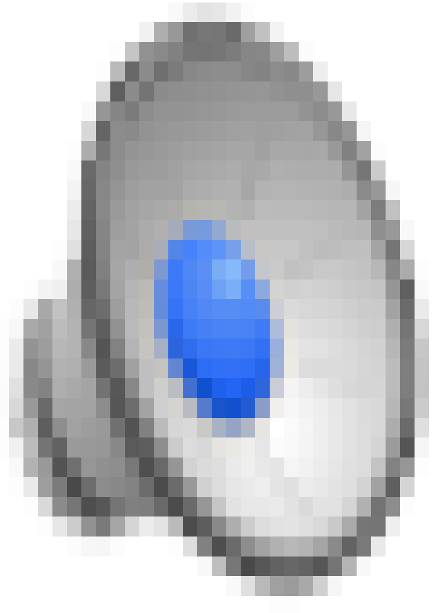
[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function) (L6D6)]

Why Pacman Starves



- A danger of replanning agents!
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

Video of Demo Thrashing -- Fixed (d=2)



[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function) (L6D7)]

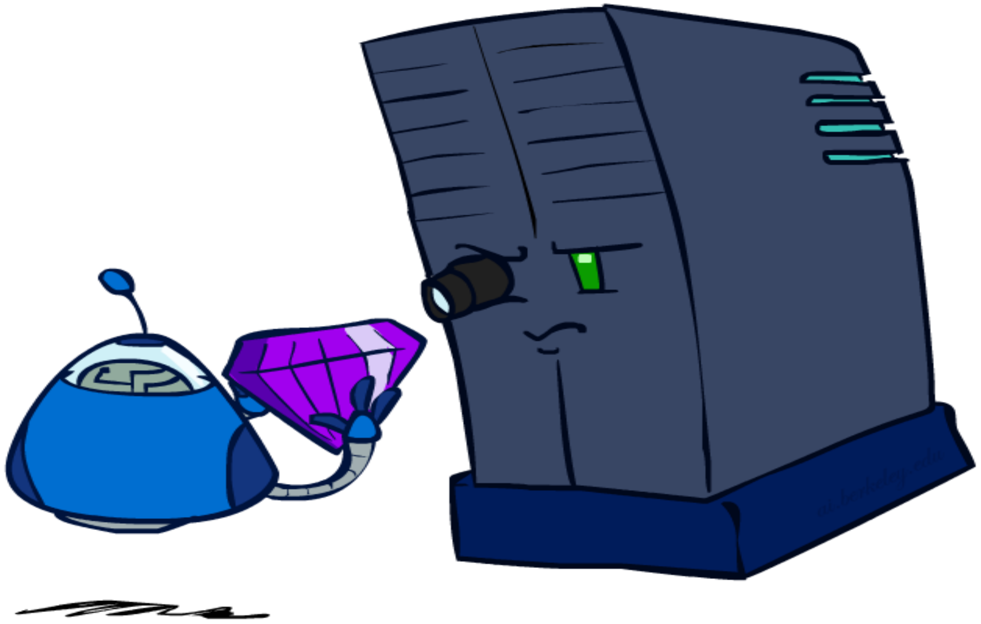
Heuristic Alpha-Beta Tree Search

- *To make use of our limited computation time, we can cut off the search early and apply a heuristic evaluation function to states, effectively treating nonterminal nodes as if they were terminal.*
- *In other words, we replace the UTILITY function with EVAL, which estimates a state's utility.*
- *We also replace the terminal test by a cutoff test, which must return true for terminal states, but is otherwise free to decide when to cut off the search, based on the search depth and any property of the state that it chooses to consider.*
- *That gives us the formula H-MINIMAX(s, d) for the heuristic minimax value of state s at search depth d*

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s, \text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if TO-MOVE}(s) = \text{MIN}. \end{cases}$$

Evaluation Functions

- *A heuristic evaluation function EVAL(s; p) returns an estimate of the expected utility of state s to player p, just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal.*
- *For terminal states, it must be that EVAL(s; p)=UTILITY(s; p) and*
- *for nonterminal states, the evaluation must be somewhere between a loss and a win: UTILITY(loss; p) EVAL(s; p) UTILITY(win; p).*



Evaluation Functions

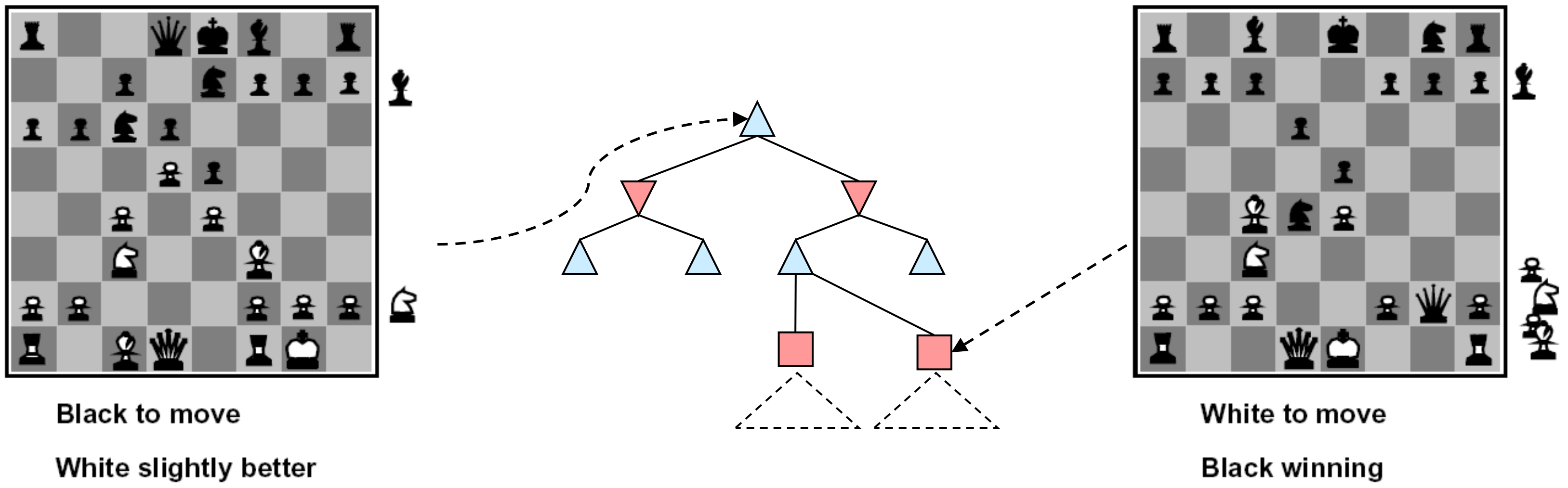
- *Beyond those requirements, what makes for a good evaluation function? First, the computation must not take too long! (The whole point is to search faster.)*
- *Second, the evaluation function should be strongly correlated with the actual chances of winning. One might well wonder about the phrase "chances of winning."*
- *For example,*
  - *chess is not a game of chance: we know the current state with certainty, and no dice are involved; if neither player makes a mistake, the outcome is predetermined.*
  - *But if the search must be cut off at nonterminal states, then the algorithm will necessarily be uncertain about the final outcomes of those states (even though that uncertainty could be resolved with infinite computing resources).*

Evaluation Functions : Features

- *Most evaluation functions work by calculating Features various features of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on.*
- *The features, taken together, define various categories or equivalence classes of states: the states in each category have the same values for all the features.*
- *In principle, the expected value can be determined for each category of states, resulting in an evaluation function that works for any state.*
- *In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities. Instead, most evaluation functions compute separate numerical contributions from each feature and then combine them to find the total value*

Evaluation Functions

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

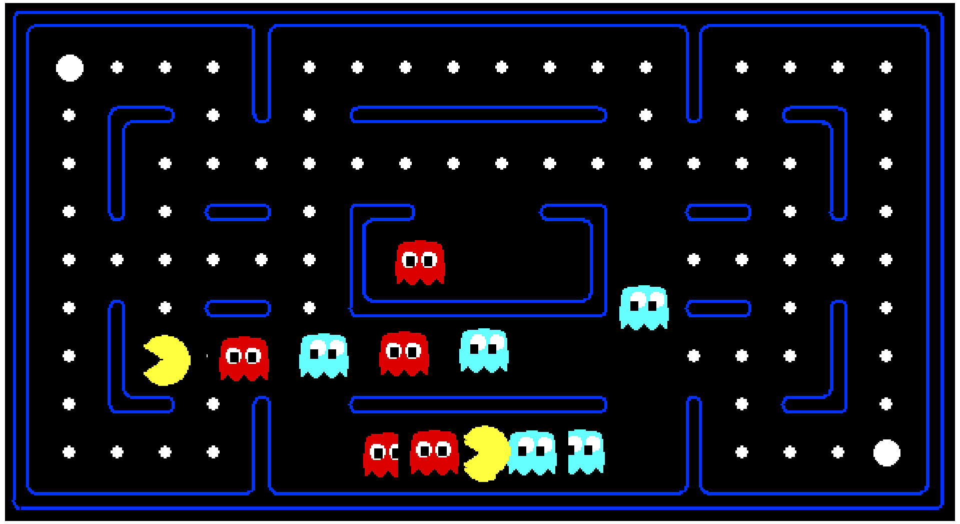


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

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

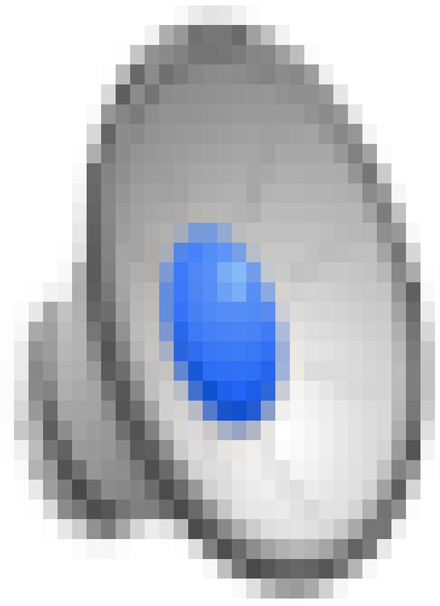
Evaluation for Pacman



- Check evaluation function, change the evaluation or your institution if it does not produce good results.
- Find smart evaluation functions.

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate (L6D6,7,8,10)]

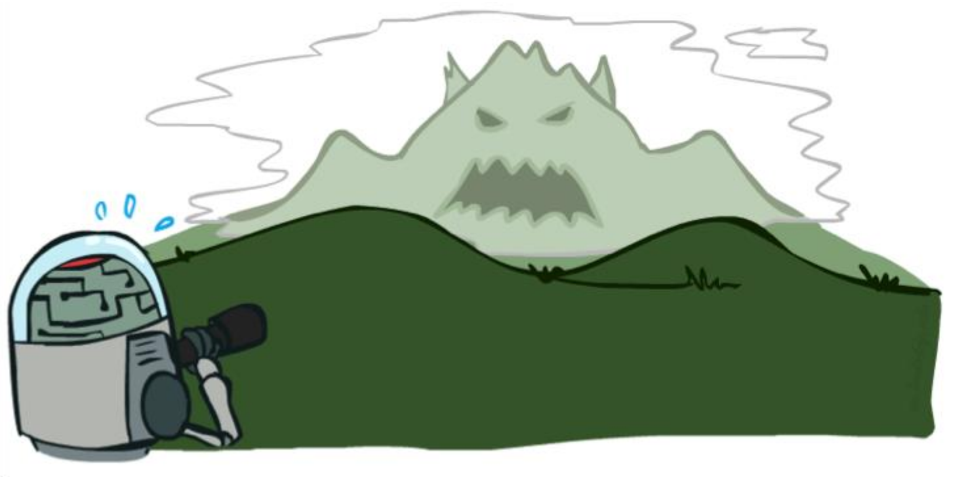
Video of Demo Smart Ghosts (Coordination)



Video of Demo Smart Ghosts (Coordination) – Zoomed In

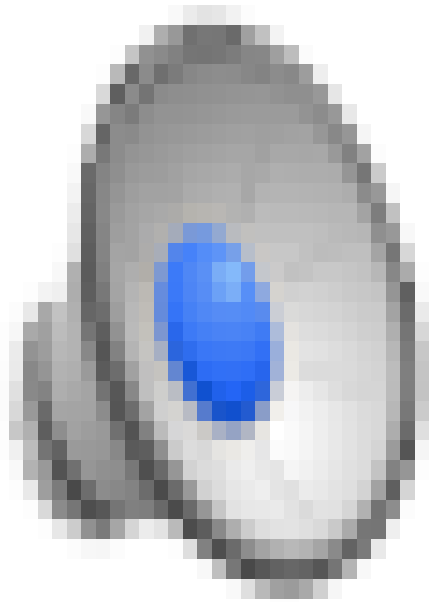
Depth Matters

- 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
- Better evaluation functions requires more time and produces better results by going deeper levels of the search tree
- Demo 1: evaluation function with 2 levels of depth
- Demo 1: evaluation function with 10 levels of depth

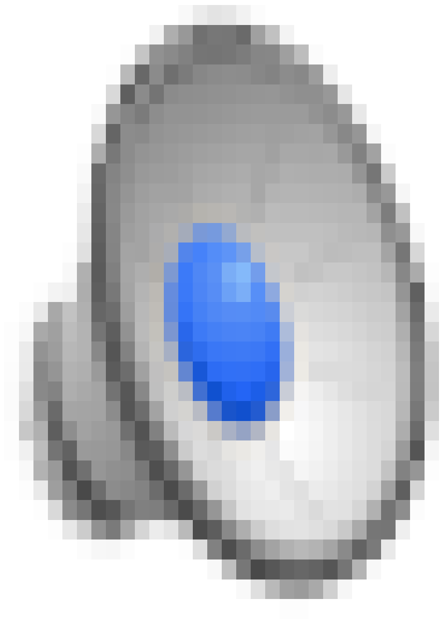


[Demo: depth limited (L6D4, L6D5)]

Video of Demo: Limited Depth (2)



Video of Demo: Limited Depth (10)

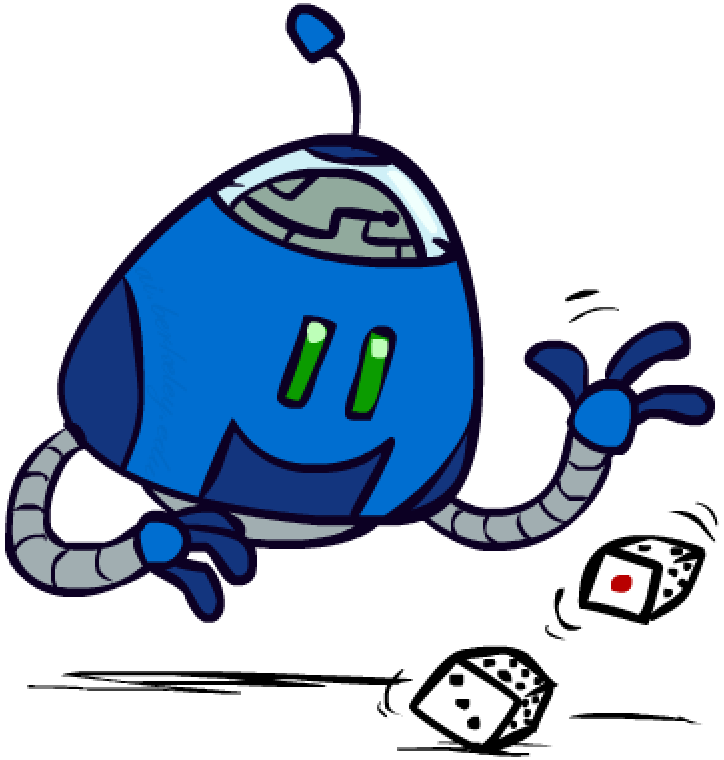


Synergies between Evaluation Function and Alpha-Beta?

- Alpha-Beta: amount of pruning depends on expansion ordering
  - Evaluation function can provide guidance to expand most promising nodes first (which later makes it more likely there is already a good alternative on the path to the root)
    - (somewhat similar to role of A* heuristic, CSPs filtering)

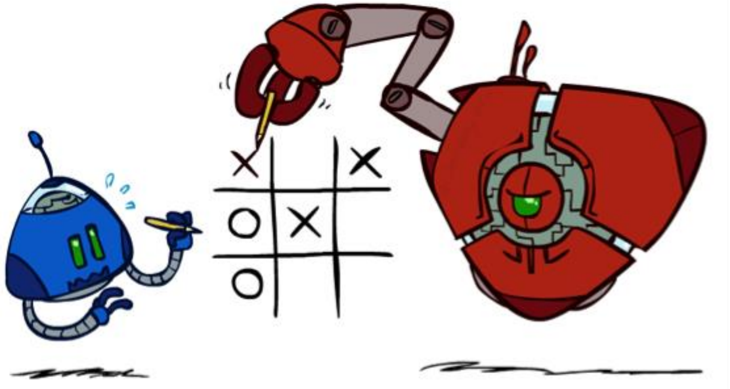
- Alpha-Beta: (similar for roles of min-max swapped)
  - Value at a min-node will only keep going down
  - Once value of min-node lower than better option for max along path to root, can prune
  - Hence: IF evaluation function provides upper-bound on value at min-node, and upper-bound already lower than better option for max along path to root THEN can prune

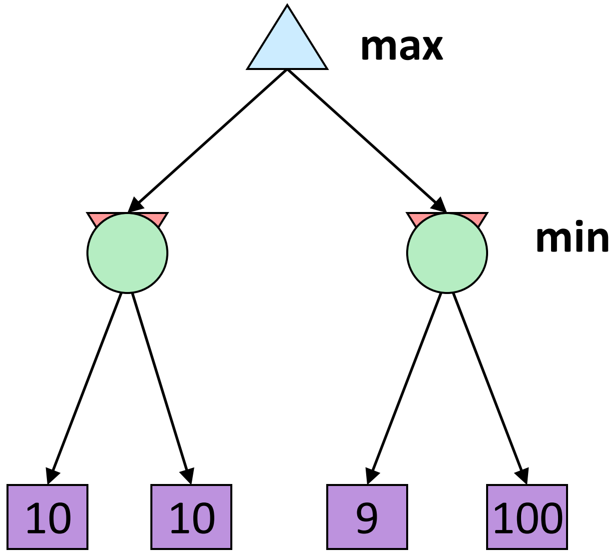
Uncertain Outcomes

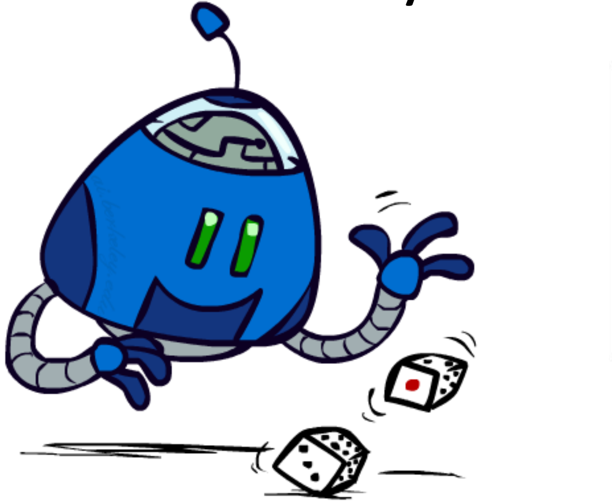


Worst-Case vs. Average Case

Idea: Uncertain outcomes controlled by chance, not an adversary!







- *Stochastic games bring us a little closer to the unpredictability of real life by including a random element, such as the throwing of dice.*
- *Backgammon is a typical stochastic game that combines luck and skill.*

-Agent(s) don't play optimal. They plays stochastically

- Agent may throw a dice for a move
- Agent not so clever

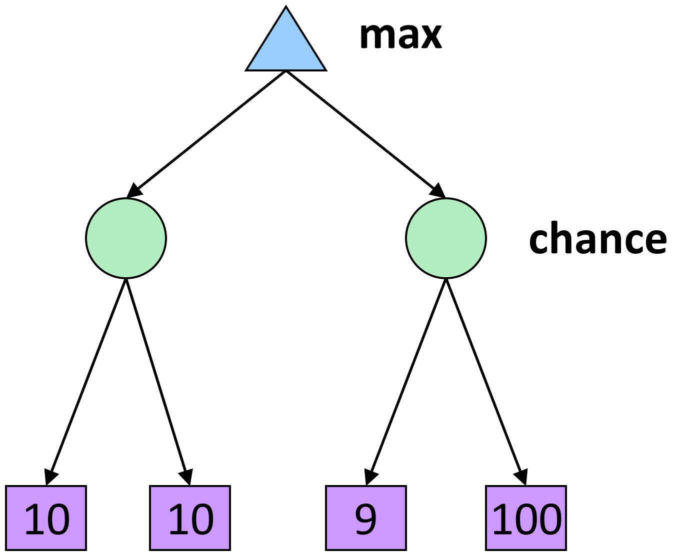
-Add chance nodes (circles)

-Find the weighted average for chance

-*We can calculate the expected value of a position: the average over all Expected value possible outcomes of the chance nodes.*

Expectimax Search

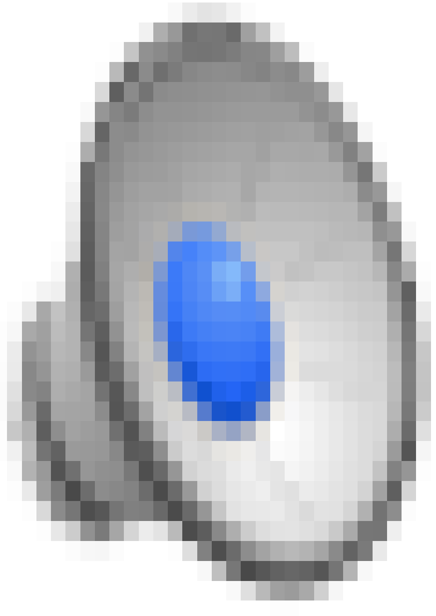
- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) 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
  - I.e. take weighted average (expectation) of children
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



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

Video of Demo Minimax vs Expectimax (Min)

Video of Demo Minimax vs Expectimax (Exp)



Expectimax Pseudocode

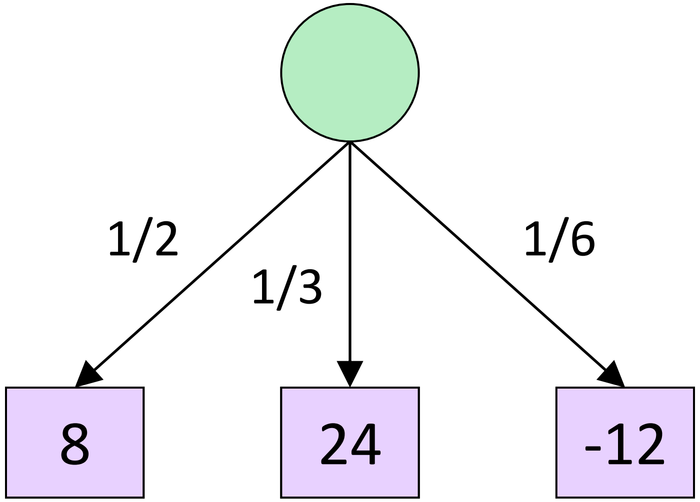
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

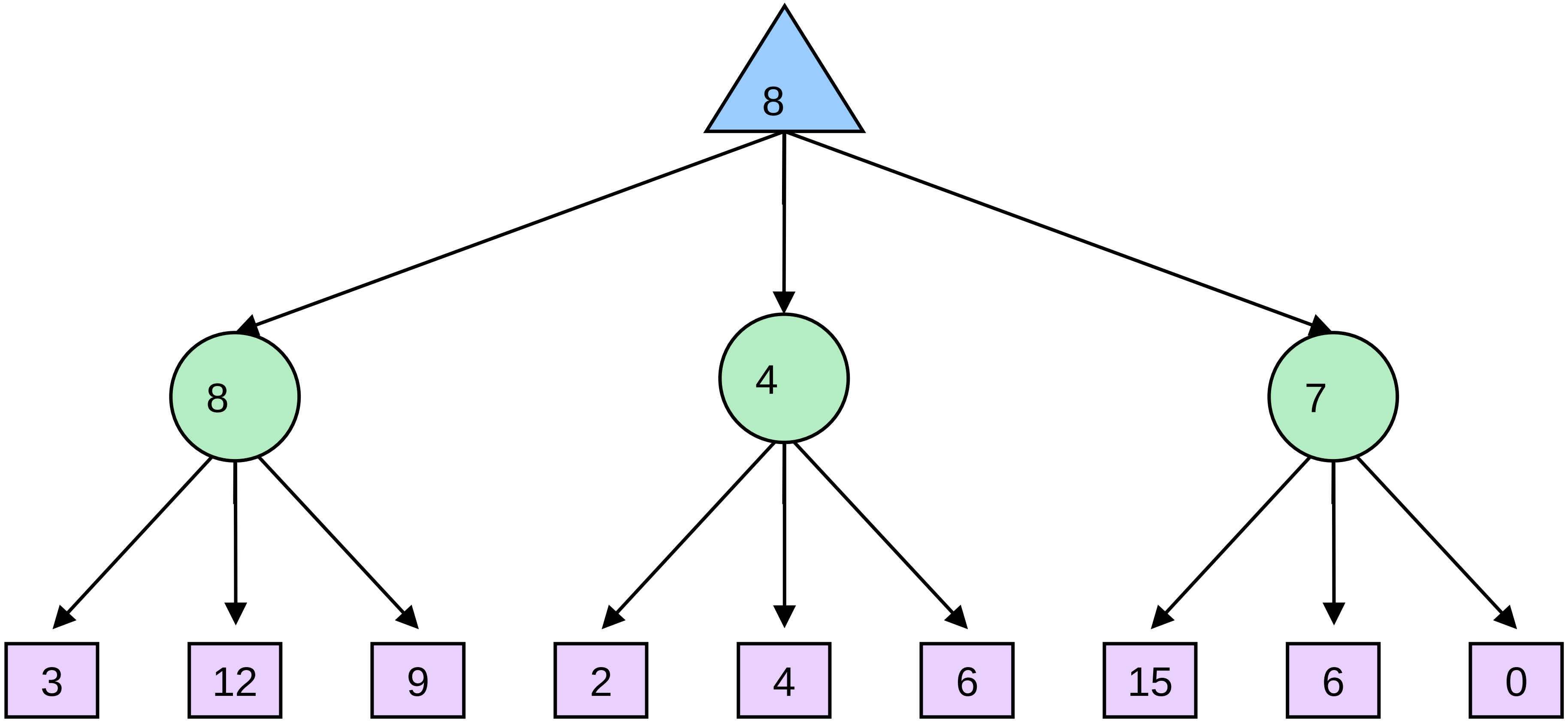
Expectimax Pseudocode

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



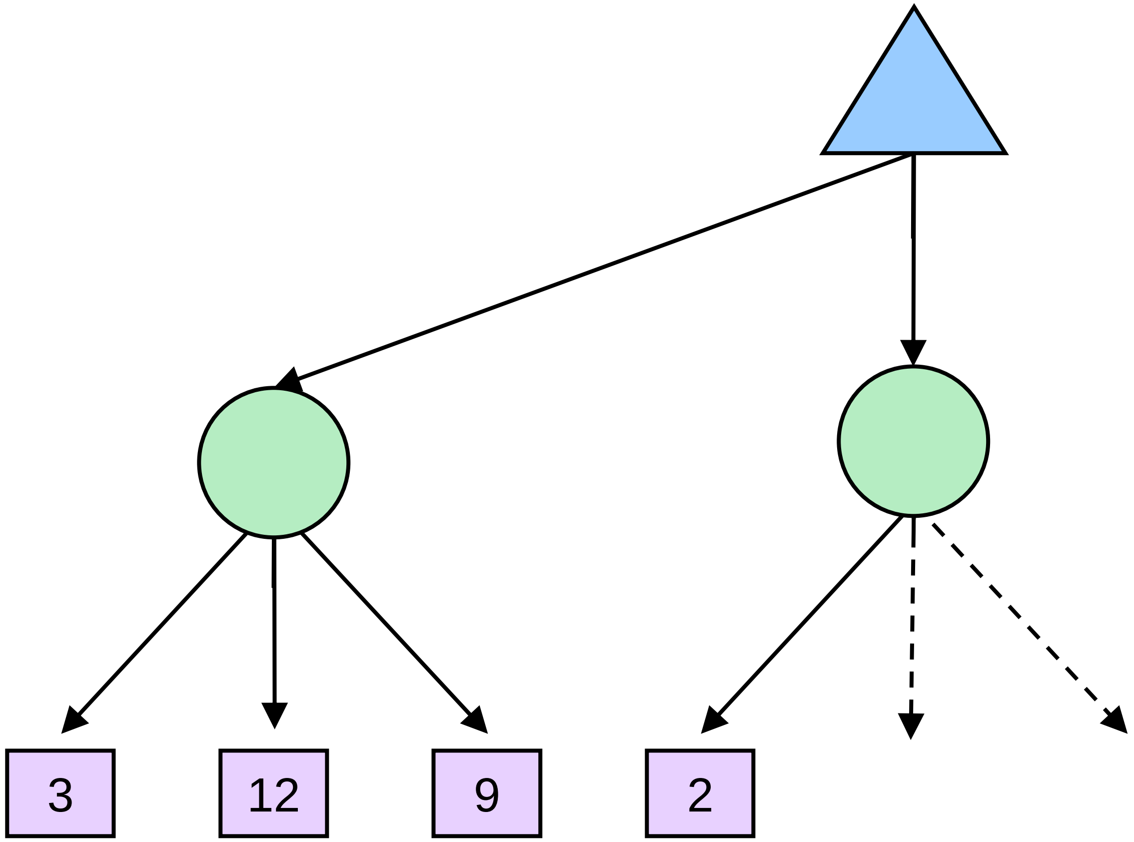
$$v = (1/2)\,(8) + (1/3)\,(24) + (1/6)\,(-12) = 10$$

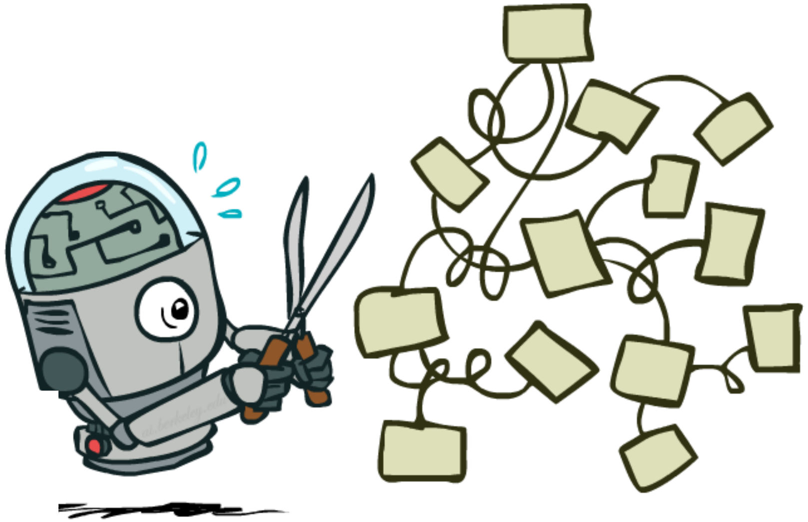
Expectimax Example



Assume that all probablities are equal

Expectimax Pruning?



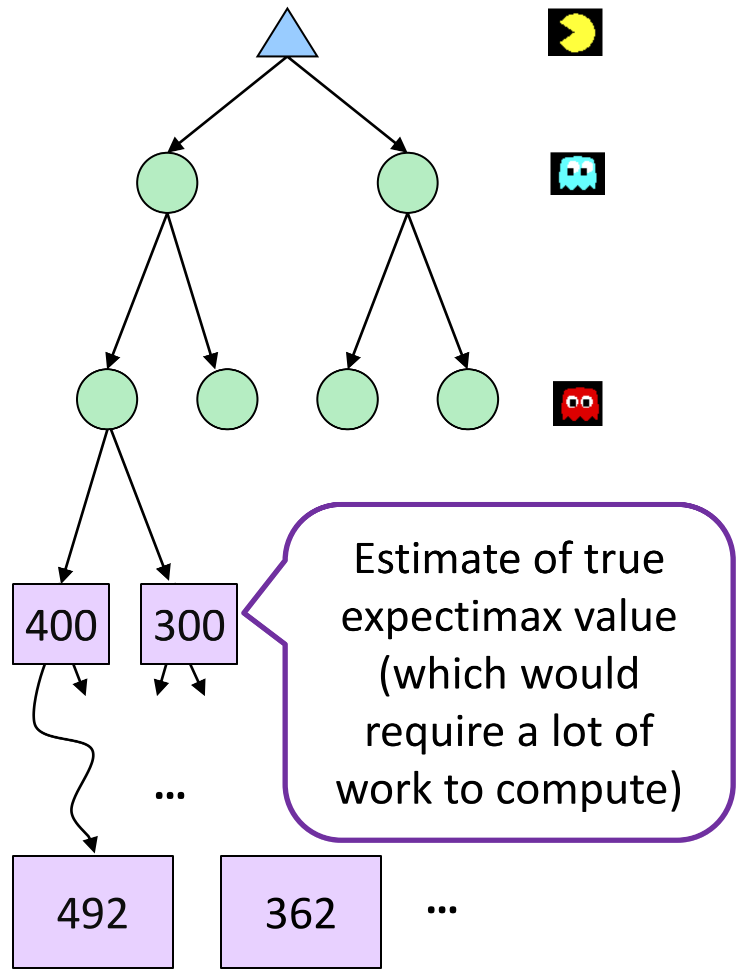


Can we apply pruning ? No. The last value in a node can be very large and change the average.

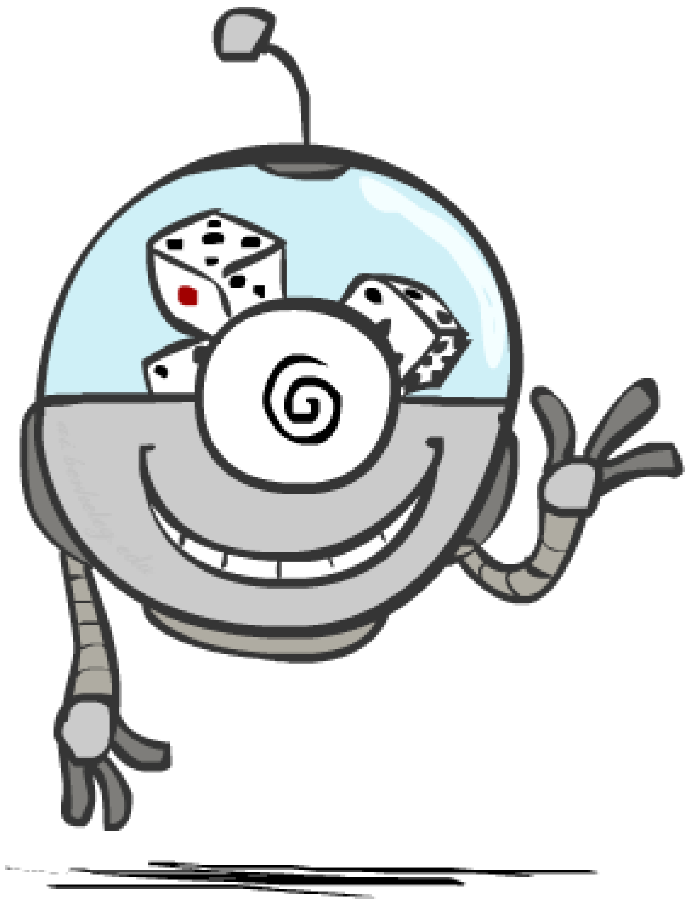
Depth-Limited Expectimax

Stop earlier without going to all the way to the bottom.

You need a good evaluation function that gives you an approximate value (estimate)



Probabilities



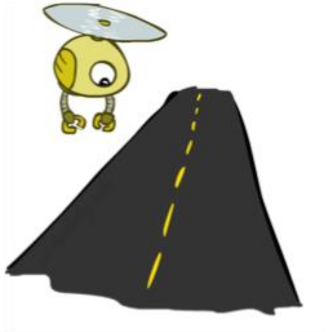
Reminder: Probabilities

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

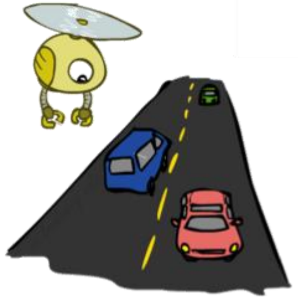
- Example: Traffic on freeway
  - Random variable: T = whether there's traffic
  - Outcomes: T in {none, light, heavy}
  - Distribution: P(T=none) = 0.25, P(T=light) = 0.50, P(T=heavy) = 0.25

- 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



0.25

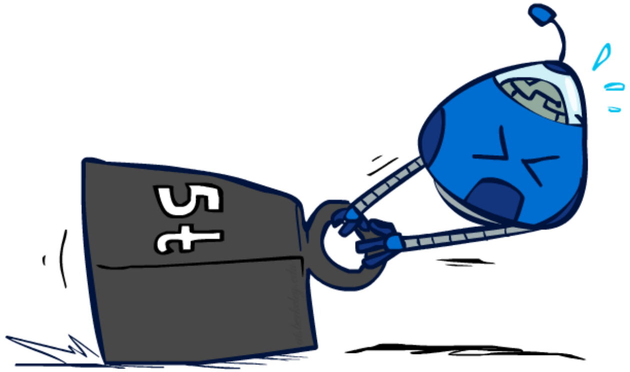


0.50

0.25

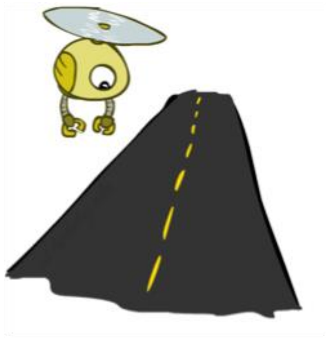
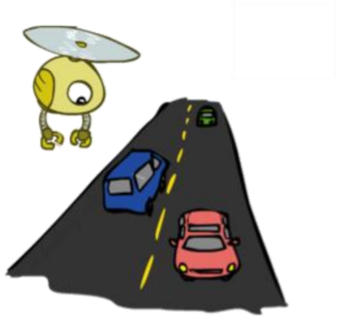
Reminder: Expectations

- The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes



- Example: How long to get to the airport?

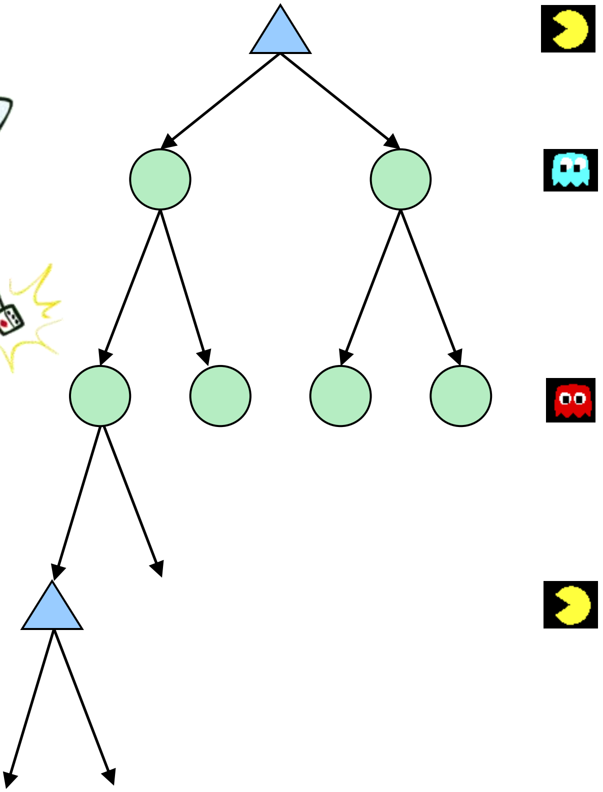
Time:	20 min		30 min		60 min		
	x	+	x	+	x	→	35 min
Probability:	0.25		0.50		0.25		


What Probabilities to Use?

- In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!

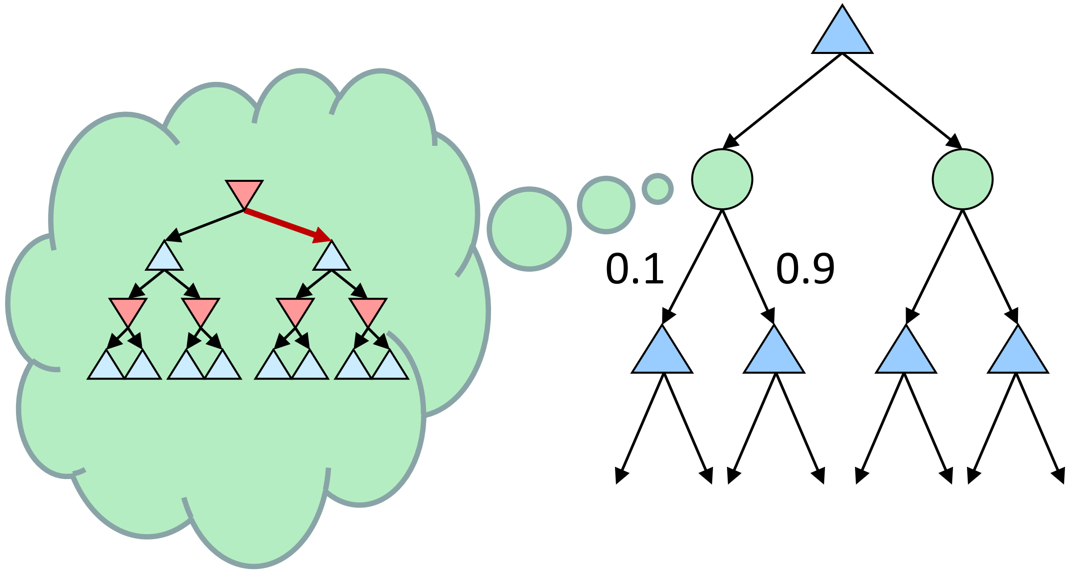
- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!

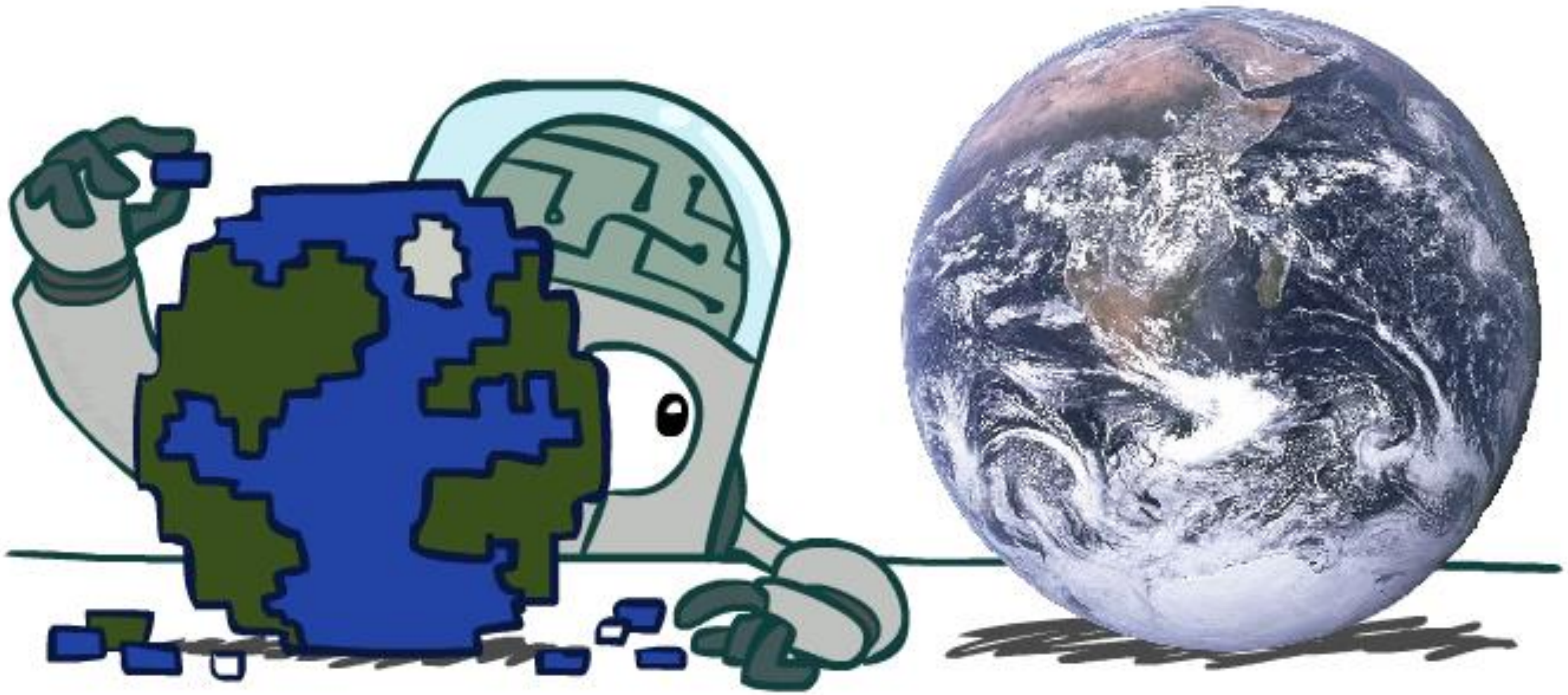
Quiz: Informed Probabilities

- 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…
  - … except for minimax, which has the nice property that it all collapses into one game tree

Modeling Assumptions



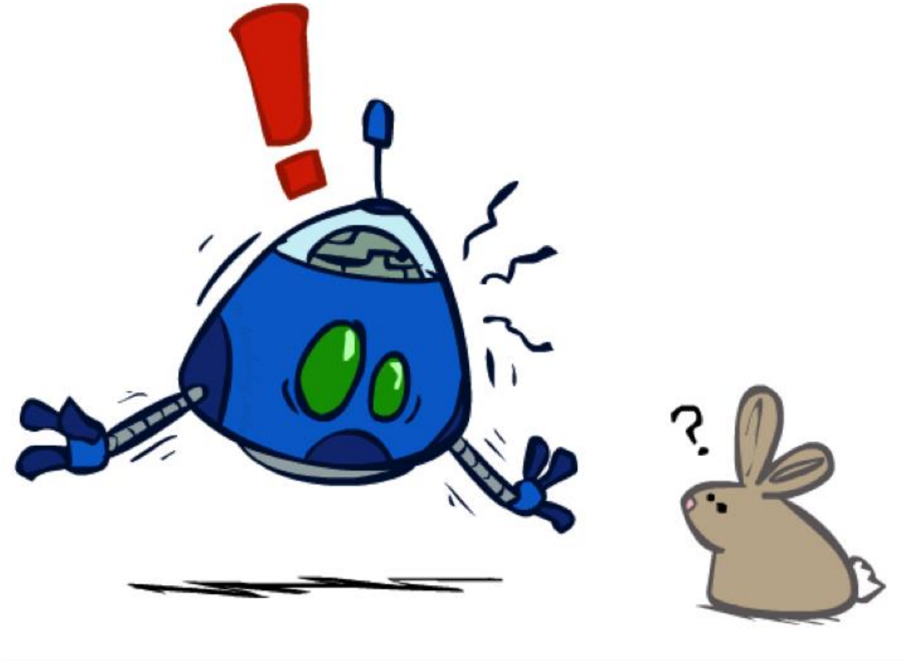
The Dangers of Optimism and Pessimism

Dangerous Optimism

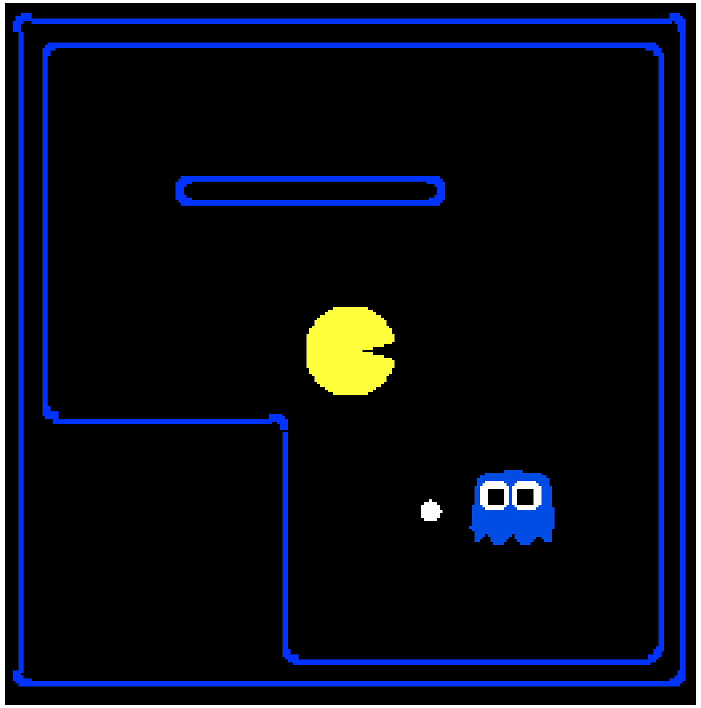
Assuming chance when the world is adversarial

Dangerous Pessimism

Assuming the worst case when it’s not likely



Assumptions vs. Reality



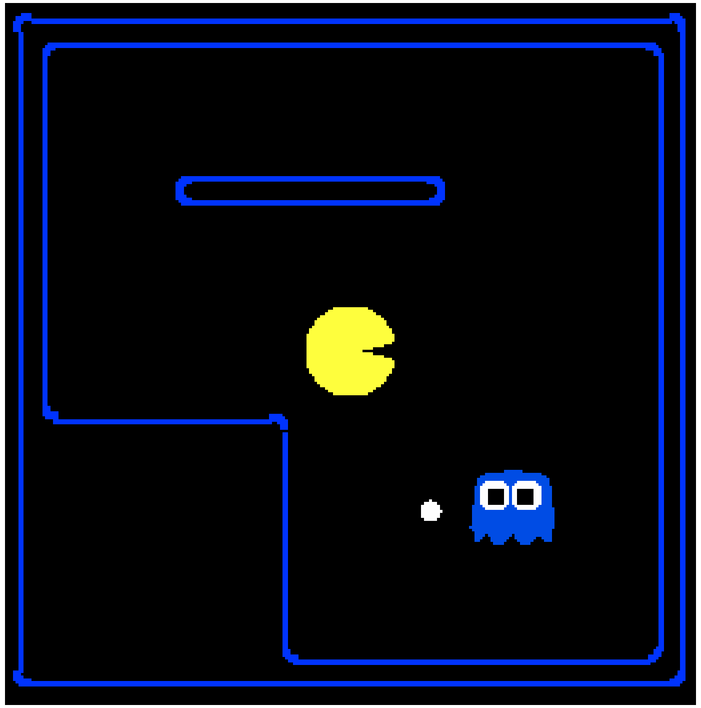
	Adversarial Ghost	Random Ghost
Minimax Pacman		
Expectimax Pacman		

Results from playing 5 games

Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

[Demos: world assumptions (L7D3,4,5,6)]

Assumptions vs. Reality



	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

Results from playing 5 games

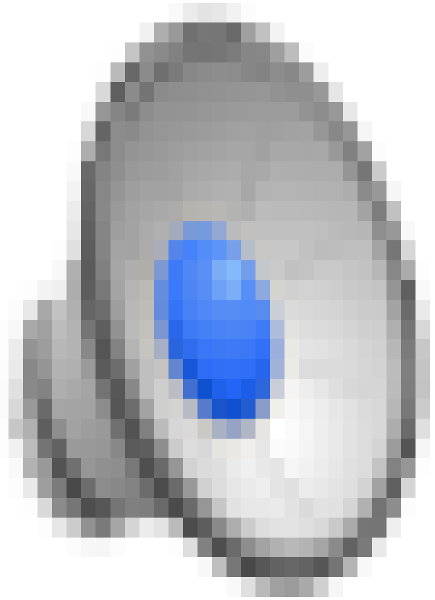
Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

[Demos: world assumptions (L7D3,4,5,6)]

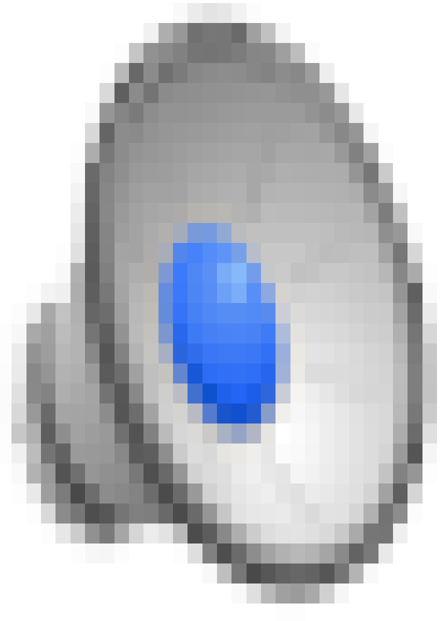
Video of Demo World Assumptions
Random Ghost – Expectimax Pacman

Video of Demo World Assumptions
Adversarial Ghost – Minimax Pacman

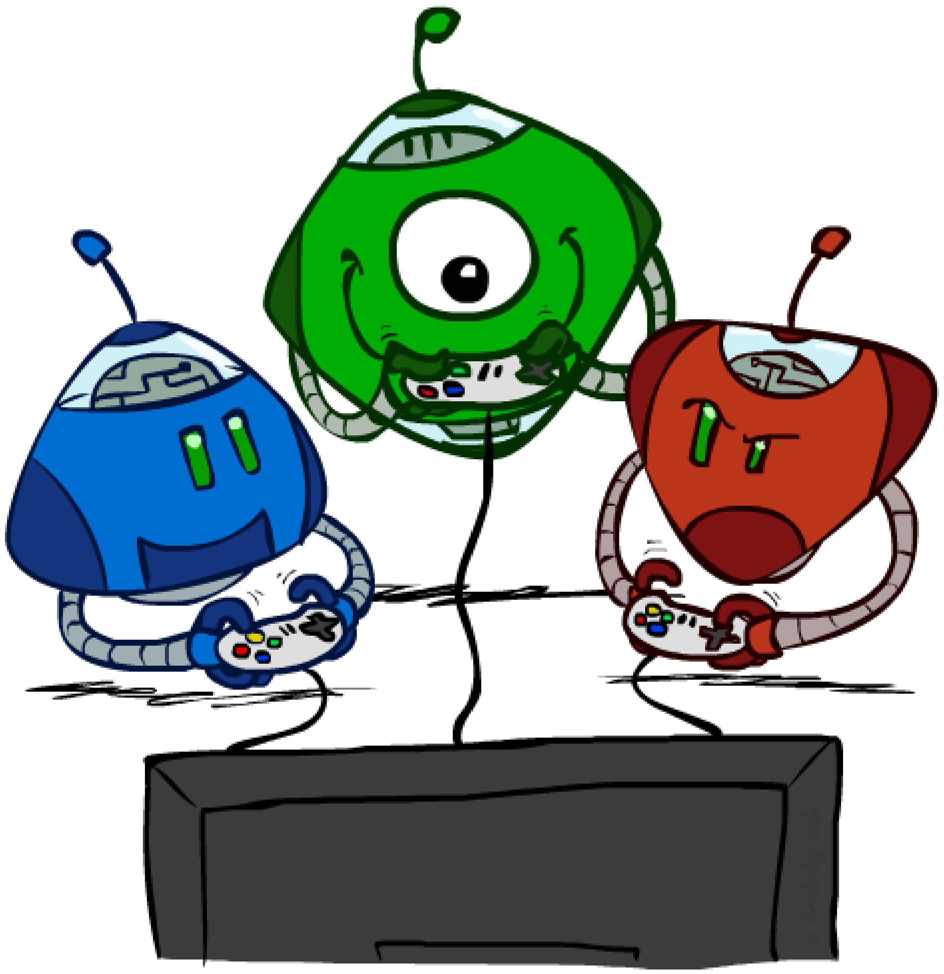
Video of Demo World Assumptions
Adversarial Ghost – Expectimax Pacman



Video of Demo World Assumptions
Random Ghost – Minimax Pacman

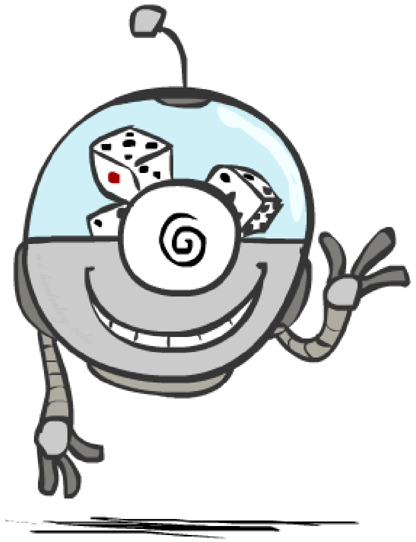


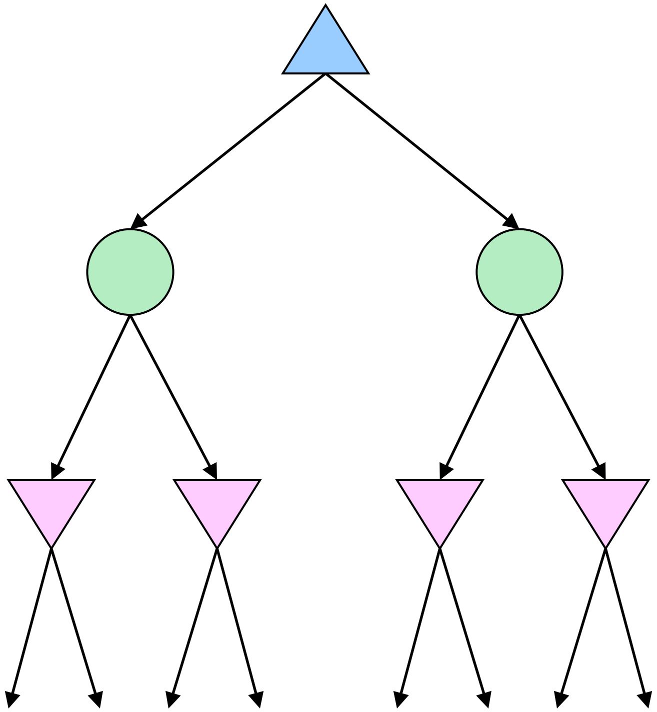
Other Game Types



Mixed Layer Types

- E.g. Backgammon
- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children



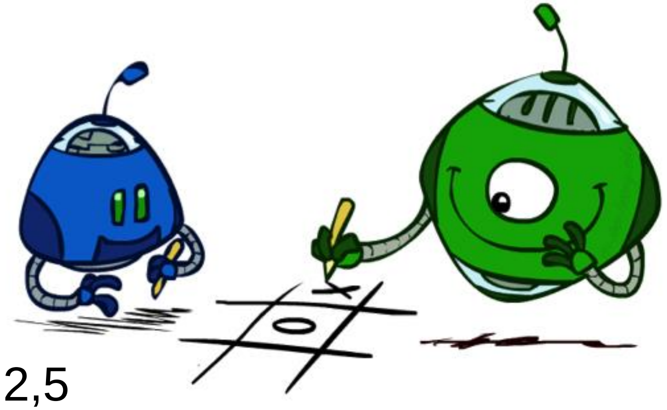


Example: Backgammon

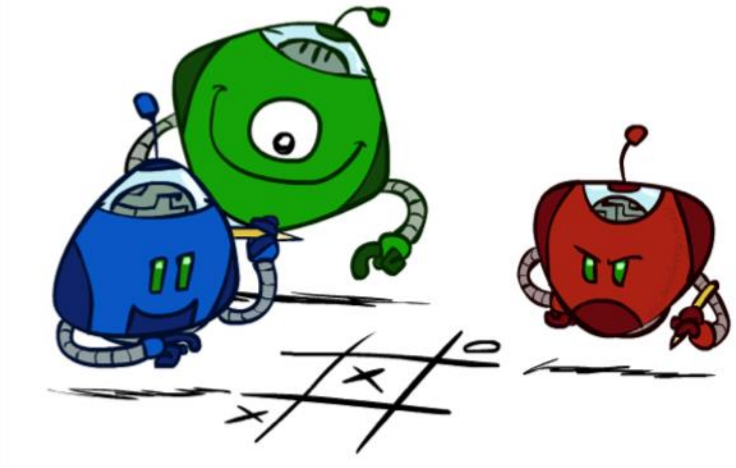
- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon ≈ 20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier…
- Historic AI: TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play
- 1st AI world champion in any game!

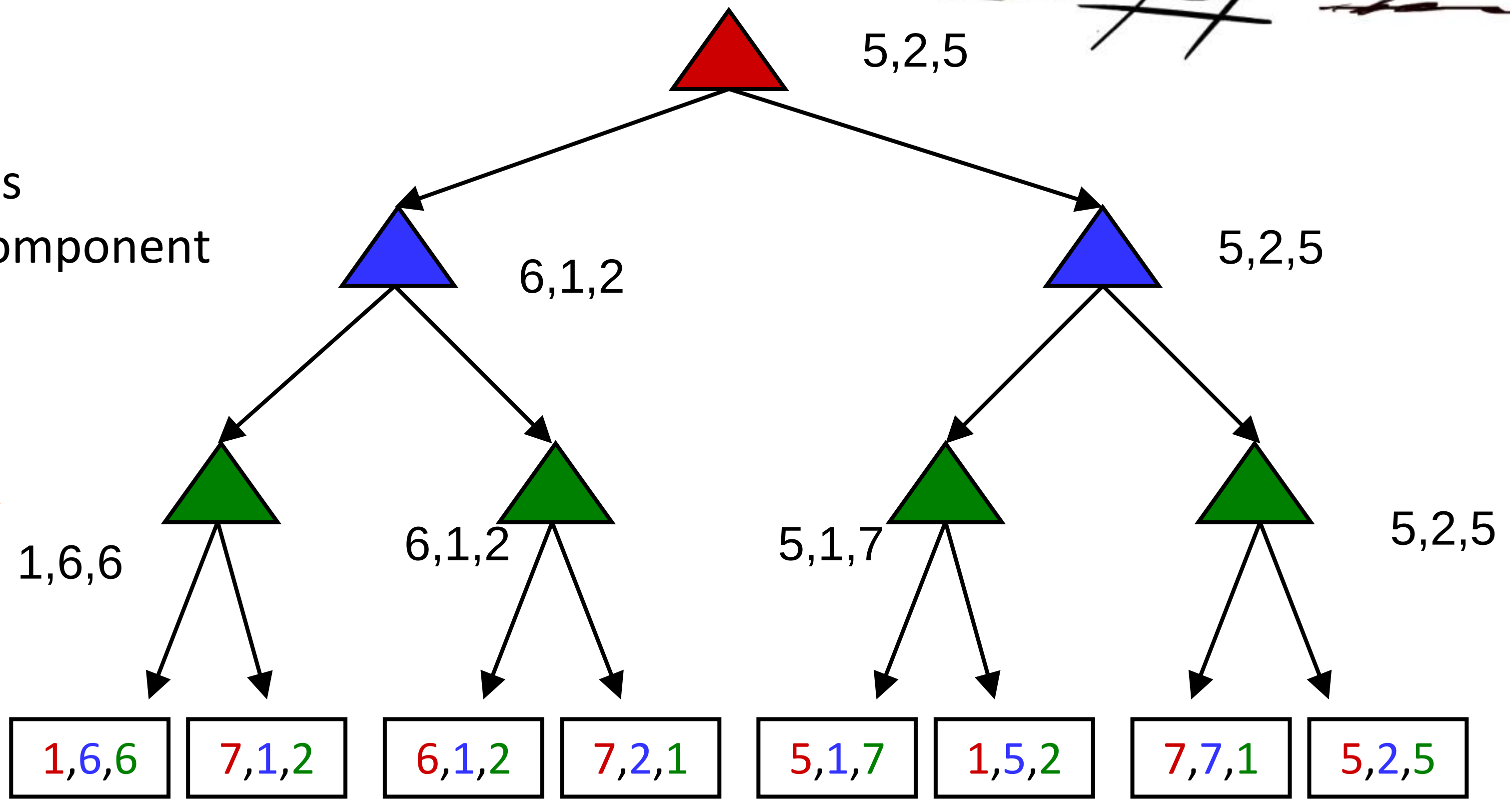
Image: Wikipedia

Multi-Agent Utilities



- 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…





in a three-player game with players A, B, and C, a vector (v_A;v_B;v_C)is associated with each node.

Multi-Agent Utilities

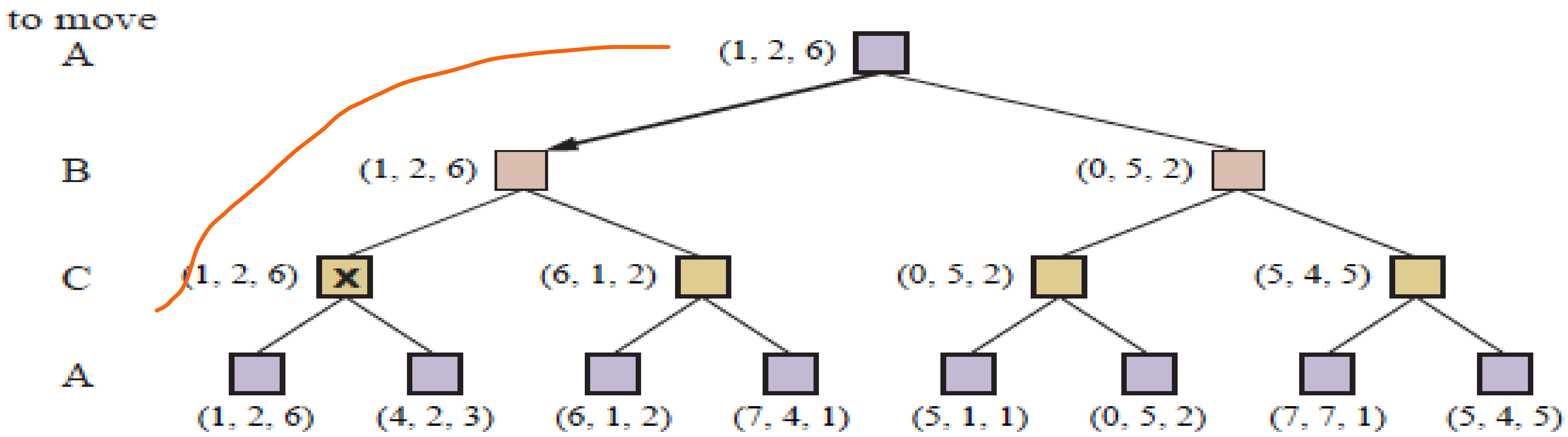


Figure 6.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure 6.4. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Hence, the backed-up value of X is this vector. In general, the backed-up value of a node n is the utility vector of the successor state with the highest value for the player choosing at n.