Other Search Models

CPSC 433: Artificial Intelligence
Fall 2024

Jonathan Hudson, Ph.D.
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

August 8, 2024

UNIVERSITY OF CALGARY

Other Search Models and Processes

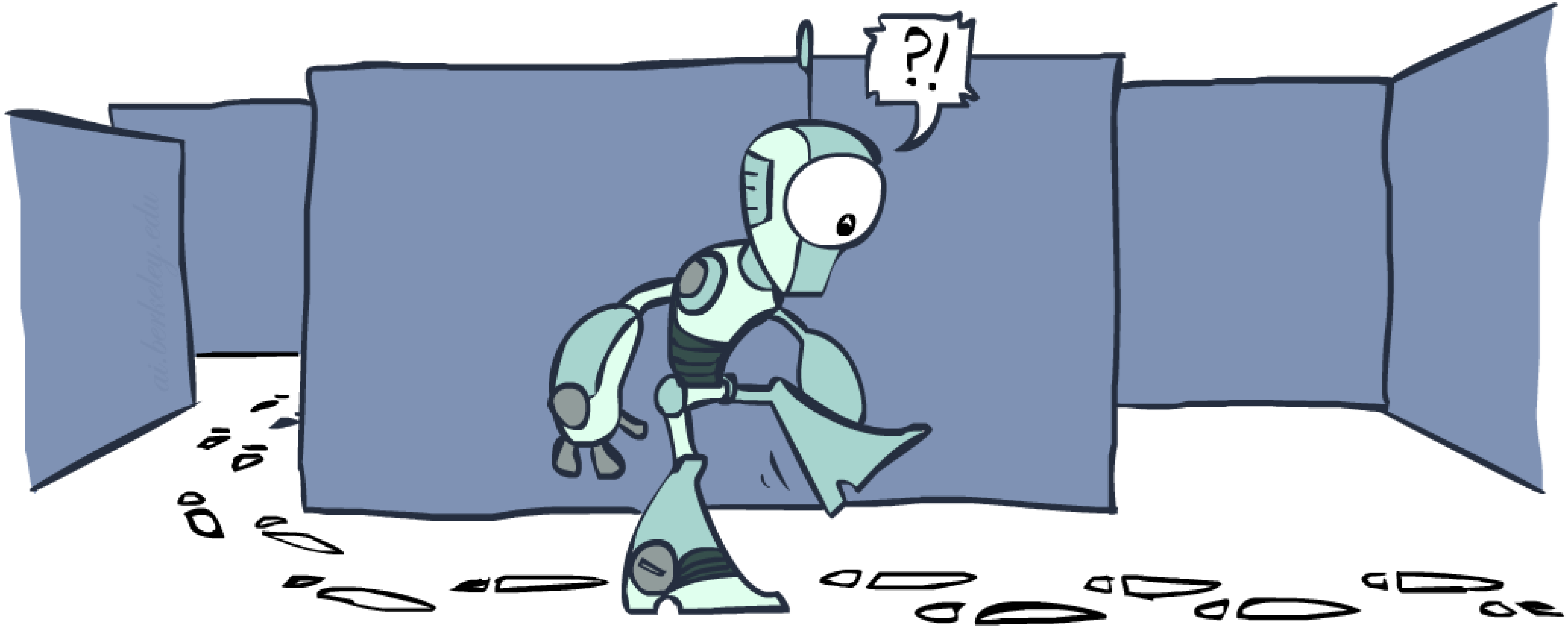
- Problems with the models/processes so far:

1. What about elements of `Prob` that appear repeatedly in a tree? Can we get rid of duplication and resulting redundancy?
   - Graph-based search!

2. What if our problem solution requires alternatives of problem divisions and we want this represented in the model?
   - And-or-tree-based search!

Graph-Based Search

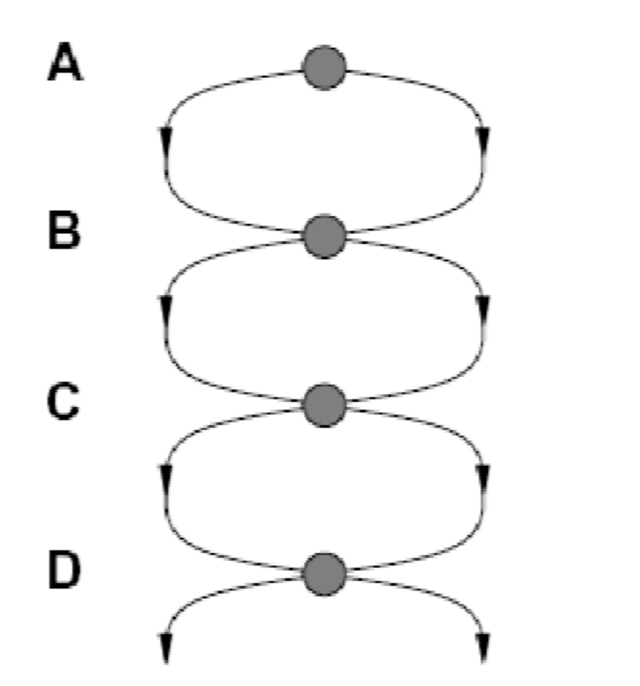
Graph Search



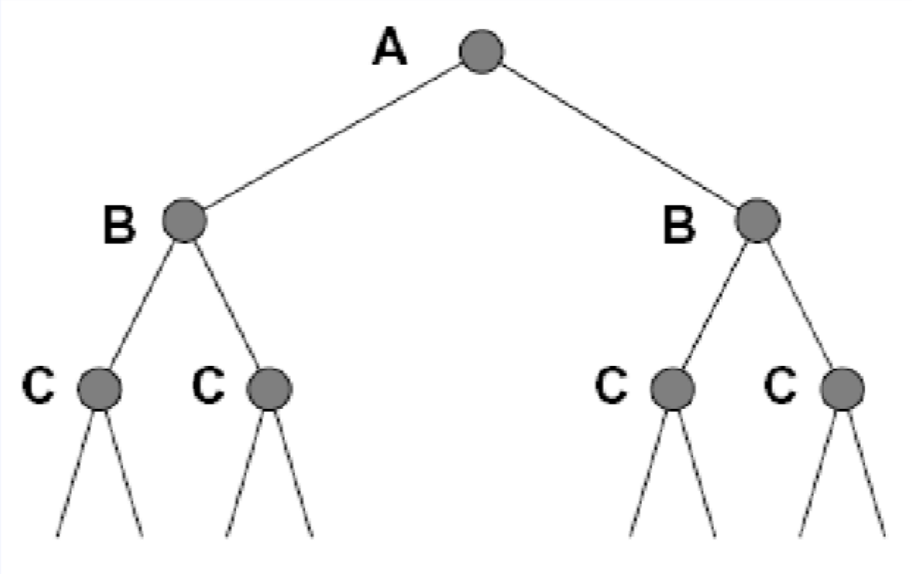
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.

State Graph



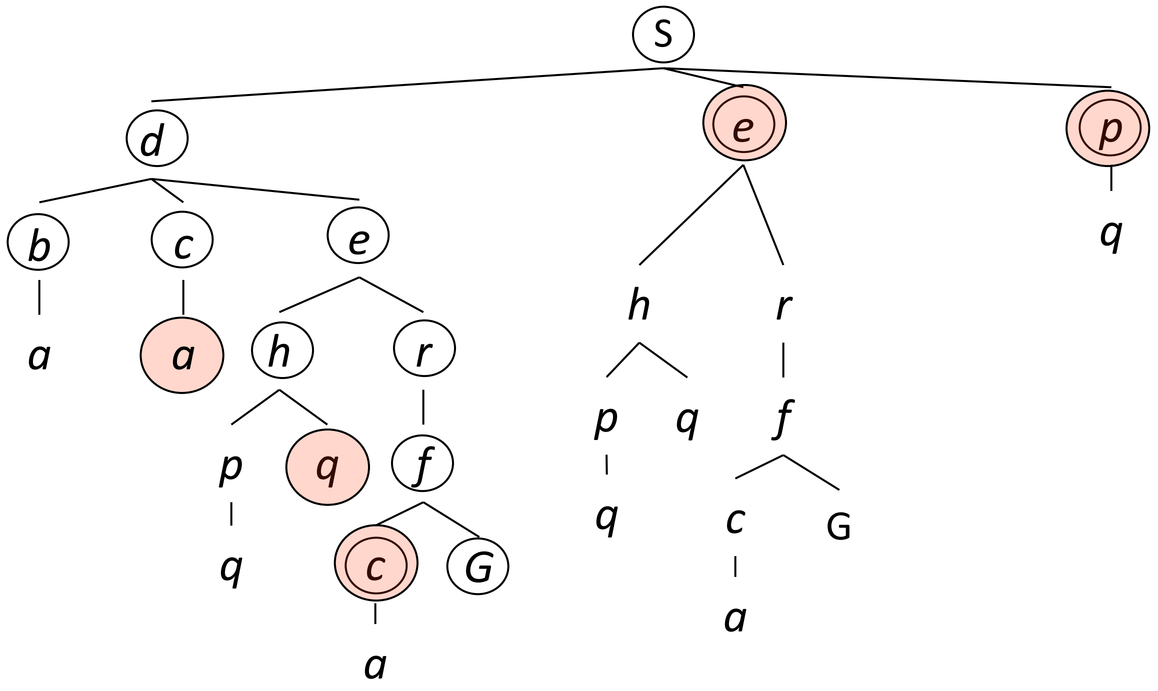
Search Tree



UNIVERSITY OF CALGARY

Graph Search

- In BFS, for example, we shouldn't bother expanding the pink circled nodes (why?)



Graph Search

- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: store the closed set as a set, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?

Graph-based search

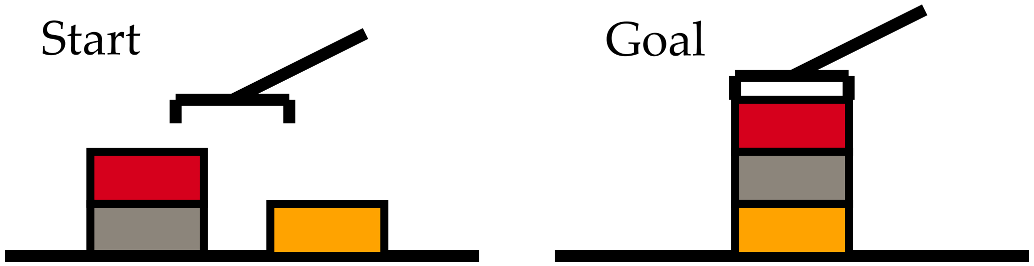
- "Improvement" of tree-based search
- Achieves that every element of `Prob` occurs in only one node
- Graph described as set of nodes with set of arcs (directed connections)
- Transitions extend nodes that have no arcs going out
- Transitions as in trees, except that we check if a certain node is already there and if yes, we do not create it again, we just add an arc

Graph-Based-search: Blocks World

Example (Sketch): Planning in the Blocks World (I)

Problem: Given a set of blocks in certain relations to each other (situation), the same blocks in a different relation and a robot arm, determine a set of actions of the arm that transform the first situation into the second one

Idea: Do an or-tree-based search using the different possible actions of the robot arm as the alternatives



Example (Sketch): Planning in the Blocks World (II)

Observation: A lot of different action combinations lead to the same result (since for each action there is an action with exactly the opposite effect)

☞a lot of problem descriptions occur in several nodes in the or-tree

☞Switching from a tree to a graph avoids redundant work and takes a lot of pressure from the search control

Pros and Cons

- Less memory consumption
- No redundant work

\- Some help (☞ overhead) necessary to quickly detect already represented elements of `Prob`

\- Graphs are more difficult to debug

☞Use only, if quite some duplication of nodes occurs in a tree

And-Or-Tree-based Search

And-or-tree-based search

- Combines and- and or-trees to represent all alternative divisions of problems in the current state
- Formal description very complex, especially end condition:
  For each collection of alternatives, one division (i.e. one alternative) has to be solved by compatible subsolutions (☞ recursive definition)
- And-or-transitions: Extend leaf by adding nodes representing alternative lists of nodes representing a division of the leaf's problem

Multiplayer Games

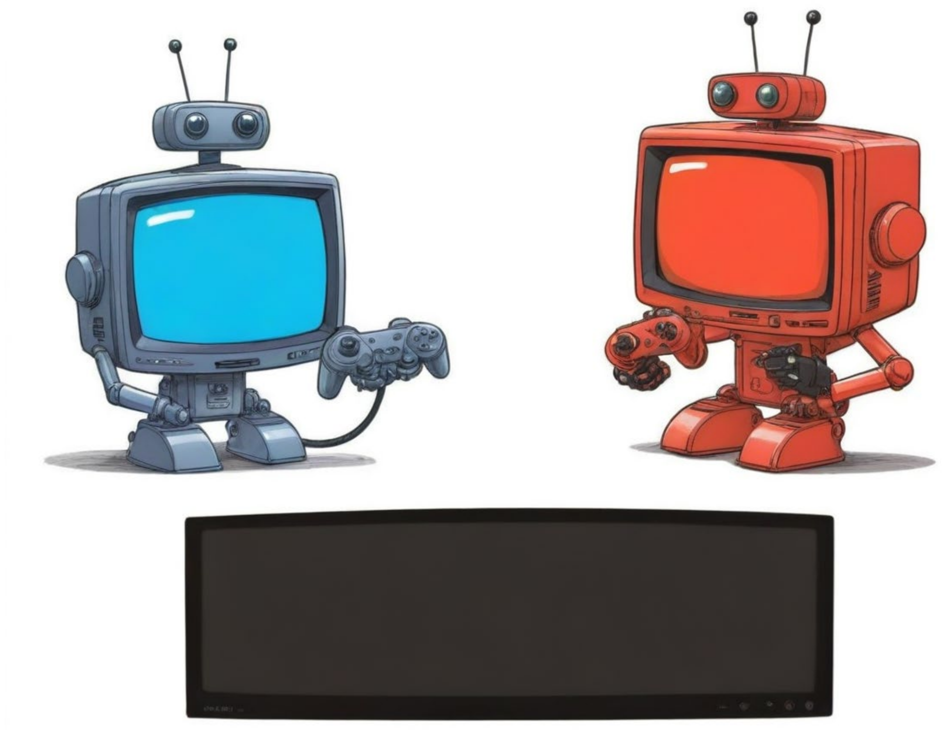
Playing multi-player games

Problem: How to determine the best next move in a game like chess or checkers given a limited amount of computing time.

Idea: Search among the possible alternative moves and their consequences

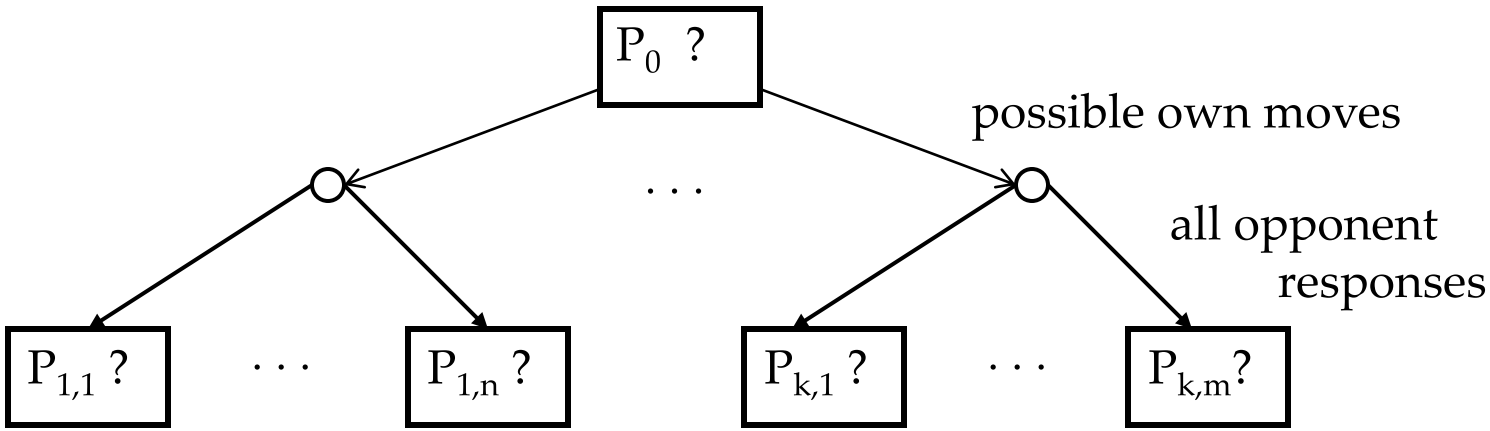
☞ **use and-or-trees to represent search state**

- and-part: select one of your possible moves
- or-part: considering all possible counter-moves of your opponent(s)
- problems: game situations



UNIVERSITY OF CALGARY

Playing multi-player games (II)



Problem: usually we can not search all branches until game is decided

☞ search control decides based on current situation where to go deeper

New: Cost -> Utility!

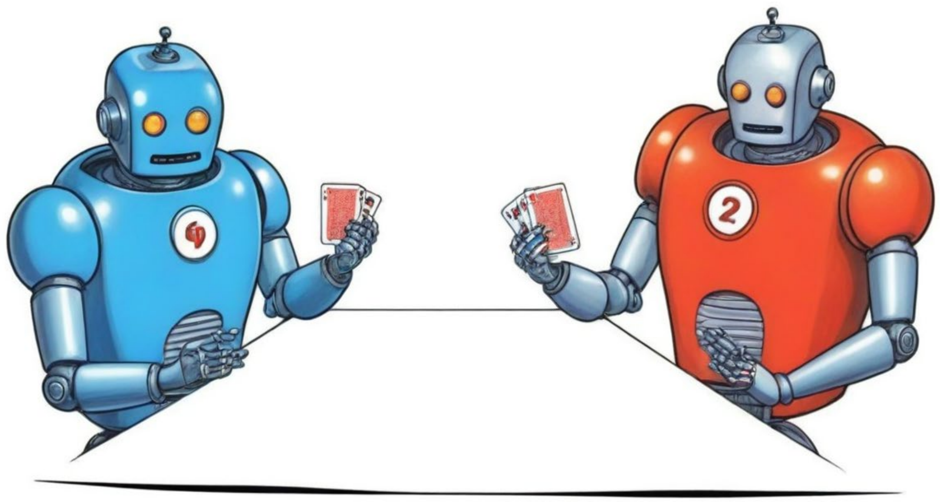
- no longer minimizing cost!
- agent now wants to maximize its score/utility!

Types of Games

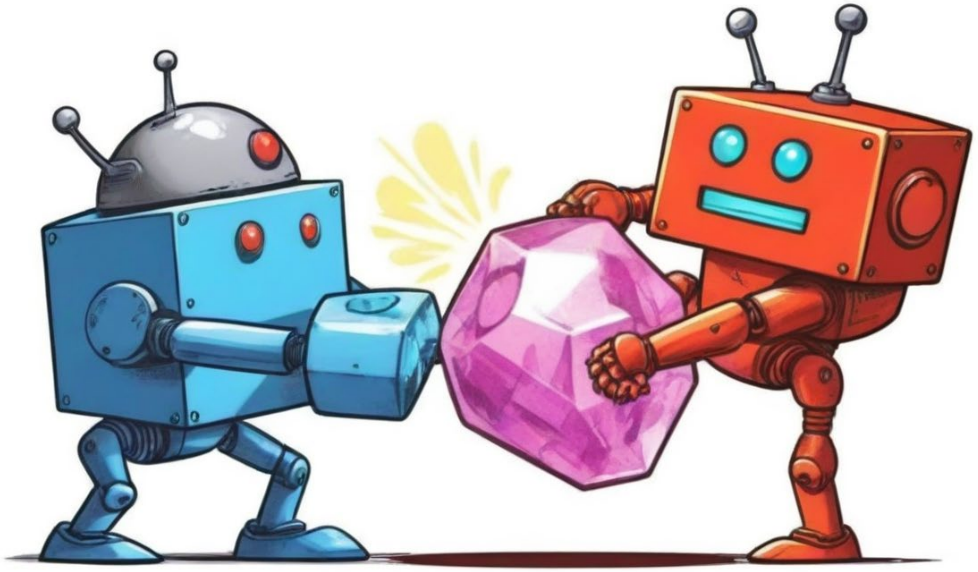
Types of Games

- Many different kinds of games!

- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?



Types of Games



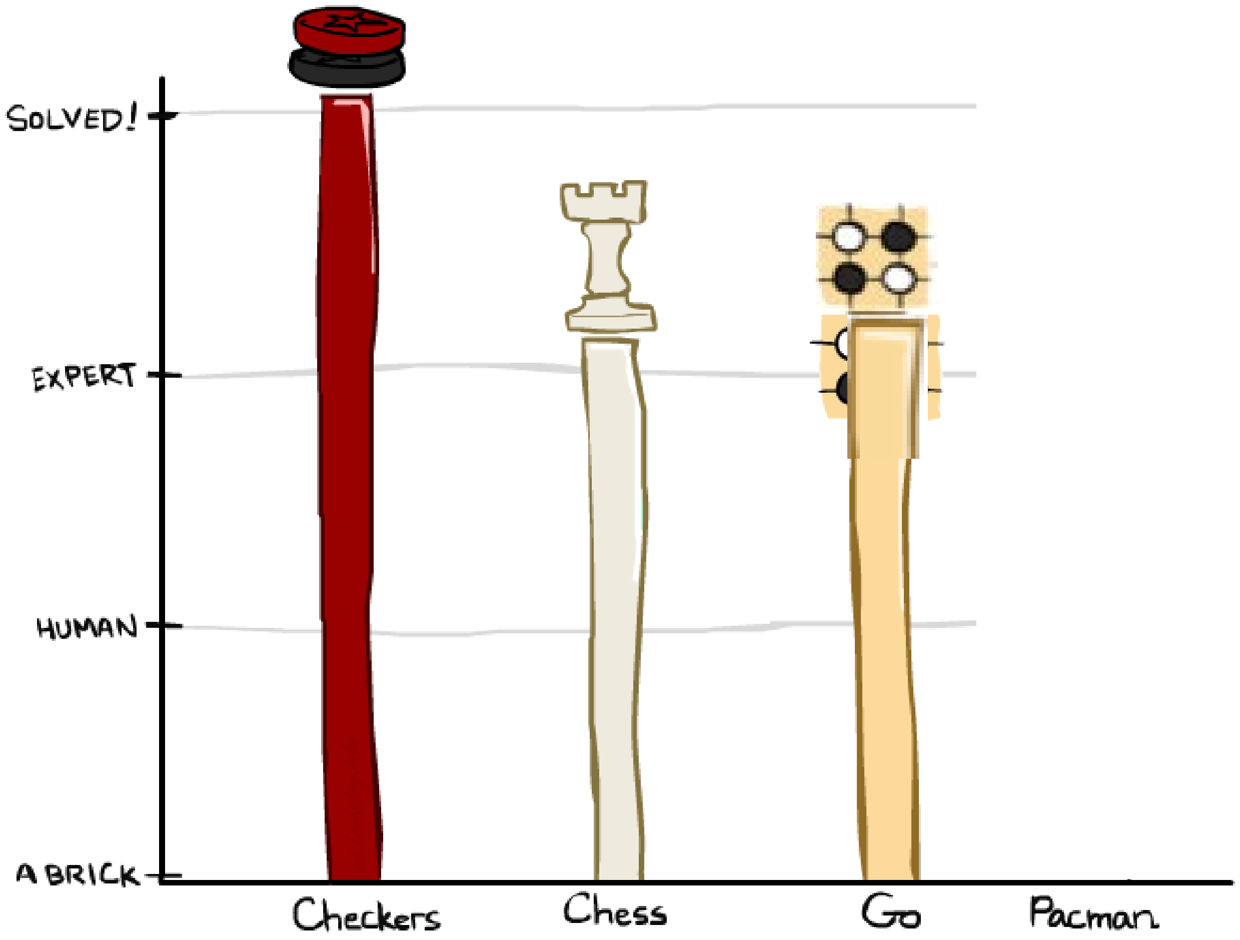
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
    - We don’t make AI to act in isolation, it should a) work around people and b) help people
    - That means that every AI agent needs to solve a game
-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

Zero-Sum Game Games ☺

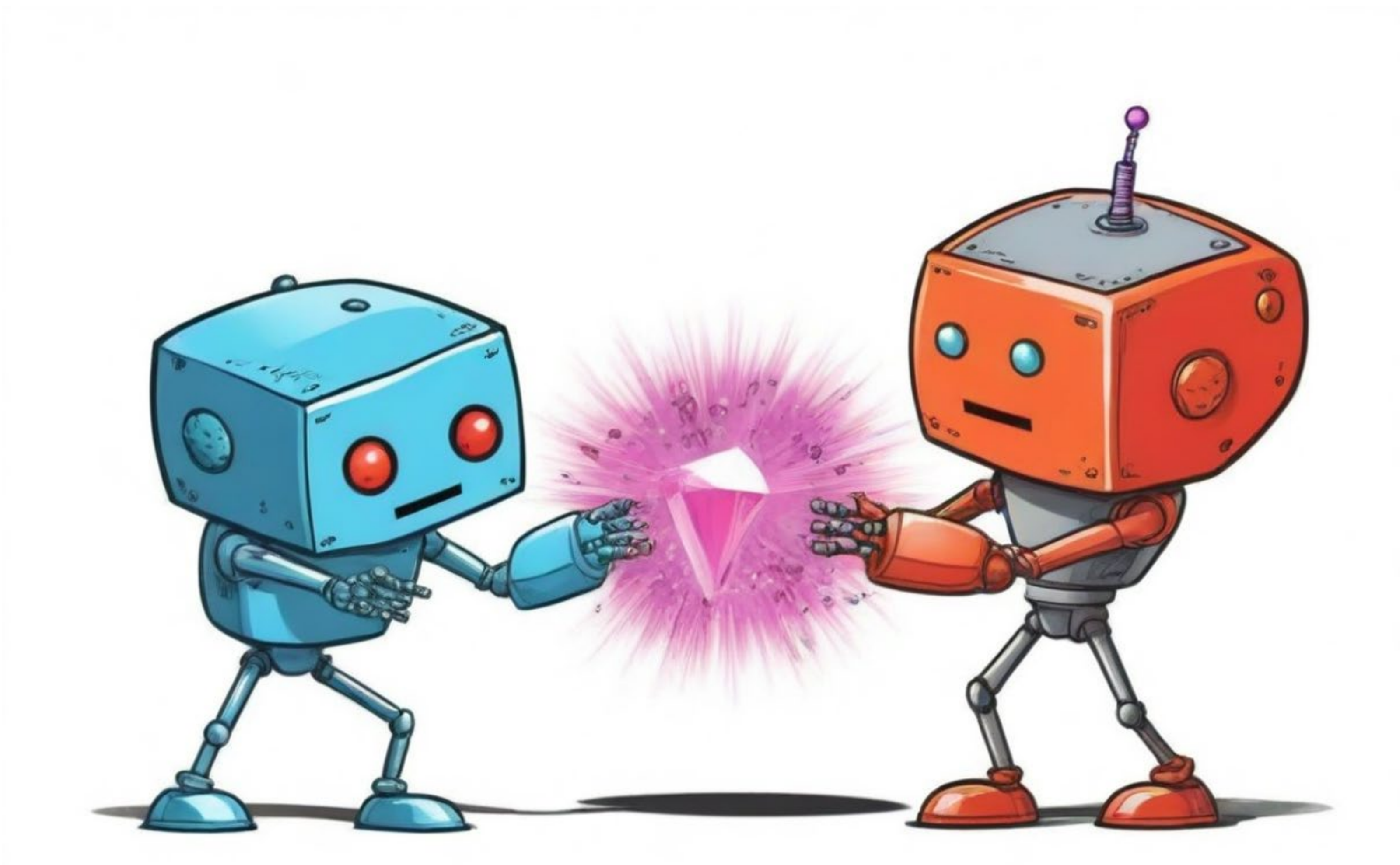
- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- **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: Google's AlphaGo defeated 9-dan professional (highest rank) Lee Sedol (live-streamed on YouTube) Branches at one action choice can be greater than 30**



Adversarial Games



Games vs. Search

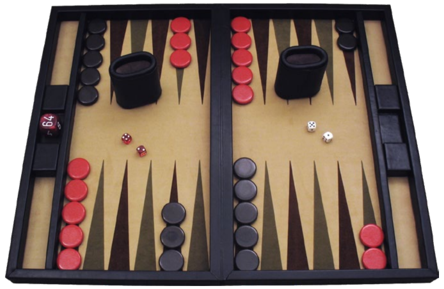
- "Unpredictable" opponent ⇒ solution is a strategy specifying a move for every possible opponent reply
- Time limits ⇒ unlikely to find goal, must approximate
- Plan of attack:
  1. Computer considers possible lines of play (Babbage, 1846)
  2. Algorithm for perfect play (Zermelo, 1912; Von Neumann,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

Types of Games

	deterministic	chance
perfect		
imperfect		

This Photo by Unknown Author is licensed under CC BY - BATTLESHIPS

This Photo by Unknown Author is licensed under CC BY - POKER



This Photo by Unknown Author is licensed under CC BY-SA - BACKGAMMON

This Photo by Unknown Author is licensed under CC BY-SA - GO

Types of Games

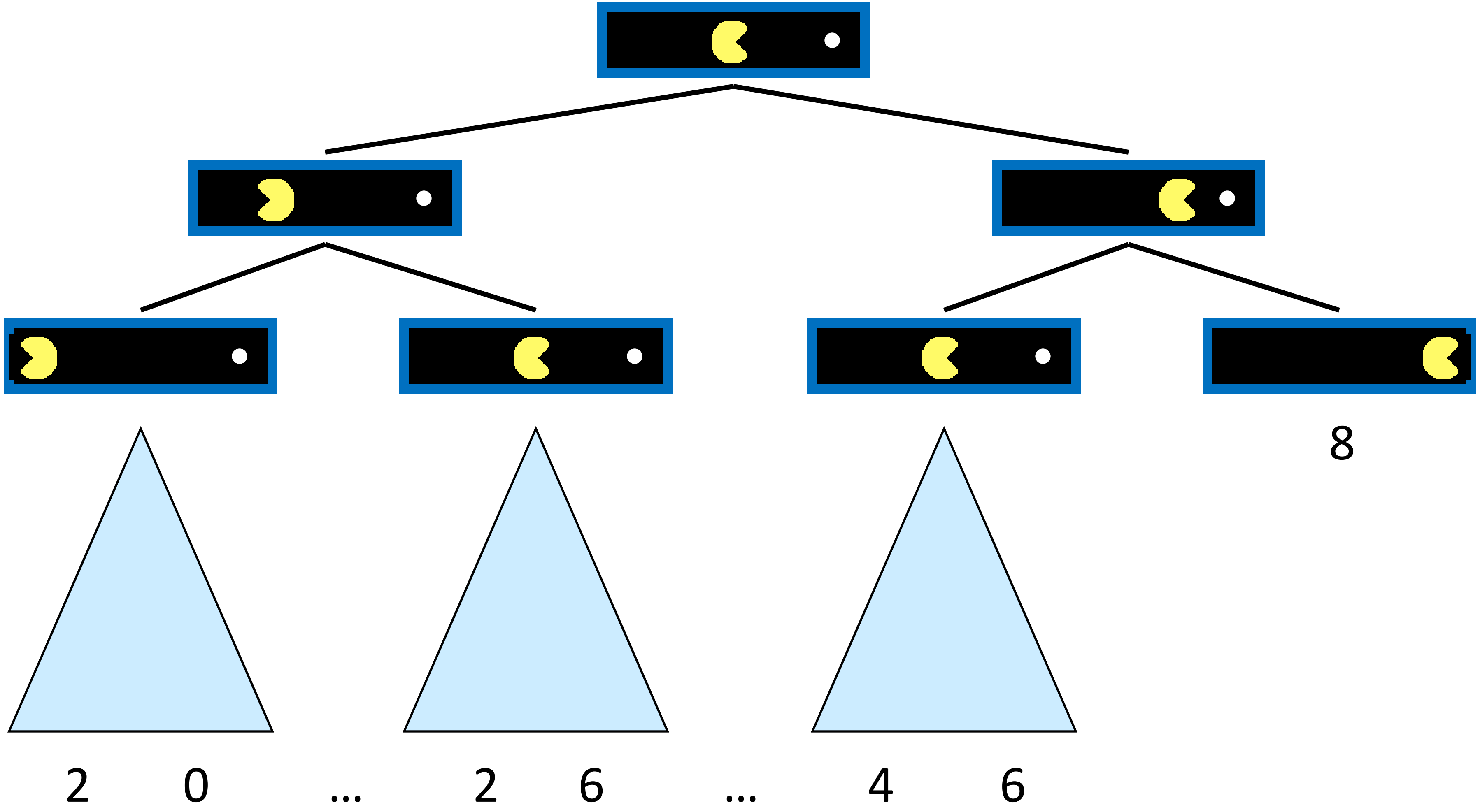
	deterministic	chance
perfect	 This Photo by Unknown Author is licensed under CC BY-SA	 This Photo by Unknown Author is licensed under CC BY-SA
imperfect	 This Photo by Unknown Author is licensed under CC BY	 This Photo by Unknown Author is licensed under CC BY

Types of Games

	deterministic	chance
perfect	chess, checkers, othello, go, tic tac toe	backgammon, monopoly
imperfect	battleships, blind tic tac toe	bridge, poker, scrabble, nuclear war

Game Trees

Single-Agent Trees (Good for and-tree)

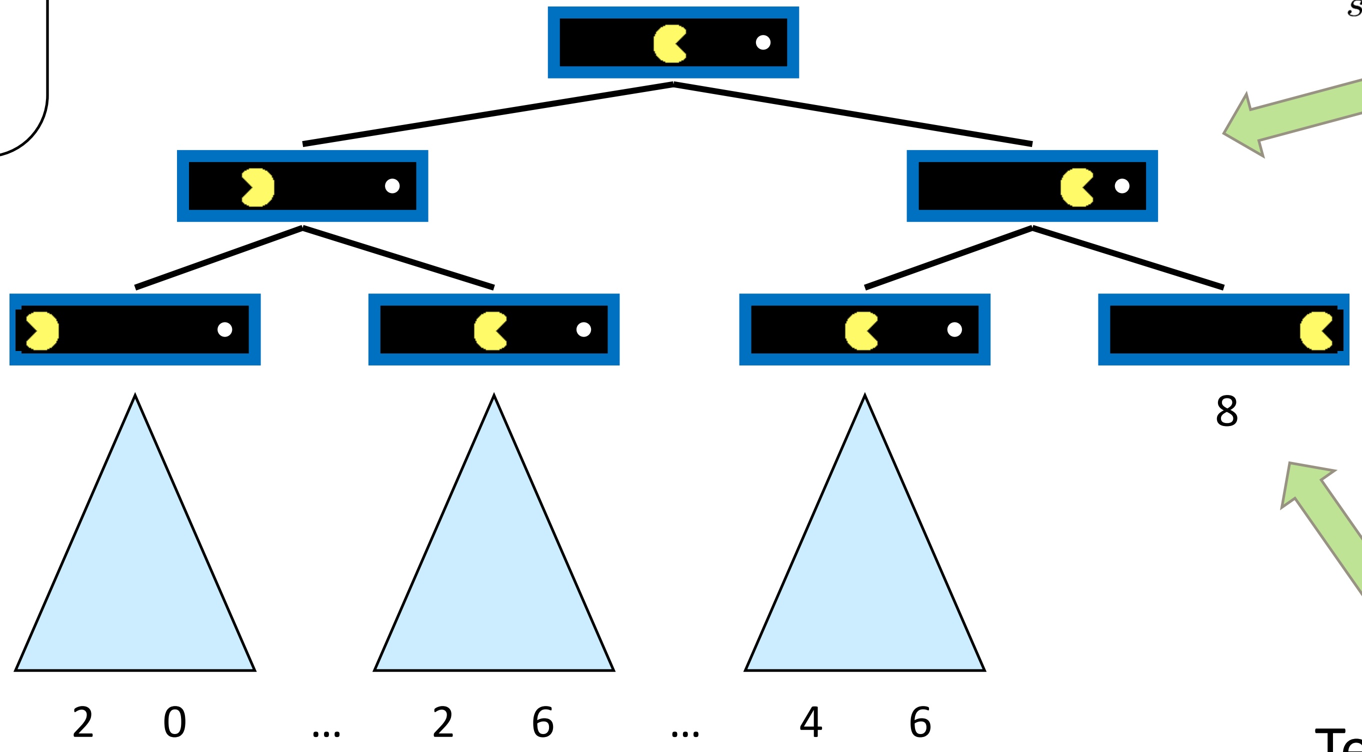


Value of a State

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

Non-Terminal States:

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

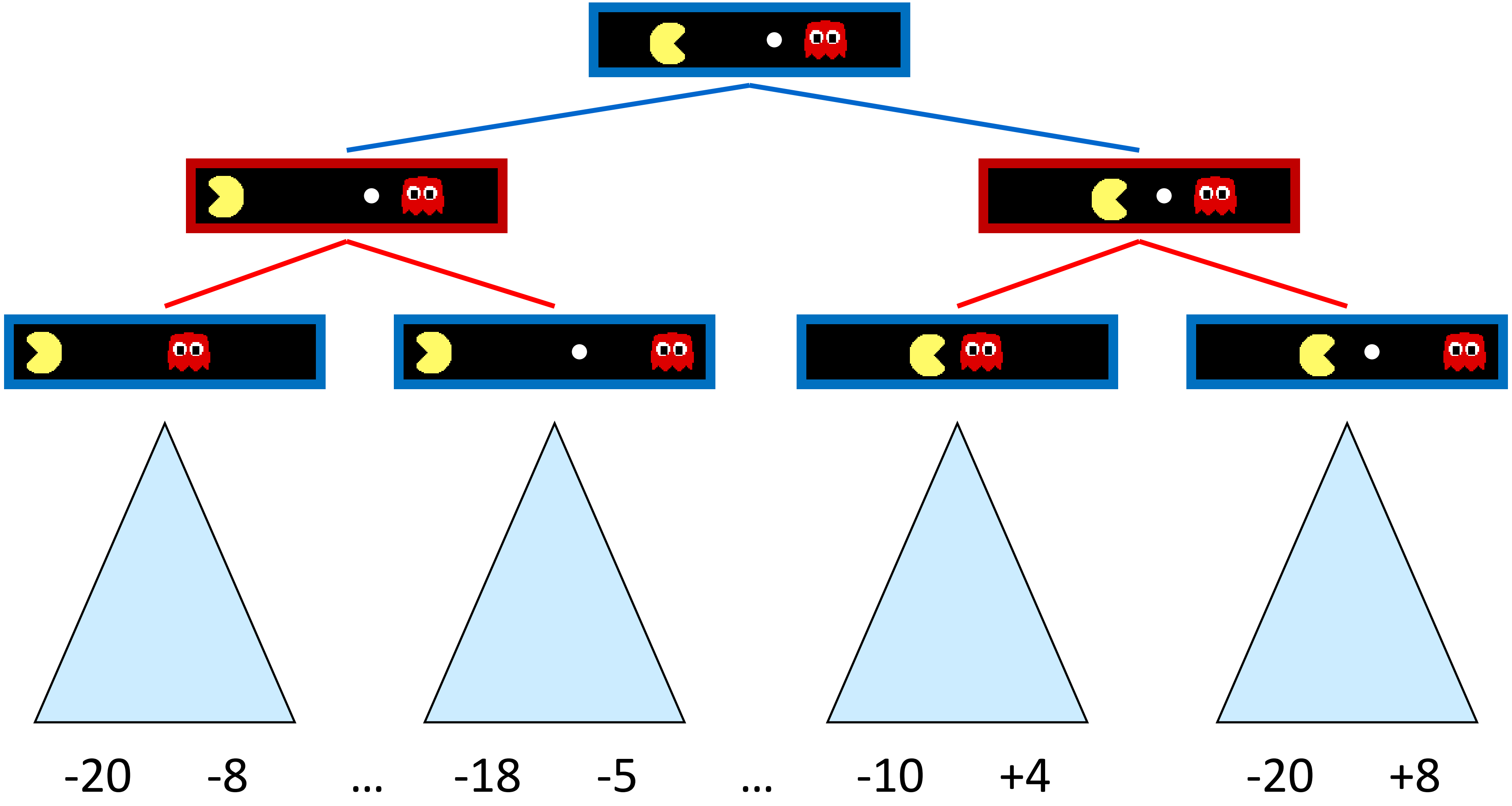


Terminal States:

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

UNIVERSITY OF CALGARY

Adversarial Game Trees (could be and-tree, better as and-or-tree)



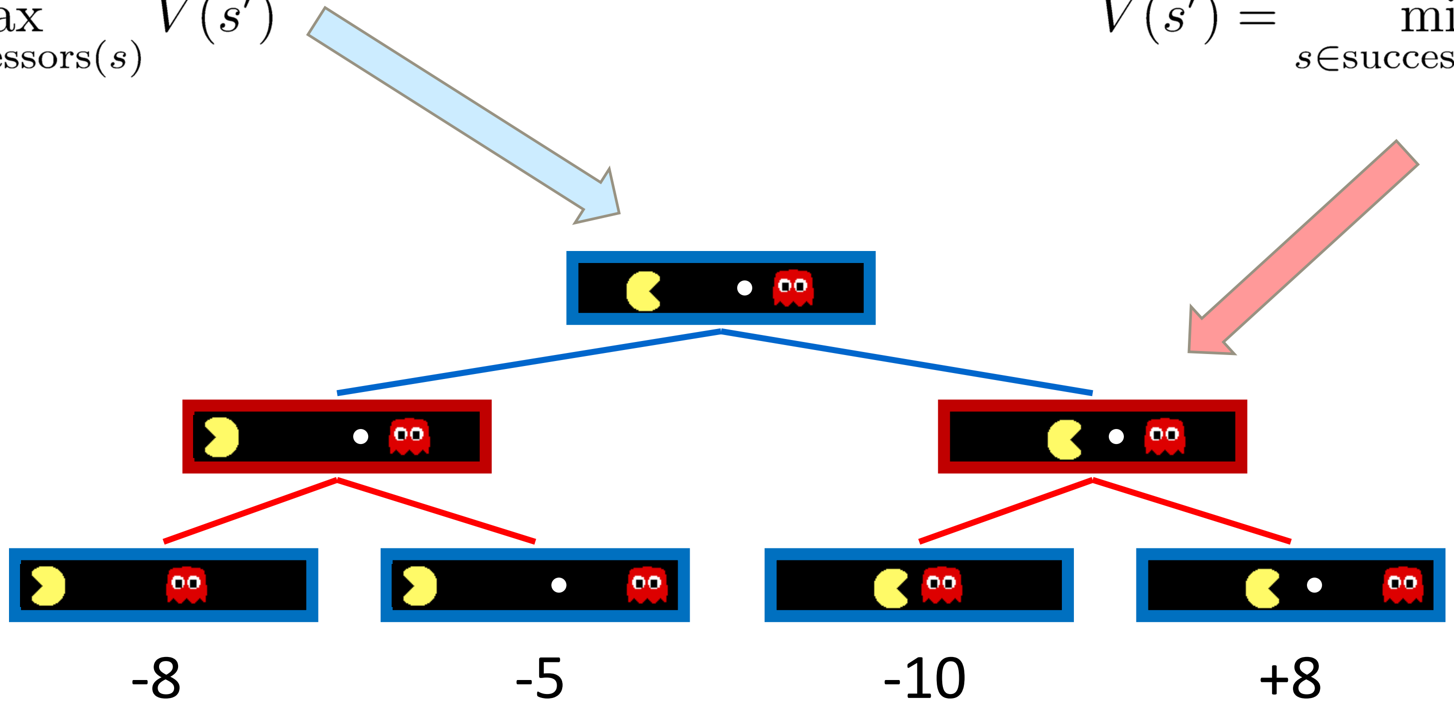
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

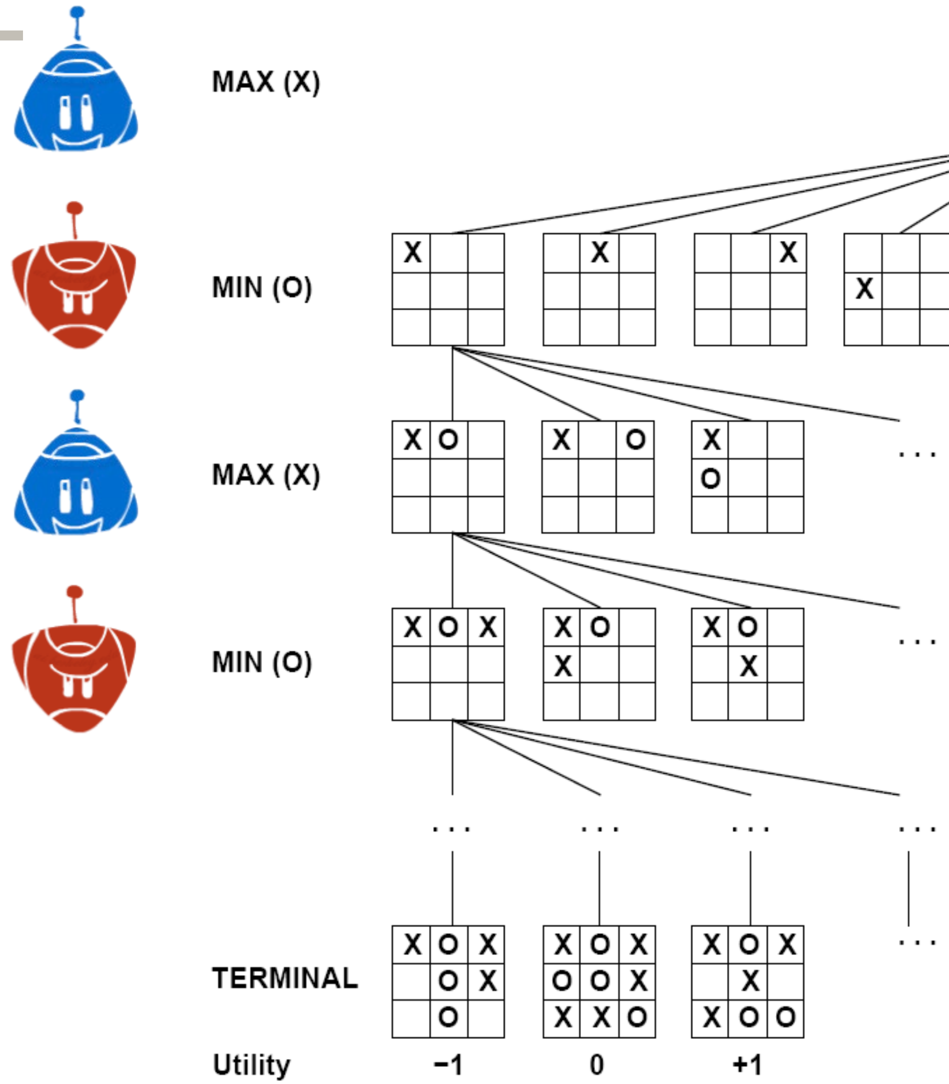
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

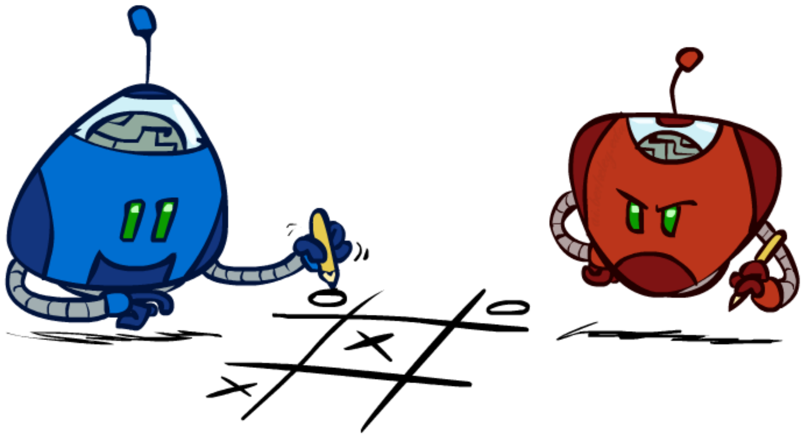


Terminal States:

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

Tic-Tac-Toe Game Tree





MiniMax

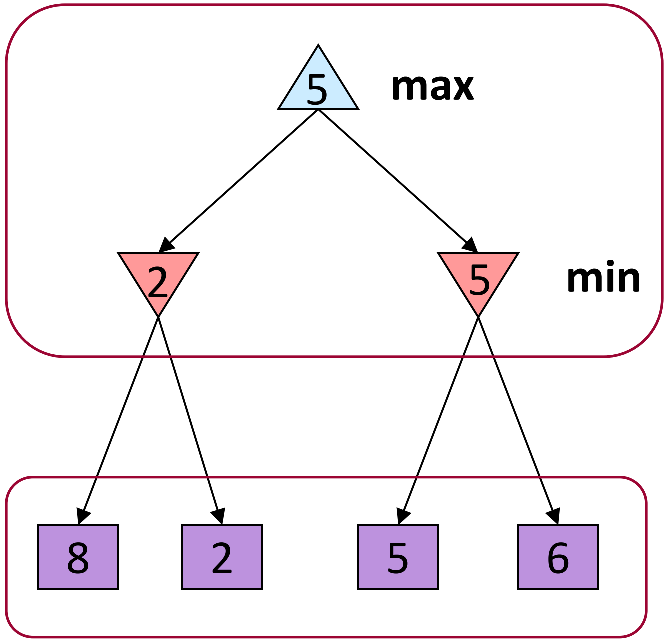
Minimax

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

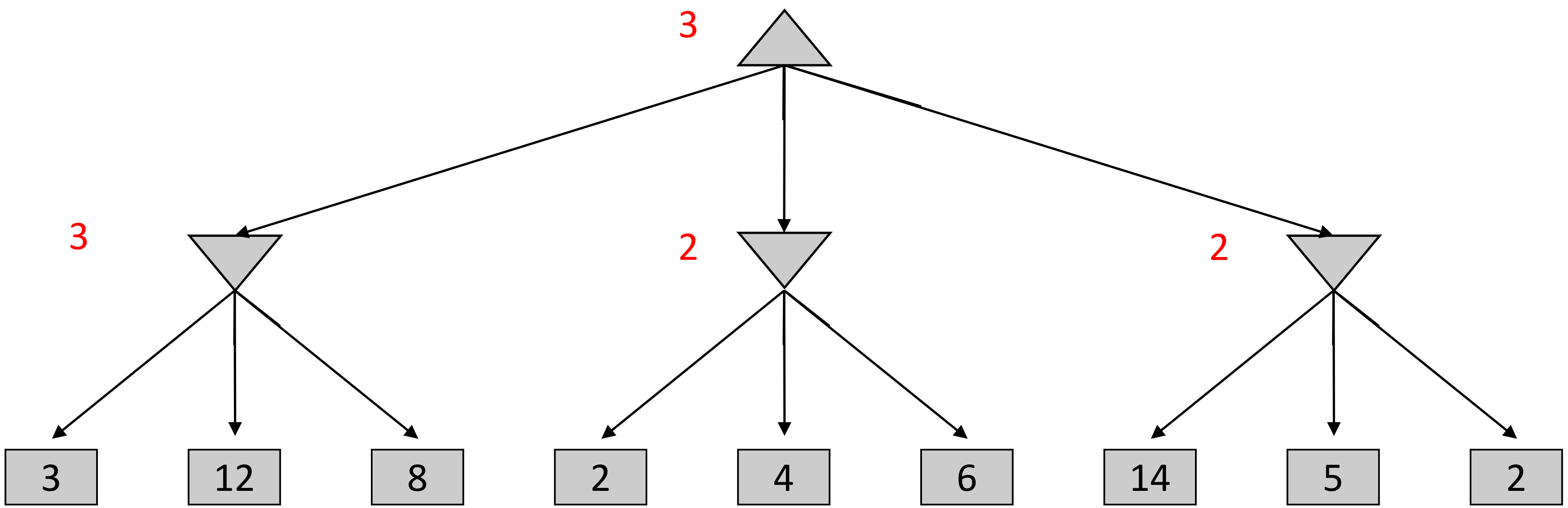
Minimax Implementation (Dispatch)

```
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)
```

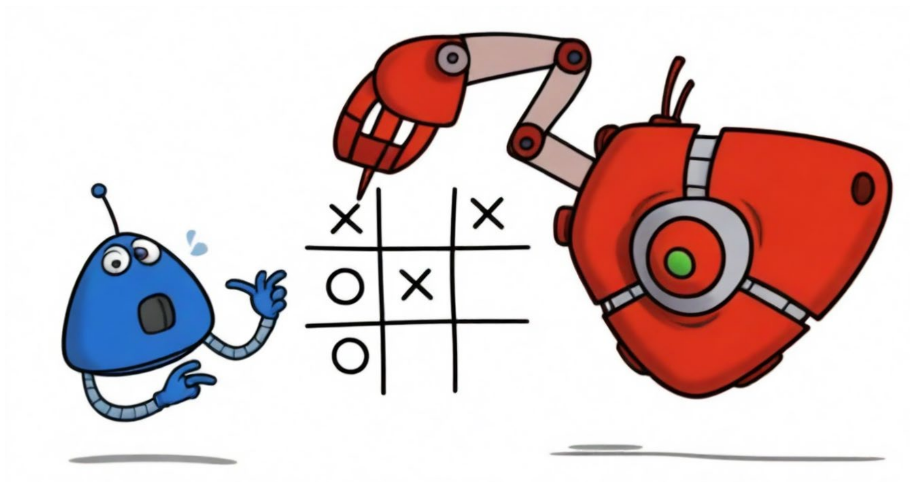
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

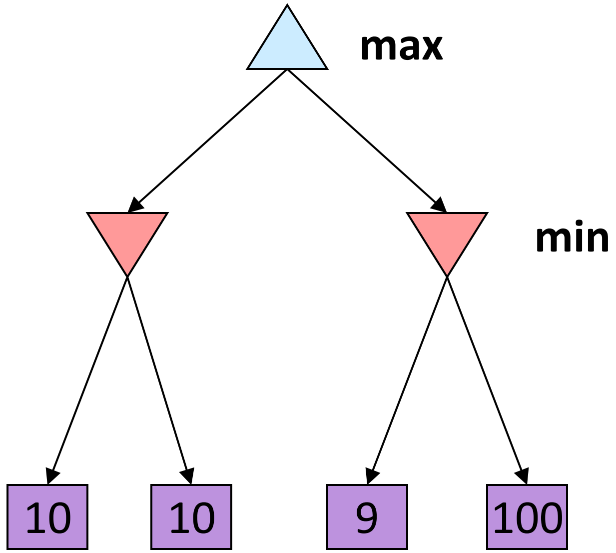
```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor))
    return v
```

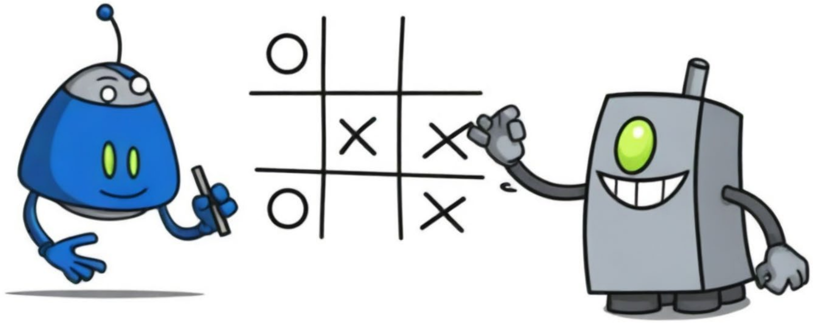
Minimax Example



Minimax Properties







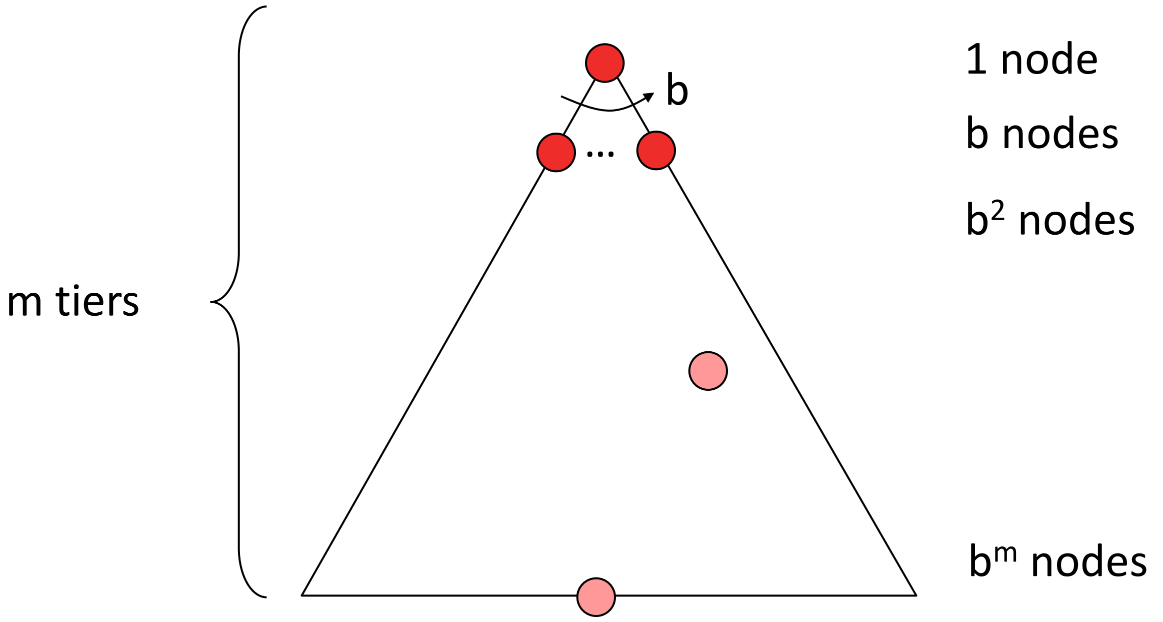
Optimal against a perfect player. Otherwise?

UNIVERSITY OF CALGARY

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

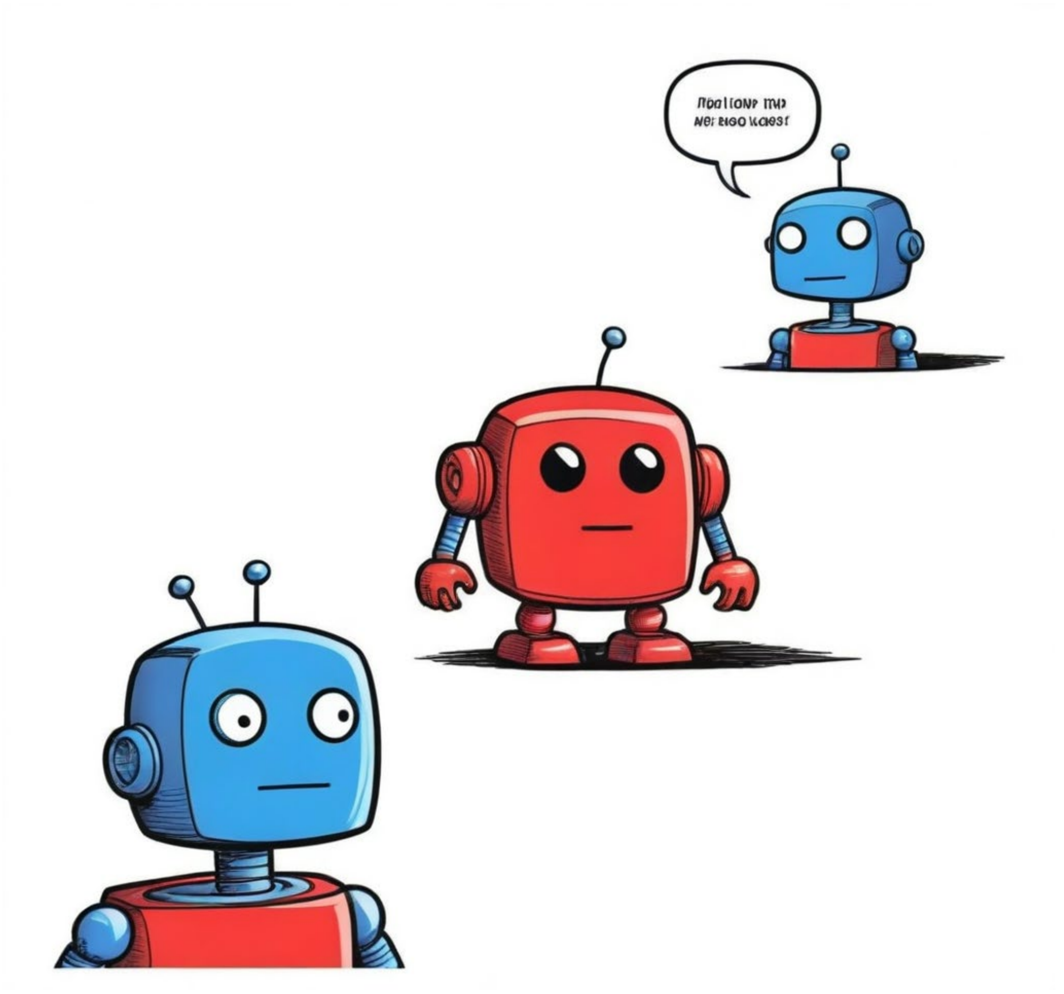


- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots.\ b^m = O(b^m)$

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?

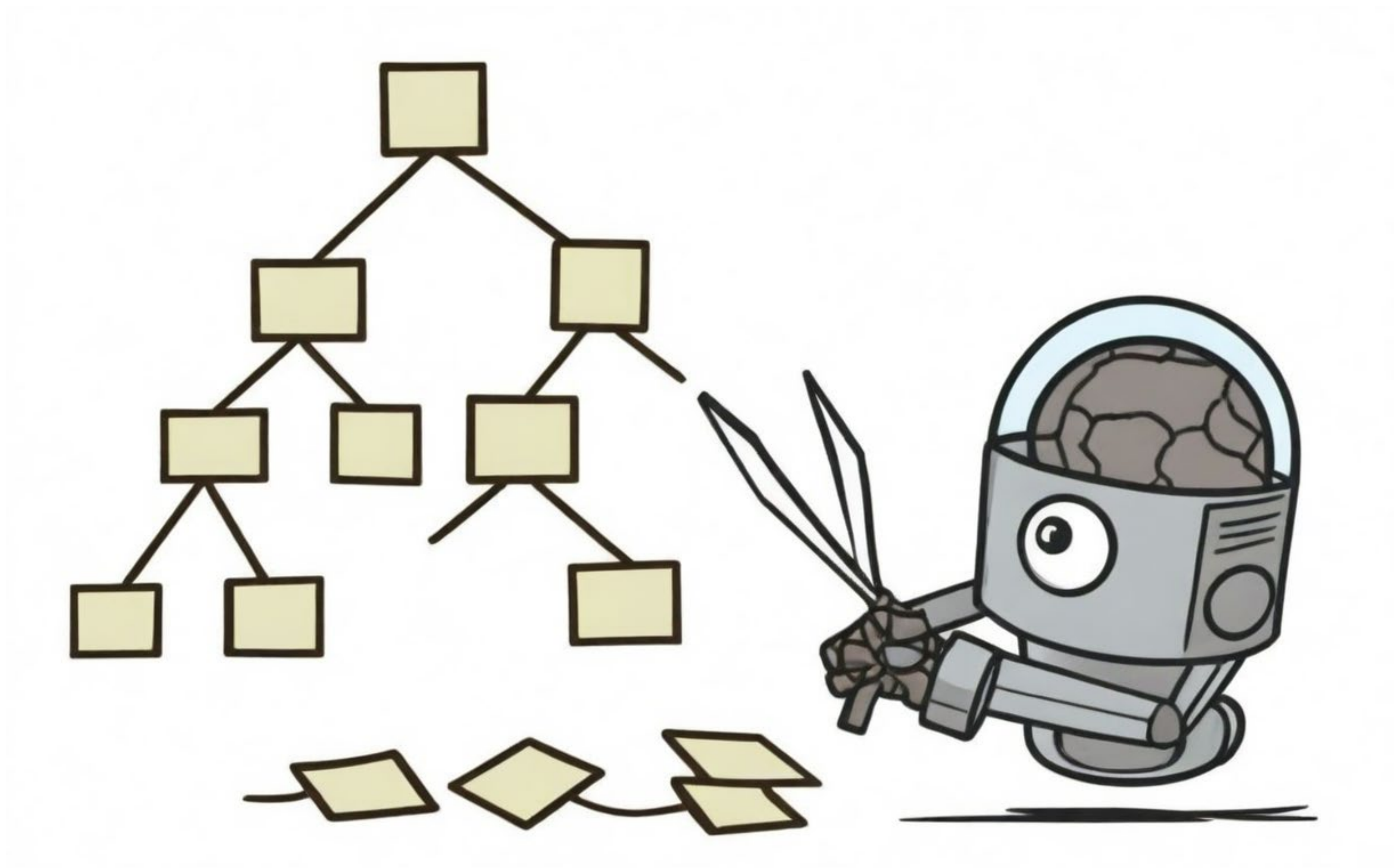


Considerations

Resource Limits

Pruning

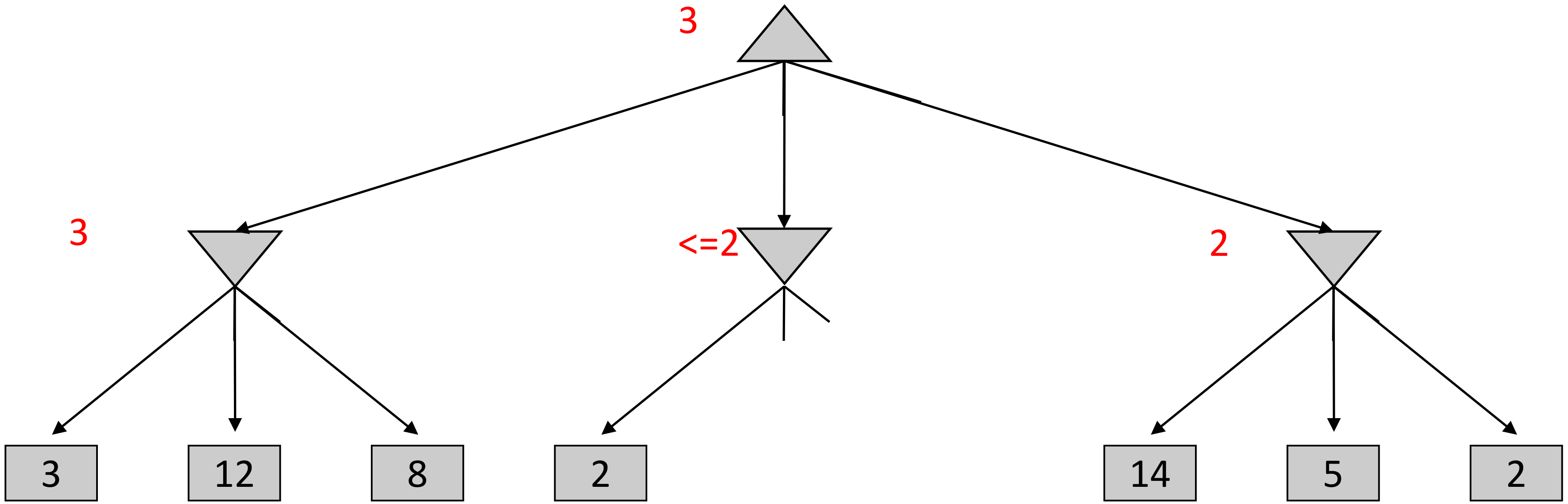
Game Tree Pruning f_{bound}



Game Tree Pruning f_{bound}

- f_{bound} is an additional f-function used in tree controls to
  - sol-entry yes a leaf in And-Trees or And-Or-Trees
  - When considering a leaf, if a f_{bound} returns true a leaf is change to sol-sentry yes even if it is an incomplete solution
  - Generally used in optimization problems to stop exploration of a tree path where a solutions in it will not be better than a currently found complete solution
  - These f-bound yes leaf nodes are still part of the complete and-tree consideration at the end, it was simply unnecessary to finish exploring them to determine (in an optimization problem) that any further exploration was necessary (the leaf nodes would be order as not as optimal as another sol-entry yes completion solution we already have)
  - We can apply this idea to minimax (to prune the and-or-tree it creates)
    - It is common way to improve the runtime performance of tree algorithms so that they are less costly than simply an exhaustive exploration of all possible combinations (now we can skip repetitive, invalid, or already more costly sub-branches)

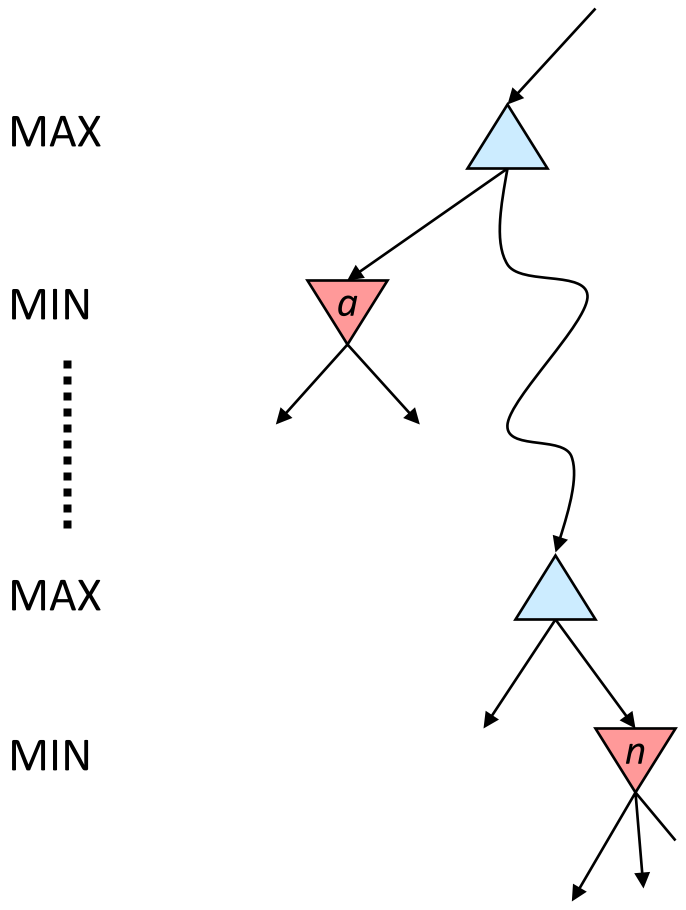
Example



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



UNIVERSITY OF CALGARY

Alpha-Beta Implementation

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

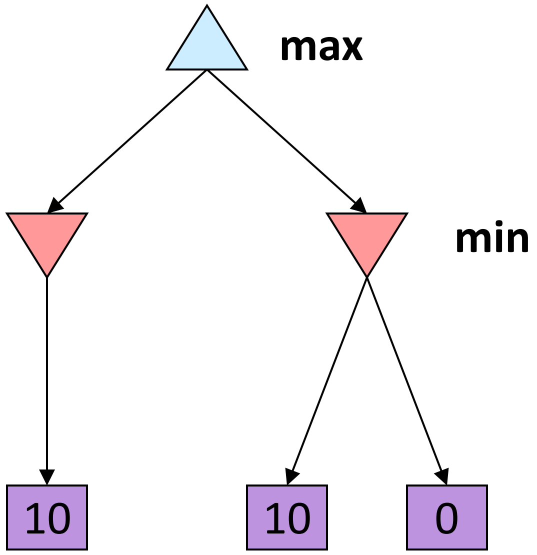
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 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

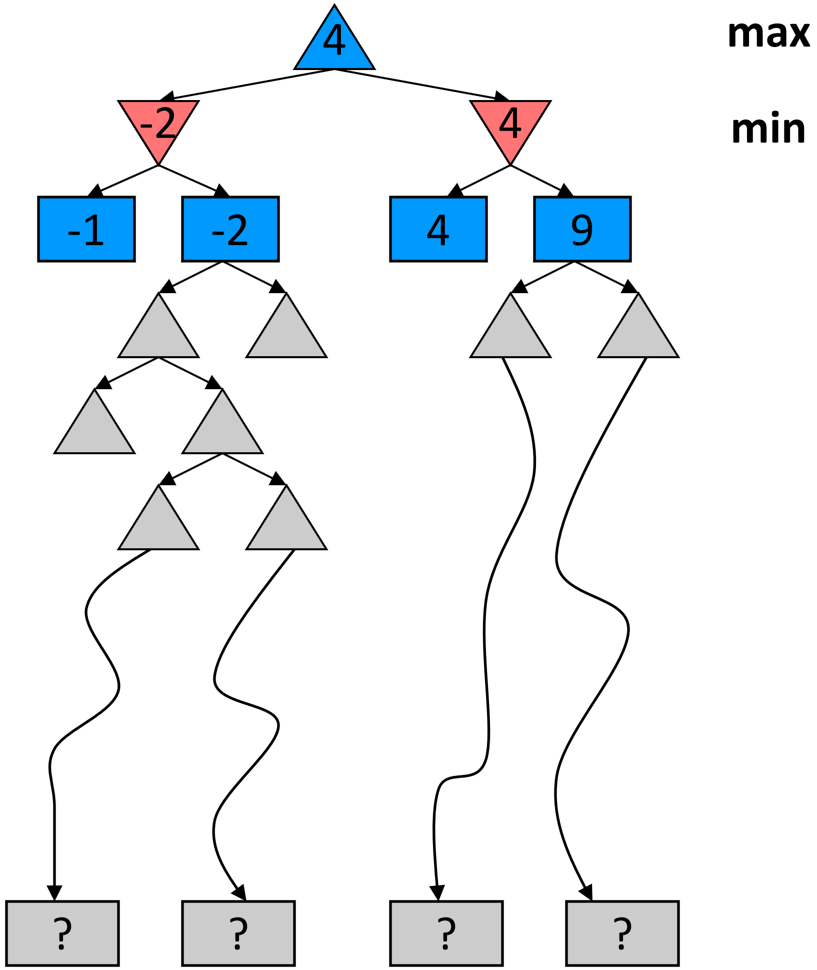
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…

- This is a simple example of metareasoning (computing about what to compute)

Depth Limit

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

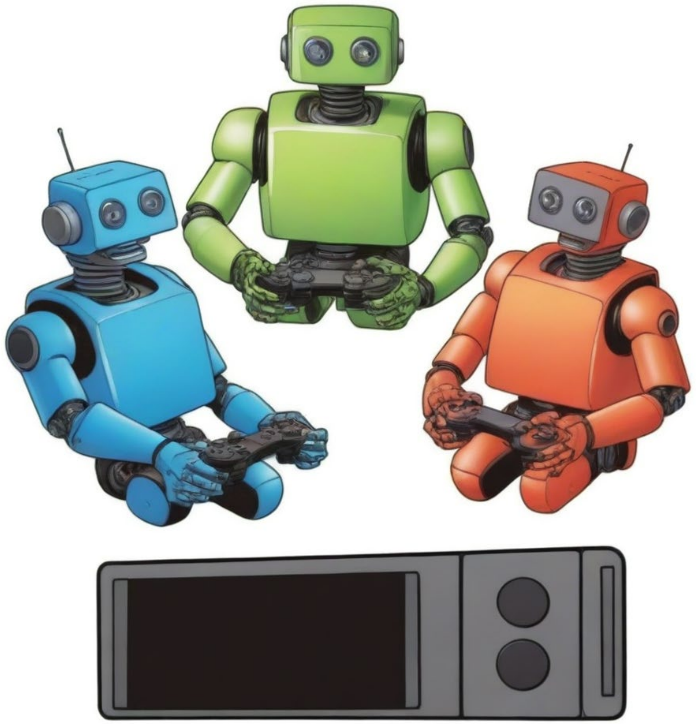


Depth Matters

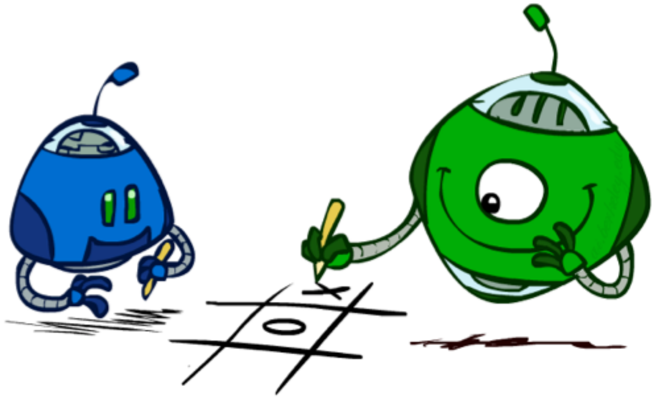
- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

Other Game Types

Other Game Types

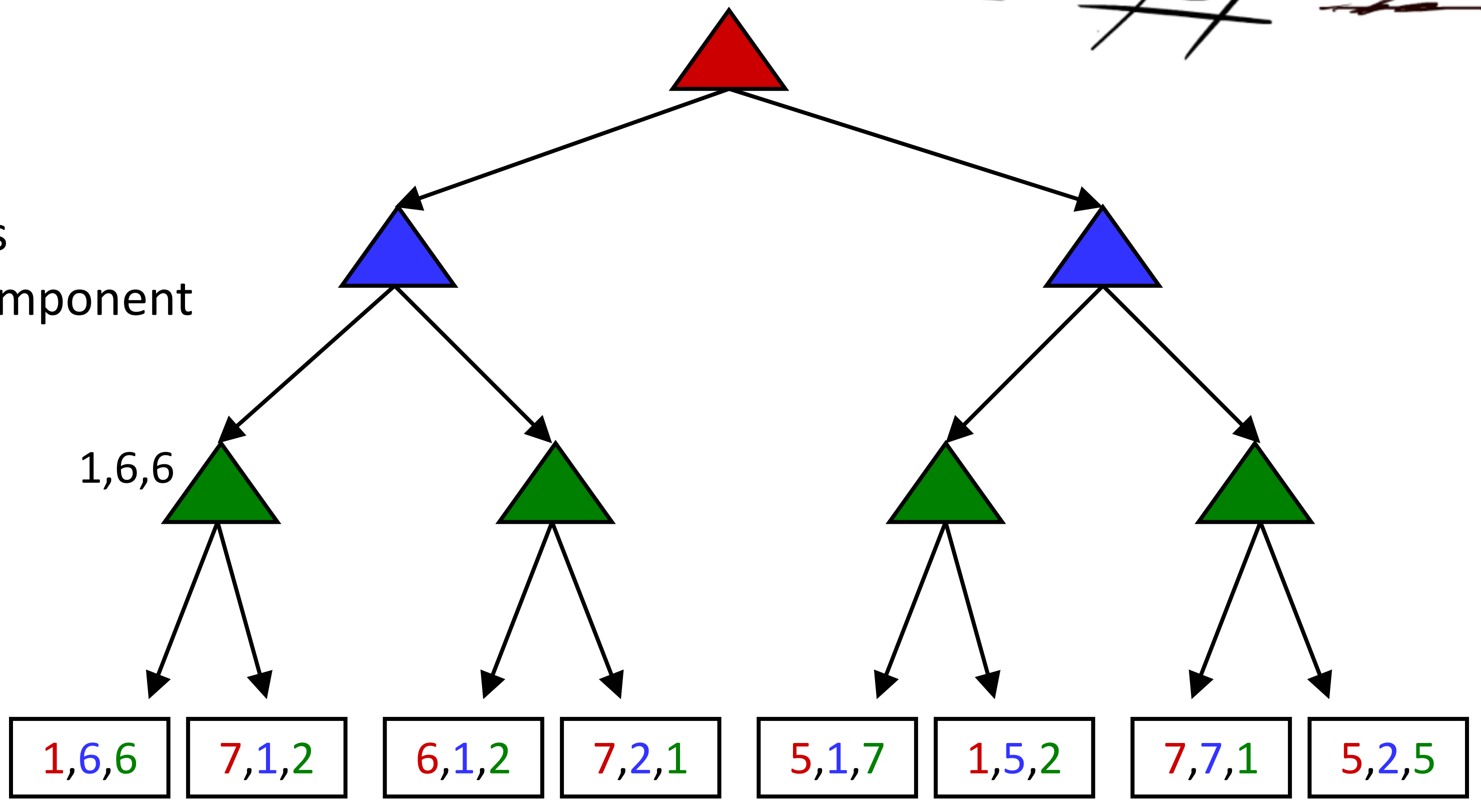


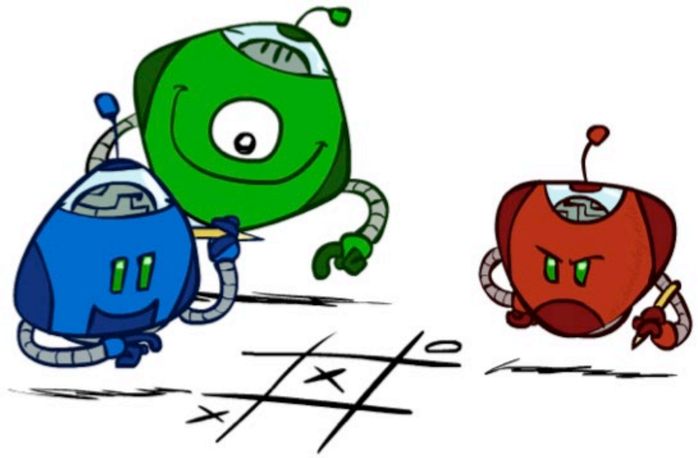
Multi-Agent Utilities



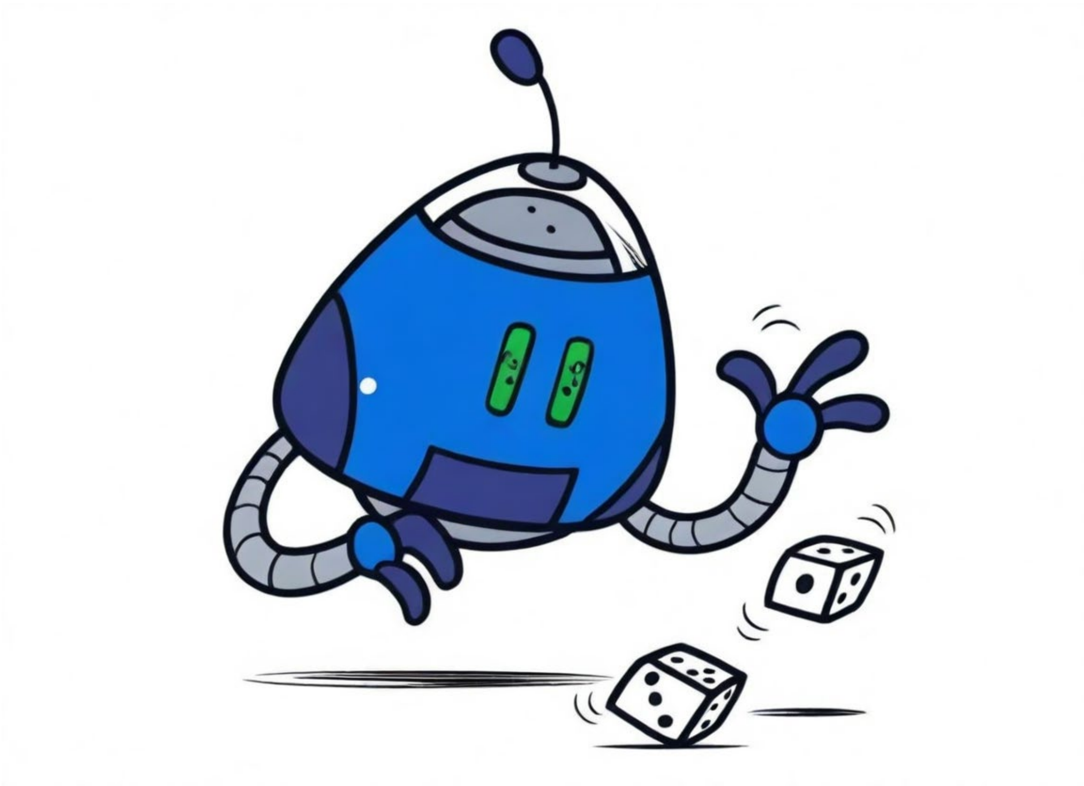
- What if the game is not zero-sum, or has multiple players?

- 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…



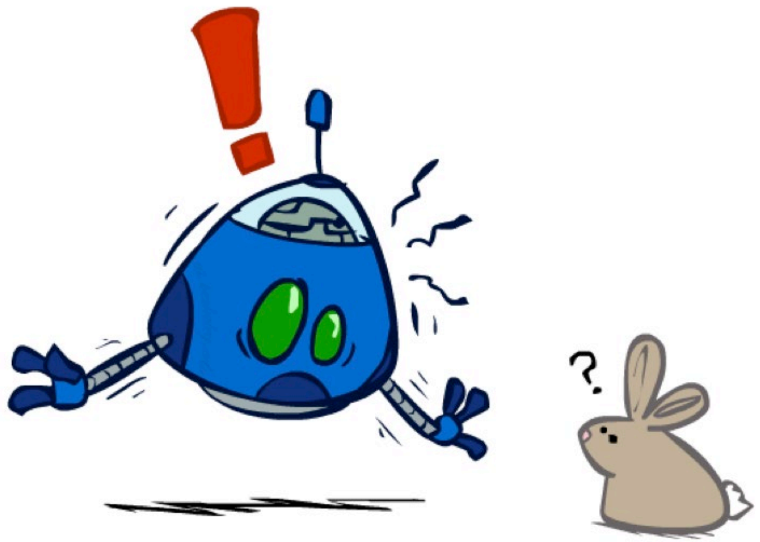


Uncertain Outcomes

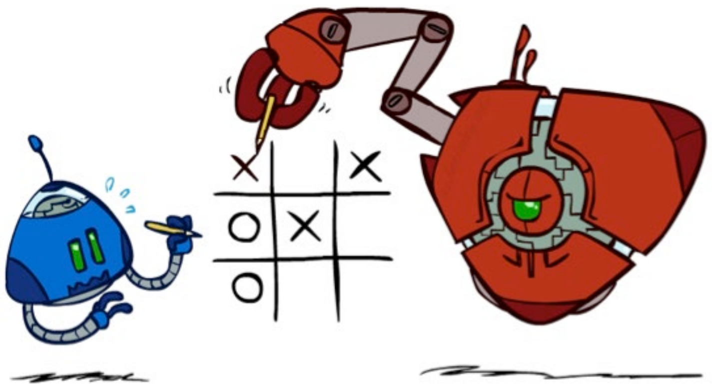


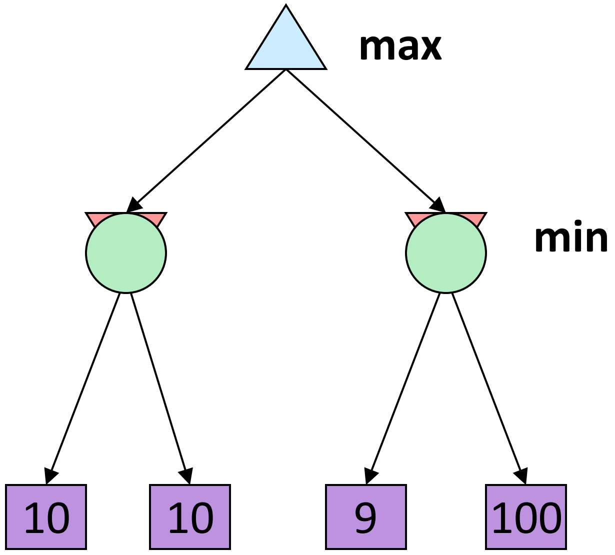
Why not minimax?

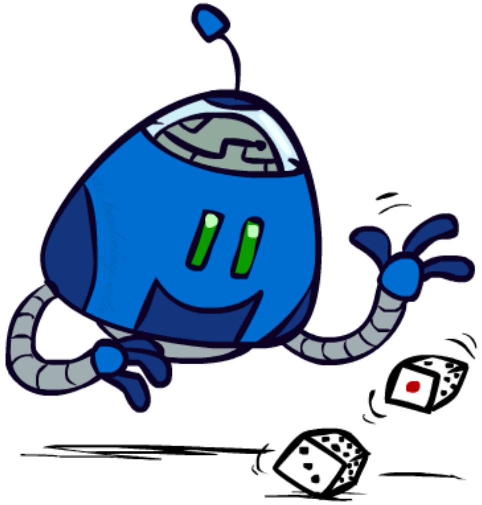
- Worst case reasoning is too conservative
- Need average case reasoning



Worst-Case vs. Average Case







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

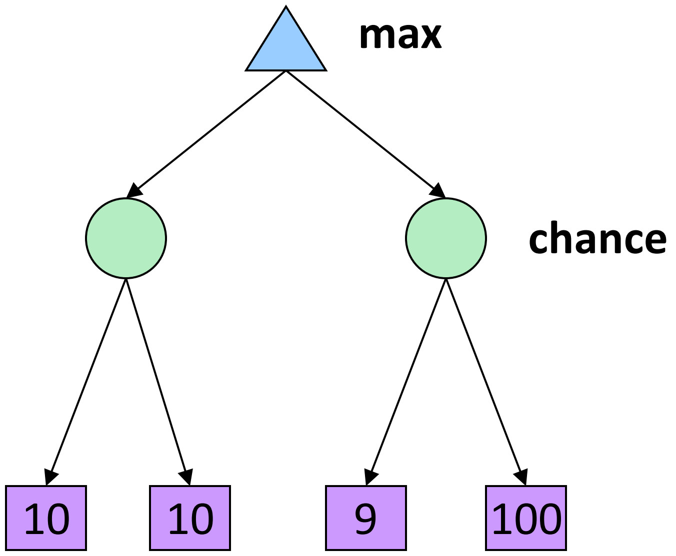
Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Unpredictable humans: 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
  - I.e. take weighted average (expectation) of children

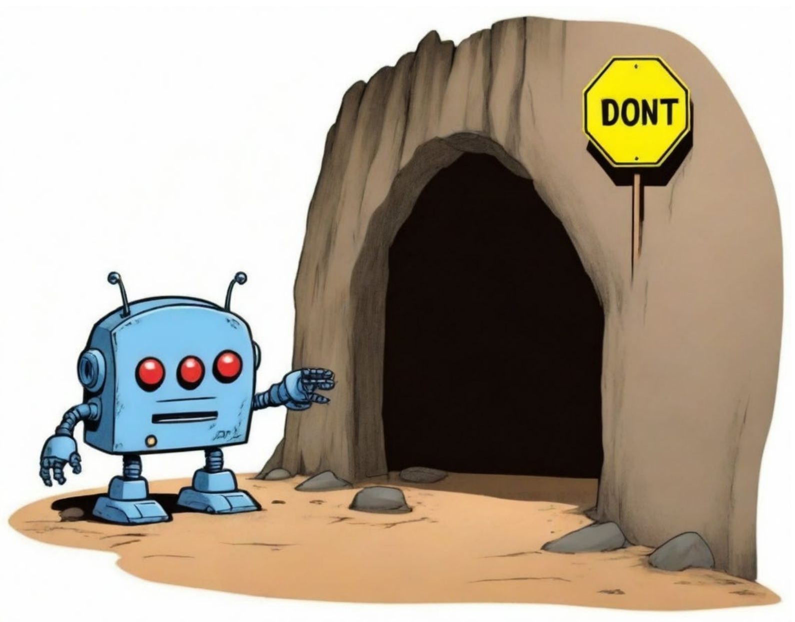
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



The Dangers of Optimism and Pessimism

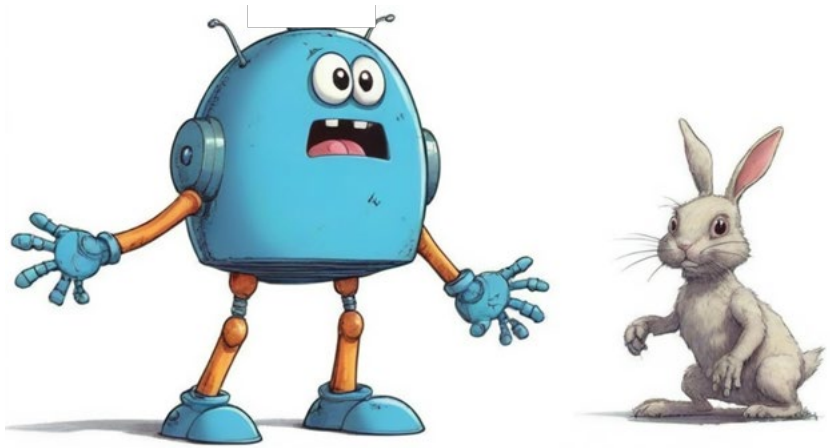
Dangerous Optimism

Assuming chance when the world is adversarial



Dangerous Pessimism

Assuming the worst case when it’s not likely



Remarks

And-Or-Tree Pros and Cons

✓And-or-trees a very general search model

✓Every alternative and its consequences are visible in state

- ✓ ☞ control knows more than in and-trees with backtracking and thus can be more intelligent

✓Needed to model certain applications (min-max search, for example)

- For some applications too complex
  (why buy an apple tree if you just want an apple)
- Finding good controls is difficult
  (there can be too much knowledge)

Graph-based Search

- Some authors have suggested and-or-graph-based search with problems in nodes represented as sets of constraints as ultimate search model
- Since or-tree- and or-graph-based search processes often use an estimate on how good (with respect to distance to a solution) a leaf is, some people see them as special and-trees, resp. and-graphs (without backtracking) for optimization

  ☞ A*-algorithm (graph-based variant of branch-and-bound search)

Onward to … search controls

Jonathan Hudson, Ph.D.
jwhudson@ucalgary.ca
https://cspages.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY