STEVENS
INSTITUTE OF TECHNOLOGY
1870

Artificial Intelligence

Computer Science, CS541 - A

Jonggi Hong

Announcements

- Project
  - The information of project has been released!
  - You are assigned to project teams. Have a meeting as soon as possible.
  - Midterm report due: 11:59 pm, March 28

STEVENS
INSTITUTE OF TECHNOLOGY
1870

Recap

CSP

Constraint satisfaction problems (CSPs)

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Simple example of a *formal representation language*
- Allows useful general-purpose algorithms with more power than standard search algorithms

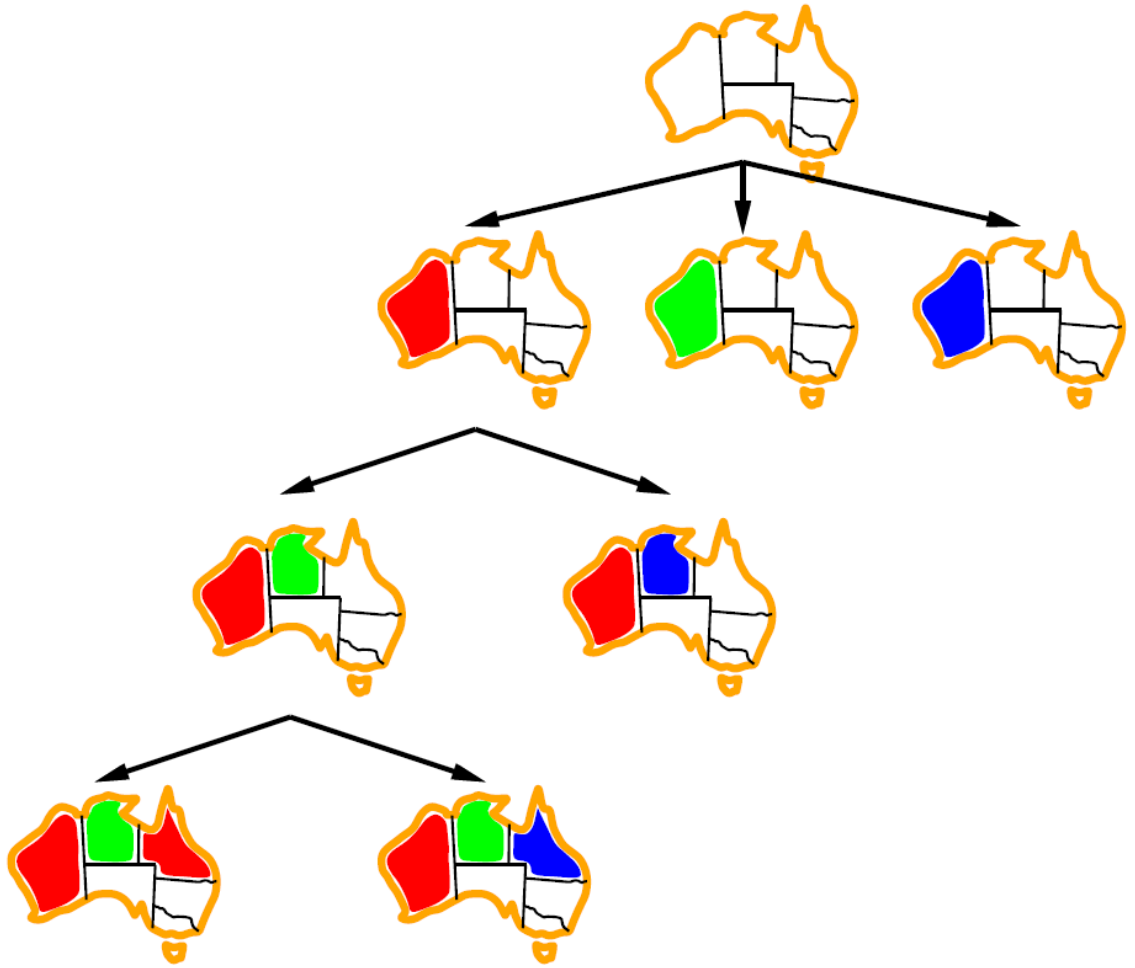
Backtracking search

- Backtracking search is the basic uninformed algorithm for solving CSPs.

- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - *I.e.*, [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - “Incremental goal test”

- Depth-first search with these two improvements is called *backtracking search.*

Backtracking example

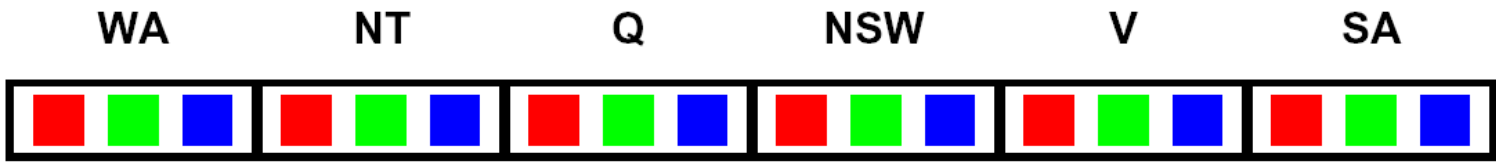


Improving backtracking

- General-purpose ideas give huge gains in speed.
- **Ordering**:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- **Filtering**: Can we detect inevitable failure early?
  - Forward checking
  - Constraint propagation
- **Structure**: Can we exploit the problem structure?

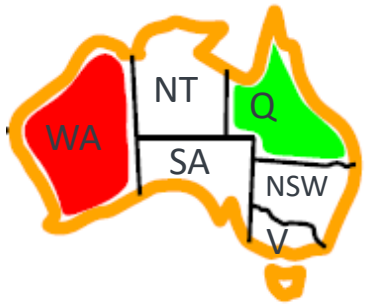
Filtering: forward checking

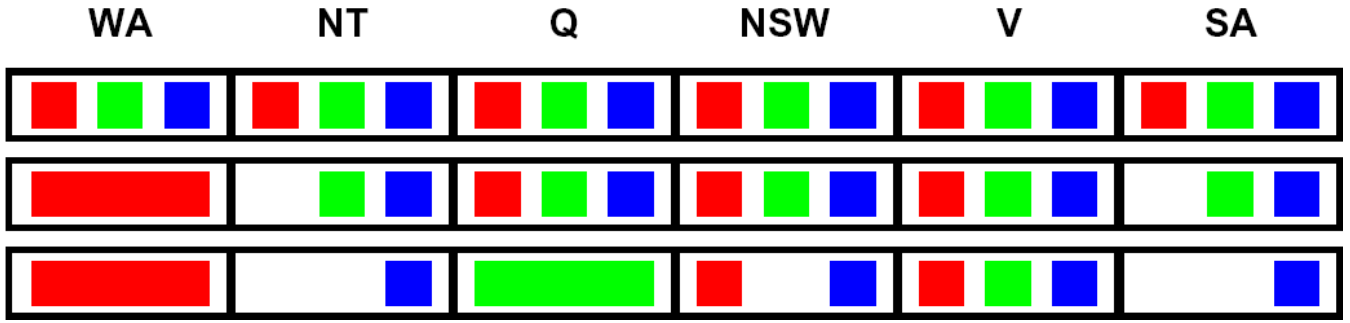
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Filtering: constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

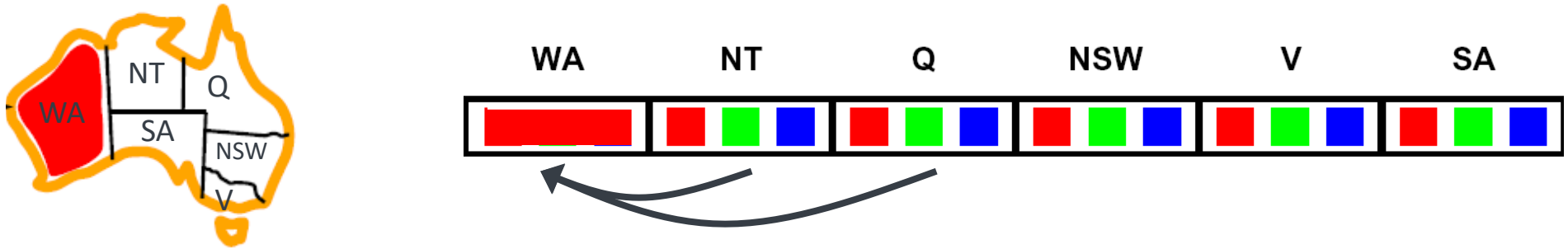




- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of a single arc

- An arc X → Y is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

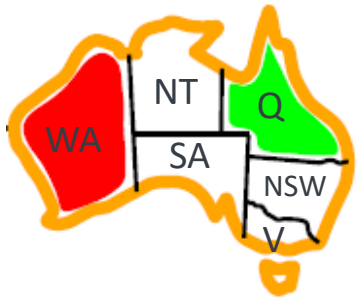


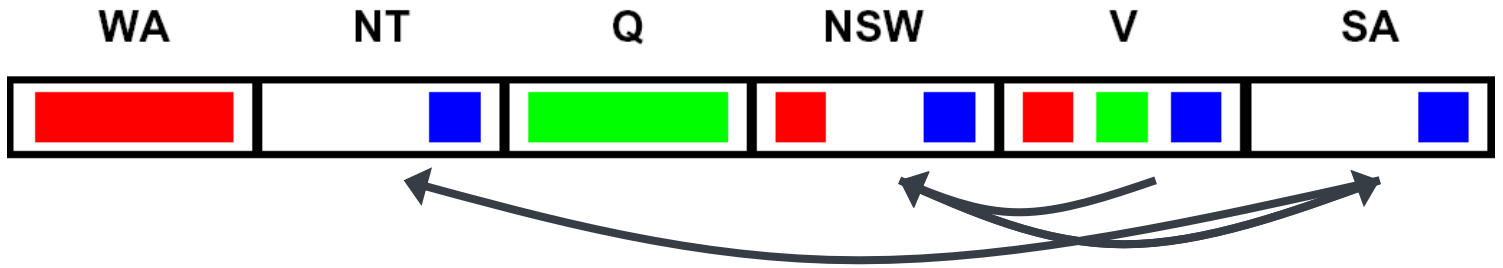
- Forward checking: enforcing consistency of arcs pointing to each new assignment

Remember: Delete from the tail!

Arc consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:



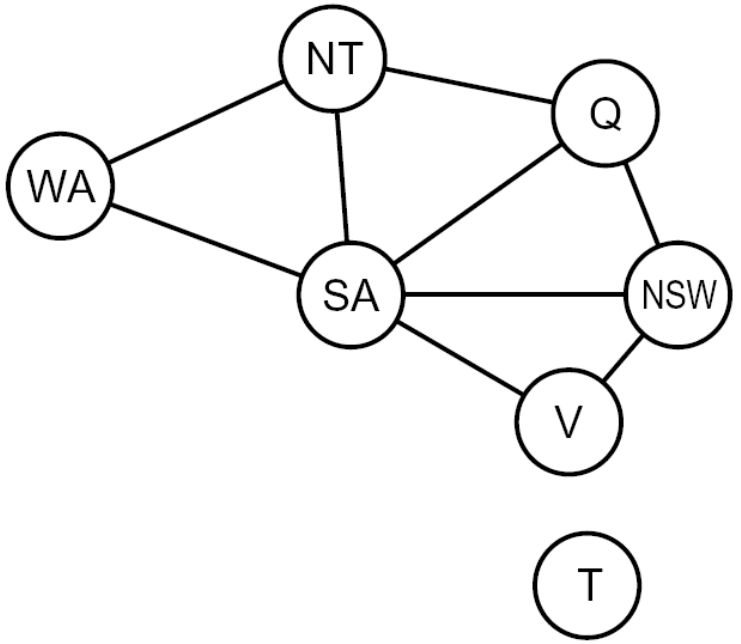


- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

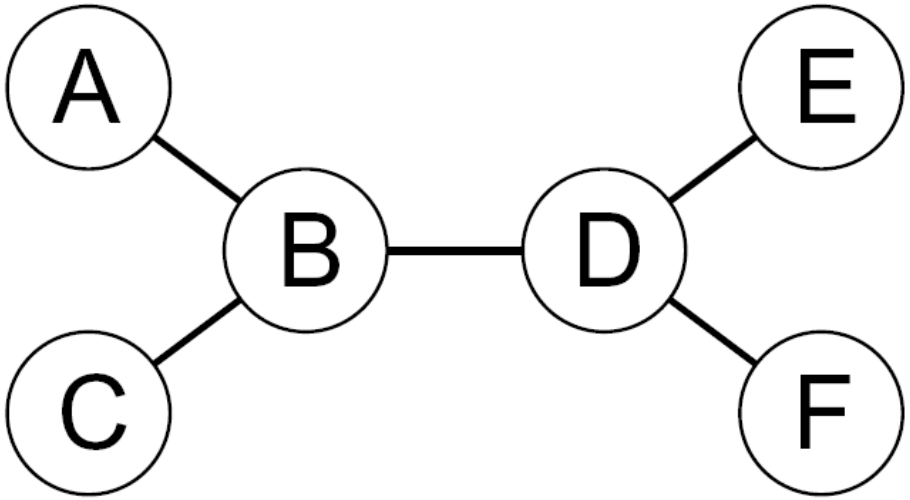
Remember: Delete from the tail!

Problem structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph
- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



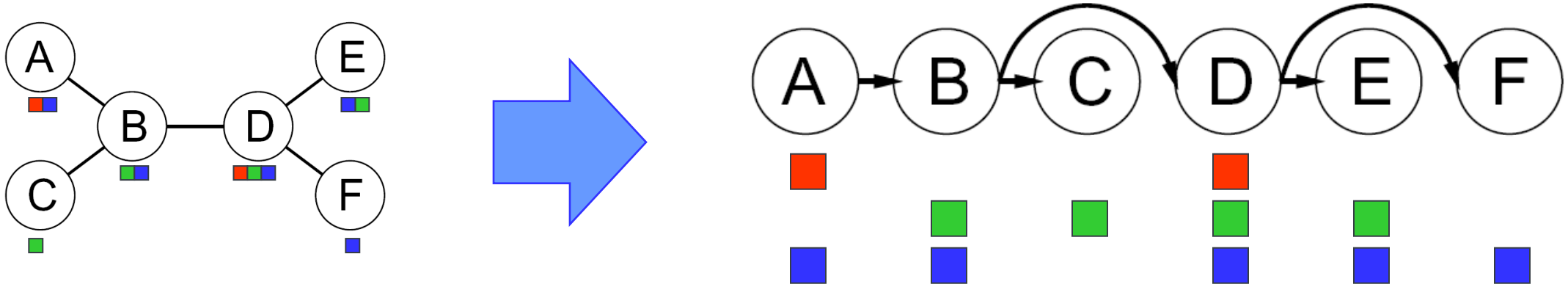
Tree-structured CSPs



- if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

Tree-structured CSPs

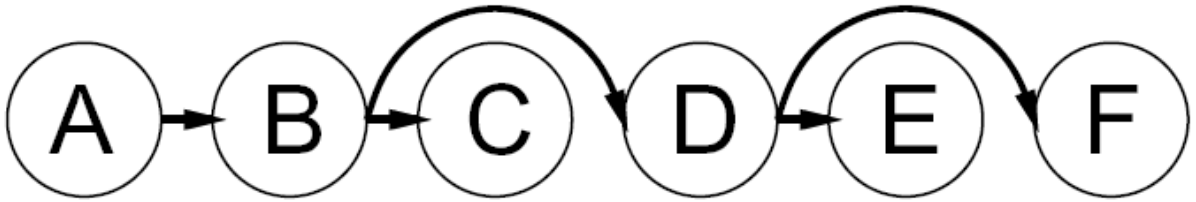
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)
- Runtime: $O(nd^2)$ (why?)

Tree-structured CSPs

- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)



- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position
- Why doesn't this algorithm work with cycles in the constraint graph?

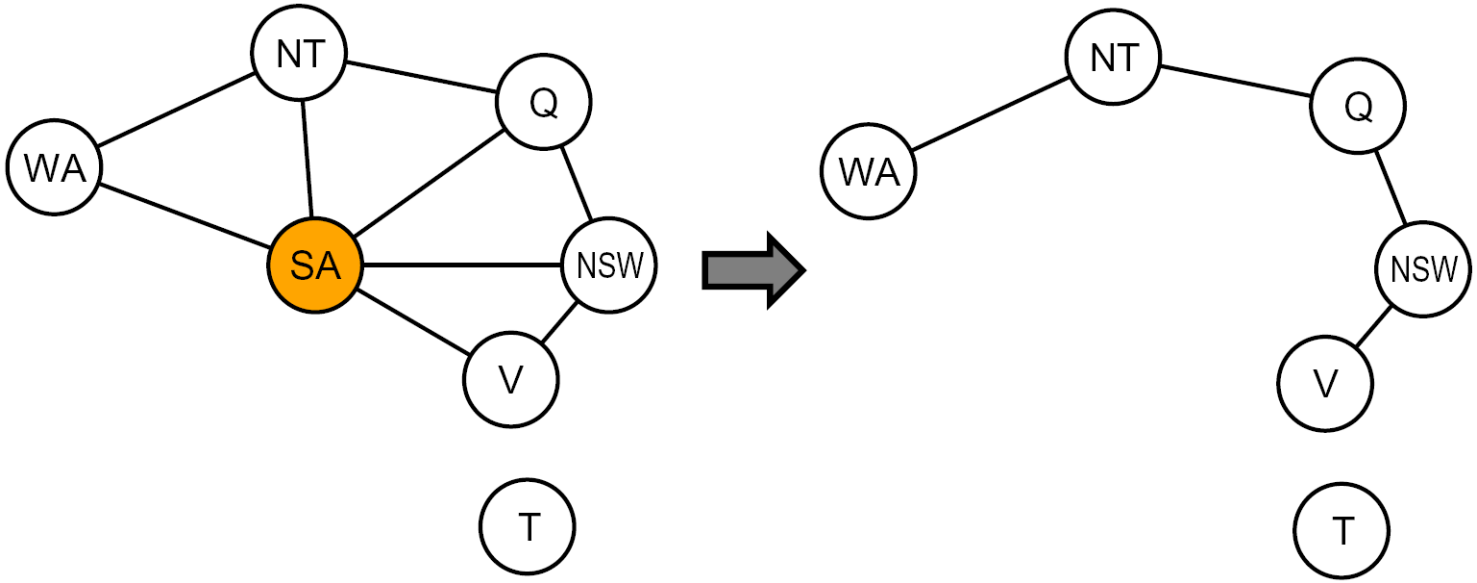
STEVENS
INSTITUTE OF TECHNOLOGY
1870

Constraint Satisfaction (cont.)

Chapter 6

Improving structure

Nearly tree-structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime O((d^c) (n-c) d^2), very fast for small c

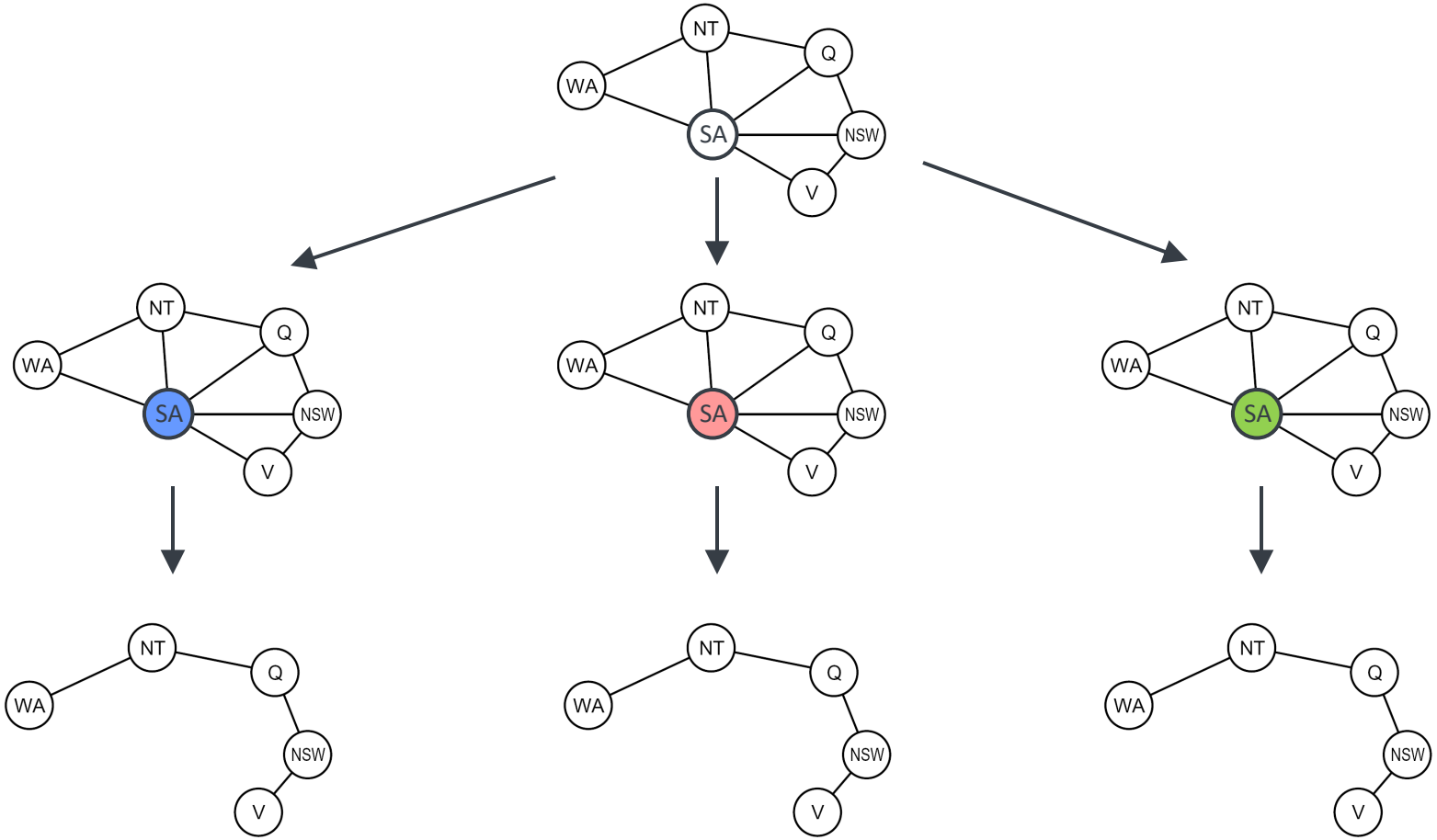
Cutset conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

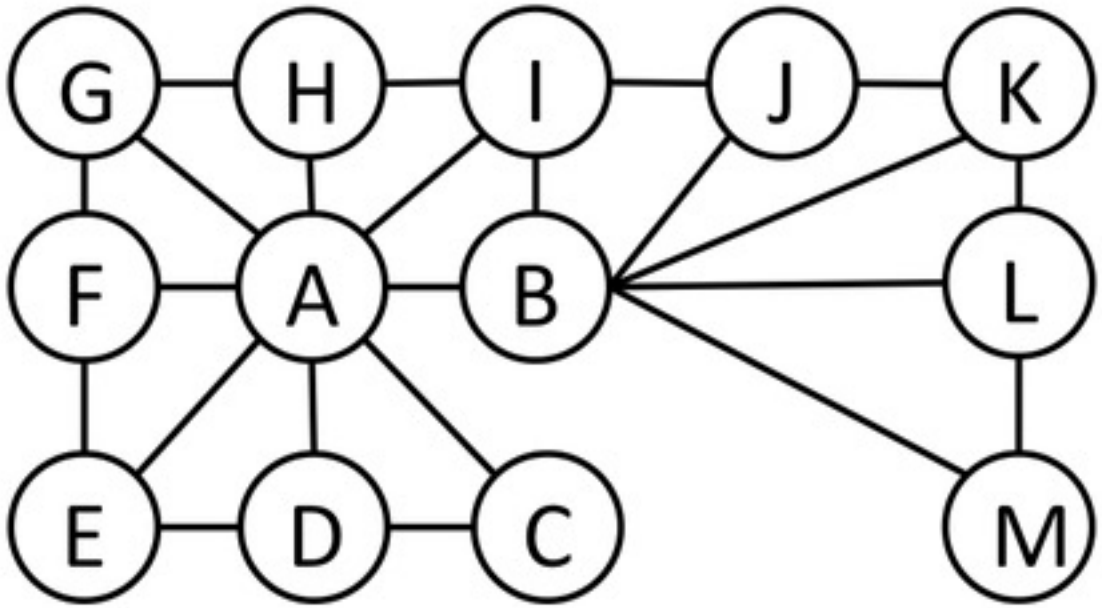
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



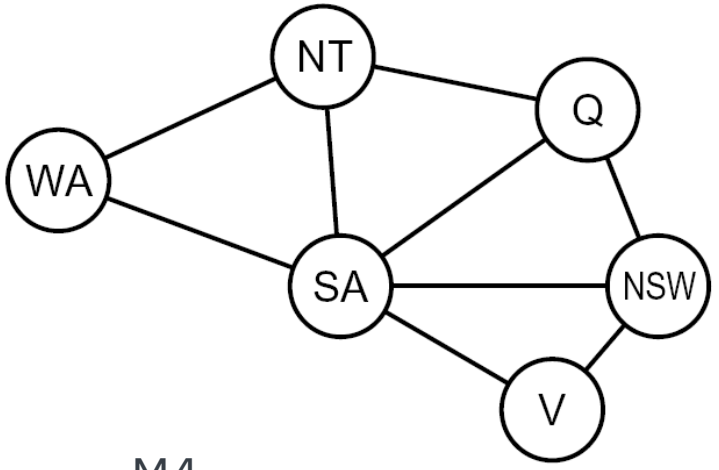
Cutset quiz

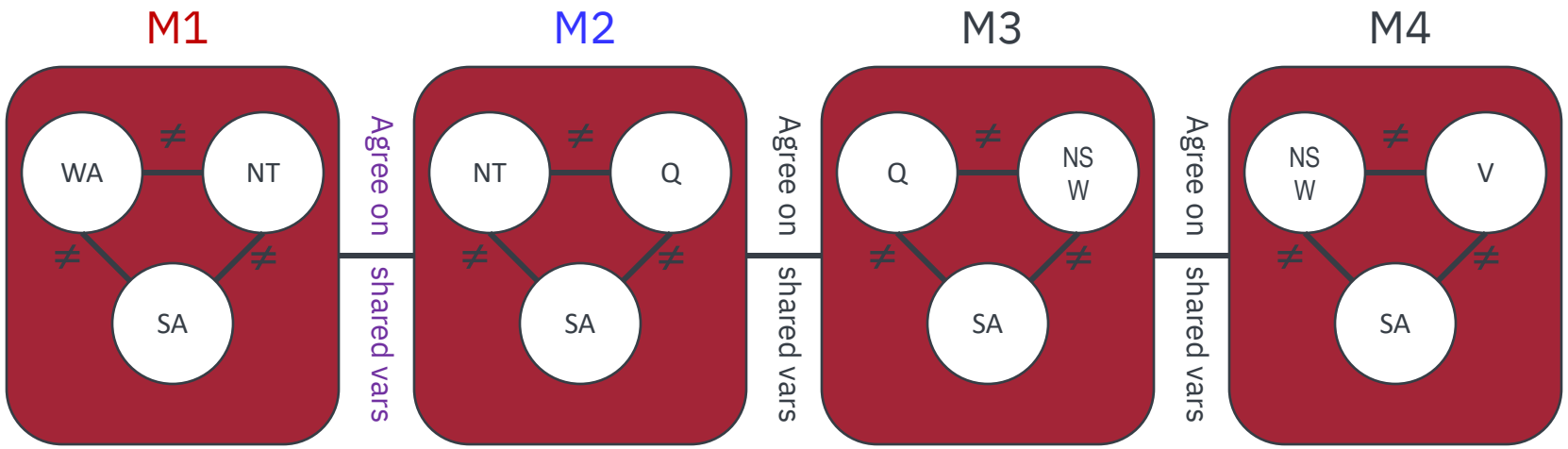
- Find the smallest cutset for the graph below.



Tree decomposition

- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions





{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
...}

{(NT=r,SA=g,Q=b),
(NT=b,SA=g,Q=r),
...}

Agree: (M1,M2) ∈
{((WA=g,SA=g,NT=g), (NT=g,SA=g,Q=g)), ...}

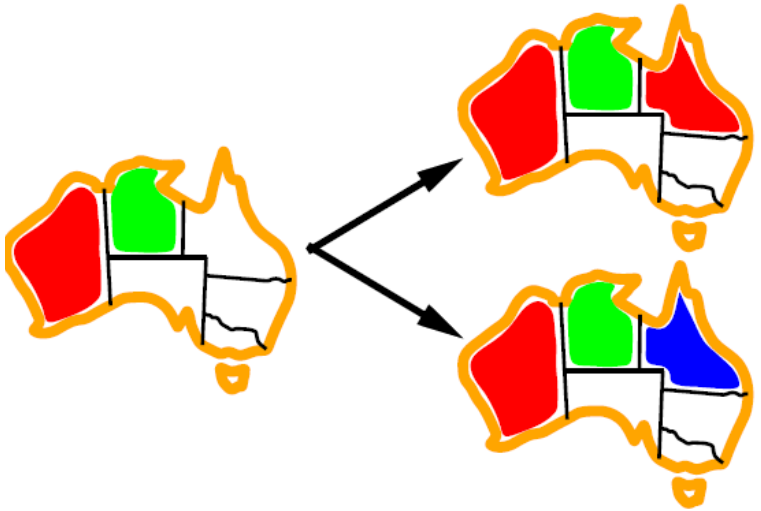
Ordering: minimum remaining values

- Variable ordering: Minimum Remaining Values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering

Ordering: least constraining value

- Value ordering: Least Constraining Value (LCV)
  - Given a choice of variable, choose the *least constraining value*
  - *I.e.*, the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (*E.g.*, rerunning filtering)



- Why least rather than most?
- Combining these ordering ideas makes 1000 queens feasible

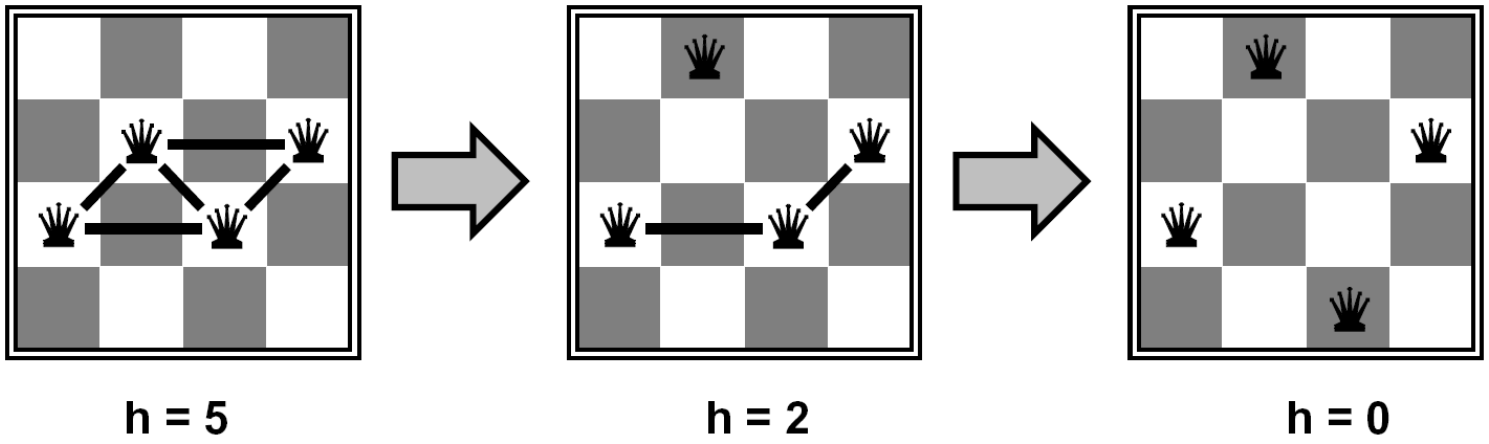
Iterative Improvement

Iterative algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.

- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - *I.e.*, hill climb with h(n) = total number of violated constraints

Example: 4-Queens

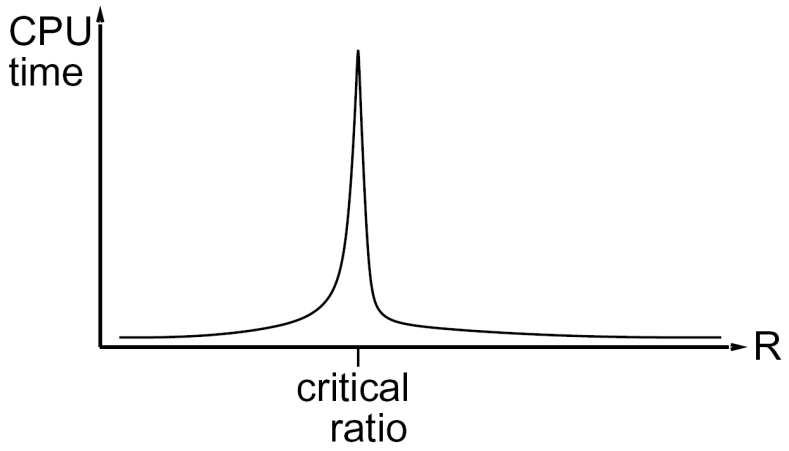


- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

Performance of min-conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (*e.g.*, n = 10,000,000)!
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
- Basic solution: backtracking search
- Speed-ups:
  - Ordering
  - Filtering
  - Structure
- Iterative min-conflicts is often effective in practice

STEVENS
INSTITUTE OF TECHNOLOGY
1870

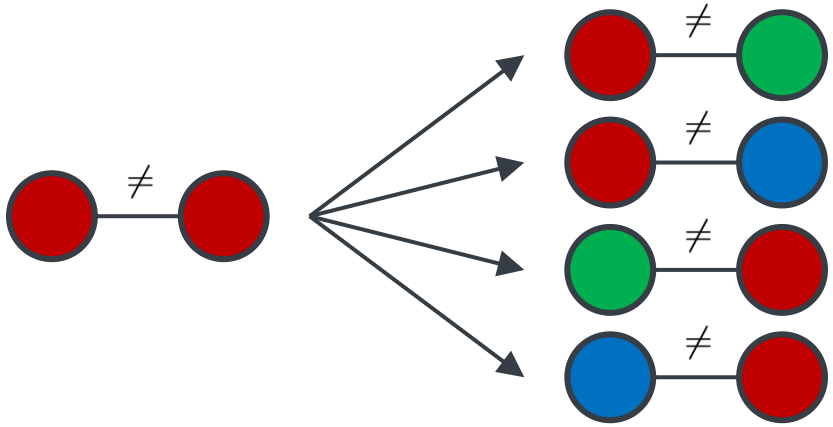
Local Search

Chapter 4

Hill climbing
Simulated annealing
Genetic algorithms

Local search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
- Local search: improve a single option until you can't make it better (no fringe!)
- New successor function: local changes

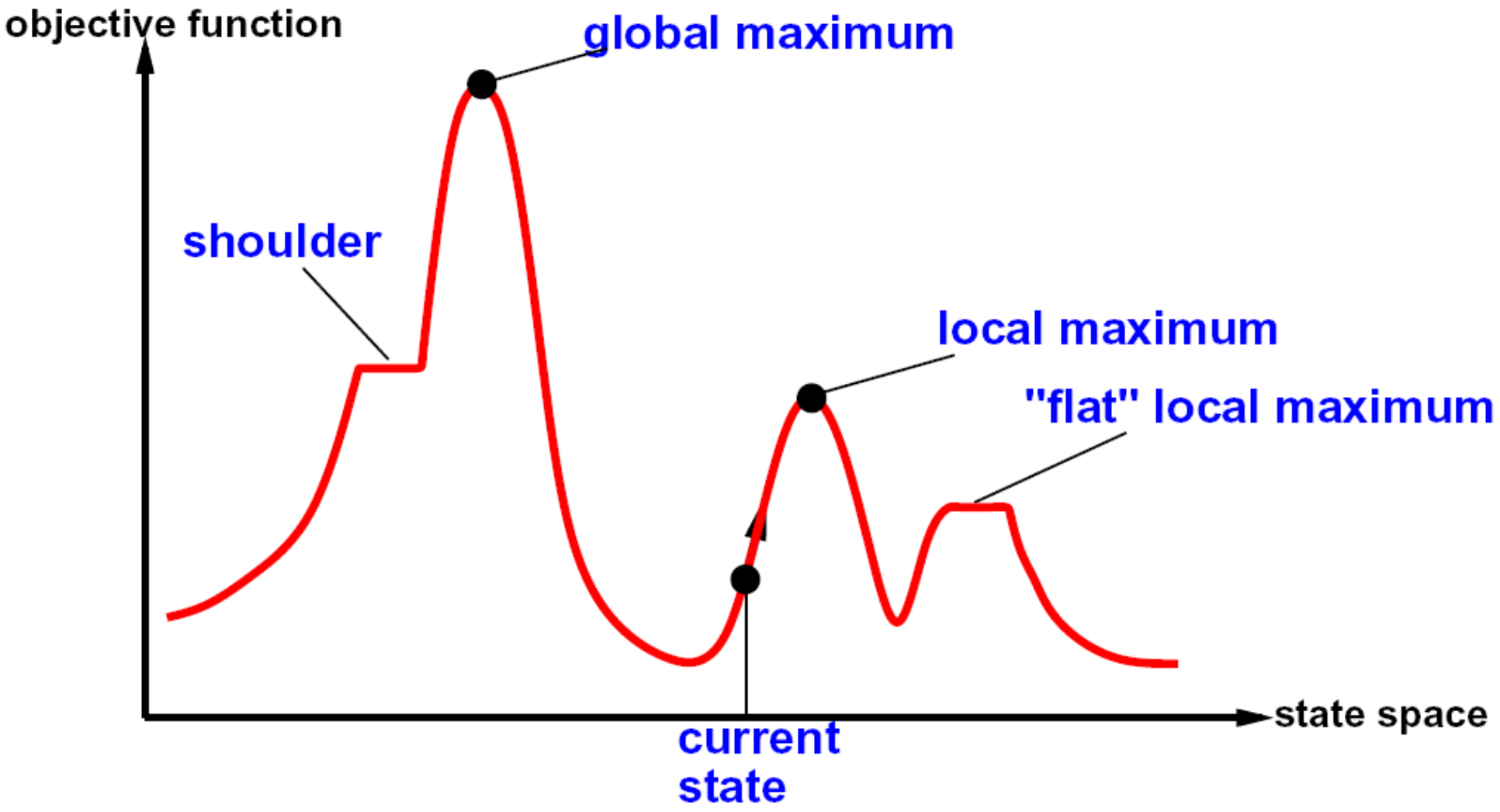


- Generally much faster and more memory efficient (but incomplete and suboptimal)

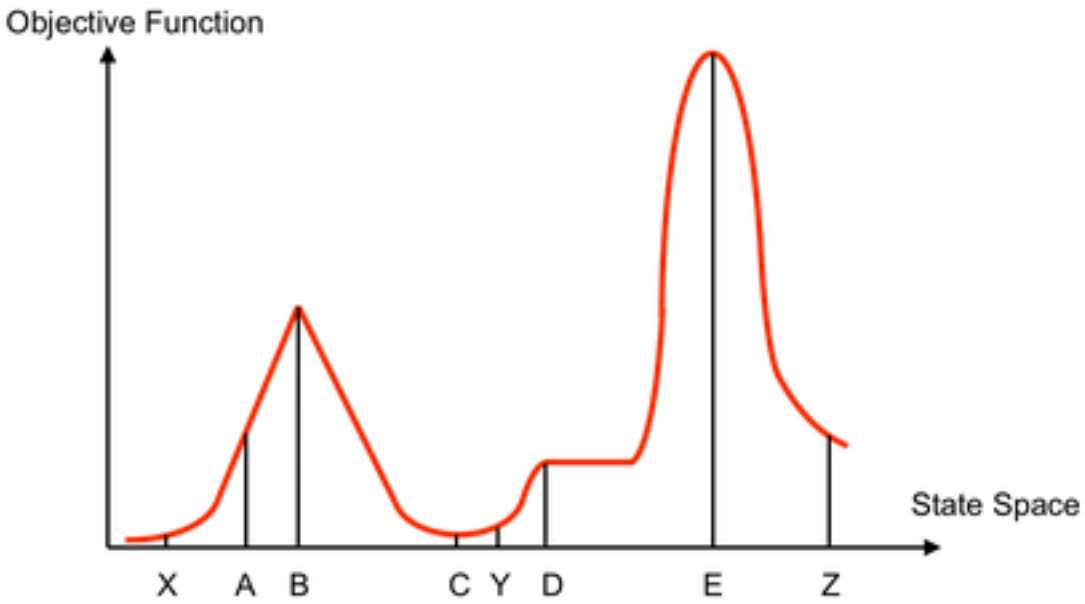
Hill climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit
- What's bad about this approach?
  - Complete?
  - Optimal?
- What's good about it?

Hill climbing diagram



Hill climbing quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

Simulated annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

Simulated annealing

- Theoretical guarantee:
  - Stationary distribution: $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough, will converge to optimal state!

- Is this an interesting guarantee?

- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways

Genetic algorithms

- Populations are encoded into a representation which allows certain operations to occur.
- An encoded candidate solution is an individual.
- Each individual has a fitness.
  - Numerical value associated with its quality of solution.
- A population is a set of individuals.
- Populations change over generations by applying operators to them.
  - Operations: selection, mutation, crossover

Typical genetic algorithm

1. Initialize: Population P←N random individuals
2. Evaluate: For each x in P, compute fitness(x)
3. Loop
   - For i=1 to N
     - **Select** 2 parents each with probability proportional to fitness scores
     - **Crossover** the 2 parents to produce a new samples (child)
     - With some small probability **mutate** child
     - Add child to population
   - Until some child is fit enough or you get bored
4. Return best child in the population according to fitness function

Selection

- Fitness proportionate selection
  - Can lead to overcrowding

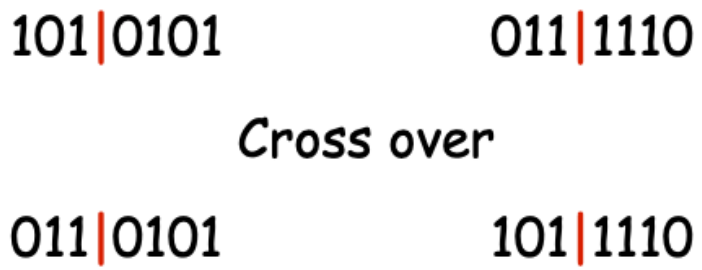
$$P(i) = \frac{fitness(i)}{\sum_j fitness(j)}$$

- Tournament selection
  - Pick i, j at random with uniform probability
  - With probability p select fitter one
- Rank selection
  - Sort all by fitness
  - Probability of selection is proportional to rank
- Softmax (Boltzmann) selection:

$$P(i) = \frac{e^{fitness(i)}}{\sum_j e^{fitness(j)}}$$

Crossover

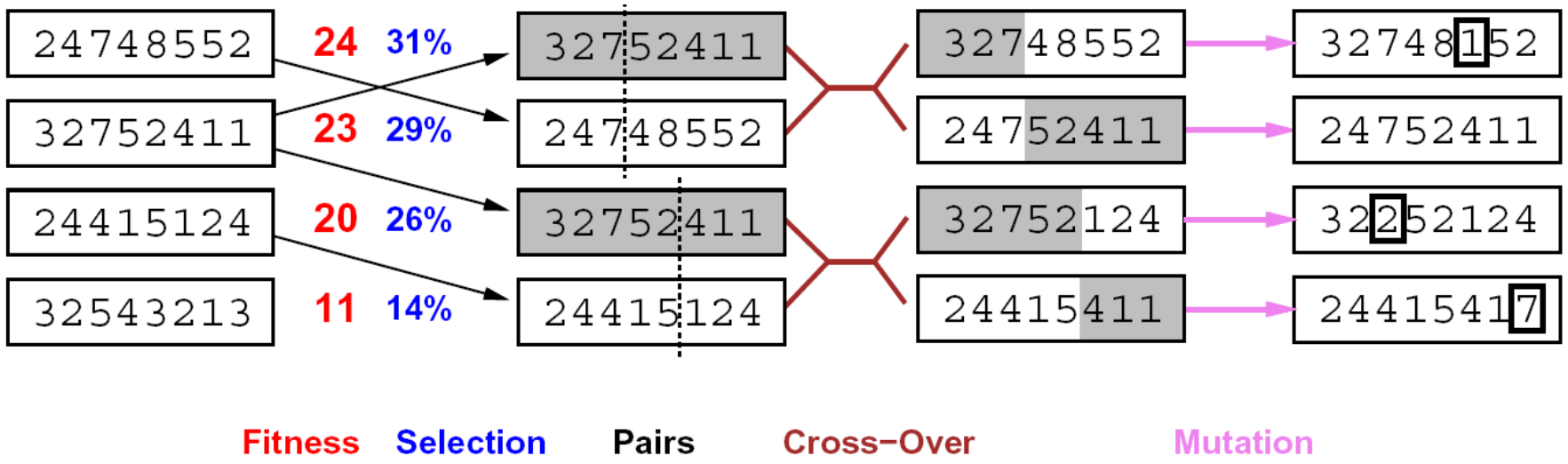
- Combine parts of individuals to create new ones.
- For each pair, choose a random crossover point.
  - Cut the individuals there and swap the pieces



Mutation

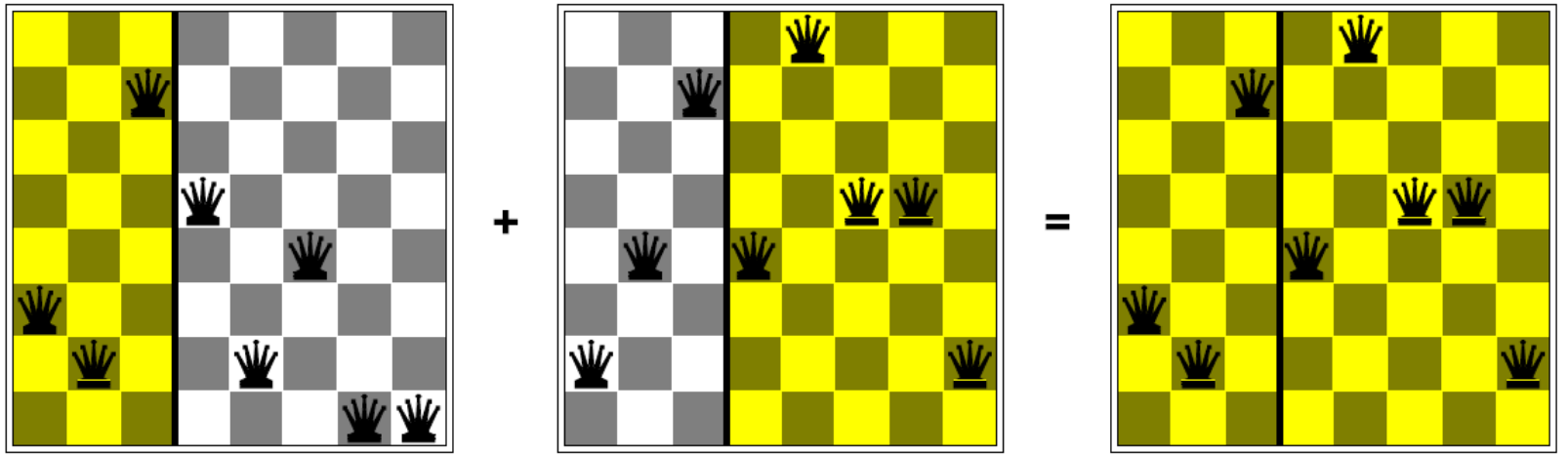
- Mutation generates new features that are not present in original population.
- Typically means flipping a bit in the string.
- Can allow mutation in all individuals or just in new offspring.

Genetic algorithms



- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety
- Possibly the most misunderstood, misapplied (and even maligned) technique around

Example: N-Queens



- Why does crossover make sense here?
- When wouldn’t it make sense?
- What would mutation be?
- What would a good fitness function be?

STEVENS
INSTITUTE OF TECHNOLOGY
1870

Adversarial Search

Chapter 5


KASPAROV VS
DEEP BLUE
the rematch
1:57
1:58


KBA
Korea Baduk Association
Google DeepMind
Challenge Match
8 - 15 March 2016
AlphaGo
ALPHAGO
00:12:30
LEE SEDOL
00:34:31
AlphaGo
Lee Sedol

Generalizing search problem

- So far: our search problems have assumed agent has complete control of environment
  - State does not change unless the agent changes it.
  - All we need to compute is a single path to a goal state.
- Assumption not always reasonable
  - Stochastic environment (e.g., the weather, traffic accidents).
  - Other agents whose interests conflict with yours (Focus of this module)
  - Search can find a path to a goal state, but the actions might not lead you to the goal as the state can be changed by other agents.

Adversarial 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)?

- Want algorithms for calculating a **strategy (policy)** which recommends a move from each state

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

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

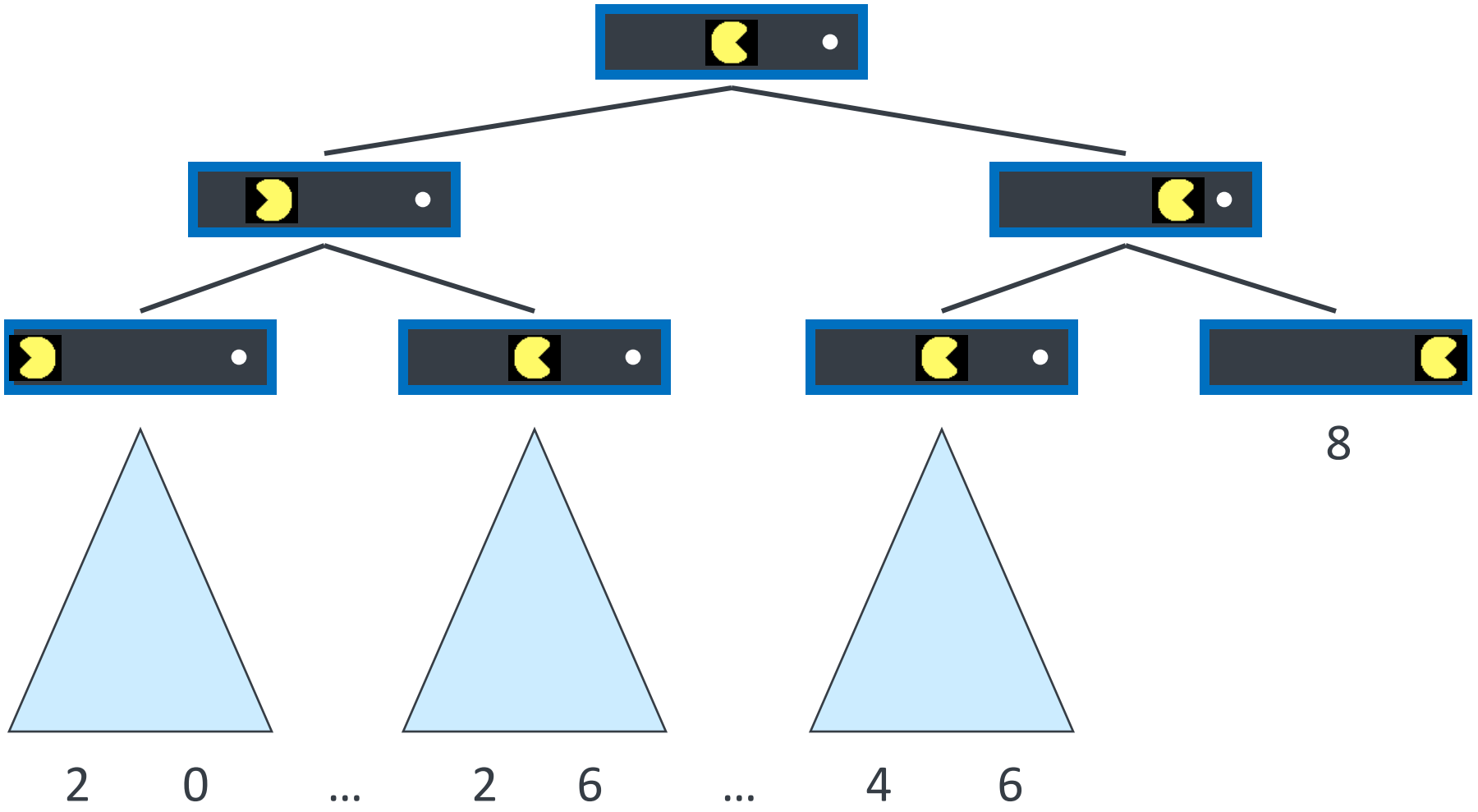
Zero-sum games

- 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

- General games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

Adversarial Search

Single-agent trees

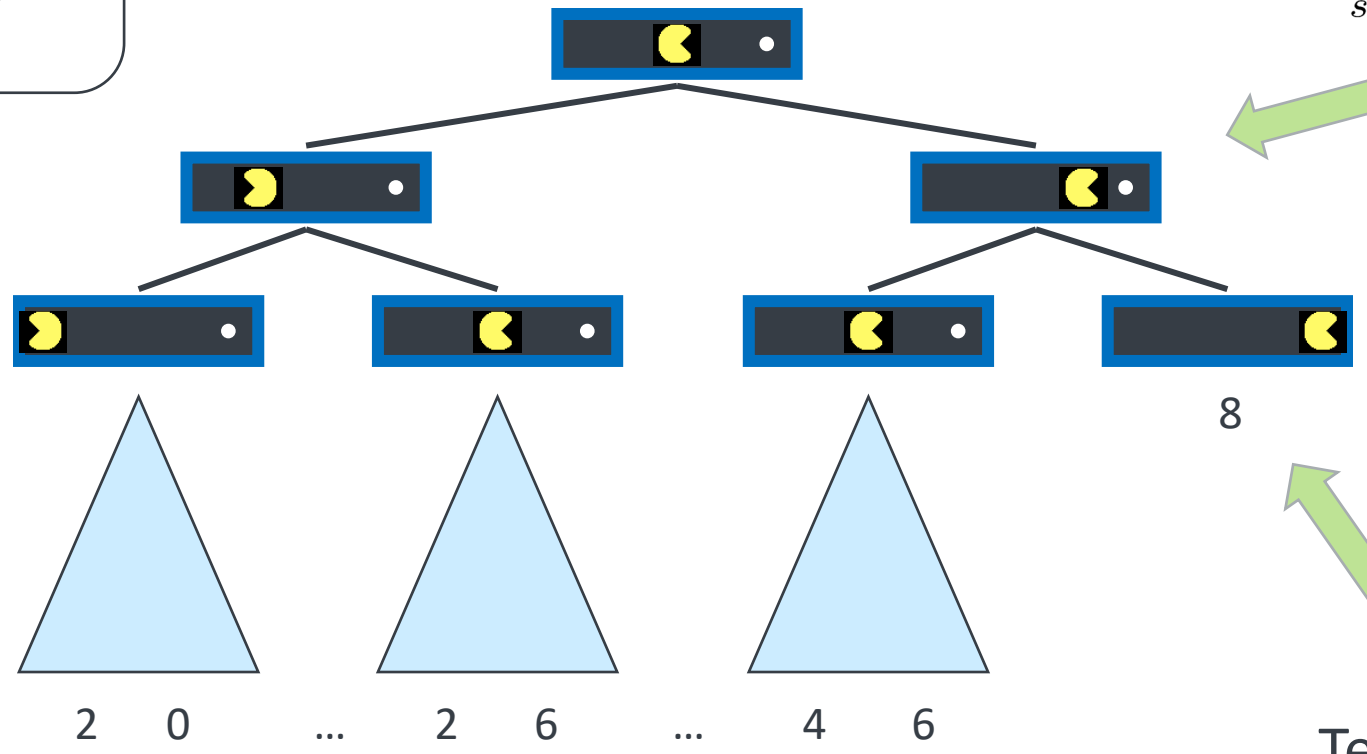


Value of a state

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

Non-Terminal States:

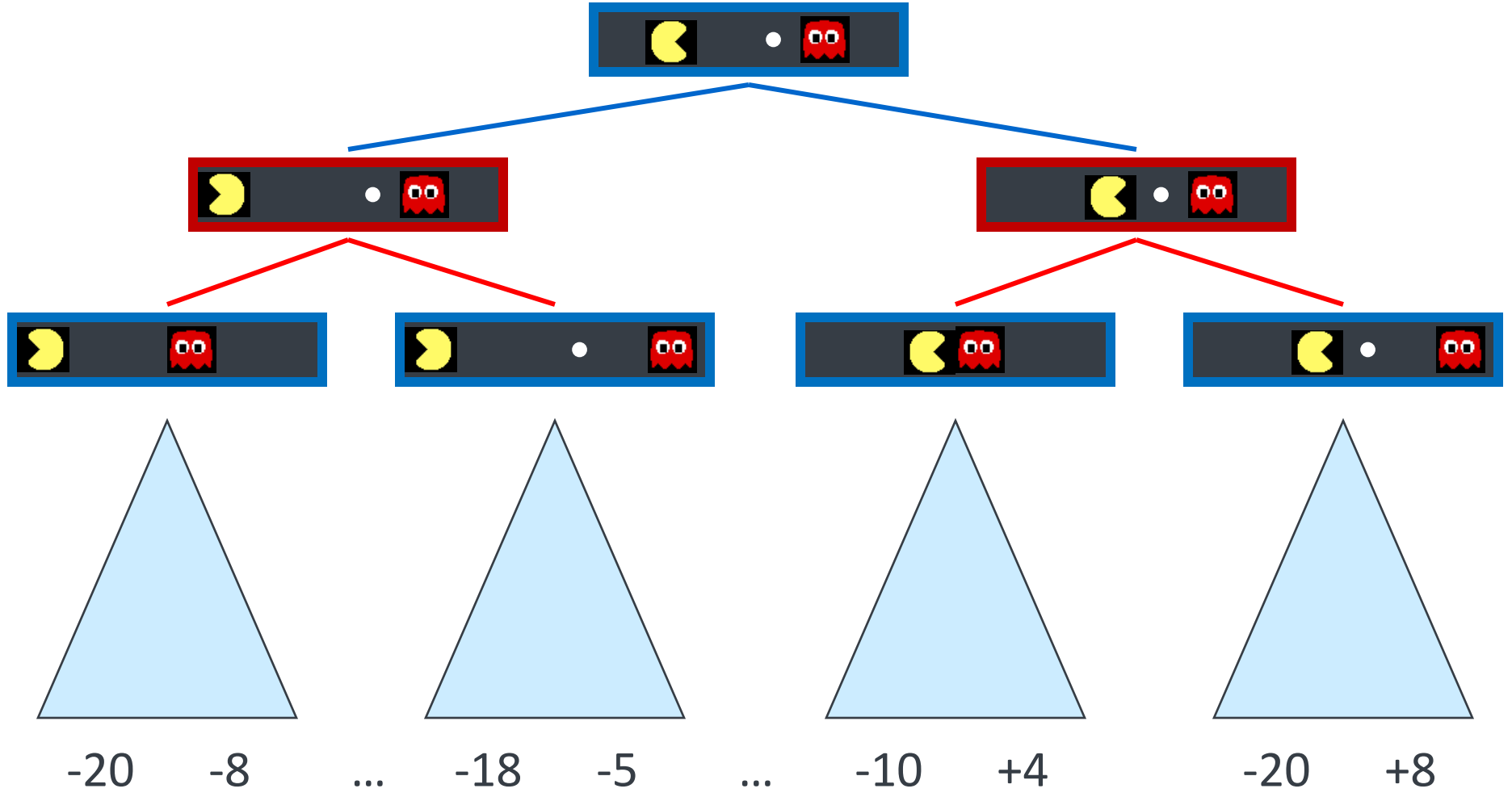
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial game trees



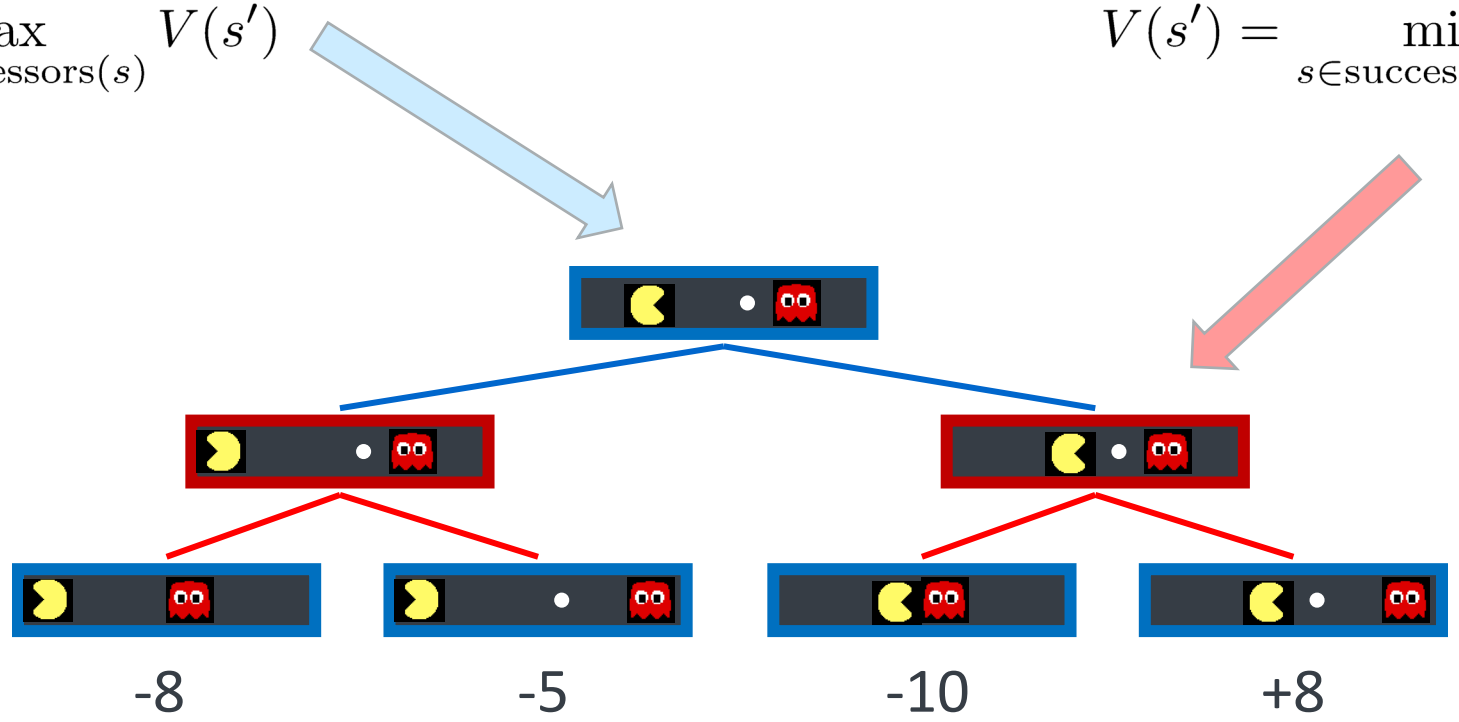
Minimax values

States under agent's control:

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

States under opponent's control:

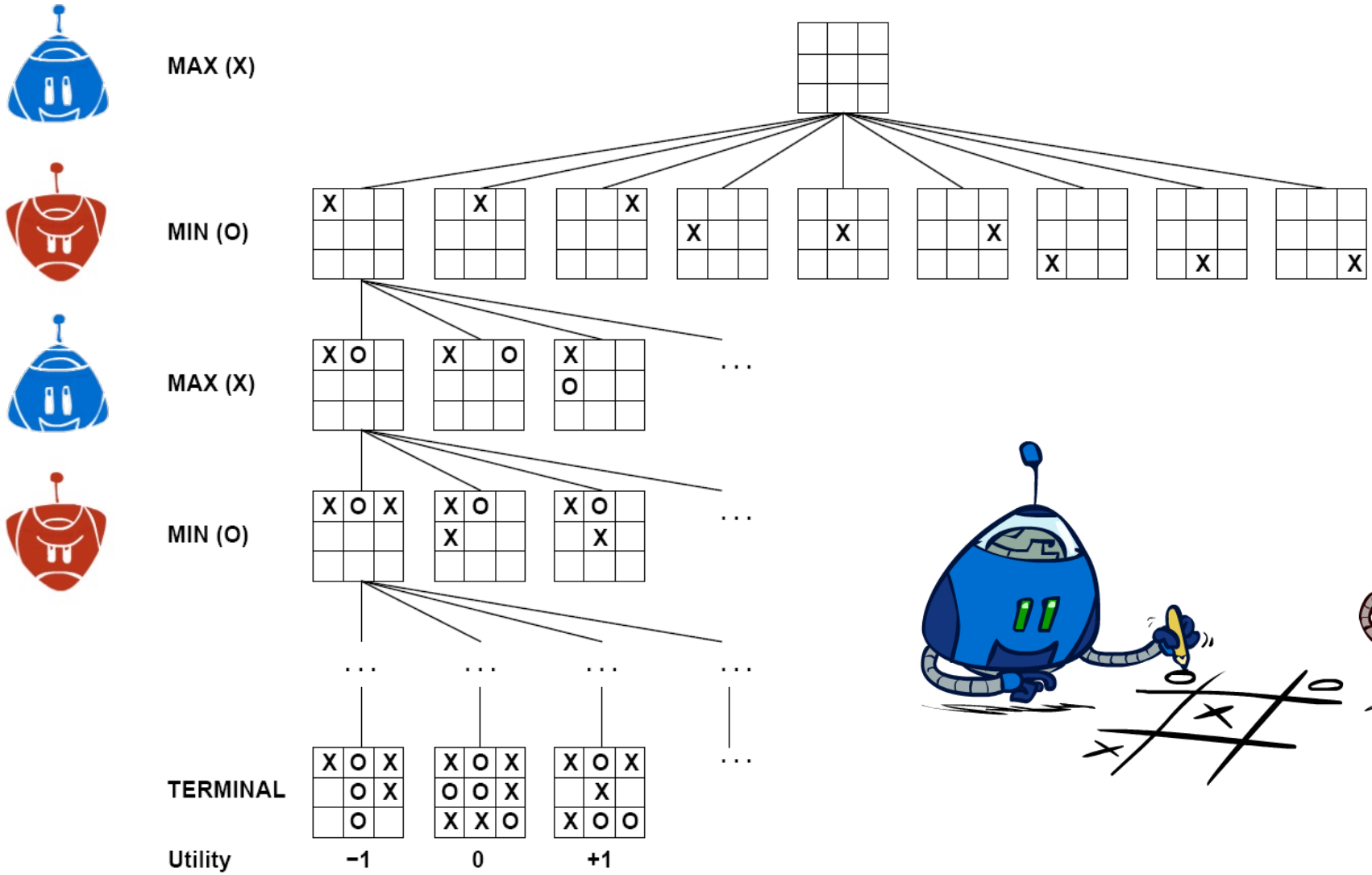
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

Tic-tac-toe game tree



STEVENS
INSTITUTE OF TECHNOLOGY
1870

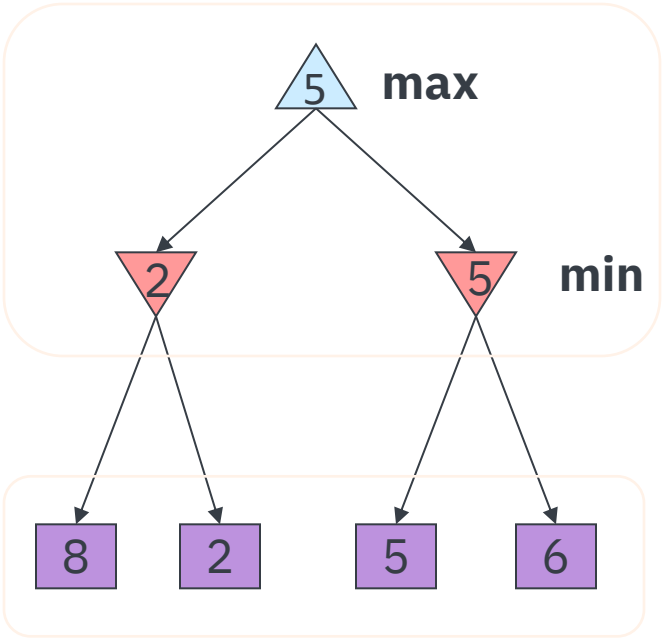
15 min. break

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

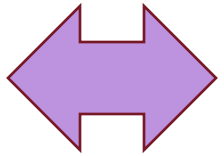
Minimax values: computed recursively



Terminal values: part of the game

Minimax implementation

```
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
```

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

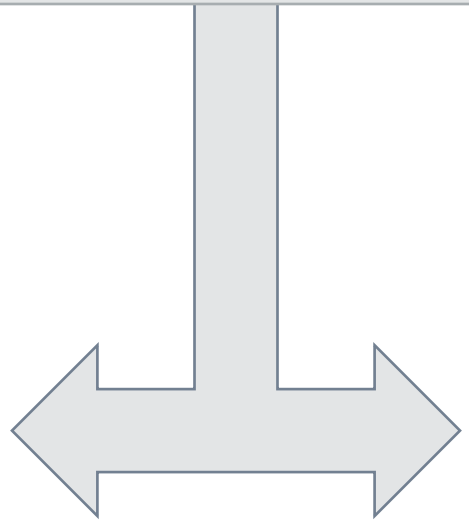
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax implementation (dispatch)

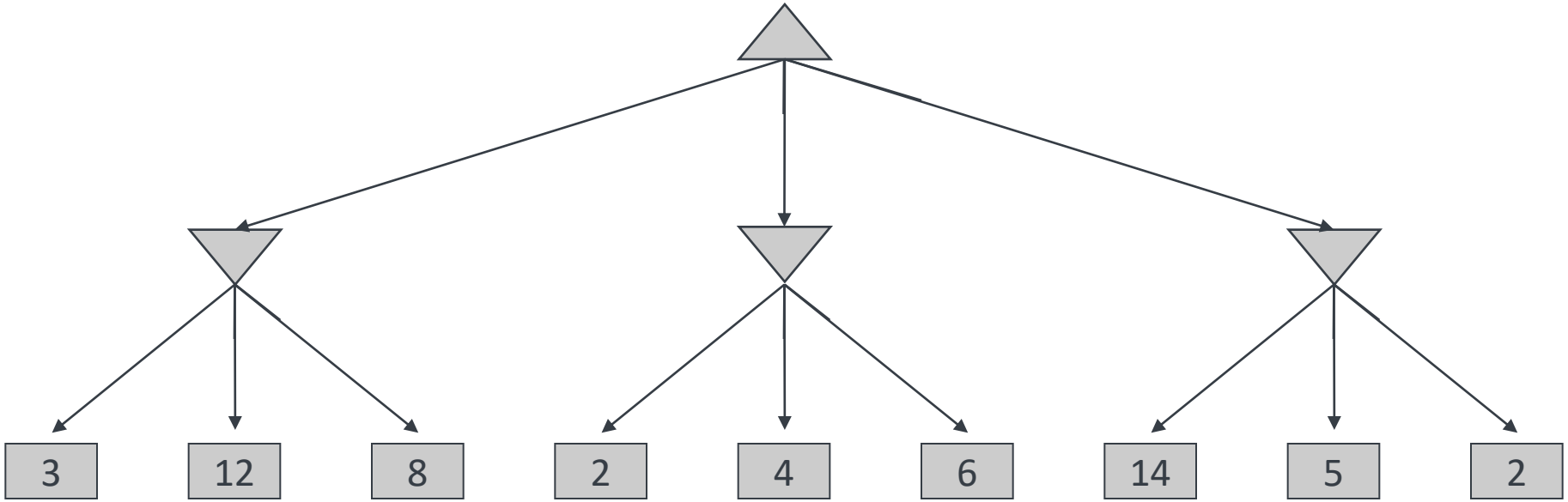
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
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

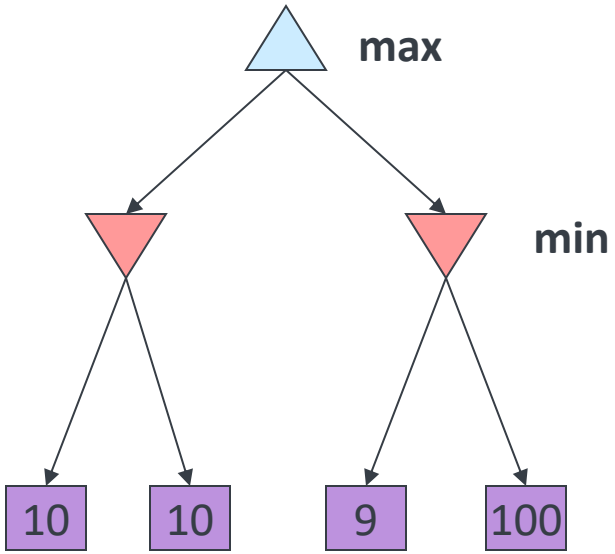


Minimax efficiency

- 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?

Minimax properties

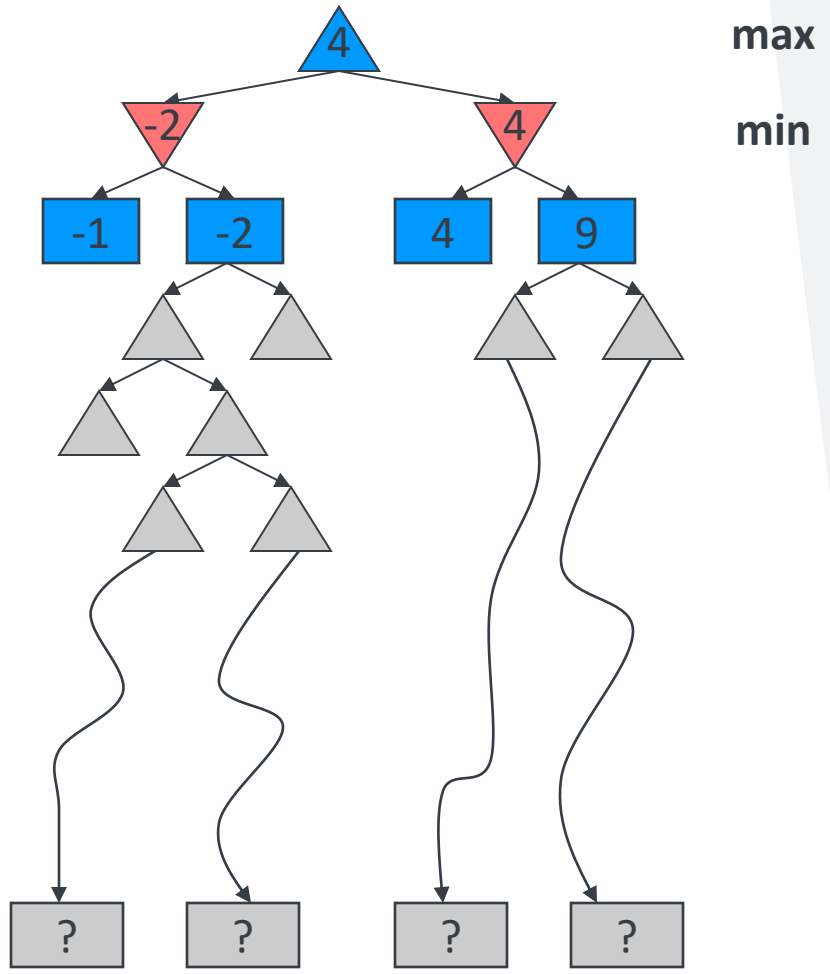


Optimal against a perfect player. Otherwise?

Resource Limits

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
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



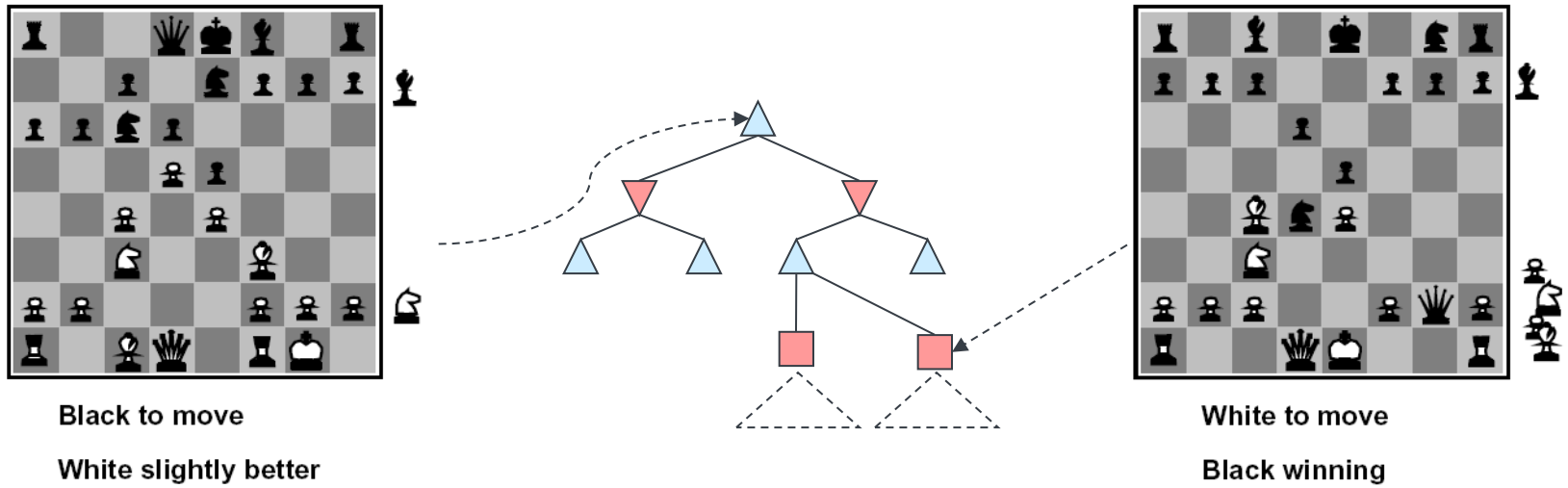
Depth matters

- 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.

Evaluation functions

Evaluation functions

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

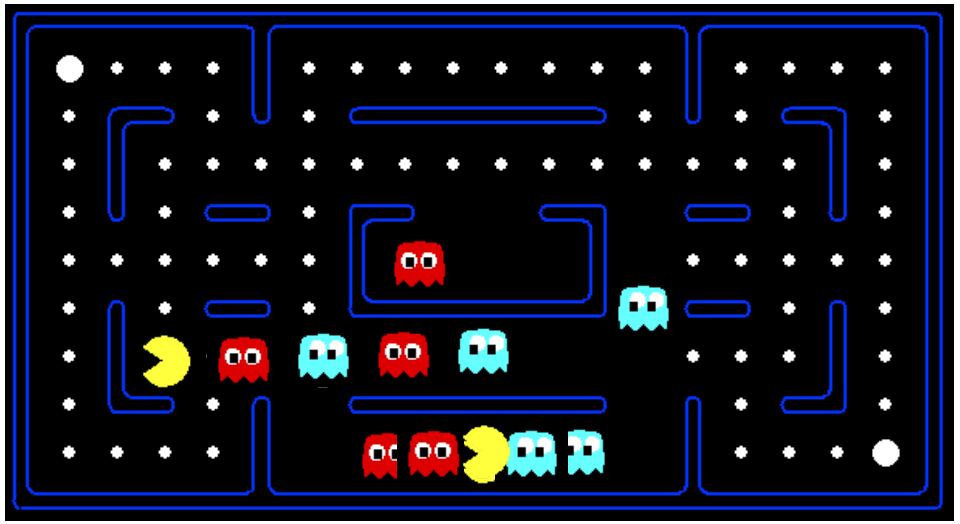


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

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

Evaluation for Pacman

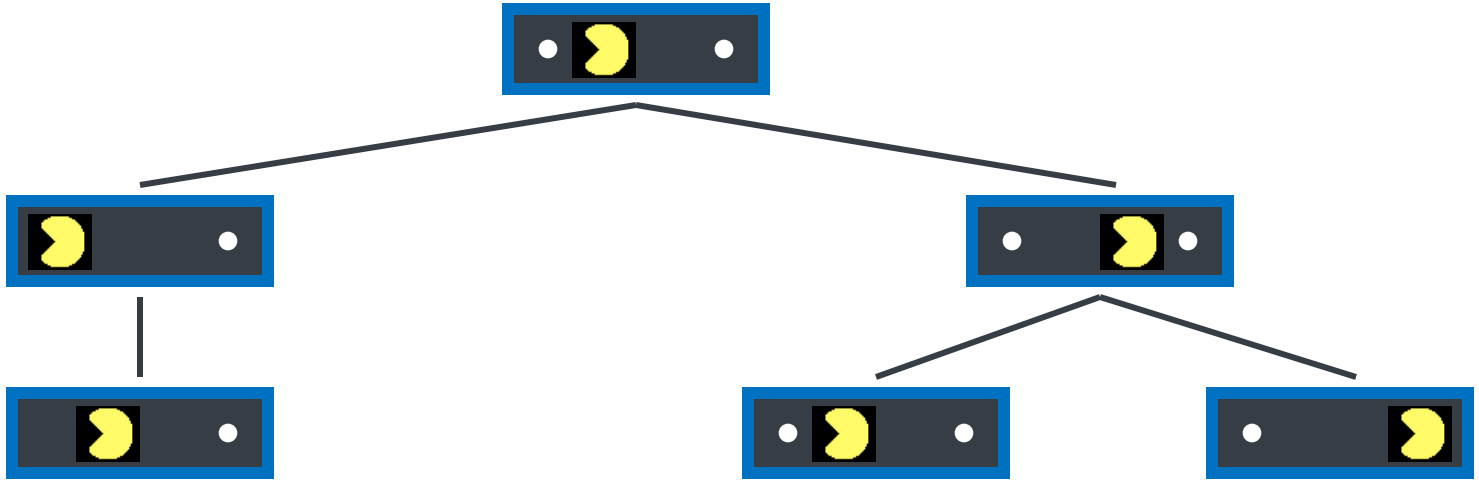


- Number of dots taken
- Distance to the ghosts
- ...

Why Pacman starves

What happens if we consider the points (i.e., how many dots the pacman can eat) only for our evaluation function?

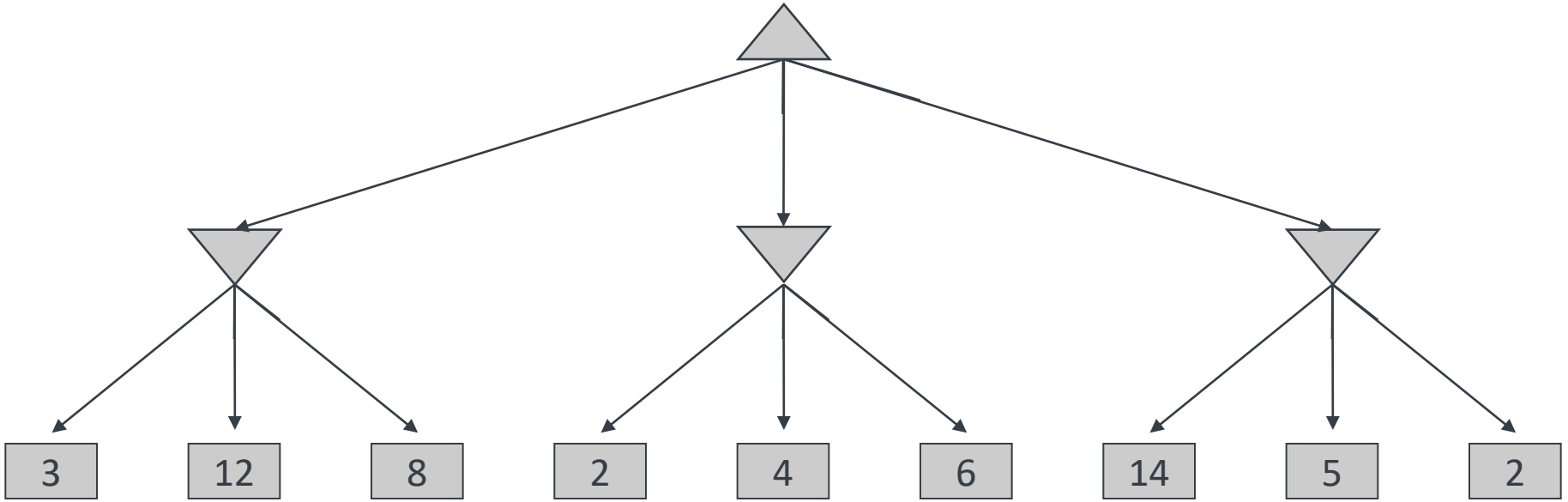
Why Pacman starves



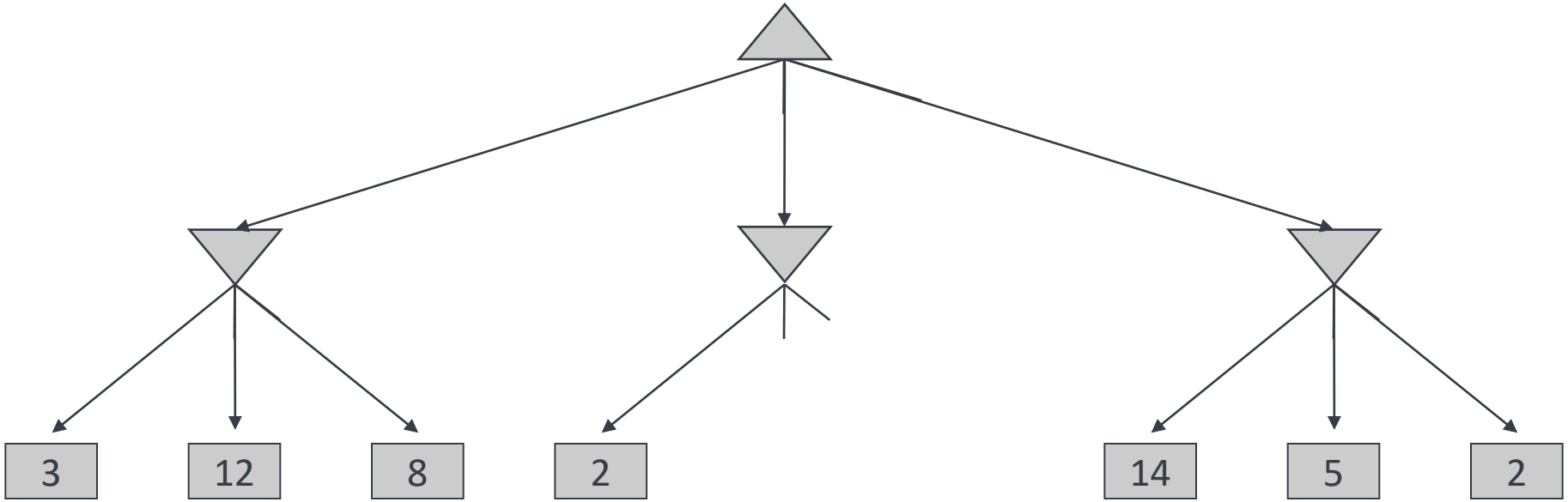
- A danger of replanning agents!
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting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

Game Tree Pruning

Minimax example

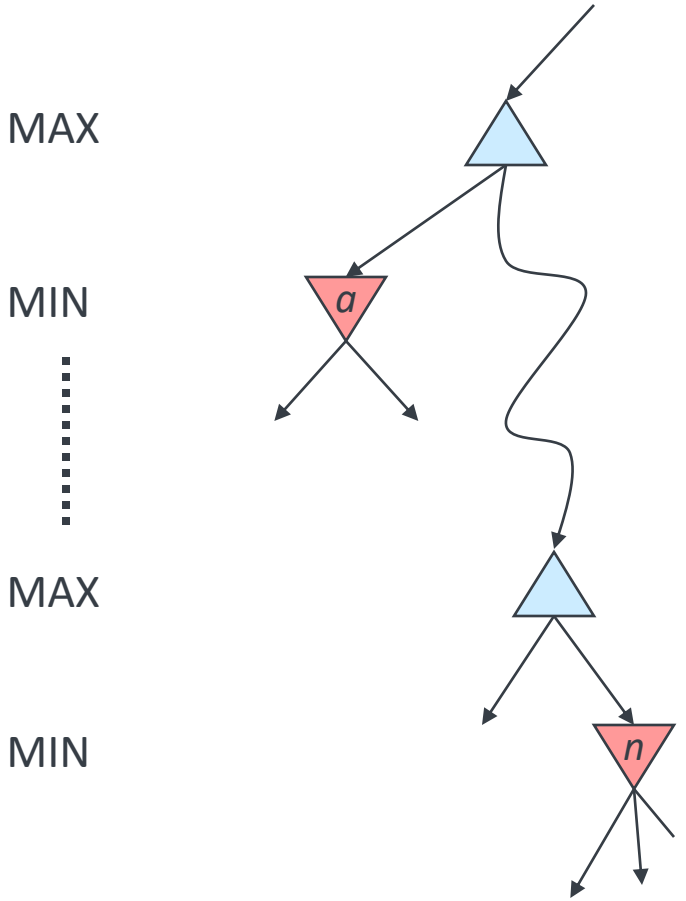


Minimax pruning



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 children's min is dropping
  - Who cares about *n*'s value? MAX
  - Let *α* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *α*, 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

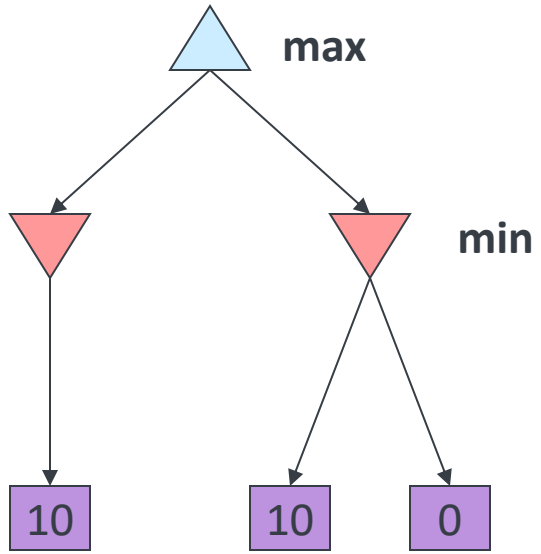
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

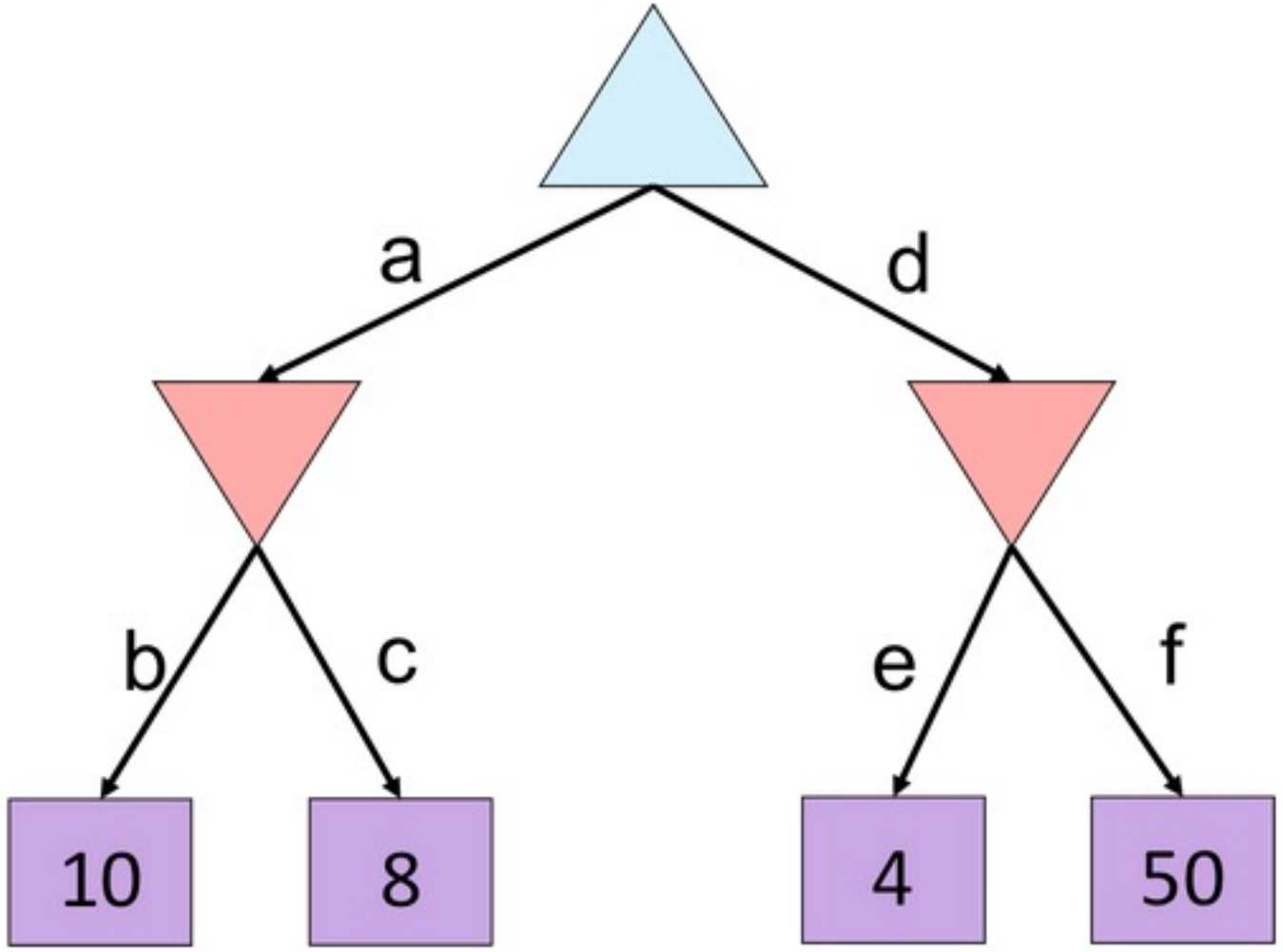
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Pruning Properties

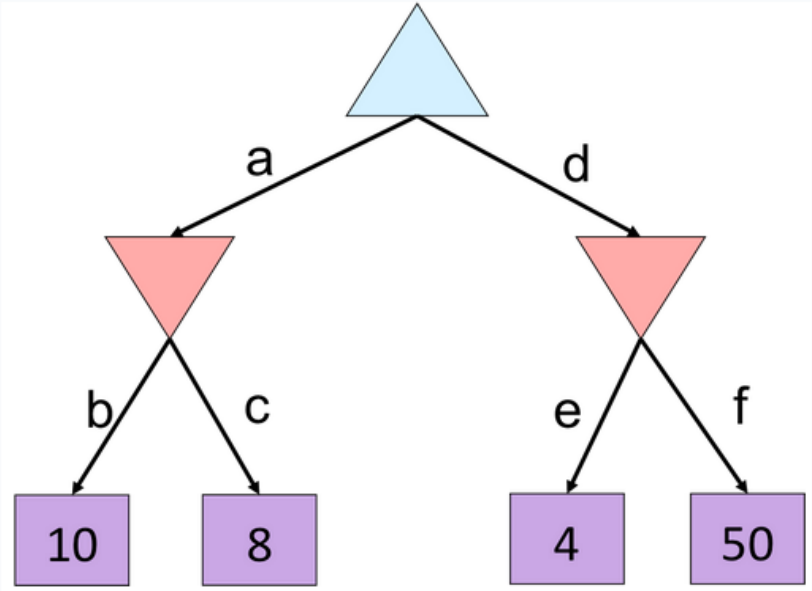
- This pruning has **no effect** on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- 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



Which edge is pruned?



a

b

c

d

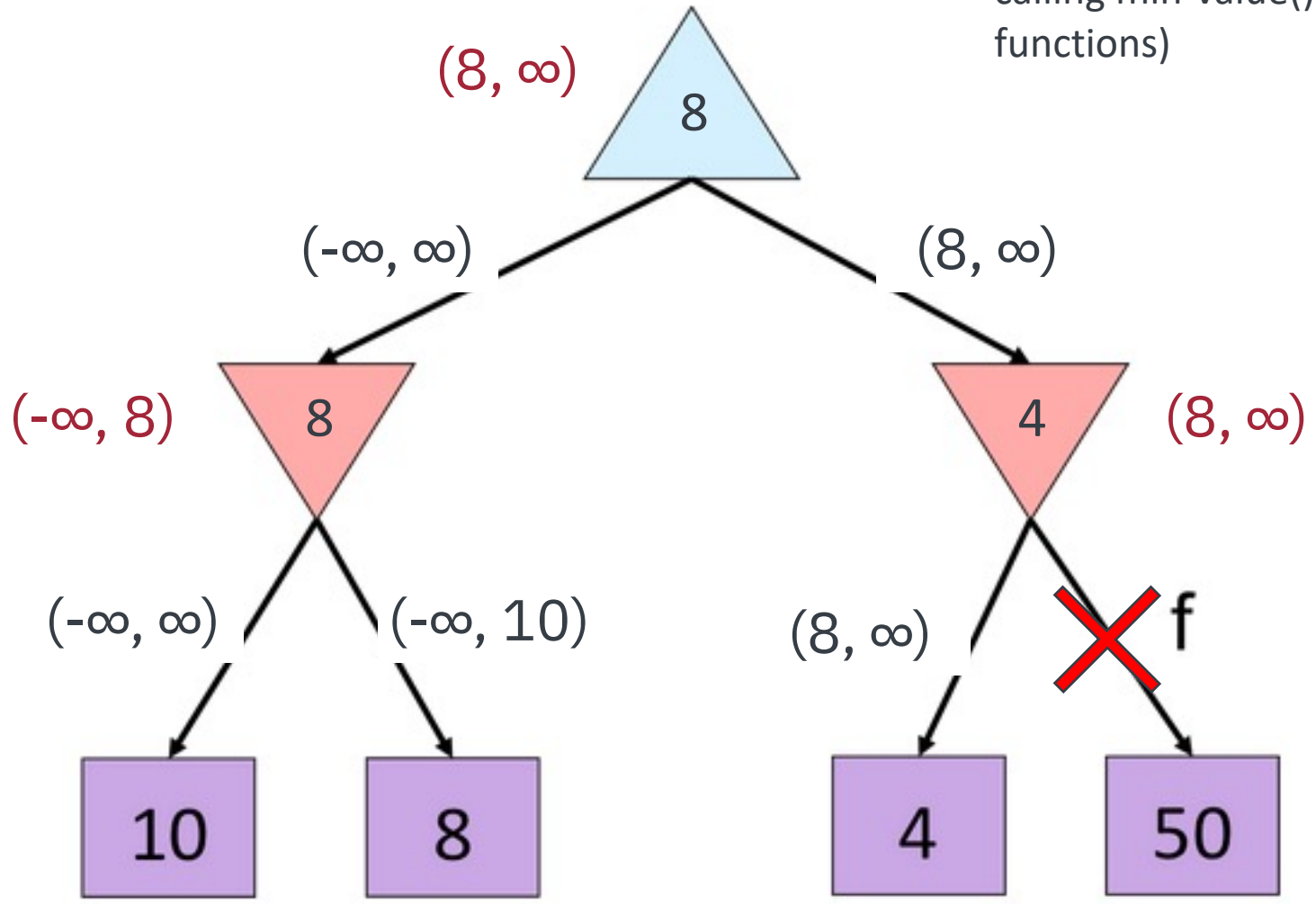
e

f

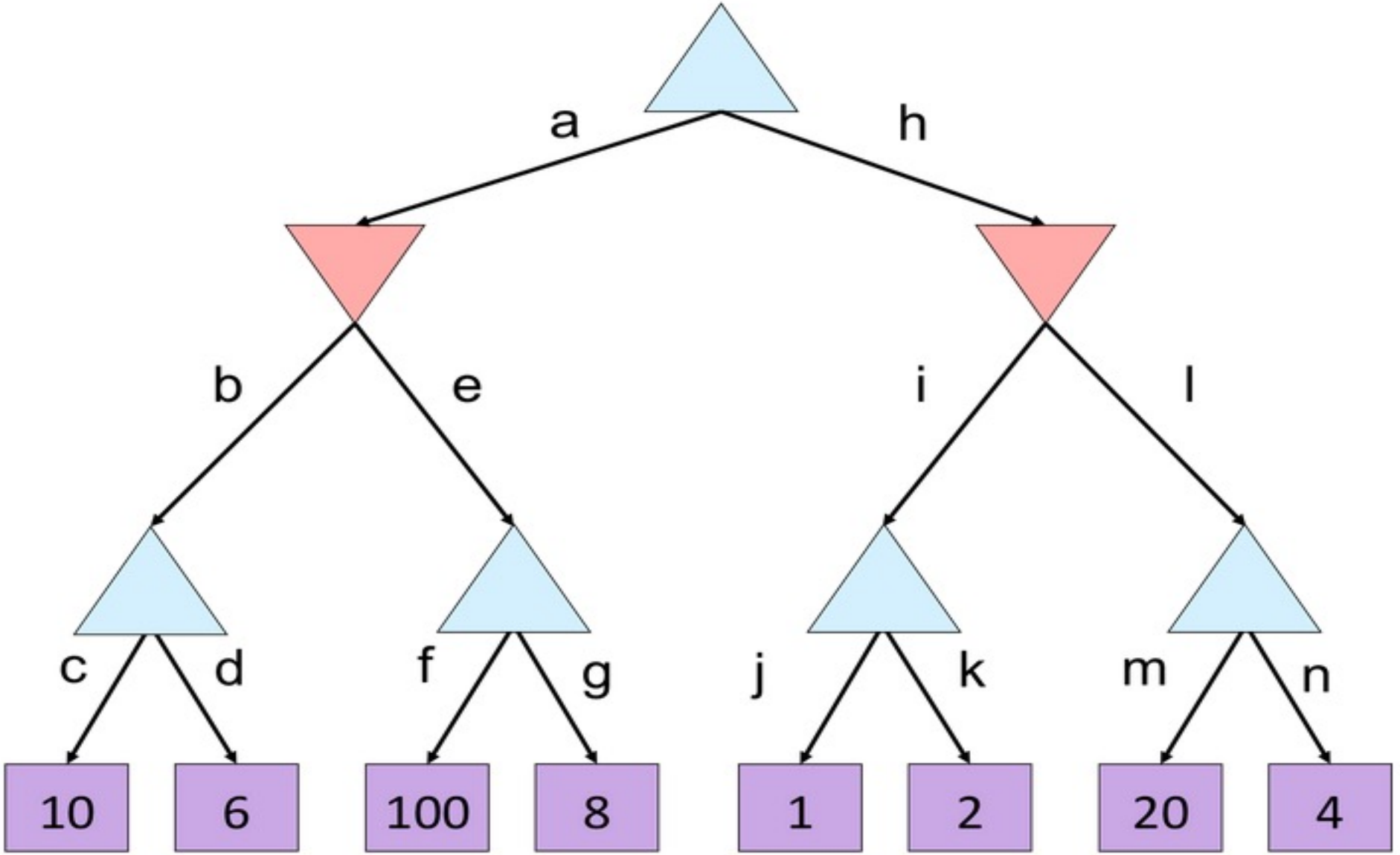
Alpha-Beta Quiz

(α, β) Alpha and Beta values passed down to the successor

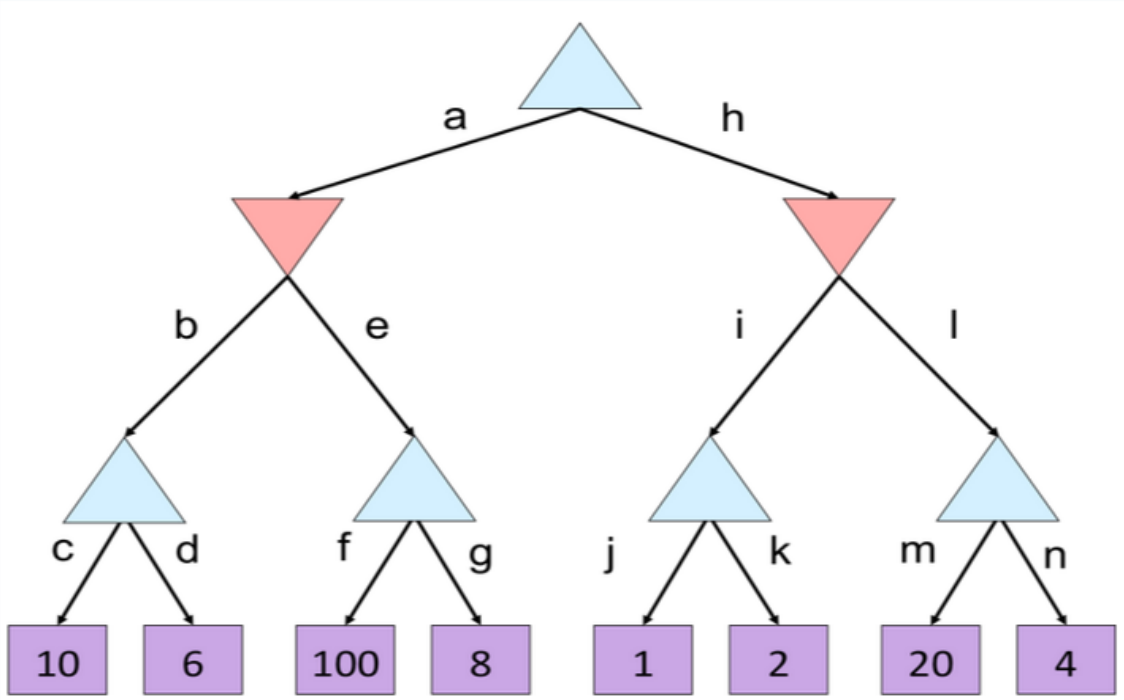
(α, β) The final alpha and beta values (after calling min-value() or max-value() functions)



Alpha-Beta Quiz 2



What is the alpha and beta values of the root at the end?



(100, 2)

(10, 2)

(10, ∞)

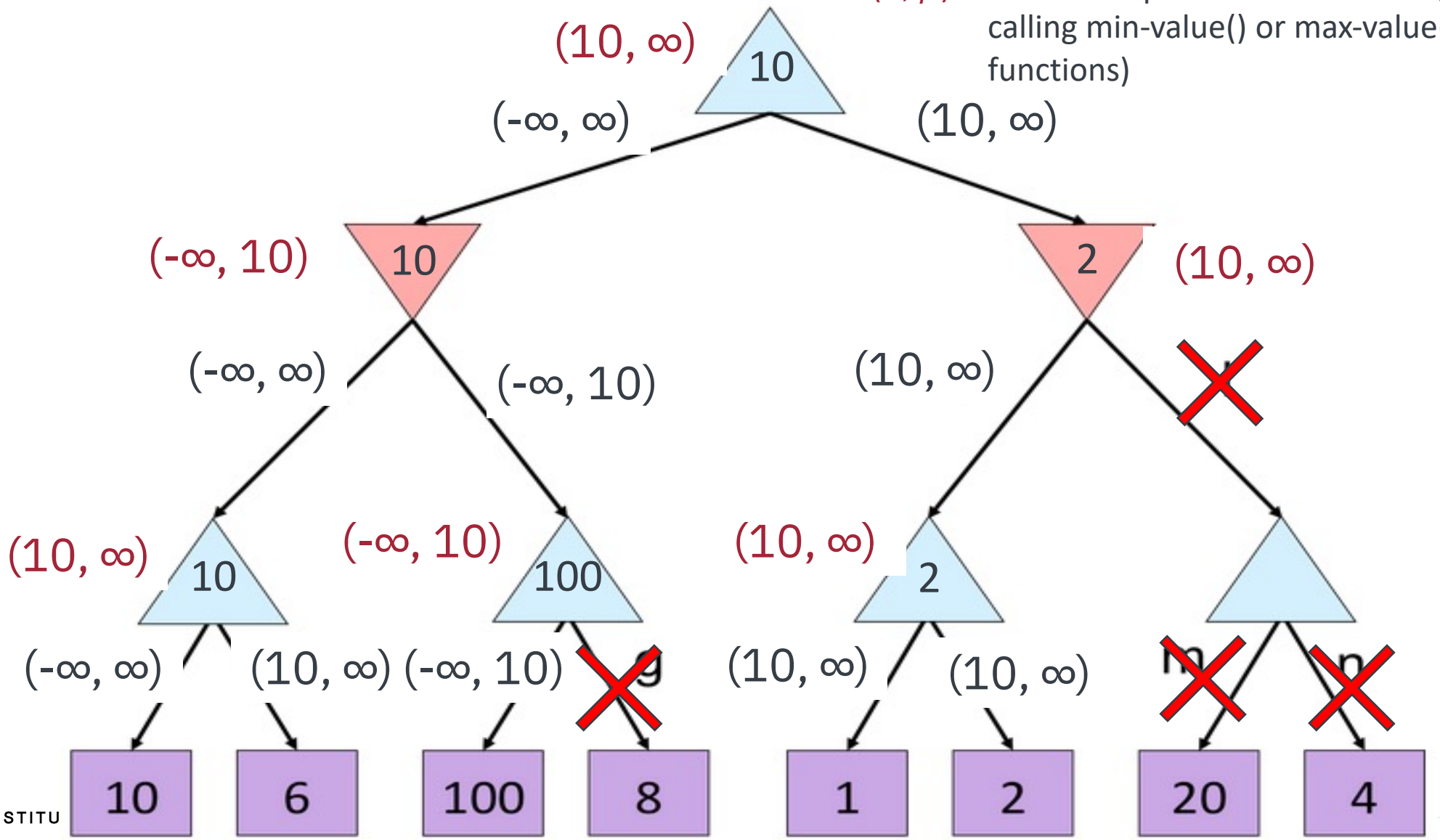
(∞, 10)

None of the above

Alpha-Beta Quiz 2

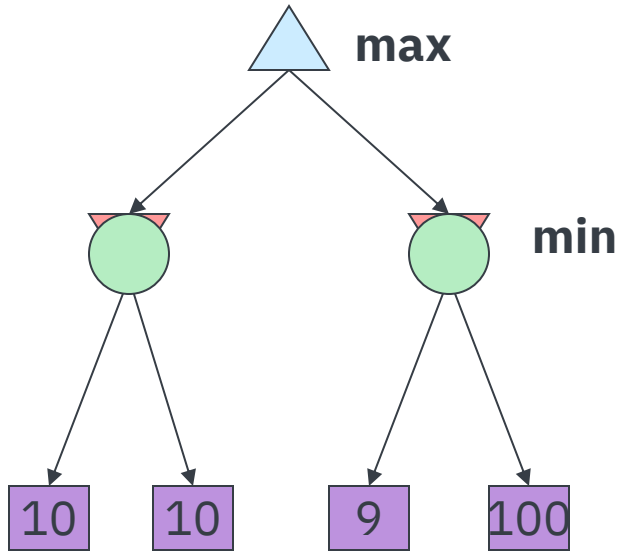
(α, β) Alpha and Beta values passed down to the successor

(α, β) The final alpha and beta values (after calling min-value() or max-value() functions)



Uncertain Outcomes

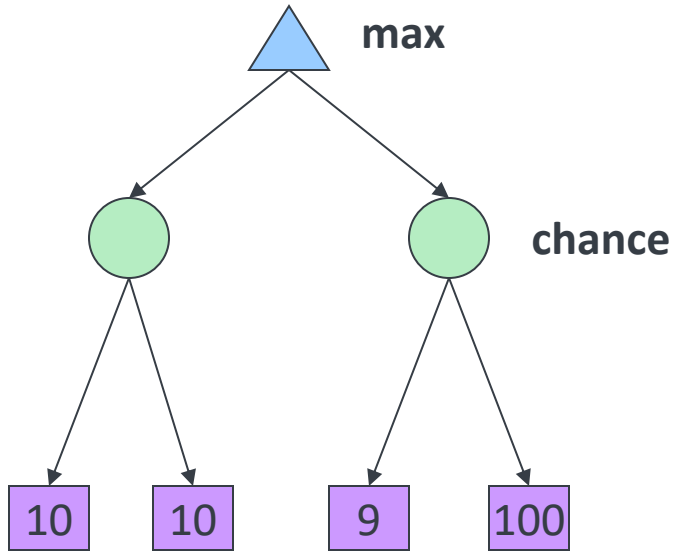
Worst-case vs. Average case



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
  - 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.**



Expectimax pseudocode

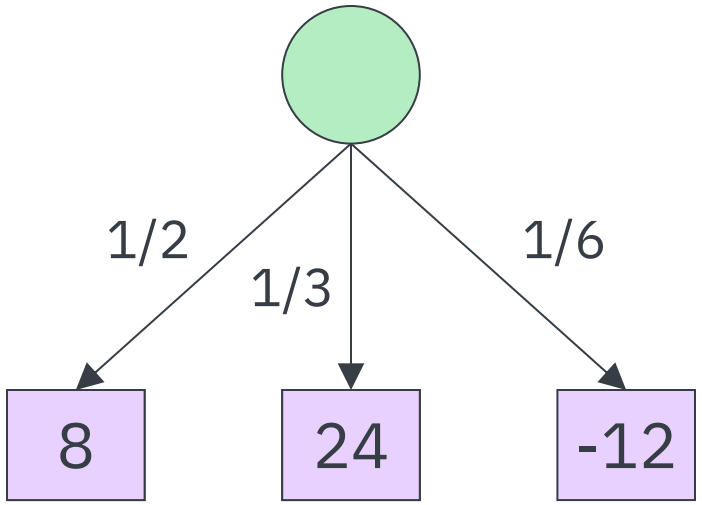
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

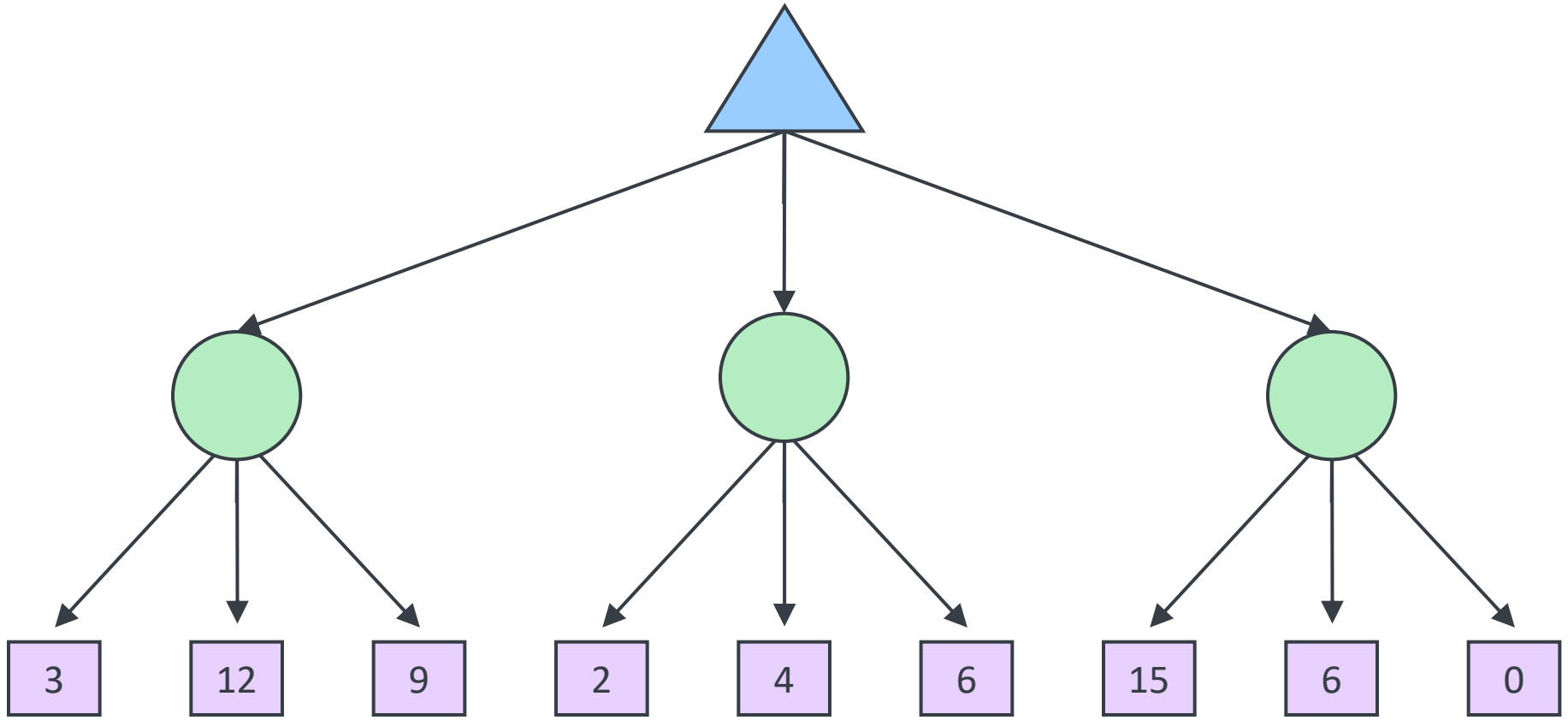
Expectimax pseudocode

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

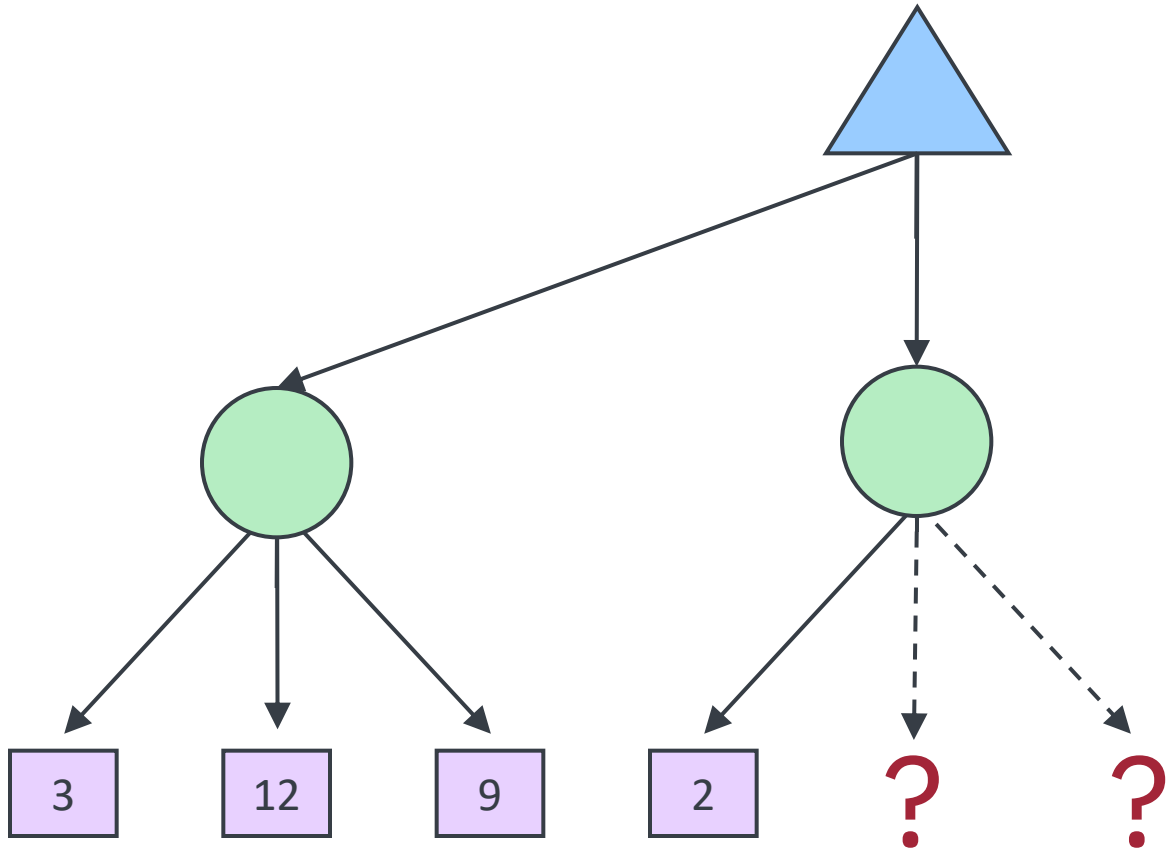


$$v = (1/2)\ (8) + (1/3)\ (24) + (1/6)\ (-12) = 10$$

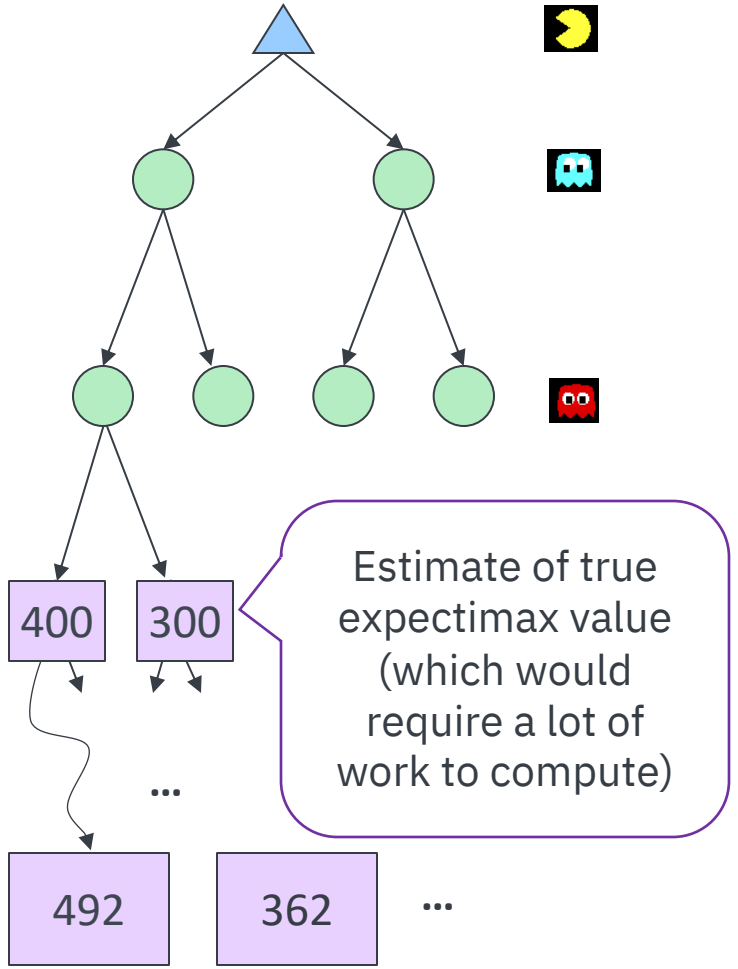
Expectimax example



Expectimax pruning?



Depth-limited expectimax



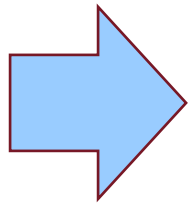
Probabilities

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

Reminder: expectations

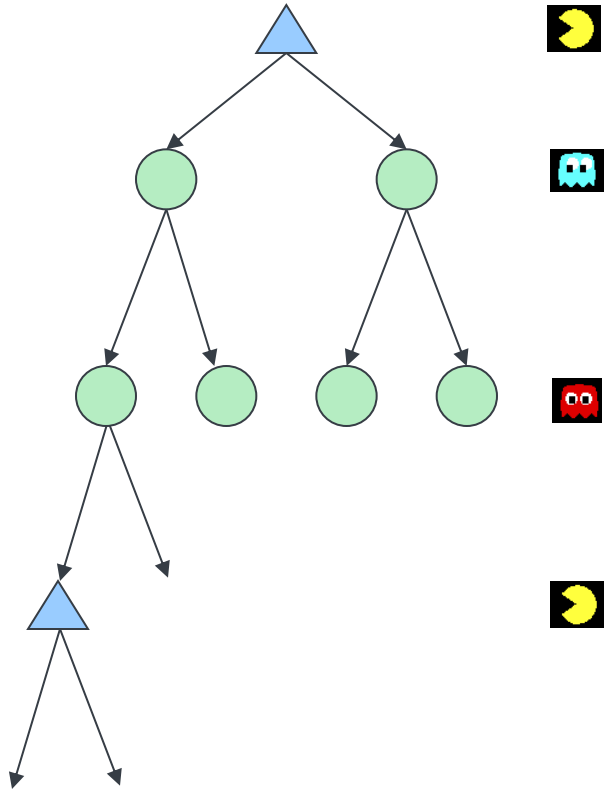
- The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes
- Example: How long to get to the airport?

Time:	20 min		30 min		60 min		
	x	+	x	+	x		35 min
Probability:	0.25		0.50		0.25		
	No traffic		Light traffic		Heavy traffic		

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!

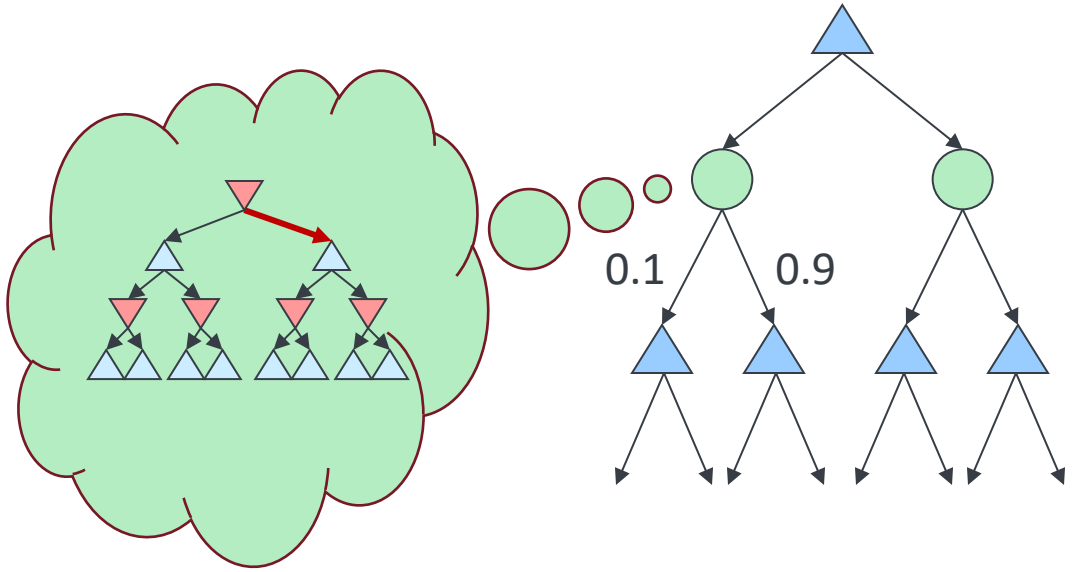
- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



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

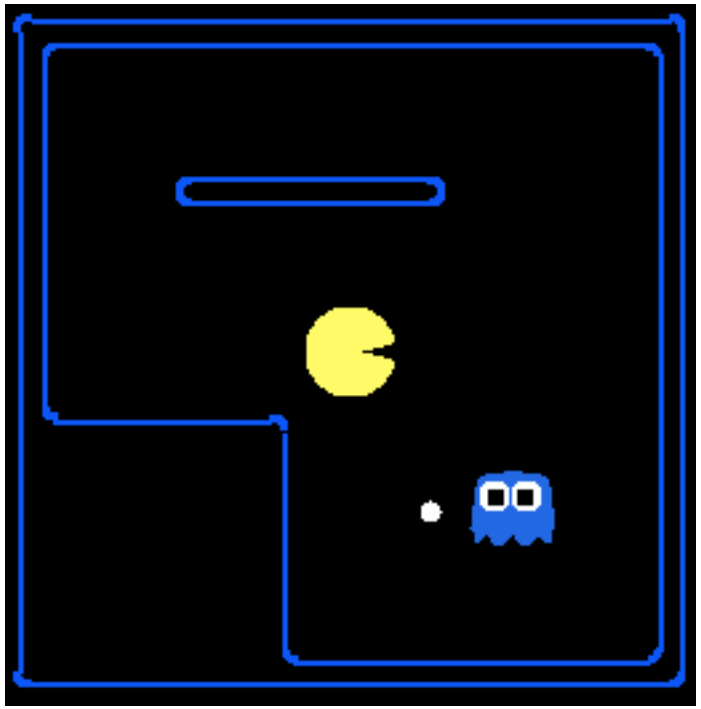
Dangerous optimism

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	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

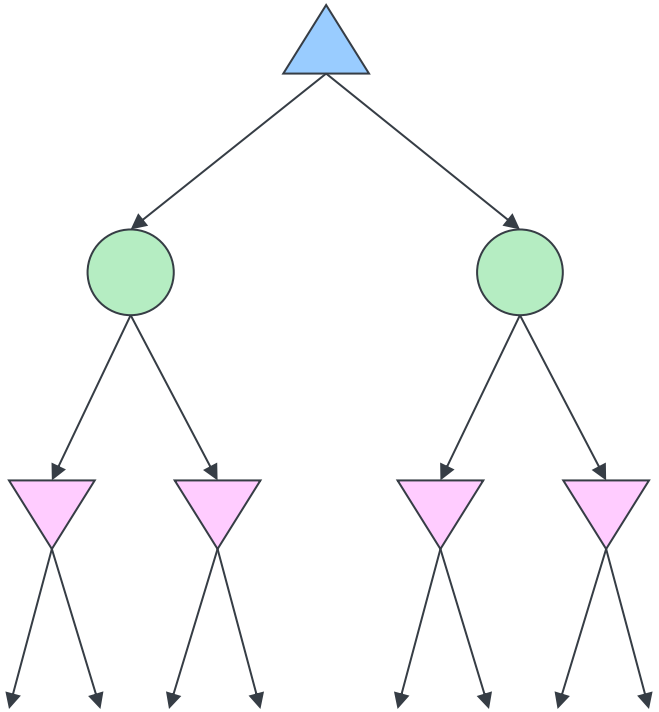
Results from playing 5 games

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

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

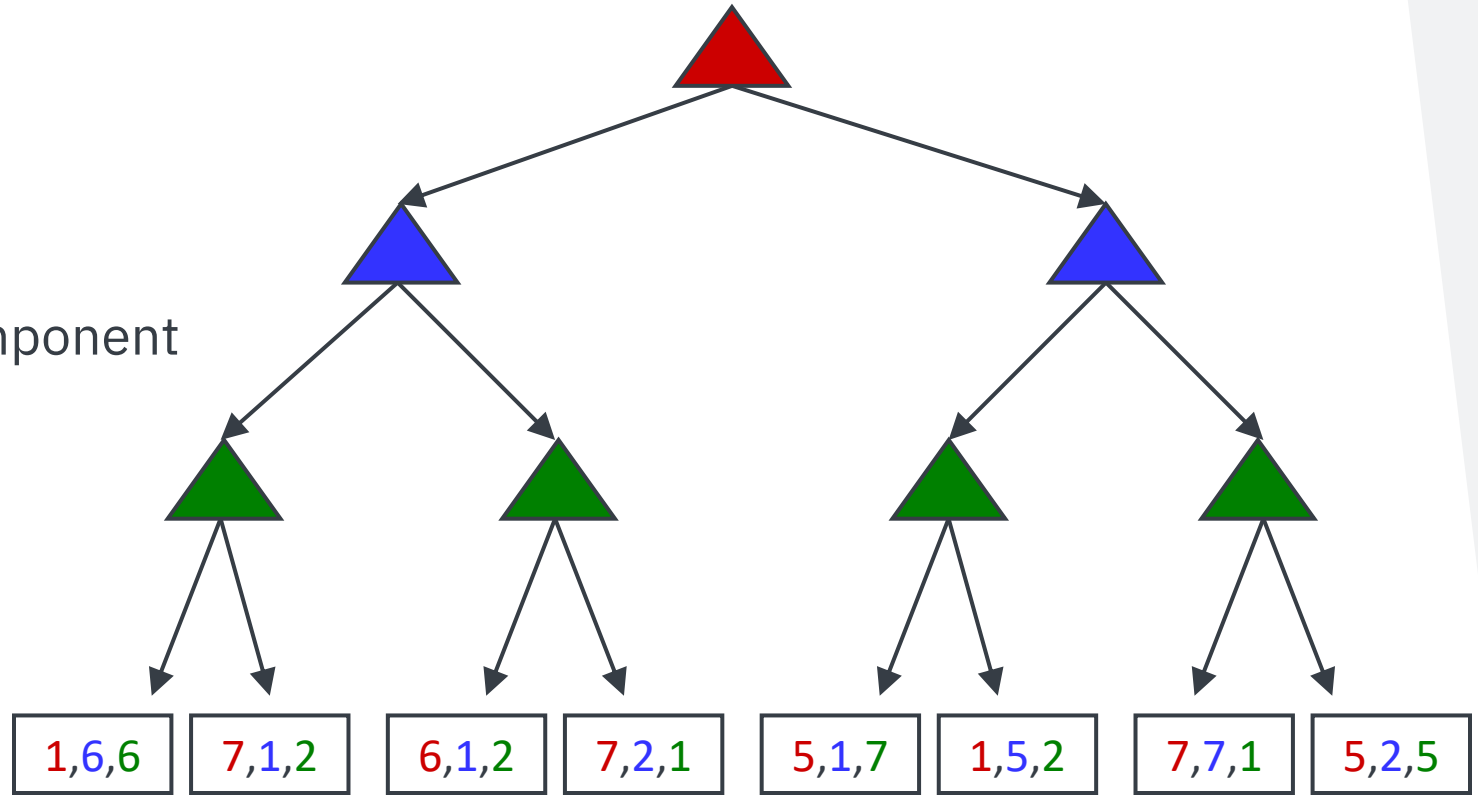
- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon ≈ 20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - 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…



Utilities

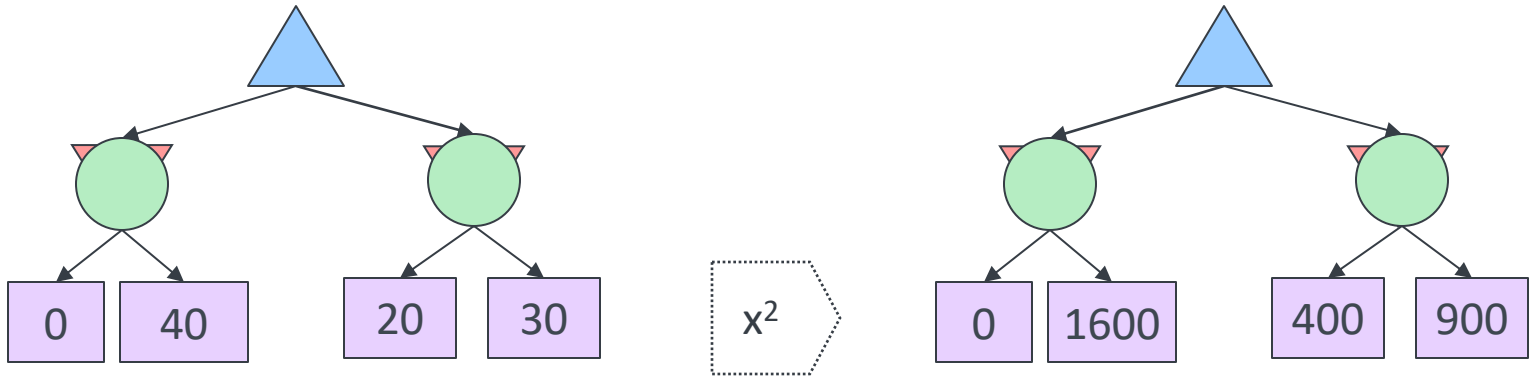
Maximum expected utility

- Why should we average utilities? Why not minimax?

- Principle of maximum expected utility:
  - A rational agent should chose the action that **maximizes its expected utility, given its knowledge**

- Questions:
  - Where do utilities come from?
  - How do we know such utilities even exist?
  - How do we know that averaging even makes sense?
  - What if our behavior (preferences) can't be described by utilities?

What utilities to use?



- For worst-case minimax reasoning, terminal function scale doesn't matter
  - We just want better states to have higher evaluations (get the ordering right)
  - We call this **insensitivity to monotonic transformations**

- For average-case expectimax reasoning, we need *magnitudes* to be meaningful

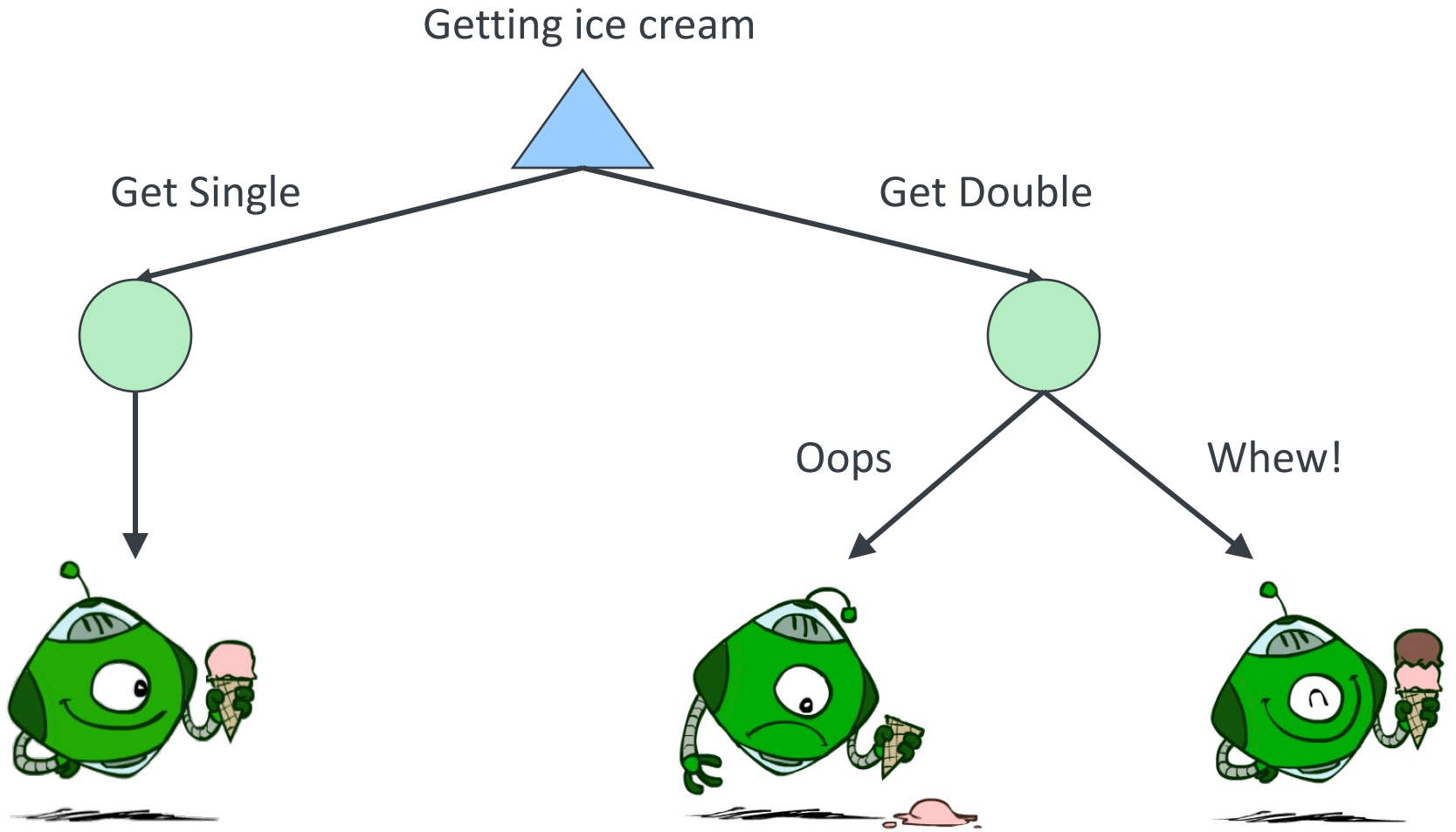
Utilities

- 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?

Utilities: uncertain outcomes



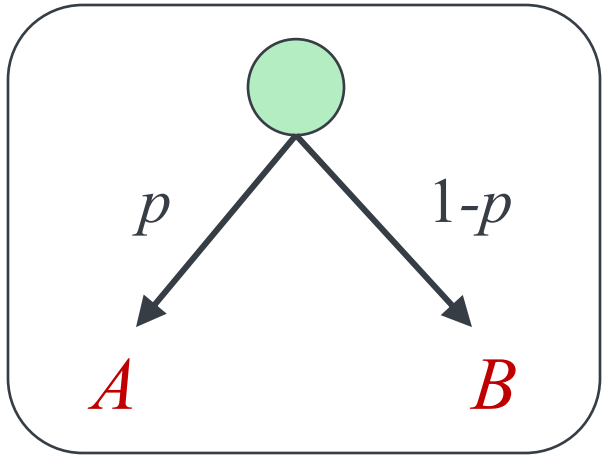
Preferences

- An agent must have preferences among:
  - Prizes: A, B, etc.
  - Lotteries: situations with uncertain prizes

$$L = [p, A;\ \ (1-p), B]$$

A Prize

A Lottery



- Notation:
  - Preference: $A \succ B$
  - Indifference: $A \sim B$

Rationality

Rational preferences

- We want some constraints on preferences before we call them rational, such as:

Axiom of Transitivity: $(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$

- For example: an agent with **intransitive preferences** can be induced to give away all of its money
  - If B > C, then an agent with C would pay (say) 1 cent to get B
  - If A > B, then an agent with B would pay (say) 1 cent to get A
  - If C > A, then an agent with A would pay (say) 1 cent to get C

Rational preferences

The axioms of rationality

Orderability

$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

Transitivity

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

Continuity

$$A \succ B \succ C \Rightarrow \exists p\ [p, A;\ 1-p, C] \sim B$$

Substitutability

$$A \sim B \Rightarrow [p, A;\ 1-p, C] \sim [p, B; 1-p, C]$$

Monotonicity

$$A \succ B \Rightarrow$$
$$(p \geq q \Leftrightarrow [p, A;\ 1-p, B] \succeq [q, A;\ 1-q, B])$$

Theorem: Rational preferences imply behavior describable as maximization of expected utility.

MEU principle

- Theorem [Ramsey, 1931; von Neumann & Morgenstern, 1944]
  - Given any preferences satisfying these constraints, there exists a real-valued function U such that:

$$U(A) \geq U(B) \;\Leftrightarrow\; A \succeq B$$

$$U([p_1, S_1;\; \ldots\; ;\; p_n, S_n]) = \sum_i p_i U(S_i)$$

  - *I.e.* values assigned by U preserve preferences of both prizes and lotteries!

- Maximum expected utility (MEU) principle:
  - Choose the action that maximizes expected utility
  - Note: an agent can be entirely rational (consistent with MEU) without ever representing or manipulating utilities and probabilities
  - E.g., a lookup table for perfect tic-tac-toe, a reflex vacuum cleaner

Human utilities

Utility scales

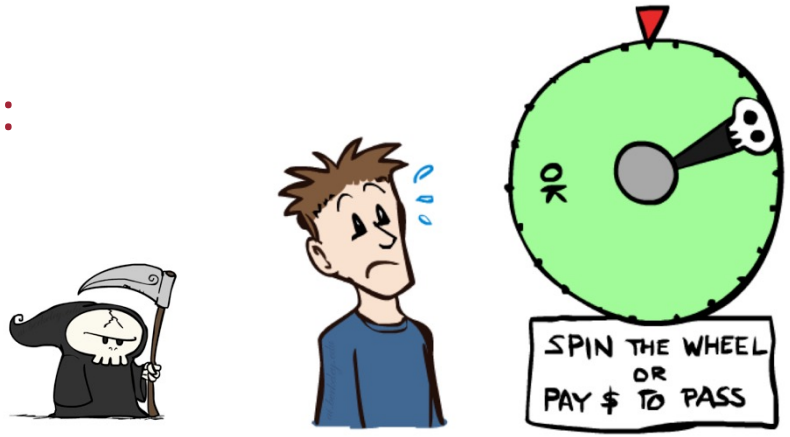
- **Normalized utilities**: $u_+ = 1.0$, $u_- = 0.0$
- **Micromorts**: one-millionth chance of death, useful for paying to reduce product risks, etc.
- **QALYs**: quality-adjusted life years, useful for medical decisions involving substantial risk
- Note: behavior is invariant under positive linear transformation

$$U'(x) = k_1 U(x) + k_2 \quad \text{where } k_1 > 0$$

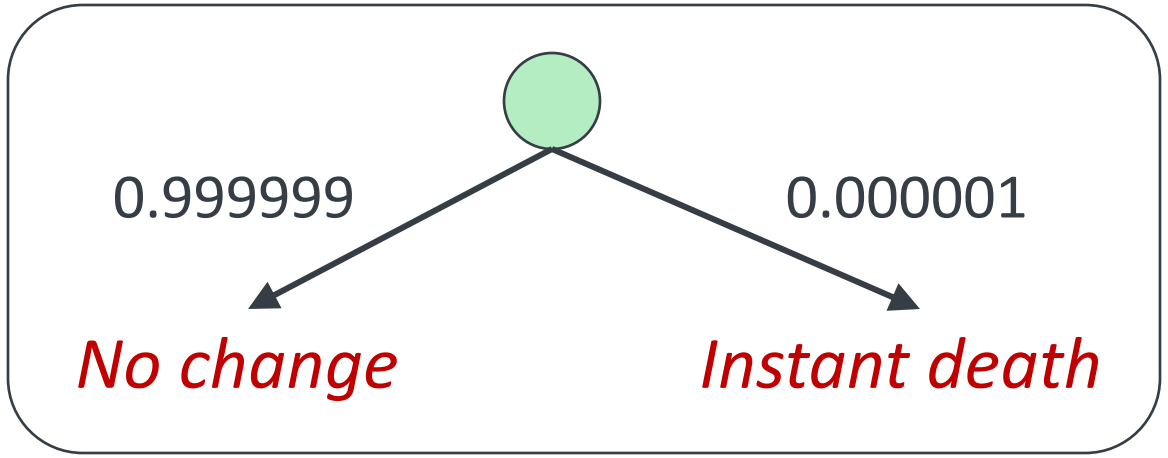
- With deterministic prizes only (no lottery choices), only ordinal utility can be determined, *i.e.*, total order on prizes

Human utilities

- Utilities map states to real numbers. Which numbers?
- Standard approach to assessment (elicitation) of human utilities:
  - Compare a prize A to a standard lottery L_p between
    - "best possible prize" u_+ with probability p
    - "worst possible catastrophe" u_- with probability 1-p
  - Adjust lottery probability p until indifference: A ~ L_p
  - Resulting p is a utility in [0,1]

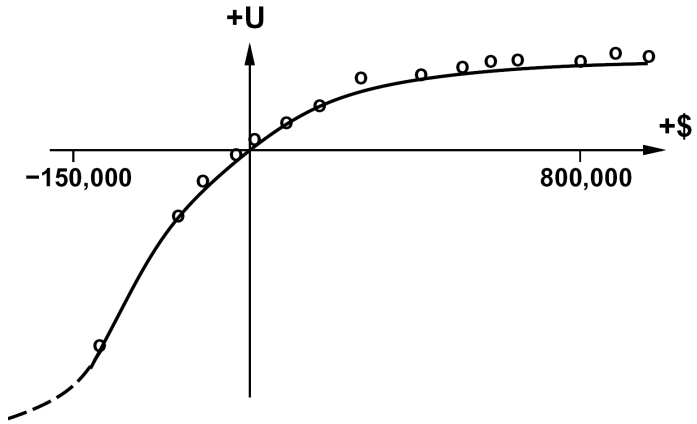


Pay $30 ~



Money

- Money does not behave as a utility function, but we can talk about the utility of having money (or being in debt)
- Given a lottery L = [p, $X; (1-p), $Y]
  - The expected monetary value EMV(L) is p*X + (1-p)*Y
  - U(L) = p*U($X) + (1-p)*U($Y)
  - Typically, U(L) < U(EMV(L))
  - In this sense, people are risk-averse
  - When deep in debt, people are risk-prone



Example: Insurance

- Consider the lottery [0.5, $1000; 0.5, $0]
  - What is its expected monetary value? ($500)
  - What is its certainty equivalent?
    - Monetary value acceptable in lieu of lottery
    - $400 for most people
  - Difference of $100 is the insurance premium
    - There's an insurance industry because people will pay to reduce their risk
    - If everyone were risk-neutral, no insurance needed!
  - It's win-win: you'd rather have the $400 and the insurance company would rather have the lottery (their utility curve is flat and they have many lotteries)

Which one do you prefer? - 1

A: [0.8, $4k; 0.2, $0]

B: [1.0, $3k; 0.0, $0]

Which one do you prefer? -2

C: [0.2, $4k; 0.8, $0]

D: [0.25, $3k; 0.75, $0]

Example: human rationality?

- Famous example of Allais (1953)
  - A: [0.8, $4k; 0.2, $0]
  - B: [1.0, $3k; 0.0, $0]
  - C: [0.2, $4k; 0.8, $0]
  - D: [0.25, $3k; 0.75, $0]
- Most people prefer B > A, C > D
- But if U($0) = 0, then
  - B > A $\Rightarrow$ U($3k) > 0.8 U($4k)
  - C > D $\Rightarrow$ 0.8 U($4k) > U($3k)

STEVENS
INSTITUTE OF TECHNOLOGY
1870

Questions?

Chapter 1, 2, 3

Acknowledgement

Fahiem Bacchus, University of Toronto
Dan Klein, UC Berkeley
Kate Larson, University of Waterloo