­CS 188 Notes 1

**LEC 1:** Stuart Russell and Sergei Levine

Designing Rational Agents

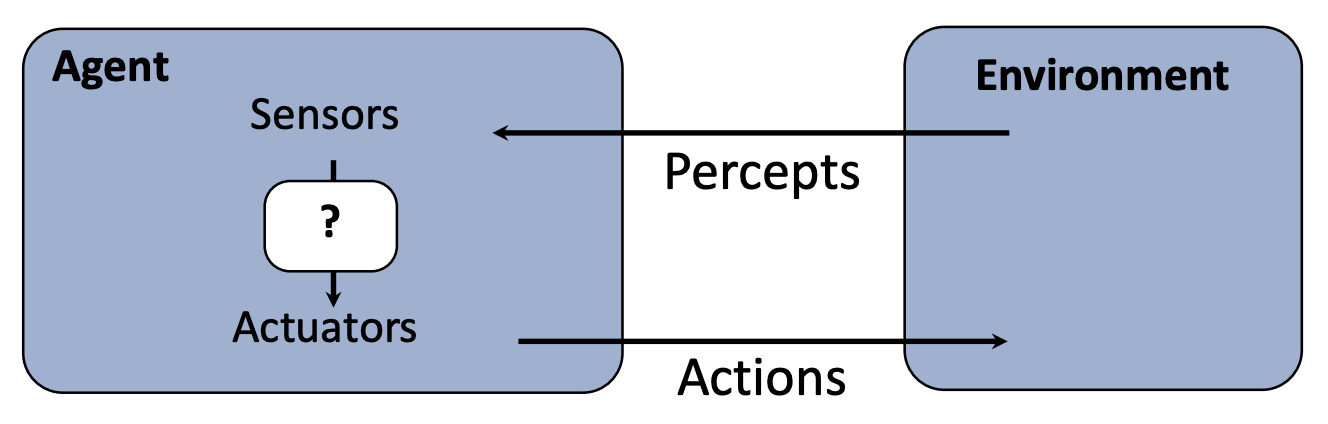
* **Agent** is an entity that *perceives* and *acts*
* **Rational Agent** selects actions that maximizes its expected **utility**
* Characteristics of the **percepts, environment**, and **action space** dictate techniques for selecting rational actions

**LEC 2: Uninformed Search (BFS, DFS)**

Agenda:

* Agents that plan ahead
  + Agents that have a goal, and a cost (reach goal with lowest cost today)
* Search problems
* Uninformed Search Methods
  + Depth first
  + Breadth first
  + Uniform cost search

An agent **perceives** its environment through **sensors** and **acts** upon it through **actuators**

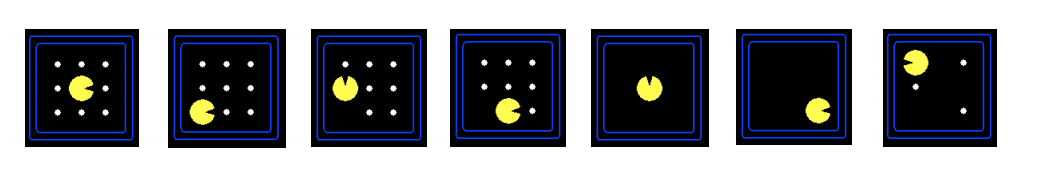


**Environment** type largely determines the agent design

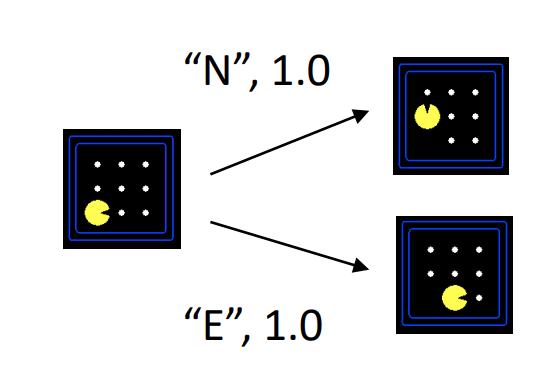
* **Fully/Partially observable** 🡪 agent requires **memory** (internal state)
* **Discrete/Continuous** 🡪 agent may not be able to enumerate **all states**
* **Stochastic/deterministic 🡪** agent may have to prepare for **contingencies**
* **Singe/multi-agent 🡪** agent may need to behave **randomly**

Planning agents:

* Asks “what if”
* Decisions based on (hypothesized) consequence of actions
* Must have a model of ow the world evolves in response to actions
* Must formulate a goal (test)
* Optimal vs. complete planning
  + Optimal means finds the best plan
  + Complete means always finds a plan that always succeeds
* Planning vs. Replanning

Search Problems have:

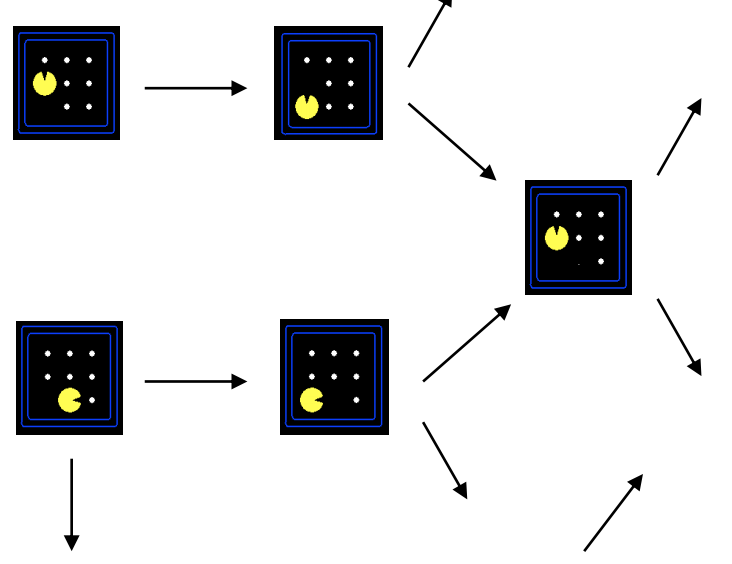
* A state space



* A successor function (with actions, costs)
* A start state and a goal test
* A solution is a sequence of actions which transforms the start state to a goal state

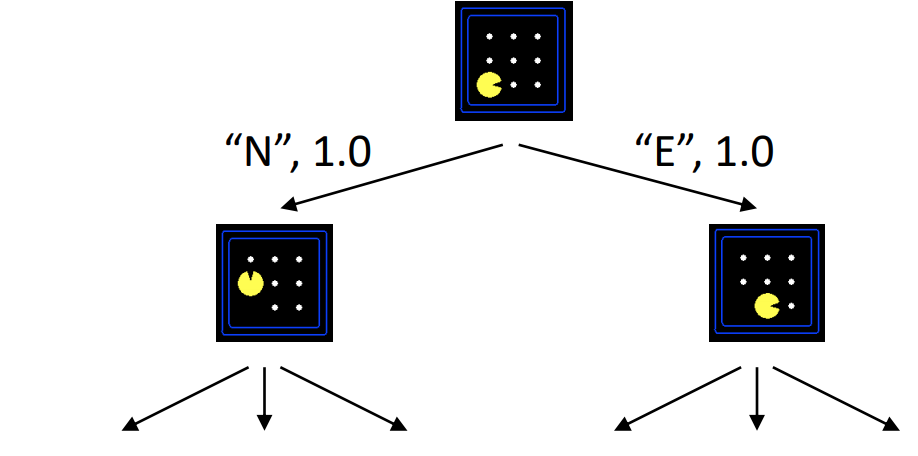
**World State =** every last detail of the environment

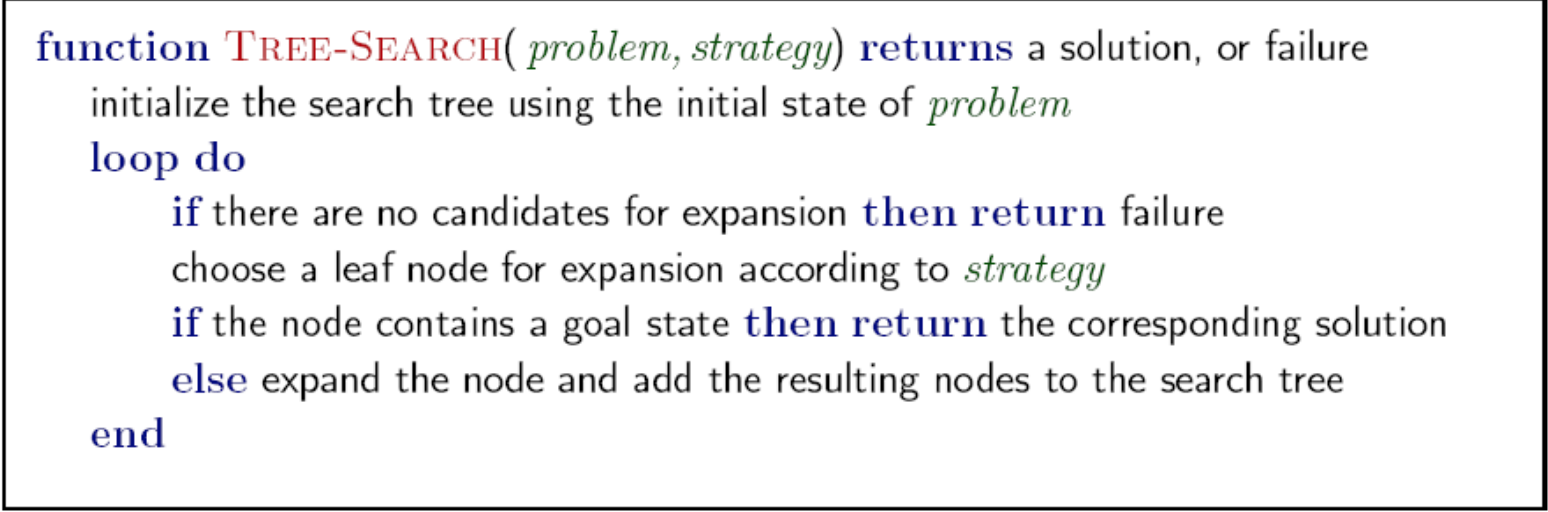
**Search State** keeps only the details needed for planning (abstraction)

State Space Graph:

* A mathematical representation of the search problem
* Nodes are abstracted world configurations
* Edges represent successors
* Goal test is a set of goal nodes

Use State Space Tree instead (less complicated):

* Start state is root note
* Children correspond to successors
* Nodes show states, but correspond to **PLANS** to achieve those states
* Can **never** actually build whole tree for most problems

PseudoCode for General Tree Search

Key Ideas:

Fringe

Expansion

Exploration Strategy

Depth-First Search

* Expand the deepest node first
* **Implementation** is a LIFO (Last In First Out) stack
* **Runtime:** if m is finite O(b^m)
* **Space:** Only has siblings on path to root, so O(bm)
* Complete but not optimal search

Breadth-first Search

* Expand the shallowest node first
* **Implementation** is a FIFO (First in First Out) queue
* **Runtime:** O(b^s)
* **Space:** O(b^s)
* Complete but not optimal search
* Optimal when edge costs are all 1

DFS with Iterative Deepening:

**Idea:** Get DFS’s space advantage with BFS’s time/shallow –solution advantages

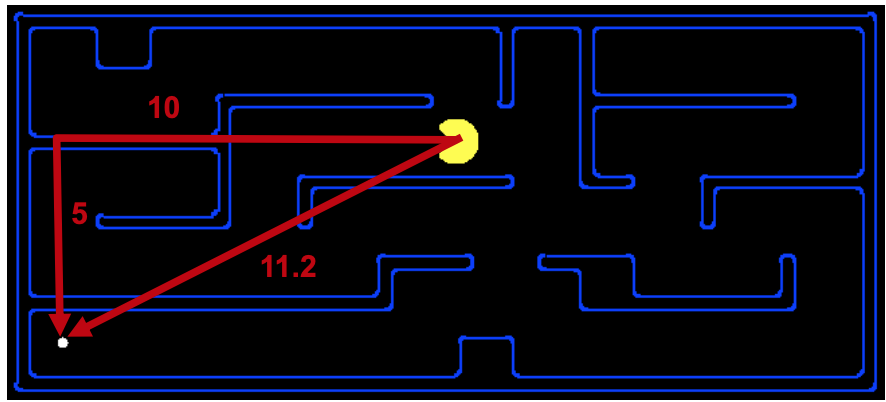
* Run a DFS with depth limit 1. If no solution…
* Run a DFS with depth limit 2. If no solution…
* Run a DFS with depth limit 3. Etc.

Uniform-Cost Search

* **Strategy:** expand a lowest cost node first
* **Implementation:** priority queue (priority = cumulative cost)
* **Runtime:**
  + If solution costs C and arcs cost at **e** least C/**e**
  + Takes time
* **Space:**
* Good: Optimal and Complete
* Bad:
  + Explores options in every direction
  + No information about goal location

**LEC 3: Informed Search (A\* Search and Heuristics)**

Informed Search:

* Heuristics
* Greedy Search
* A\* Search

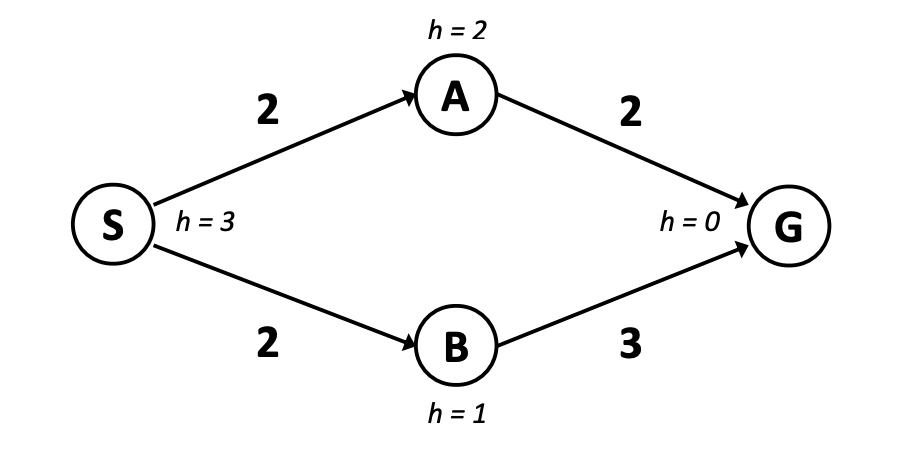
Heuristic:

* A function that *estimates* how close a state is to a goal
* Designed for a particular search problem
* Examples:
  + Manhattan Distance (x+y distance, 15 in example above), Euclidean Distance (11.2 in example above)

Greedy Search:

* **Strategy:** Use a heuristic to expand the node that seems closest to the goal
  + **Heuristic:** estimate of distance to nearest goal for each state
* Complete but not always optimal since it only uses heuristic (not actual edge costs)

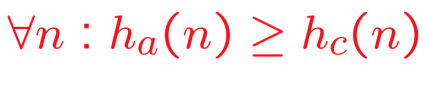
A\* Search

* Combination of Greedy and Uniform Cost Search
* **Unform-cost** orders by path cost, or *backward* cost **g(n)**
* **Greedy** orders by goal proximity, or forward cost **h(n)**
* **A\* Search** orders by the sum: **f(n) = g(n) + h(n)**
* When should A\* terminate?
  + Only stop when you DEQUEUE the goal (not enqueue it)
  + Ex of why not to terminate when you enqueue the goal 🡪 🡪

Admissible Heurisitics:

* Heurisitcs where the heurisitic estimates are less than the actual costs
* A heuristic **h** is **admissible** (optimistic) if:
  + 0 ≤ h(n) ≤ h\*(n) where h\*(n) is the true cost to a nearest goal
* Slow down bad plans but never outweigh true costs

Creating Admissible Heuristics

* Admissible heuristics are solutions to relaxed problems, where new actions are available
* Ex:
  + Euclidean distance from start city to end city (assumes you can go directly from 1 city to another) as heuristic instead of using roads
* Dominant Heuristics:
  + h\_a ≥ h\_b if:
  + means that if we use h\_a, it will do less work than if we had used h\_c
* Lower bound of heurisitic is 0
* Top of heuristic is actual cost

Advantages of A\* Search:

* **Uniform-cost** expands equally in all “directions”
* **A\*** expands mainly toward the goal, but does hedge its bets to ensure optimality
* A trade-off between quality of estimate and work per node

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
* **Store the closed set as a set not as a list**
* Complete but not optimal necessarily

In order to make Graph Search optimal:

* Consistency of Heuristics
* Main idea: **estimated heuristic costs ≤ actual costs**
  + Admissibility:
    - heuristic cost ≤ actual cost to goal h(A) ≤ actual cost from A to G
  + Consistency:
    - cost from current node to a successor node + estimate cost from the successor node to the goal (heuristic) ≤ estimated cost from the curr node to the goal (heuristic)
    - C(n, n’) + h(n’) ≤ h(n)

LEC 4: Adversarial Search and Game Trees

Axes:

* deterministic or stochastic?
* one, two, or more players?
* zero sum?
* perfect info (can you see the whole state space)?

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

Deterministic Games:

1 player tries to maximize result, the other tries to minimize result (Tic-tac-toe, chess, checkers)

* States: S (start at s0 )
* Players: P={1...N} (usually take turns)
* Actions: A (may depend on player / state)
* Transition Function: SxA → S
* Terminal Test: S → {t,f}
* Terminal Utilities: SxP → R
* **Solution for a player is a policy:** S 🡪 A

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

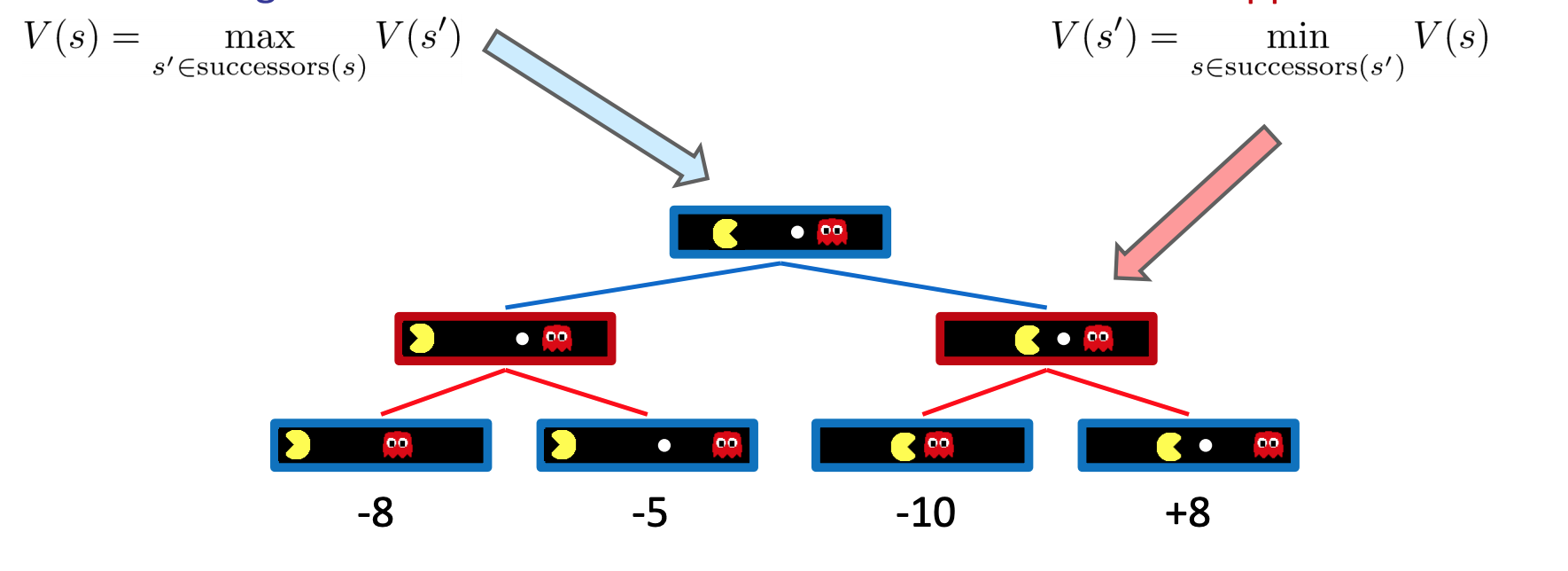
General Games

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

Value of a State:

* the best achievable outcome (utility) from that state

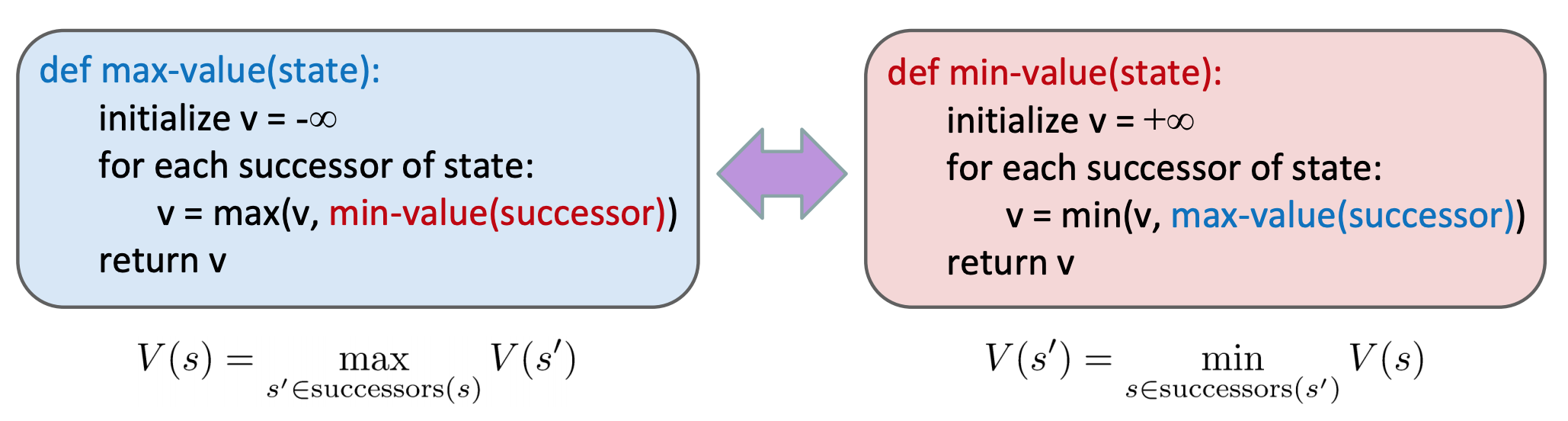
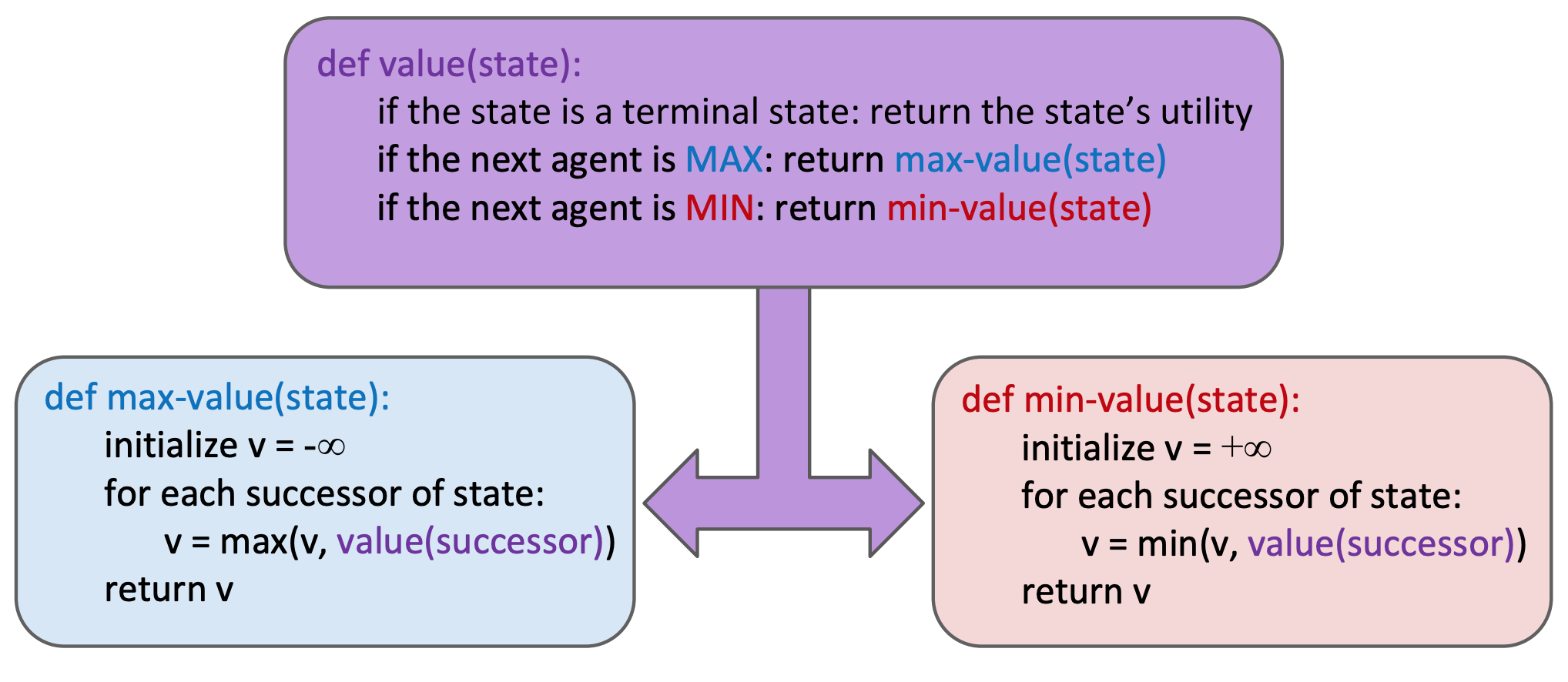
Adversarial Game Trees:



* Assuming Pacman and ghost alternate moves
* Value of right red node is -10 (min when ghost gets to choose)
* Value of left red node is -8
* Value of root node is therefore -8

Minimax search:

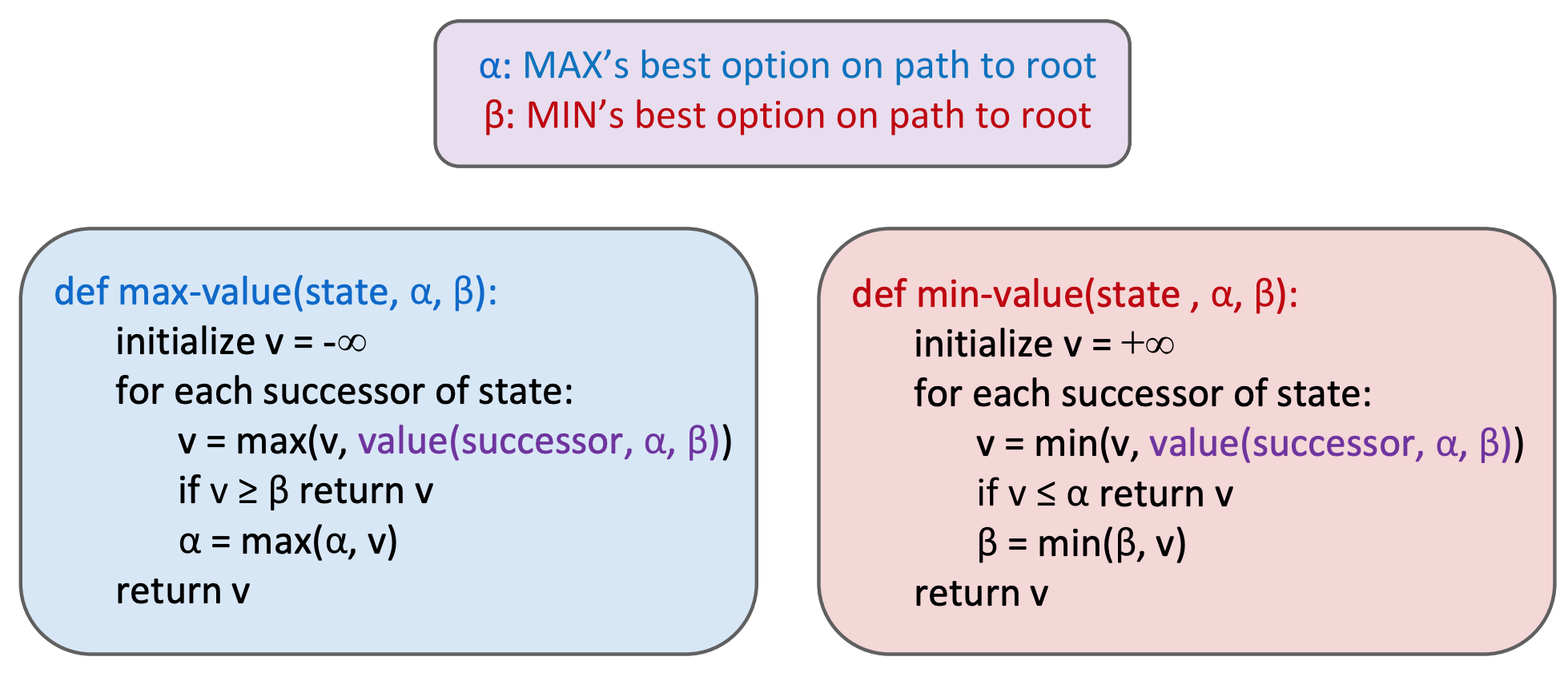
* A state-space search tree
* Players alternate turns
* Compute each node’s **minimax value**: the best achievable utility against a rational (optimal) adversary
* Like an exhaustive DFS
* Time: O(b^m)
* Space: O(bm)

Implementation (2 diff ways):

Game Tree Pruning (Alpha-Beta Pruning):

* General Alg for 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

**α-β** Implementation:

* 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

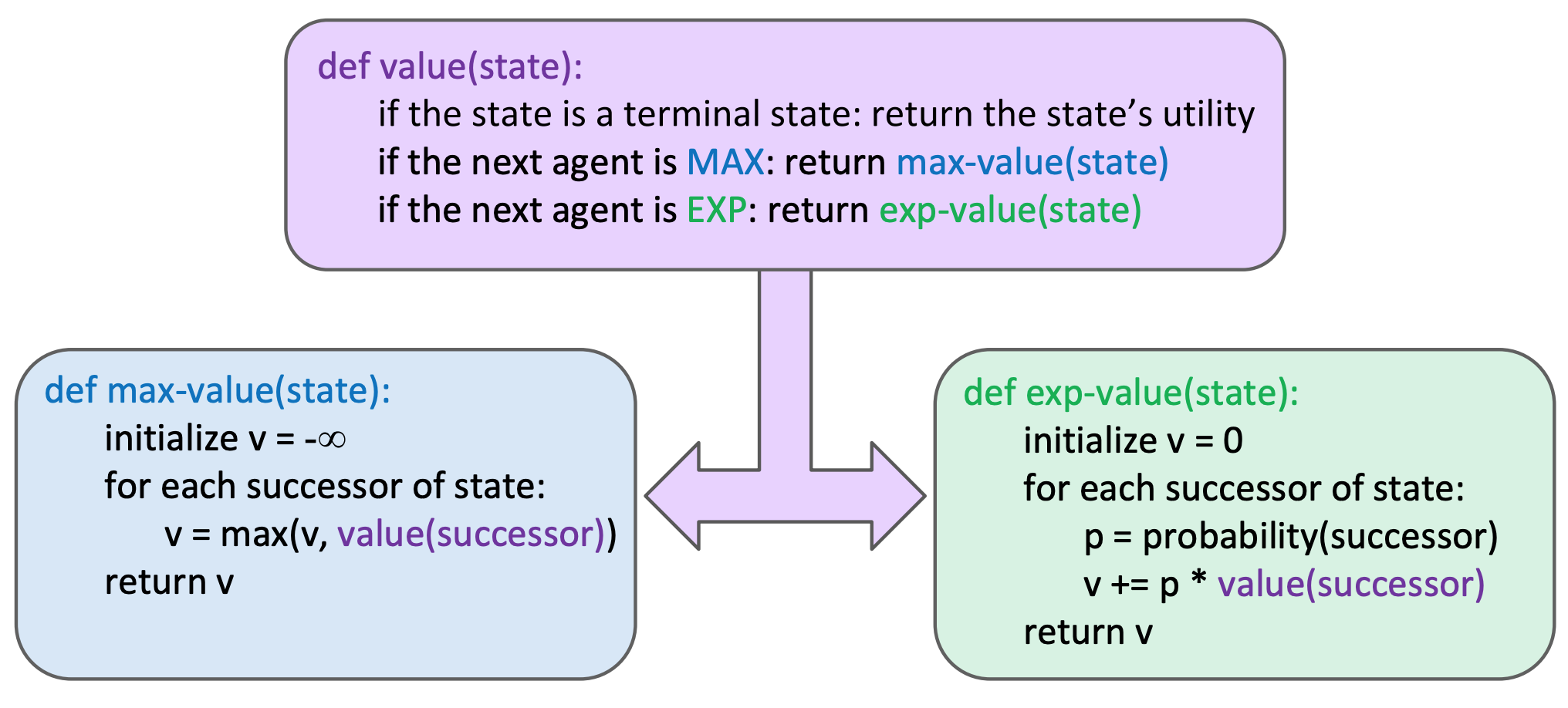
Resource Limits:

* **Problem:** in realistic games, cannot search entire tree
* **Depth-Limited Search:** 
  + search only to a limited depth in the tree
  + replace terminal utilities with an **evaluation function** for non-terminal positions
  + **May cause have problems** (thrashing/starving example from Lecture)
* **Evaluation function:**
  + ideal function returns the actual minima value of the position
  + In practice: typically a weighted linear sum of features
  + the deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters

**LEC 5: Expectimax & Markov Decision Processes**

­ If there is randomness/chance involved: **Expectimax Search:**

compute the average score under optimal play

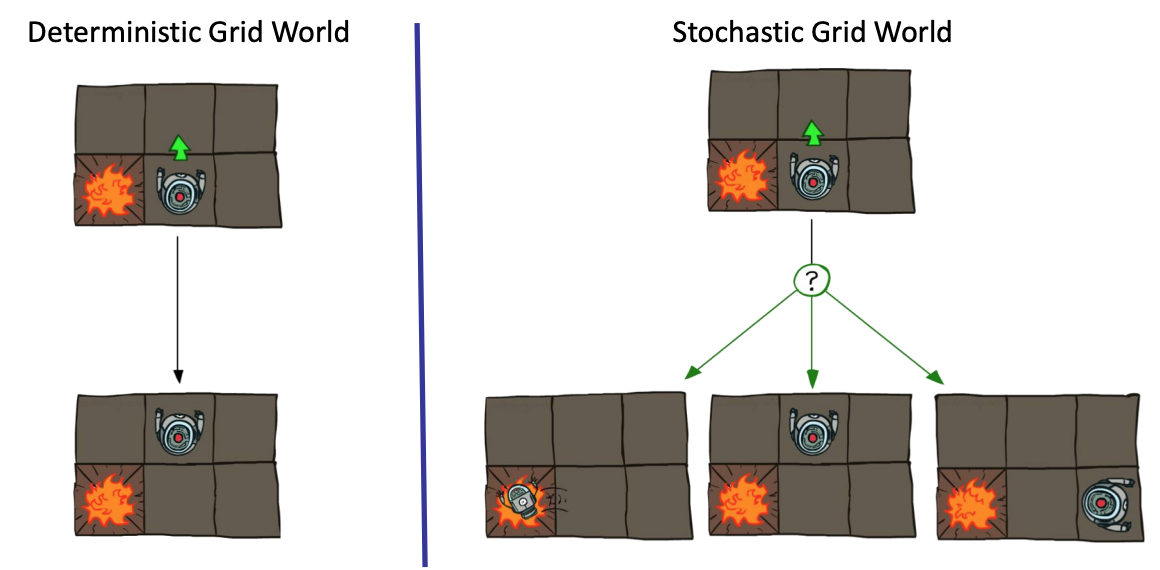
* Explicit randomness:
  + rolling dice
  + Unpredictable opponents: the ghosts respond randomly
  + Actions can fail: when moving a robot, wheels might slip
* Values should now reflect average-case (**expectimax**) outcomes, not worst-case (**minimax**) outcomes

Review: Probability

* A **random variable** represents an event whose outcome is unknown
* A **probability distribution** is an assignment of weights to outcomes
* **Expected value** of a function of a random variable is the average, weighted by the probability distribution over outcomes

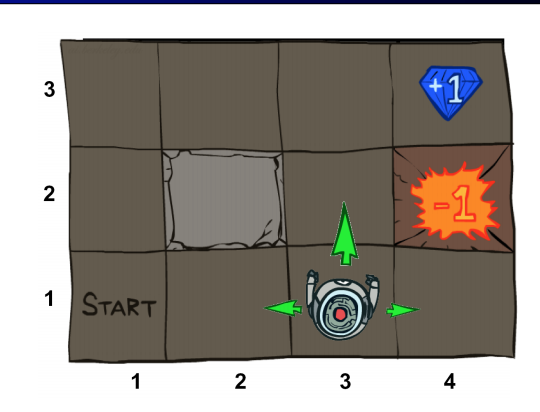
Non-Deterministic Search:

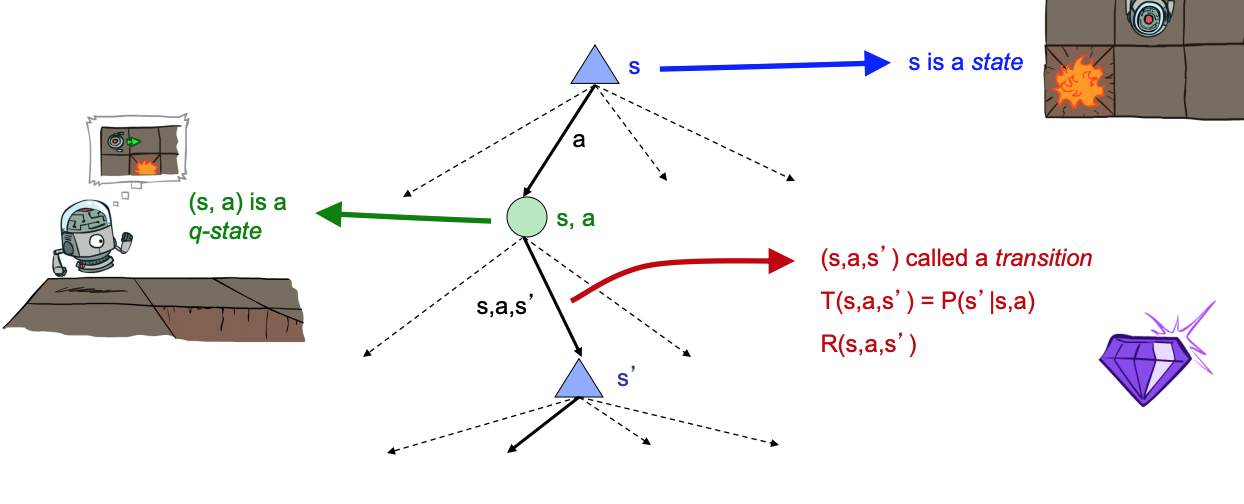
Example: Grid World

* A maze-like problem
  + The agent lives in a grid
  + Walls block the agent’s path
* Noisy movement: actions do not always go as planned
  + 80% of the time, the action North takes the agent North (if there is no wall there)
  + 10% of the time, North takes the agent West; 10% East
* If there is a wall in the direction the agent would have been taken, the agent stays put
* The agent receives rewards each time step
  + Small “living” reward each step (can be negative)
  + Big rewards come at the end (good or bad)
* Goal: maximize sum of rewards

Markov Decision Processes:

* An MDP is defined by:
  + A set of **states s** ∈ S
  + A set of **actions a** ∈ A
  + A **transition function** T(s, a, s’)
    - Probability that a from s leads to s’, i.e., P(s’| s, a)
    - Also called the model or the dynamics ▪
  + A **reward function** R(s, a, s’)
    - Sometimes just R(s) or R(s’)
  + A **start state**
  + Maybe a **terminal state**
* For MDP’s, we want to output an **optimal policy π\*: S🡪A**
  + policy π gives an action for each state (maps states to actions)
  + an optimal policy is one that maximizes the **expected utility** if followed
    - Utility = Sum of discounted rewards
  + **Values** = expected future utility from a state (max node
  + **Q-Values** = expected future utility from a q-state (chance node)
  + An explicit policy defines a reflex agent
* Expectimax didn’t compute entire processes
  + It computed the action for a single state only

Optimal Policies for the State Space given a R(s), the cost per step

MDP Search Trees:

Utilities of Sequences:

* What preferences should an agent have over reward sequences:
  + More or less? 🡪 [1,2,2] or [2,3,4]
  + Now or later? 🡪 [0,0,1] or [1,0,0]
* **Discounting:**
  + The values of rewards decay exponentially as time moves on (reasonable to prefer rewards now to rewards later and to maximize sum of rewards)

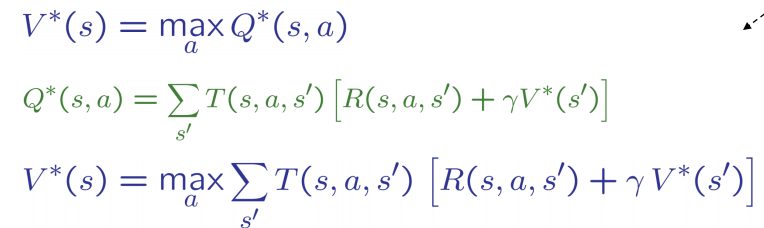
Discounting:

* How to discount:
  + Each time we descend a level, we multiply in the discount once
* Why discount?
  + Sooner rewards probably do have higher utility than later rewards
  + Also helps our algorithms converge

Solutions:

* Finite horizon: (similar to depth-limited search)
  + Terminate episodes after a fixed T steps (e.g. life)
  + Gives nonstationary policies (π depends on time left)
* Discounting: use 0 < γ < 1
* Absorbing state: guarantee that for every policy, a terminal state will eventually be reached

Solving MDP’s: **Optimal Quantities**

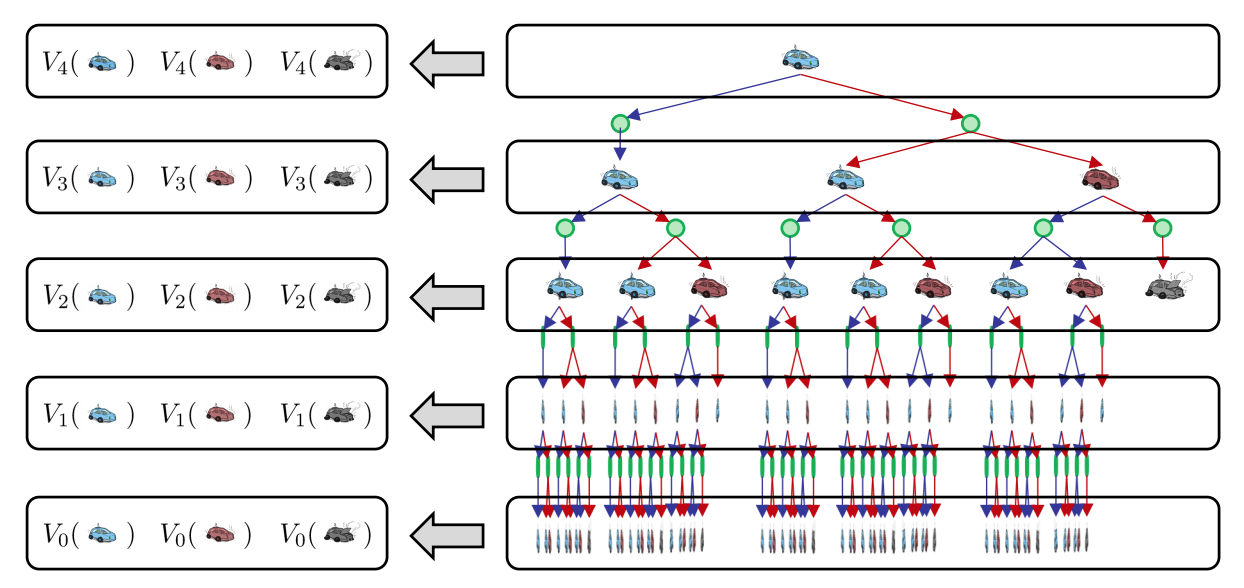
* The value (utility) of a **state s**:
  + **V\*(s)** = expected utility starting in s and acting optimally
* The value (utility) of a **q-state (s,a):**
  + **Q\*(s,a)** = expected utility starting out having taken action a from state s and (thereafter) acting optimally
* The **optimal policy**:
  + **π\*(s)** = optimal action from state s

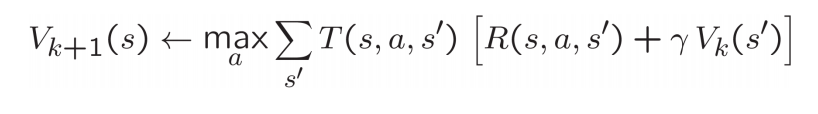
**LEC 6: How to Solve Markov Decision Processes**

­ MDP Search Trees:

* **Problem**: States are repeated
  + **Idea**: Only compute needed quantities once
* **Problem**: Tree goes on forever
  + **Idea**: Do a depth-limited computation, but with increasing depths until change is small
  + Note: deep parts of the tree eventually don’t matter if γ < 1
* **Key Idea:**
  + Time limited values
  + **Define V\_k (s)** to be the optimal value of s if the game ends in k more time steps
    - Equivalently, it’s what a depth-k expectimax would give from s

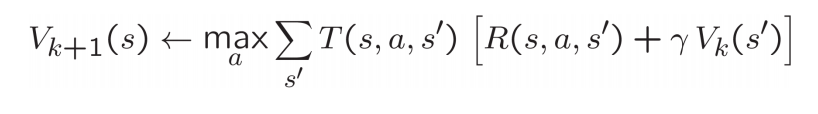
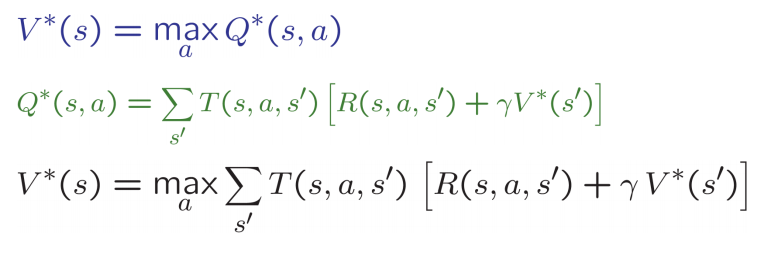
Computing Time-limited values (Value-Iteration)

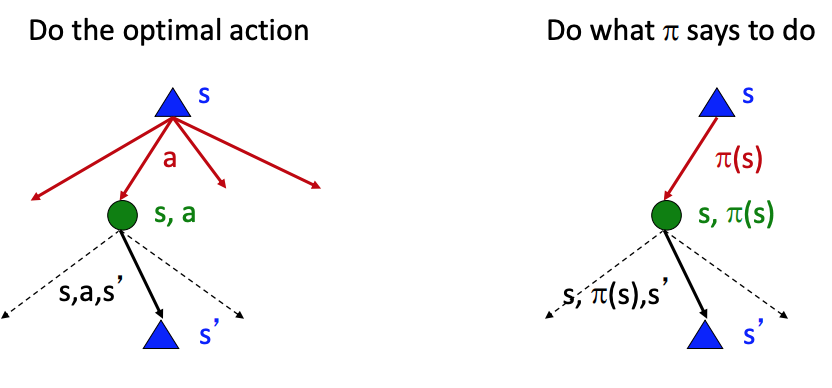


* ▪ Start with V0 (s) = 0:
  + no time steps left means an expected reward sum of zero
* Given vector of (s) values, do one step of expectimax from each state:
* Repeat until convergence
* Complexity of each iteration: O(A)
* Problems:
  + It’s slow : O(A)
  + The “max” at each state rarely changes
  + The policy often converges long before the values (policy below)

Bellman Equations:

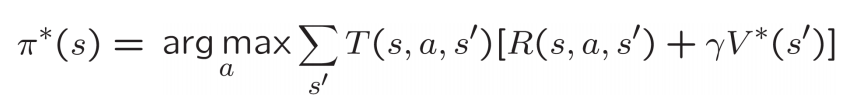
**Bellman equations** characterize the optimal values: **Value iteration** computes them:



Fixed Policies:

* Expectimax trees max over all actions to compute the optimal values
* If we **fixed** some **policy π(s),** then the tree would be simpler – only **one action per state** 
  + … though the tree’s value would depend on which policy we fixed

Policy Extraction (Computing **actions** from **values)**:

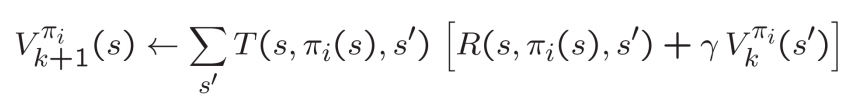
Given we have optimal values V\*(s), do a mini-expectimax (one-step) and take the action with the maximum utility):

Policy Iteration:

* Step 1:
  + **Policy evaluation**: calculate utilities for some fixed policy (not optimal utilities!) until convergence
* Step 2:
  + **Policy improvement**: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
* Repeat steps until policy converges
  + Optimal and can converge much faster than value iteration

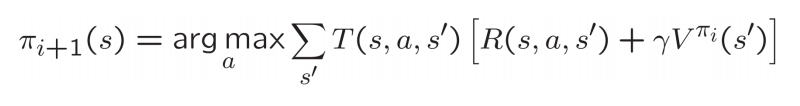
**Evaluation**: For fixed current policy π, find values with policy evaluation:

* Iterate until values converge:



**Improvement**: For fixed values, get a better policy using policy extraction

* One-step look-ahead:



Comparison between Value and Policy Iteration:

* Both value iteration and policy iteration compute the same thing (all optimal values)
* In **value iteration**:
  + Every iteration updates both the values and (implicitly) the policy
  + We don’t track the policy, but taking the max over actions implicitly recomputes it
* In **policy iteration**:
  + We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  + After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  + The new policy will be better (or we’re done)
* Both are dynamic programs for solving MDPs

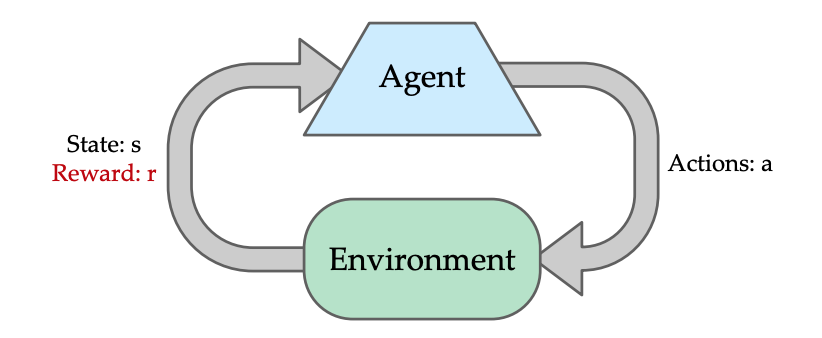
Summary LEC:

* So you want to….
  + Compute optimal values: use value iteration or policy iteration
  + Compute values for a particular policy: use policy evaluation
  + Turn your values into a policy: use policy extraction (one-step lookahead)
* These all look the same!
  + They basically are – they are all variations of Bellman updates
  + They all use one-step lookahead expectimax fragments
  + They differ only in whether we plug in a fixed policy or max over actions

LEC 7: Reinforcement Learning

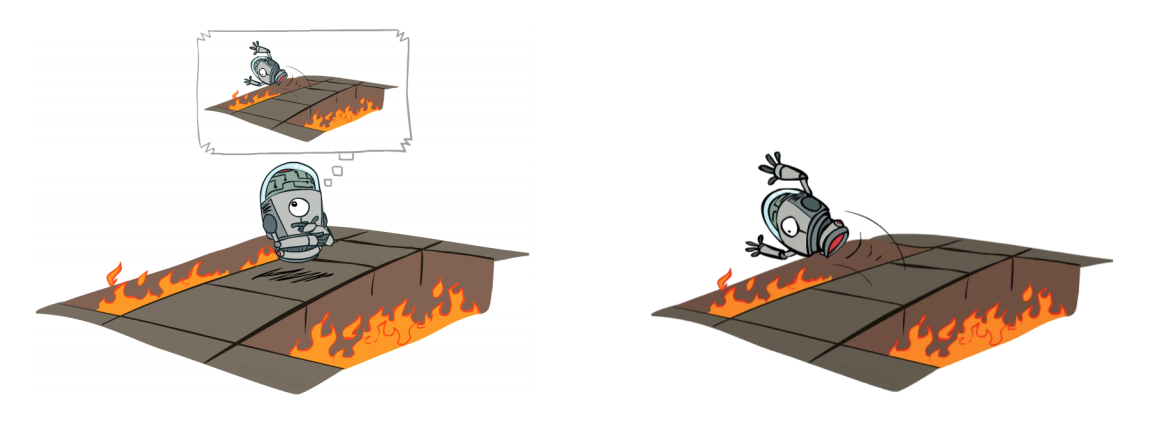
Important ideas in reinforcement learning

* **Exploration**: you have to try unknown actions to get information
* **Exploitation**: eventually, you have to use what you know
* **Regret**: even if you learn intelligently, you make mistakes
* **Sampling**: because of chance, you have to try things repeatedly
* **Difficulty**: learning can be much harder than solving a known MDP

Reinforcement Learning:

* Like MDP but **dont know T (Transition States) or R (reward function)**
* **Basic Idea:**
  + Receive feedback in the form of **rewards**
  + Agent’s utility is defined by the **reward function**
  + Must (learn to) act so as to **maximize expected rewards**
  + All learning is based on observed samples of outcomes!

Offline (MDPs) vs. Online (RL) Learning



Model-Based Learning:

* **Model-Based Idea:** 
  + Learn an approximate model based on experiences
  + Solve for values as if the learned model were correct
* **Step 1:** Learn empirical MDP model
  + Count outcomes s’ for each s, a
  + Normalize to give an estimate of (**Transition)**

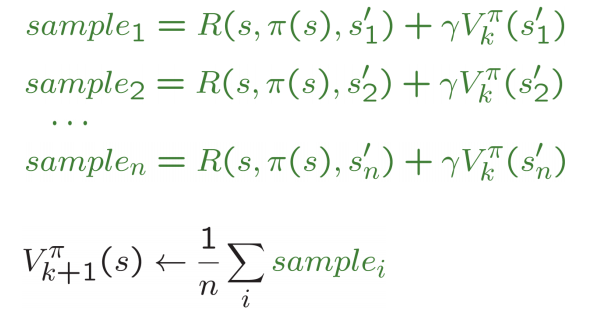


* + Discover each (**reward**) when we experience (s, a, s’) (**state**)
* **Step 2**: Solve the learned MDP
  + For example, use value iteration, as before

Passive Reinforcement Learning:

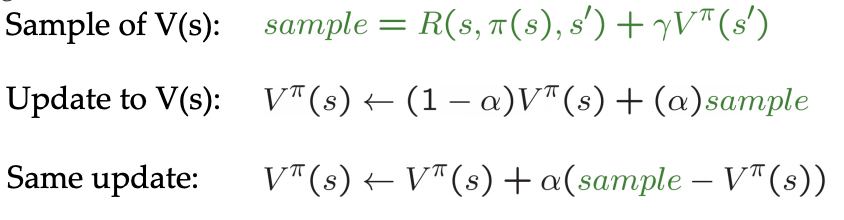
* Simplified task: **policy evaluation**
  + Input: a fixed policy π(s)
  + You don’t know the transitions T(s,a,s’)
  + You don’t know the rewards R(s,a,s’)
  + **Goal: learn the state values**
* In this case:
  + Learner is “along for the ride”
  + No choice about what actions to take
  + Just execute the policy and learn from experience
  + This is NOT offline planning! You actually take actions in the world.
* **Direct Evaluation:**
  + **Goal**: Compute values for each state under π
  + **Idea**: Average together observed sample values
    - Act according to **π**
    - Every time you visit a state, write down what the sum of discounted rewards turned out to be
    - Average those samples

Sample Based Policy Evaluation:

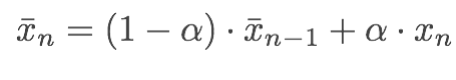
* We want to improve estimate of V by computing average of state values (from Bellman Equations)
* **Idea**: Take samples of outcomes s’ (by doing the action!) and average) :

Temporal Difference Learning Algorithm:

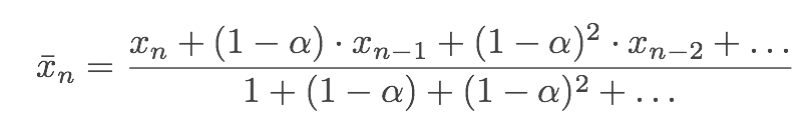
* Big idea: learn from every experience!
  + Update V(s) each time we experience a transition (s, a, s’, r)
  + Likely outcomes s’ will contribute updates more often
* Temporal difference learning of values
  + Policy still fixed, still doing evaluation!
  + Move values toward value of whatever successor occurs: running average



* + **Alpha** represents how you weight **new** vs. **old** experiences

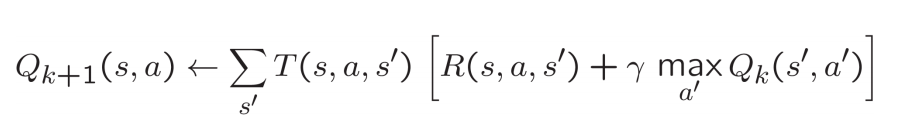


Exponential Moving Average:

* The running interpoliation update:
* Makes recent samples more important:
  + Forgets about past (distant past values were wrong anyway)
  + Decreasing learning rate (**alpha**) can give converging averages

Active Reinforcement Learning:

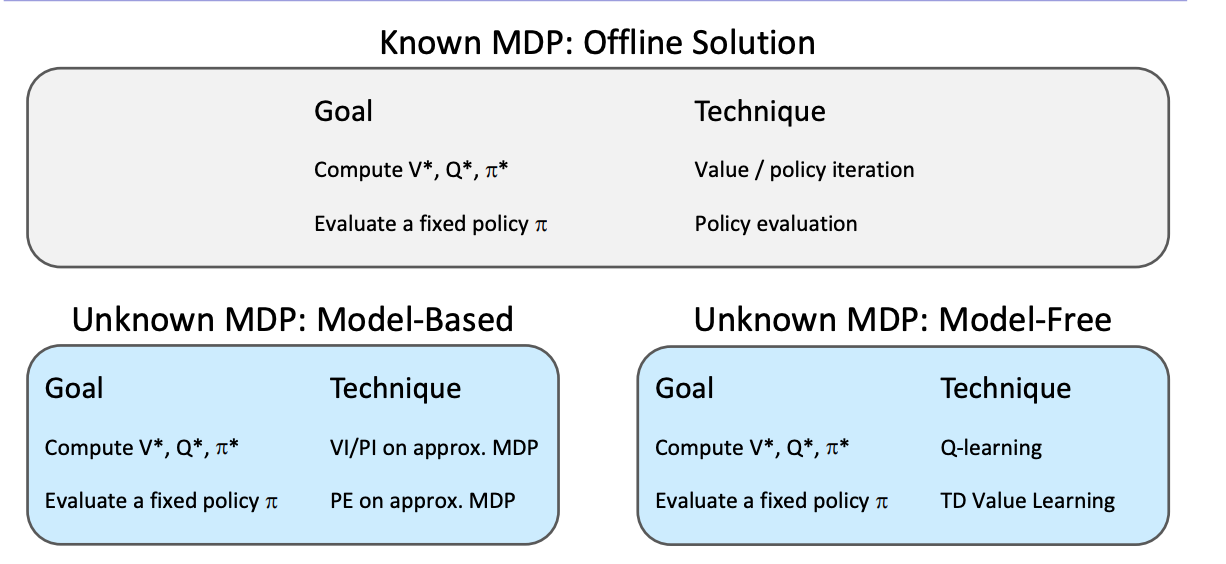
* TD Learning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
* But if we want to **turn values into a new policy**, we’re stuck
* **Idea**:
  + **learn Q-values**, not values
  + Makes action selection model-free too!
* **Q-Value iteration:**



* **Q-Learning:** sample based Q-value iteration ^^
  + Learn Q(s,a) values as you go
  + Consider your old estimate: Q(s,a)
  + Consider the new sample estimate (R(s,a,s’) + gamma max Q\_k(s’, a’))
  + Incorporate the new estimate into a running average

LEC 8: Reinforcement Learning 2

What we know so far:



Model-Free Learning:

* Experience world through episodes
* update estimates on each transition (s,a,r,s’)
* Over time, updates will mimic Bellman updates

Q-Learning:

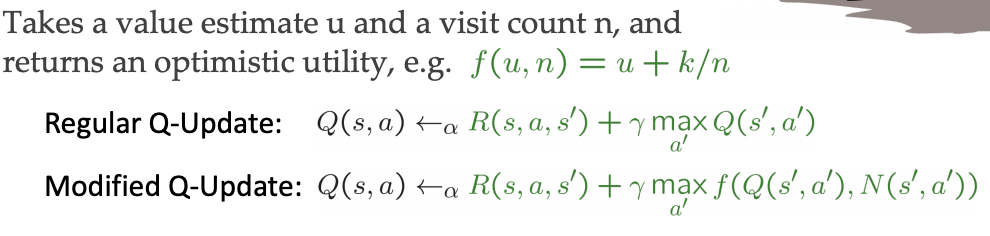
* Q-learning converges to optimal policy -- even if you’re acting suboptimally!
  + This is called **off-policy learning**
* Caveats:
  + You have to explore enough
  + You have to eventually make the learning rate small enough
    - … but not decrease it too quickly
  + Basically, in the limit, it doesn’t matter how you select actions

Exploration vs Exploitation:

* How to explore:
  + Simplest (**ε-greedy**)
    - With small probability ε, act randomly
    - With large probability 1- ε, act on current policy
  + Problems with random:
    - Eventualy explore the space, but keep thrashing around once learning is done
    - **One-soln:** lower ε over time
    - **Another soln**: exploration functions

Exploration Functions:

* When to explore:
  + **Random actions:** explore a fixed amount
  + **Better idea:** explore area whose values have not yet been established, eventually stop-exploring
* **Simple Exploration function:**



* **k** is a bonus, **N** is the number of times you have explored
  + inflate the value of the next state by a bonus divided by the number of times you have visited that state/done that action
  + The bonus propagates back to states that lead to unknown states as well (rewards agent for exploring as well as taking long paths to great rewards)

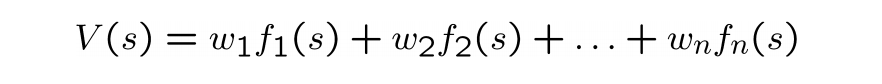
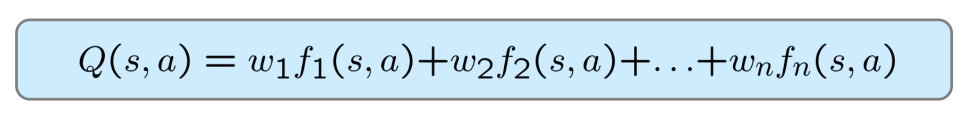
Regret:

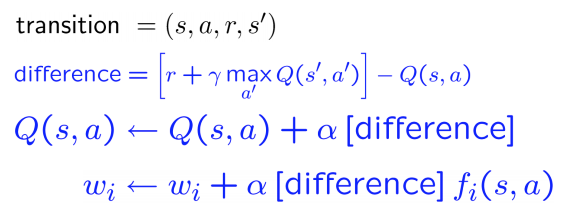
* **Regret is a measure of your total mistake cost**
  + Difference between (expected) rewards (including youthful sub optimality) and optimal (expected) rewards
* Minimizing regret requires **optimally learning** to be optimal

Generalizing Across States:

* **Basic** Q-Learning keeps a **table of all q-values**
* **Problem**; In realistic situations, we cannot possibly learn about every single state! o Too many states to visit them all in training
  + Too many states to hold the Q-tables in memory
* Instead, we want to generalize:
* Learn about some small number of training states from experience
  + **Generalize that experience** to new, similar situations
  + This is a fundamental idea in **machine learning**, and we’ll see it over and over again



* Solution**: Feature-Based Representation**
  + describe a state using a vector of features (properties)
  + Example features:
    - distance to closest ghost
    - distance to closest dot
    - number of ghosts
    - 1 / (dist to dot)
* **Linear Value Functions:**
  + **Advantage**: experience summed up in a few powerful numbers
  + **Disadvantage:** States may share features but actually be very different in value

Approximate Q-Learning:

* Q-learning with linear Q-functions:
* **Intuitive interpretation:** 
  + Adjust weights of active features
  + **E.g.,** if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state’s features

Linear Approximation: **Regression**

* **Optimization:** Least Squares\* (minimizing error)

Policy Search (Simplest):

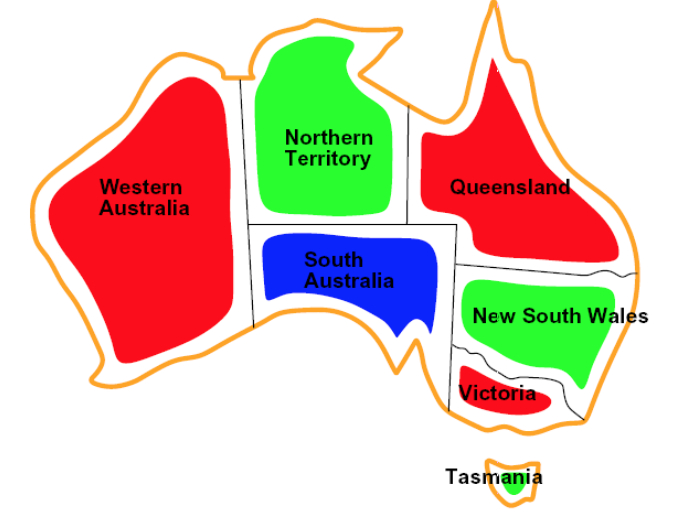
* Start with an initial linear estimator (e.g., random weights on features, like the ones you used for Q-learning)
* Nudge each feature weight up and down and see if your policy is better than before
* Problems:
  + How do we tell the policy got better?
  + Need to run many sample episodes!
  + If there are a lot of features, this can be impractical

LEC 9: Constraint Satisfaction Problems

­­CSP's

* A special subset of search problems
* **State** is defined by **variables Xi** 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 w/ more power than standard search algs

Example:

Variables:

* WA, NT, Q, NSW, V, SA, T

Domains:

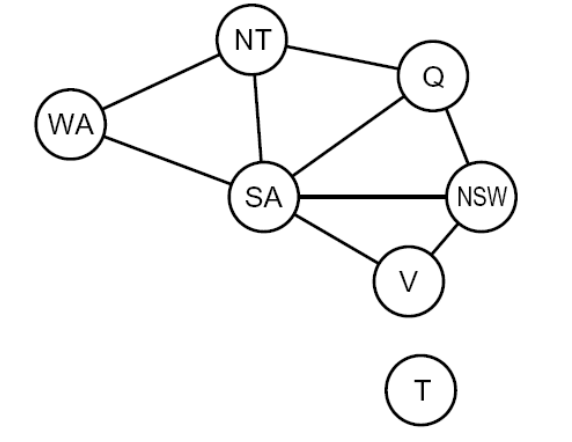
* D = {red, green, blue}

Constraints: adjacent regions must have different colors

* **Implicit:** WA ≠ NT
* **Explicit:** (WA, NT) in {(red, green), (red, blue), …}

Solutions are assignments satisfying all constraints, e.g.:

* {WA = red, NT = green, Q = red, NSW= green, V = red, SA = blue, T = green}

Constrain Graphs:

* **Binary CSP**: each constraint relates (at most) two variables
* **Binary constraint graph**: nodes are variables, arcs show constraints

Varieties of CSPs:

* Discrete Variables
  + **Finite domains (doing these in 188)**
    - Size d means O(d n ) complete assignments
    - E.g., Boolean CSPs, including Boolean satisfiability (NP-complete)
  + **Infinite domains** (integers, strings, etc.)
    - E.g., job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable
* Continuous variables
  + E.g., start/end times for Hubble Telescope observations
  + Linear constraints solvable in polynomial time by Linear Programming methods
* Varieties of Constraints:
  + **Unary constraints** involve a single variable (equivalent to reducing domains)
  + **Binary constraints** involve pairs of variables
  + **Higher-order constraints** involve 3 or more variables:
    - cryptarithmetic column constraint
* Standard Search Formulation:
  + States defined by the values assigned so far (partial assignments):
    - **Initial state**: the empty assignment, {}
    - **Successor function:** assign a value to an unassigned variable
    - **Goal test:** the current assignment is complete and satisfies all constraints

Backtracking Search:

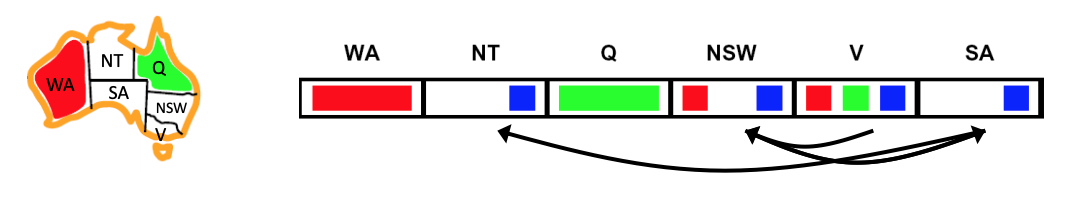
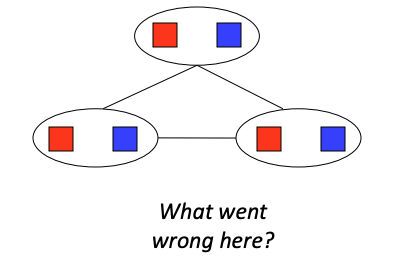
* **Idea 1:** One variable at a time
  + Variable assignments are commutative, so fix ordering
    - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  + Only need to consider assignments to a single variable at each step
* **Idea 2:** Check constraints as you go
  + I.e. consider only values which do not conflict with 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** (not the best name)

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:
* **Arc Consistency**
  + 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
  + 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



* Limitations of Arc Consistency:
  + After enforcing arc consistency:
    - Can have one solution left
    - Can have multiple solutions left
    - Can have no solutions left (and not know it)
  + Arc consistency still runs inside a backtracking search!
    - Just assigning consistency between 2 variables is not enough in this case 🡪

Ordering:

* Variable Ordering: **Minimum Remaining Values:**
  + Choose the variable with the fewest legal values left in its domain
  + Why min rather than max?
    - Going to have to assign everything, try the hardest variable first, might as well see if it works first instead of backtracking through everything afterwards
* Value Ordering: **Least Constraining Value:**
  + 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)

K-Consistency:

* **1-Consistency (Node Consistency):** Each single node’s domain has a value which meets that node’s unary constraints
* **2-Consistency** (**Arc Consistency**): For each pair of nodes, any consistent assignment to one can be extended to the other
* **K-Consistency:** For each k nodes, any consistent assignment to k-1 can be extended to the k th node.
  + Higher k more expensive to compute

LEC 10: CSP’s II

Structure:

* Extreme Case:
  + Independent sub-problems
* Independent sub-problems are identifiable as connected components of constraint graph
* Suppose a graph of n variables can be broken into n/c subproblems of only c variables each:
  + Worst case solution cost is O((n/c)d^c), linear in n
  + e.g:
    - 2^80 = 4 billion years
    - 4\*(2^20) = 0.

Tree Structured CSPs:

* **Theorem**:
  + if the constraint graph has no loops, the CSP can be solved in O(nd^2) time
  + Compare to general CSPs where the worst case time is O(d^n)
* **Algorithm**:
  + **Order:** choose a root variable, order variables so that parents precede children
  + **Remove backward:** For I = n : 2, apply **RemoveInconsistent(**Parent(), )
  + **Assign forward:** For I = 1 : n, assign consistently with Parent()
* After backward pass, all root-to-leaf arcs are consistent
* If root-to-leaf arcs are consistent, forward assignment will not backtrack

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(), very fast for small c

Iterative Improvement (**Min-Conflicts algorithm**):

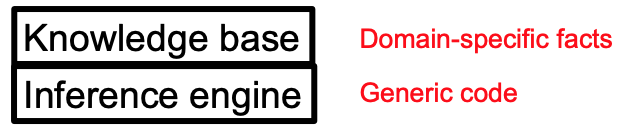
* Local search methods typically work with “complete” states ( all variables assigned)
* **To apply to CSPs:**
  + Take an assignment with unsatisfied constraints
  + Operators reassign variable values
  + No tree, no fringe! “New age” algorithm
* **Algorithm: while not solved:**
  + **variable selection:** randomly select any conflicted variable
  + **value selection:** min conflicts heuristic
    - choose a value that violates the fewest constraints
* Algorithms is very fast for any randomly-generated CSP except in a narrow range of the **the critical ratio**:
  + R =
  + Given random initial state, however, can solve n-queens in almost constant time for arbitrary n with high probability (e.g. n = 10,000,000)

Local Search:

* Improve a single option until you can’t make it better
* **New successor function**: local changes
* Generally much faster and more memory efficient (but incomplete and suboptimal)
* **Algorithm (Hill Climbing)**
  + Start wherever
  + Repeat: most to best neighboring state
  + if no state is better, stop
* **Algorithm (Simulated Annealing)**
* Escape local maxima by allowing downhill moves
* But make them rarer as time goes on
* Theoretical guarantee
* If T decreased slowly enough, will converge to optimal state

LEC 11: Propositional Logic, Semantics, Inference, Agents

Knowledge

* **Knowledge Base** = set of sentences in a formal language
* Declarative approach to build an agent
  + **Tell** it what it needs to know (or have it **Learn** the knowledge)
  + Then it can **Ask** itself wh at to do–answers should follow from the KB
* A single inference algorithm can answer any answerable question
  + Cf. a search algorithm answers only “how to get from A to B” questions

Logic:

* **Syntax:** What sentences are allowed?
* **Semantics:**
  + What are the **possible worlds**?
  + Which sentences are **true** in which worlds?
* **Propositional Logic**
  + Syntax: P ∨ (¬Q ∧ R)
  + Possible world: {P=true,Q=true,R=false,S=true} or 1101
  + Semantics: α ∧ β is true in a world iff α is true and β is true (etc.)
* **First-Order Logic**
  + Syntax: ∀x ∃y P(x,y) ∧ ¬Q(Joe,f(x)) ⇒ f(x)=f(y)
  + Possible world: Objects o1, o2, o3; P holds for ; Q holds for < o1, o3>; f(o1)=o1; Joe=o3; etc.
  + Semantics: φ(σ) is true in a world if σ=oj and φ holds for oj ; etc.

Inference:

* **Entailment:**
  + α |= β (“α entails β” or “β follows from α”) iff in every world where α is true, β is also true
    - I.e., the **α-worlds** are a **subset** of the β-worlds [models(α) ⊆ models(β)]
* **Proofs:**
  + A proof is a demonstration of entailment between α and β
  + Method 1: **model-checking**
    - For every possible world, if α is true make sure that is β true too
    - OK for propositional logic (finitely many worlds); not easy for first-order logic
  + Method 2: **theorem-proving**
    - Search for a sequence of proof steps (applications of **inference rules**) leading from α to β
    - E.g., from P ∧ (P ⇒ Q), infer Q by **Modus Ponens**
* **Sound algorithm**:
  + everything it claims to prove is in fact entailed
* **Complete algorithm**:
  + every that is entailed can be proven

Simple theorem proving: **Forward Chaining**

* **Forward chaining** applies Modus Ponens to generate new facts:
  + Given X1 ∧ X2 ∧ … Xn ⇒ Y and X1, X2, …, Xn
  + Infer Y
* Forward chaining keeps applying this rule, adding new facts, until nothing more can be added
* Requires KB to contain only **definite clauses**
* **Properties** (Sound and Complete)

Satisfiability and Entailment:

* A sentence is **satisfiable** if it is true in at least one world (cf CSPs!)
  + Suppose α |= β
  + Then α ⇒ β is true in all worlds
  + Hence ¬(α ⇒ β) is false in all worlds
  + Hence α ∧ ¬β is false in all worlds, i.e., unsatisfiable
* So, add the negated conclusion to what you know, test for (un)satisfiability; also known as **reductio ad absurdum** (proof by contradiction)

Conjuctive Normal Form (CNF):

* Every sentence can be expressed as a **conjunction** of **clauses**
* Each clause is a **disjunction** of **literals**
* Each literal is a symbol or a negated symbol
* Replace implies with its –α V β form, etc.

Efficient Satisfiability solvers (provide efficient inference):

* DPLL (Davis-Putnam-Logemann-Loveland) is the core of modern solvers
* Essentially a backtracking search over models with some extras:
  + **Early termination:** stop if
    - all clauses are satisfied; e.g., (A ∨ B) ∧ (A ∨ ¬C) is satisfied by {A=true}
    - any clause is falsified; e.g., (A ∨ B) ∧ (A ∨ ¬C) is satisfied by {A=false,B=false}
  + **Pure literals:** if all occurrences of a symbol in as-yet-unsatisfied clauses have the same sign, then give the symbol that value
    - E.g., A is pure and positive in (A ∨ B) ∧ (A ∨ ¬C) ∧ (C ∨ ¬B) so set it to true
  + **Unit clauses:** if a clause is left with a single literal, set symbol to satisfy clause
    - E.g., if A=false, (A ∨ B) ∧ (A ∨ ¬C) becomes (false ∨ B) ∧ (false ∨ ¬C), i.e. (B) ∧ (¬C)
    - Satisfying the unit clauses often leads to further propagation, new unit clauses, etc.

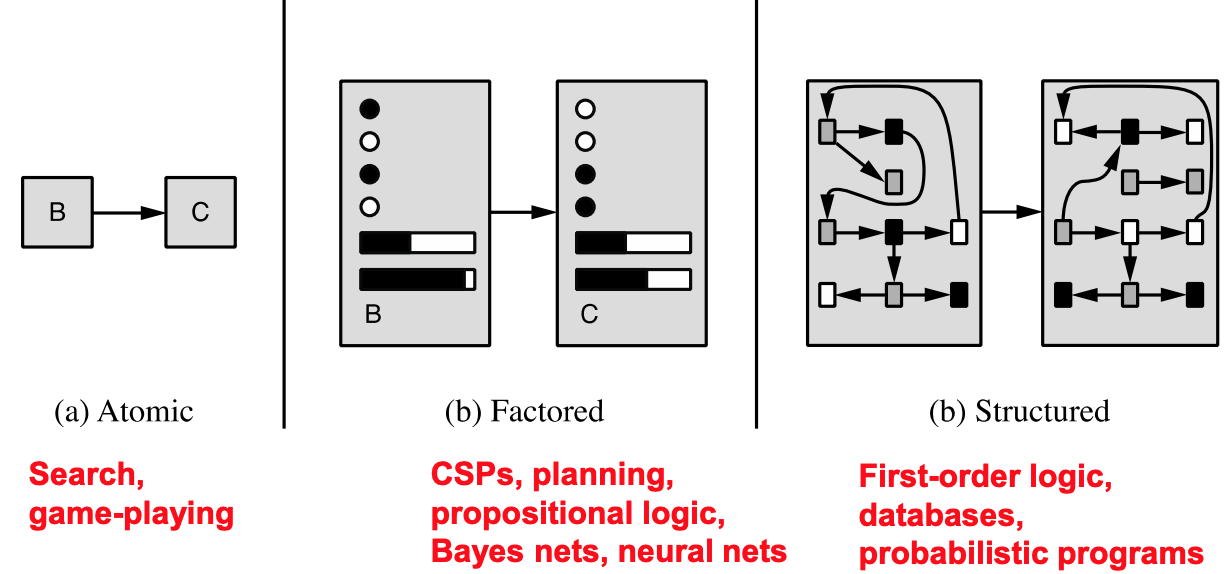
Example:

* **Initial State**
  + Pacman may know its initial location:
    - At\_1,1\_0 ∧ ¬At\_1,2\_0 ∧ ¬At\_1,3\_0 ….
  + Or, it may not:
    - At\_1,1\_0 v At\_1,2\_0 v At\_1,3\_0 v … v At\_3,3\_0
  + We also need a domain constraint – exactly one thing at a time
    - ¬(W\_0 ∧ E\_0) ∧ ¬(W\_0 ∧ S\_0) ∧ …
    - ¬(W\_1 ∧ E\_1) ∧ ¬(W\_1 ∧ S\_1) ∧ …
    - … ∧ (W\_0 v E\_0 v N\_0 v S\_0) ∧ …
* **Transition Model:**
  + A state variable gets its value according to a **successor-state axiom**
    - Xt ⇔ [Xt-1 ∧ ¬(some actiont-1 made it false)] v [¬Xt-1 ∧ (some actiont-1 made it true)
* **Goal:**
  + Alive\_T ∧ —Food\_1,1\_T ∧ ­—Food\_1,2\_2 ∧ …

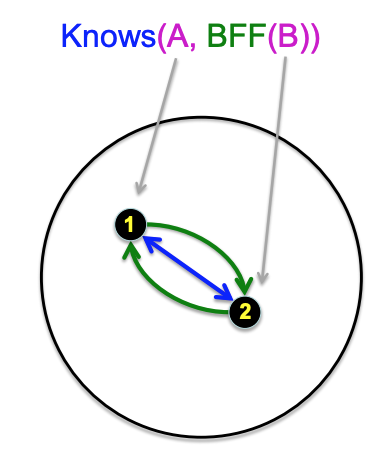
LEC 12: First-Order Logic (FOL)

**Propositional logic** is in sufficient and inefficient for solving problems 🡪 First-Order Logic

* Example:
  + Rules of chess: 100,000 page in propositional logic
  + 1 page in first-order logic



Possible worlds for FOL:

* Consists of:
  + A non-empty set of objects
  + For each k-ary predicate in the language, a set of k-tuples of objects (i.e., the set of tuples of objects that satisfy the predicate in this world)
  + For each k-ary function in the language, a mapping from k-tuples of objects to objects
  + For each constant symbol, a particular object (can think of constants as 0-ary functions)

Syntax and Semantics:

* **Terms refer to objects**
  + Constant symbols
  + Function symbols
  + Logical variables
* **Atomic Sentences: Elementary proposition (CF symbols in PL)**
  + A predicate symbol with terms as arguments
  + An equality between terms
* **Complex Sentences**:
  + Sentences with logical connectives
    - ¬α, α ∧ β, α ∨ β, α ⇒ β, α ⇔ β
  + Sentences with universal or existential quantifiers
    - ∀x... or ∃x...

Inference in FOL (not heavily covered on test, just know gen idea):

* Entailment is defined exactly as for PL:
  + α |= β (“α entails β” or “β follows from α”) iff in every world where α is true, β is also true
* Given an existentially defined query, provide answer in the form of a **substitution** (or **binding)** for the variables
  + Applying the substitution should produce a sentence that is entailed by KB
  + Ex:
    - **KB** = ∀x Knows(x,Obama)
    - **Query** = ∃y∀x Knows(x,y)
    - **Answer** = Yes, {y/Obama}
* **Propositionalization:**
  + Convert (KB ∧ ¬α) to PL, use a PL SAT solver to check (un)satisfiability
  + Trick:
    - replace variables with ground terms, convert atomic sentences to symbols
  + Real trick:
    - for k = 1 to infinity, use terms of function nesting depth k
* **Lifted Inference**
  + Apply inference rules directly to first-order sentences, e.g.,
  + Examples:
    - Prolog (backward chaining), Datalog (forward chaining), production rule systems (forward chaining), resolution theorem provers

LEC 13: Probability

Uncertainty:

* **Observed variables (evidence)**
  + Agent knows certain things about the state of the world
* **Unobserved variables**
  + Agent needs to reason about other aspects
* **Model**
  + Agent knows something about how the known variables relate to the unknown variables

Random Variable:

* Some aspect of the world about which we (may) have uncertainty (deterministic function from a possible world to some range of values)
* **Probability Distributions:**
  + Associate a probability with each value
  + Unobserved random variables have distributions
  + A distribution for a **discrete** **variable** is a **TABLE** of probabilities of values
* **Joint Distributions over a set of random variables:**
  + Specifies a real number for possible combination of values in the set

Probability Model:

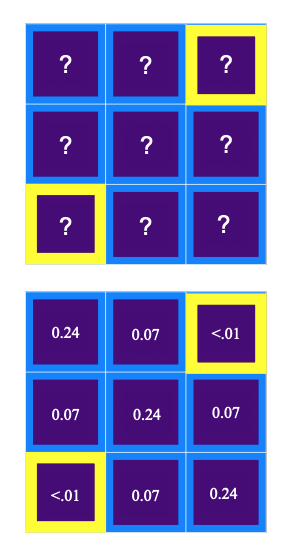
* A joint distribution over a set of random variables
  + Random variables with domains
  + Assignments are called outcomes
  + Joint distributions say whether assignments are likely
  + Ideally only certain variables directly interact
* Can compare with Constraint Satisfaction Problems
  + Variables with domains, constraints state whether assignments are possible, ideally only certain variables interact
* **Event:**
  + Event E is a set of outcomes
* **Marginal Distribution:** Sub-tables which eliminate variables
  + **Marginalization (summing out):** combine collapsed rows by adding
* **Conditional Probability:**
  + Simple relation between joint and conditional probabilities
    - P(a | b) = P(a,b) / P(b)
  + Normalizing:
    1. Compute Z = sum over all entries
    2. Divide by every entry by Z
* **Product Rule**
  + P(x | y) \* P(y) = P(x, y)
* **Chain Rule**
  + P(x1, x2, … , xn) = P(x1) \* P(x2 | x1) \* P(x3 | x1, x2)…
* **Bayes’ Rule**
  + P(x | y) = (P( y | x) \* P(x)) / P(y)

Probabilistic Inference:

* Compute a desired probability from other known probabilities
  + E.g. conditional from joint
* Generally compute conditional probabilities
  + These represent the agent’s belief given the evidence
  + Observing new evidence causes these probabilities to change
* **Inference by Enumeration**
  + X1, X2, ..., Xn = All variables
    - Evidence Variables (E1, E2, …, Ek)
    - Query\* variables (Q)
    - Hidden variables (H1, …, Hr)
  + We want P(Q | e1 … ek)
  + Steps:
    1. Select the entries consistent with the evidence
    2. Sum out H to get joint of query and evidence
    3. Normalize
  + Problems:
    - Worst-case time complexity O(d^n)
    - Space complexity O(d^n) to store joint distribution

Independence:

* Two variables are independent if:
  + Joint Distribution = Product of marginal distributions
  + P(x, y) = P(x) \* P(y)
  + P(x | y) = P(x)
* Unconditional (absolute) independence is very rare (most variables are correlated)
* **Conditional Independence**
  + Example:
    - P(Toothache, Cavity, Catch)
    - The probability that the probe catches the cavity doesn’t depend on whether I have a toothache:
      * P(catch | toothache, cavity) = P(catch | cavity)



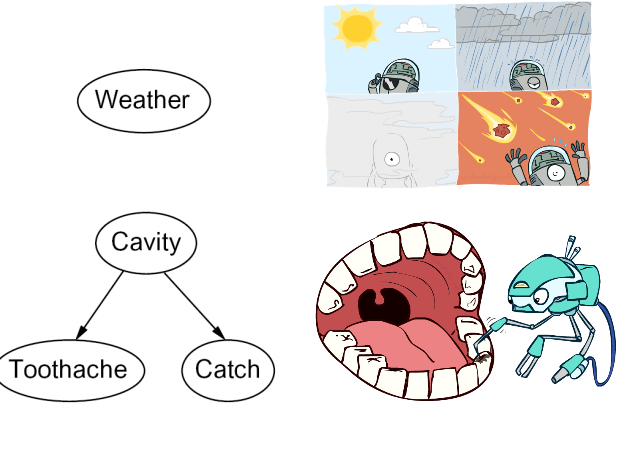
LEC 14: Bayes’ Nets

Bayesian updating:

* Using low-dimensional conditional distributions
* Ghostbusters example (ghost readings based on probabilities)
  + Given two reading P(r1, r2 | g), the readings are conditionally independent given the ghost location!
    - P(r1, r2 | g) = P(r1 | g) \* P(r2 | g)

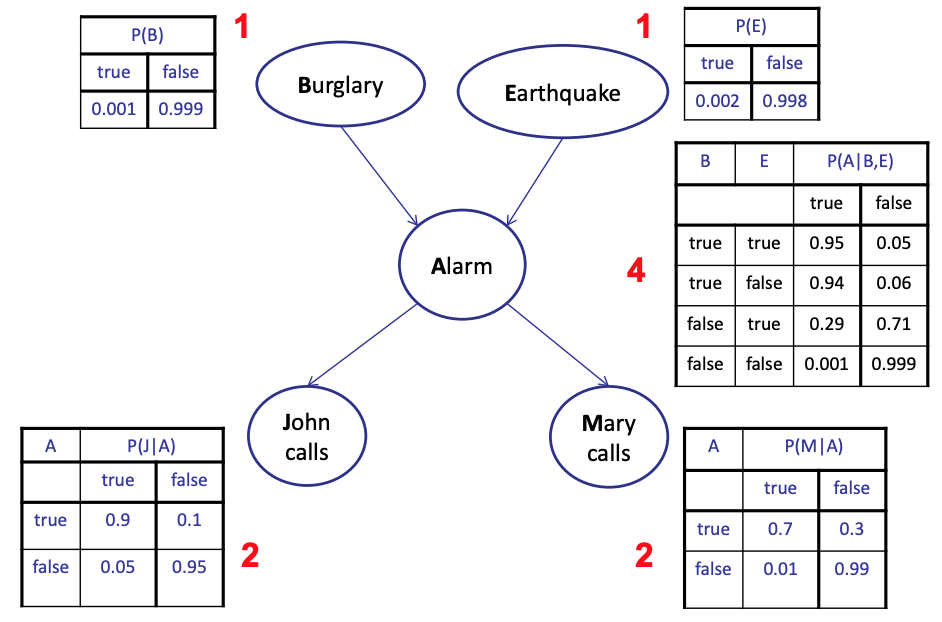
Bayes Nets: Big Picture

* **Bayes nets:**
  + a technique for describing complex joint distributions (models) using simple, local distributions (conditional probabilities)
  + Take advantage of **local causality**:
    - the world is composed of many variables,
    - each interacting locally with a few others
  + Bayes net = Topology (graph) + Local Conditional Probabilities

Graphical Model Notation:

* **Nodes: variables** (with domains)
  + Can be assigned (observed) or unassigned (unobserved)
* **Arcs: interactions** 
  + Indicate “direct influence” between variables
  + Formally: encode conditional independence (more later)

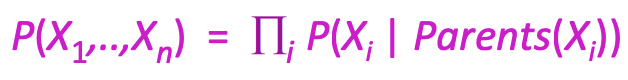
Bayes Net Syntax and Semantics:

* **Syntax:**
  + A set of nodes, one per variable Xi
  + A directed, acyclic graph (DAG)
  + A conditional distribution for each node given its **parent variables** in the graph
    - conditional probability table (**CPT) : each row is a distribution for child given a configuration of its parents**
  + Example: Alarm Network:
  + Number of free parameters in each CPT:
    - Parent domain sizes d1, …, dk
    - Each child domain size d
    - Each table row must sum to 1

Sparse Bayes Nets:

* Suppose n variables
  + Maximum domain size is d
  + Maximum number of parents is k
* Full joint distribution has size O(d^n)
* Bayes net has size O(n \* d^k)

Bayes Net Global Semantics:

* Bayes nets encode joint distributions as a product of conditional distributions on each variables:
* Every variable is **conditionally independent** of its non-descendants given its parents

LEC 15: Bayesian Networks Inference

Markov Blanket:

* Variable’s **Markov Blanket** consists of parents, children, children’s other parents
* Every **variable** is **conditionally independent** of **all other variables** given its Markov blankets

Inference:

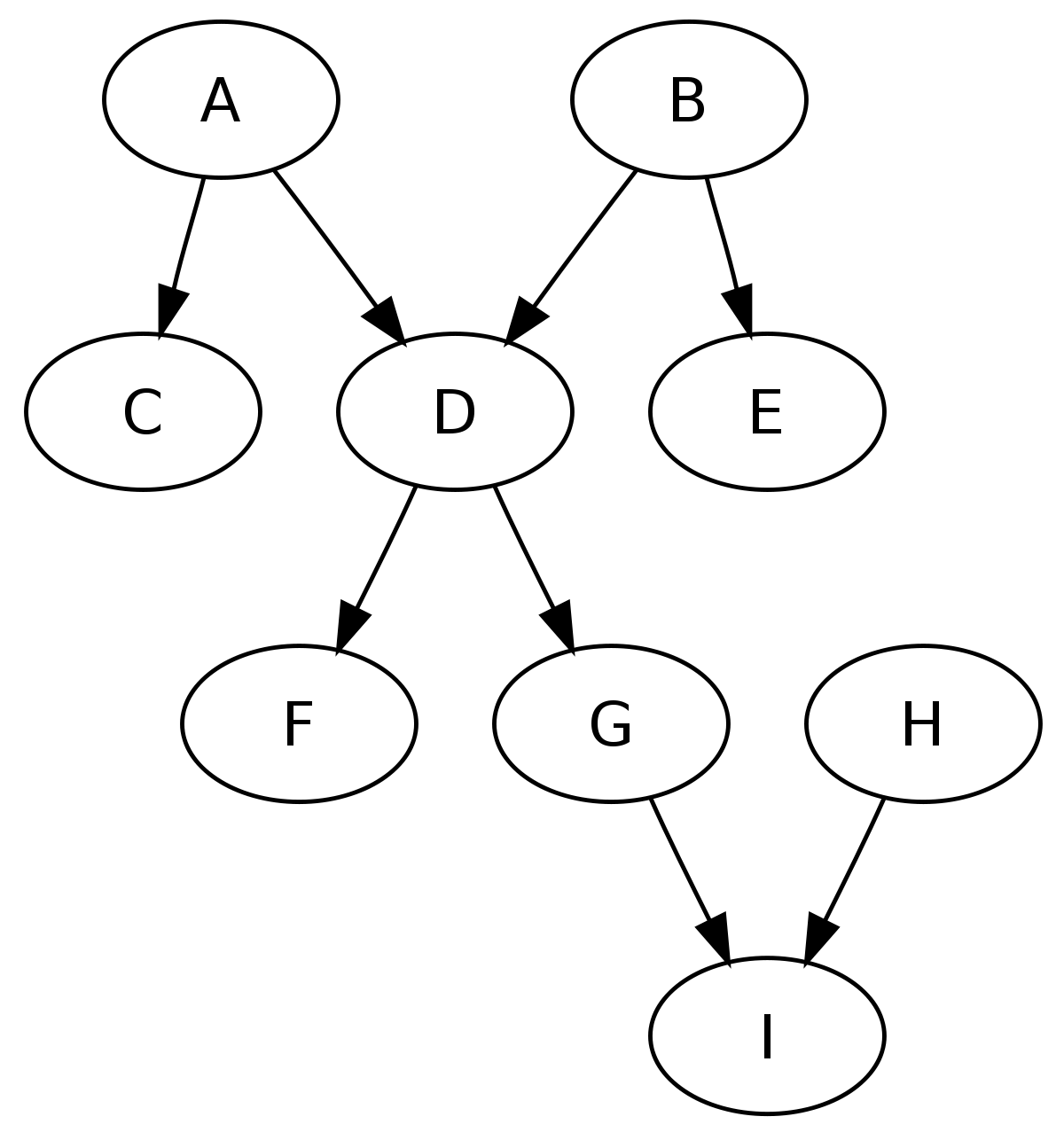
* Calculating some useful quantity from a probability model (joint probability distribution)
* **Inference by Enumeration in Bayes Net:**
  + Any probability of interest can be computed by summing entries from the joint distr.
  + Entries from the joint distr. Can be obtained from a BN by multiplying the corresponding conditional probabilities

Variable Elimination: The basic ideas

* Move Summations inwards as far as possible
* Do the calculation from the inside out
* Query: P(Q } E1 = e1, …, Ek = ek)
  + Start w initial factors While there are still
* Operation 1: Pointwise Point
  + Pointwise product of factors (similar to a database join, not matrix multiply)
    - New factor has union of variables of the two original factors
    - Each entry is the product of the corresponding entries from the original factors
* Operation 2: Summing out a Variable
  + or eliminating a variable from a factor

VE Computational and Space Complexity:

* Computational and space complexity of variable elimination is determined by the largest factor (and it’s the space that kills you)

Polytrees:

* Directed graph with no undirected cycles
* For poly-trees the complexity of variable elimination is **linear in the network size** if you eliminate the leaf towards the roots
  + This is essentially the same theorem as for tree-structured CSPs