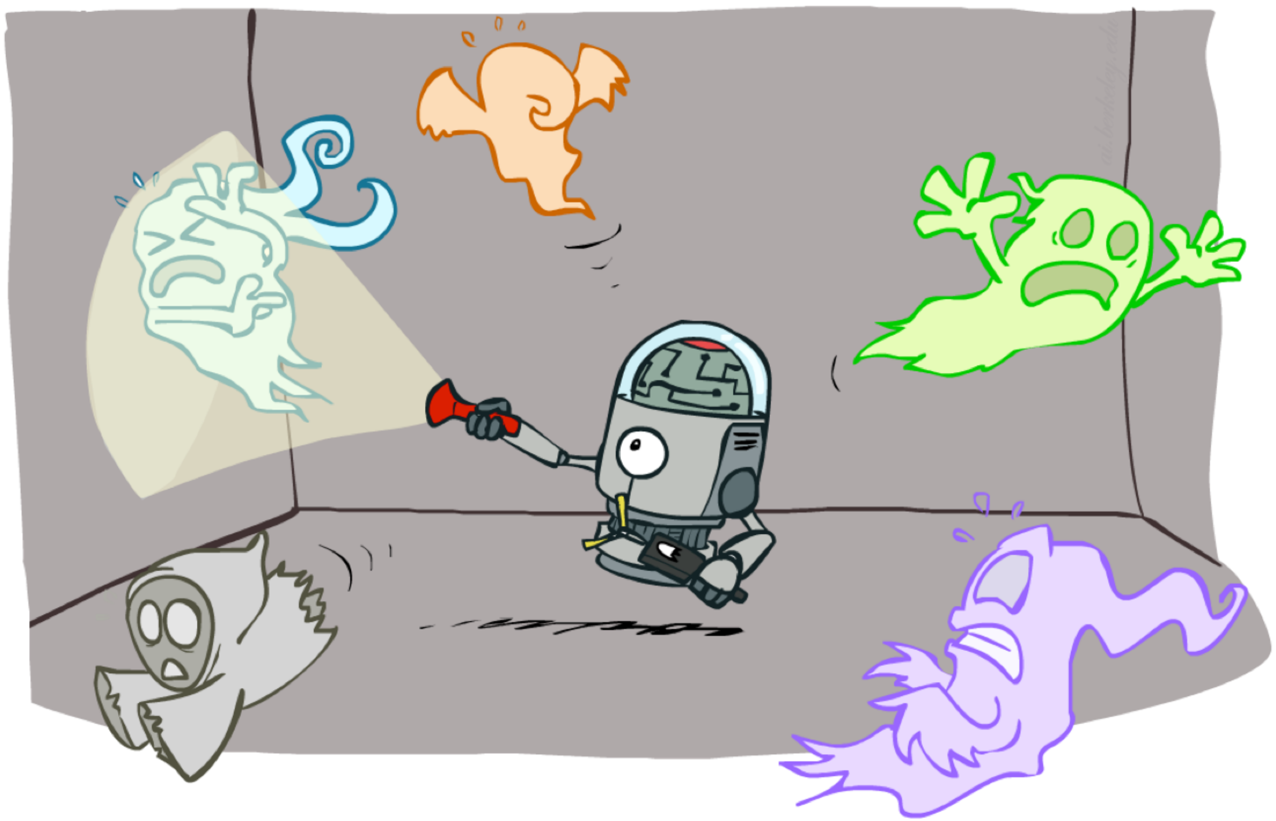
CS3317: Artificial Intelligence

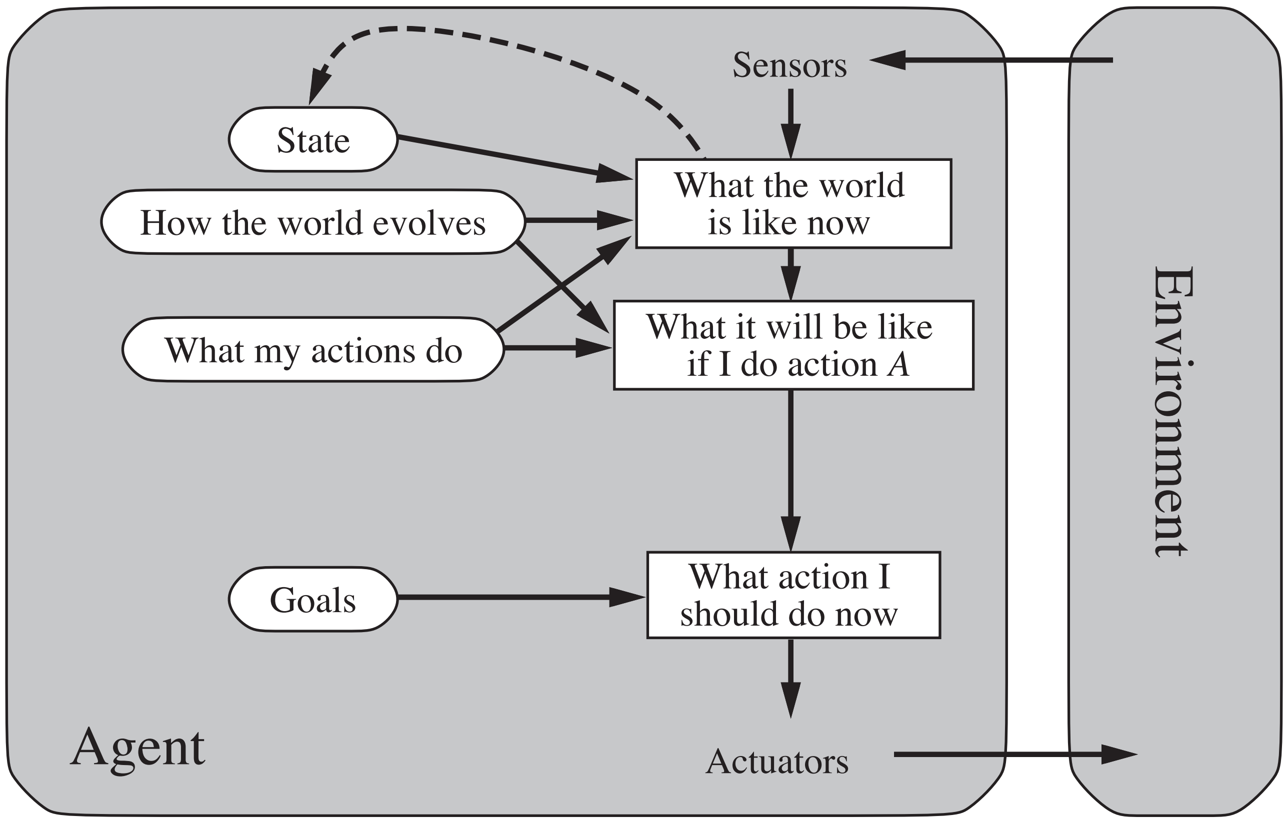
Review



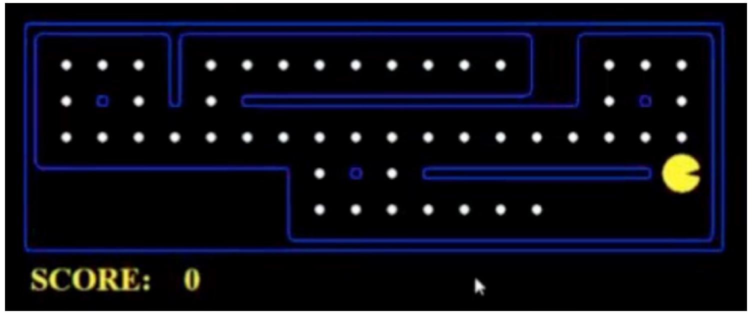
Instructor: Panpan Cai

[Slides adapted from UC Berkeley CS188]

Planning Agents, Search

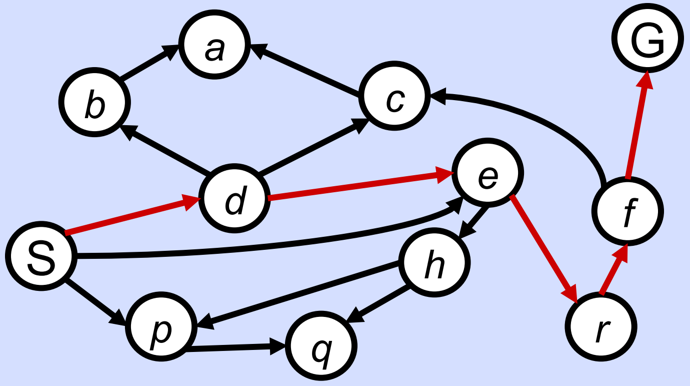


- Search problem:
  - States (configurations of the world)
  - Actions (associated with costs)
  - Successor function (world dynamics)
  - Start state
  - Goal test
- A *search state* keeps *only* the details needed for planning
  - Pathing: (x,y) location
  - Eating-all-dots: {(x,y), dot booleans}



State Space Graphs vs. Search Trees

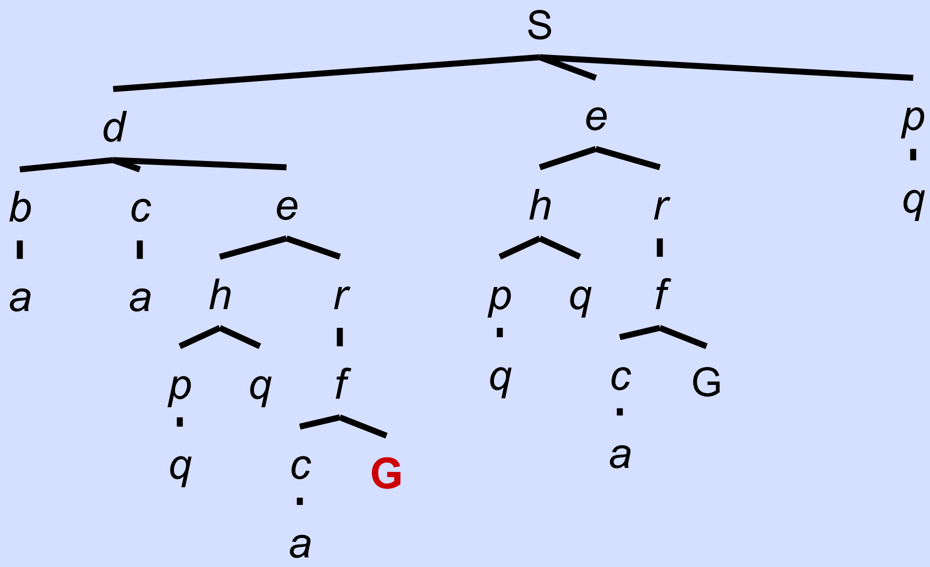
State Space Graph



Each *node* *in in the search tree is an entire* *path* *in the state space graph.*

Search algorithms construct a search tree *as little as possible* *to solve planning tasks.*

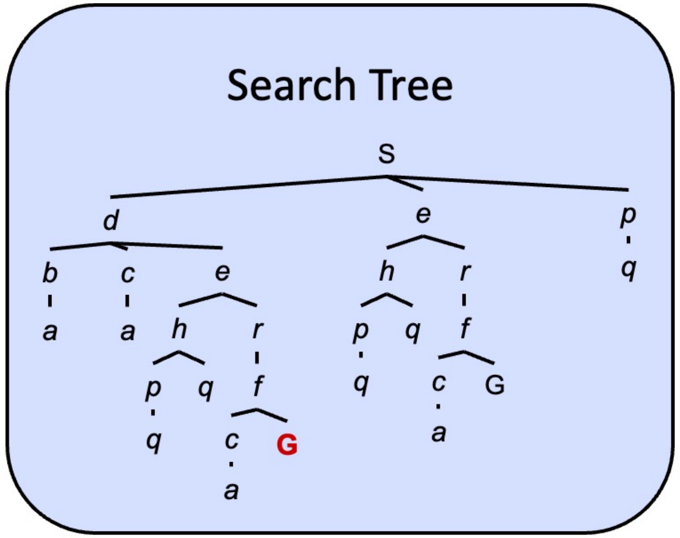
Search Tree



Tree Search Pseudo-code

- Core ideas:
  - Iteratively builds a search tree, until finding the goal
  - Maintains a fringe / priority queue of *unexpanded* nodes, to determine order of expansions

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```



Graph Search Pseudo-code

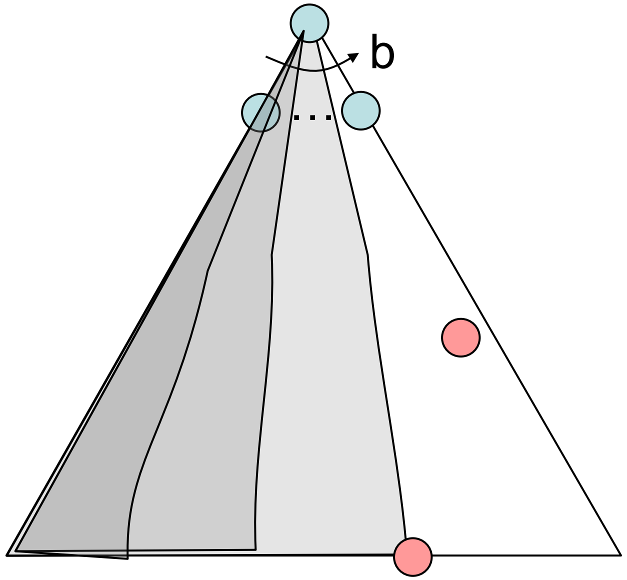
- Considers that tree search can repeatedly expand the same state
- *Graph search* maintains a *closed set* to avoid expanding a state more than once

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

Basic Search Algorithms

- *Different* search algorithms mostly differ in *ordering of fringe* or the *priority values*

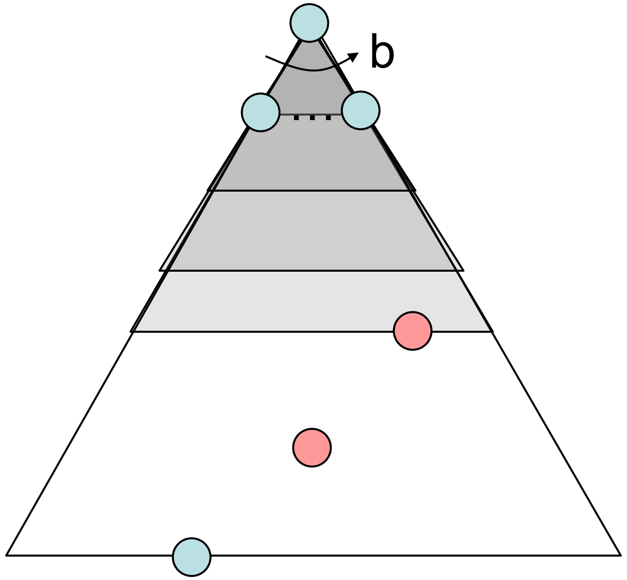
DFS



Expand the deepest node first

$f(n) = depth(n)$

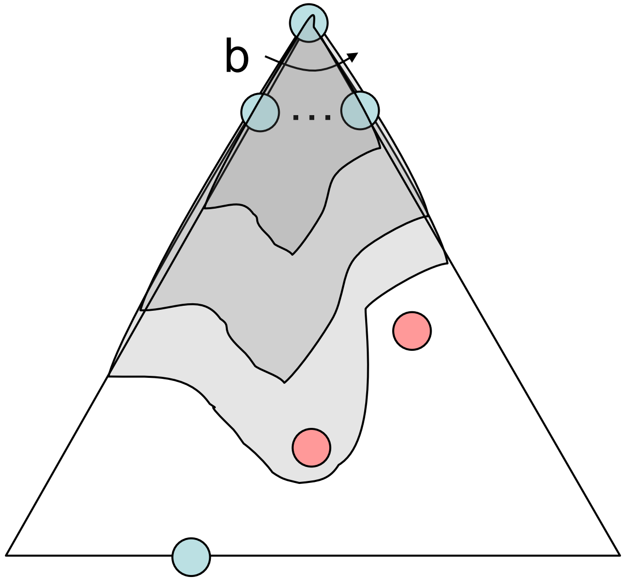
BFS



Expand the shallowest node first

$f(n) = -\ depth(n)$

UCS (Dijkstra)



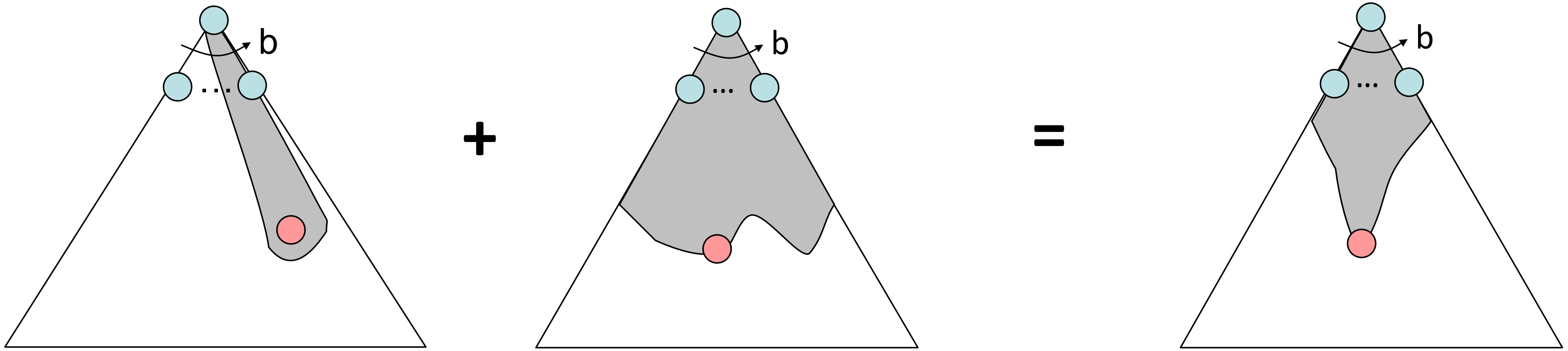
Expand the cheapest node first

$f(n) = g(n)$

Heuristic Search, A*

- A heuristic h is a function that *estimates* how close a state is to a goal (*cost-to-go*)

Greedy | Uniform-Cost | **A***



Greedy

f(n) = h(n)

Exploitation only, expand the node that seems closest to goal (*fast, not optimal*)

Uniform-Cost

f(n) = g(n)

Uniform exploration, not informed by goal at all (*slow, optimal*)

A*

f(n) = g(n) + h(n)

Optimally trading-off exploration and exploitation (*fast, and optimal!*)

A* Optimality Requires "Good" Heuristics

- Main idea: estimated heuristic costs ≤ actual costs
  - *Admissibility* *(tree search)*: heuristic cost-to-goal ≤ actual cost-to-goal

    $h(A) \leq$ actual cost from A to G

  - *Consistency* *(graph search)*: heuristic "arc" cost ≤ actual arc cost

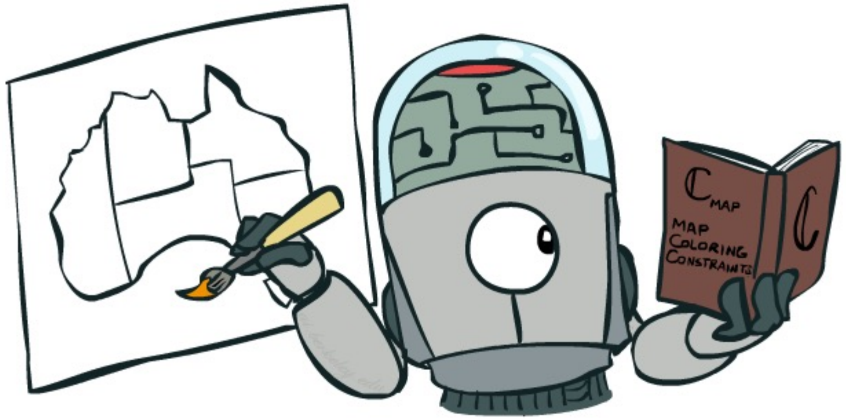
    $h(A) - h(C) \leq \text{cost}(A \text{ to } C)$

  - Construct admissible heuristics as solutions to *relaxed* problems
  - Can *combine* heuristics to get even better:

    $h(n) = \max(h_1(n), h_2(n))$

Constraint Satisfaction Problems

- *CSPs*: a particular type of search problem on assigning variables
  - Variables
  - Domains
  - Constraints
    - Implicit (provide code to compute)
    - Explicit (provide a list of the legal tuples)
    - Unary / Binary / N-ary



- Goals of CSPs:
  - Here: identify a solution

Backtracking Search

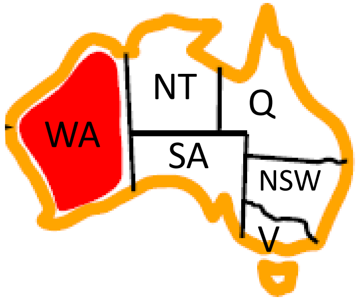
- Backtracking = DFS + variable-ordering + fail-on-violation
  - Consider assignments to a single variable at each step
  - Consider only values *not* in conflict with *previous* assignments

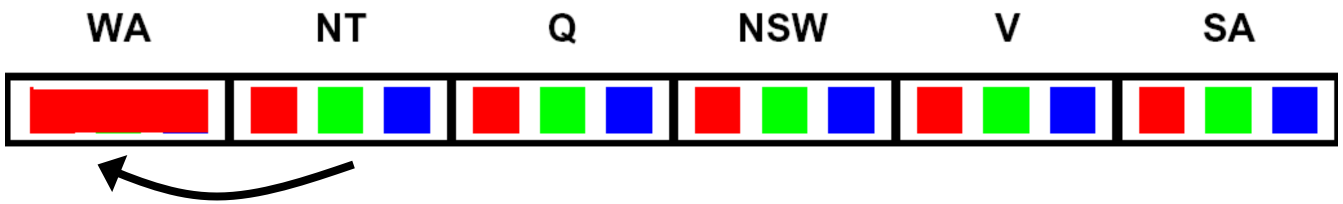
```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
       if value is consistent with assignment given CONSTRAINTS[csp] then
           add {var = value} to assignment
           result ← RECURSIVE-BACKTRACKING(assignment, csp)
           if result ≠ failure then return result
           remove {var = value} from assignment
   return failure
```

Filtering: conflict with future assignments

- *Consistency of Arcs*: An arc X $\rightarrow$ 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

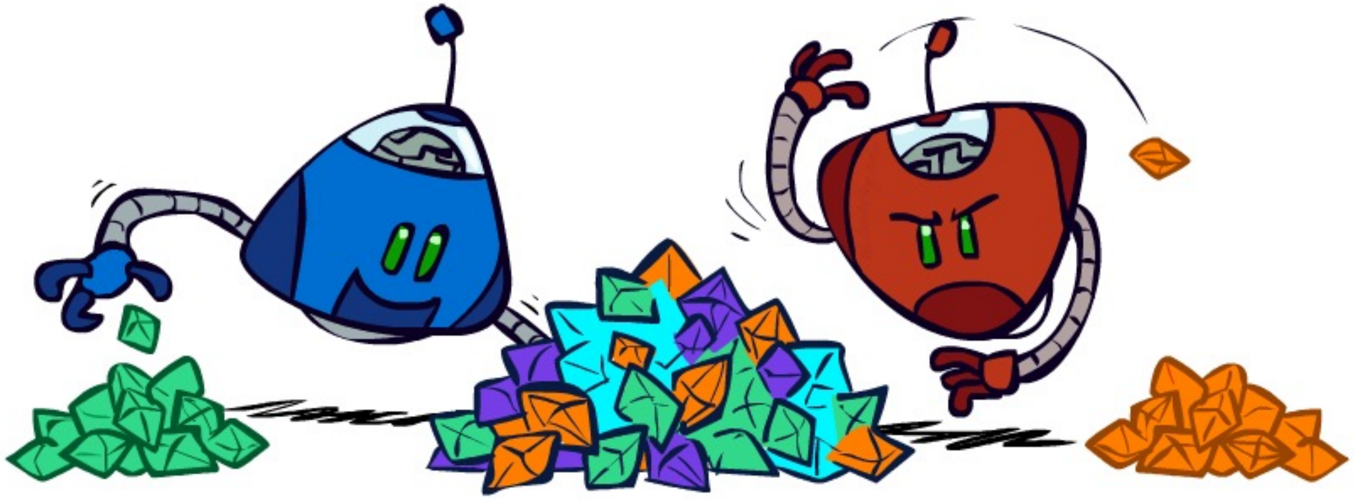




  - Tail = NT, head = WA
    - If NT = red: there is no remaining assignment to WA that we can use
    - Deleting NT = red from the *tail* makes this arc consistent

- Filtering algorithms:
  - *Forward checking*:
    - Enforces consistency of arcs pointing to the new assignment
  - *AC-3*:
    - Enforces consistency of all arcs in the CSP;
    - Whenever a node loses value, re-check all neighbors.

Games



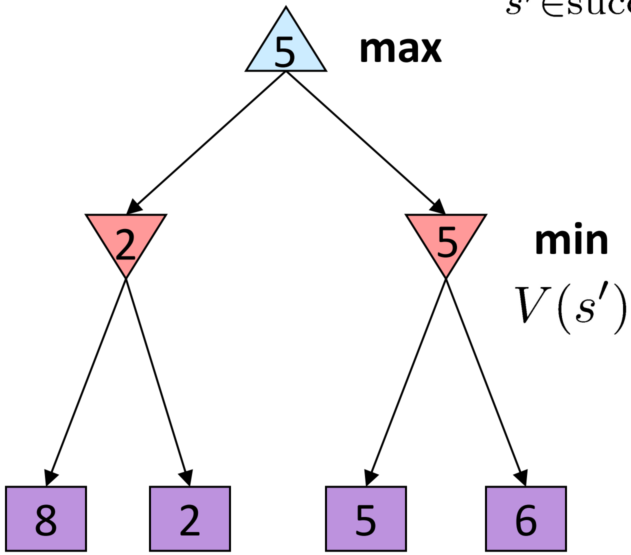
- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - A single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible

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 adversary

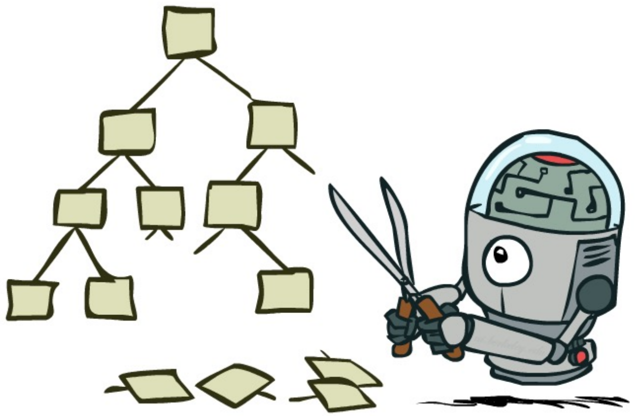
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$



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

Terminal values: part of the game

Alpha-Beta Pruning



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

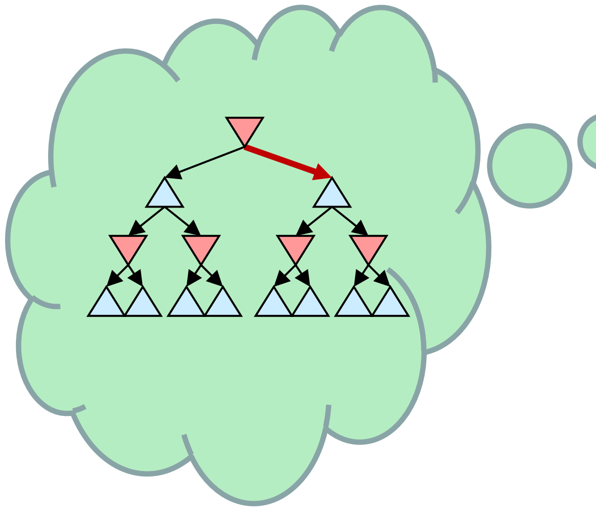
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

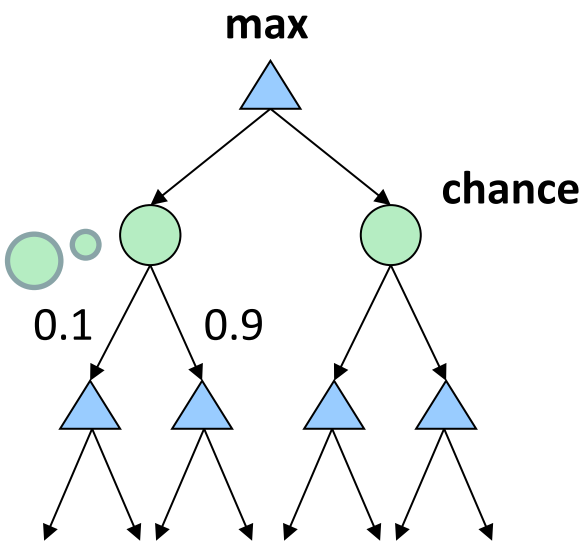
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Expectimax Search

- *Expectimax search* computes the average score under optimal play
- Values reflect *average-case* (expectimax) outcomes, *not worst-case* (minimax) outcomes
- *Max nodes* as in minimax search
- *Chance nodes* have uncertain outcomes
  - Probabilities from an opponent model
  - Calculate the *expected value* of successors

"Opponent model"

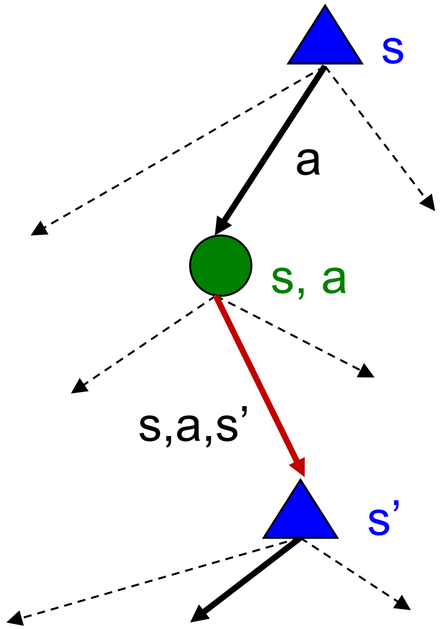





```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Markov Decision Processes

- Markov decision processes:
  - States S
  - Actions A
  - Transitions P(s'|s,a) (or T(s,a,s'))
  - Rewards R(s,a,s') (and discount γ)
  - Start state s_0



s is a *state*

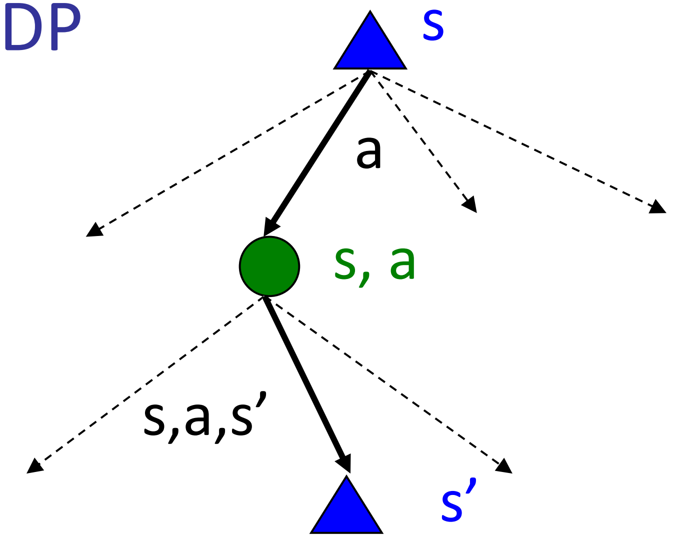
(s, a) is a *q-state*

(s,a,s') is a *transition*

- Quantities:
  - Policy = mapping from states to actions
  - Utility = sum of discounted rewards
  - Value= expected future utility from a state (max node)
  - Q-Value= expected future utility from a q-state (chance node)

The Bellman Equations

- Bellman equations characterize *optimal values* in an MDP



$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^*(s')\right]$$

Value Iteration

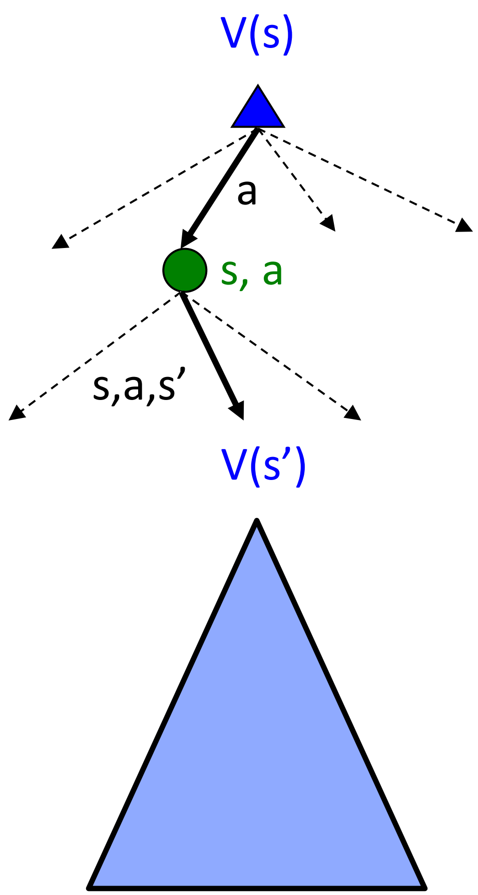
- Bellman equations *characterize* the optimal values:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

- Value iteration *computes* them:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

- Value iteration converges to optimal values



Policy Iteration

- Evaluation: For a *given* policy π, find values with policy evaluation:
  - Iterate until values converge:

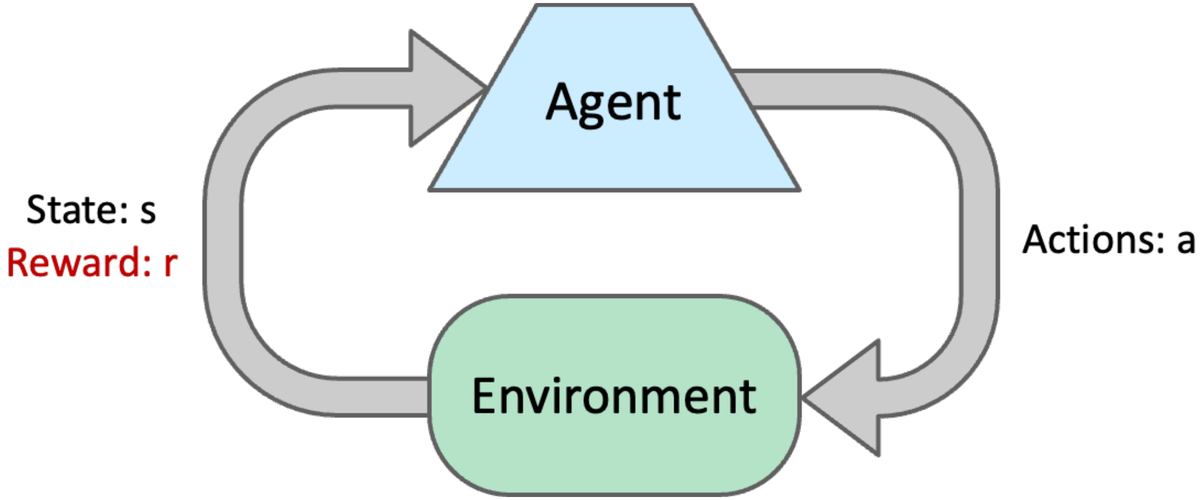
$$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]$$

- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^{\pi_i}(s') \right]$$

Reinforcement Learning

- Still assume a Markov decision process (MDP):
  - A set of states $s \in S$
  - A set of actions (per state) $A(s)$
  - A transition model $T(s,a,s')$
  - A reward function $R(s,a,s')$
- Still looking for a policy $\pi(s)$



- New twist: don't know T or R
  - I.e. we don't know the consequence of actions and goodness of states
  - Must explore new states and actions
    - -- to bravely go where no robot has gone before

Model-Based RL

- Core ideas:
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

- Step 1: Learn empirical MDP model
  - Given a set of experiences {…, (*s*,*a*,*s'*,r), …}
  - Estimate each probability in *T*(*s*,*a*,*s'*) from counts
    - Count the frequency of visiting *s'* for each (*s*,*a*) pair
    - Fill number in transition table
  - Discover each *R*(*s*,*a*,*s'*) when we experience the transition
    - Fill number in reward table
- Step 2: Solve the learned MDP
  - Use, e.g., value or policy iteration

T(*s*,*a*,*s'*)

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
…

R(*s*,*a*,*s'*)

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Model-free RL: TD Learning

- Learn $V^{\pi}(s)$ as you go
  - Receive a transition <$s, \pi(s), s', r$>
  - Consider your old estimate: $V(s)$
  - Consider your new sample estimate:

    $$sample = R(s,\pi(s),s') + \gamma V^{\pi}(s')$$

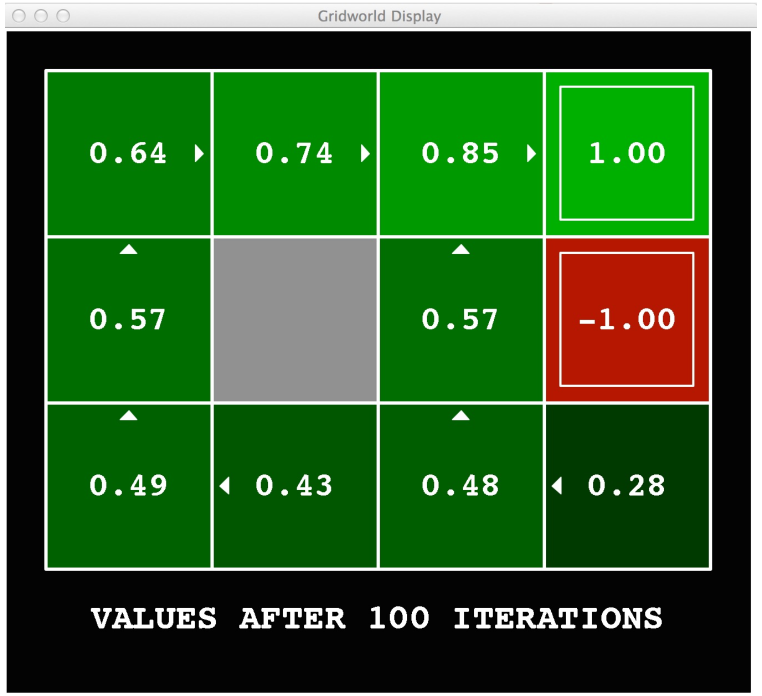
  - Incorporate the new estimate into a running average:

    $$V^{\pi}(s) \leftarrow (1-\alpha) \cdot V^{\pi}(s) + \alpha \cdot sample$$

    or $$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha \cdot [sample - V^{\pi}(s)]$$

    ($[sample - V^{\pi}(s)]$ is the "*TD error*"; α is the *learning rate*)

  - Property: TD-learning will converge to true values of the *given* policy



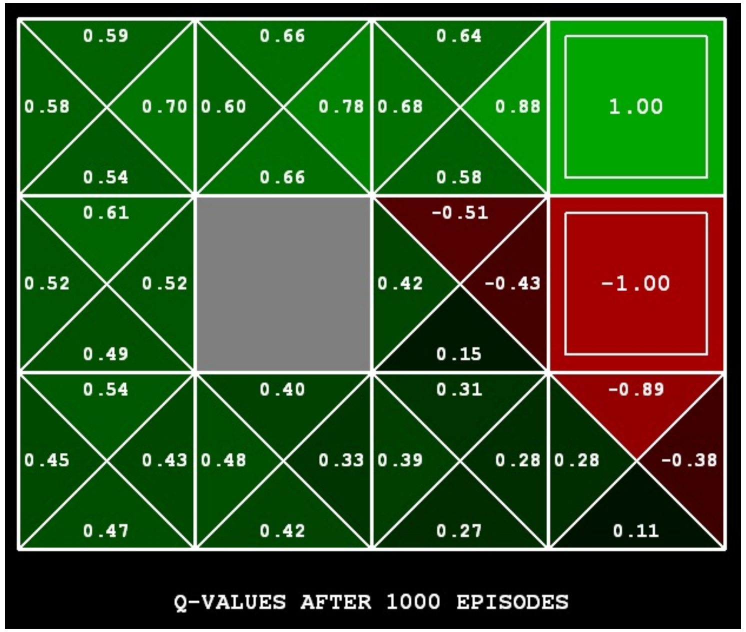
Model-free RL: Q-Learning

- Learn Q(s,a) values as you go
  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

    $$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

  - Incorporate the new estimate into a running average:

    $$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha \cdot [sample]$$



  - Property: Q-learning will converge to the *optimal* policy, under *any* exploration policy that allows visiting the full state space for infinitely many times (*off-policy*).
    - Requirements: $\sum_t \alpha(t) = \infty$, $\sum_t \alpha^2(t) < \infty$

Linear Value Functions

- We can express V and Q (approximately) as weighted linear functions of feature values:
  - $V_{\mathbf{w}}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
  - $Q_{\mathbf{w}}(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$

- Approximate Q-learning: update the *weights* to reduce the error at s,a:
  - Receive a sample (s,a,s',r)
  - $w_i \leftarrow w_i + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)]\, \partial Q_{\mathbf{w}}(s,a) / \partial w_i$

    $= w_i + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)]\, f_i(s,a)$

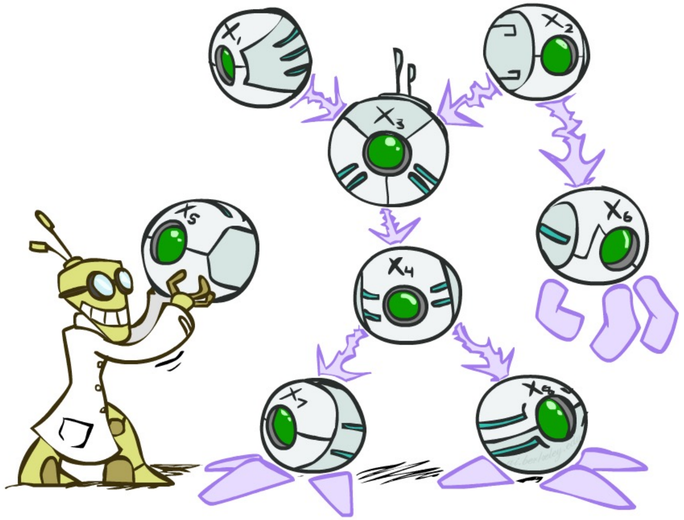
Bayes Net Representation

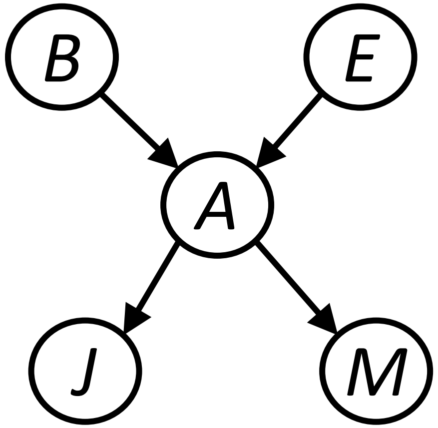
- A *directed, acyclic graph*, with node = random variable
- Each node stores a *conditional probability table (CPT)*
  - A collection of conditional distributions over X, one for each combination of parents' values

$$P(X|a_1 \ldots a_n)$$

- Bayes nets implicitly encode joint distributions
  - As a product of local conditional distributions

$$P(x_1, x_2, \ldots x_n) = \prod_{i=1}^{n} P(x_i|parents(X_i))$$





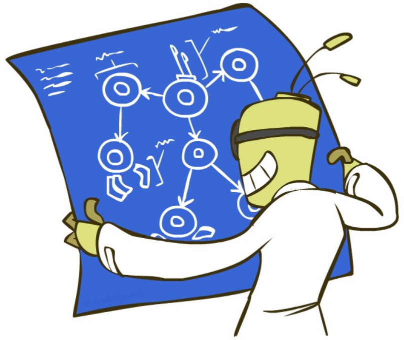
Independences: D-Separation Algorithm

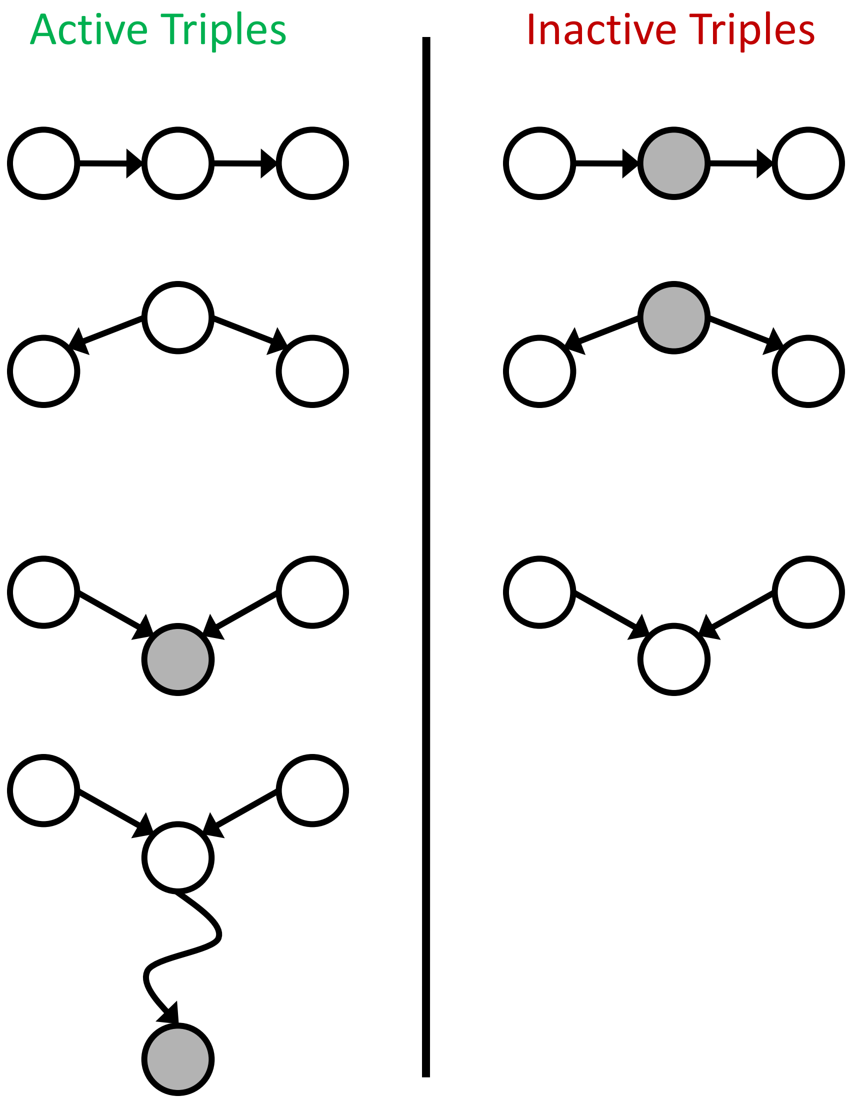
- Query: $X_i \perp\!\!\!\perp X_j | \{X_{k_1}, ..., X_{k_n}\}$
- Check all *undirected* paths between X_i and X_j
  - If one or more path active, independence broken

    $$X_i \not\perp\!\!\!\perp X_j | \{X_{k_1}, ..., X_{k_n}\}$$

  - Otherwise (i.e. if all paths are inactive), independence guaranteed

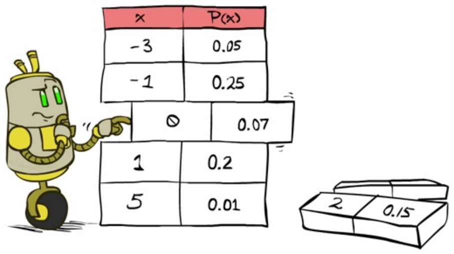
    $$X_i \perp\!\!\!\perp X_j | \{X_{k_1}, ..., X_{k_n}\}$$





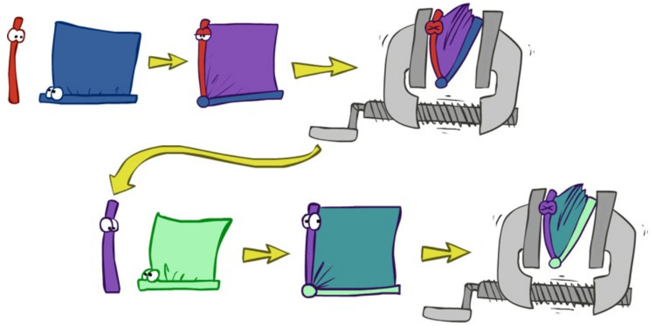
Exact Inference: Variable Elimination

- Query: $P(Q|E_1 = e_1, \ldots E_k = e_k)$



- Start with initial factors:
  - Local CPTs (but instantiated by evidence)

- While there are still hidden variables (not Q or evidence):
  - Pick a hidden variable H
  - Join all factors mentioning H
  - Eliminate (sum out) H

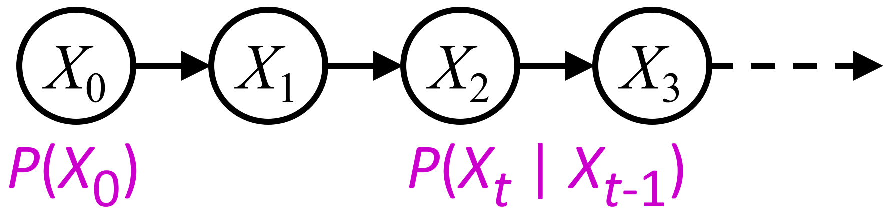


- Join all remaining factors and normalize

$$\times \frac{1}{Z}$$

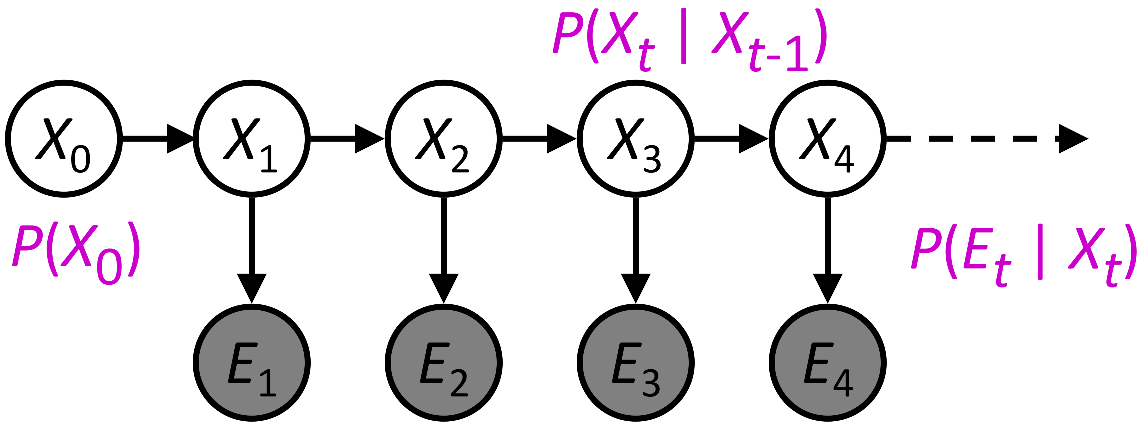
Bayes Nets -> Markov Models

- Markov chains



- The *transition model* $P(X_t \mid X_{t-1})$ specifies how the state evolves over time
- *Stationarity* assumption: transition probabilities are the same at all times
- *Markov* assumption: “future is independent of the past given the present”

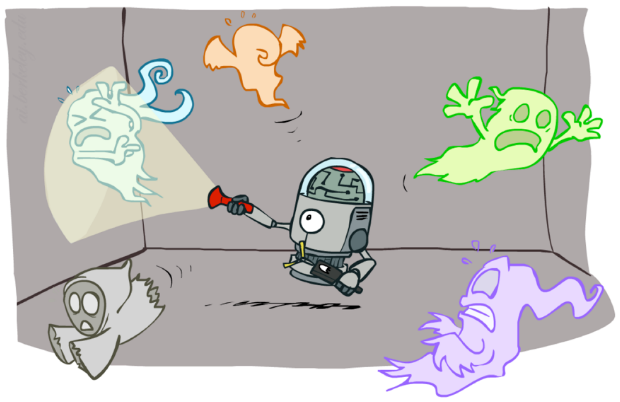
- Hidden Markov models (HMMs)

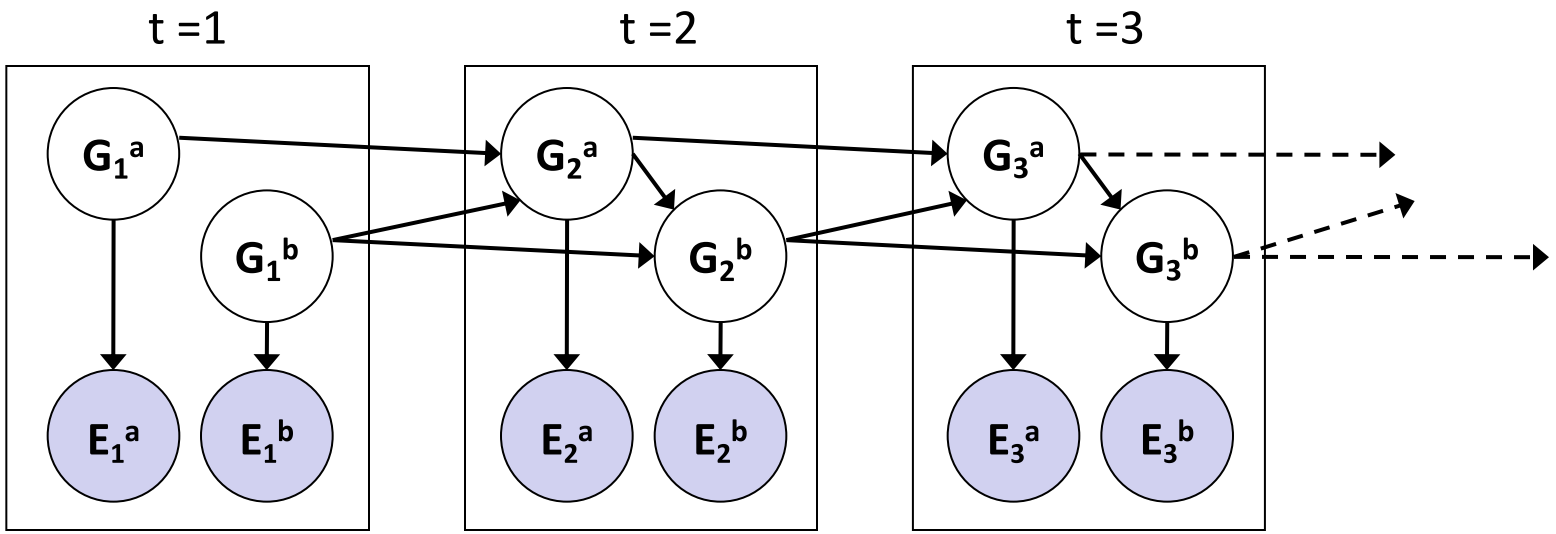


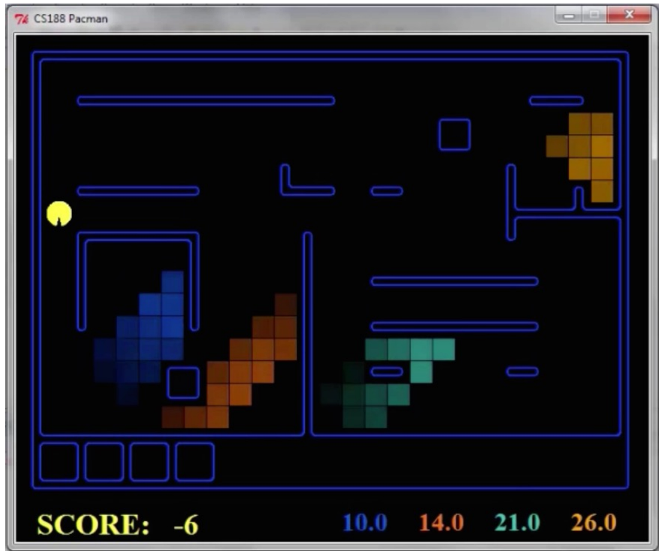
- *Partial observability*: there is an underlying Markov chain over states X, you observe an evidence E emitted by X at each time step
- The *sensor model* $P(E_t \mid X_t)$ specifies the likelihood of observing the evidence from the underlying state

Markov Models -> Dynamic Bayes Nets

- DBNs track *multiple* variables over time, using *multiple* sources of evidence
- Idea:
  - Repeat a fixed Bayes net structure at each time
  - Variables at time *t* can condition on those at *t-1*







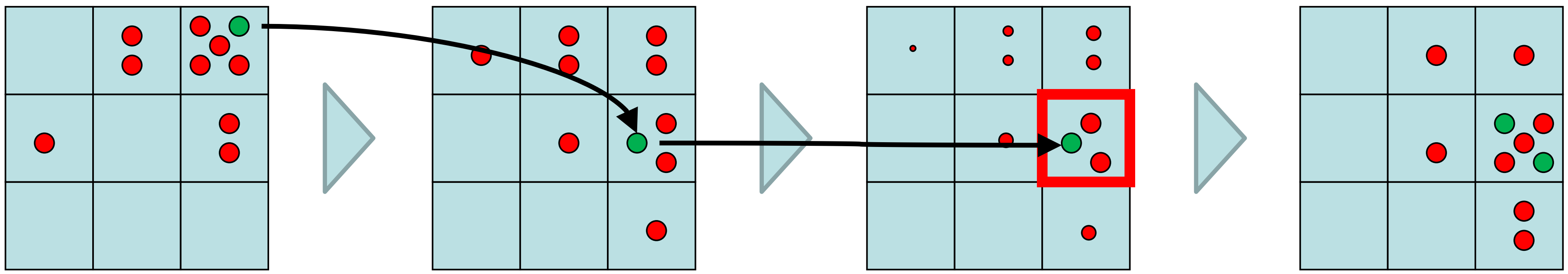
Filtering / Belief Tracking

- **Exact**: online filtering / forward algorithm

$$P(X_{t+1}|e_{1:t+1}) = \underbrace{\alpha}_{\text{Normalize}}\ \underbrace{P(e_{t+1}|X_{t+1})}_{\text{Update}}\ \underbrace{\textstyle\sum_{x_t} P(X_{t+1}|x_t)\, P(x_t|e_{1:t})}_{\text{Predict}}$$

- **Approximate**: particle filtering

Predict Update Resample



$x' = \text{sample}(P(X'|x))$ $w(x) = P(e|x)$

Good Luck!