Final Review

Shuai Li

John Hopcroft Center, Shanghai Jiao Tong University

https://shuaili8.github.io

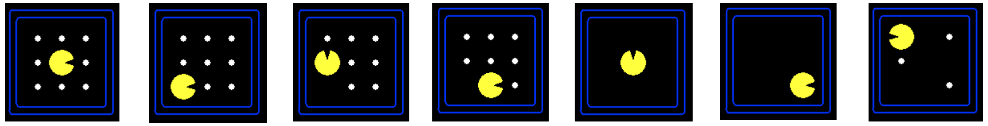
https://shuaili8.github.io/Teaching/CS3317/index.html

Part of slide credits: CMU AI & http://ai.berkeley.edu

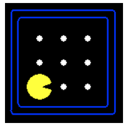
Search Problems

Search Problems

- A search problem consists of:
  - A state space

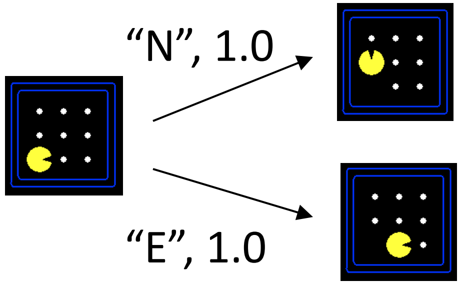


  - For each state, a set Actions(s) of successors/actions



{N, E}

  - A successor function
    - A transition model T(s,a)
    - A step cost(reward) function c(s,a,s')



  - A start state and a goal test

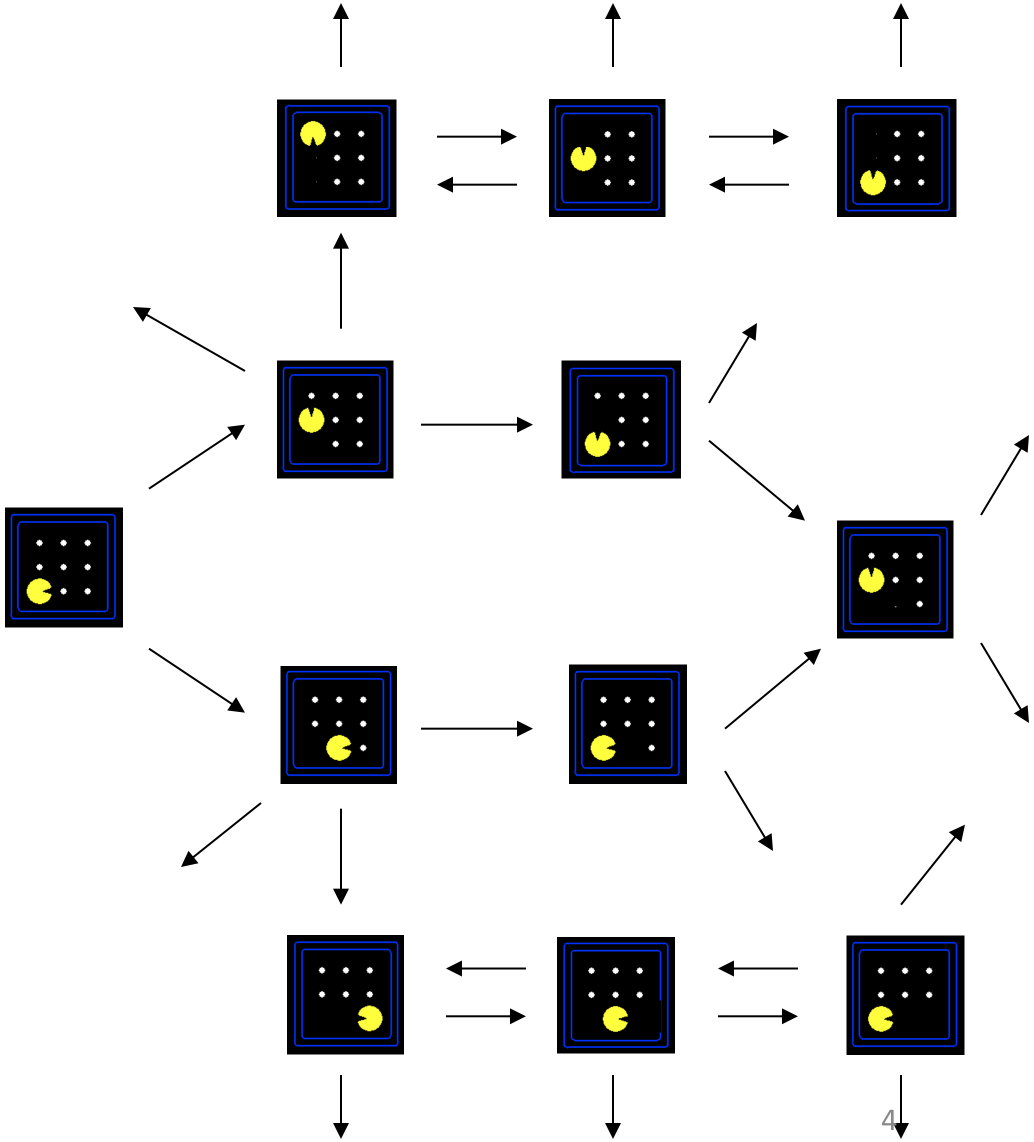
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

State Space Graphs

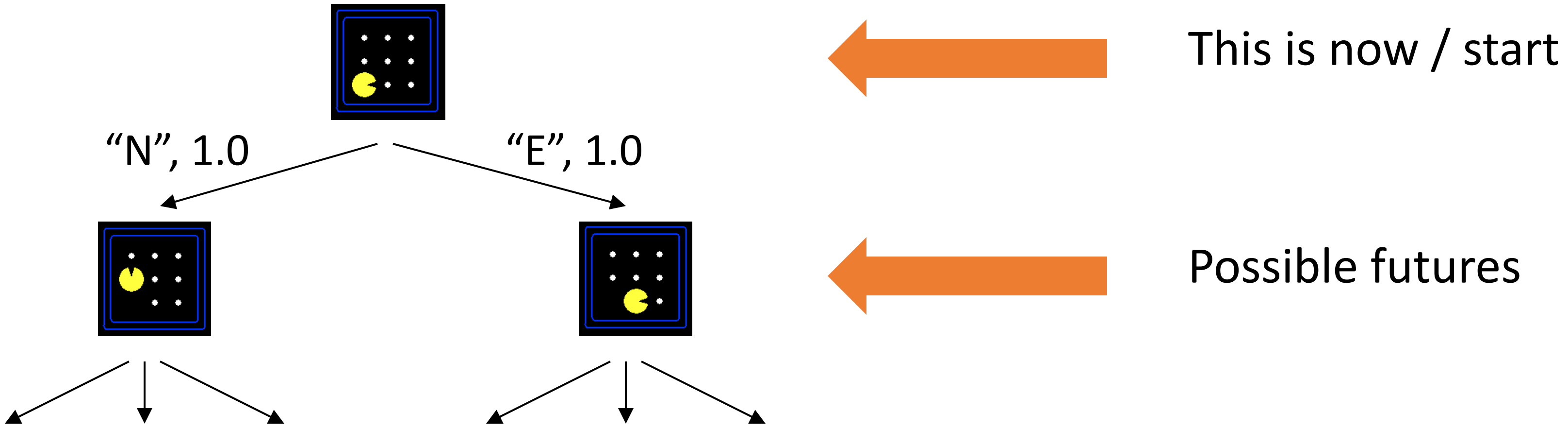
- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

- In a state space graph, each state occurs only once!

- We can rarely build this full graph in memory (it's too big), but it's a useful idea

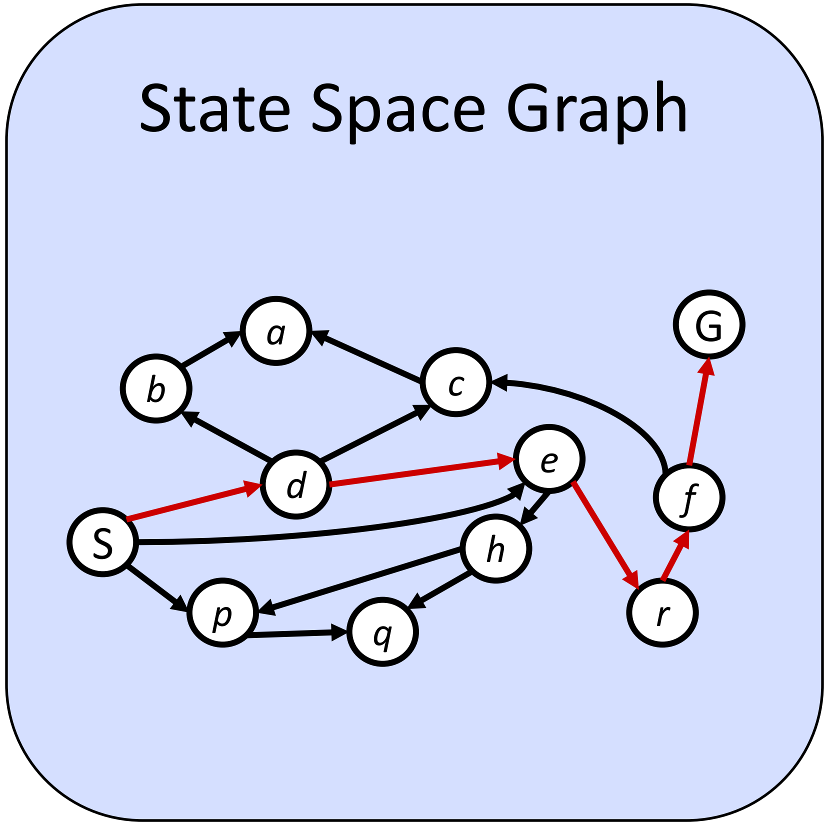


Search Trees



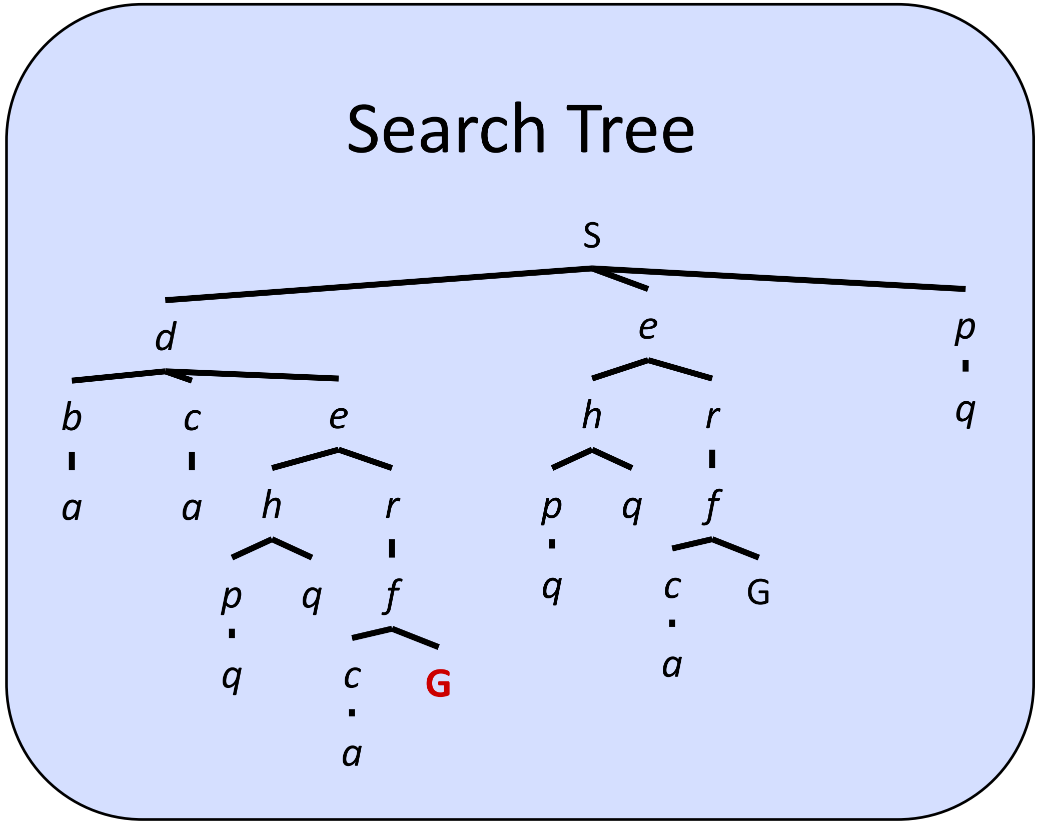
- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

State Space Graphs vs. Search Trees

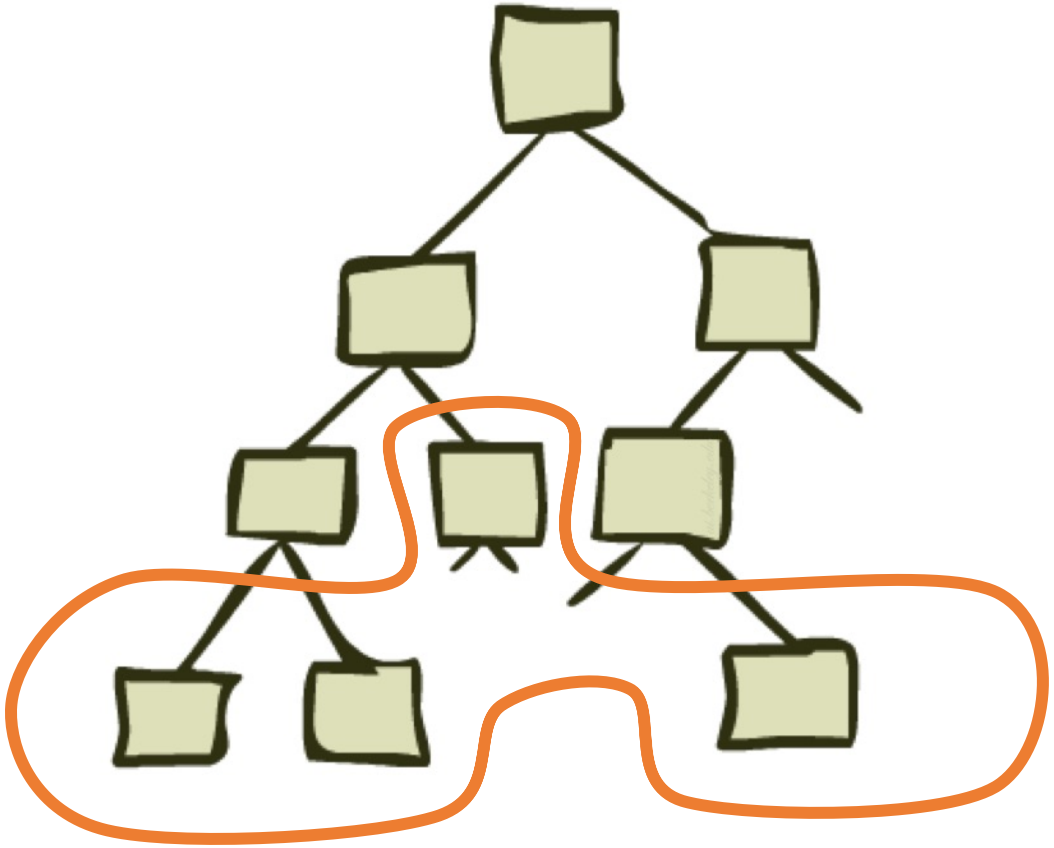


Each NODE in in the search tree is an entire PATH in the state space graph.

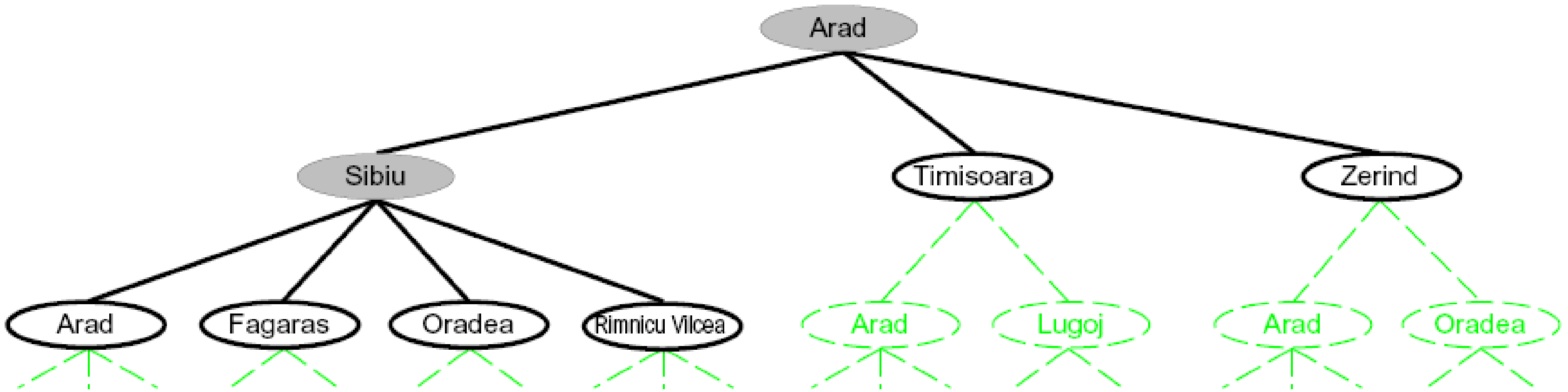
We construct both on demand – and we construct as little as possible.



Tree Search



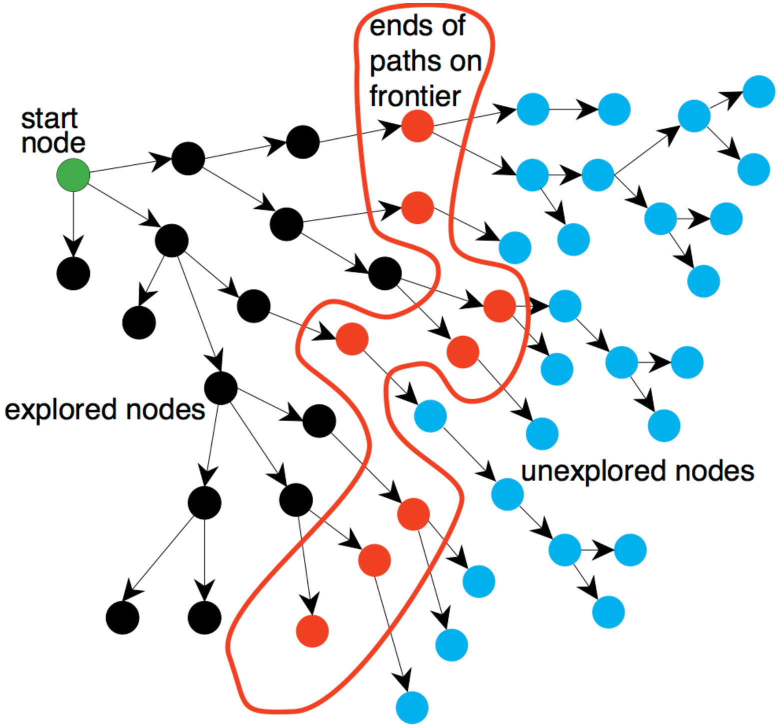
Searching with a Search Tree



- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe of partial plans under consideration
  - Try to expand as few tree nodes as possible

General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```



- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy

- Main question: which fringe nodes to explore?

General Tree Search 2

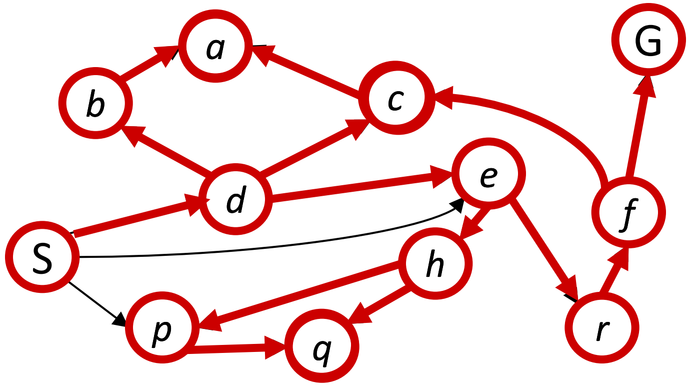
function TREE_SEARCH(problem) returns a solution, or failure

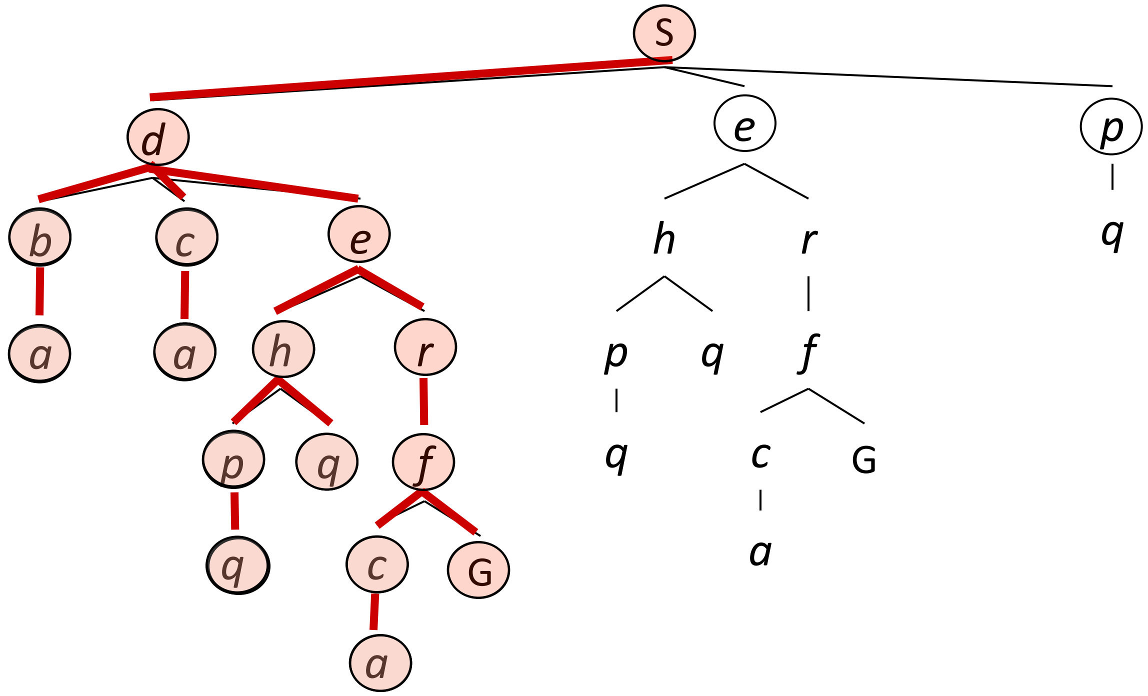
    initialize the frontier as **a specific work list (stack, queue, priority queue)**
    add initial state of problem to frontier
    loop do
        if the frontier is empty then
            return failure
        choose a node and remove it from the frontier
        if the node contains a goal state then
            return the corresponding solution
        for each resulting child from node
            add child to the frontier

Depth-First (Tree) Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack

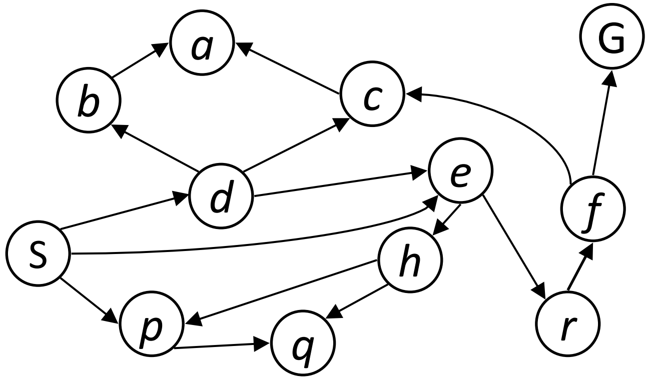




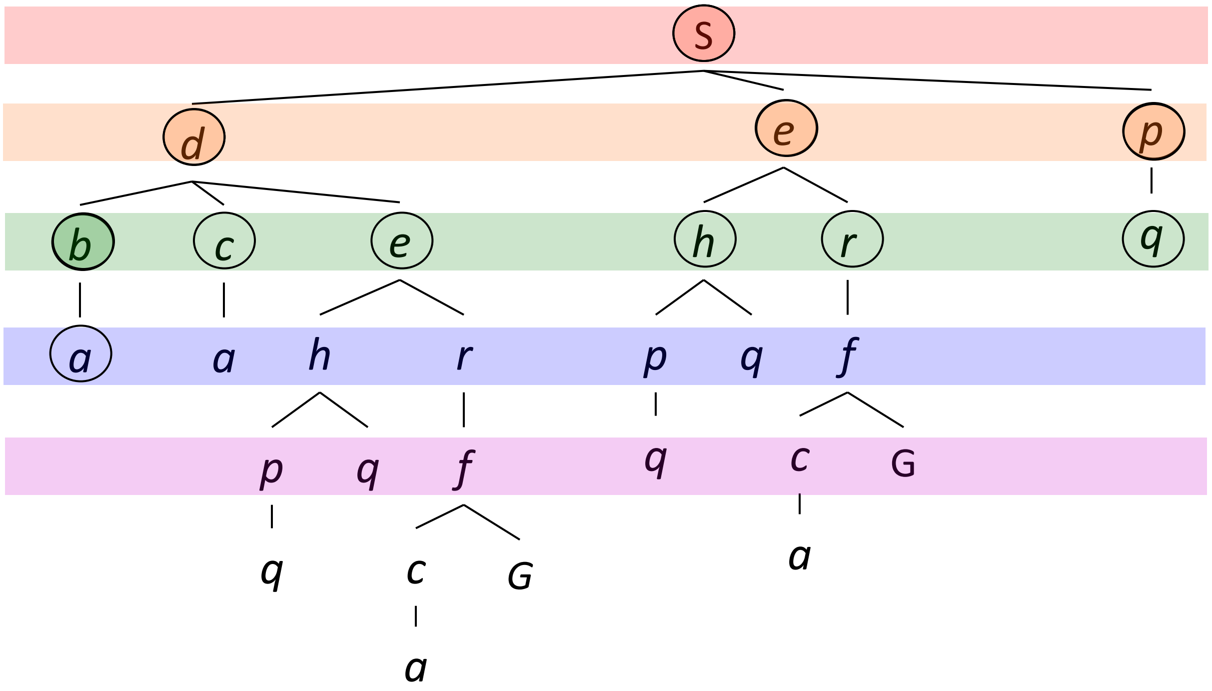
Breadth-First (Tree) Search

Strategy: expand a shallowest node first

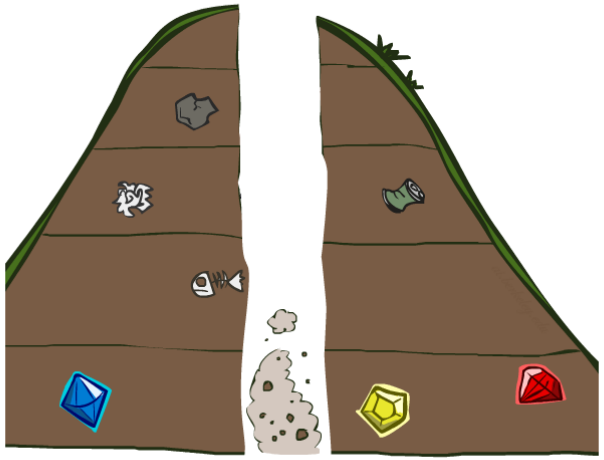
Implementation: Fringe is a FIFO queue



Search Tiers

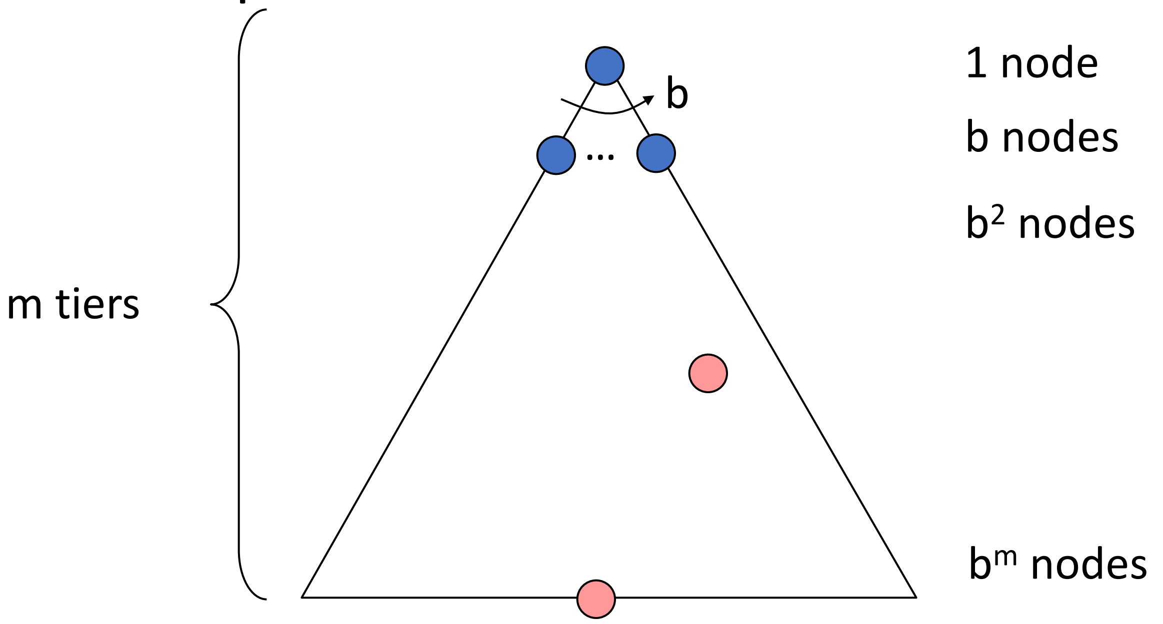


Search Algorithm Properties



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

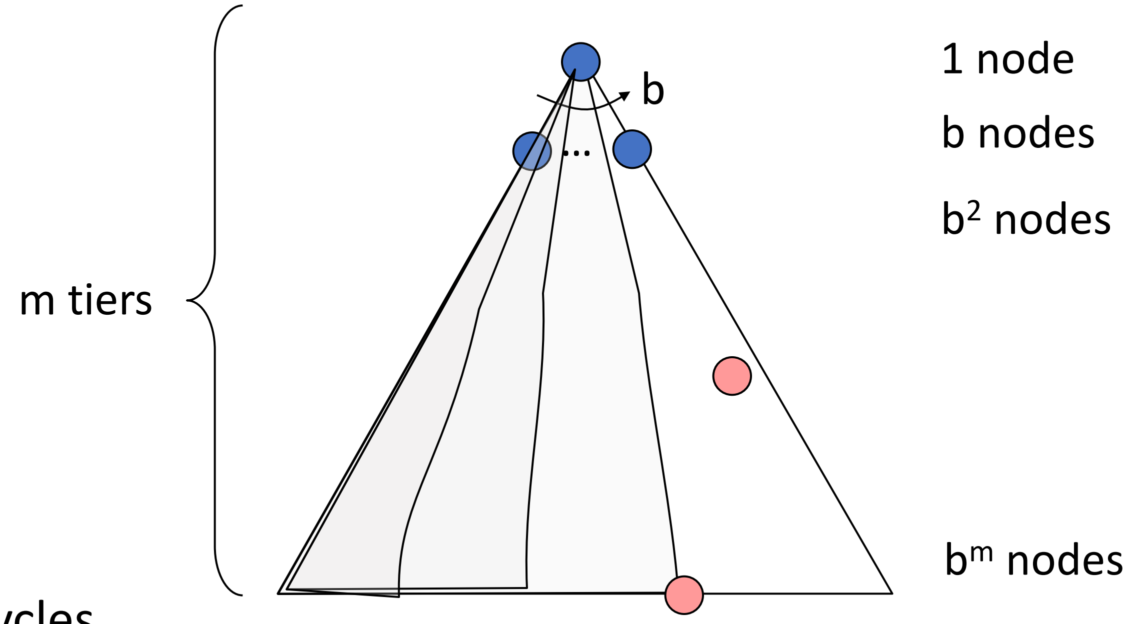
- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths



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

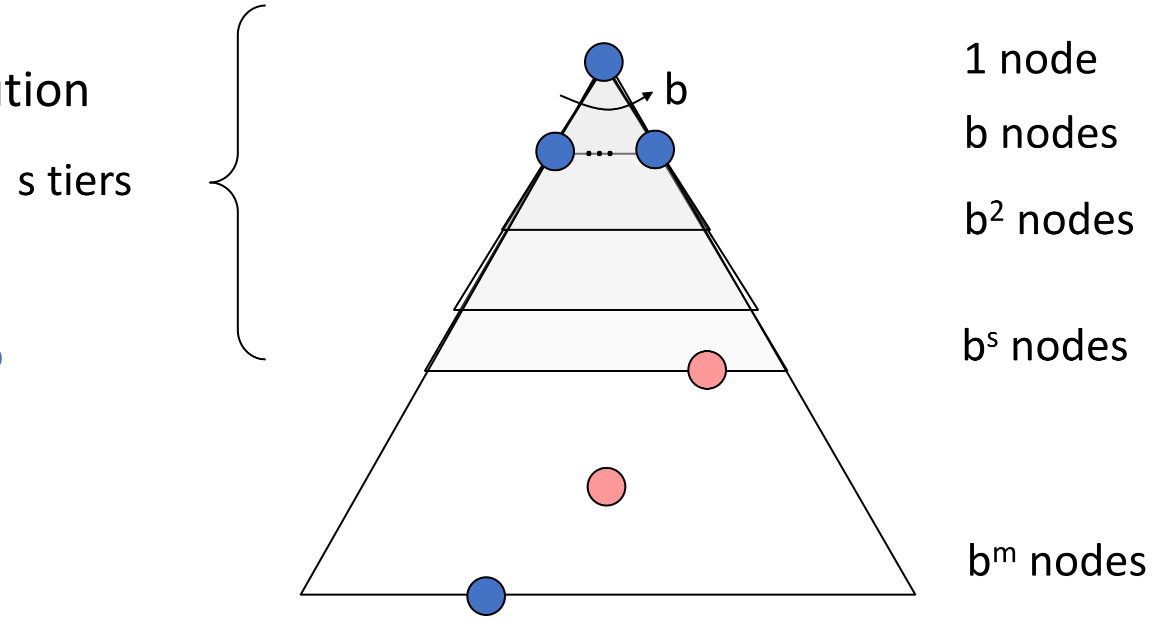
Depth-First Search (DFS) Properties

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so $O(bm)$
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost



Breadth-First Search (BFS) Properties

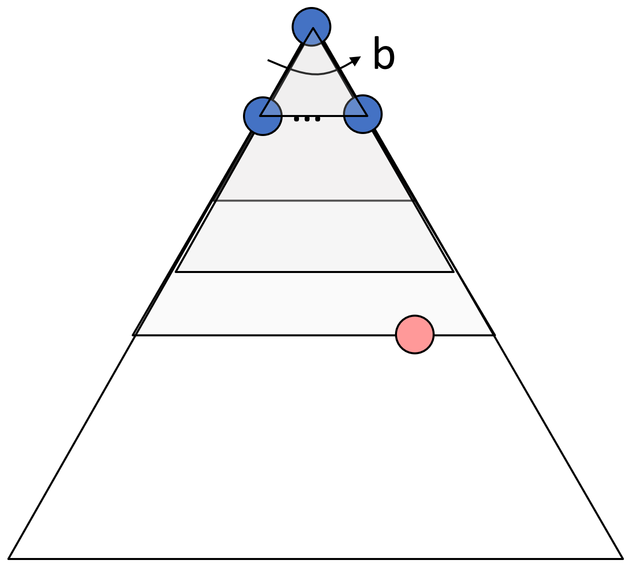
- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



Iterative Deepening

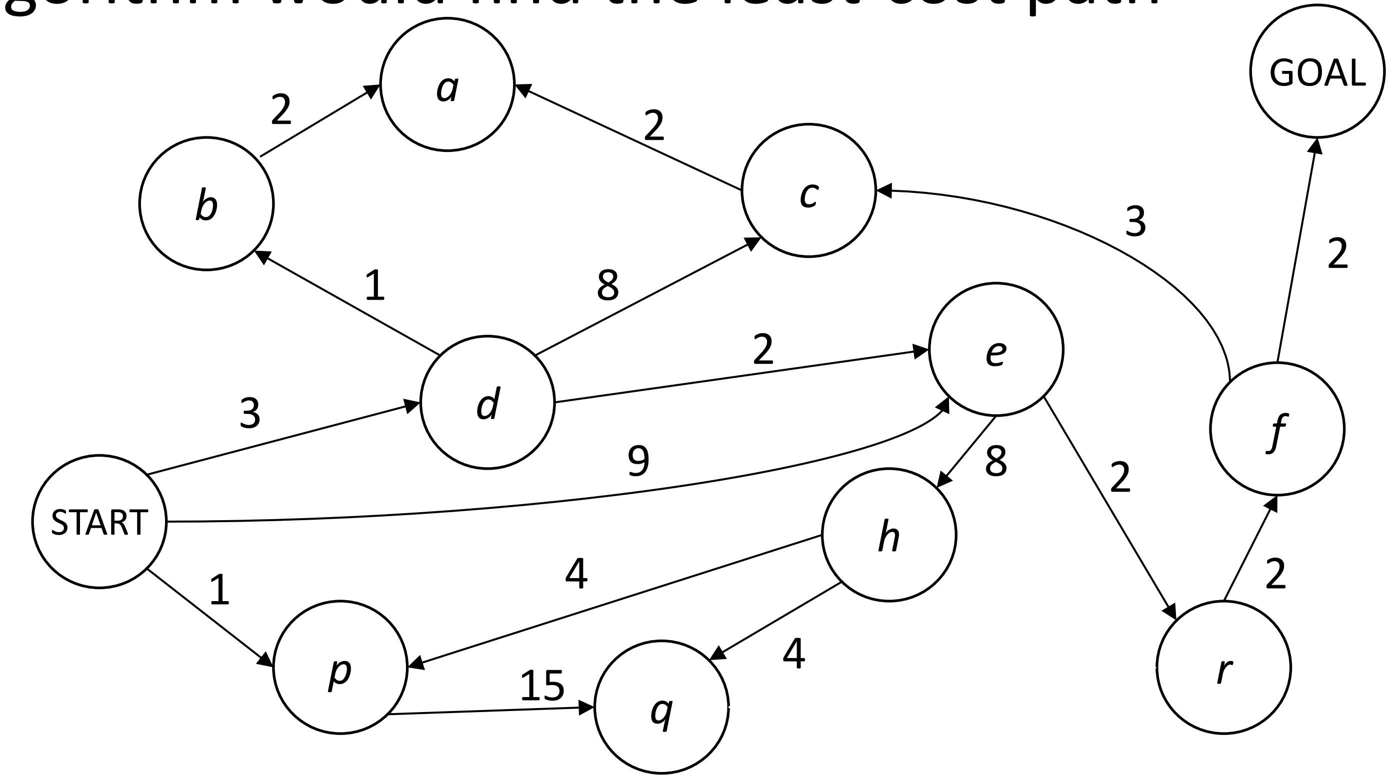
- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..

- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!



Finding a Least-Cost Path

- BFS finds the shortest path in terms of number of actions, but not the least-cost path
- A similar algorithm would find the least-cost path

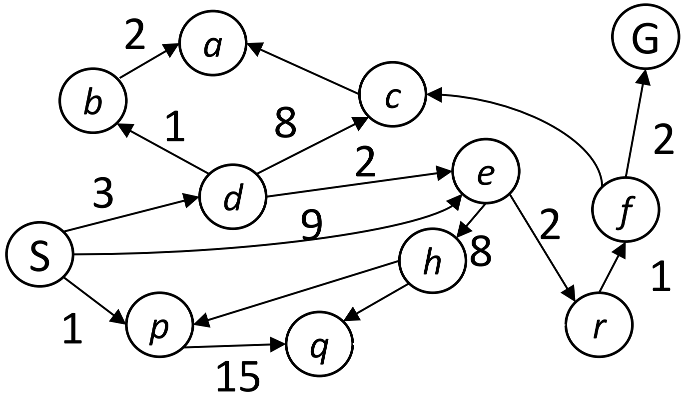


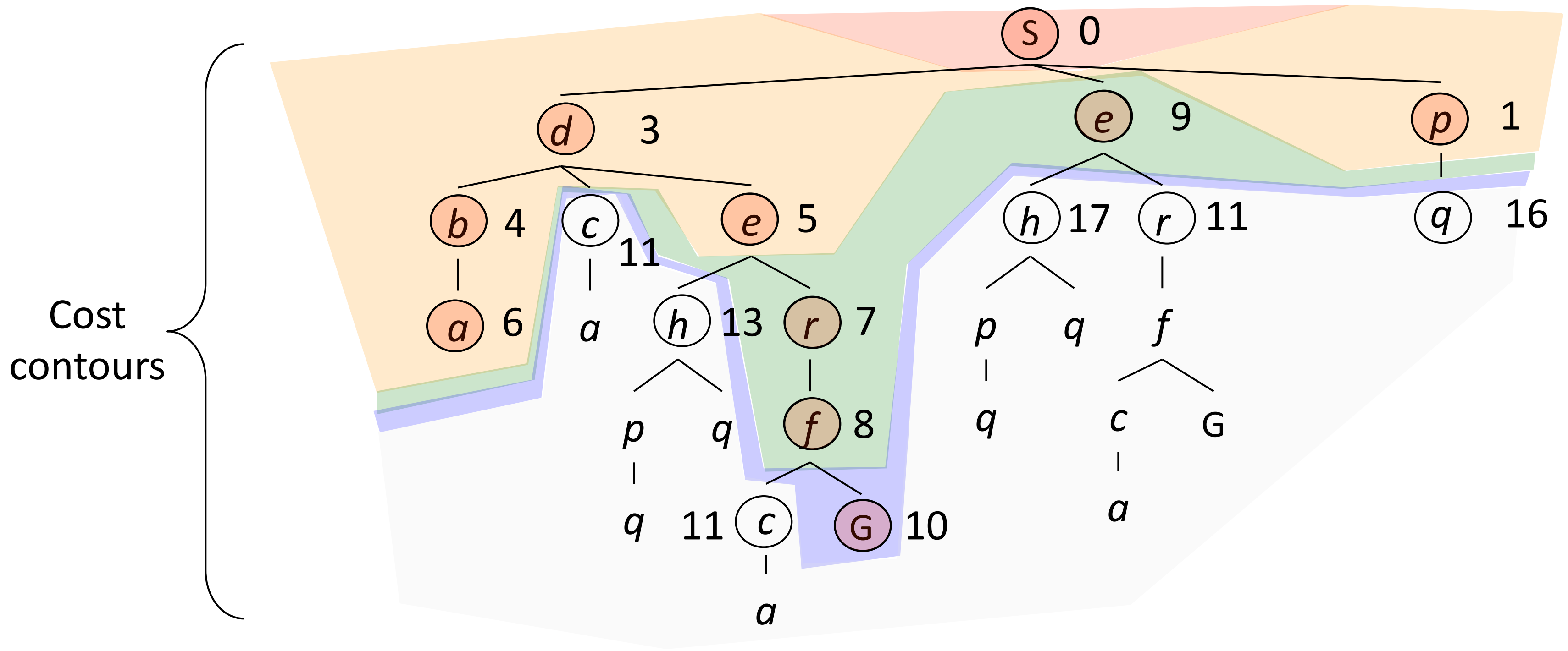
How?

Uniform Cost Search

Strategy: expand a cheapest node first:

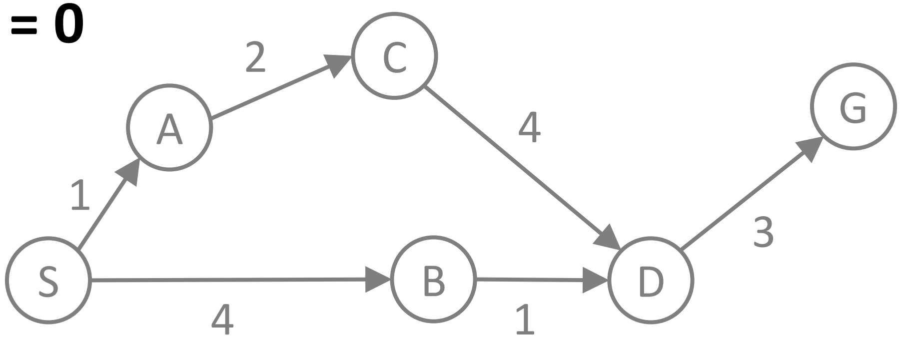
Fringe is a priority queue (priority: cumulative cost)





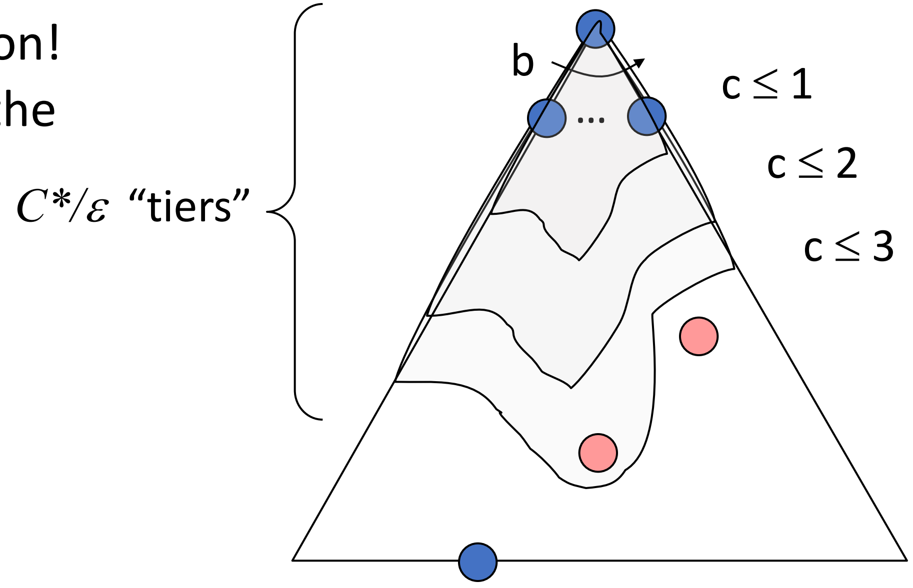
Uniform Cost Search 2

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    initialize the frontier as a **priority queue using node's path_cost as the priority**
    add initial state of problem to frontier **with path_cost = 0**
    loop do
        if the frontier is empty then
            return failure
        choose a node (with minimal path_cost) and remove it from the frontier
        if the node contains a goal state then
            return the corresponding solution
        for each resulting child from node
            add child to the frontier with **path_cost** = **path_cost**(node) + cost(node, child)



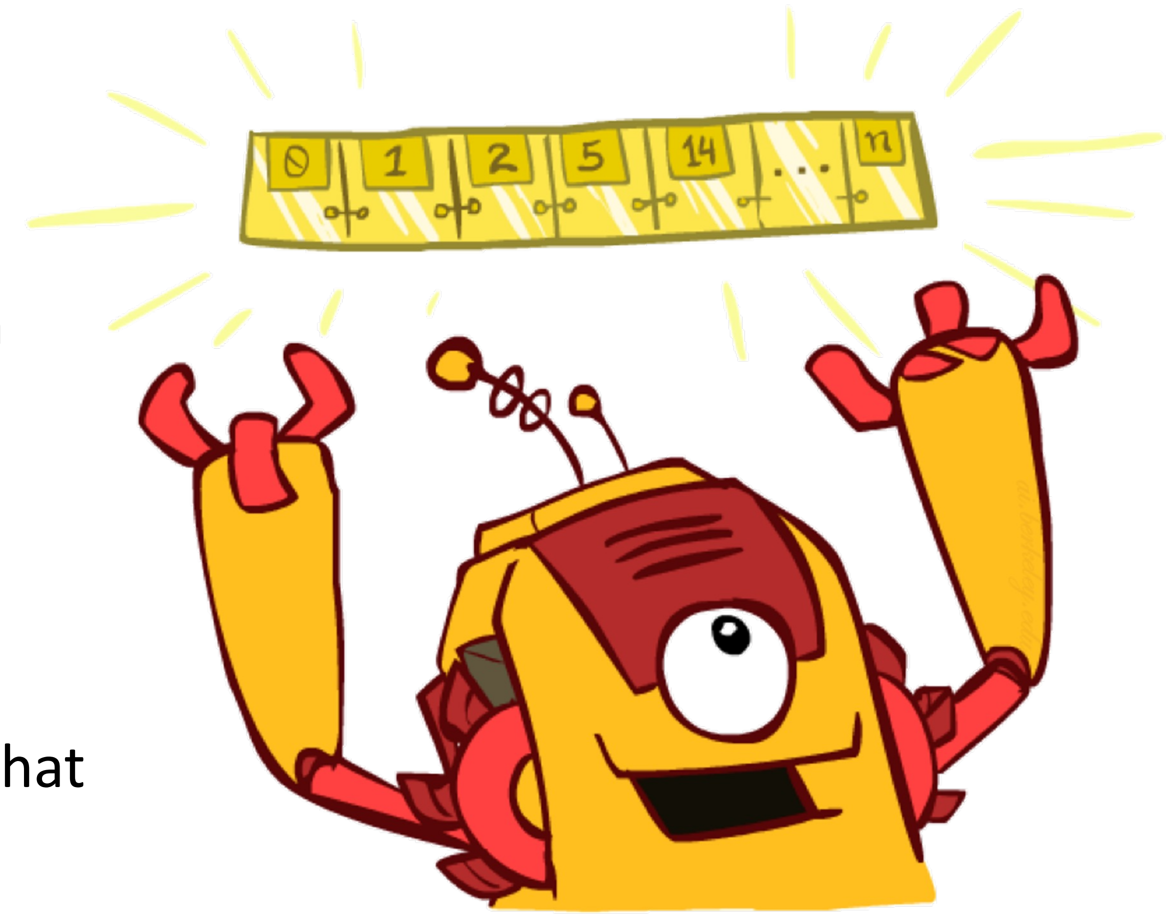
Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time O($b^{C^*/\varepsilon}$) (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so O($b^{C^*/\varepsilon}$)
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (Proof next via A*)



The One Queue

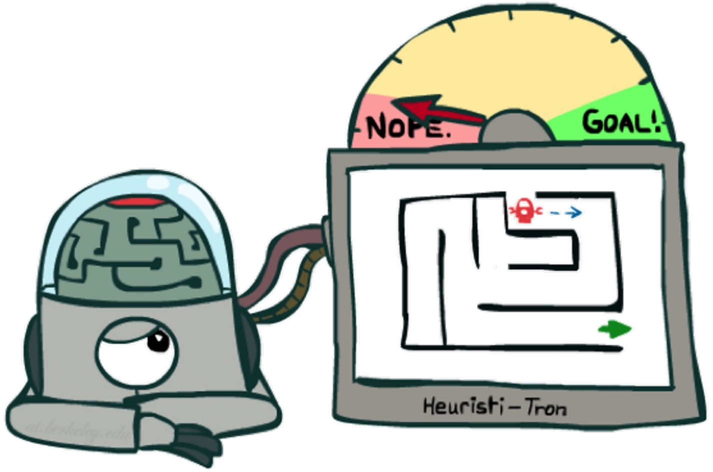
- All these search algorithms are the same except for fringe strategies
  - Conceptually, all fringes are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object

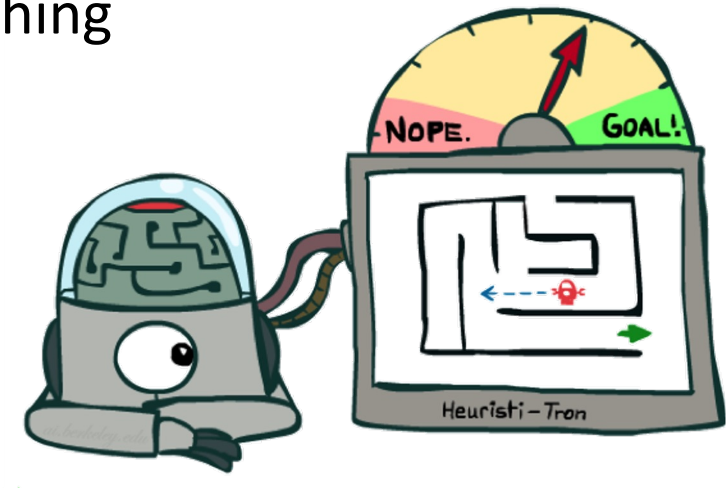


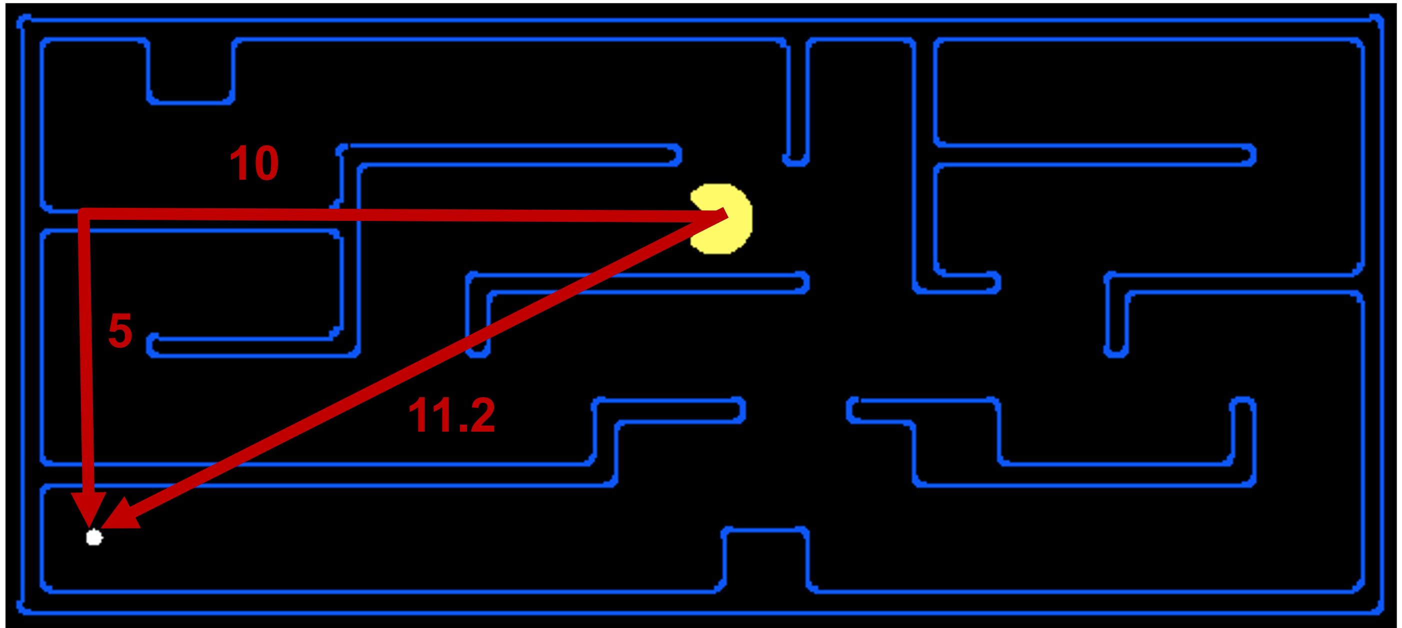
Informed Search

Search Heuristics

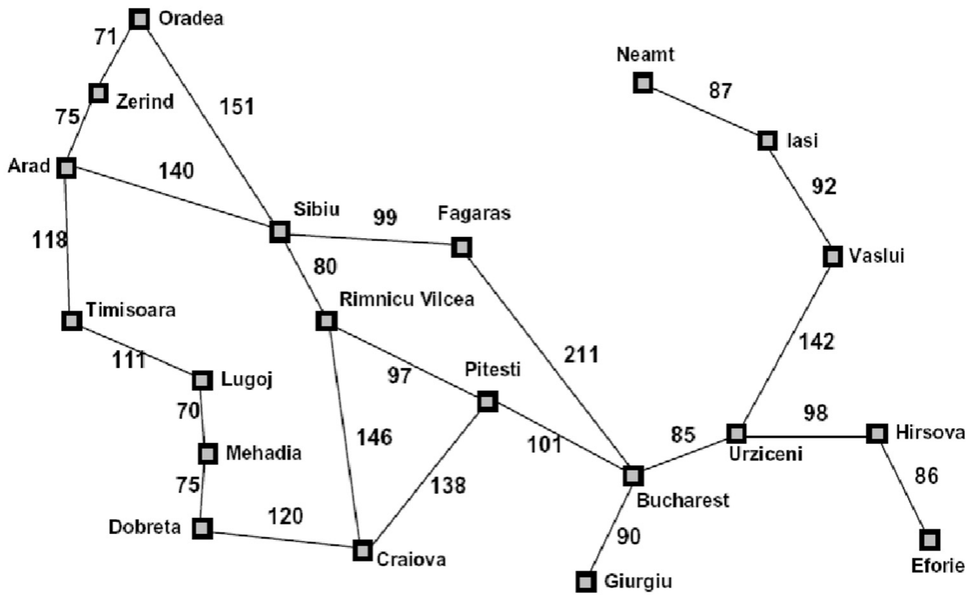
- A heuristic is:
  - A function that estimates how close a state is to a goal
  - Designed for a particular search problem
  - Pathing?
  - Examples: Manhattan distance, Euclidean distance for pathing



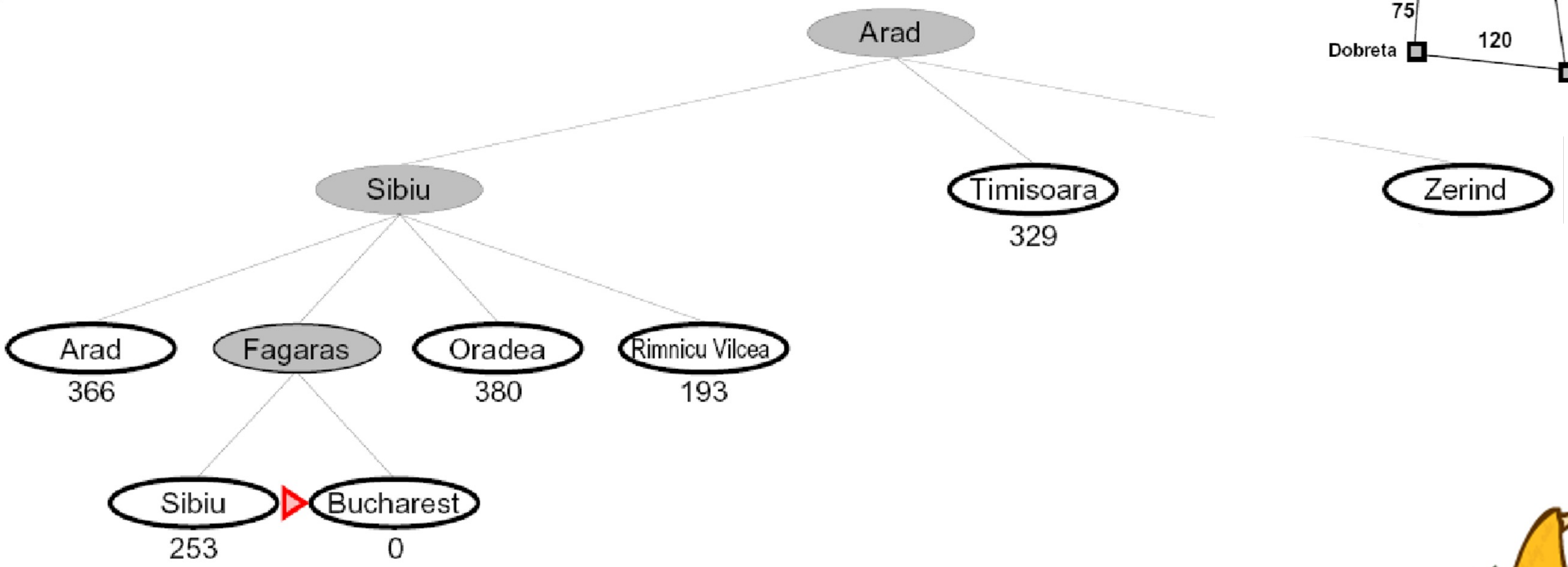


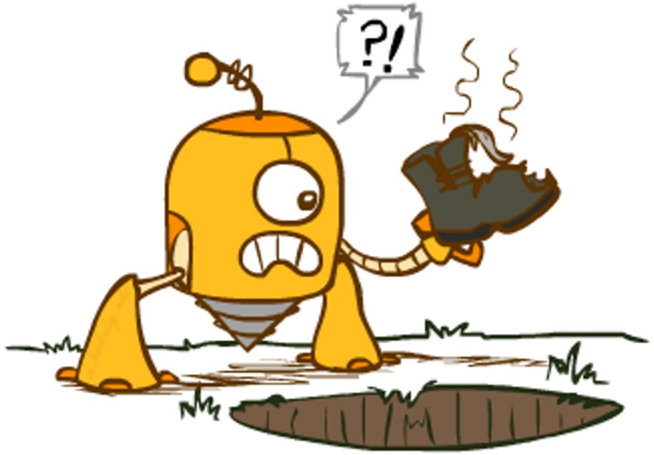


Greedy Search



- Expand the node that seems closest to the goal

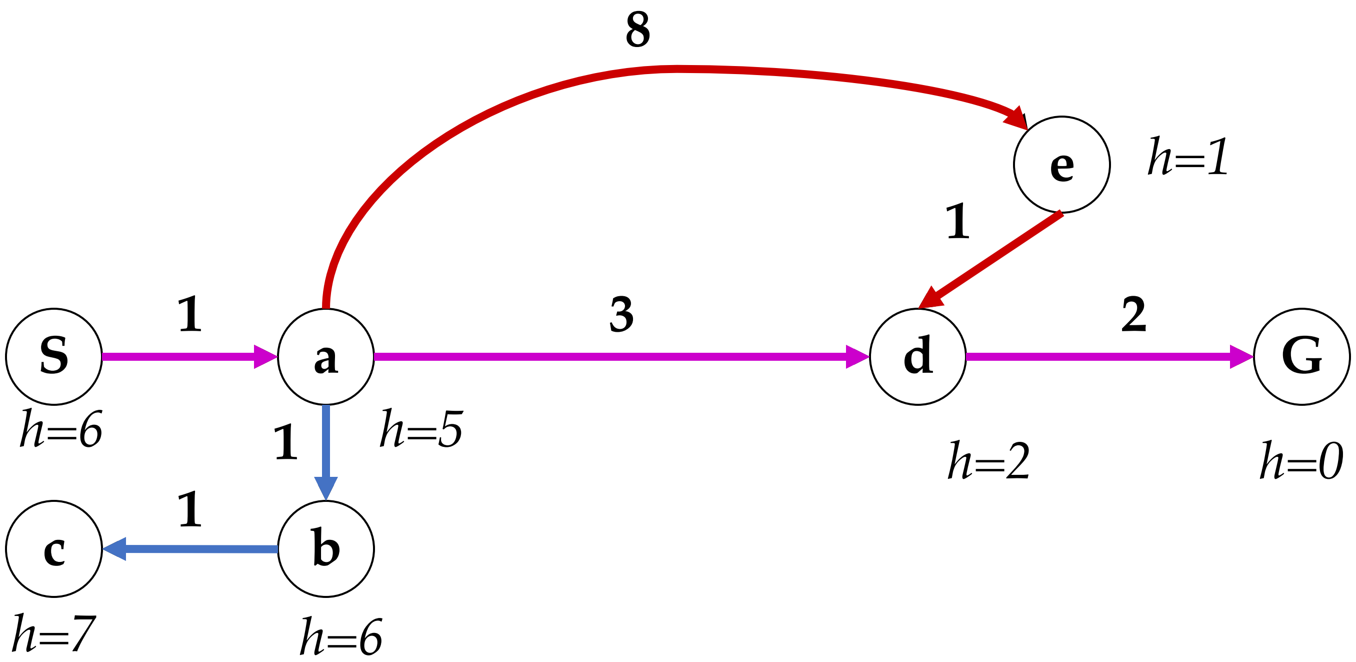


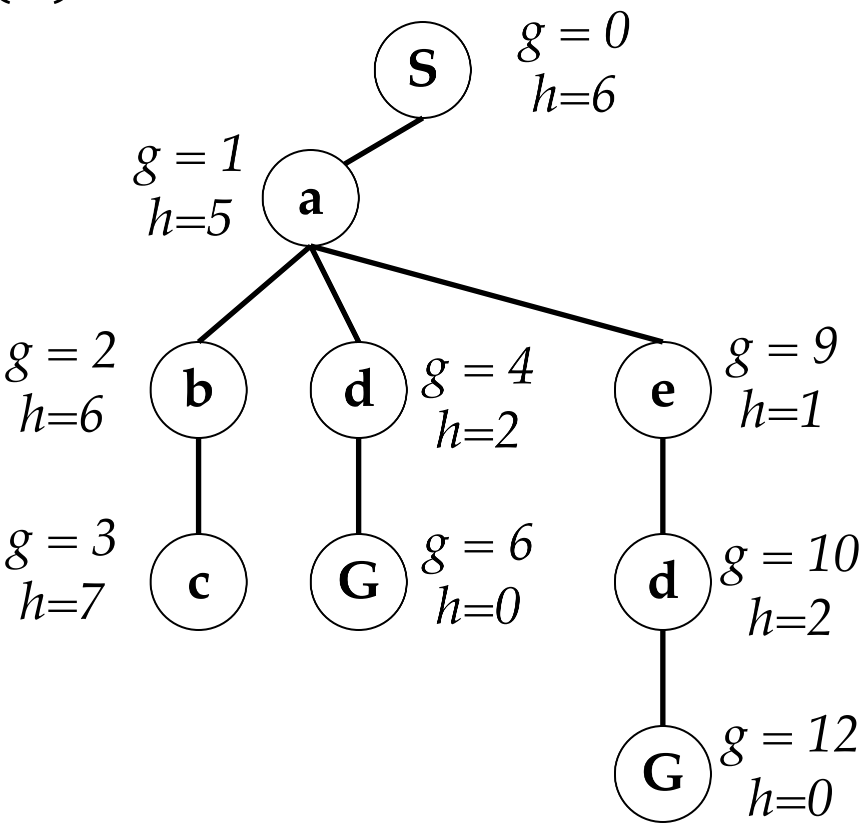


- Is it optimal?
  - No. Resulting path to Bucharest is not the shortest!
  - Why?
  - Heuristics might be wrong

A* Search: Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* $g(n)$
- Greedy orders by goal proximity, or *forward cost* $h(n)$



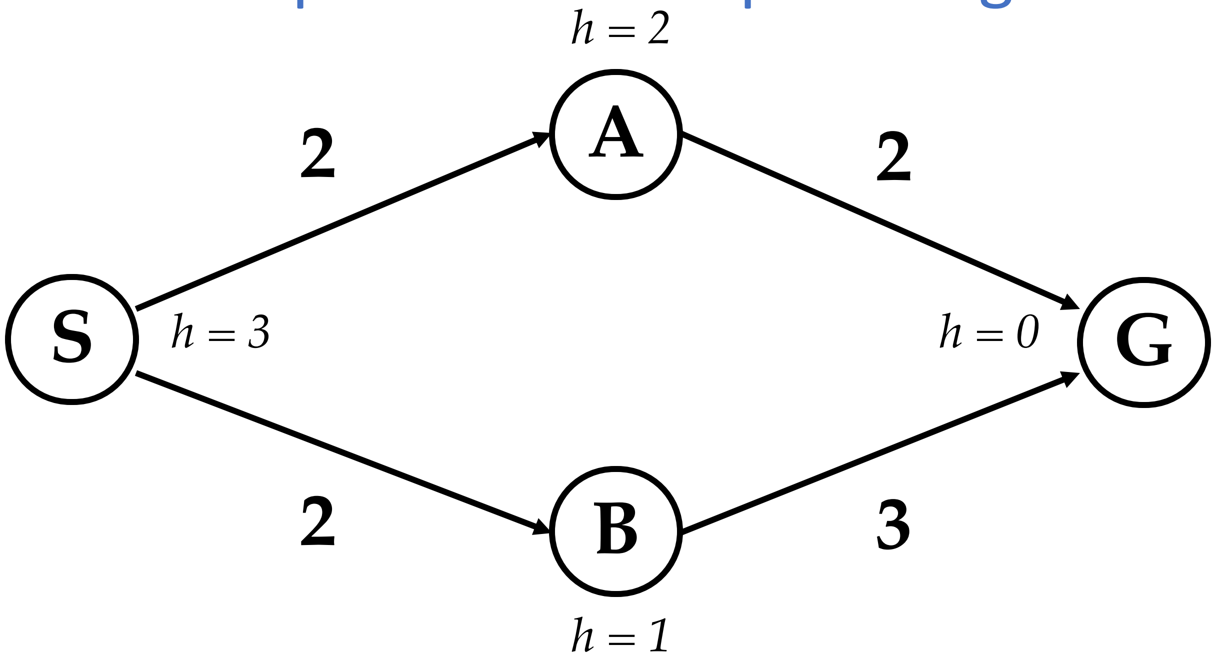


- A* Search orders by the sum: $f(n) = g(n) + h(n)$

Example: Teg Grenager

When should A* terminate?

- Should we stop when we enqueue a goal?



g h +

~~S 0 3 3~~

~~S->A 2 2 4~~

~~S->B 2 1 3~~

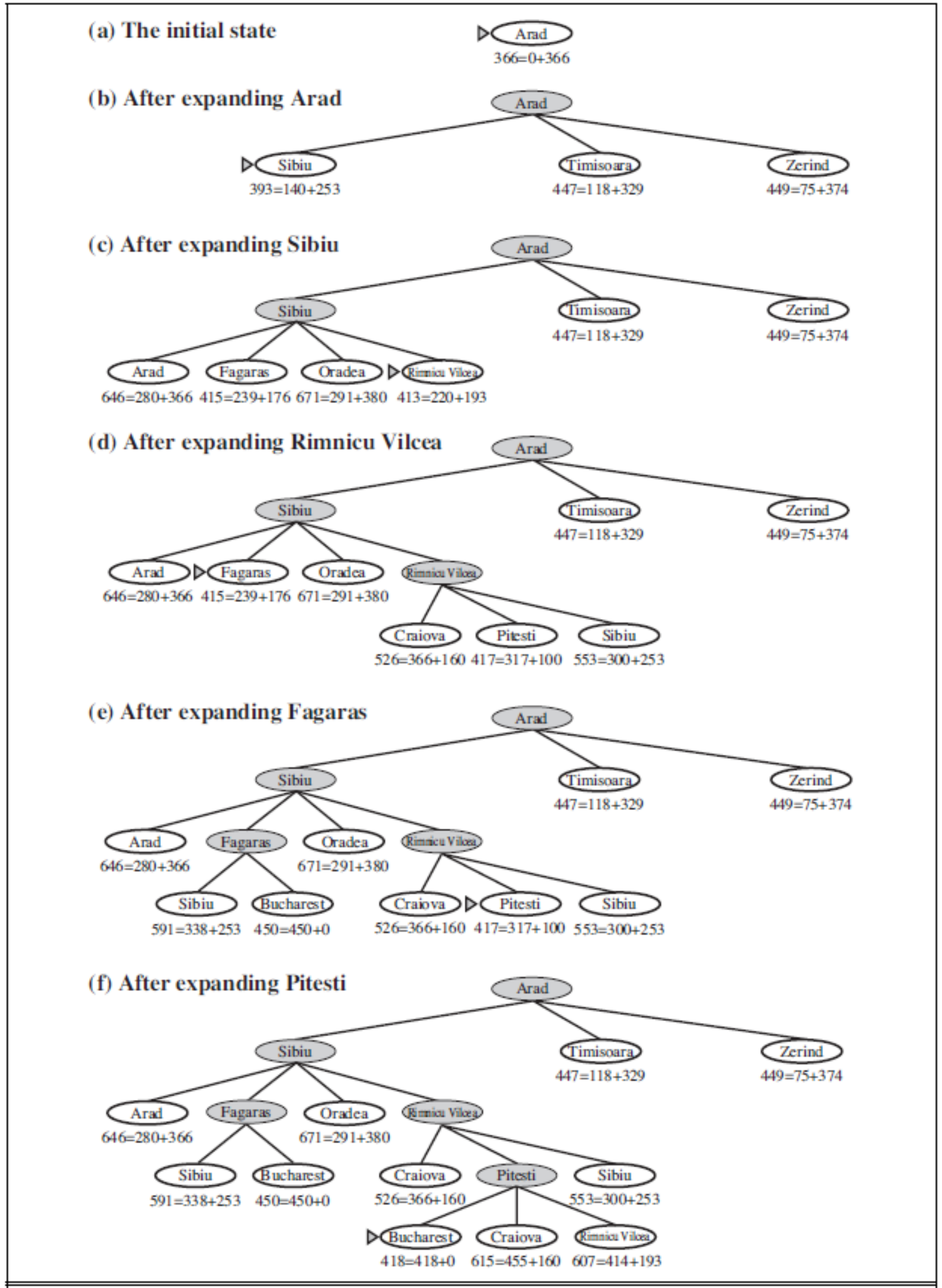
S->B->G 5 0 5

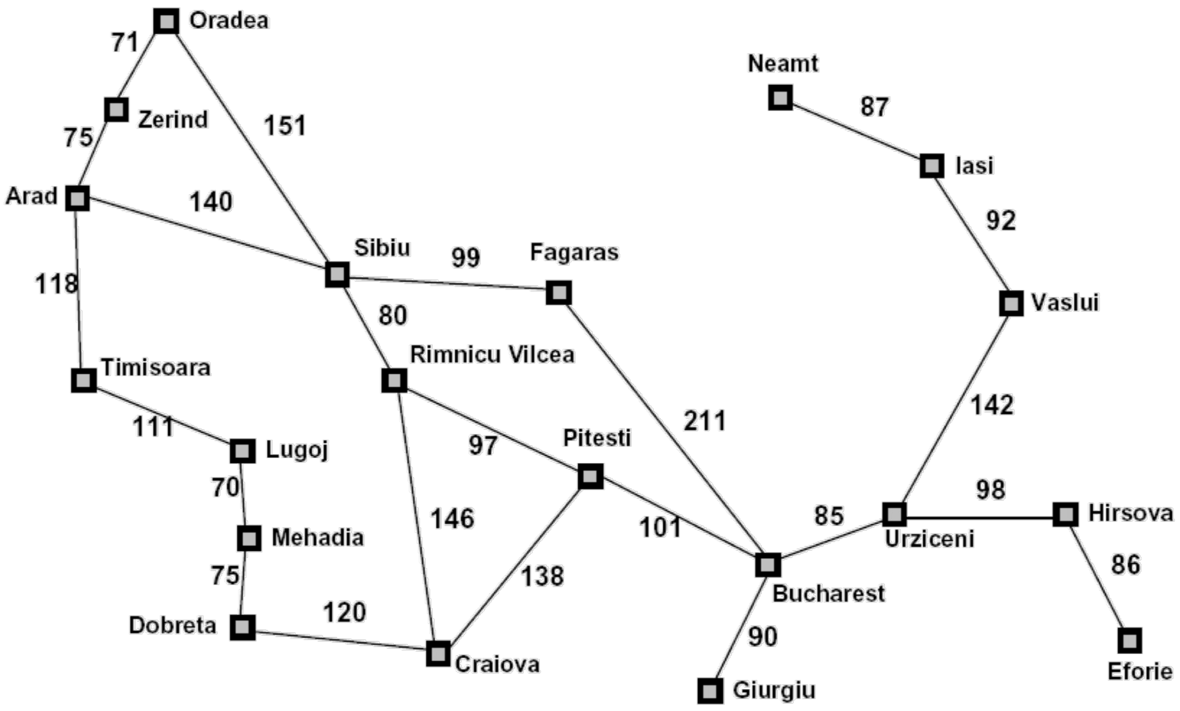
S->A->G 4 0 4

- No: only stop when we dequeue a goal

A* Search

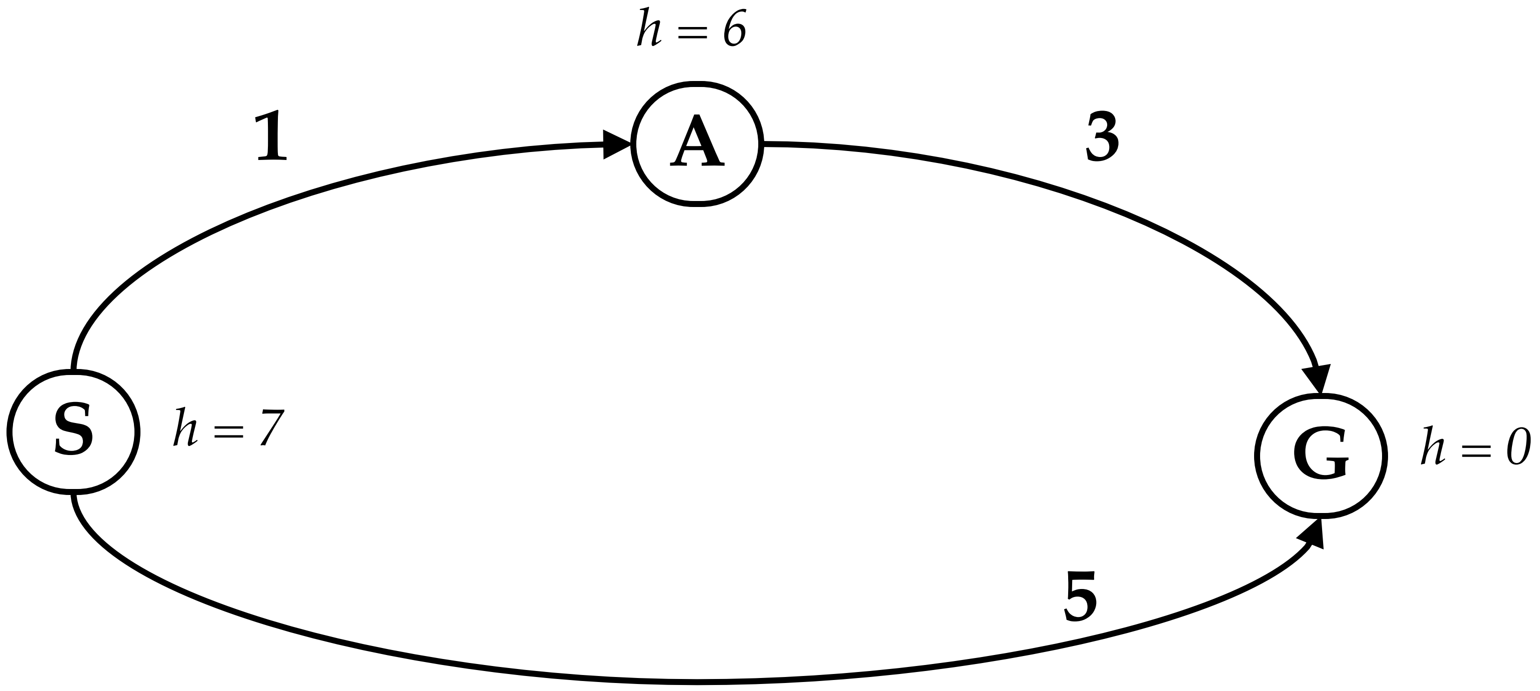
```
function A-STAR-SEARCH(problem) returns a solution, or failure
    initialize the frontier as a priority queue using f(n)=g(n)+h(n) as the priority
    add initial state of problem to frontier with priority f(S)=0+h(S)
    loop do
        if the frontier is empty then
            return failure
        choose a node and remove it from the frontier
        if the node contains a goal state then
            return the corresponding solution
        for each resulting child from node
            add child to the frontier with f(n)=g(n)+h(n)
```





Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Is A* Optimal?



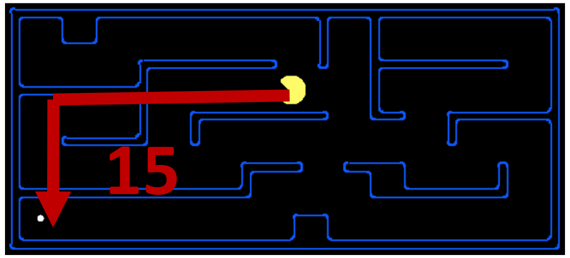
	g	h	+
~~S~~	~~0~~	~~7~~	~~7~~
S->A	1	6	7
S->G	5	0	5

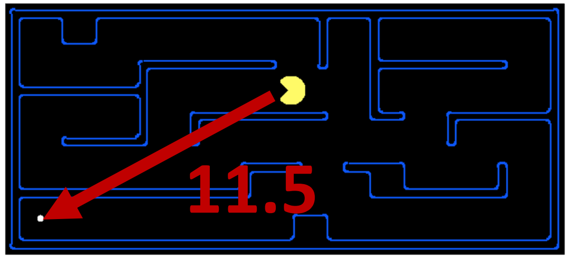
- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

Admissible Heuristics

- A heuristic h is *admissible* (optimistic) if
$$0 \leq h(n) \leq h^*(n)$$
where $h^*(n)$ is the true cost to a nearest goal

- Examples:





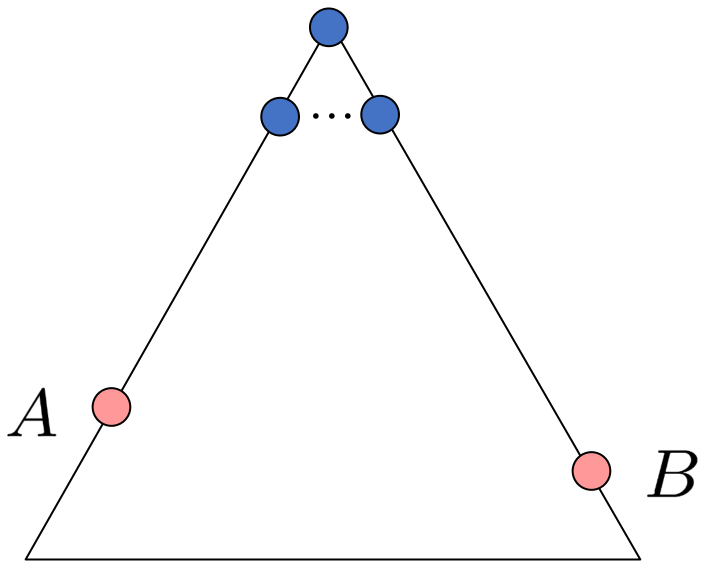
0.0

- Coming up with admissible heuristics is most of what's involved in using A* in practice

Optimality of A* Tree Search

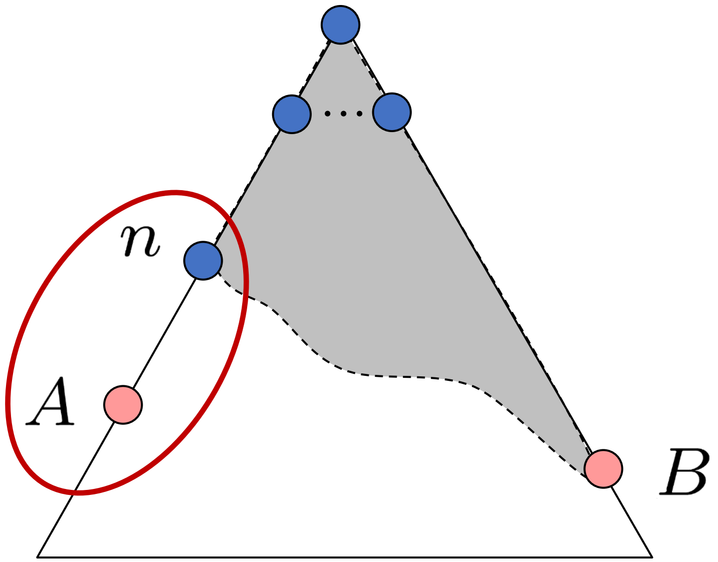
- Assume:
  - A is an optimal goal node
  - B is a suboptimal goal node
  - h is admissible

- Claim:
  - A will exit the fringe before B



Optimality of A* Tree Search: Blocking

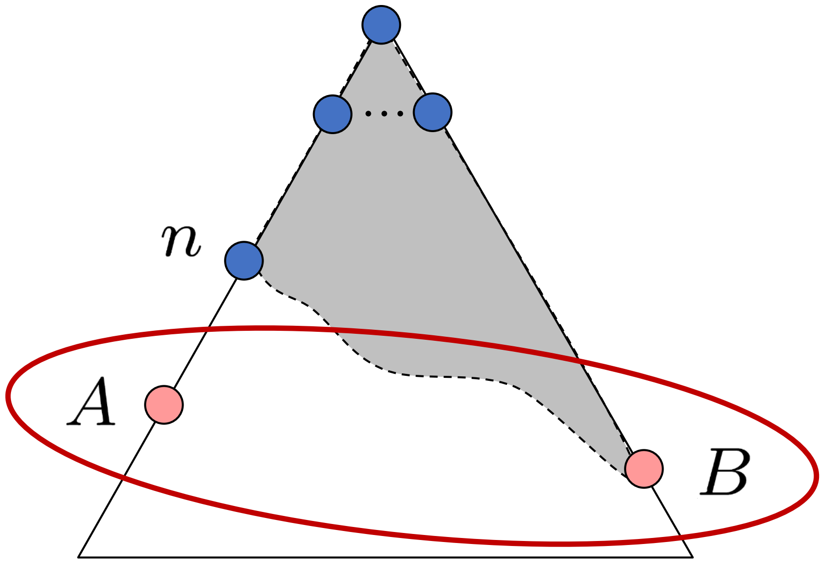
- Proof:
  - Imagine B is on the fringe
  - Some ancestor n of A is on the fringe, too (maybe A!)
  - Claim: n will be expanded before B
    1. f(n) is less or equal to f(A)



$f(n) = g(n) + h(n)$	Definition of f-cost
$f(n) \leq g(A)$	Admissibility of h
$g(A) = f(A)$	h = 0 at a goal

Optimality of A* Tree Search: Blocking 2

- Proof:
  - Imagine B is on the fringe
  - Some ancestor n of A is on the fringe, too (maybe A!)
  - Claim: n will be expanded before B
    1. f(n) is less or equal to f(A)
    2. f(A) is less than f(B)

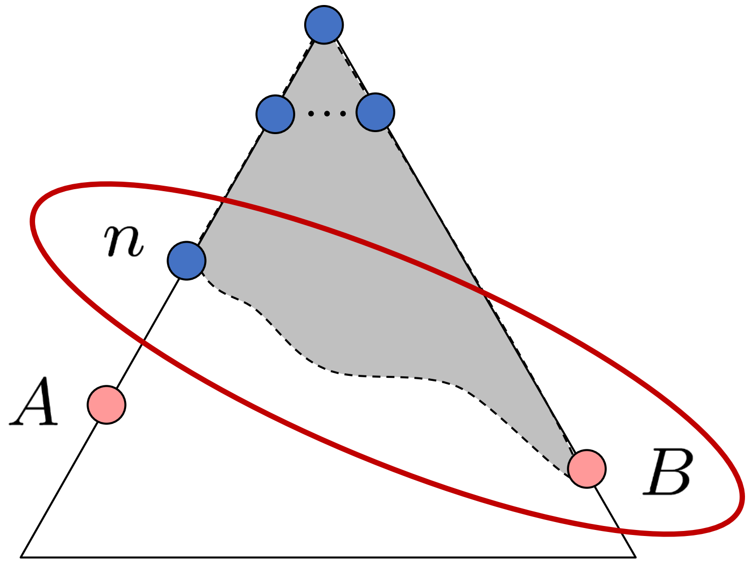


$g(A) < g(B)$ B is suboptimal

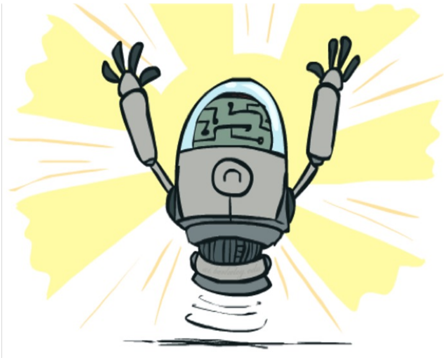
$f(A) < f(B)$ h = 0 at a goal

Optimality of A* Tree Search: Blocking 3

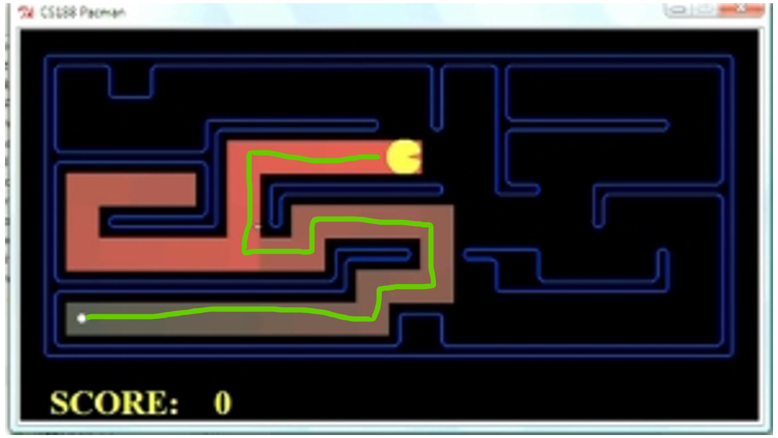
- Proof:
  - Imagine B is on the fringe
  - Some ancestor n of A is on the fringe, too (maybe A!)
  - Claim: n will be expanded before B
    1. f(n) is less or equal to f(A)
    2. f(A) is less than f(B)
    3. *n* expands before B
  - All ancestors of A expand before B
  - A expands before B
  - A* search is optimal

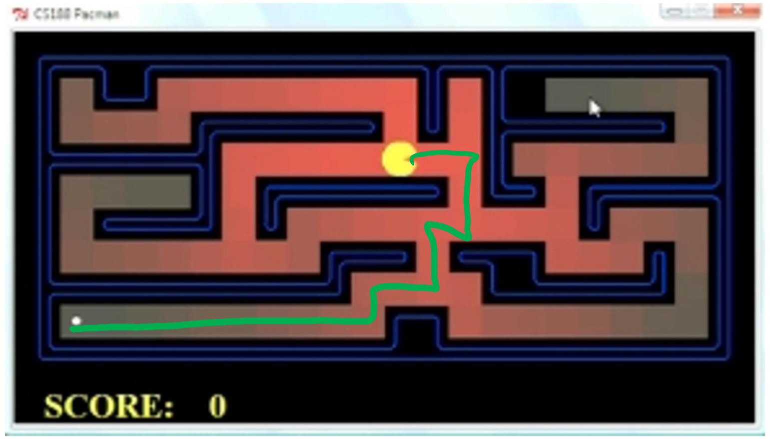


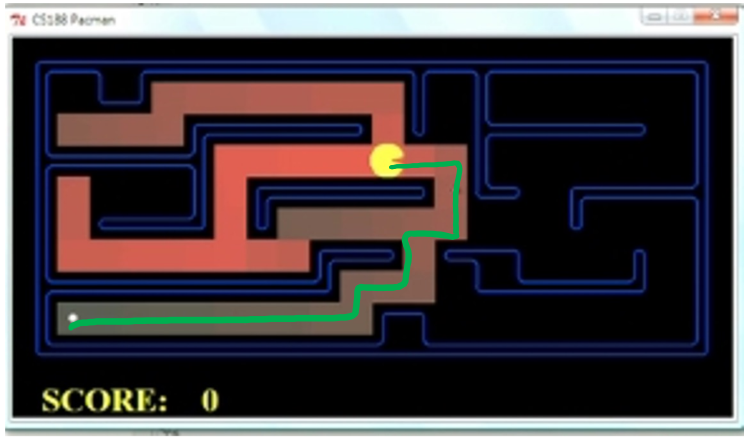
$$f(n) \leq f(A) < f(B)$$



Comparison





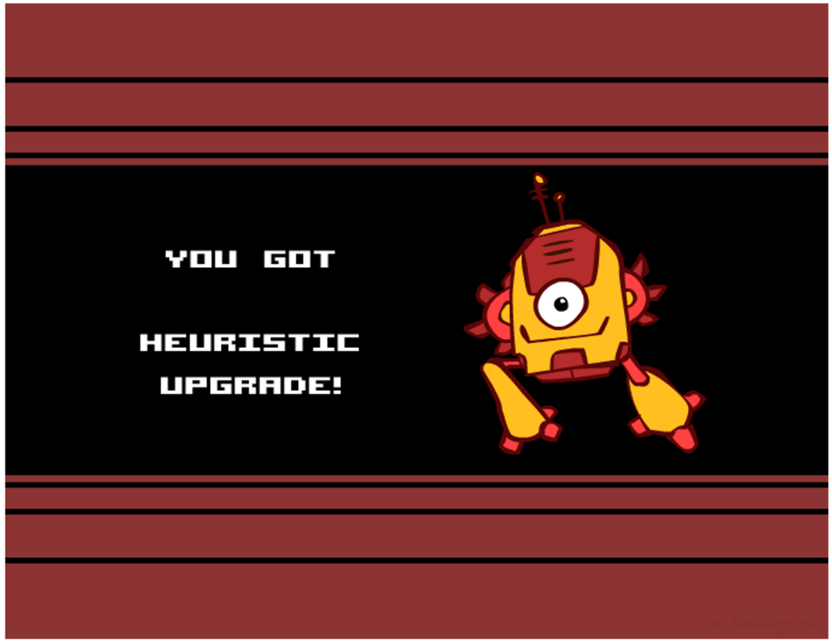


Greedy

Uniform Cost

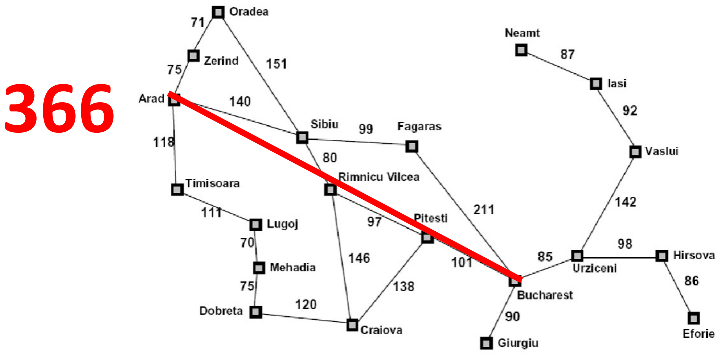
A*

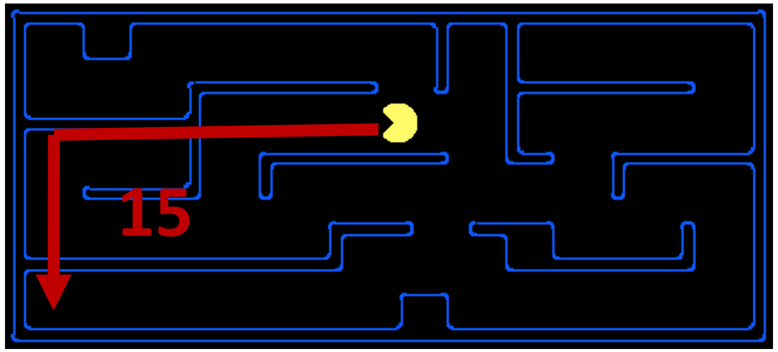
Creating Heuristics



- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics

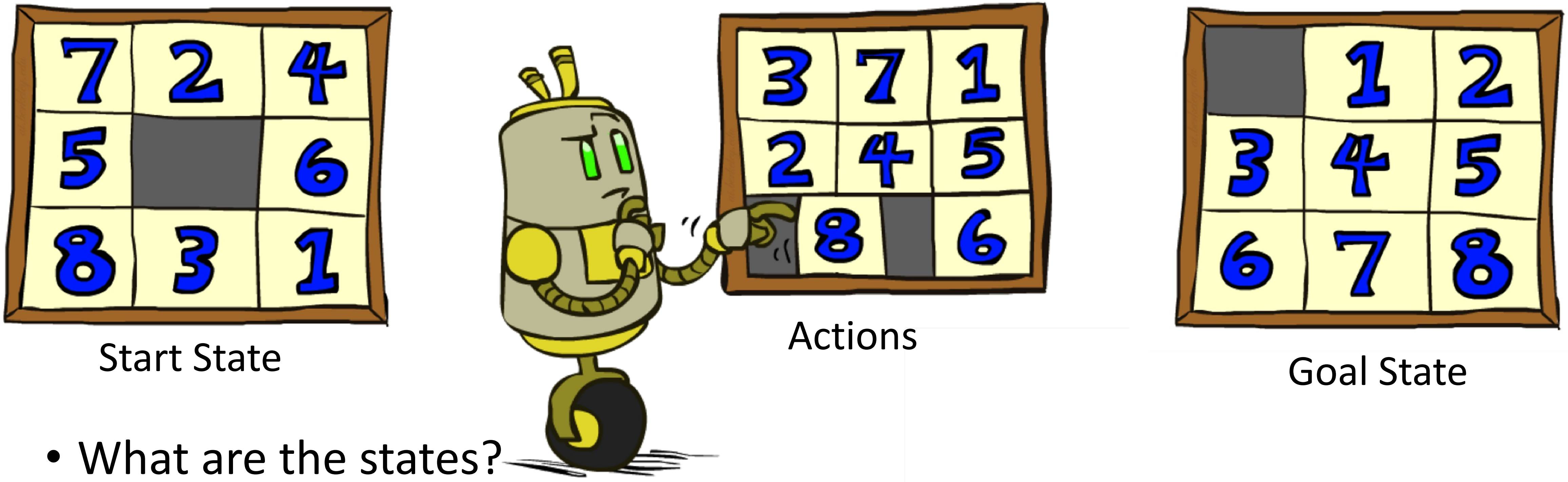
- Often, admissible heuristics are solutions to relaxed problems, where new actions are available





- Inadmissible heuristics are often useful too

Example: 8 Puzzle

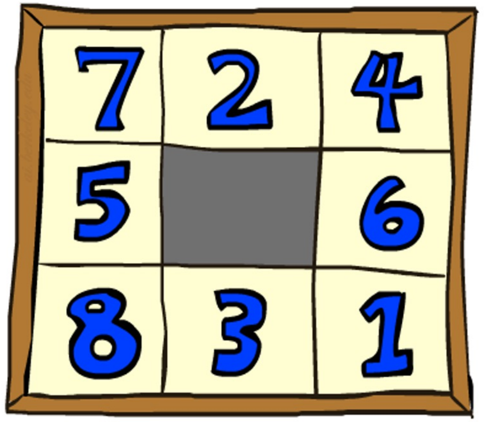


- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

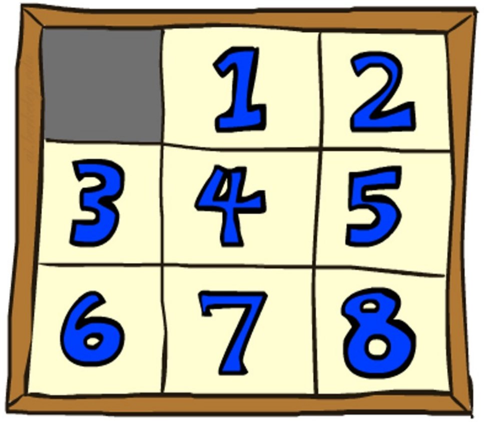
Admissible heuristics?

Example: 8 Puzzle - 2

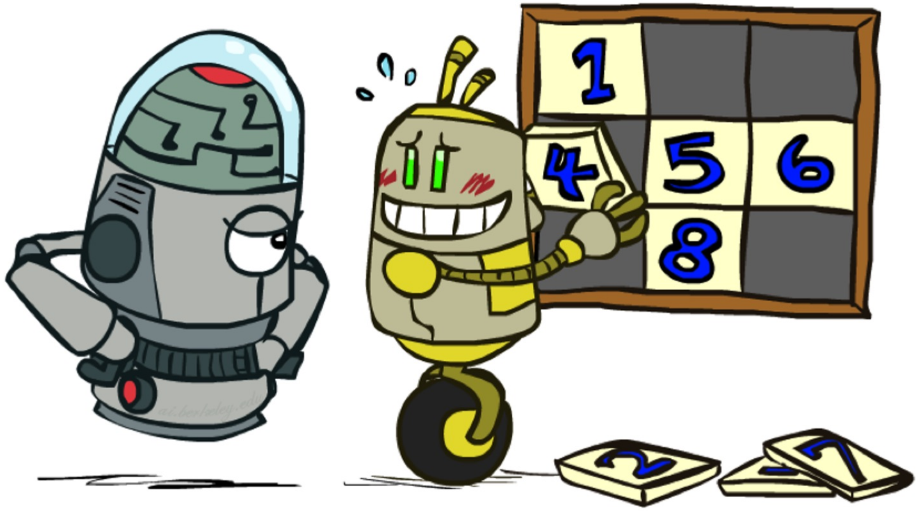
- Heuristic: Number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a relaxed-problem heuristic



Start State



Goal State

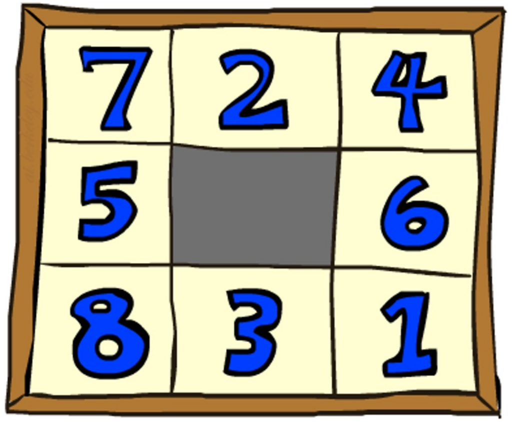


	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
TILES	13	39	227

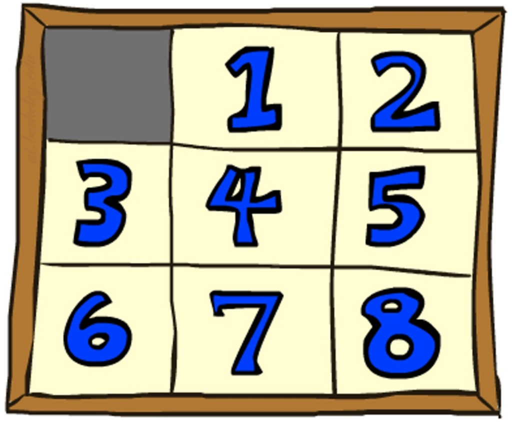
Statistics from Andrew Moore

Example: 8 Puzzle - 3

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?



Start State



Goal State

- Total Manhattan distance

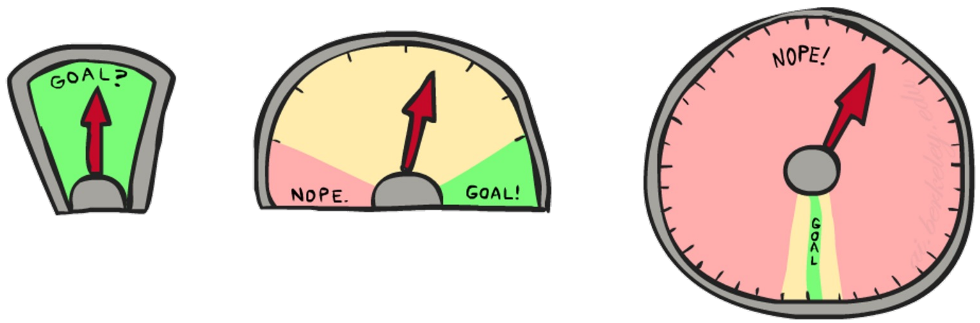
- Why is it admissible?

- h(start) = 3 + 1 + 2 + … = 18

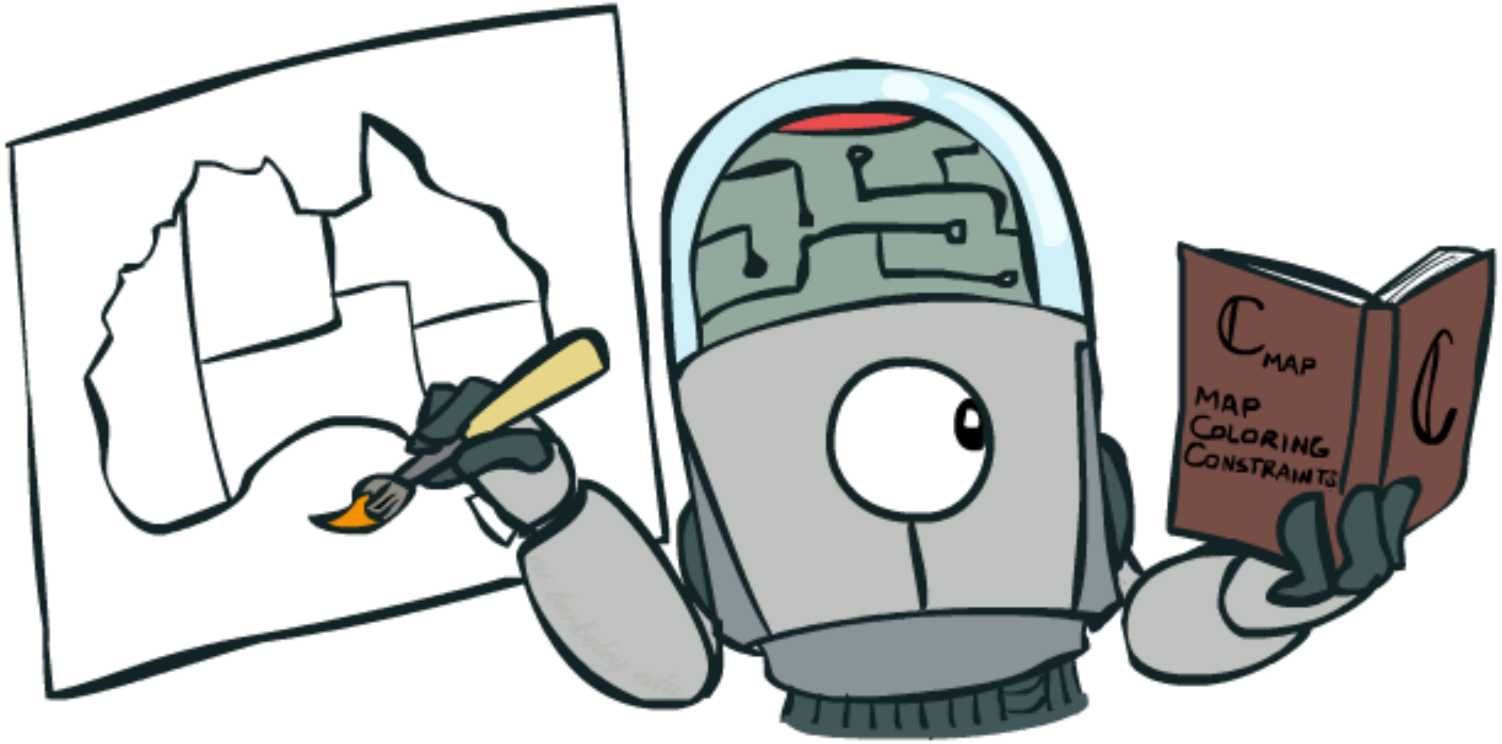
	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
TILES	13	39	227
MANHATTAN	12	25	73

Example: 8 Puzzle - 4

- How about using the actual cost as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What’s wrong with it?



- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself



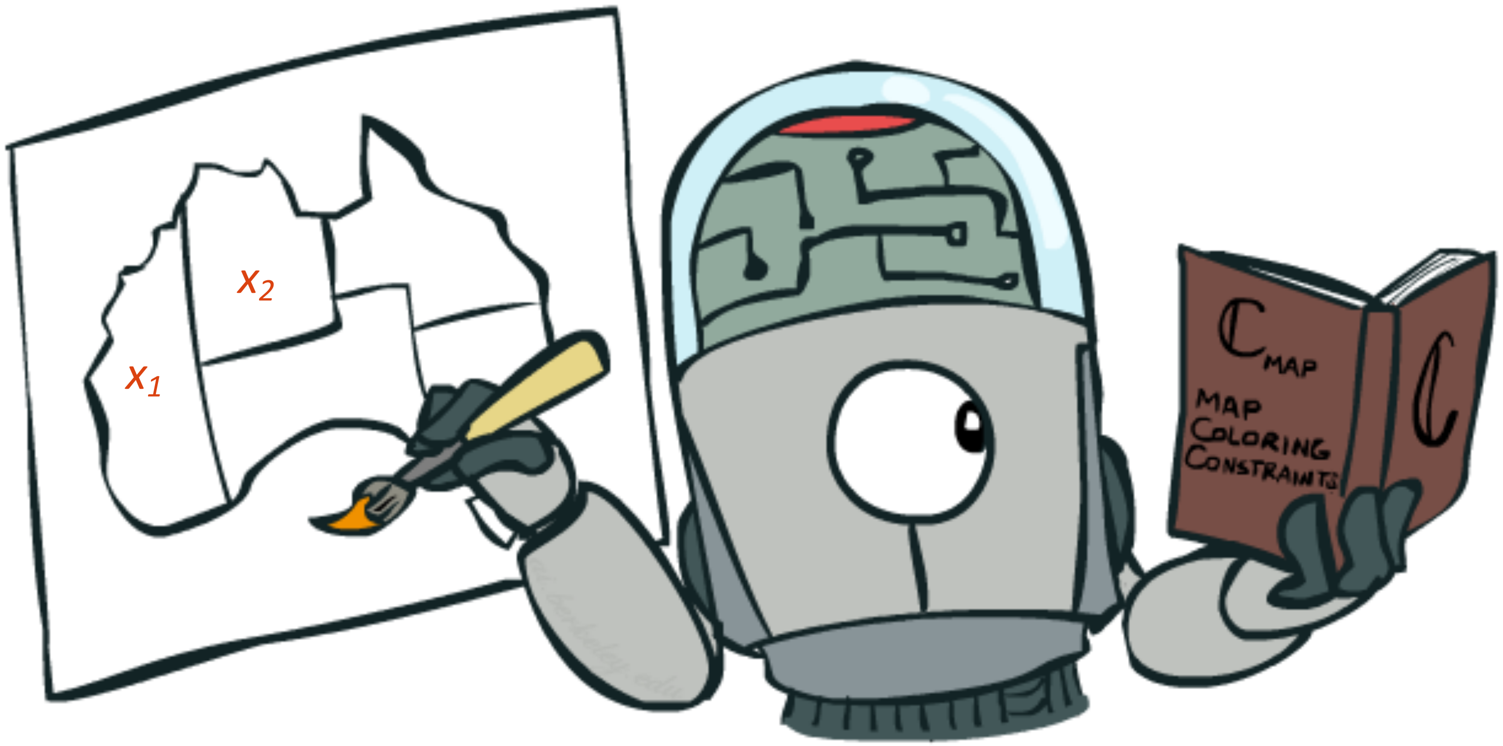
Constraint Satisfaction Problems

Constraint Satisfaction Problems

N variables

domain D

constraints



states

partial assignment

goal test

complete; satisfies constraints

successor function

assign an unassigned variable

What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

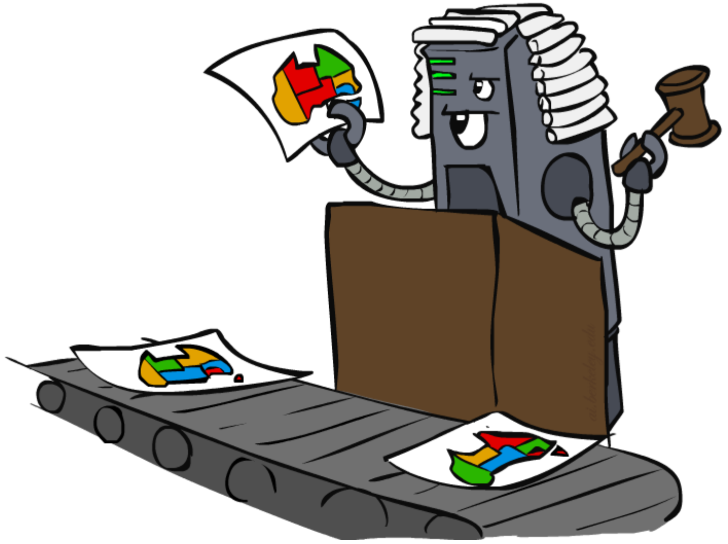
- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems

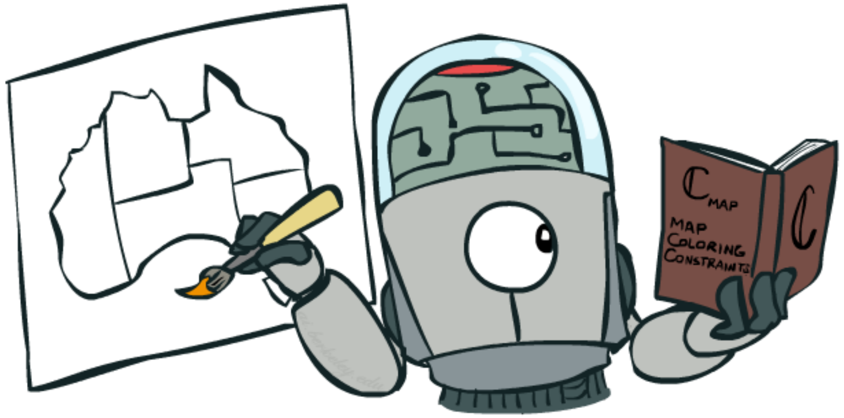
Constraint Satisfaction Problems

- Standard search problems:
  - State is a "black box": arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

- 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

- Allows useful general-purpose algorithms with more power than standard search algorithms



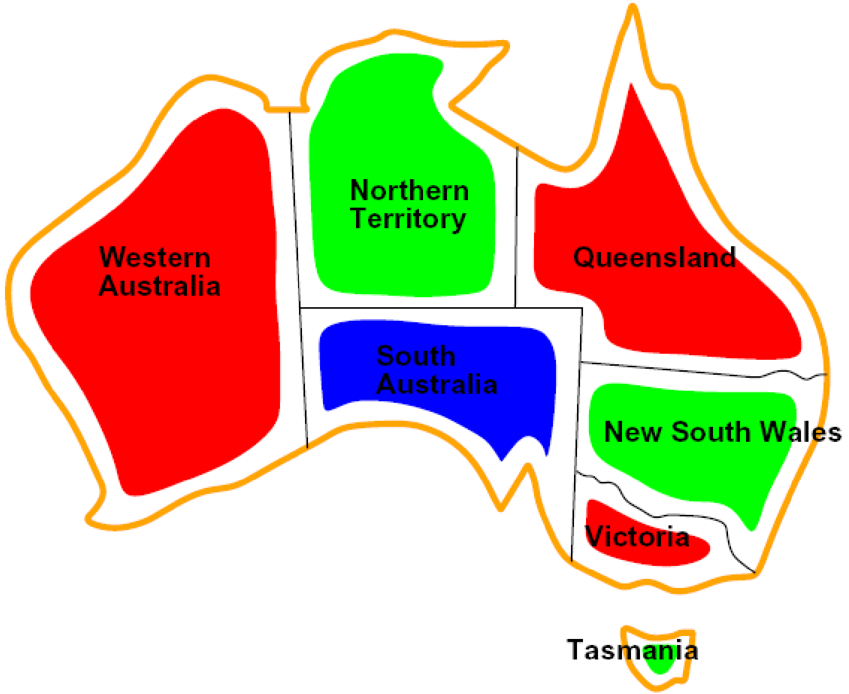


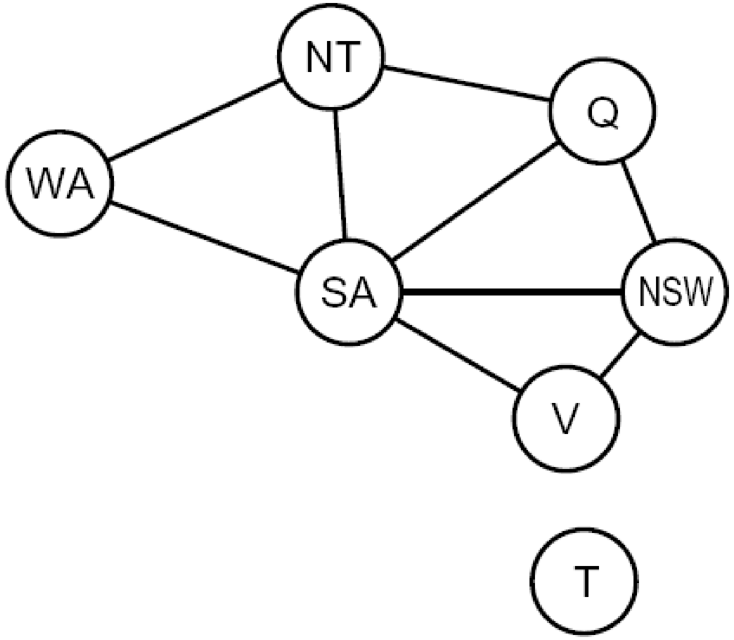
Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables

- Binary constraint graph: nodes are variables, arcs show constraints

- General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!



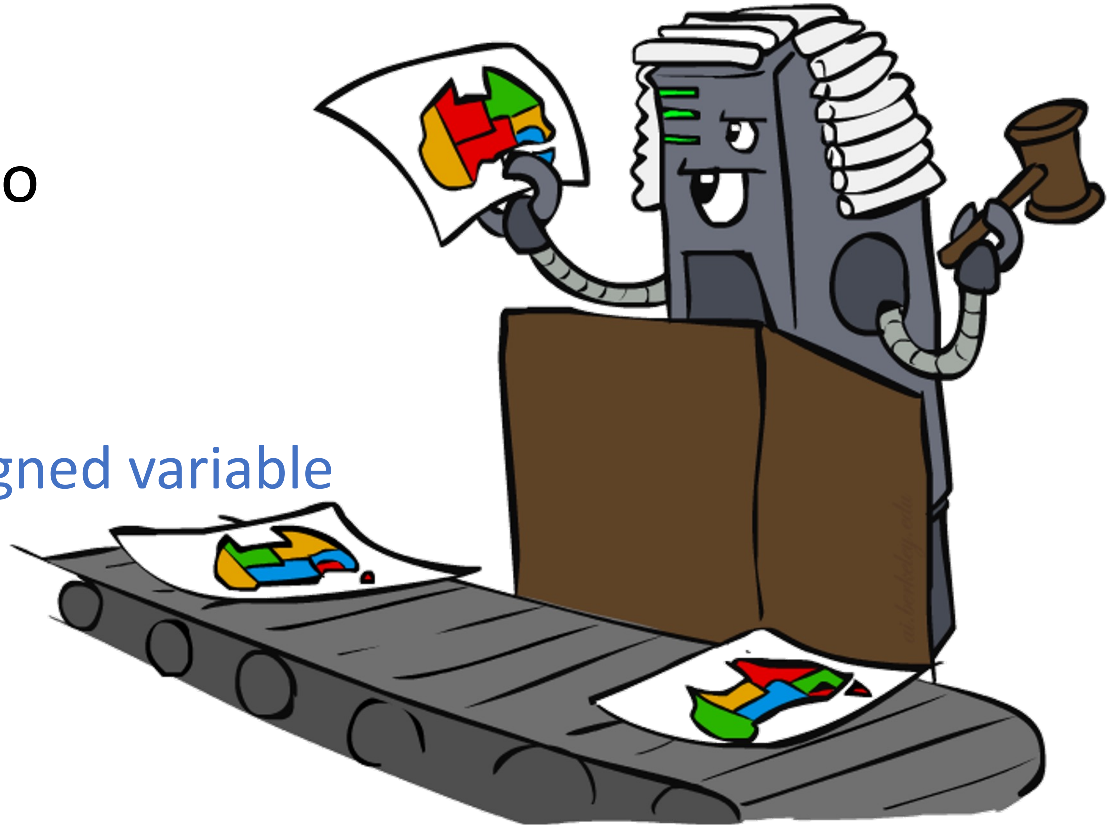


Standard Search Formulation

- Standard search formulation of CSPs

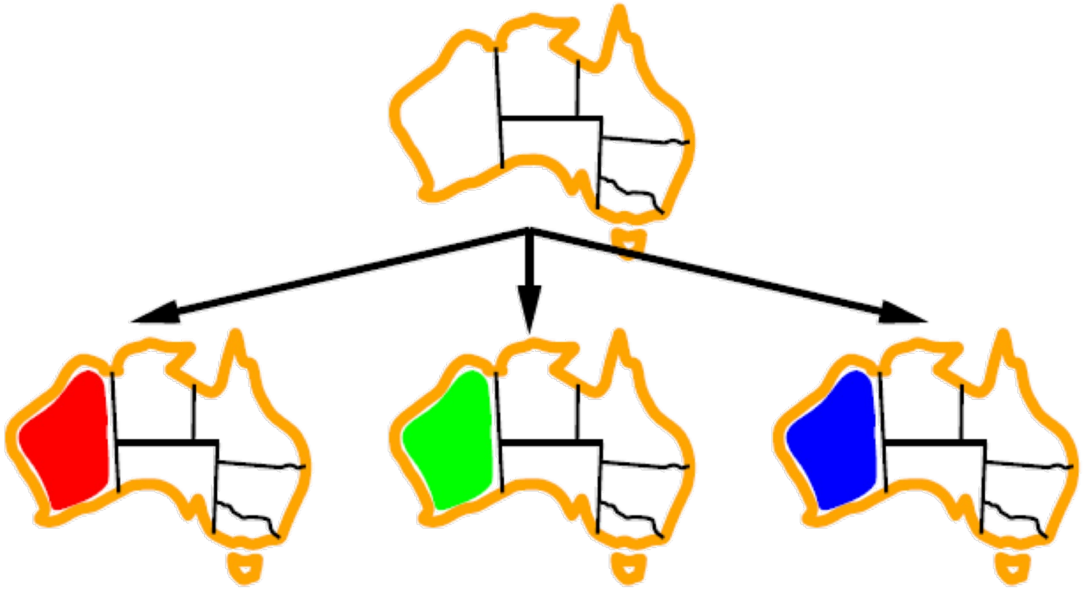
- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable →Can be any unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- We'll start with the straightforward, naïve approach, then improve it



Search Methods: DFS

- At each node, assign a value from the domain to the variable
- Check feasibility (constraints) when the assignment is complete



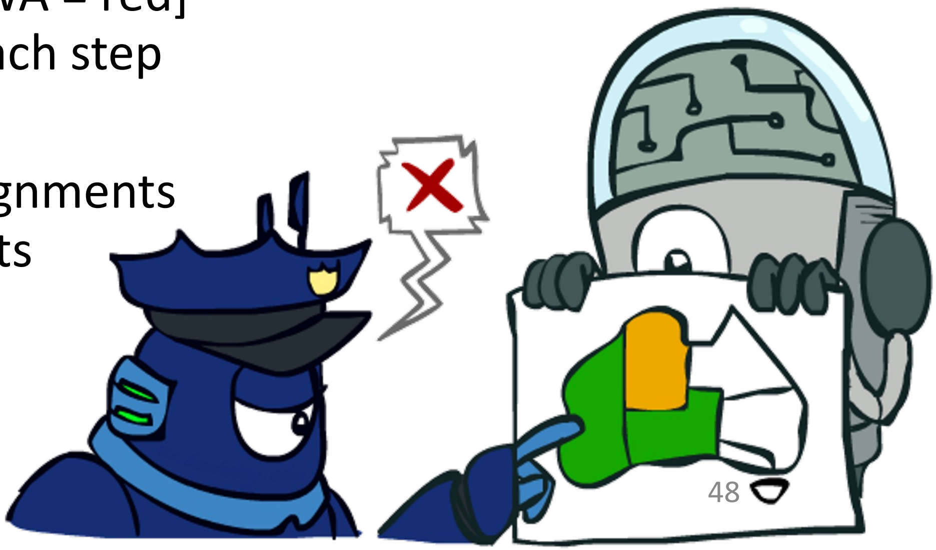
- What problems does the naïve search have?

[Demo: coloring -- dfs]

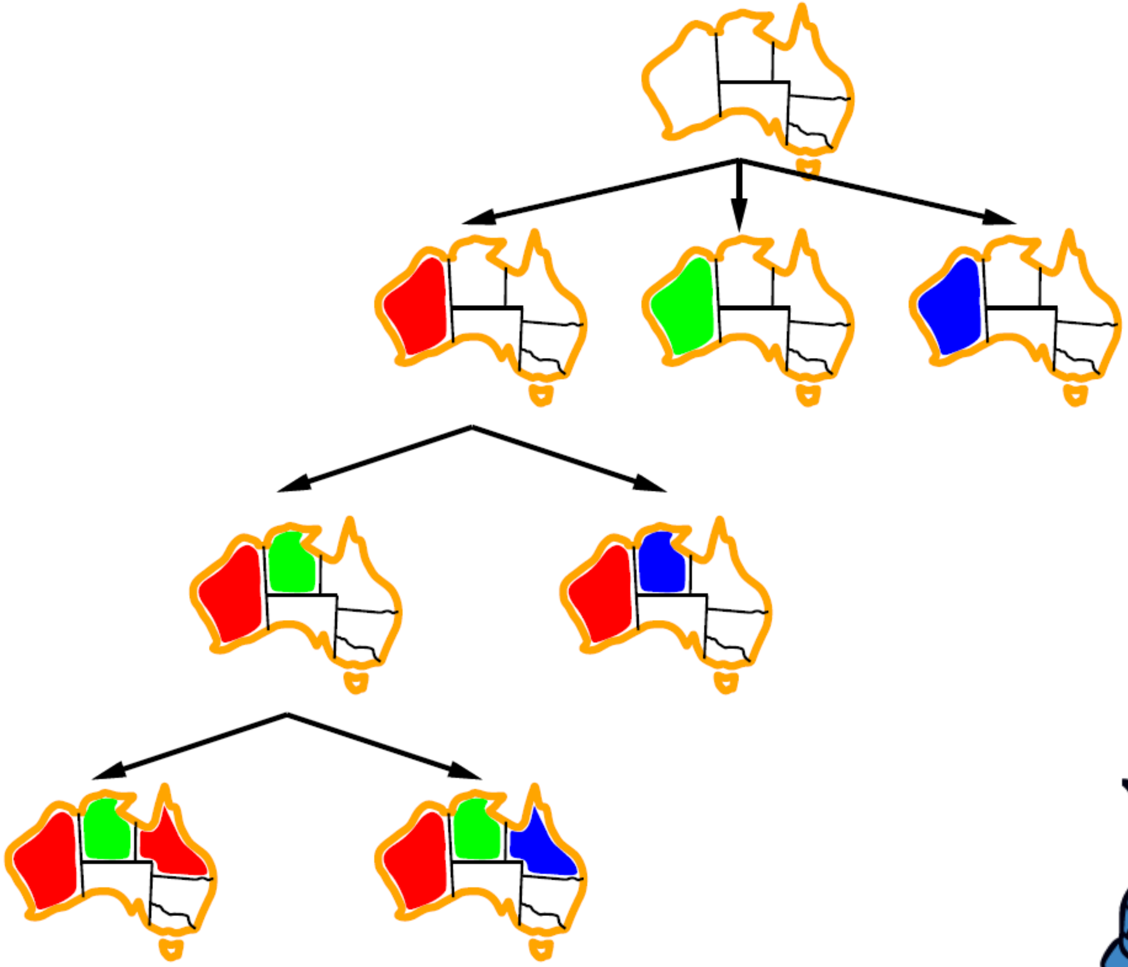
Backtracking Search

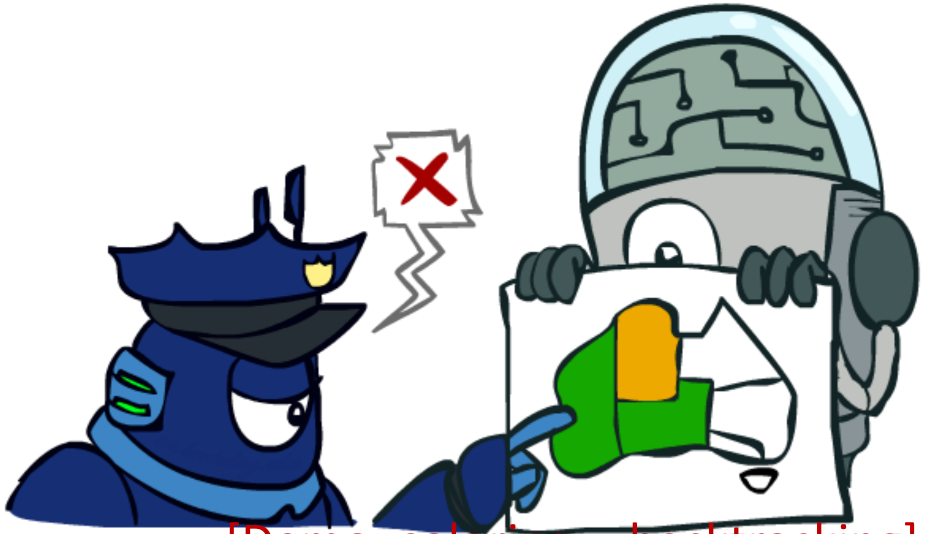
- Backtracking search is the basic uninformed algorithm for solving CSPs
- Backtracking search = DFS + two improvements

- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - 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"
- Can solve N-queens for $N \approx 25$



Example





[Demo: coloring -- backtracking]

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return RECURSIVE_BACKTRACKING({}, csp)

function RECURSIVE_BACKTRACKING(assignment, csp) returns a solution, or 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
```

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return RECURSIVE_BACKTRACKING({}, csp)

function RECURSIVE_BACKTRACKING(assignment, csp) returns a solution, or 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
```

No need to check consistency for a complete assignment

What are choice points?

Checks consistency at each assignment

Backtracking = DFS + variable-ordering + fail-on-violation

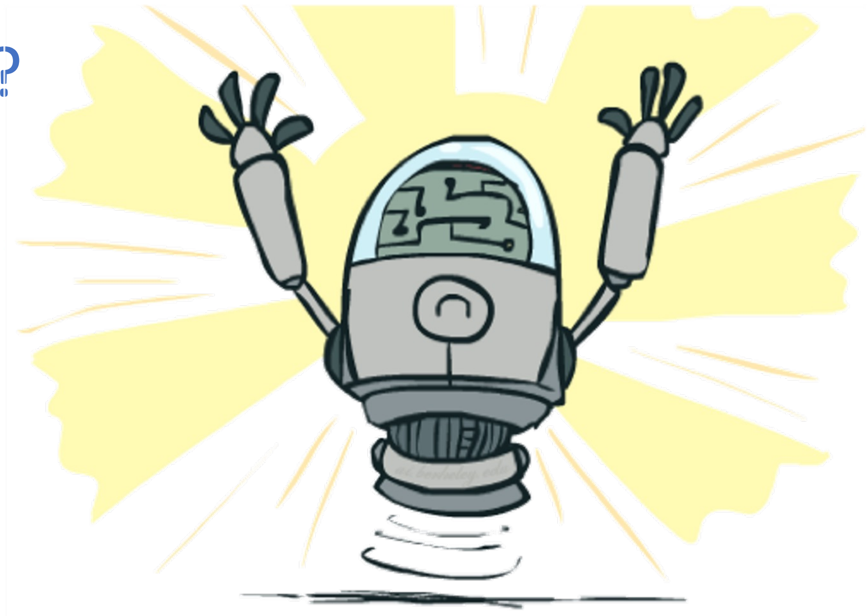
Improving Backtracking

- General-purpose ideas give huge gains in speed

- Filtering: Can we detect inevitable failure early?

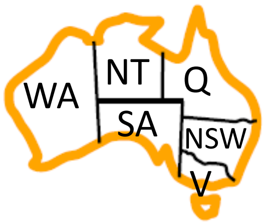
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

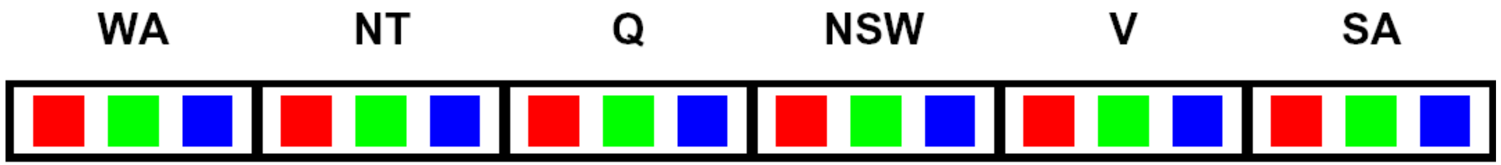
- Structure: Can we exploit the problem structure?



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 failure is detected if some variables have no values remaining

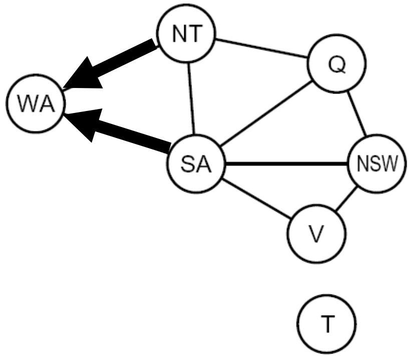


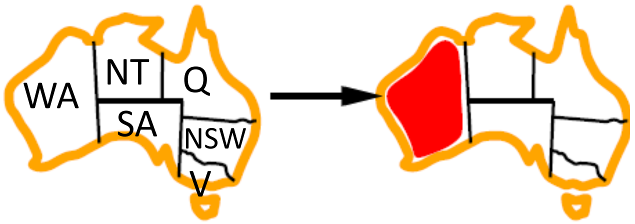


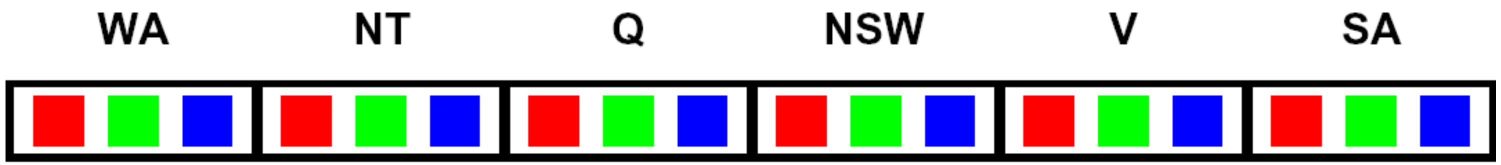
[Demo: coloring -- forward checking]

Filtering: Forward Checking 2

- 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





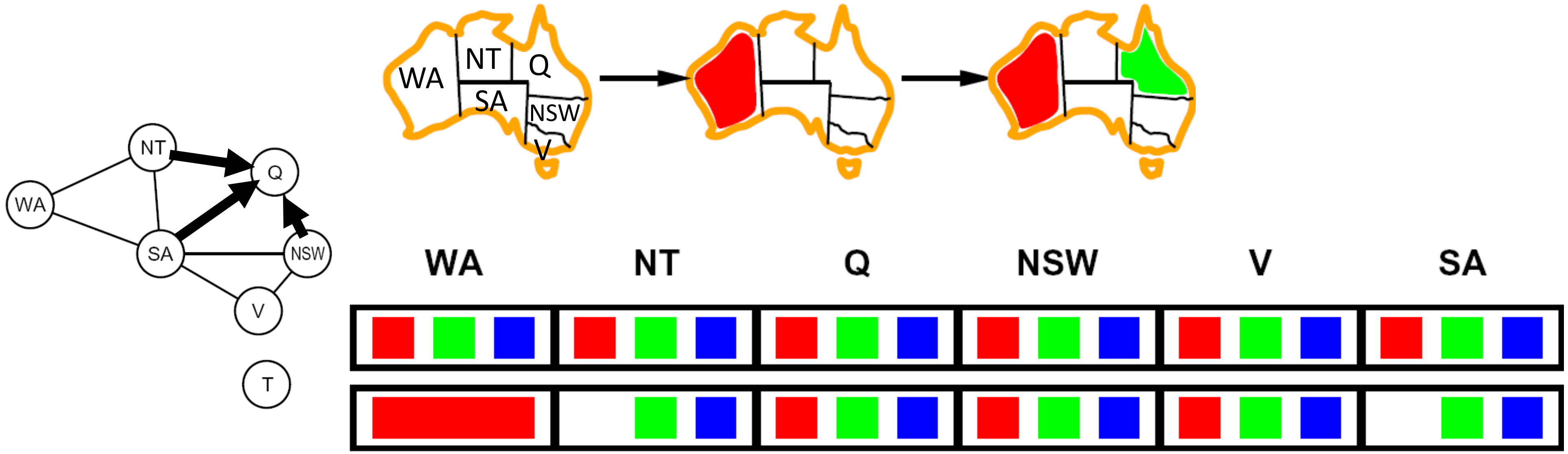


Recall: Binary constraint graph for a binary CSP (i.e., each constraint has most two variables): nodes are variables, edges show constraints

[Demo: coloring -- forward checking]

Filtering: Forward Checking 3

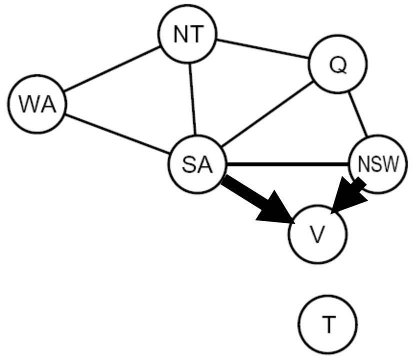
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment

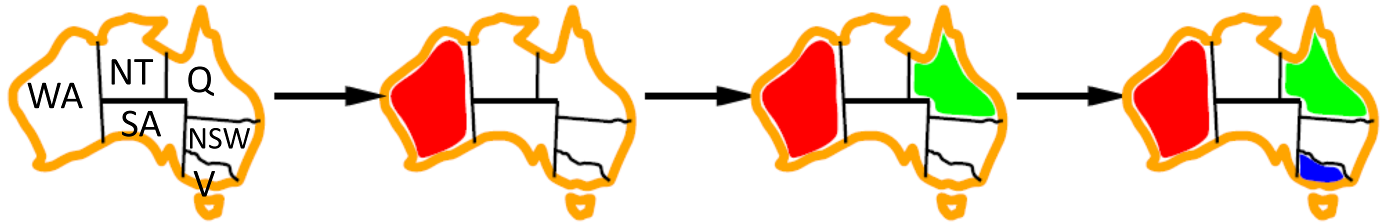


[Demo: coloring -- forward checking]

Filtering: Forward Checking 4

- 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



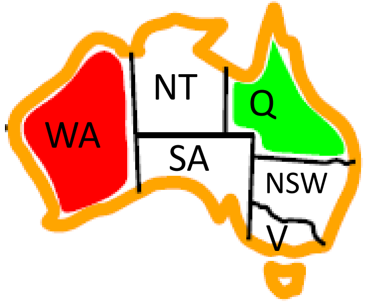


FAIL – variable with no possible values

[Demo: coloring -- forward checking]

Filtering: Constraint Propagation

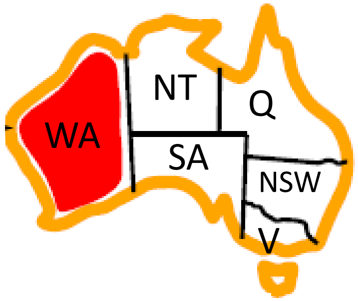
- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

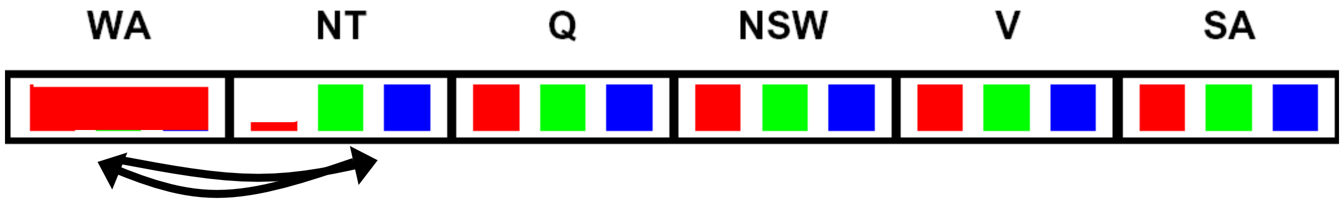


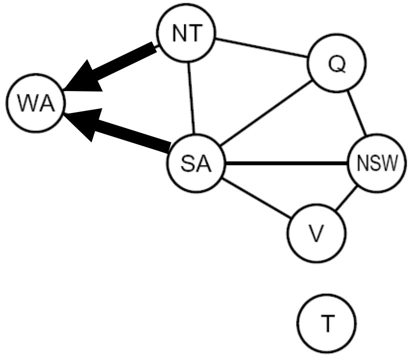
- NT and SA cannot both be blue!
- Why didn't we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

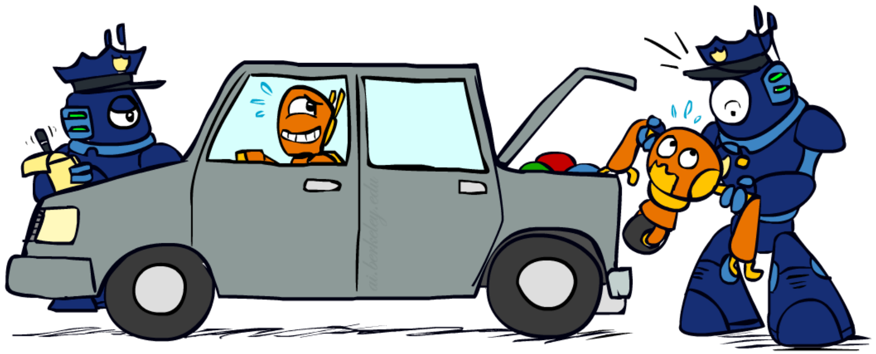
Consistency of A Single Arc

- An arc X $\rightarrow$ 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









Delete from the tail!

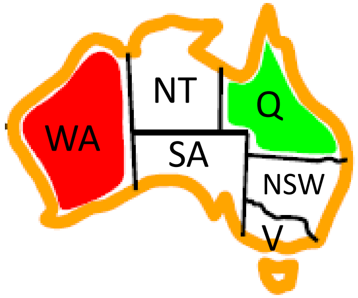
Forward checking?

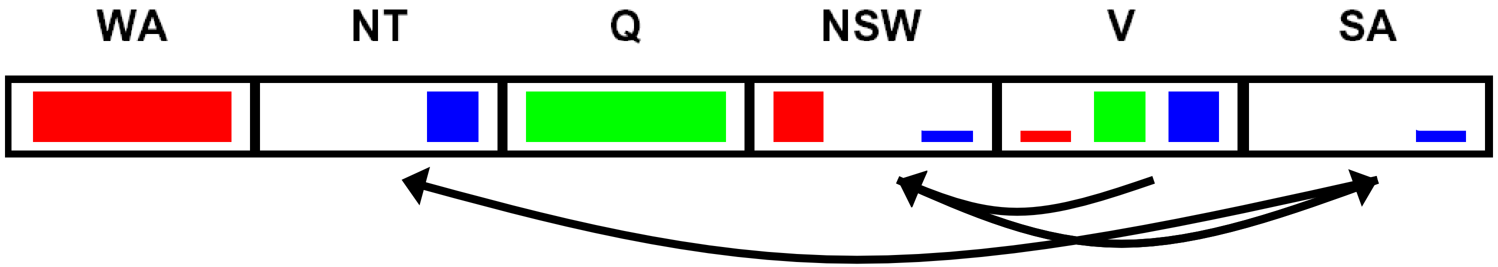
A special case

Enforcing consistency of arcs pointing to each new assignment

Arc Consistency of an Entire CSP

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



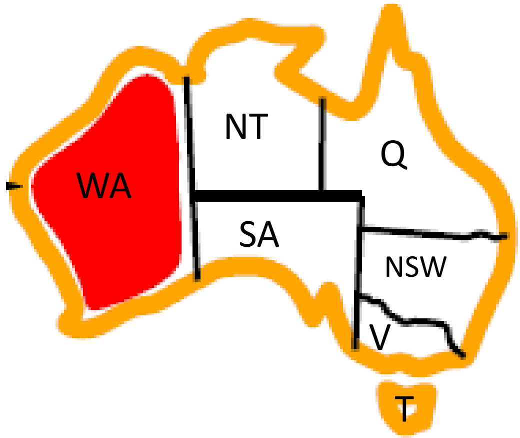


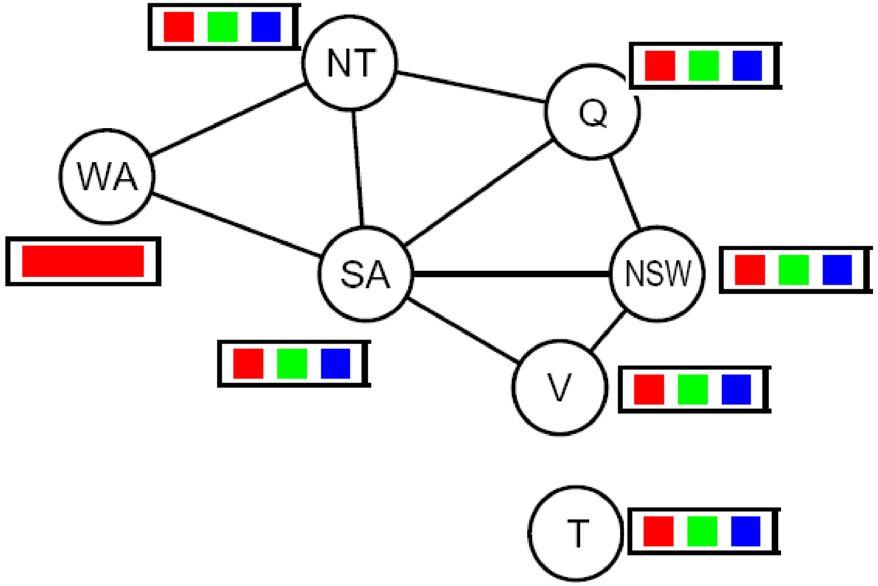
- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Arc Consistency of Entire CSP 2

- A simplistic algorithm: Cycle over the pairs of variables, enforcing arc-consistency, repeating the cycle until no domains change for a whole cycle
- AC-3 (Arc Consistency Algorithm #3):
  - A more efficient algorithm ignoring constraints that have not been modified since they were last analyzed





```
function AC-3(csp) returns the CSP, possibly with reduced domains
    initialize a queue of all the arcs in csp
    while queue is not empty do
        (X_i, X_j) ⟵ REMOVE_FIRST(queue)
        if REMOVE_INCONSISTENT_VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue

function REMOVE_INCONSISTENT_VALUES(X_i, X_j)  returns true iff succeeds
    removed ⟵ false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ⟷ X_j then
            delete x from DOMAIN[X_i]; removed ⟵ true
    return removed
```

```
function AC-3(csp) returns the CSP, possibly with reduced domains
    initialize a queue of all the arcs in csp
    while queue is not empty do
        (X_i, X_j) ⟵ REMOVE_FIRST(queue)
        if REMOVE_INCONSISTENT_VALUES(X_i, X_j) then          Constraint Propagation!
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue

function REMOVE_INCONSISTENT_VALUES(X_i, X_j)  returns true iff succeeds
    removed ⟵ false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ⟷ X_j then
            delete x from DOMAIN[X_i]; removed ⟵ true
    return removed
```

function AC-3(csp) returns the CSP, possibly with reduced domains

```
function AC-3(csp) returns the CSP, possibly with reduced domains
    initialize a queue of all the arcs in csp
    while queue is not empty do
        (X_i, X_j) ⟵ REMOVE_FIRST(queue)
        if REMOVE_INCONSISTENT_VALUES(X_i, X_j) then
            for each X_k in NEIGHBORS[X_i] do
                add (X_k, X_i) to queue
```

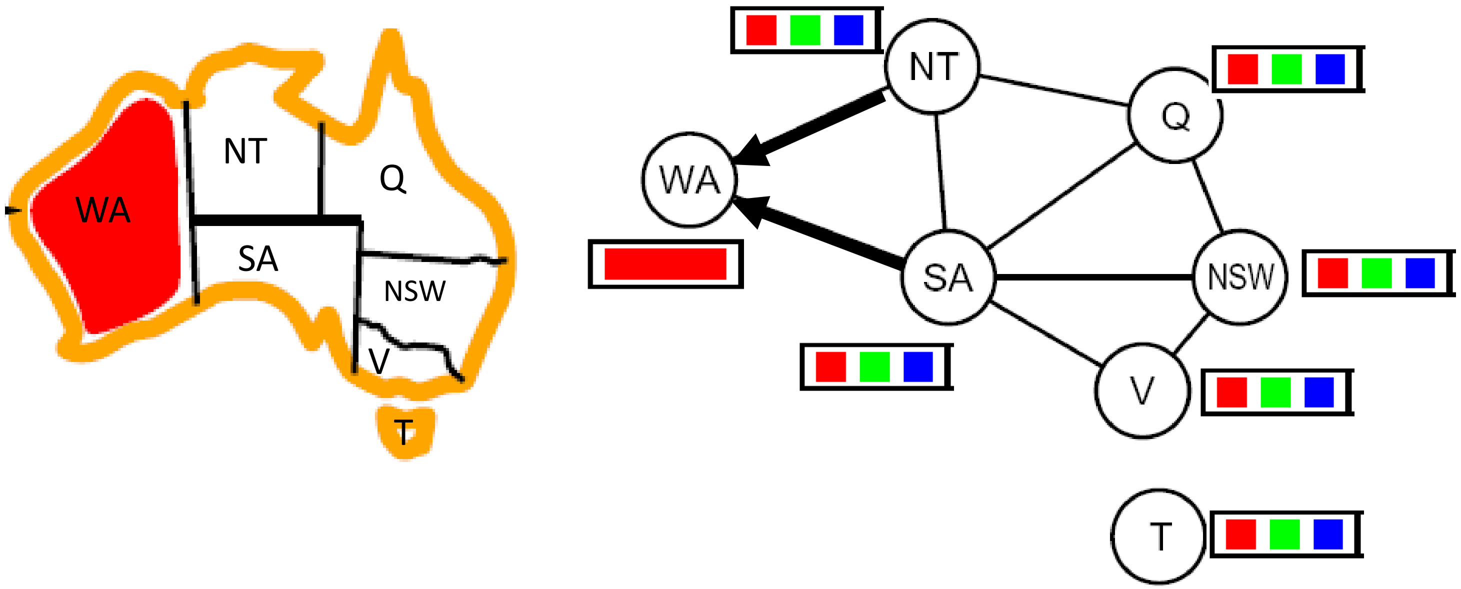
- An arc is added after a removal of value at a node
- n node in total, each has $\leq d$ values
- Total times of removal: $O(nd)$
- After a removal, $\leq n$ arcs added
- Total times of adding arcs: $O(n^2 d)$

```
function REMOVE_INCONSISTENT_VALUES(X_i, X_j) returns true iff succeeds
    removed ⟵ false
    for each x in DOMAIN[X_i] do
        if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ⟷ X_j then
            delete x from DOMAIN[X_i]; removed ⟵ true
    return removed
```

- Check arc consistency per arc: $O(d^2)$
- Complexity: $O(n^2 d^3)$
- Can be improved to $O(n^2 d^2)$

… but detecting all possible future problems is NP-hard – why?

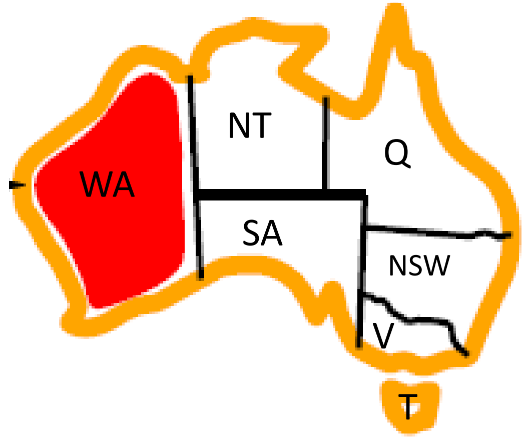
Example of AC-3

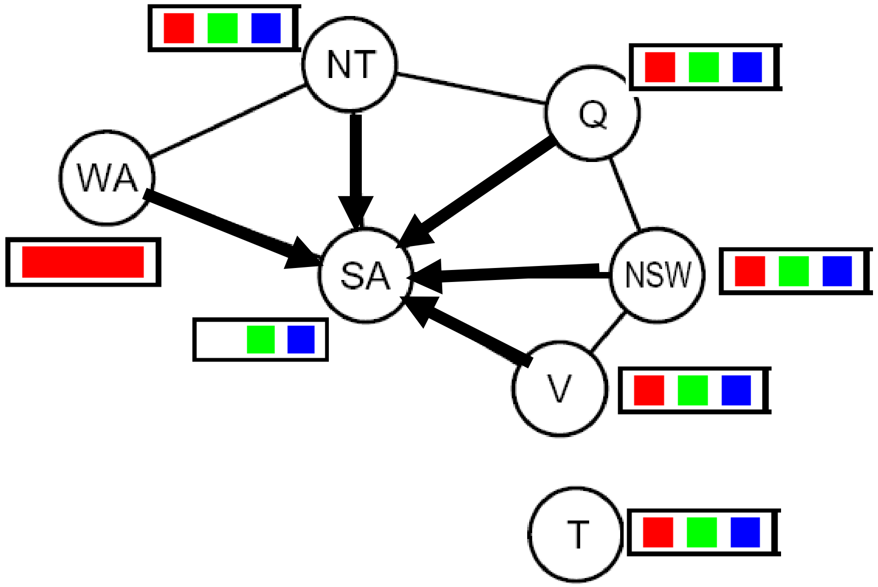


Queue:
SA->WA
NT->WA

Remember: Delete from the tail!

Example of AC-3 2

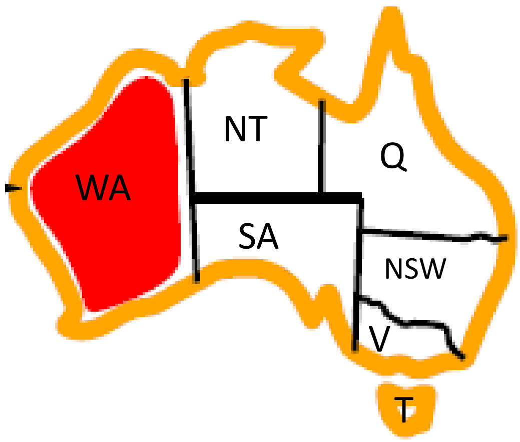


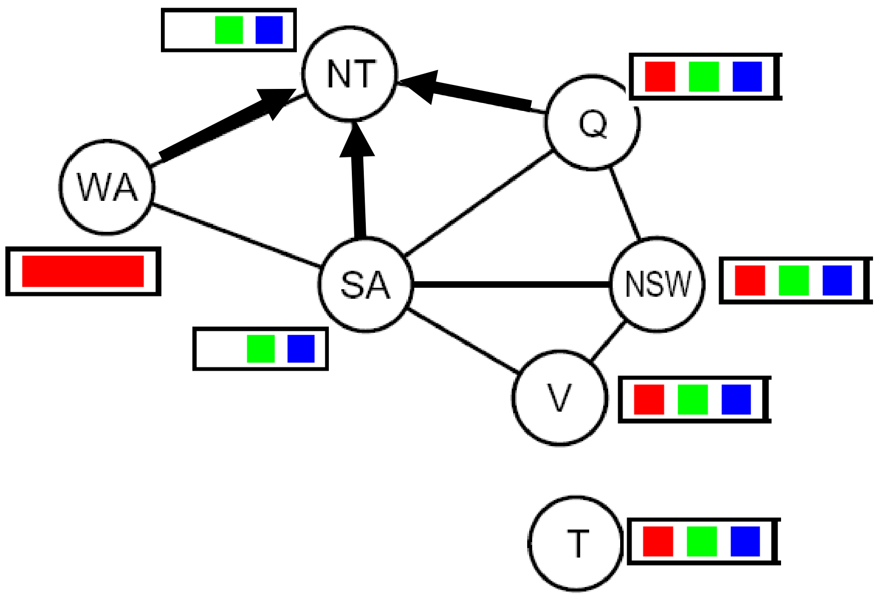


Queue:
~~SA->WA~~
NT->WA
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA

Remember: Delete from the tail!

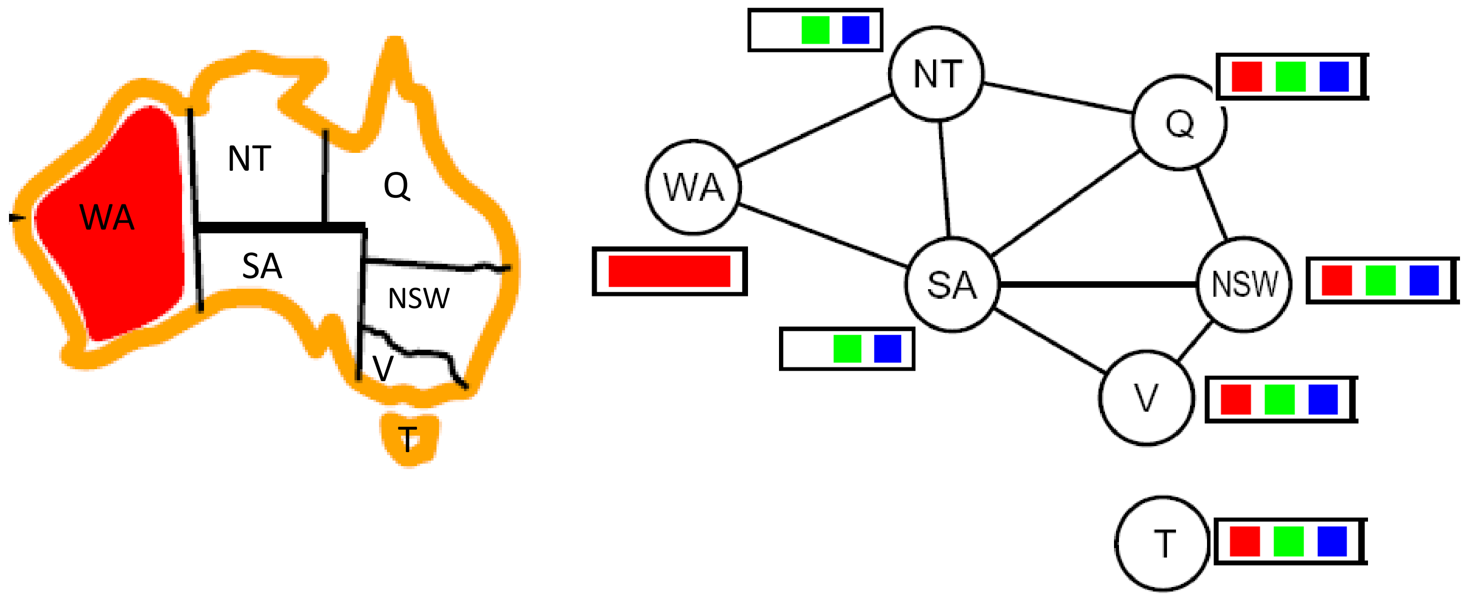
Example of AC-3 3





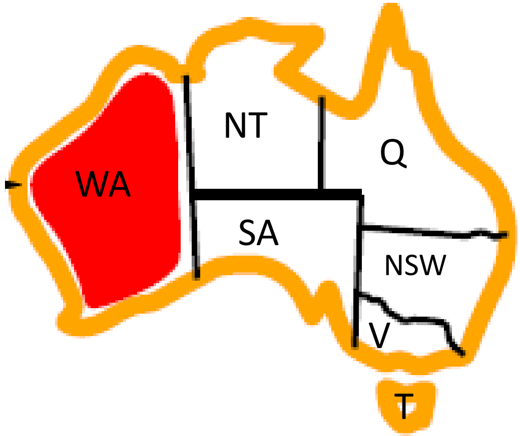
Queue:
~~SA->WA~~
~~NT->WA~~
WA->SA
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

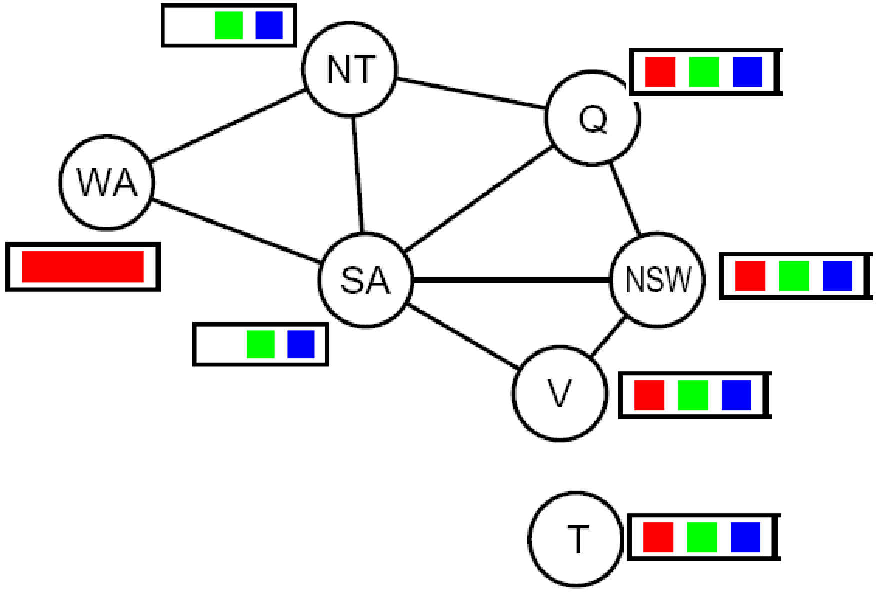
Example of AC-3 4



Queue:
~~SA->WA~~
~~NT->WA~~
~~WA->SA~~
NT->SA
Q->SA
NSW->SA
V->SA
WA->NT
SA->NT
Q->NT

Example of AC-3 5





Queue:

~~SA->WA~~

~~NT->WA~~

~~WA->SA~~

~~NT->SA~~

~~Q->SA~~

~~NSW->SA~~

~~V->SA~~

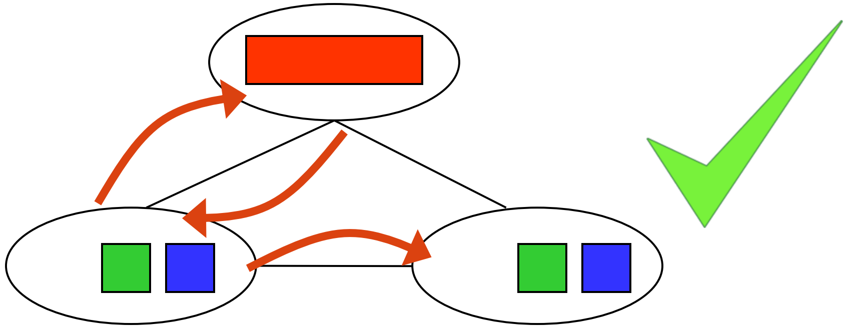
~~WA->NT~~

~~SA->NT~~

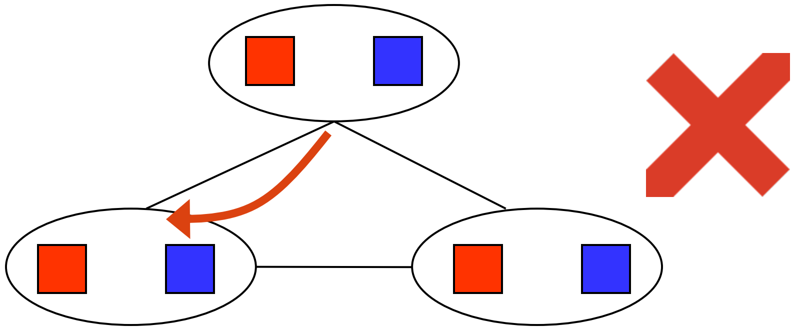
~~Q->NT~~

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!
- And will be called many times

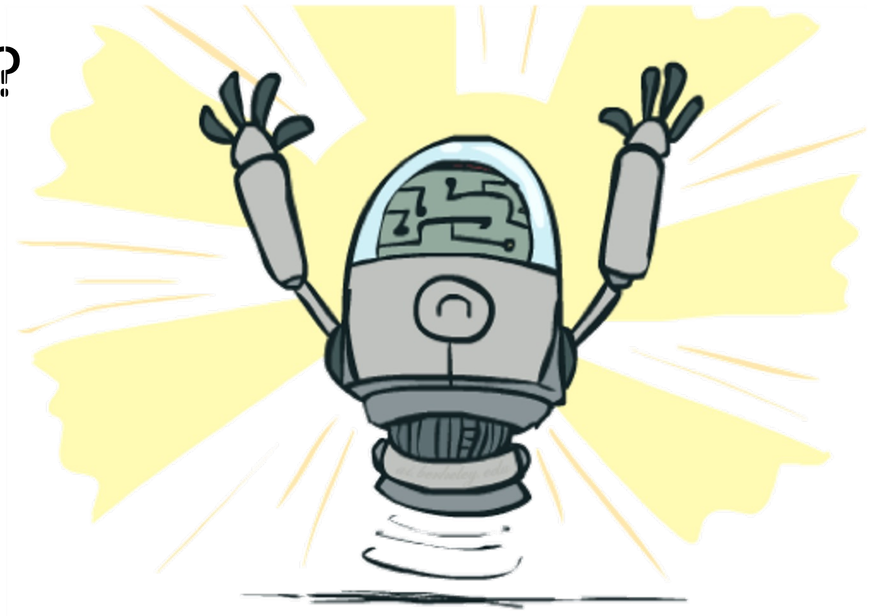


[Demo: coloring -- forward checking]
[Demo: coloring -- arc consistency]

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return RECURSIVE_BACKTRACKING({}, csp)

function RECURSIVE_BACKTRACKING(assignment, csp) returns a solution, or 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
                                                           AC-3(csp)
            result ⟵ RECURSIVE_BACKTRACKING(assignment, ~~csp~~)
            if result ≠ failure, then
                return result
            remove {var=value} from assignment
    return failure
```

Improving Backtracking

- General-purpose ideas give huge gains in speed
- Filtering: Can we detect inevitable failure early?
- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?

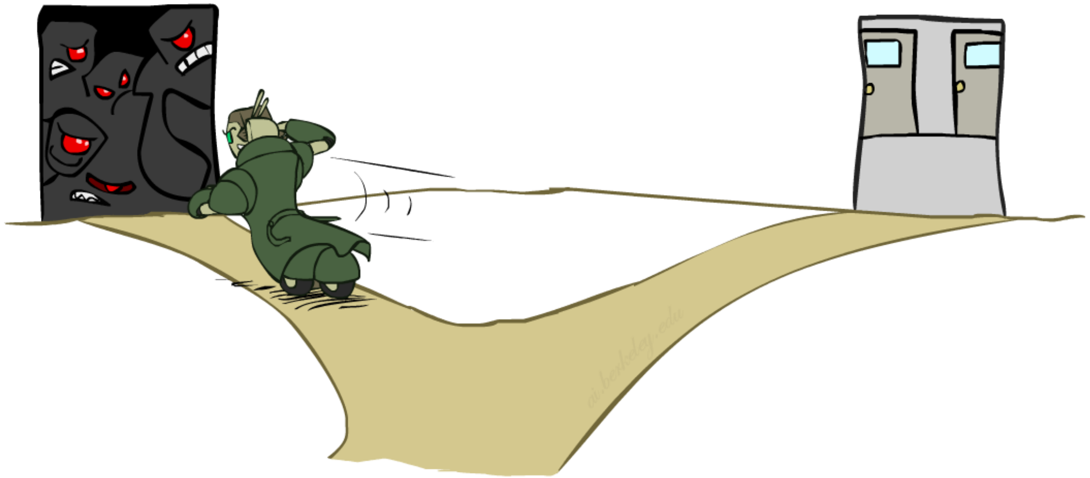


- Structure: Can we exploit the problem structure?

Ordering: Minimum Remaining Values

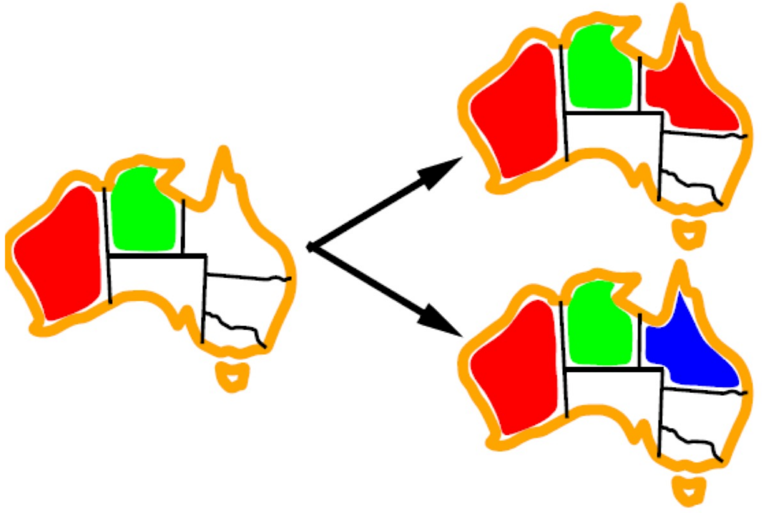
- Variable Ordering: Minimum remaining values (MRV):
  - Choose the variable with the fewest legal left values in its domain

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

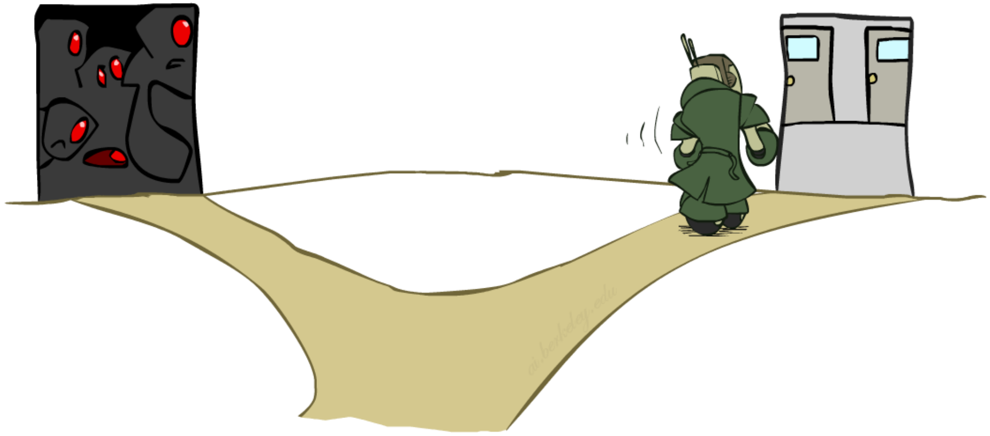


Ordering: Least Constraining Value

- Value Ordering: Least Constraining Value
  - 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?



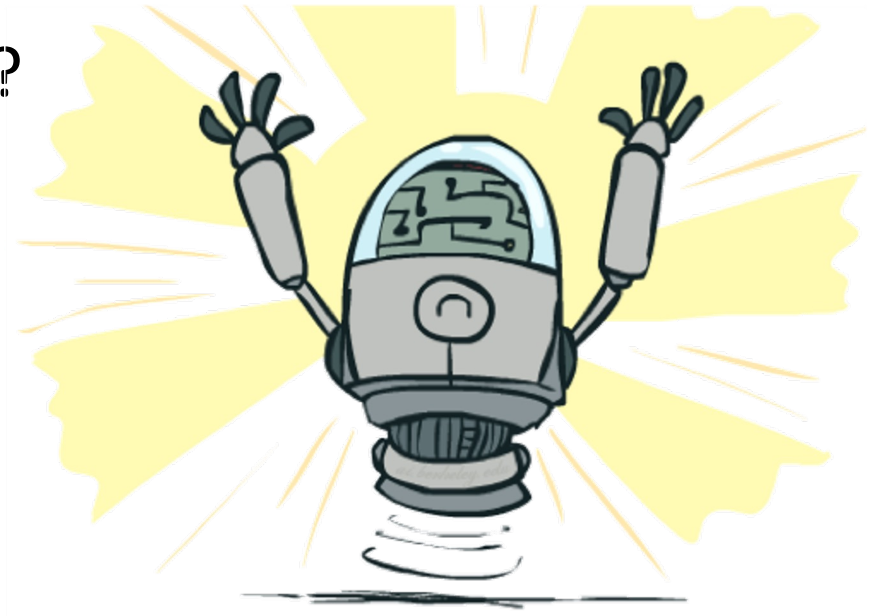
- Combining these ordering ideas makes 1000 queens feasible

Improving Backtracking

- General-purpose ideas give huge gains in speed

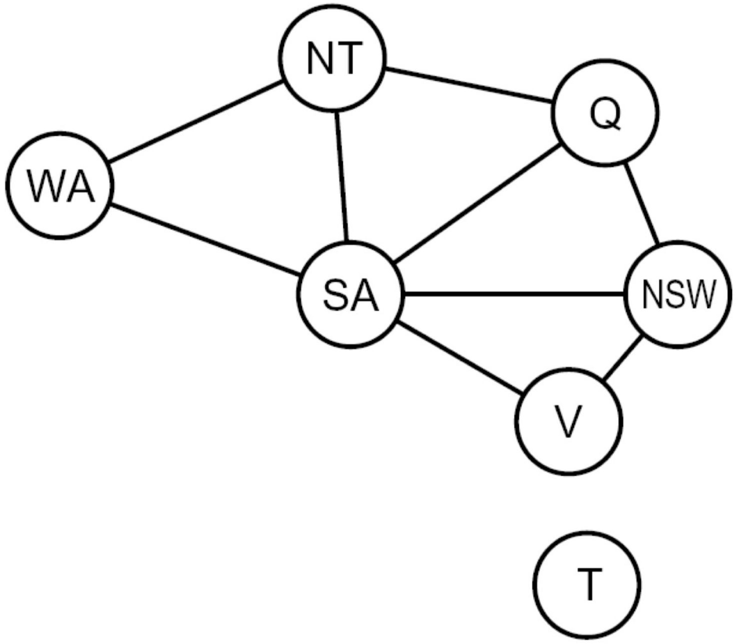
- Filtering: Can we detect inevitable failure early?

- Ordering:
  - Which variable should be assigned next?
  - In what order should its values be tried?



- Structure: Can we exploit the problem structure?

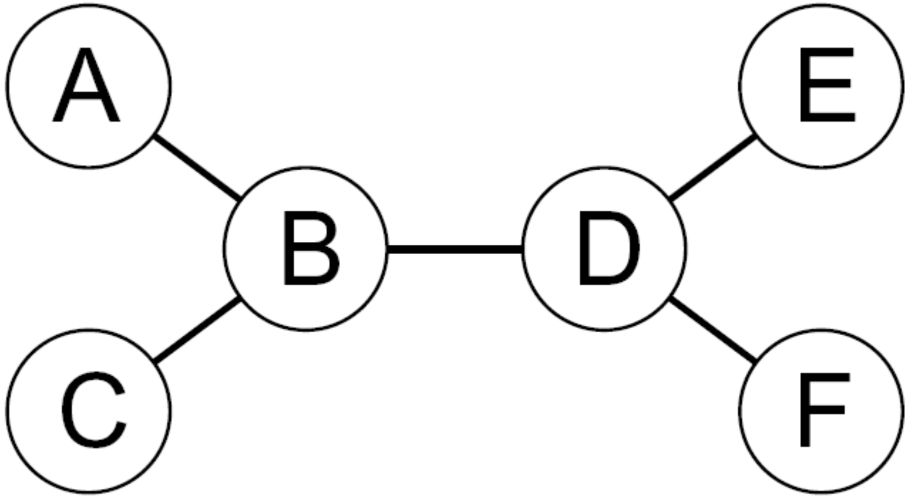
Problem Structure



- For general CSPs, worst-case complexity with backtracking algorithm is $O(d^n)$
- When the problem has special structure, we can often solve the problem more efficiently

- Special Structure 1: Independent subproblems
  - Example: Tasmania and mainland do not interact
  - Connected components of constraint graph
  - Suppose a graph of n variables can be broken into subproblems, each of only c variables:
    - Worst-case complexity 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

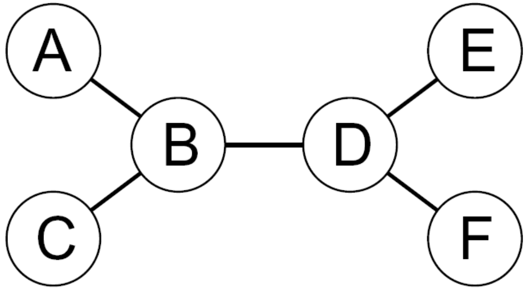
Tree-Structured CSPs

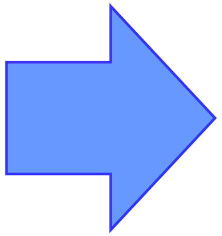


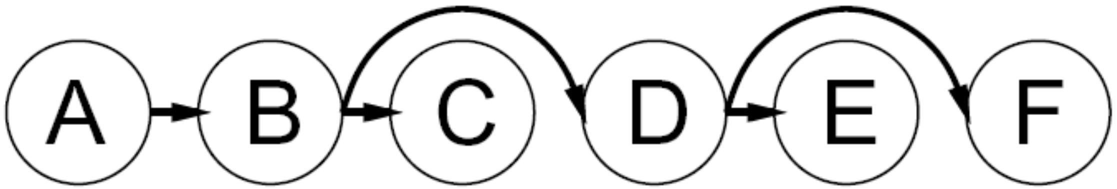
- Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$
  - How?
- This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning

Tree-Structured CSPs 2

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

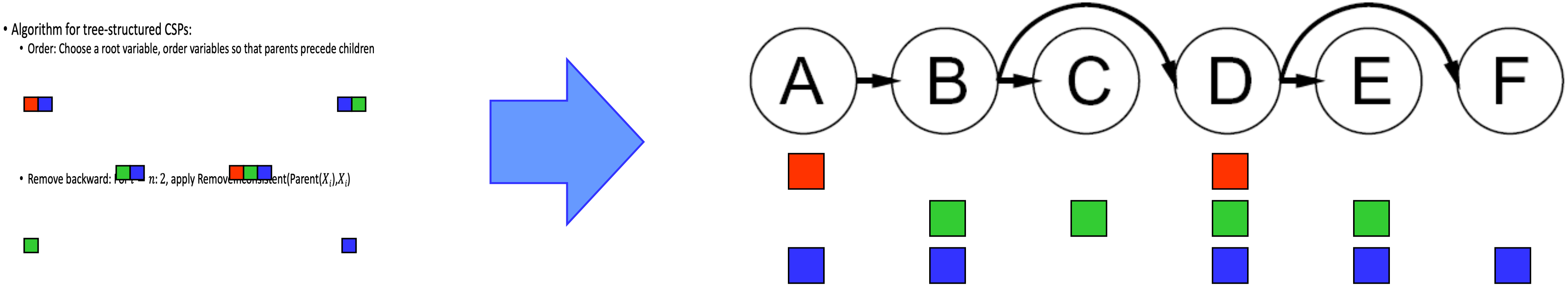






Tree-Structured CSPs 3

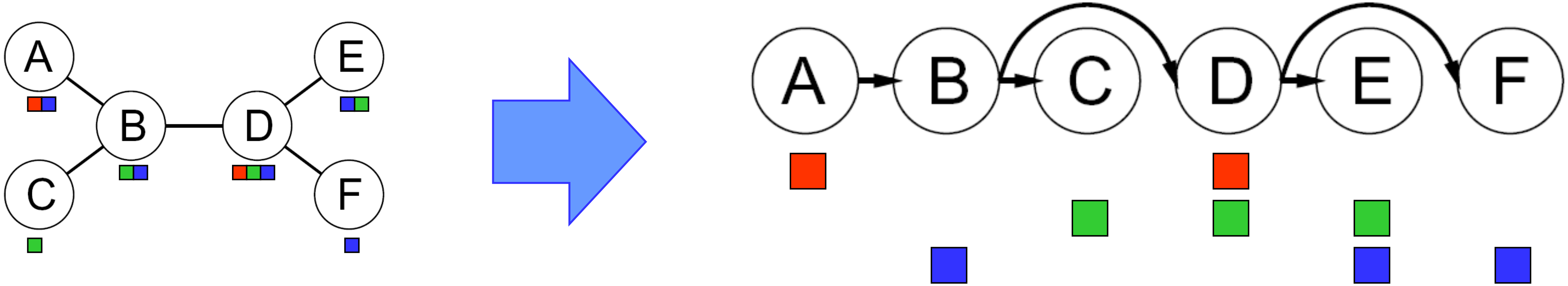
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For $i = n: 2$, apply RemoveInconsistent(Parent(X_i),X_i)

Tree-Structured CSPs 4

- 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)

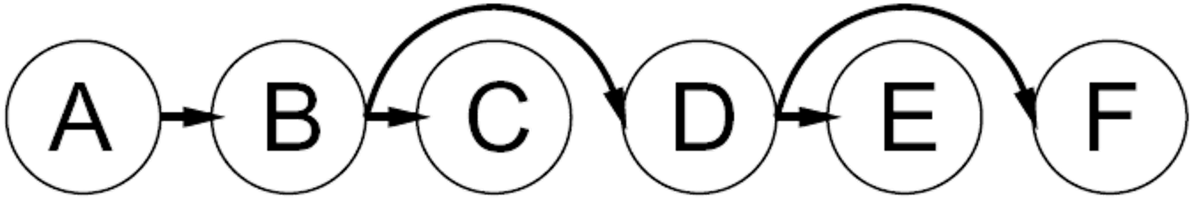
Remove backward $O(nd^2)$: $O(d^2)$ per arc and $O(n)$ arcs

Assign forward $O(nd)$: $O(d)$ per node and $O(n)$ nodes

- Runtime: $O(nd^2)$ (why?)
- Can always find a solution when there is one (why?)

Tree-Structured CSPs 5

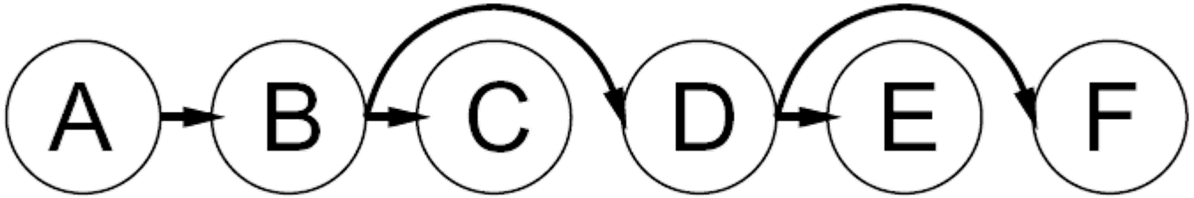
- Remove backward: For $i = n$: 2, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was "visited" once
  - a. Parent$(X_i) \rightarrow X_i$ was made consistent when X_i was visited
  - b. After that, Parent$(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass

Tree-Structured CSPs 6

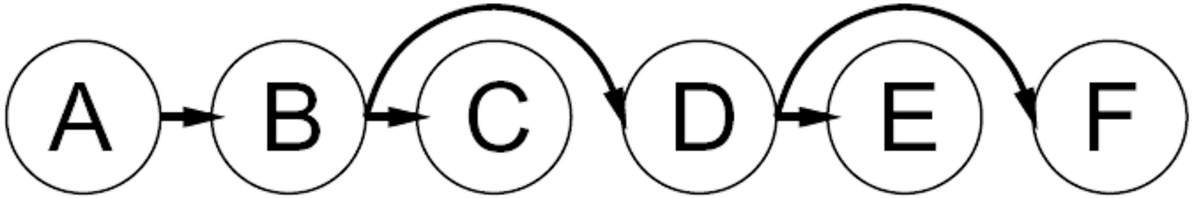
- Remove backward: For $i = n: 2$, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was “visited” once
  - a. Parent$(X_i) \rightarrow X_i$ was made consistent when X_i was visited
    - When X_i was visited, we enforced arc consistency of Parent$(X_i) \rightarrow X_i$ by reducing the domain of Parent(X_i). By definition, for every value in the reduced domain of Parent(X_i), there was some x in the domain of X_i which could be assigned without violating the constraint involving Parent(X_i) and X_i
  - b. After that, Parent$(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass

Tree-Structured CSPs 7

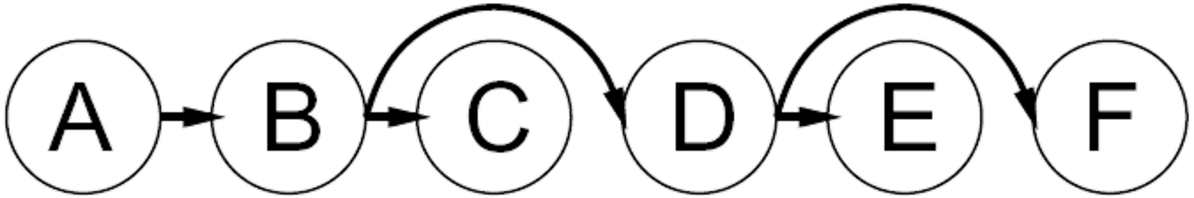
- Remove backward: For $i = n: 2$, apply RemoveInconsistent(Parent(X_i),X_i)



- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: During backward pass, every node except the root node was "visited" once.
  - a. Parent$(X_i) \rightarrow X_i$ was made consistent when X_i was visited
  - b. After that, Parent$(X_i) \rightarrow X_i$ kept consistent until the end of the backward pass
    - Domain of X_i would not have been reduced after X_i is visited because X_i's children were visited before X_i. Domain of Parent(X_i) could have been reduced further. Arc consistency would still hold by definition.

Tree-Structured CSPs 8

- Assign forward: For i=1:n, assign X_i consistently with Parent(X_i)



- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Follow the backtracking algorithm (on the reduced domains and with the same ordering). Induction on position Suppose we have successfully reached node X_i. In the current step, the potential failure can only be caused by the constraint between X_i and Parent(X_i), since all other variables that are in a same constraint of X_i have not assigned a value yet. Due to the arc consistency of Parent(X_i) $\rightarrow X_i$, there exists a value x in the domain of X_i that does not violate the constraint. So we can successfully assign value to X_i and go to the next node. By induction, we can successfully assign a value to a variable in each step of the algorithm. A solution is found in the end.

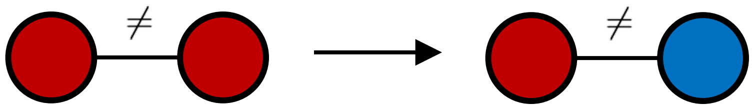
Local Search

Local Search

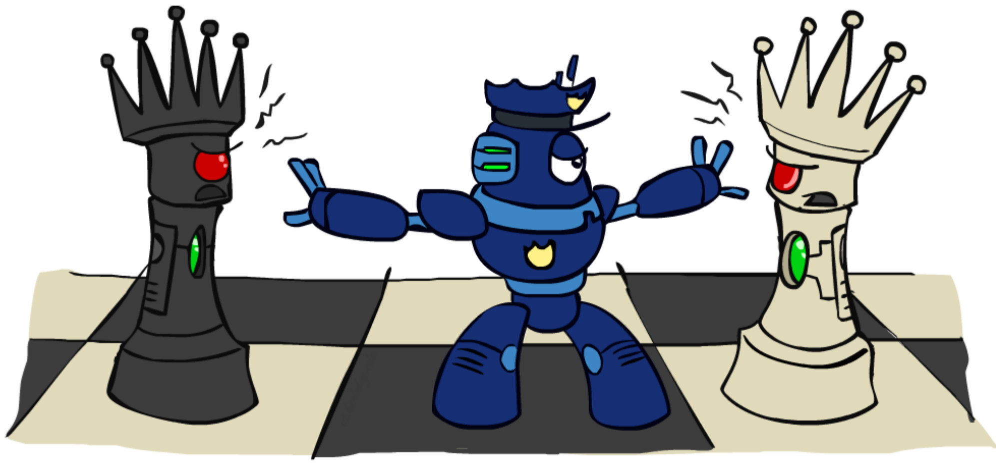
- Can be applied to identification problems (e.g., CSPs), as well as some planning and optimization problems
- Typically use a complete-state formulation
  - e.g., all variables assigned in a CSP (may not satisfy all the constraints)
- Different “complete”:
  - An assignment is complete means that all variables are assigned a value
  - An algorithm is complete means that it will output a solution if there exists one

Iterative Algorithms for CSPs

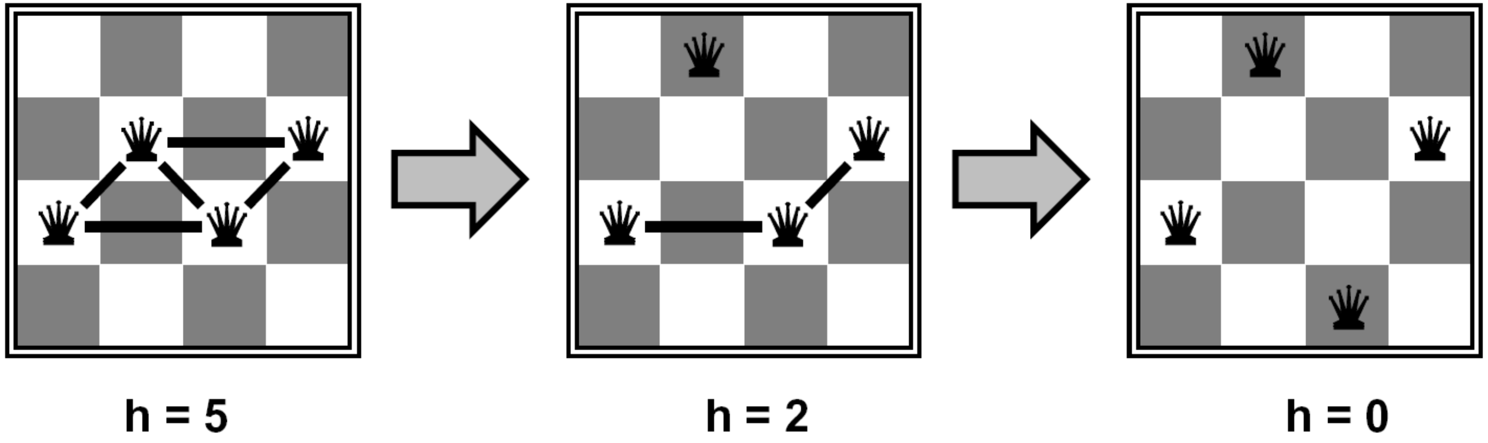
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



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



Example: 4-Queens

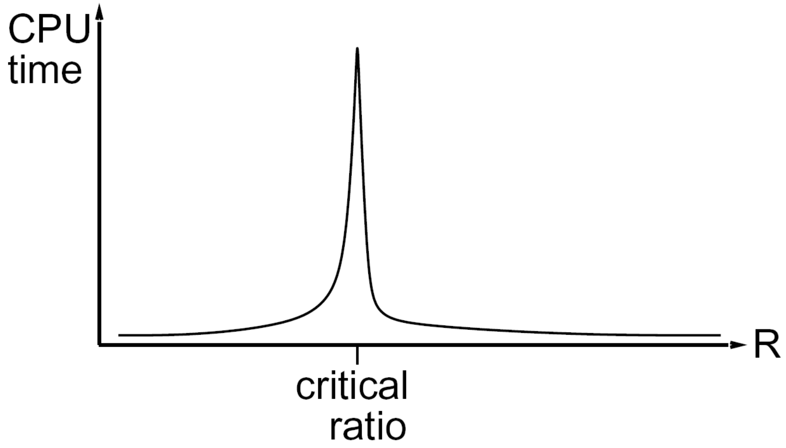


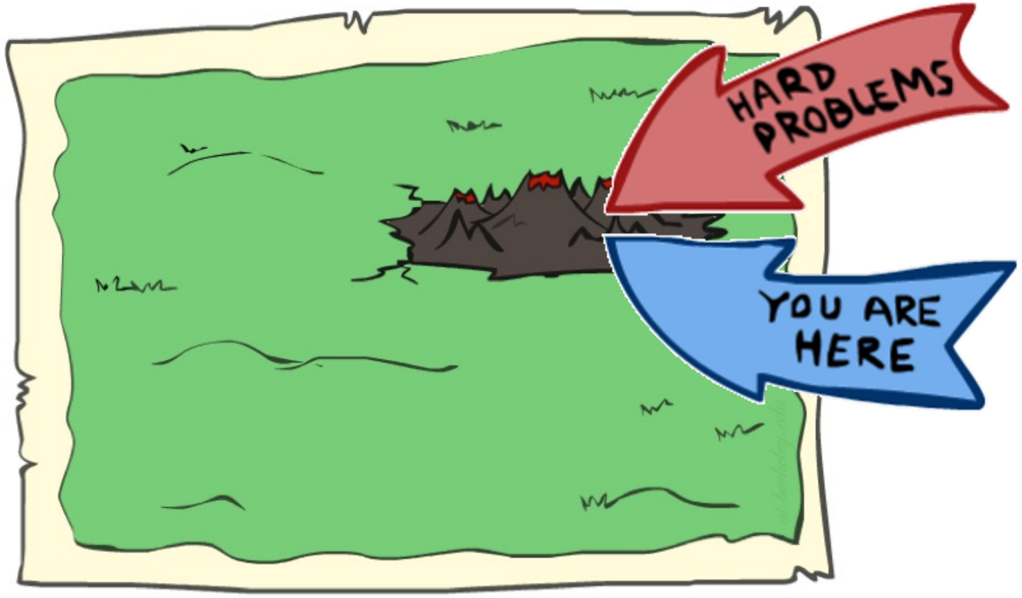
- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: h(n) = number of attacks

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

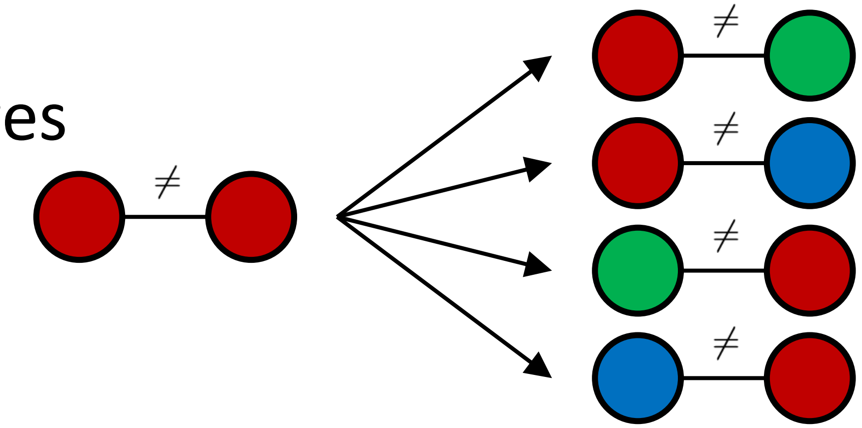
$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$





Local Search vs Tree 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

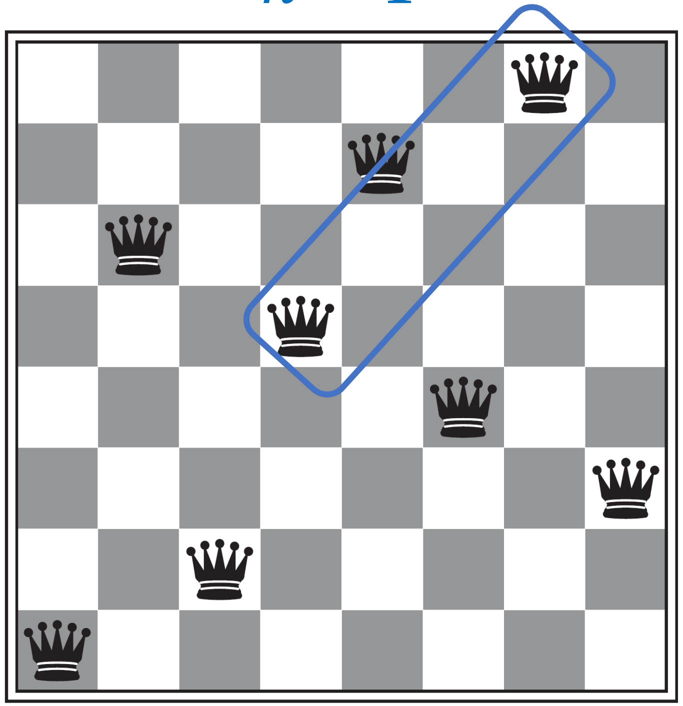


- Generally much faster and more memory efficient (but incomplete and suboptimal)

Example

- Local search may get stuck in a local optima

$h = 1$



Hill Climbing

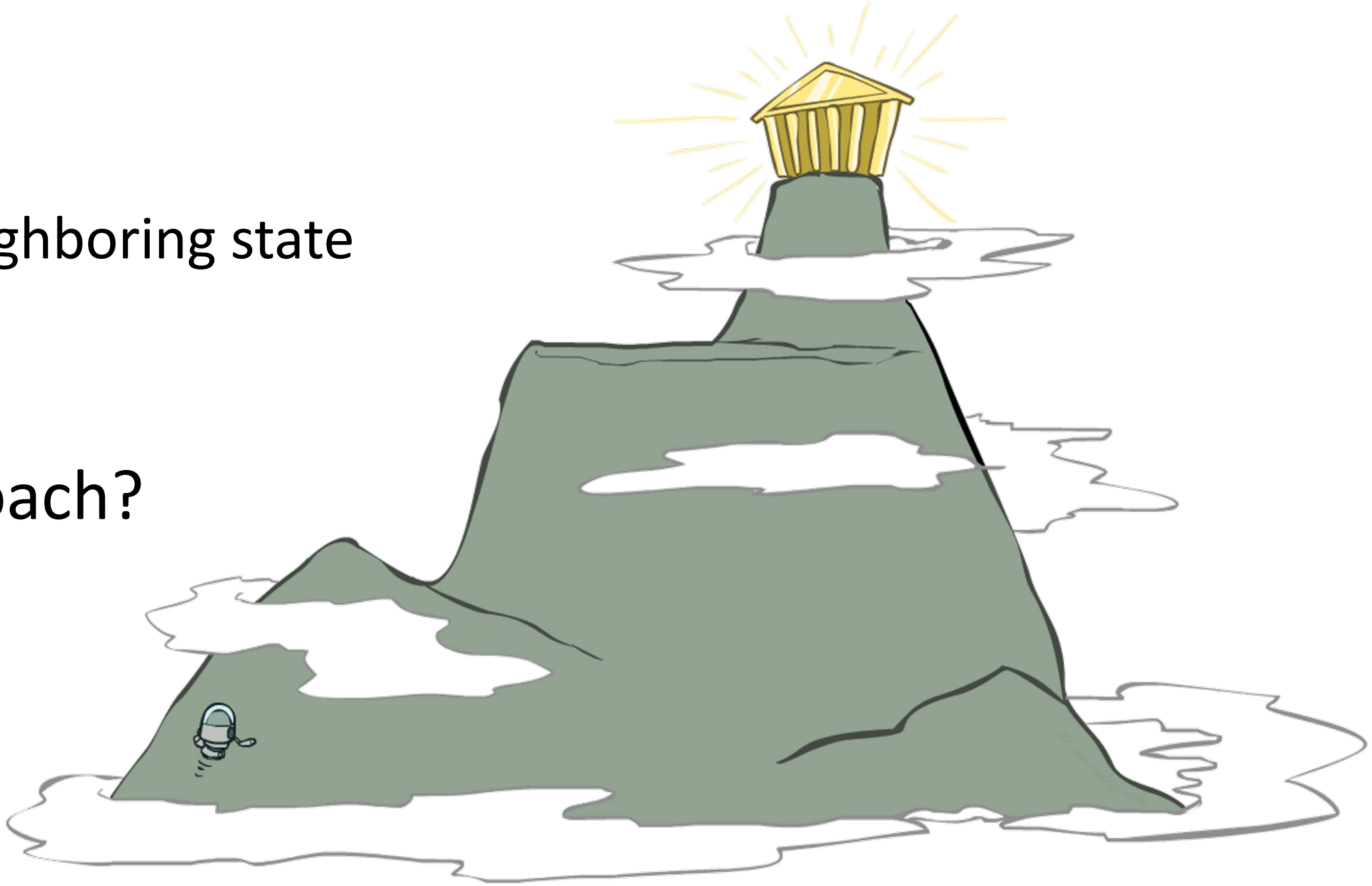
- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no for current, quit

- What's bad about this approach?

  Complete? No!

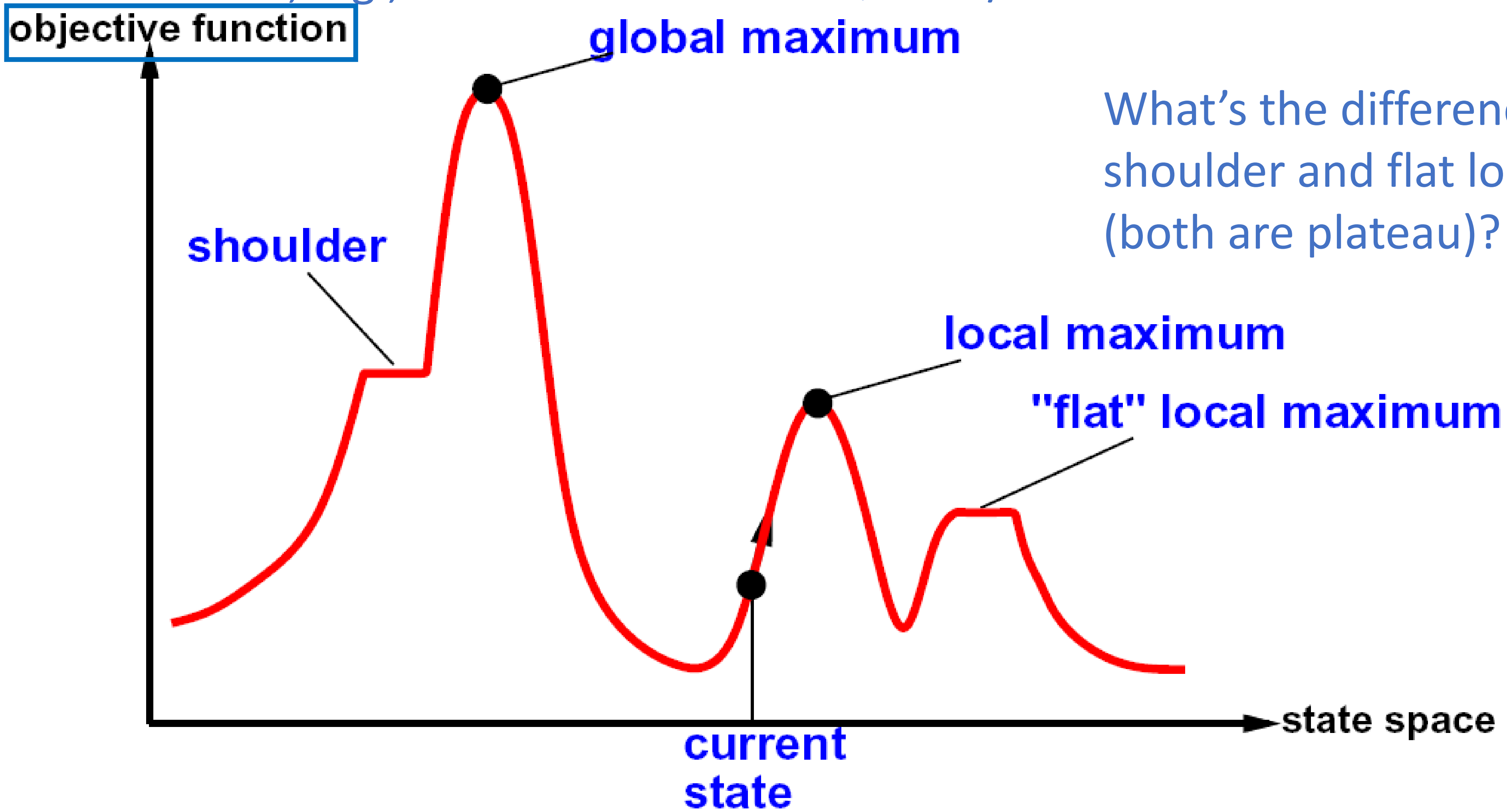
  Optimal? No!

- What's good about it?



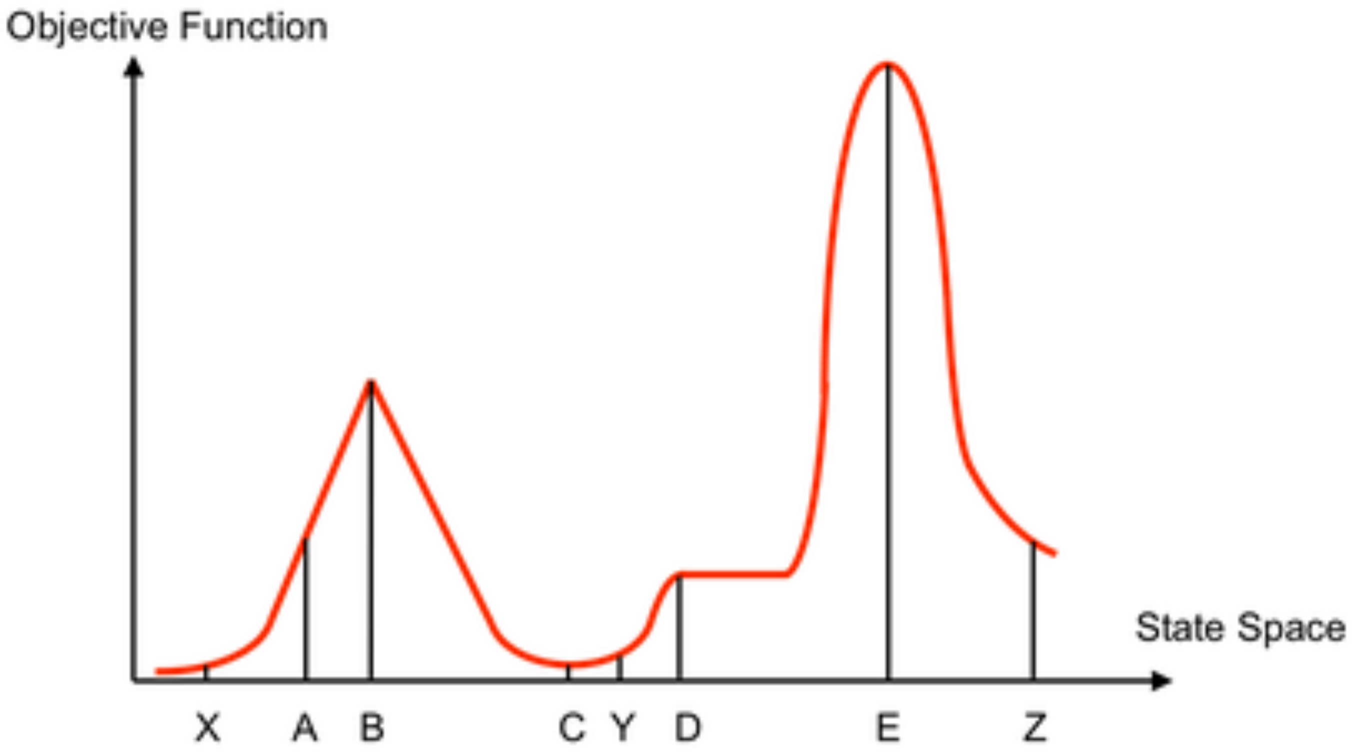
Hill Climbing Diagram

In identification problems, could be a function measuring how close you are to a valid solution, e.g., $-1\times$ #conflicts in n-Queens/CSP



What's the difference between shoulder and flat local maximum (both are plateau)?

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 ?

Hill Climbing (Greedy Local Search)

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of *current*
    if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
    current ← *neighbor*

How to apply Hill Climbing to n-Queens? How is it different from Iterative Improvement?

Define a state as a board with n queens on it, one in each column
Define a successor (neighbor) of a state as one that is generated by moving a single queen to another square in the same column

Hill Climbing (Greedy Local Search) 2

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
     neighbor ← a highest-valued successor of current
     if neighbor.VALUE ≤ current.VALUE then return current.STATE
     current ← neighbor
```

What if there is a tie?

Typically break ties randomly

What if we do not stop here?

- In 8-Queens, steepest-ascent hill climbing solves 14% of problem instances
  - Takes 4 steps on average when it succeeds, and 3 steps when it fails
- When allow for ≤100 consecutive sideway moves, solves 94% of problem instances
  - Takes 21 steps on average when it succeeds, and 64 steps when it fails

Local Search: Summary

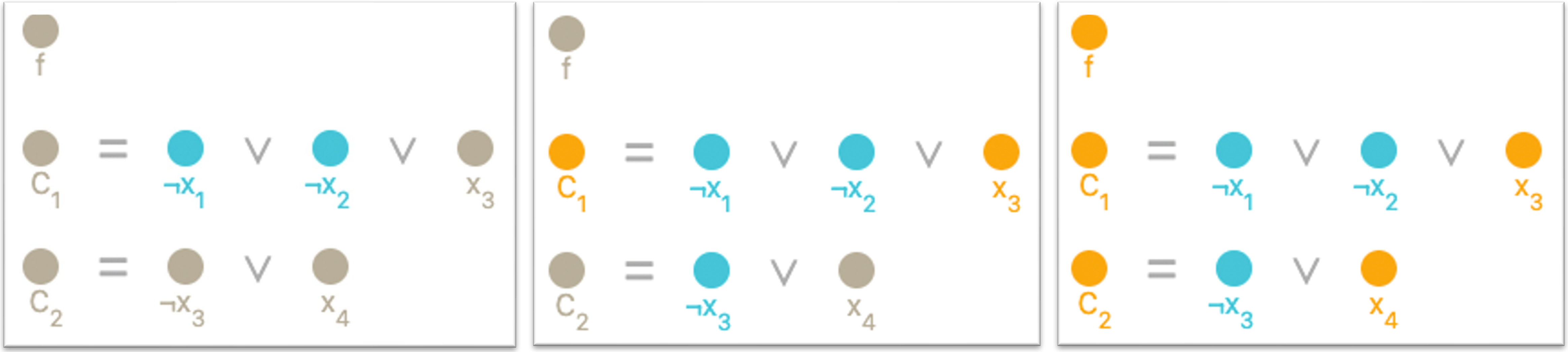
- Maintain a constant number of current nodes or states, and move to "neighbors" or generate "offsprings" in each iteration
  - Do not maintain a search tree or multiple paths
  - Typically do not retain the path to the node

- Advantages
  - Use little memory
  - Can potentially solve large-scale problems or get a reasonable (suboptimal or almost feasible) solution

Boolean Satisfiability Problem

Boolean Constraint Propagation (BCP)

- Unit clause: A clause is unit under a partial assignment when that assignment makes every literal in the clause unsatisfied but leaves a single literal undecided
- Example: f = (¬x1 ∨ ¬x2 ∨ x3) ∧ (¬x3 ∨ x4), guess x1 and x2 are true



Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

- A SAT solver: recursive backtracking + BCP

- DPLL:
  - Run BCP on the formula
  - If the formula evaluates to True, return True
  - If the formula evaluates to False, return False
  - If the formula is still Undecided:
    - Choose the next unassigned variable
    - Return (DPLL with that variable True) || (DPLL with that variable False)

- Demo

Shortcomings of DPLL

- DPLL:
  - Run BCP on the formula
  - If the formula evaluates to True, return True
  - If the formula evaluates to False, return False
  - If the formula is still Undecided:
    - Choose the next unassigned variable
    - Return (DPLL with that variable True) || (DPLL with that variable False)

No learning: throws away all the work performed to conclude that the current partial assignment (PA) is bad. Revisits bad PAs that lead to conflict due to the same root cause

Naive decisions: picks an arbitrary variable to branch on. Fails to consider the state of the search to make heuristically better decisions

Chronological backtracking: backtracks one level, even if it can be deduced that the current partial assignment became doomed at a lower level

Conflict Driven Clause Learning (CDCL)

- CDCL improves on all three aspects!

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

Decision heuristics: choose the next literal to add to the current partial assignment based on the state of the search

Learning: F augmented with a conflict clause that summarizes the root cause of the conflict

Non-chronological backtracking: backtracks b levels, based on the cause of the conflict

CDCL by example

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

…

…

CDCL by example 2

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

...

...

CDCL by example 3

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \vee x_2 \vee \neg x_4$

$c_2 : \neg x_1 \vee \neg x_2 \vee x_3$

$c_3 : \neg x_3 \vee \neg x_4$

$c_4 : x_4 \vee x_5 \vee x_6$

$c_5 : \neg x_5 \vee x_7$

$c_6 : \neg x_6 \vee x_7 \vee \neg x_8$

…

…

CDCL by example 4

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

...

...

CDCL by example 5

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

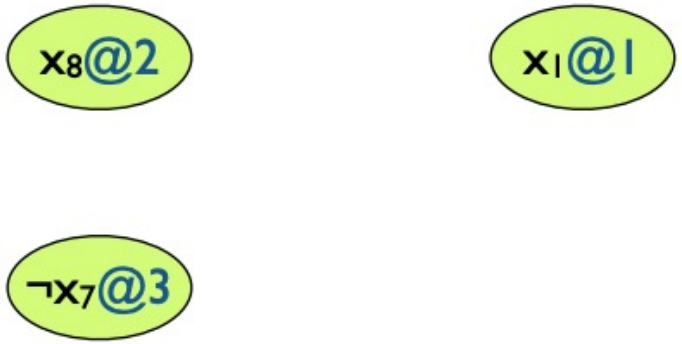
$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

…

…



CDCL by example 6

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

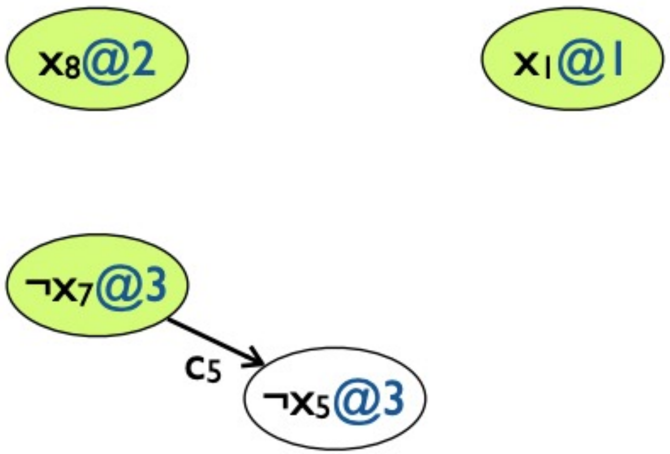
$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

…

…



CDCL by example 7

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \vee x_2 \vee \neg x_4$

$c_2 : \neg x_1 \vee \neg x_2 \vee x_3$

$c_3 : \neg x_3 \vee \neg x_4$

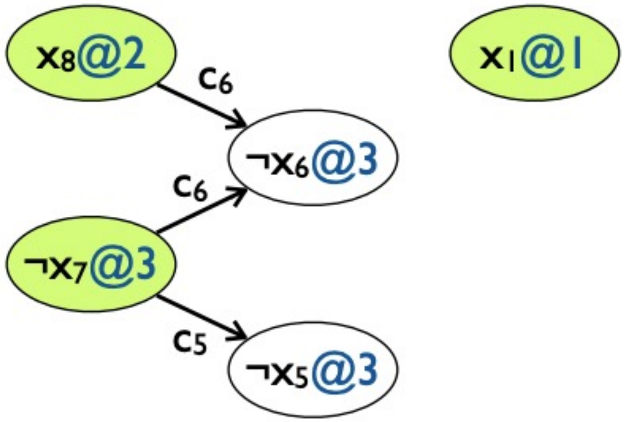
$c_4 : x_4 \vee x_5 \vee x_6$

$c_5 : \neg x_5 \vee x_7$

$c_6 : \neg x_6 \vee x_7 \vee \neg x_8$

…

…



CDCL by example 8

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

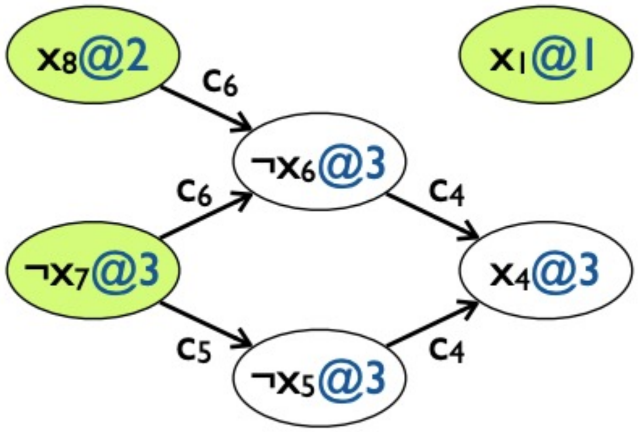
$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

…

…



CDCL by example 9

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \vee x_2 \vee \neg x_4$

$c_2 : \neg x_1 \vee \neg x_2 \vee x_3$

$c_3 : \neg x_3 \vee \neg x_4$

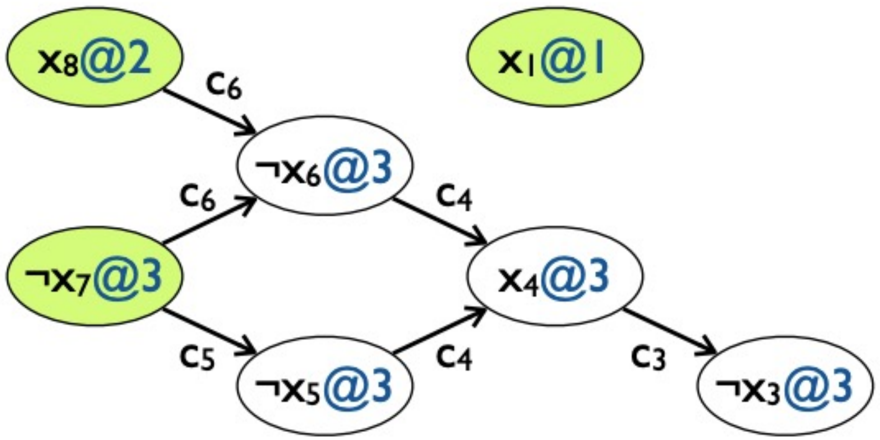
$c_4 : x_4 \vee x_5 \vee x_6$

$c_5 : \neg x_5 \vee x_7$

$c_6 : \neg x_6 \vee x_7 \vee \neg x_8$

…

…



CDCL by example 10

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \vee x_2 \vee \neg x_4$

$c_2 : \neg x_1 \vee \neg x_2 \vee x_3$

$c_3 : \neg x_3 \vee \neg x_4$

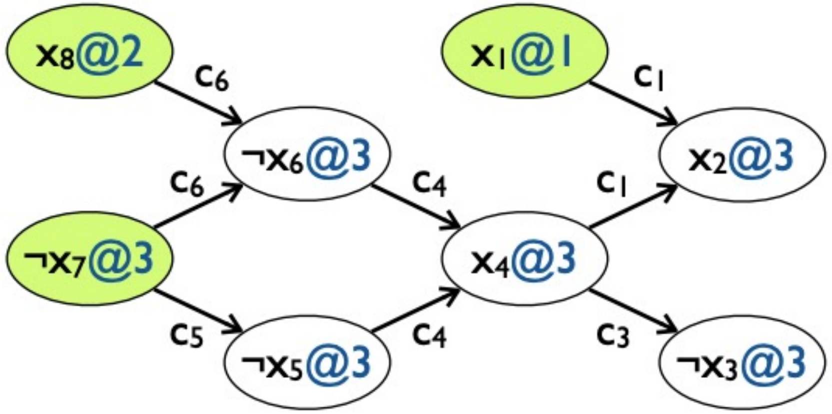
$c_4 : x_4 \vee x_5 \vee x_6$

$c_5 : \neg x_5 \vee x_7$

$c_6 : \neg x_6 \vee x_7 \vee \neg x_8$

…

…



CDCL by example 11

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

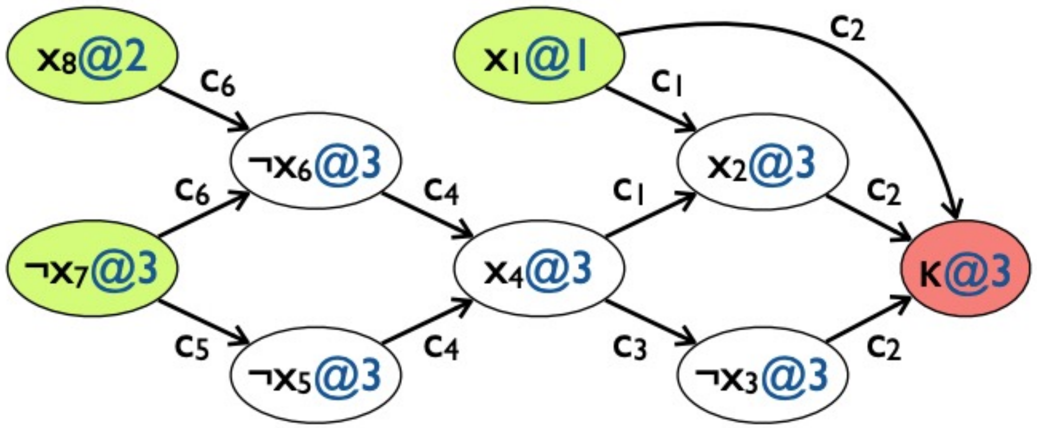
$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

…

…



CDCL by example 12

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

(1,-x1 v -x4)

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \vee x_2 \vee \neg x_4$

$c_2 : \neg x_1 \vee \neg x_2 \vee x_3$

$c_3 : \neg x_3 \vee \neg x_4$

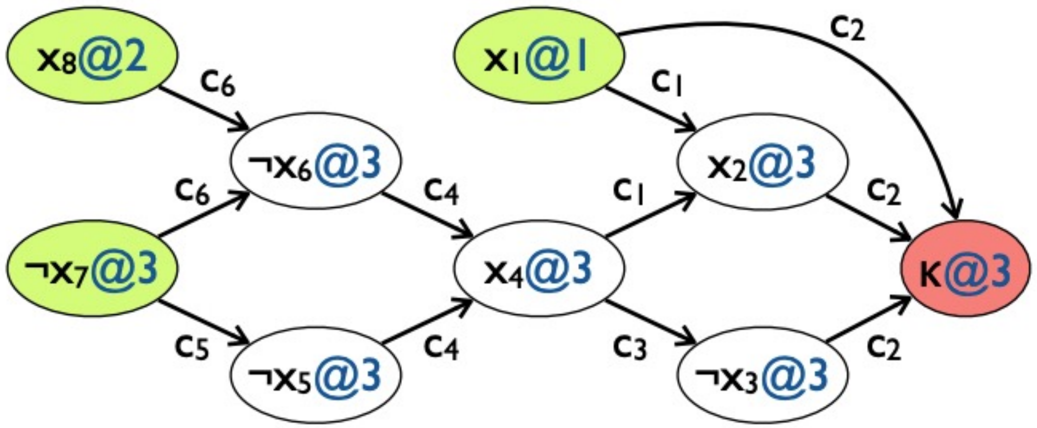
$c_4 : x_4 \vee x_5 \vee x_6$

$c_5 : \neg x_5 \vee x_7$

$c_6 : \neg x_6 \vee x_7 \vee \neg x_8$

…

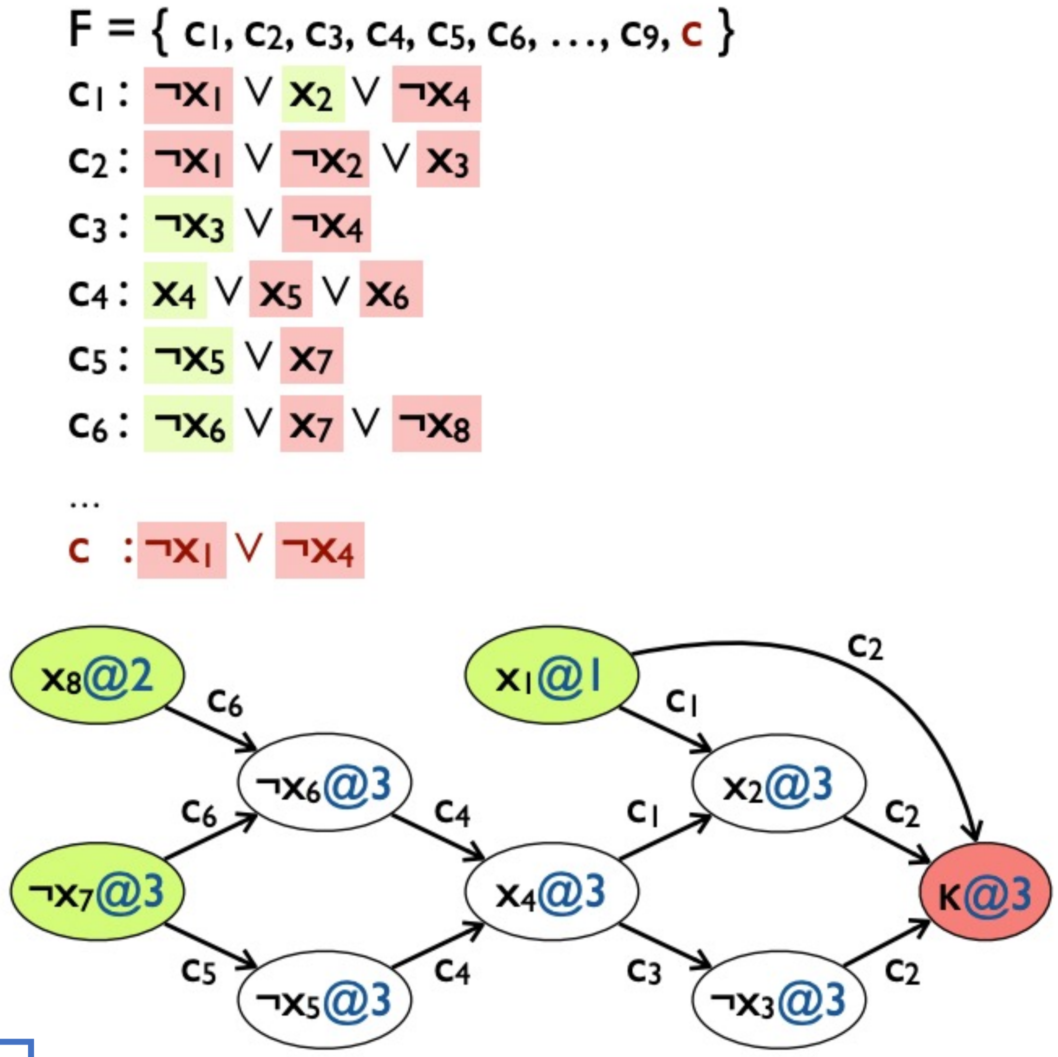
…



CDCL by example 13

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

(1,-x1 v -x4)



CDCL by example 14

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

(1,-x1 v -x4)

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9, c \}$

$c_1 : \neg x_1 \lor x_2 \lor \neg x_4$

$c_2 : \neg x_1 \lor \neg x_2 \lor x_3$

$c_3 : \neg x_3 \lor \neg x_4$

$c_4 : x_4 \lor x_5 \lor x_6$

$c_5 : \neg x_5 \lor x_7$

$c_6 : \neg x_6 \lor x_7 \lor \neg x_8$

...

$c : \neg x_1 \lor \neg x_4$

x_1@1

CDCL by example 14

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

(1,-x1 v -x4)

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9, c \}$

$c_1 : \neg x_1 \vee x_2 \vee \neg x_4$

$c_2 : \neg x_1 \vee \neg x_2 \vee x_3$

$c_3 : \neg x_3 \vee \neg x_4$

$c_4 : x_4 \vee x_5 \vee x_6$

$c_5 : \neg x_5 \vee x_7$

$c_6 : \neg x_6 \vee x_7 \vee \neg x_8$

…

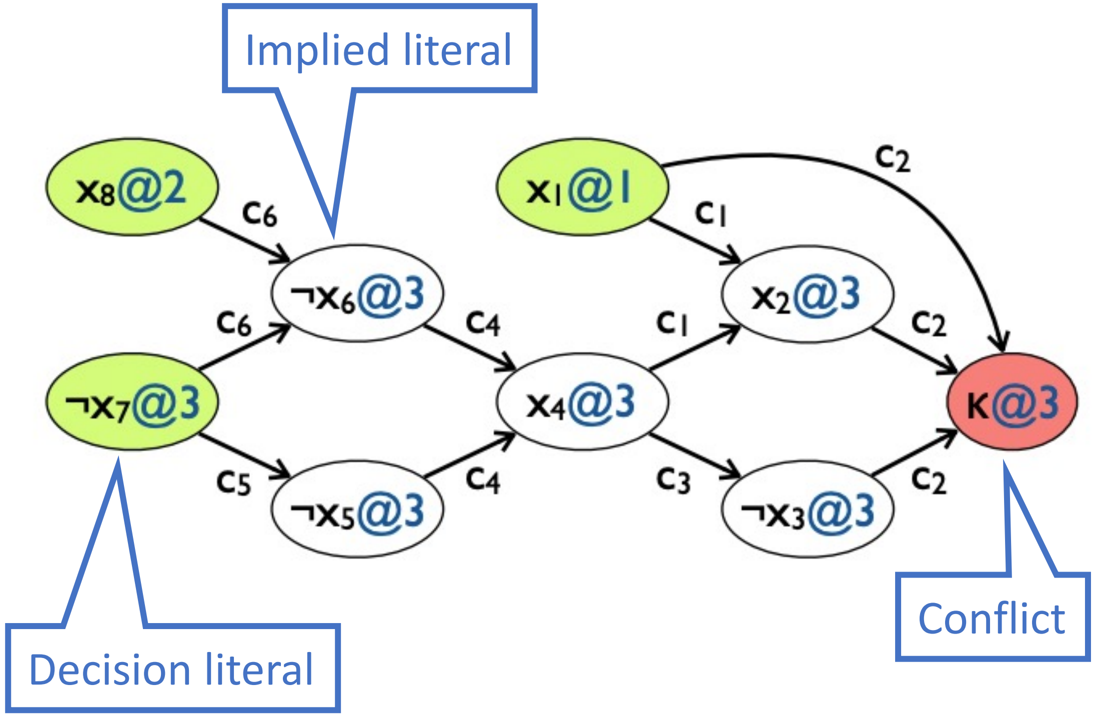
$c : \neg x_1 \vee \neg x_4$

Conflict clause is unit after backtracking

x_1@1

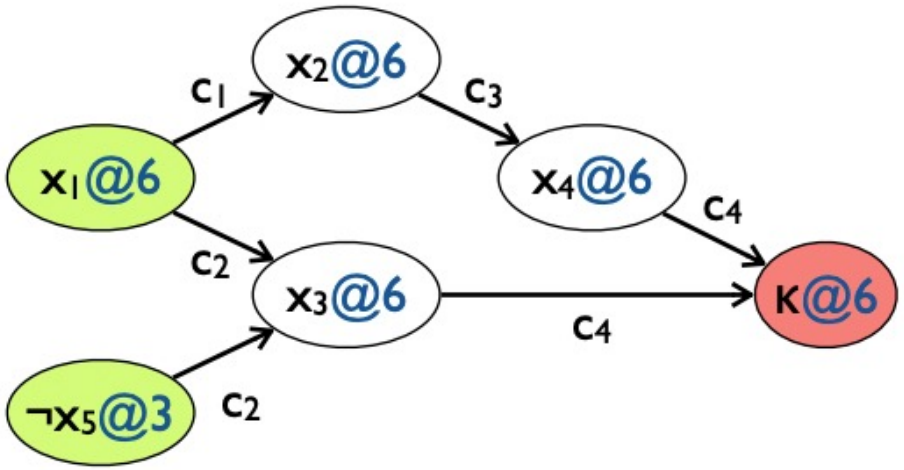
Implication graph

- An implication graph G = (V, E) is a DAG that records the history of decisions and the resulting deductions derived with BCP
    - v ∈ V is a literal (or κ) and the decision level at which it entered the current partial assignment (PA)
    - ⟨v, w⟩ ∈ E iff v ≠ w, ¬v ∈ antecedent(w), and ⟨v, w⟩ is labeled with antecedent(w)
- A unit clause c is an antecedent of its sole unassigned literal



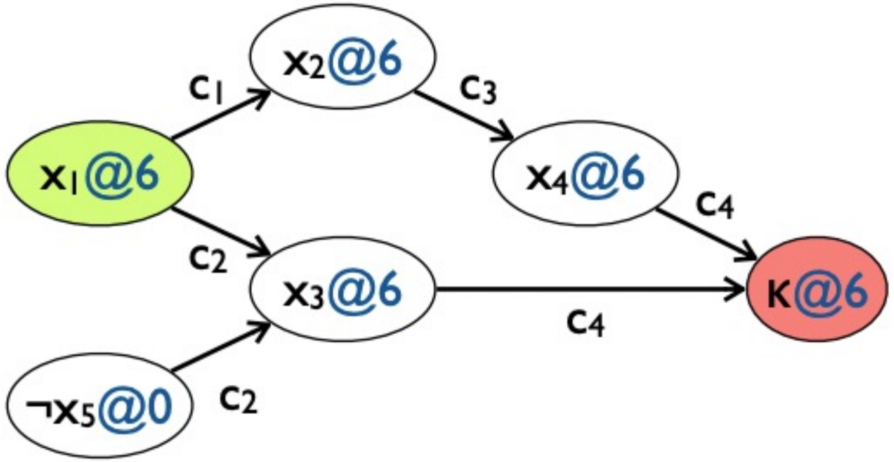
Quiz a

- What clauses gave rise to this implication graph?
- c1 :
- c2 :
- c3 :
- c4 :



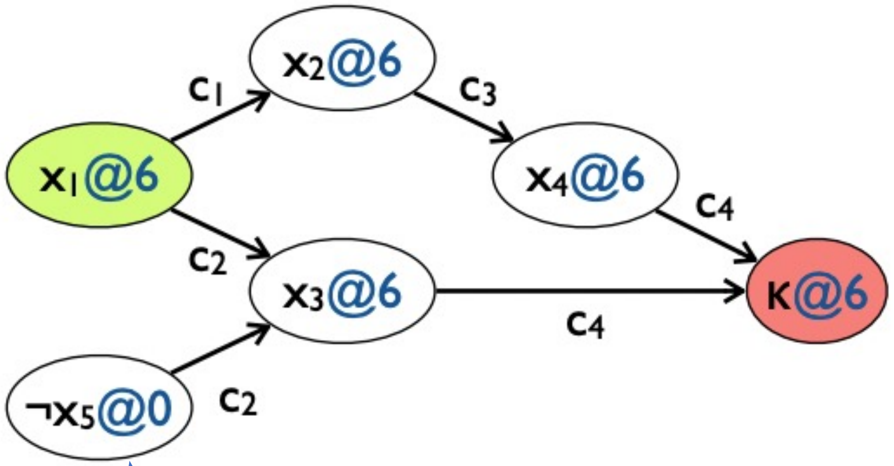
Quiz b

- What clauses gave rise to this implication graph?
- c1 :
- c2 :
- c3 :
- c4 :



Quiz b-2

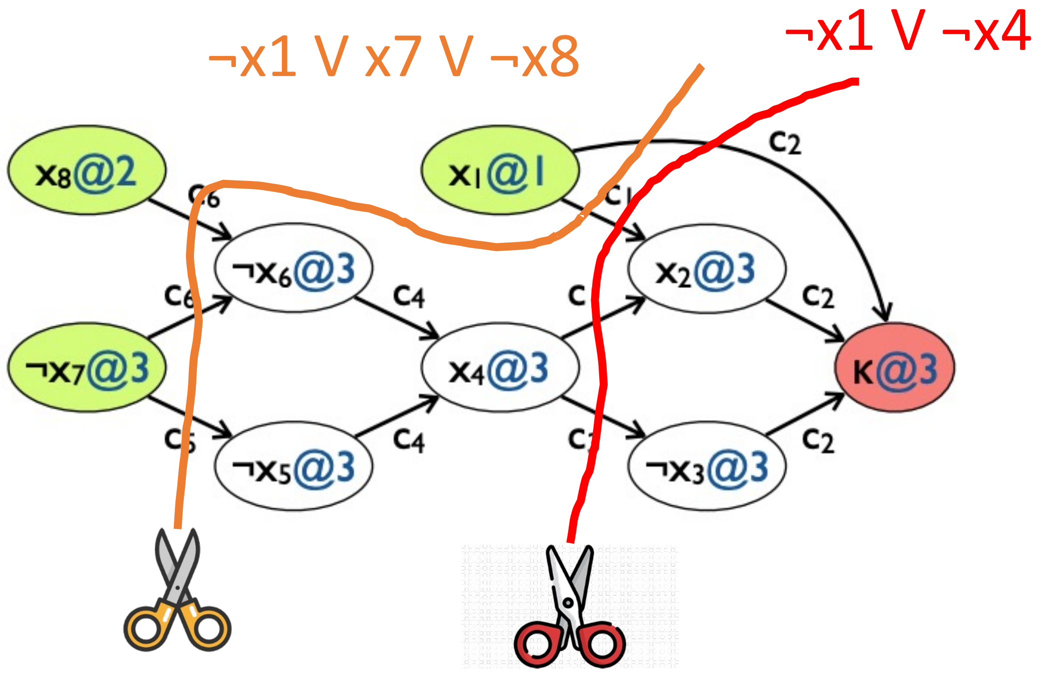
- What clauses gave rise to this implication graph?
- c1 :
- c2 :
- c3 :
- c4 :
- c5: ¬x5



Assignments at ground (0) level are implied by unary clauses

How to learn a conflict clause?

```
CDCL(F)
 A ← {}
 if BCP(F,A) = conflict then return false
 level ← 0
 while hasUnassignedVars(F)
   level ← level + 1
   A ← A ∪ { DECIDE(F,A) }
   while BCP(F,A) = conflict
     ⟨b, c⟩ ← ANALYZECONFLICT()
     F ← F ∪ {c}
     if b < 0 then return false
     else BACKTRACK(F,A, b)
          level ← b
 return true
```

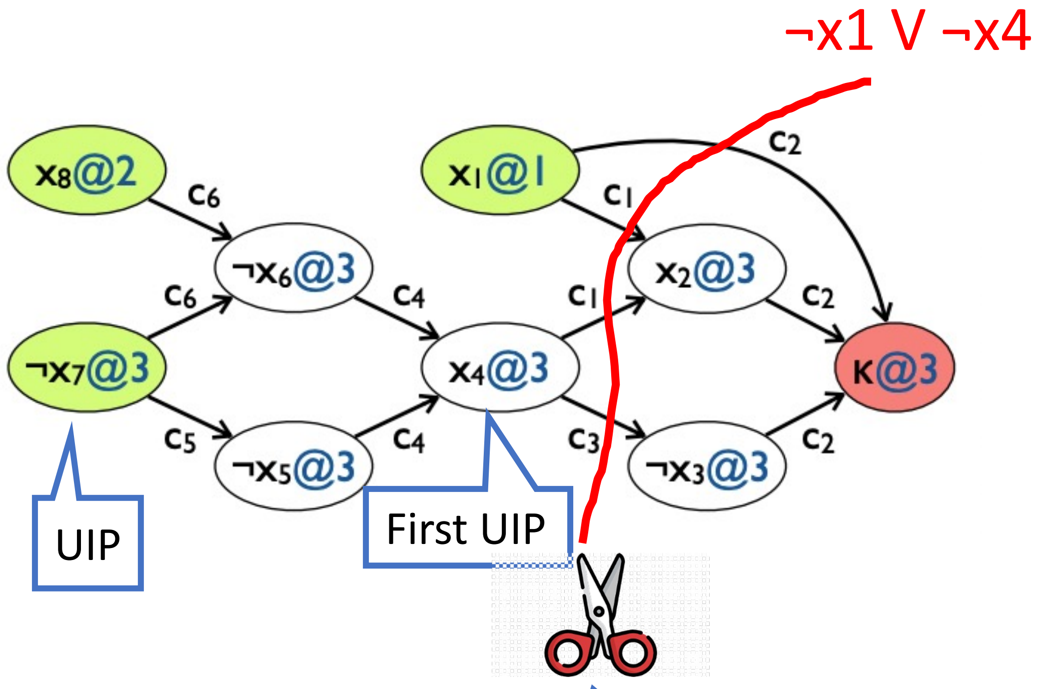


- A conflict clause is implied by F and it blocks PAs that lead to the current conflict
- Every cut that separates sources from the sink defines a valid conflict clause

Unique implication points (UIPs)

- A UIP is any node in the implication graph other than the conflict that is on all paths from the current decision literal (lit@d) to the conflict (κ@d)

- A first UIP is the UIP that is closest to the conflict



Cut after the first unique implication point to get the shortest conflict clause

ANALYZECONFLICT: Computing the conflict clause

$F = \{ c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_9 \}$

$c_1 : \neg x_1 \vee x_2 \vee \neg x_4$

$c_2 : \neg x_1 \vee \neg x_2 \vee x_3$

$c_3 : \neg x_3 \vee \neg x_4$

$c_4 : x_4 \vee x_5 \vee x_6$

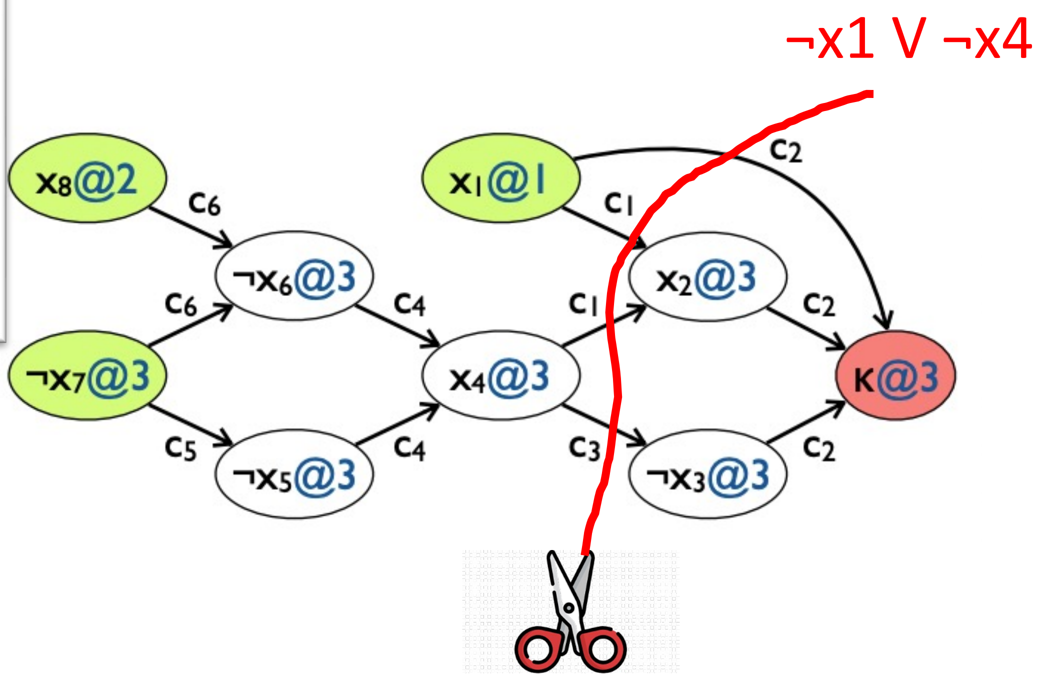
$c_5 : \neg x_5 \vee x_7$

$c_6 : \neg x_6 \vee x_7 \vee \neg x_8$

…

…

- ANALYZECONFLICT()
  - d ← level(conflict)
  - if d = 0 then return -1
  - c ← antecedent(conflict)
  - repeat
    - t ← lastAssignedLitAtLevel(c, d)
    - v ← varOfLit(t)
    - ante ← antecedent(t)
    - c ← resolve(ante, c, v)
  - until oneLitAtLevel(c, d)
  - b ←…
  - return ⟨b, c⟩



Binary resolution rule

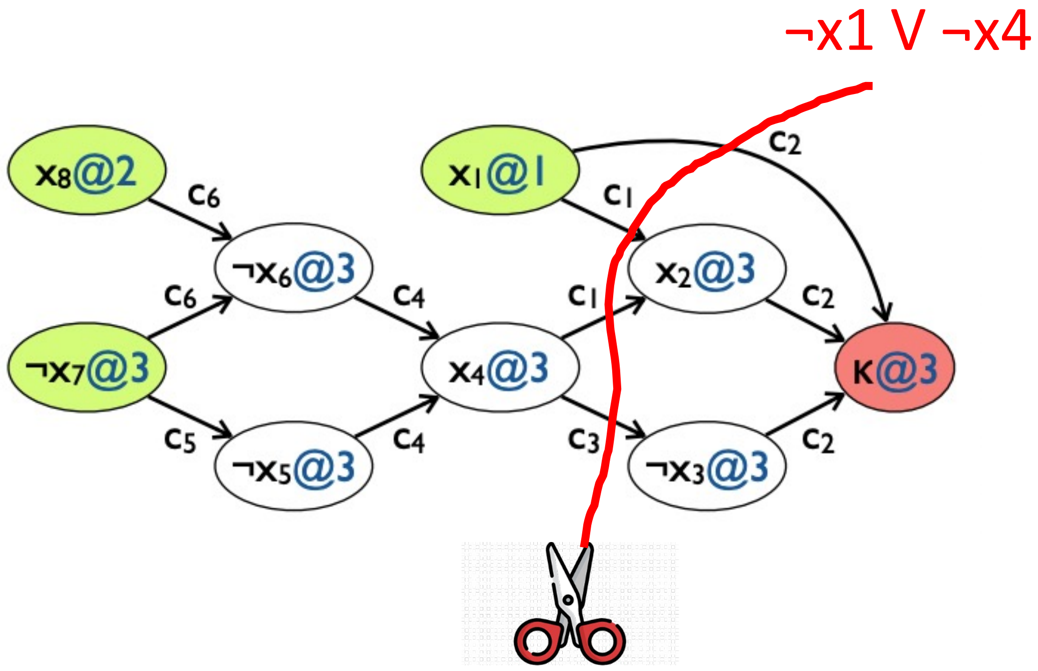
$$\frac{A \vee B, \ \neg B \vee C}{A \vee C}$$

Resolution is a basic operation in the propositional logic. To satisfy both $A \vee B$ and $\neg B \vee C$, we must satisfy $A \vee C$

Example:
- c = c2, t = x2, v = x2, ante = c1
- c = ¬x1 V x3 V ¬x4, t = x3, v = x3, ante = c3
- c = ¬x1 V ¬x4, done!

ANALYZECONFLICT: Computing the conflict clause 2

- ANALYZECONFLICT()
  - d ← level(conflict)
  - if d = 0 then return -1
  - c ← antecedent(conflict)
  - repeat
    - t ← lastAssignedLitAtLevel(c, d)
    - v ← varOfLit(t)
    - ante ← antecedent(t)
    - c ← resolve(ante, c, v)
  - until oneLitAtLevel(c, d)
  - b ← assertingLevel(c)
  - return ⟨b, c⟩



Second highest decision level for any literal in c, unless c is unary. In that case, its asserting level is zero

By construction, c is unit at b (since it has only one literal at the current level d)

Decision heuristics

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

Dynamic Largest Individual Sum (DLIS)

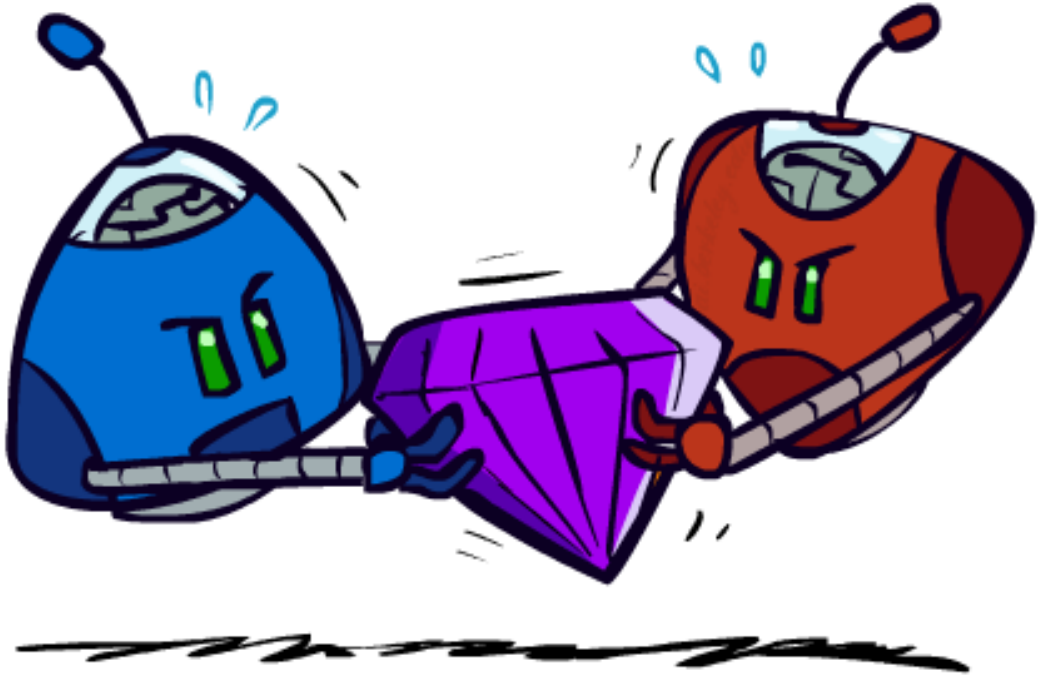
- Choose the literal that satisfies the most unresolved clauses
  - Let cnt(l) = number of occurrences of literal l in unsatisfied clauses
  - Set the l with highest cnt(l)
- Simple and intuitive
- But expensive:
  - complexity of making a decision proportional to the number of clauses

Decision heuristics 2

- CDCL(F):
  - A ← {}
  - if BCP(F, A) = conflict then return false
  - level ← 0
  - while hasUnassignedVars(F)
    - level ← level + 1
    - A ← A ∪ { DECIDE(F, A) }
    - while BCP(F, A) = conflict
      - ⟨b, c⟩ ← ANALYZECONFLICT()
      - F ← F ∪ {c}
      - if b < 0 then return false
        else BACKTRACK(F, A, b)
        level ← b
  - return true

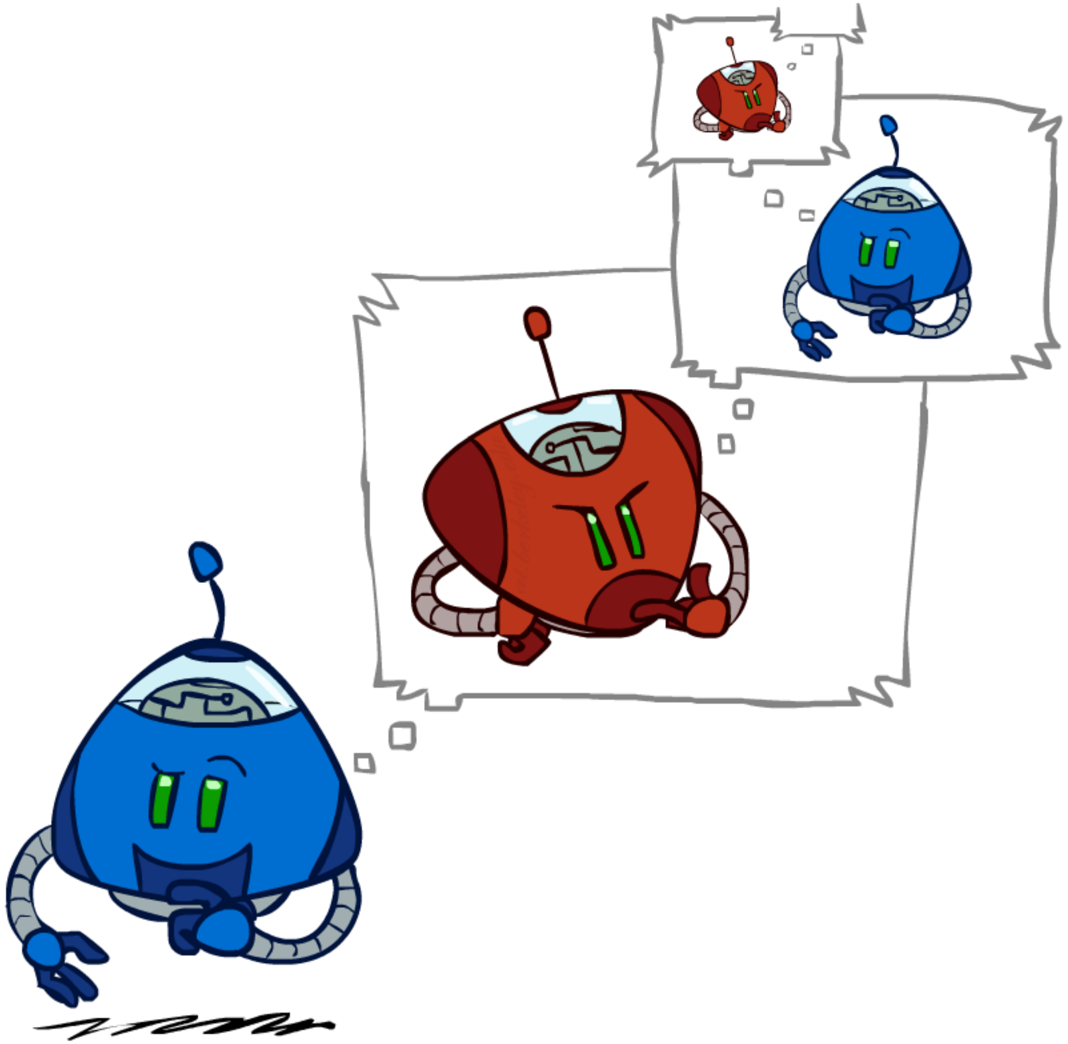
Variable State Independent Decaying Sum (VSIDS)

- Count the number of all clauses in which a literal appears, and periodically divide all scores by a constant (e.g., 2)
  - For each literal l, maintain a VSIDS score
  - Initially: set to cnt(l)
  - Increment score by 1 each time it appears in an added (conflict) clause
  - Divide all scores by a constant (say 2) periodically (say every N backtracks)
- Variables involved in more recent conflicts get higher scores
- Constant decision time when literals kept in a sorted list



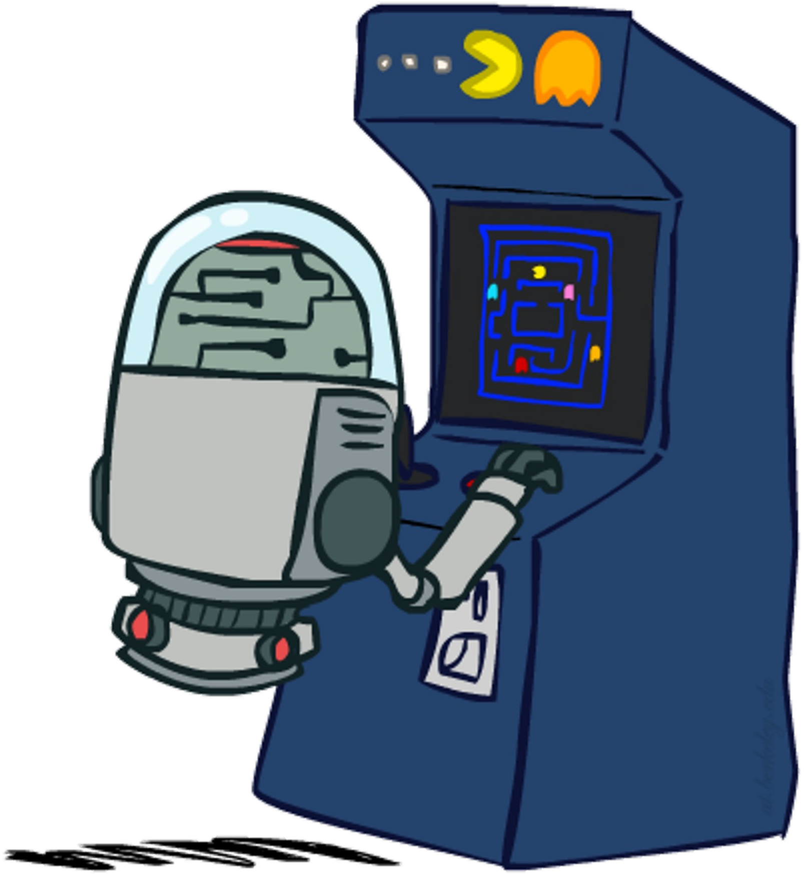
Adversarial Search

Cost -> Utility!



“Standard” Games

- Standard games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
  - 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



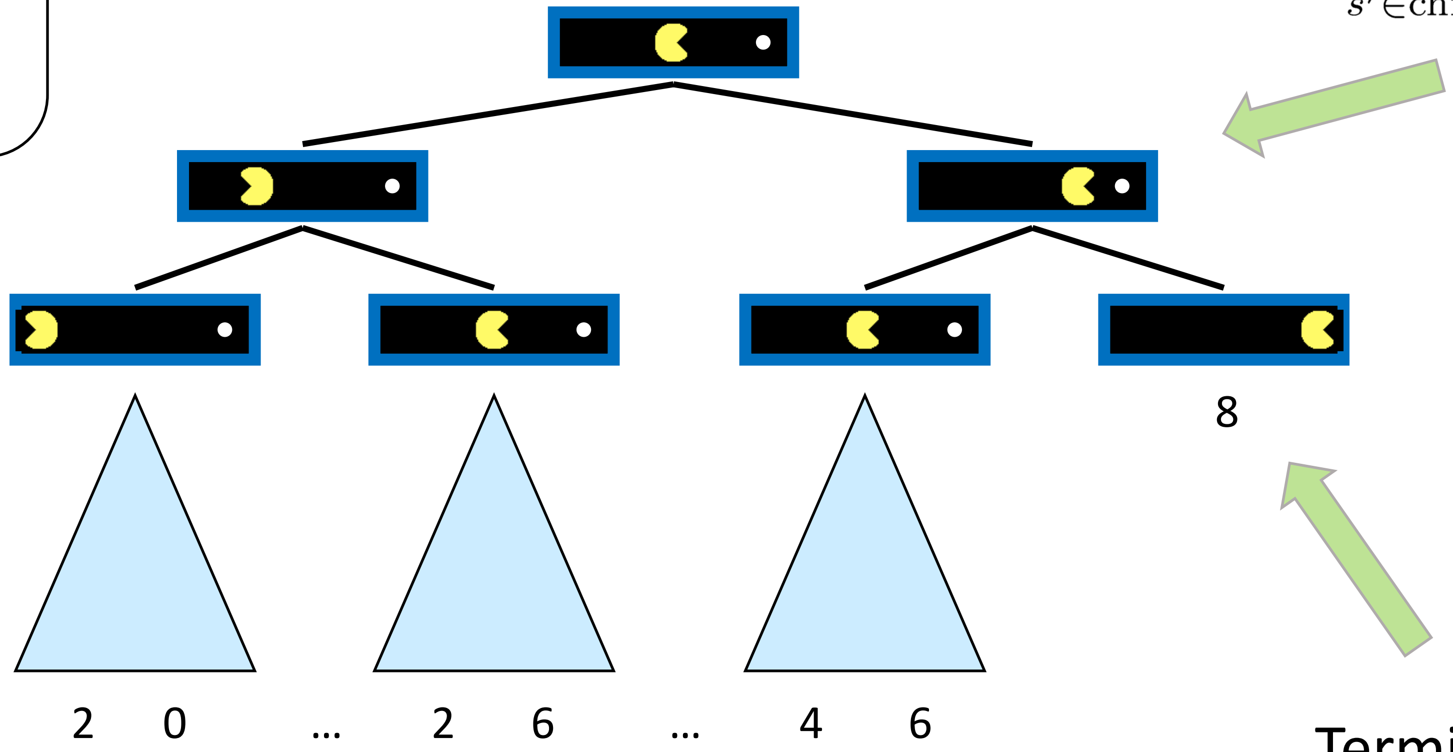
- Solution for a player is a policy: S $\rightarrow$ A

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:

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

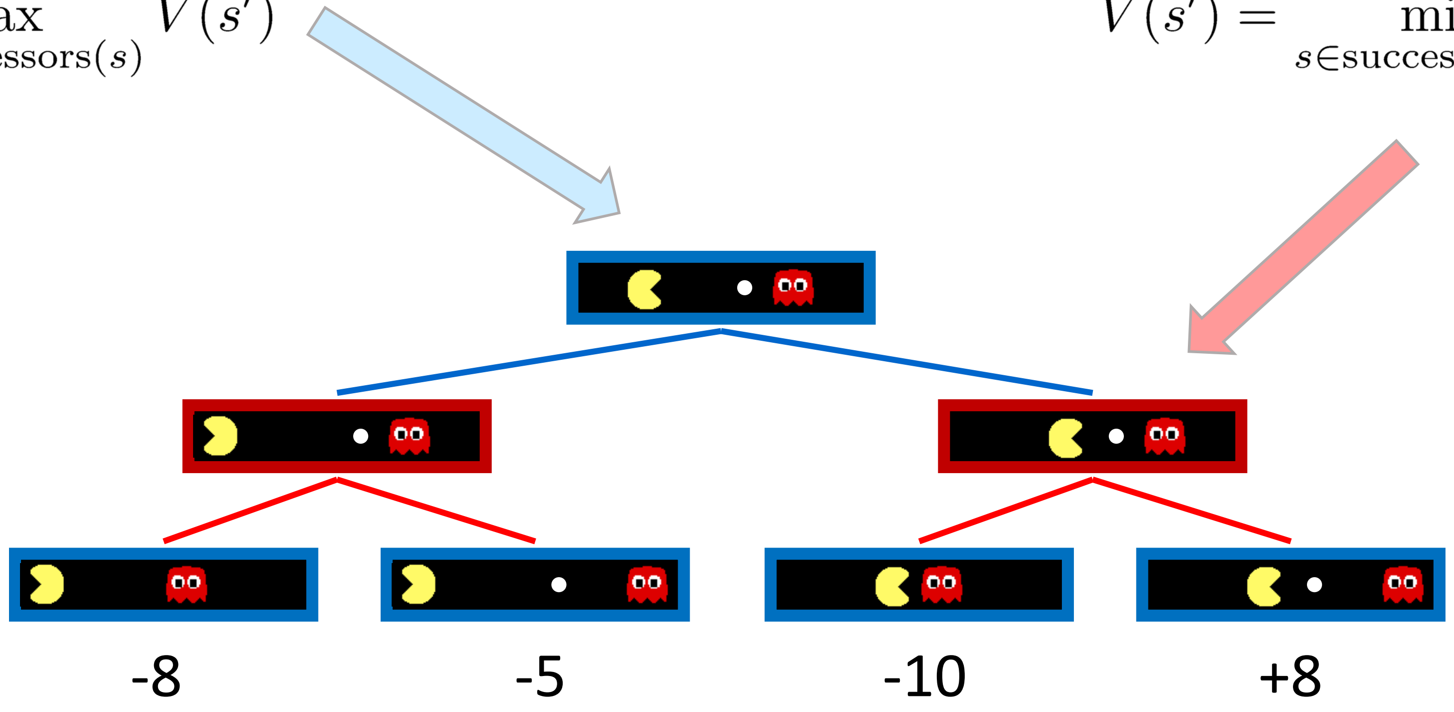
Adversarial Game Trees: Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

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



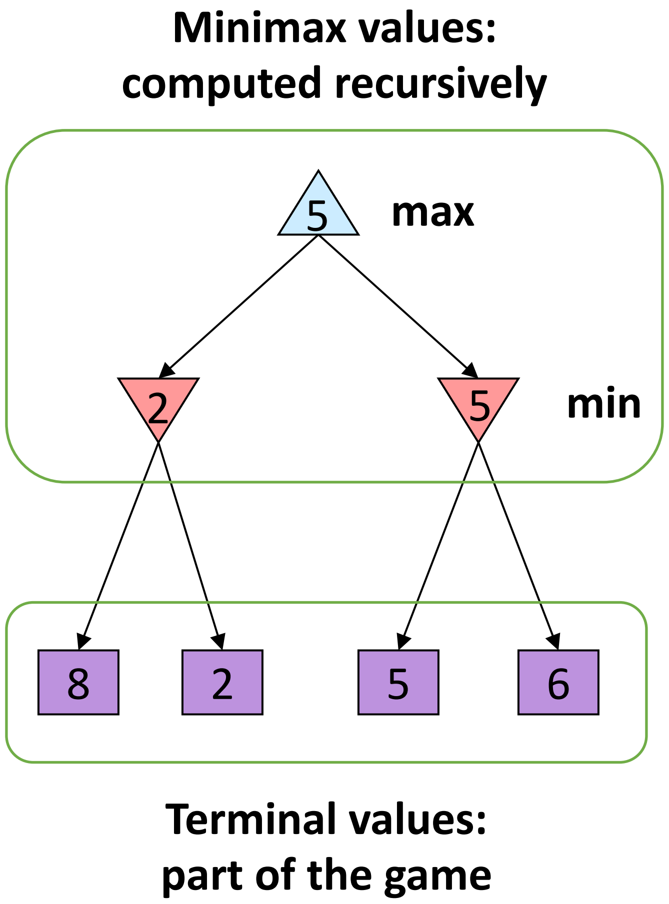
Terminal States:

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

Minimax Search

- 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 Implementation (Dispatch)

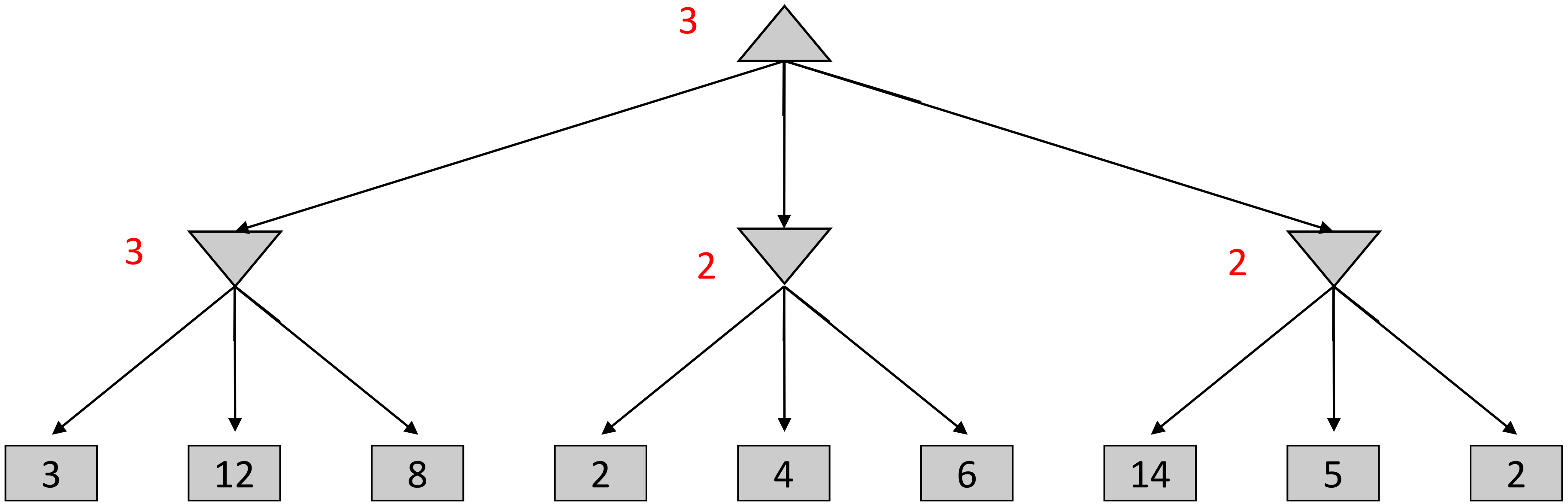
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
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
```

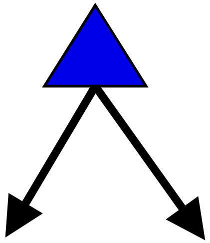
Example

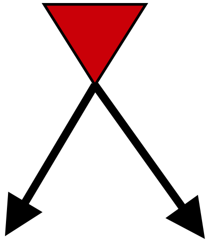
- Actions?



Pseudocode for Minimax Search

```
def max_value(state):

    if state.is_leaf:
        return state.value
    # TODO Also handle depth limit

    best_value = -10000000

    for action in state.actions:
        next_state = state.result(action)

        next_value = min_value(next_state)

        if next_value > best_value:
            best_value = next_value

    return best_value

def min_value(state):
```





$$V(s) = \max_a V(s'),$$
where $s' = result(s,a)$

$$\hat{a} = \underset{a}{\operatorname{argmax}}\, V(s'),$$
where $s' = result(s,a)$

Quiz

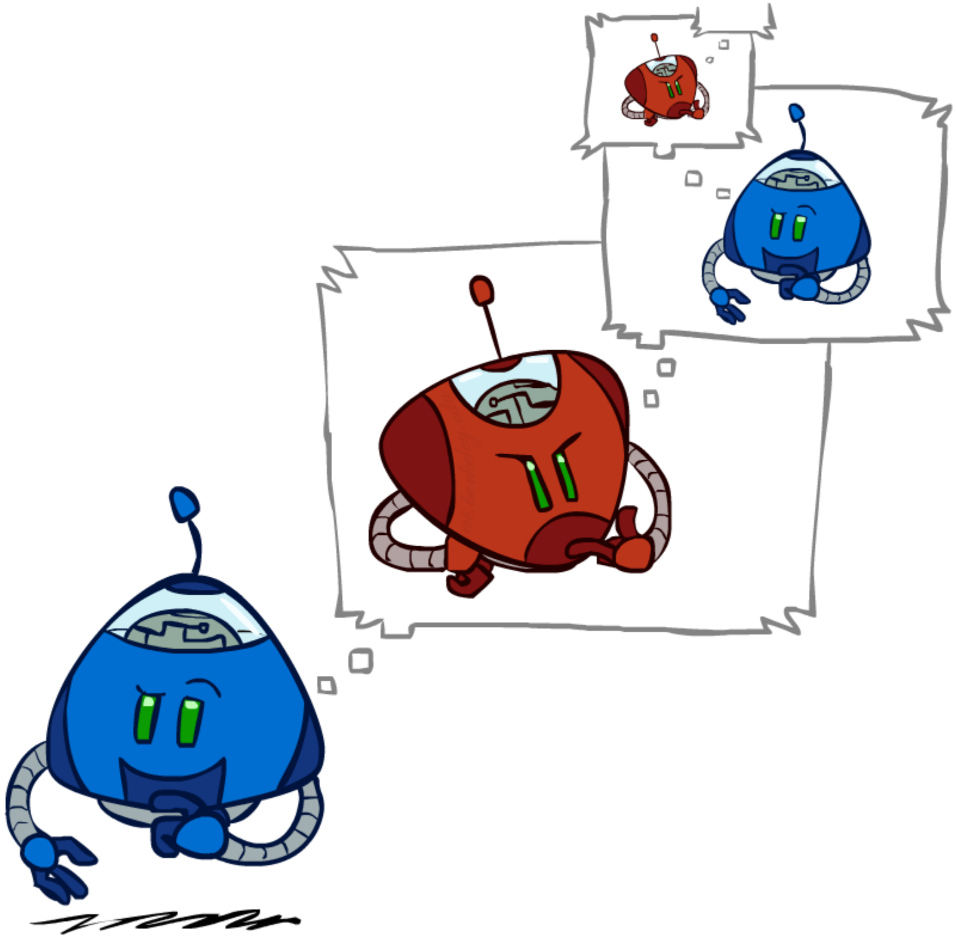
- Minimax search belongs to which class?

A) BFS
B) DFS
C) UCS
D) A*

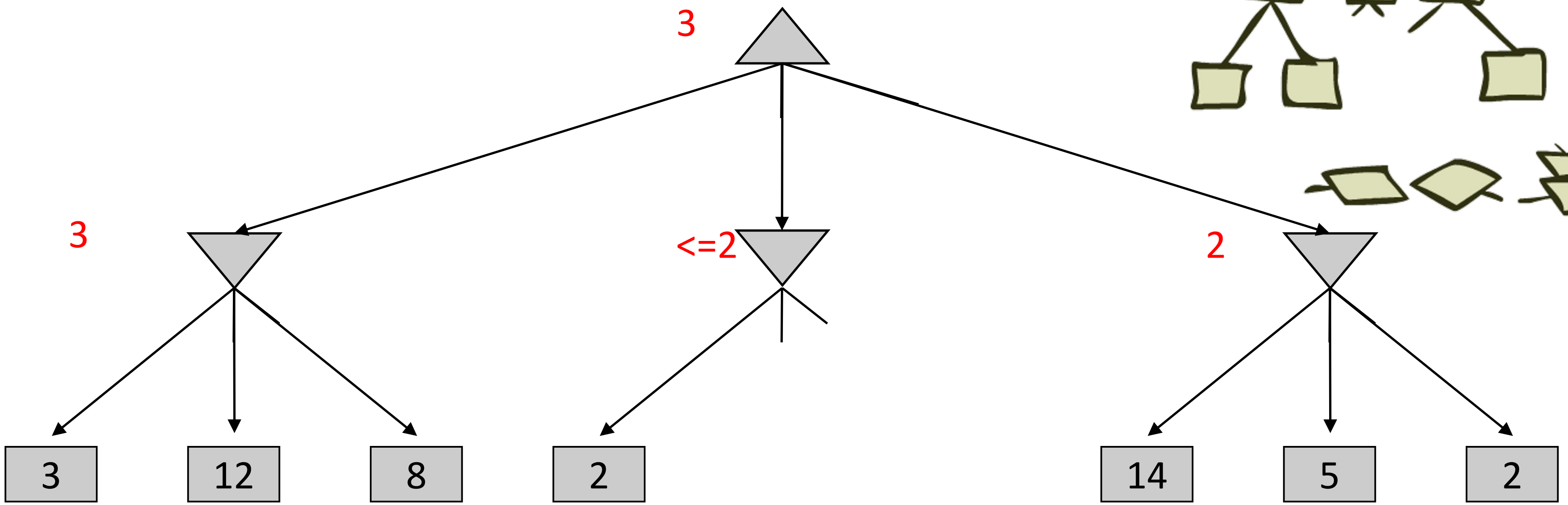
Minimax Efficiency

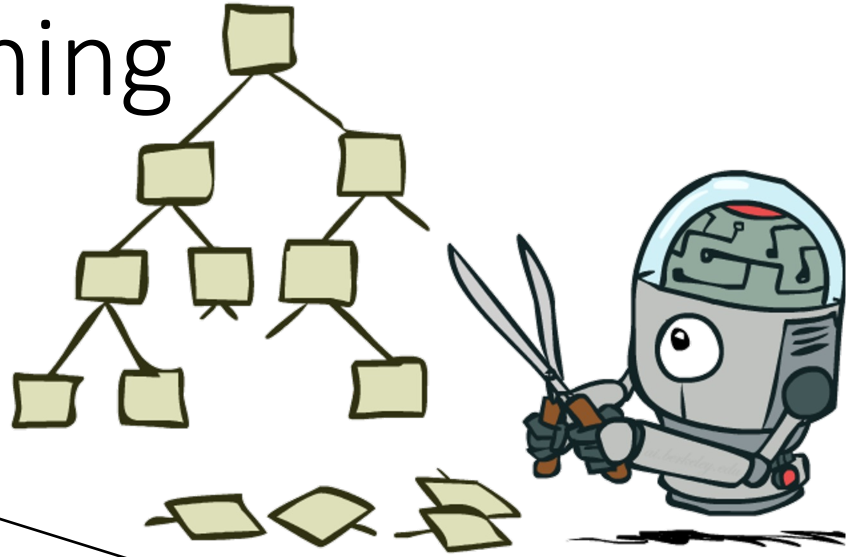
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
  - Humans can't do this either, so how do we play chess?
  - Bounded rationality – Herbert Simon



Resource Limits: Game Tree Pruning

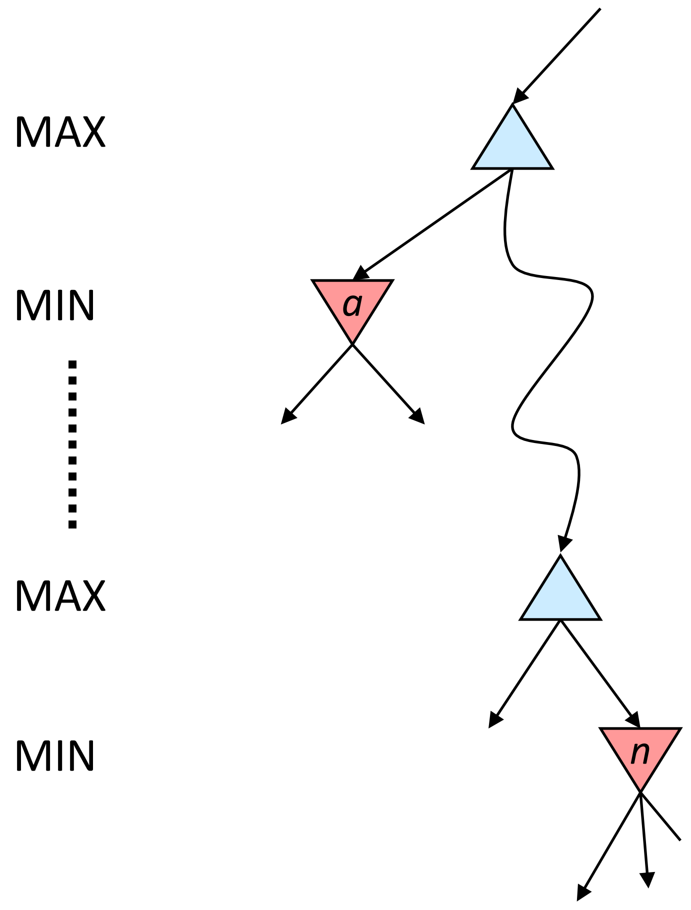




The order of generation matters: more pruning is possible if good moves come first

Game Tree 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 childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



- MAX version is symmetric

Alpha-Beta Implementation

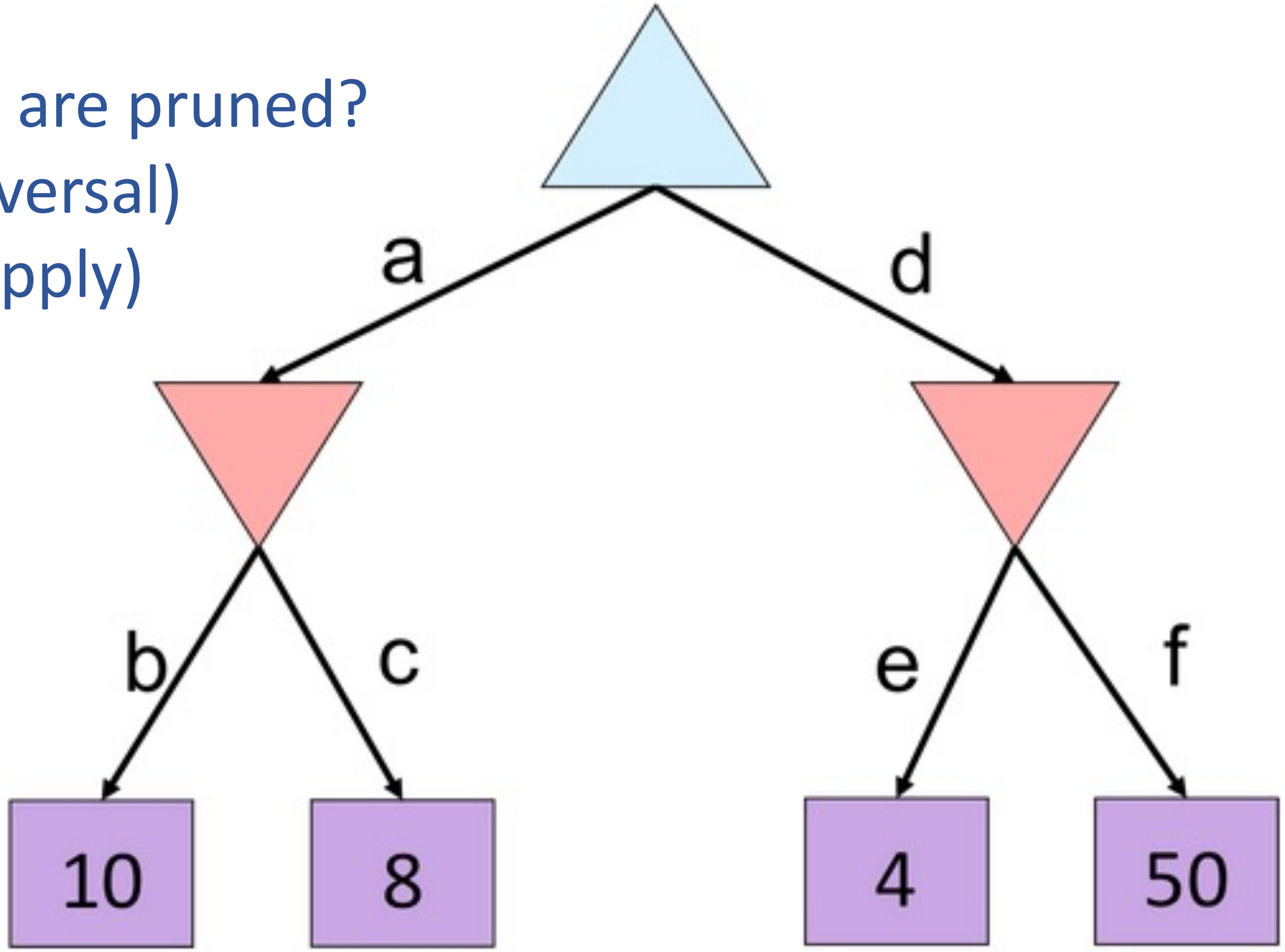
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

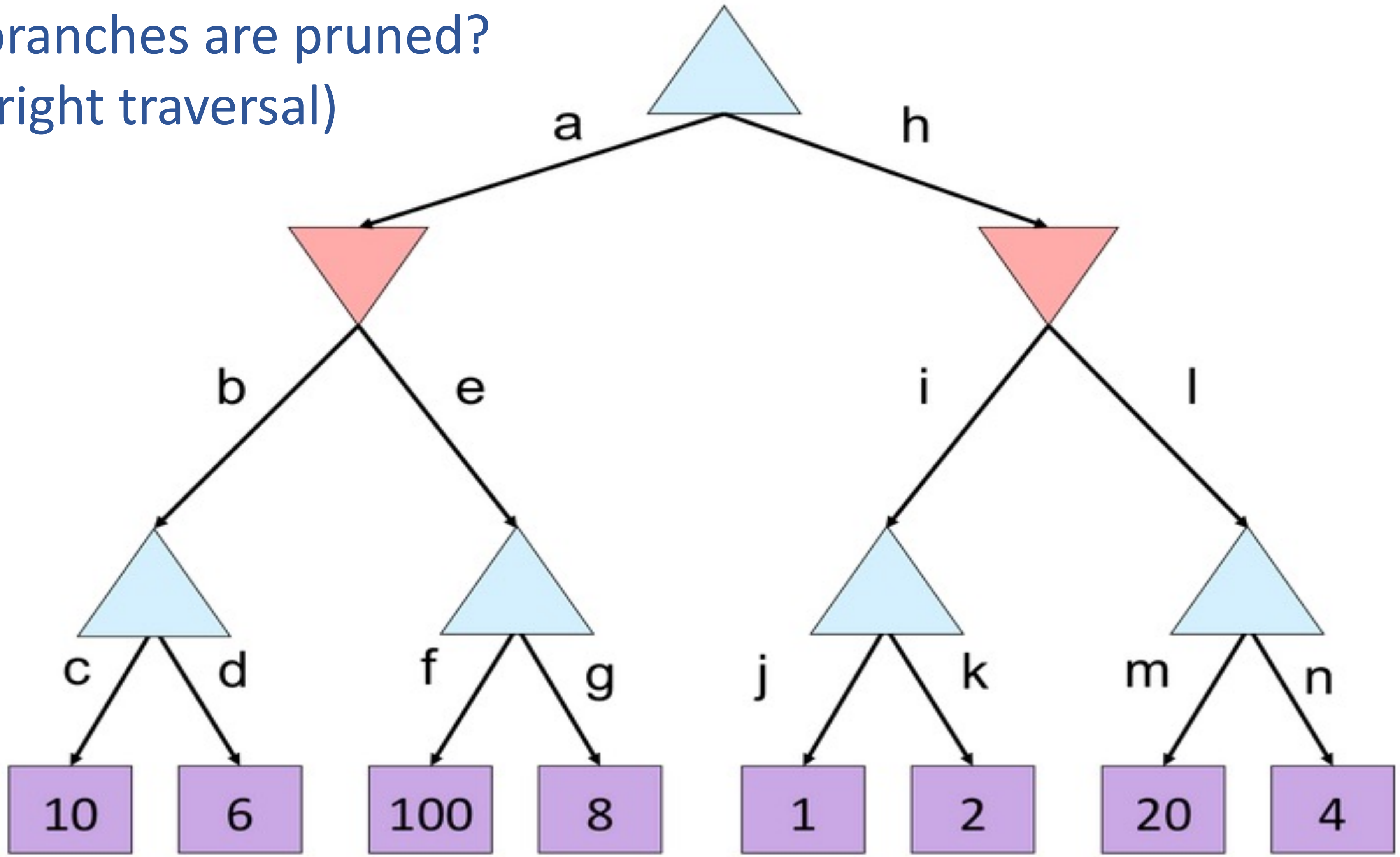
Quiz

Which branches are pruned?
(Left to right traversal)
(Select all that apply)

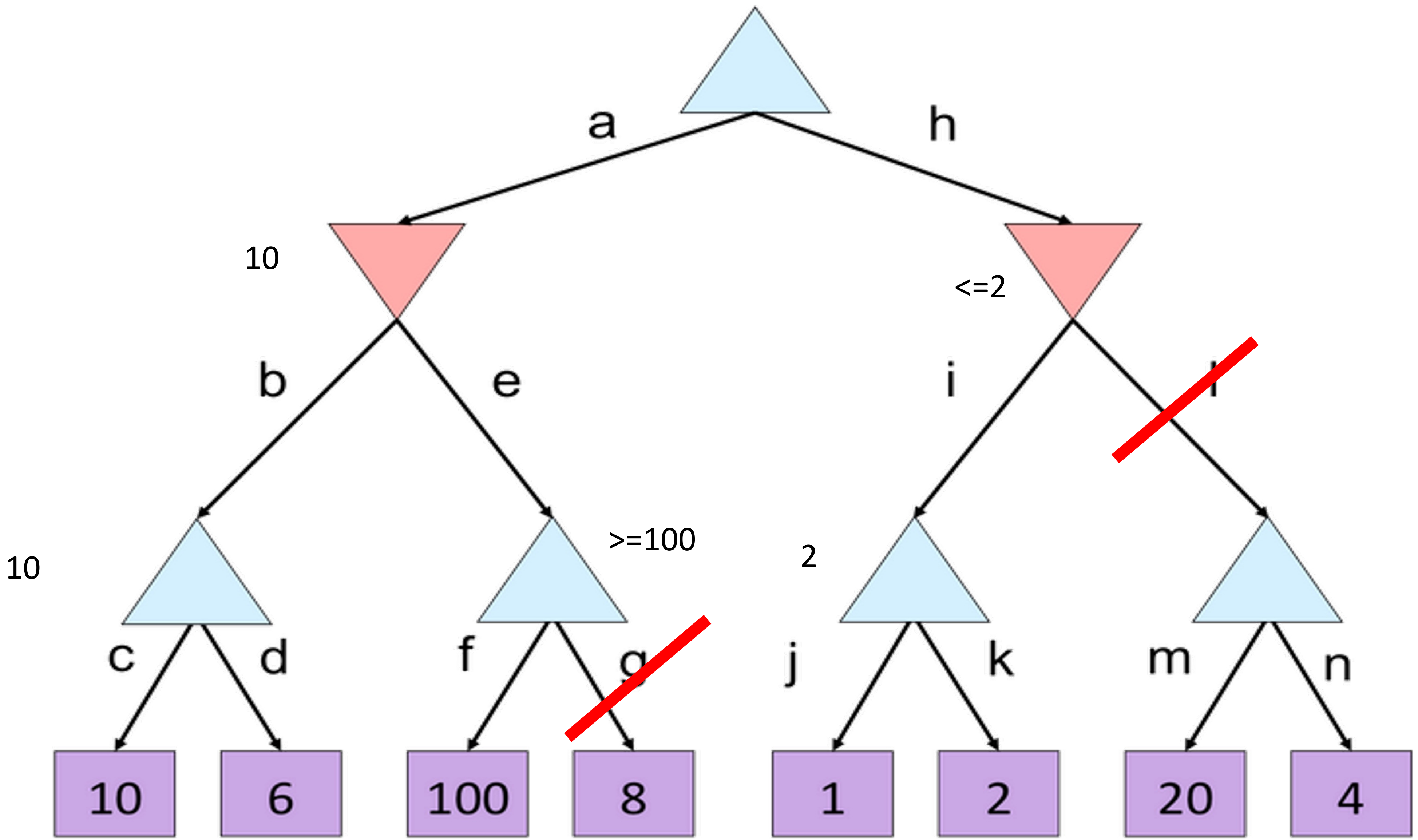


Quiz 2

Which branches are pruned?
(Left to right traversal)
A) e, l
B) g, l
C) g, k, l
D) g, n



Quiz 2 - 1



Alpha-Beta Pruning Properties

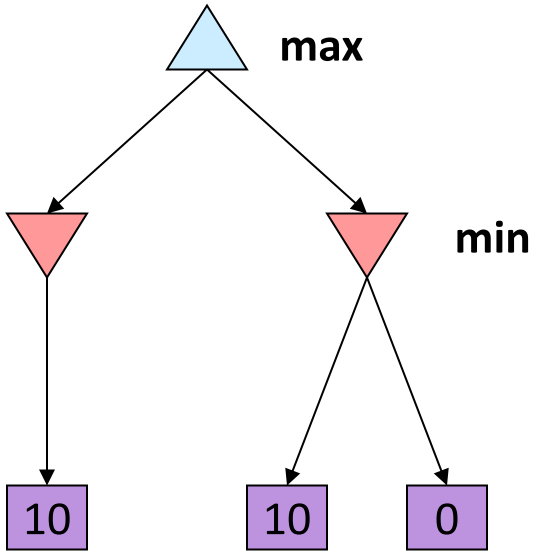
- This pruning has no effect on minimax value computed for the root!

- 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!
  - Chess: 1M nodes/move => depth=8, respectable
  - Full search of complicated games, is still hopeless…

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



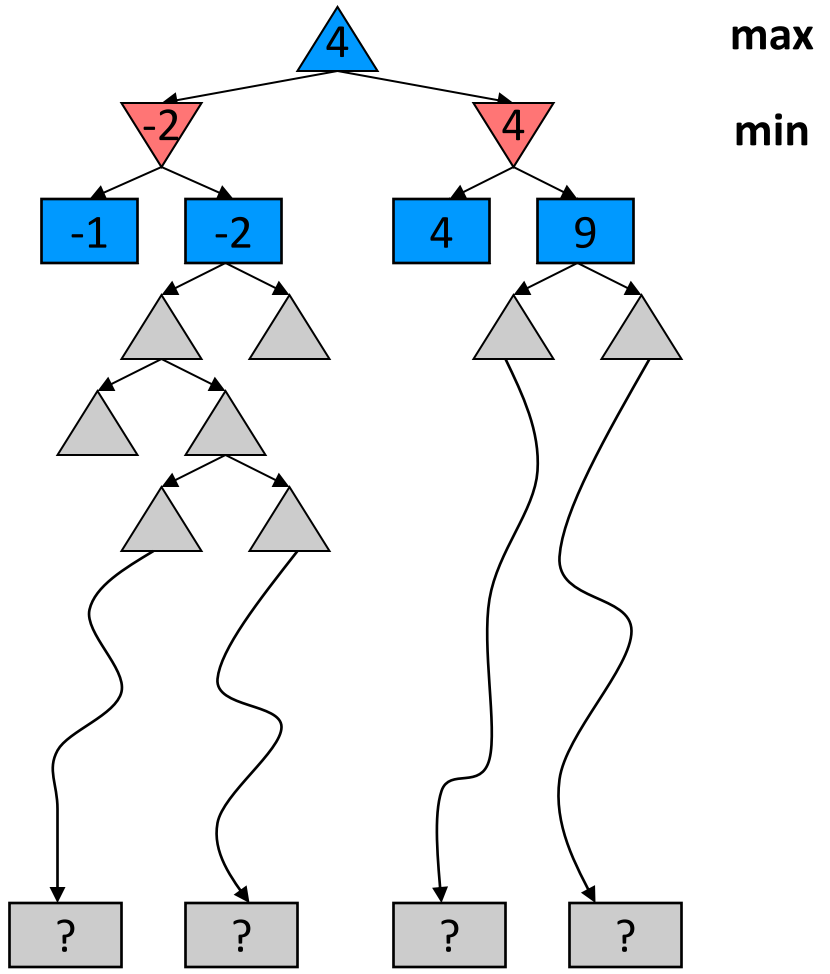
Depth-limited search

- Problem: In realistic games, cannot search to leaves!

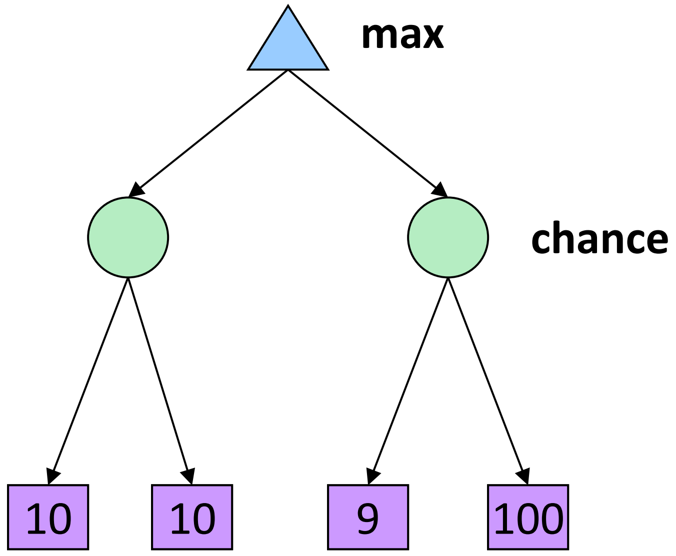
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions

- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - For chess, $b \approx 35$ so reaches about depth 4 – not so good
  - α-β reaches about depth 8 – decent chess program

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



Expectimax Search



- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Unpredictable humans: humans are not perfect
  - Actions can fail: when moving a robot, wheels might slip

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children

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

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

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)
```

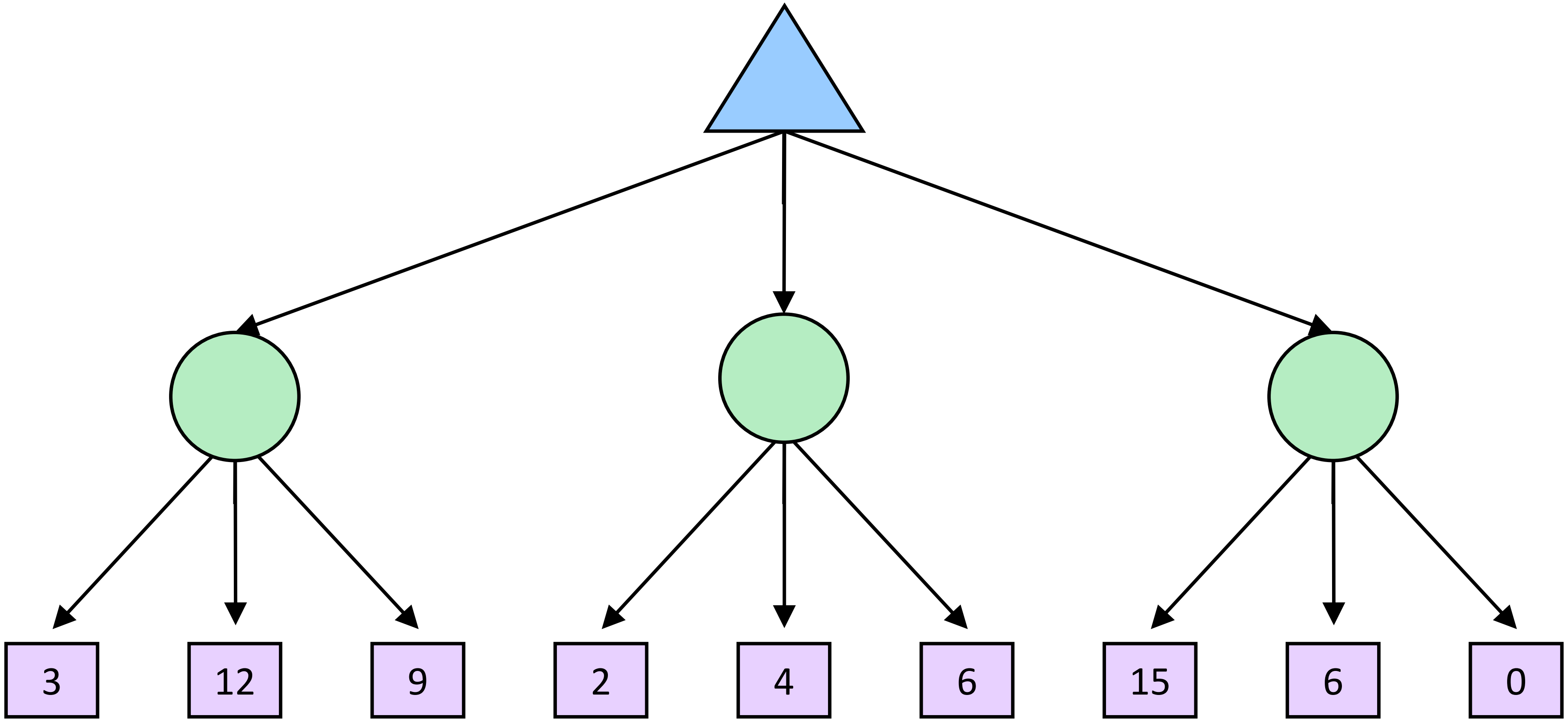
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

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

Expectimax Pseudocode 3

- function value(state)
  - if state.is_leaf
  - return state.value
  - if state.player is MAX
  - return $\max_{\text{a in state.actions}}$ value(state.result(a))
  - if state.player is MIN
  - return $\min_{\text{a in state.actions}}$ value(state.result(a))
  - if state.player is CHANCE
  - return $\text{sum}_{\text{s in state.next\_states}}$ P(s) * value(s)

Example



Quiz

Expectimax tree search:

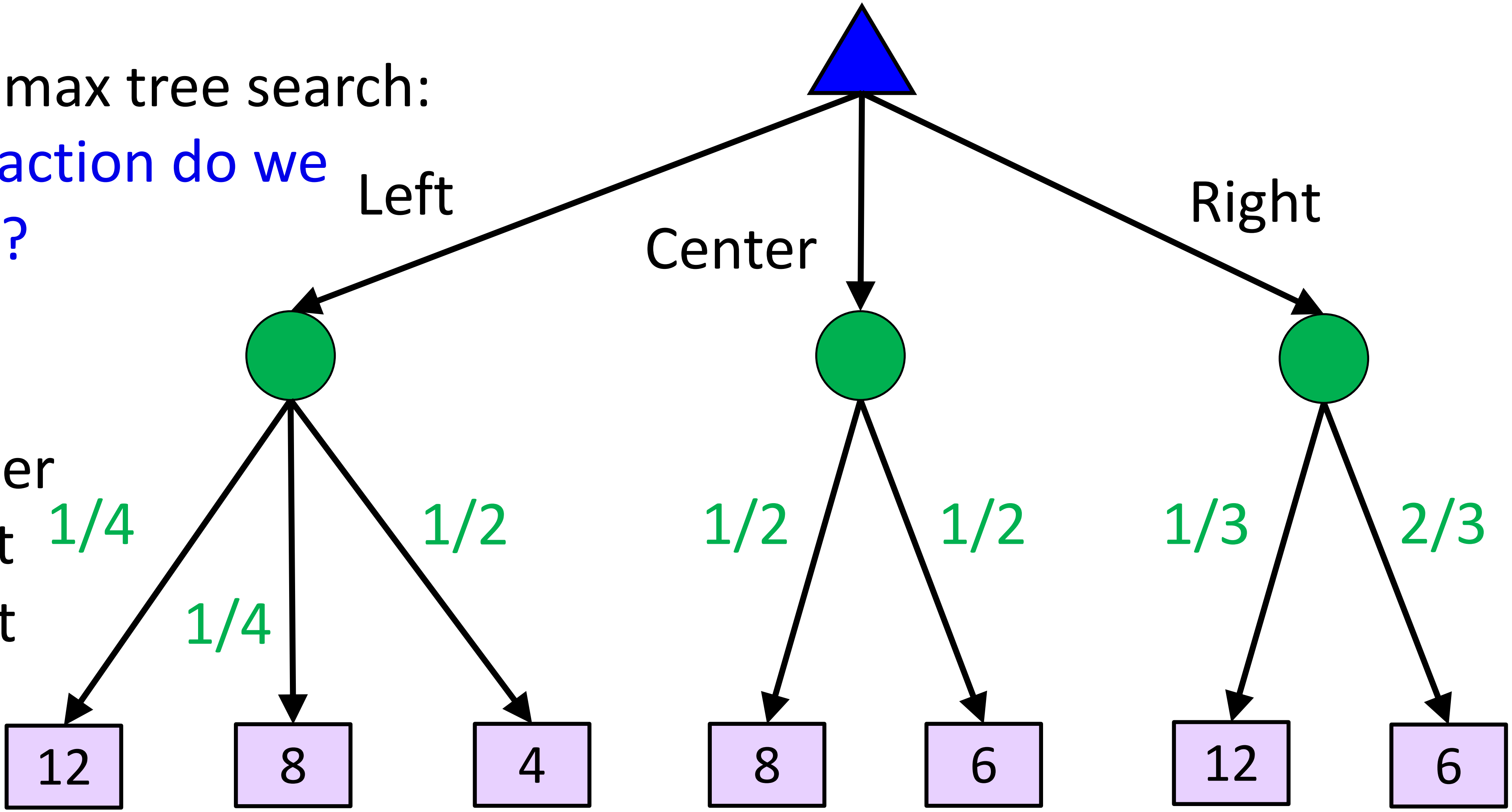
Which action do we choose?

A: Left

B: Center

C: Right

D: Eight



Quiz 2

Expectimax tree search:

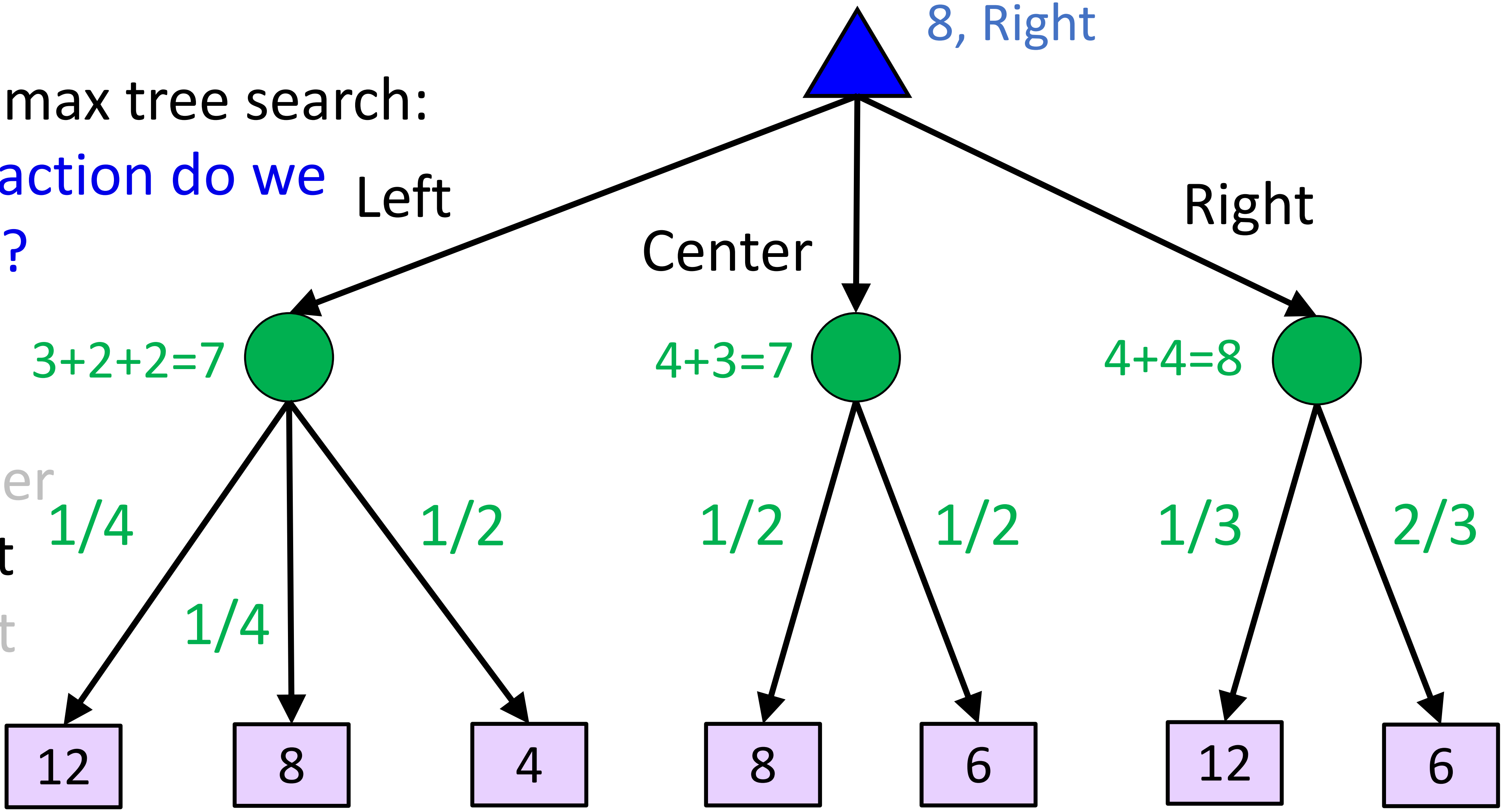
Which action do we choose?

A: Left

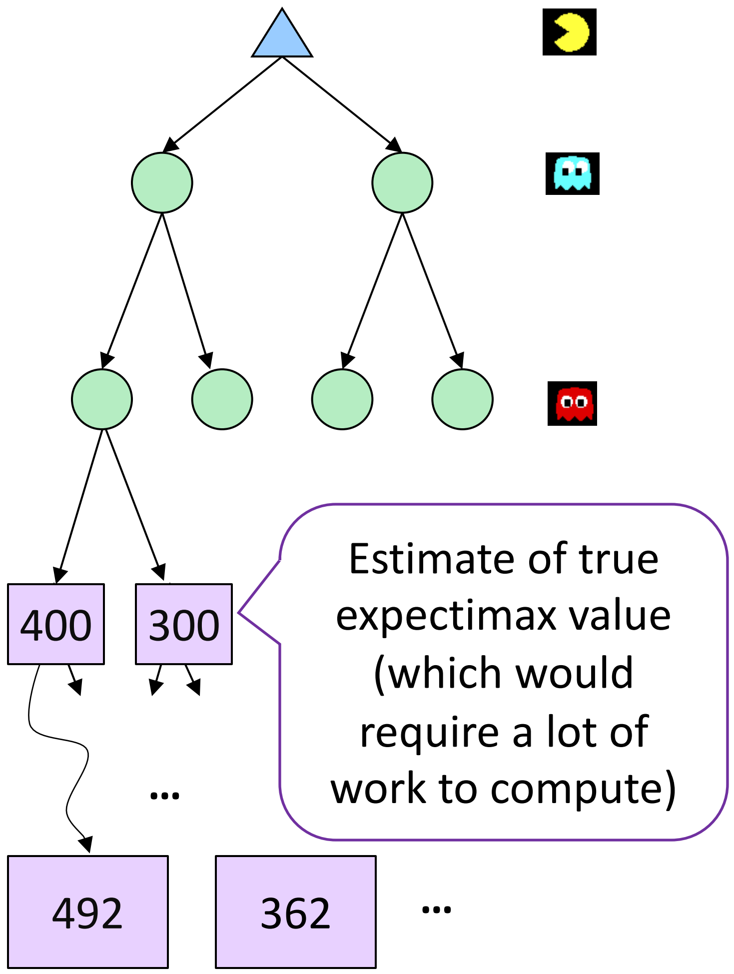
B: Center

C: Right

D: Eight

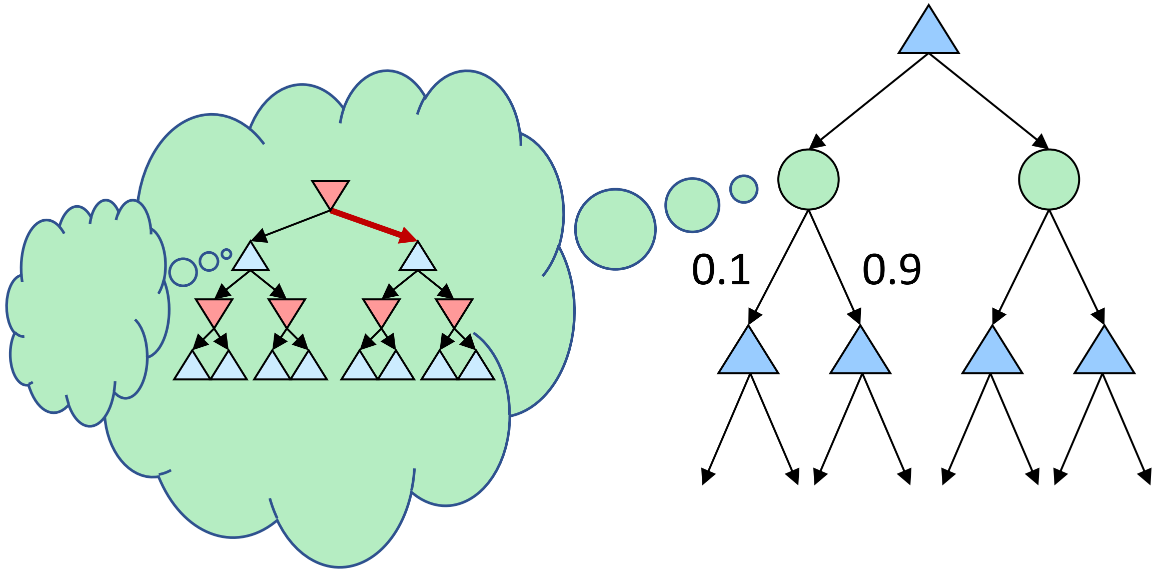


Expectimax: Depth-Limited



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 and maximax, which have the nice property that it all collapses into one game tree

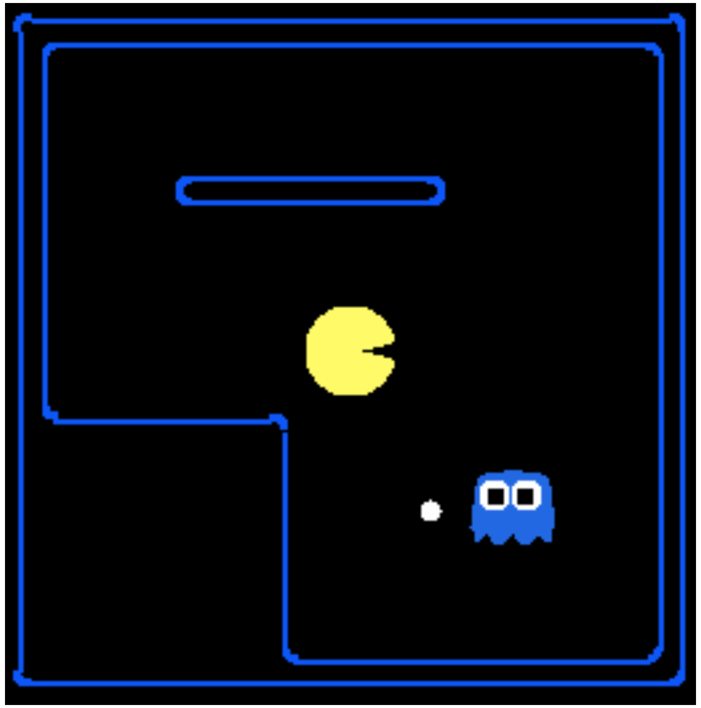
This is basically how you would model a human, except for their utility: their utility might be the same as yours (i.e. you try to help them, but they are depth 2 and noisy), or they might have a slightly different utility (like another person navigating in the office)

Dangerous Pessimism/Optimism

Assuming the worst case when it's not likely

Assuming chance when the world is adversarial

Assumptions vs. Reality



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

Results from playing 5 games

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

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

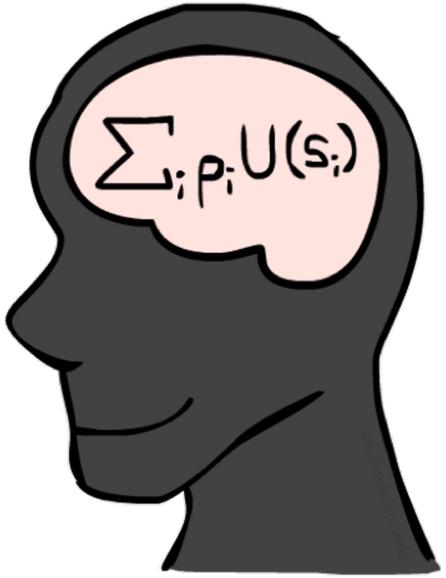
MEU Principle

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

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

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

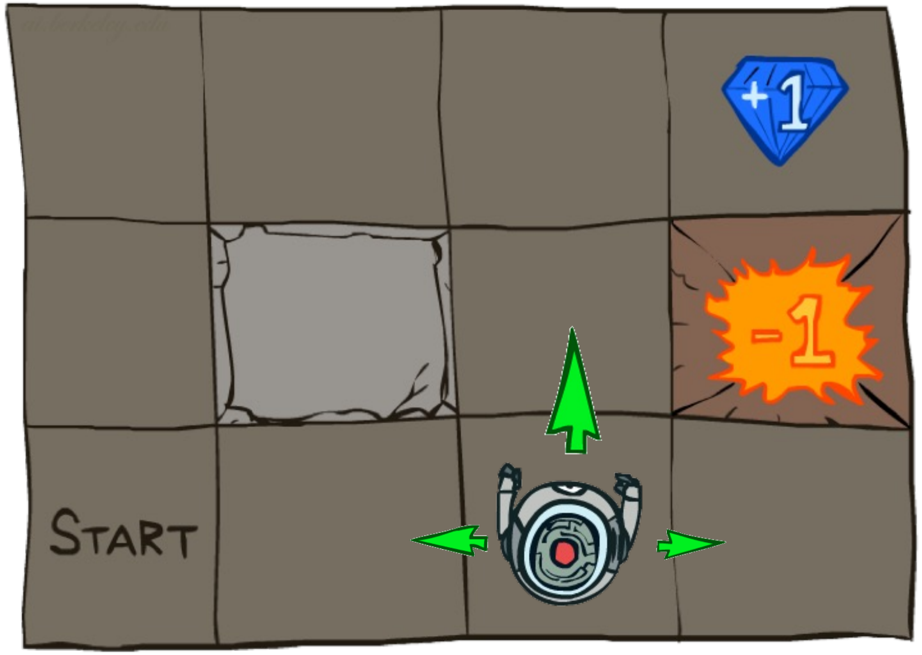
  - i.e. values assigned by U preserve preferences of both prizes and lotteries!



- 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

Markov Decision Processes

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

- MDPs are non-deterministic search problems
  - One way to solve them is with expectimax search
  - We'll have a new tool soon

[Demo – gridworld manual intro (L8D1)]

What is Markov about MDPs?

- "Markov" generally means that given the present state, the future and the past are independent
- For Markov decision processes, "Markov" means action outcomes depend only on the current state

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots S_0 = s_0)$$

$$=$$

$$P(S_{t+1} = s' | S_t = s_t, A_t = a_t)$$

- This is just like search, where the successor function could only depend on the current state (not the history)

Andrey Markov
(1856-1922)

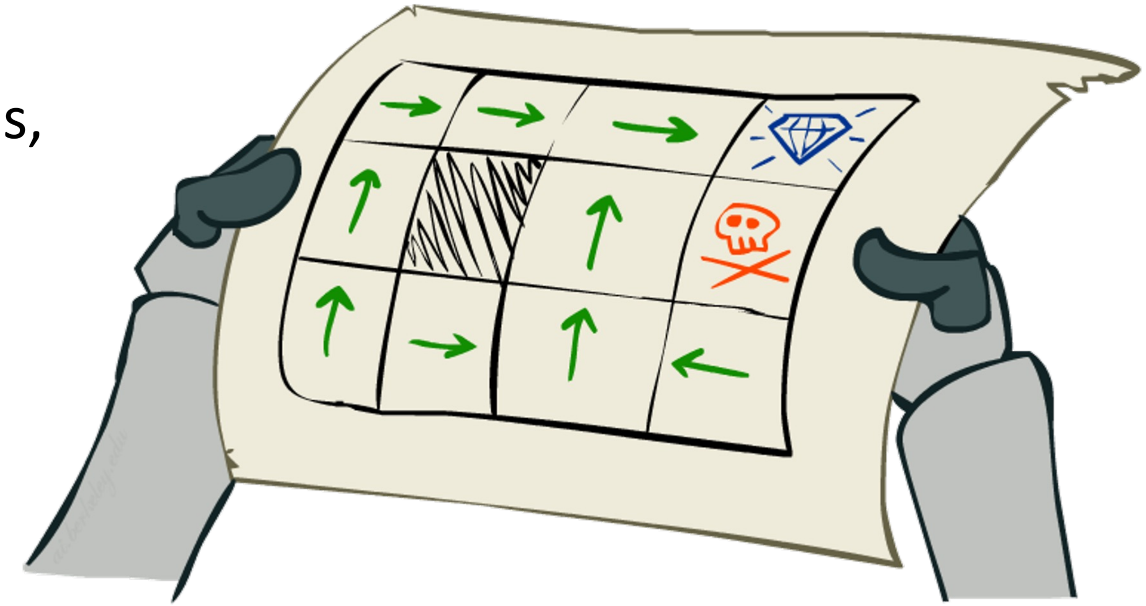
Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal

- For MDPs, we want an optimal

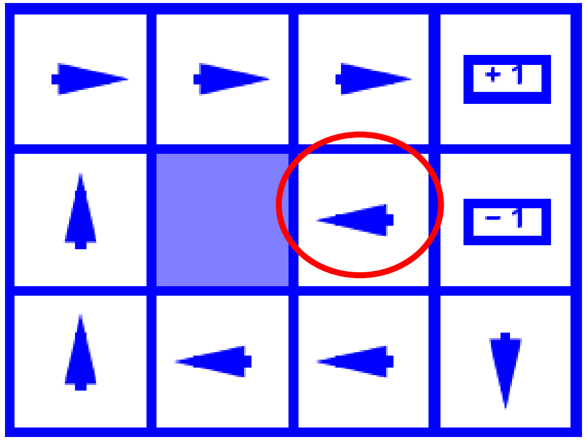
  policy $\pi^*: S \rightarrow A$

  - A policy π gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent

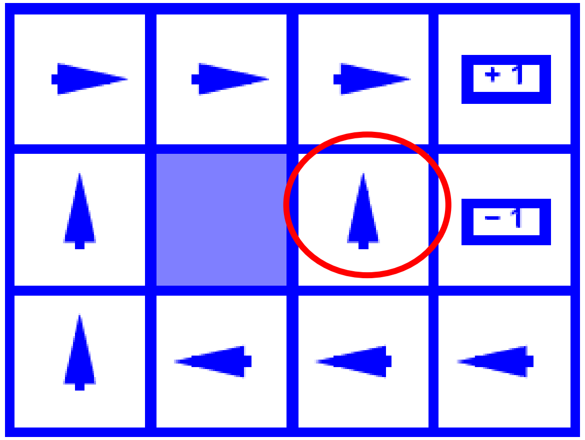


Optimal policy when R(s, a, s') = -0.03 for all non-terminals s

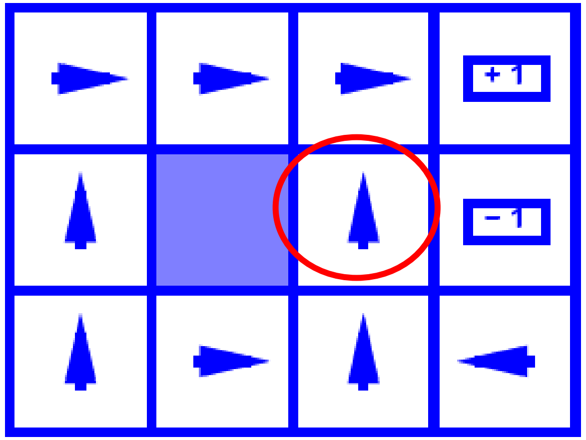
Optimal Policies



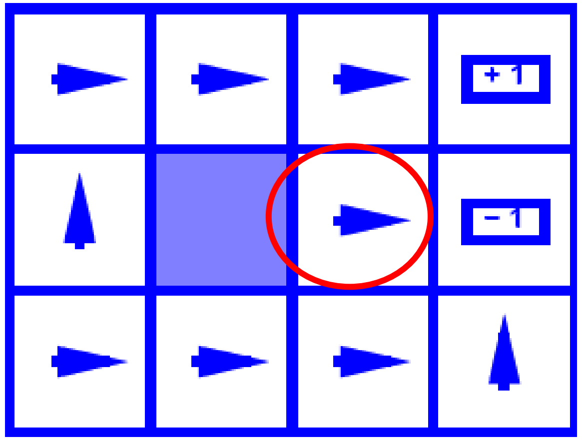
R(s) = -0.01



R(s) = -0.03



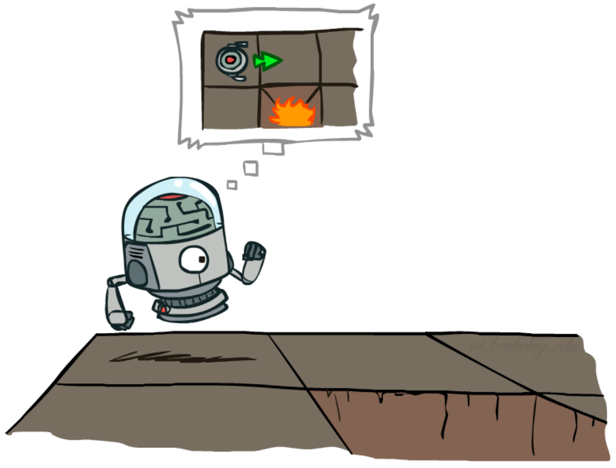
R(s) = -0.4

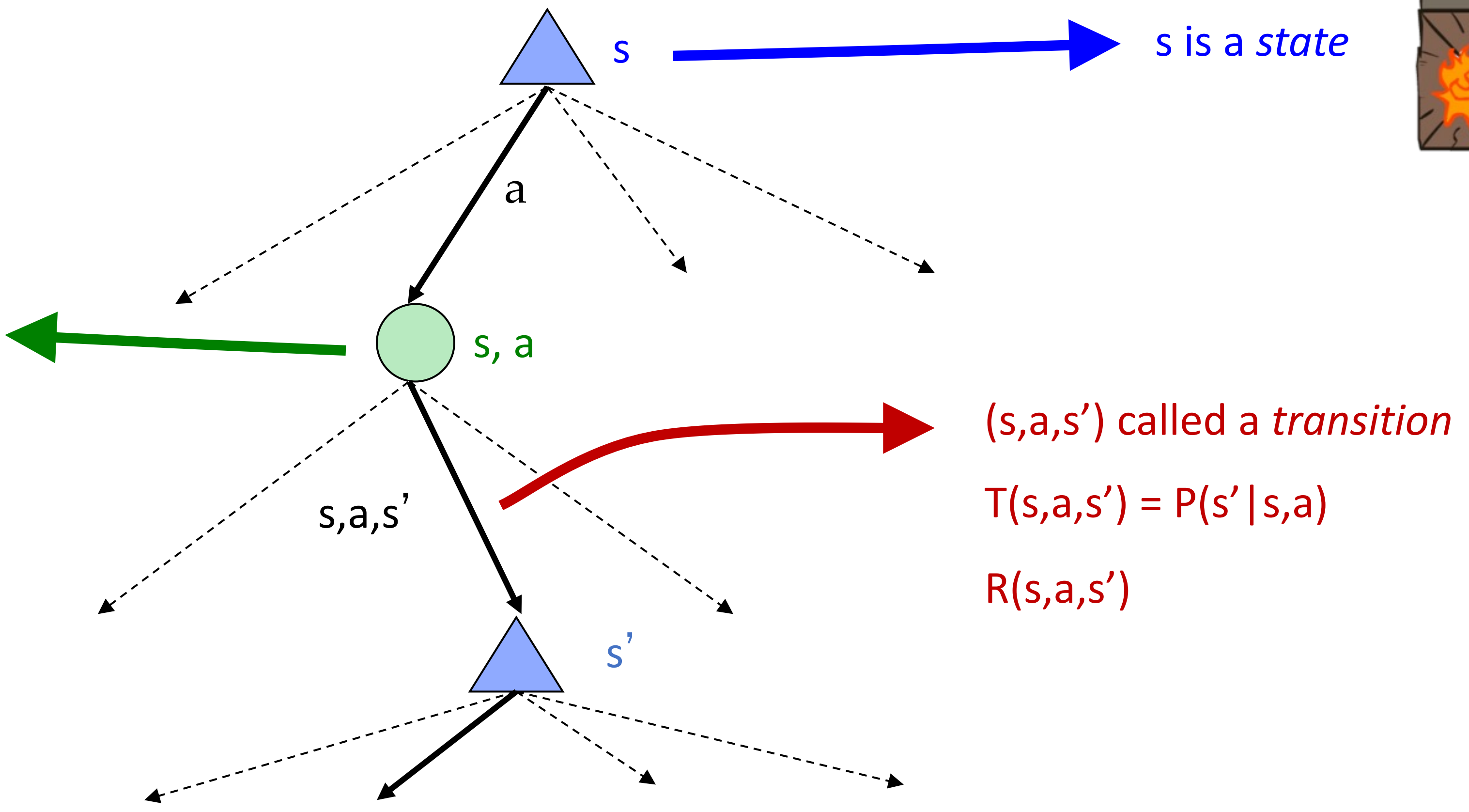


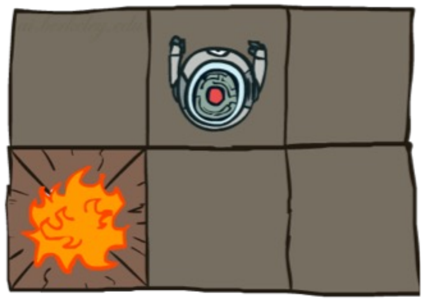
R(s) = -2.0

MDP Search Trees

- Each MDP state projects an expectimax-like search tree

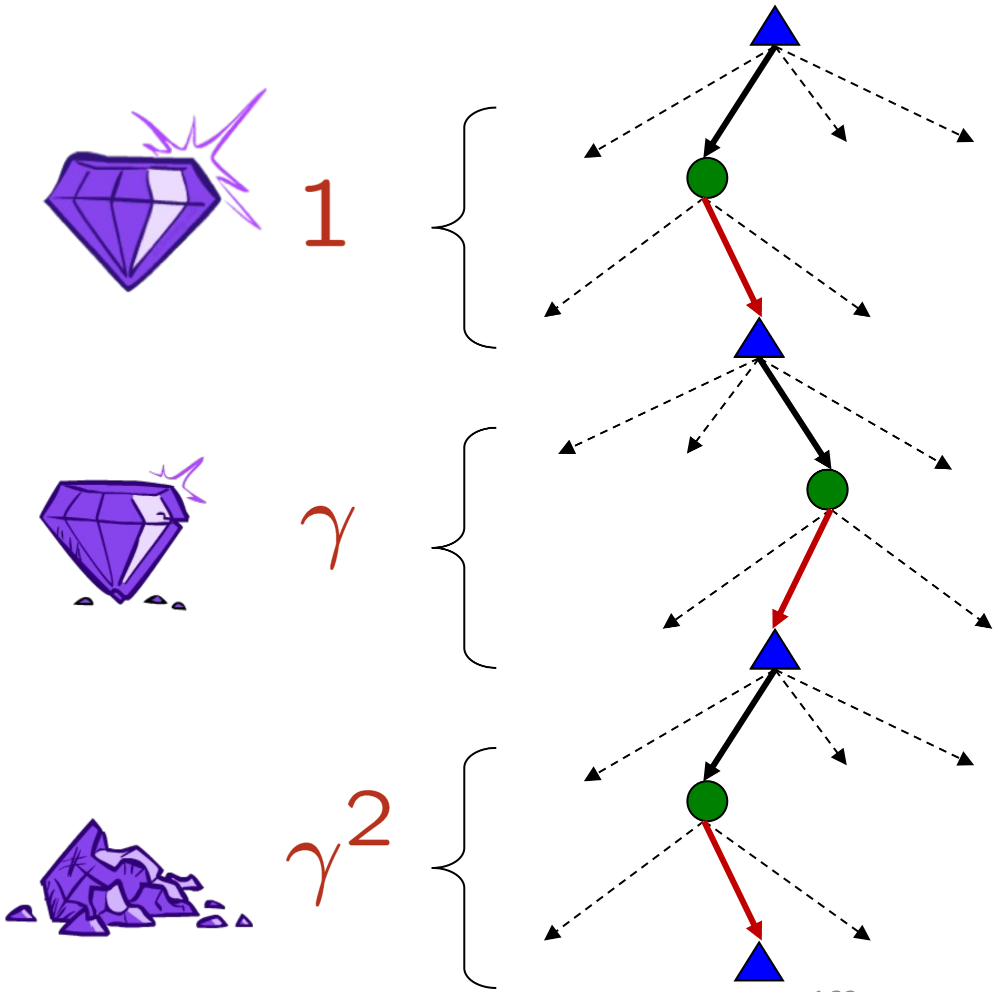






Utilities of Sequences: Discounting

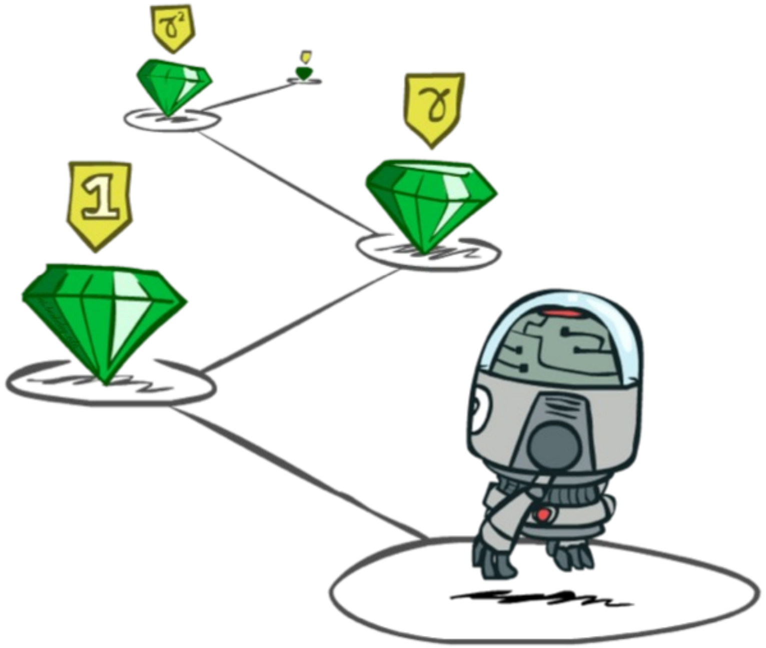
- How to discount?
  - Each time we descend a level, we multiply in the discount once
- Why discount?
  - Reward now is better than later
  - Can also think of it as a 1-gamma chance of ending the process at every step
  - Also helps our algorithms converge
- Example: discount of 0.5
  - U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3
  - U([1,2,3]) < U([3,2,1])



Utilities of Sequences: Stationary Preferences

- Theorem: if we assume stationary preferences:

$$[a_1, a_2, \ldots] \succ [b_1, b_2, \ldots]$$
$$\Updownarrow$$
$$[r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots]$$



- Then: there are only two ways to define utilities
  - Additive utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots$
  - Discounted utility: $U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \cdots$

Quiz: Discounting

- Given:

10				1
a	b	c	d	e

  - Actions: East, West, and Exit (only available in exit states a, e)
  - Transitions: deterministic

- Quiz 1: For γ = 1, what is the optimal policy?

10	<-	<-	<-	1

- Quiz 2: For γ = 0.1, what is the optimal policy?

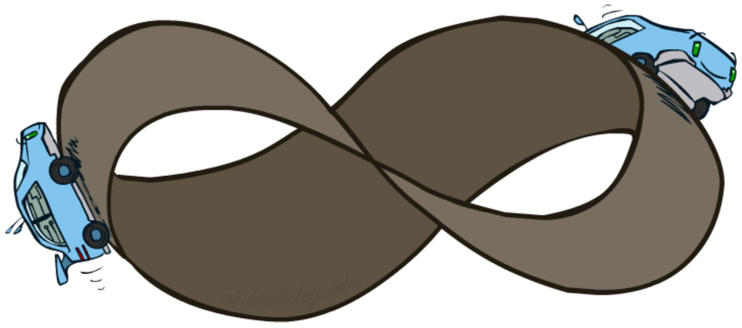
10	<-	<-	->	1

- Quiz 3: For which γ are West and East equally good when in state d?

$1\gamma = 10\,\gamma^3$

Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?
- 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)



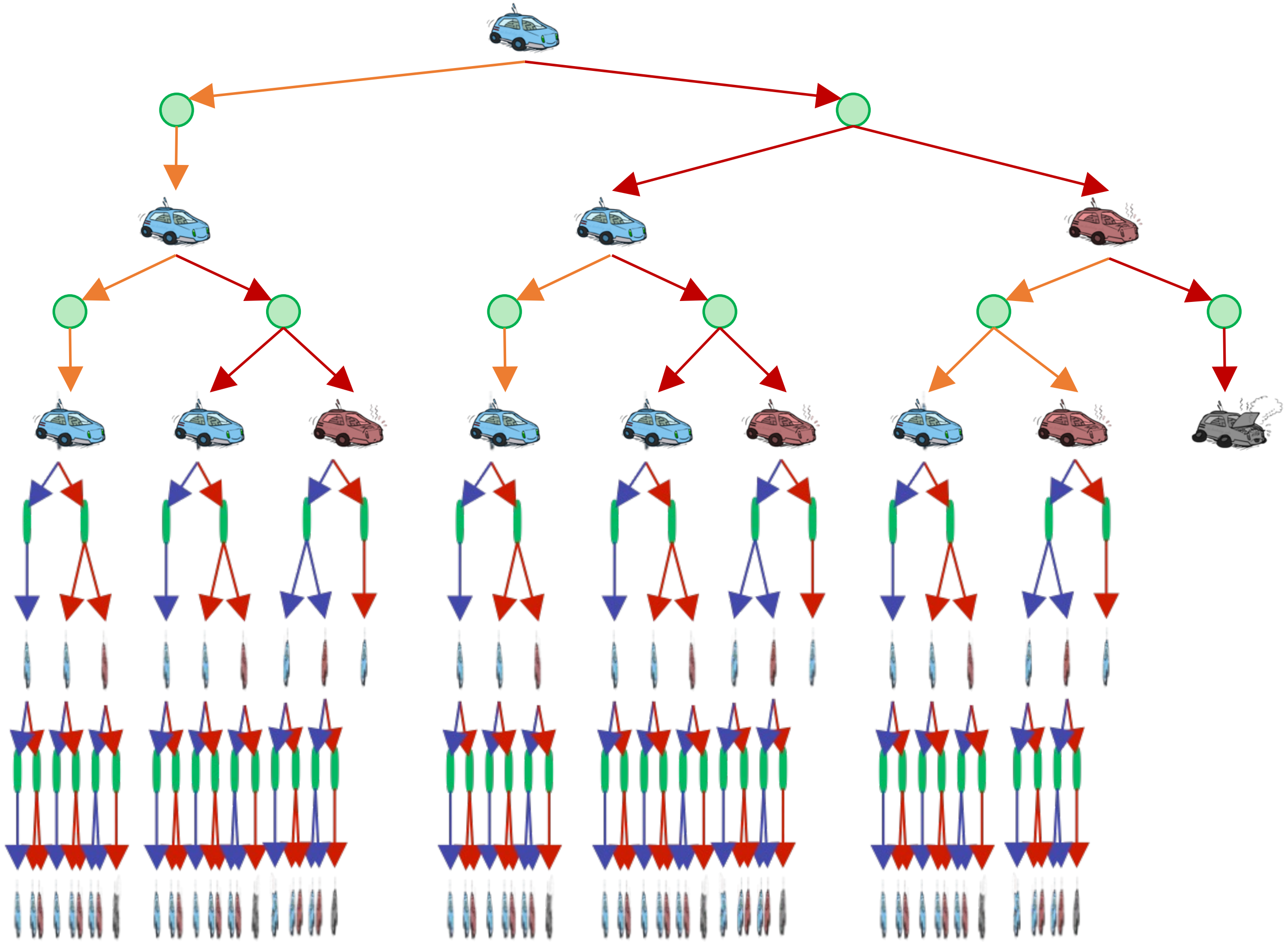
  - Discounting: use $0 < \gamma < 1$

$$U([r_0, \ldots r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\max}/(1-\gamma)$$

    - Smaller γ means smaller "horizon" – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like "overheated" for racing)

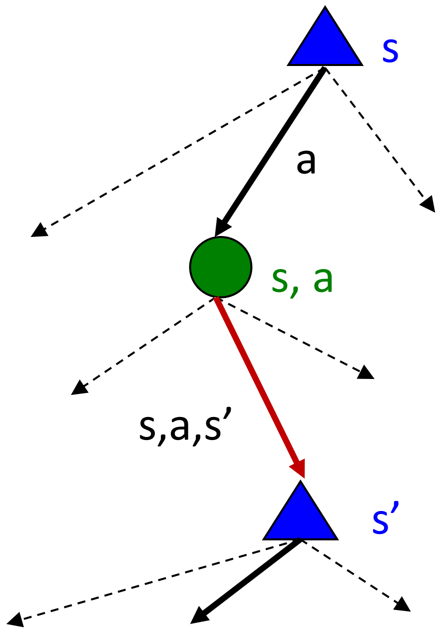
Racing Search Tree

- We're doing way too much work with expectimax!
- 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 $\gamma < 1$



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



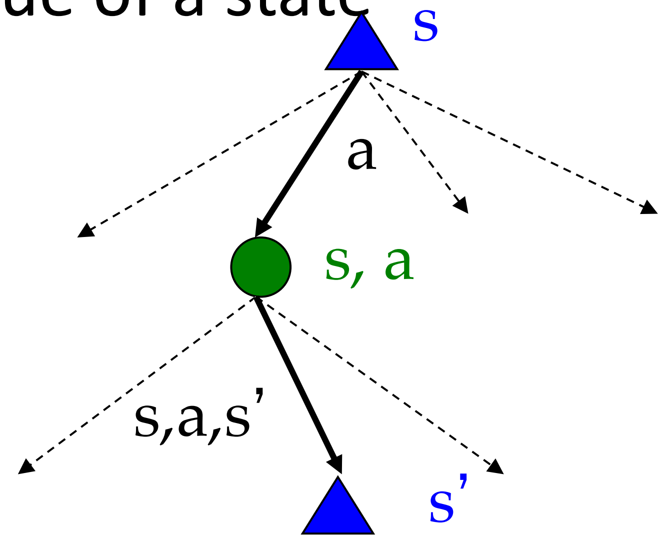
s is a *state*

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

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

Values of States

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!



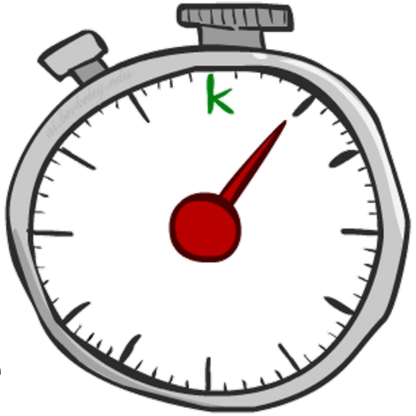
- Recursive definition of value:

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

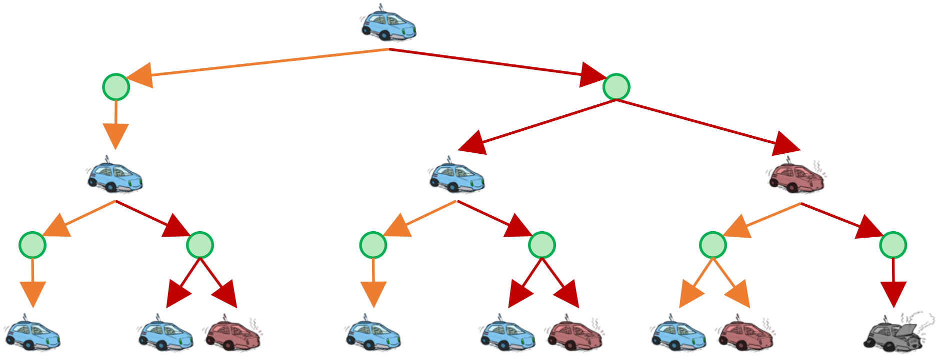
$$Q^*(s,a) = \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma\, V^*(s') \right]$$

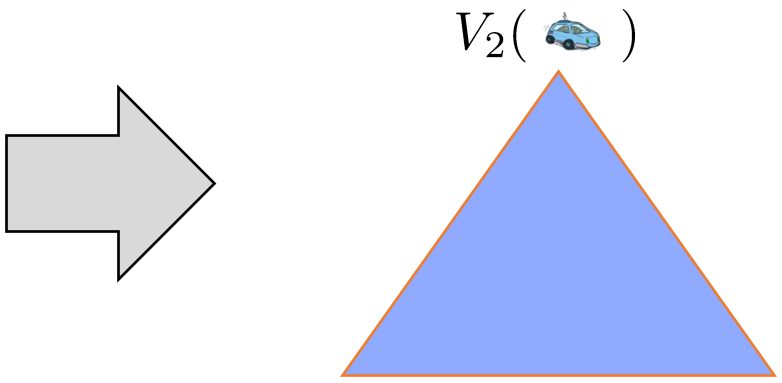
$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Time-Limited Values



- 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





[Demo – time-limited values (L8D4)]

Value Iteration

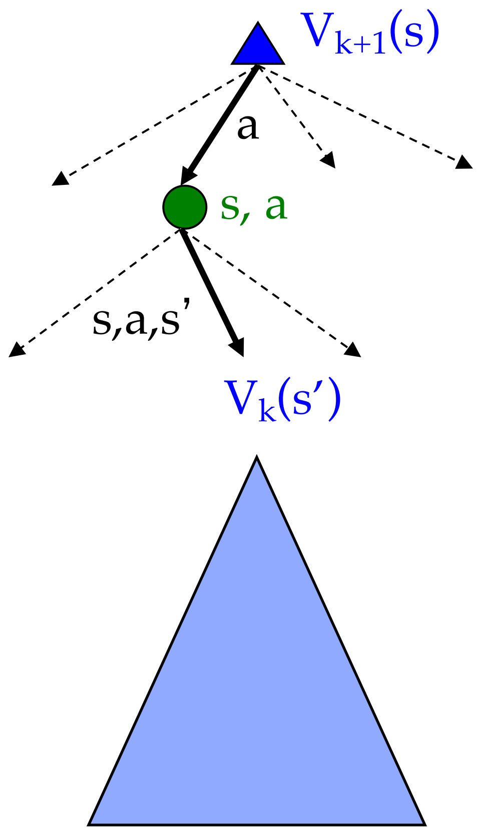
- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

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

- Repeat until convergence, which yields V*

- Complexity of each iteration: $O(S^2A)$

- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

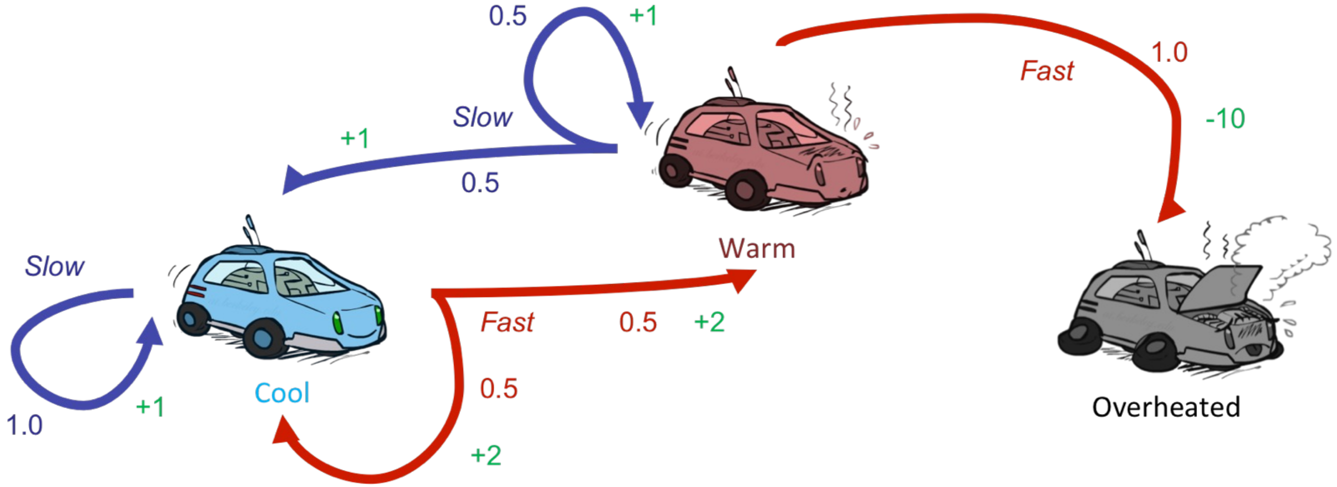


Example

V_2

V_1
S: 1
F: .5*2+.5*2=2

V_0
0 0 0

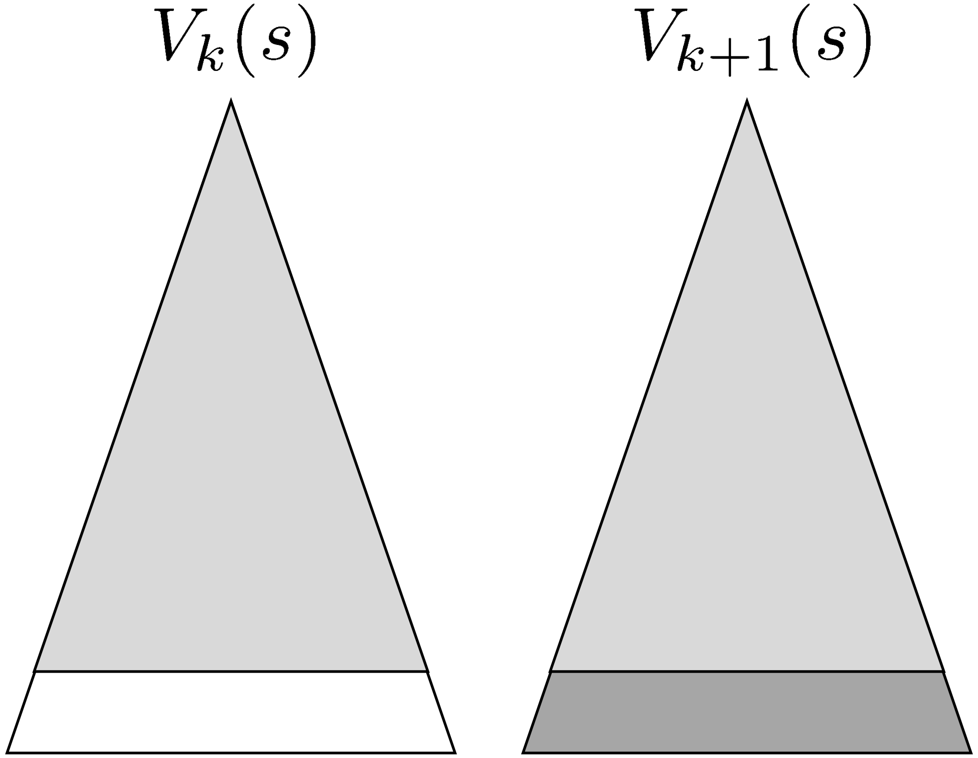


Assume no discount!

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

Convergence

- How do we know the V_k vectors are going to converge?
- Case 1: If the tree has maximum depth M, then V_M holds the actual untruncated values
- Case 2: If the discount is less than 1
- Proof Sketch:
  - For any state V_k and V_{k+1} can be viewed as depth k+1 expectimax results in nearly identical search trees
  - The difference is that on the bottom layer, V_{k+1} has actual rewards while V_k has zeros
  - That last layer is at best all R_{MAX}
  - It is at worst R_{MIN}
  - But everything is discounted by γ^k that far out
  - So V_k and V_{k+1} are at most γ^k max|R| different
  - So as k increases, the values converge



Value Iteration (Revisited)

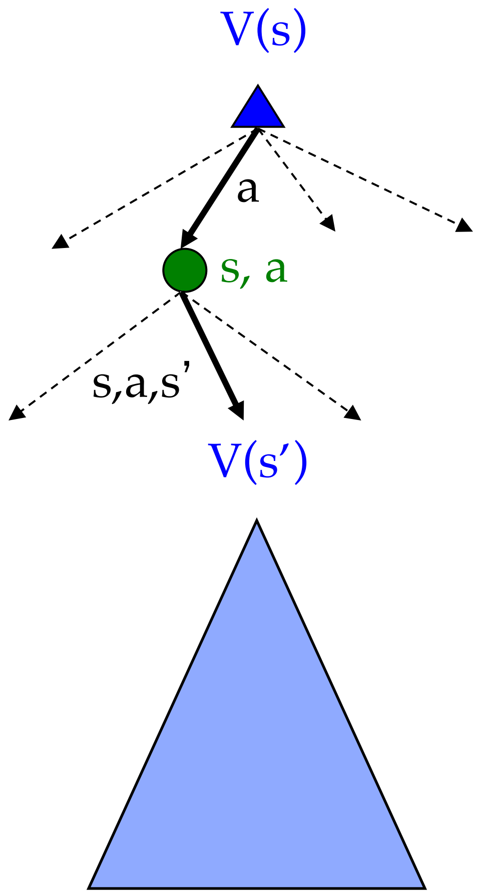
- Bellman equations characterize the optimal values:

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

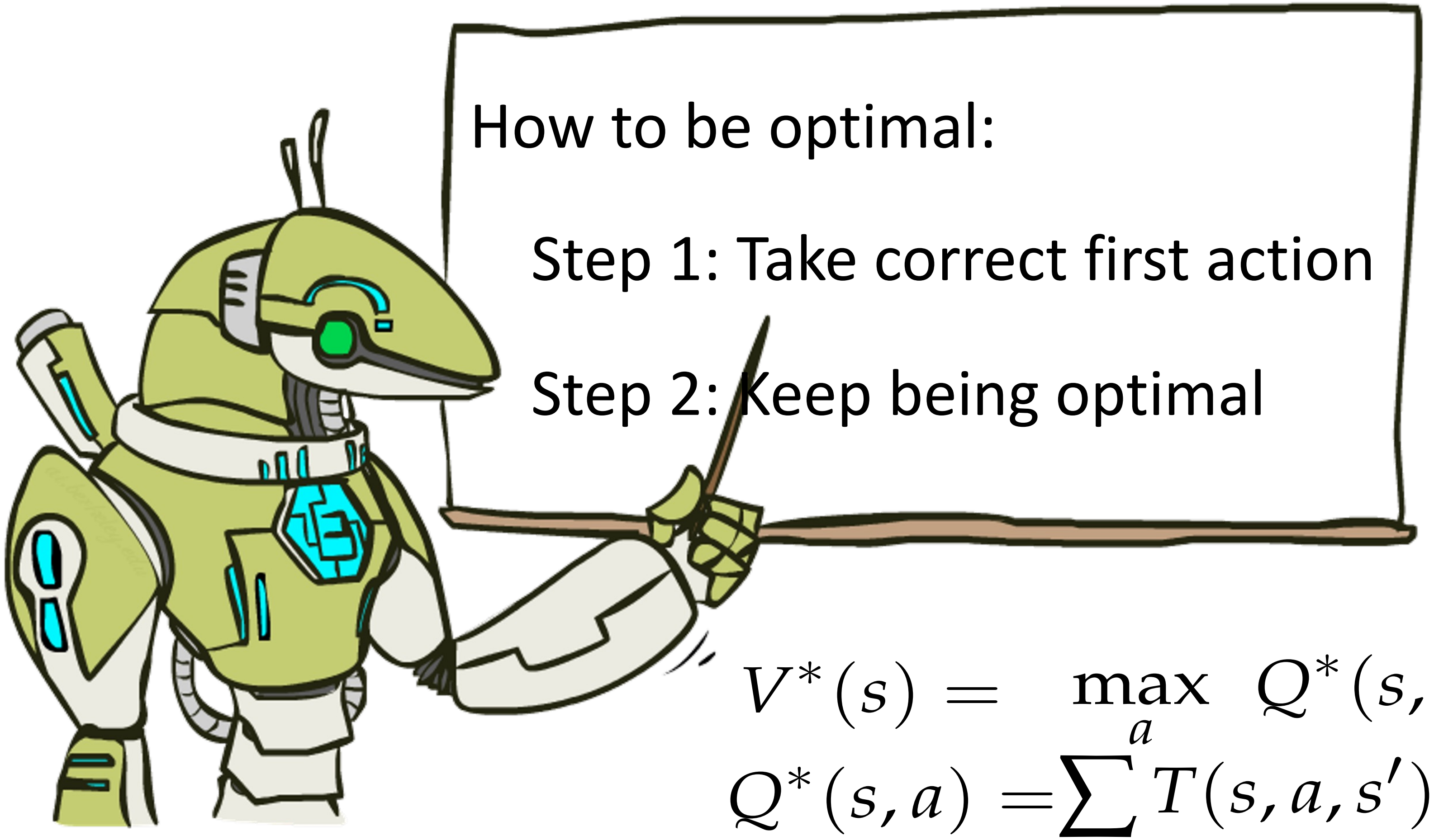
- Value iteration computes them:

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

- Value iteration is just a fixed point solution method
  - … though the V_k vectors are also interpretable as time-limited values



The Bellman Equations

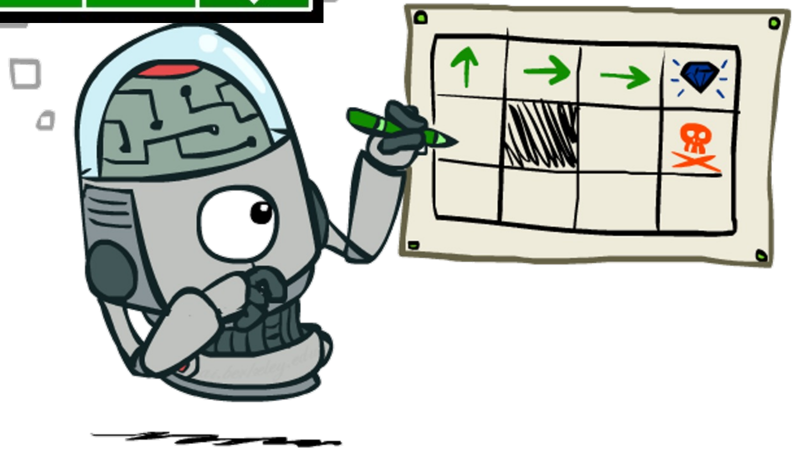


$$V^*(s) = \max_a Q^*(s,a)$$

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

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

Policy Extraction: Computing Actions from Values



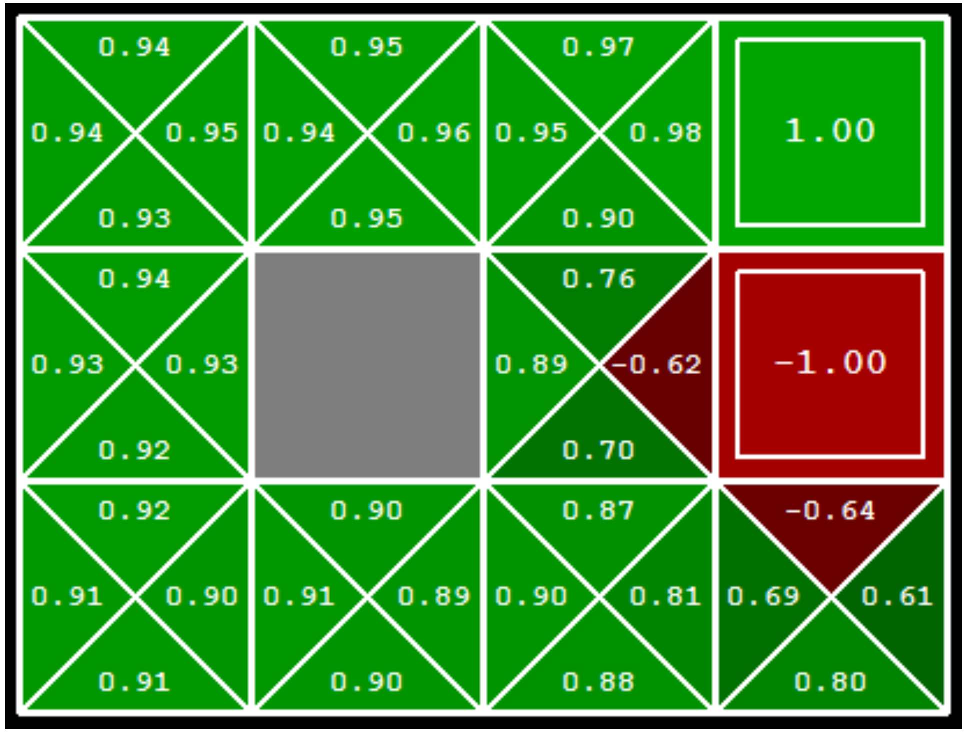
- Let's imagine we have the optimal values V*(s)
- How should we act?
  - It's not obvious!
- We need to do a mini-expectimax (one step)

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

- This is called policy extraction, since it gets the policy implied by the values

Policy Extraction: Computing Actions from Q-Values

- Let's imagine we have the optimal q-values:



- How should we act?
  - Completely trivial to decide!

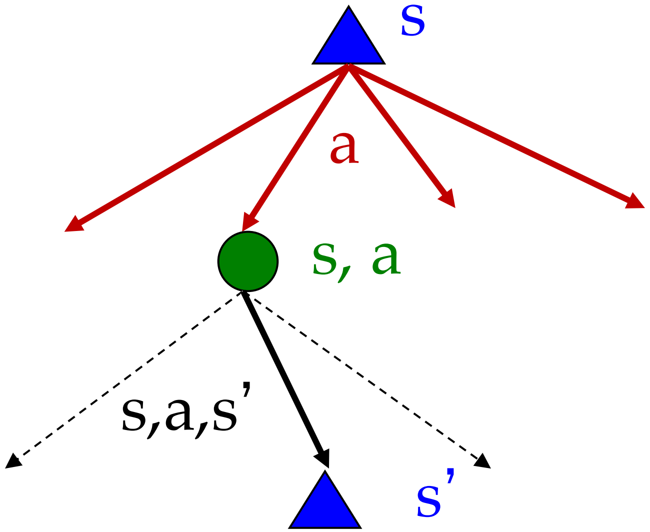
$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

- Important lesson: actions are easier to select from q-values than values!

Problems with Value Iteration

- Value iteration repeats the Bellman updates:

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



- Problem 1: It's slow – $O(S^2A)$ per iteration

- Problem 2: The "max" at each state rarely changes

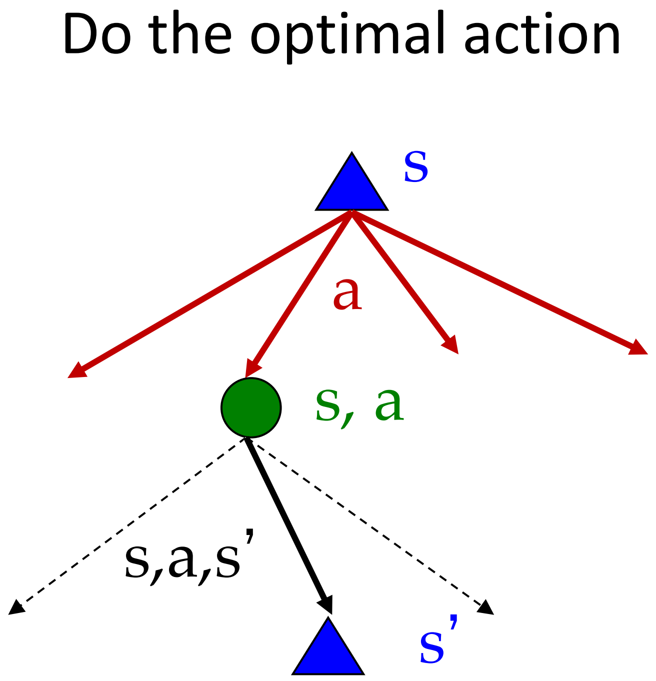
- Problem 3: The policy often converges long before the values

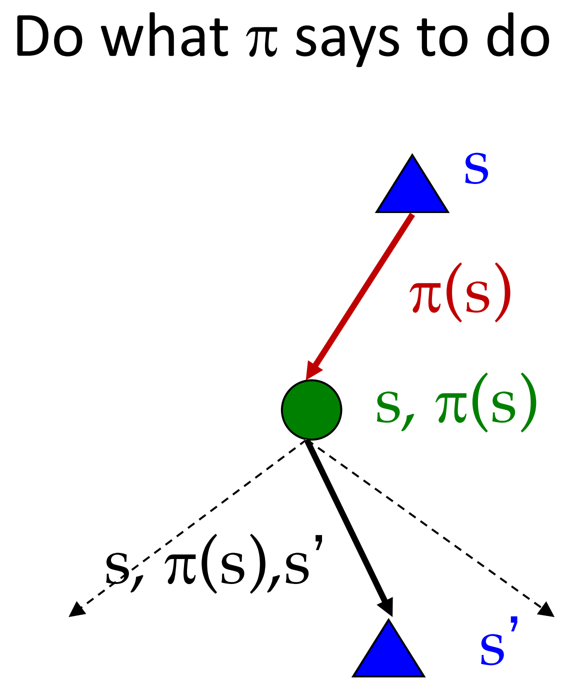
Policy Iteration

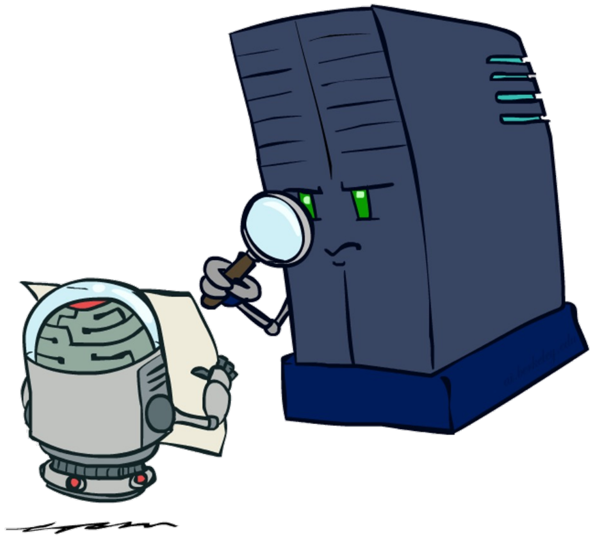
- Alternative approach for optimal values:
  - Step 1: Policy Evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - Step 2: Policy Improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is Policy Iteration
  - It's still optimal!
  - Can converge (much) faster under some conditions

Policy Evaluation: Fixed Policies







- Expectimax trees max over all actions to compute the optimal values
- If we fix 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

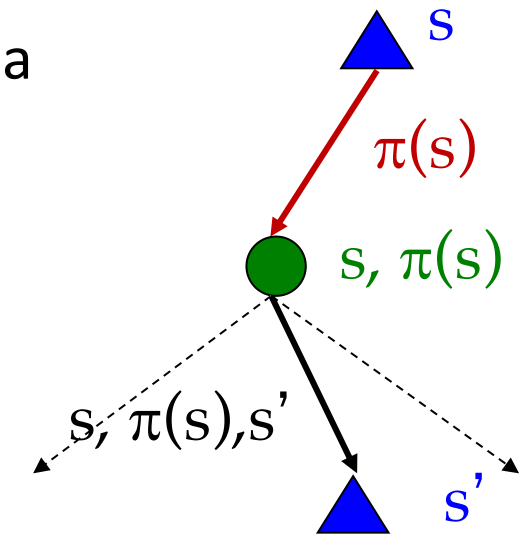
Policy Evaluation: Utilities for a Fixed Policy

- Another basic operation: compute the utility of a state s under a fixed (generally non-optimal) policy

- Define the utility of a state s, under a fixed policy π:

  $V^{\pi}(s)$ = expected total discounted rewards starting in s and following π

- Recursive relation (one-step look-ahead / Bellman equation):



$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Policy Evaluation: Implementation

- How do we calculate the V's for a fixed policy π?

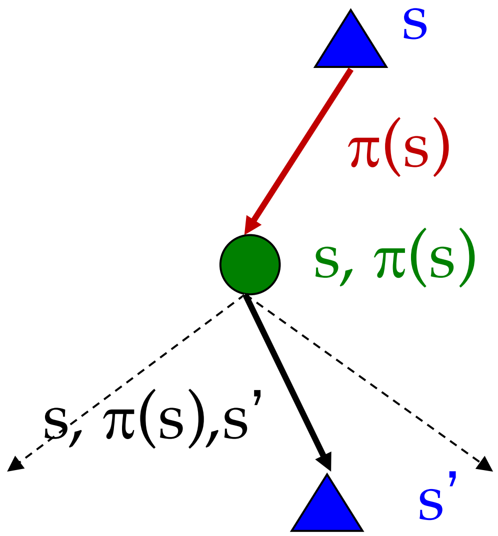
- Idea 1: Turn recursive Bellman equations into updates (like value iteration)

$$V_0^\pi(s) = 0$$

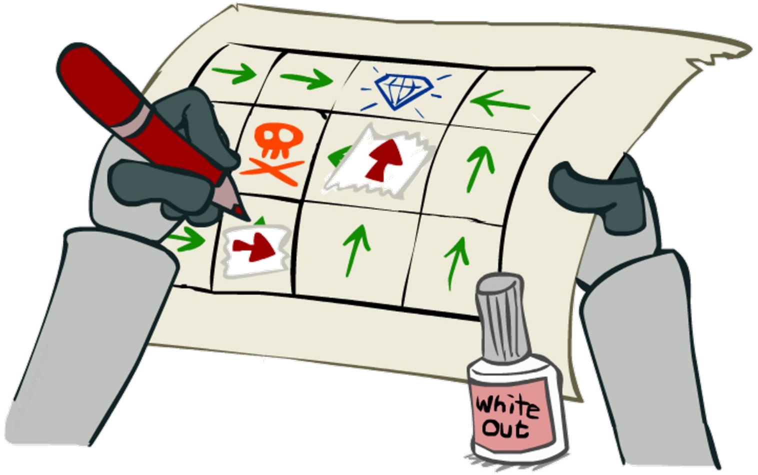
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$

- Efficiency: $O(S^2)$ per iteration

- Idea 2: Without the maxes, the Bellman equations are just a linear system
  - Solve with MATLAB (or your favorite linear system solver)



Policy Iteration



- Evaluation: For fixed current 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 (why? exercise) policy using policy extraction
  - One-step look-ahead:

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

Value Iteration vs. 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

Reinforcement Learning

What Just Happened?

- That wasn't planning, it was learning!
  - Specifically, reinforcement learning
  - There was an MDP, but you couldn't solve it with just computation
  - You needed to actually act to figure it out

- Important ideas in reinforcement learning that came up
  - 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

- What if we didn't know $P(s'|s,a)$ and $R(s,a,s')$?

Value iteration: $$V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \qquad \forall\, s$$

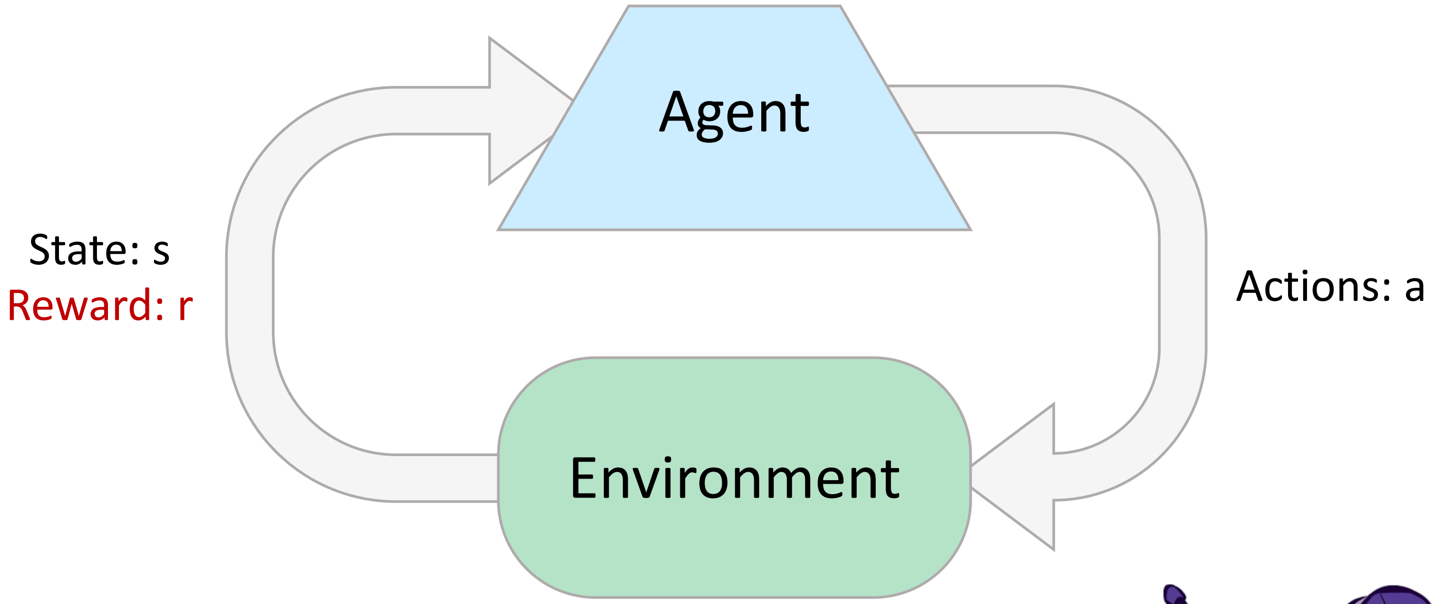
Q-iteration: $$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')], \quad \forall\, s,a$$

Policy extraction: $$\pi_V(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')], \qquad \forall\, s$$

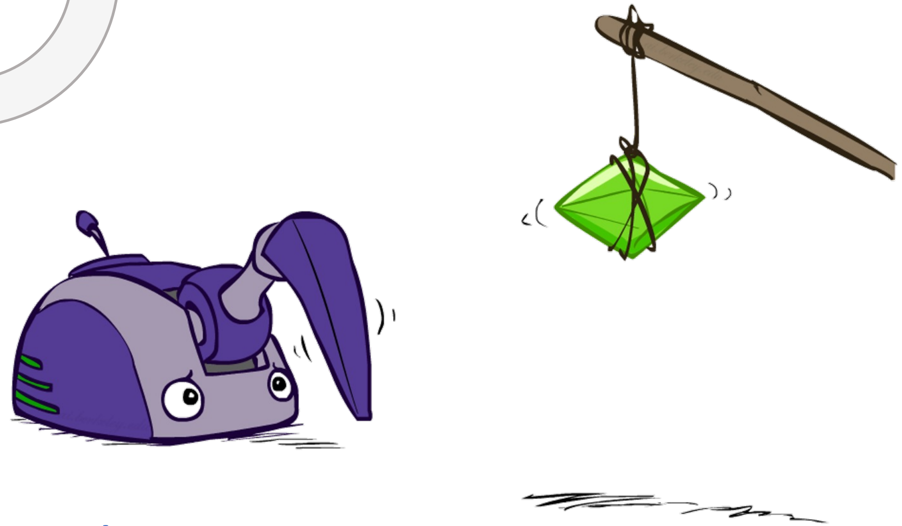
Policy evaluation: $$V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')], \qquad \forall\, s$$

Policy improvement: $$\pi_{new}(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')], \qquad \forall\, s$$

Reinforcement Learning 2



- 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!

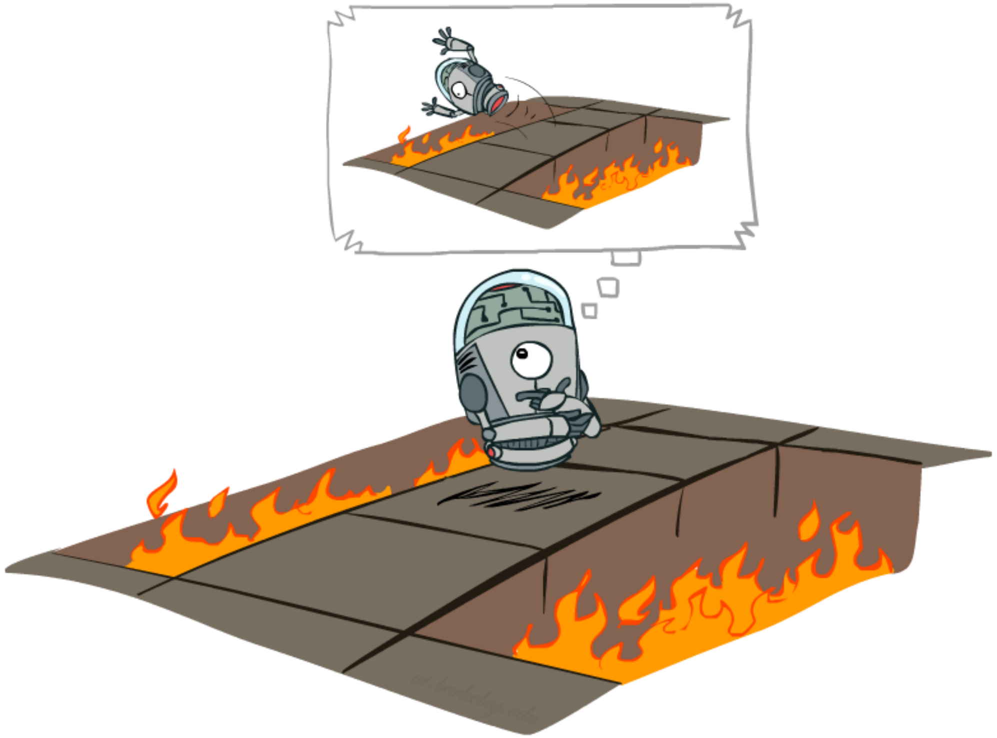


Reinforcement Learning 3

- Still assume a Markov decision process (MDP):
  - A set of states $s \in S$
  - A set of actions (per state) A
  - A 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 which states are good or what the actions do
  - Must actually try actions and states out to learn

Offline (MDPs) vs. Online (RL)



Offline Solution

Online Learning

Reinforcement Learning -- Overview

- Passive Reinforcement Learning (= how to learn from experiences)
  - Model-based Passive RL
    - Learn the MDP model from experiences, then solve the MDP
  - Model-free Passive RL
    - Forego learning the MDP model, directly learn V or Q:
      - Value learning – learns value of a fixed policy; 2 approaches: Direct Evaluation & TD Learning
      - Q learning – learns Q values of the optimal policy (uses a Q version of TD Learning)
- Active Reinforcement Learning (= agent also needs to decide how to collect experiences)
  - Key challenges:
    - How to efficiently explore?
    - How to trade off exploration <> exploitation
  - Applies to both model-based and model-free.
    we'll cover only in context of Q-learning

Model-Based Reinforcement Learning

- Model-Based Idea:
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct

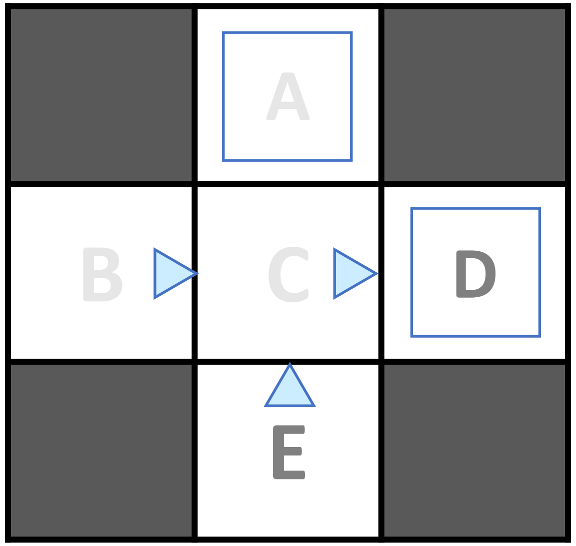
- Step 1: Learn empirical MDP model
  - Count outcomes s’ for each s, a
  - Normalize to give an estimate of $\hat{T}(s, a, s')$
  - Discover each $\hat{R}(s, a, s')$ when we experience (s, a, s’)

- Step 2: Solve the learned MDP
  - For example, use value iteration, as before

(and repeat as needed)

Example: Model-Based RL

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

$\hat{T}(s, a, s')$

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

$\hat{R}(s, a, s')$

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

Analogy: Expected Age

Goal: Compute expected age of students

Known P(A)

$$E[A] = \sum_a P(a) \cdot a \qquad = 0.35 \times 20 + \ldots$$

Without P(A), instead collect samples [a_1, a_2, … a_N]

Unknown P(A): "Model Based"

$$\hat{P}(a) = \frac{\text{num}(a)}{N}$$

$$E[A] \approx \sum_a \hat{P}(a) \cdot a$$

Why does this work? Because eventually you learn the right model.

Unknown P(A): "Model Free"

$$E[A] \approx \frac{1}{N} \sum_i a_i$$

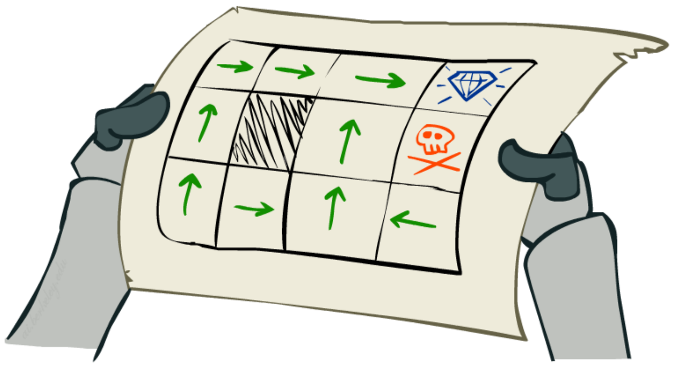
Why does this work? Because samples appear with the right frequencies.

Reinforcement Learning -- Overview

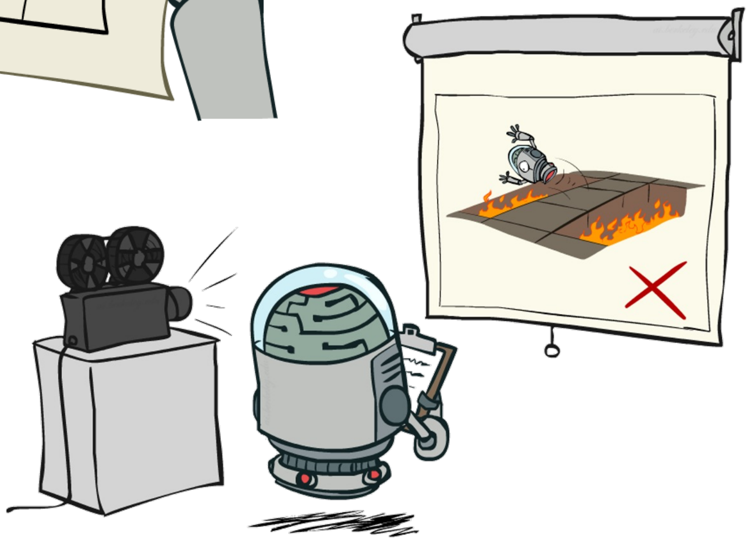
- Passive Reinforcement Learning (= how to learn from experiences)
  - Model-based Passive RL
    - Learn the MDP model from experiences, then solve the MDP
  - Model-free Passive RL
    - Forego learning the MDP model, directly learn V or Q:
      - Value learning – learns value of a fixed policy; 2 approaches: Direct Evaluation & TD Learning
      - Q learning – learns Q values of the optimal policy (uses a Q version of TD Learning)
- Active Reinforcement Learning (= agent also needs to decide how to collect experiences)
  - Key challenges:
    - How to efficiently explore?
    - How to trade off exploration <> exploitation
  - Applies to both model-based and model-free.
    we'll cover only in context of Q-learning

Passive Model-Free Reinforcement Learning

- Simplified task: policy evaluation
  - Input: a fixed policy $\pi(s)$
  - 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

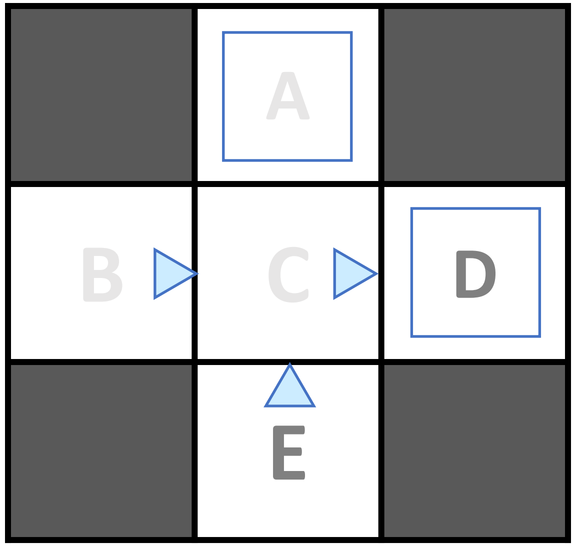
- Goal: Compute values for each state under π

- 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

- This is called direct evaluation

Example: Direct Evaluation

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

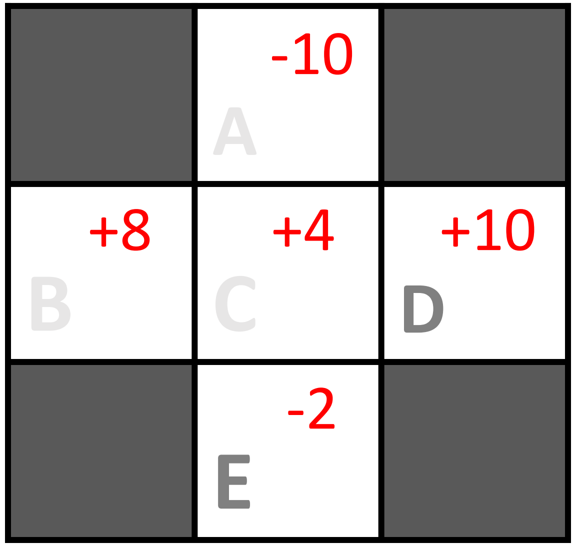
Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Output Values



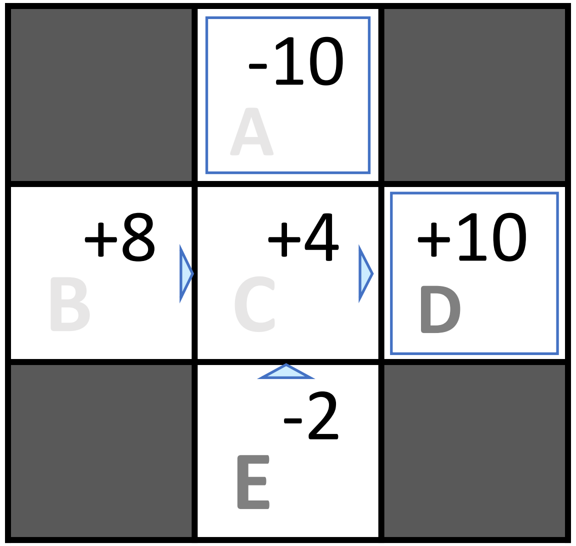
If B and E both go to C under this policy, how can their values be different?

Problems with Direct Evaluation

- What's good about direct evaluation?
  - It's easy to understand
  - It doesn't require any knowledge of T, R
  - It eventually computes the correct average values, using just sample transitions

- What bad about it?
  - It wastes information about state connections
  - Each state must be learned separately
  - So, it takes a long time to learn

Output Values



If B and E both go to C under this policy, how can their values be different?

Reinforcement Learning -- Overview

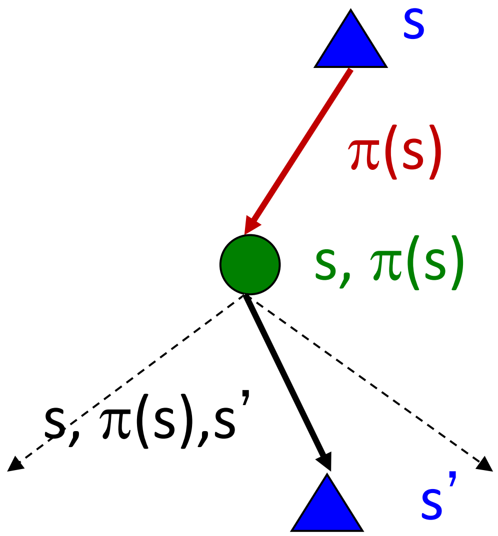
- Passive Reinforcement Learning (= how to learn from experiences)
  - Model-based Passive RL
    - Learn the MDP model from experiences, then solve the MDP
  - Model-free Passive RL
    - Forego learning the MDP model, directly learn V or Q:
      - Value learning – learns value of a fixed policy; 2 approaches: Direct Evaluation & **TD Learning**
      - Q learning – learns Q values of the optimal policy (uses a Q version of TD Learning)
- Active Reinforcement Learning (= agent also needs to decide how to collect experiences)
  - Key challenges:
    - How to efficiently explore?
    - How to trade off exploration <> exploitation
  - Applies to both model-based and model-free.
    we'll cover only in context of Q-learning

Why Not Use Policy Evaluation?

- Simplified Bellman updates calculate V for a fixed policy:
  - Each round, replace V with a one-step-look-ahead layer over V

$$V_0^\pi(s) = 0$$

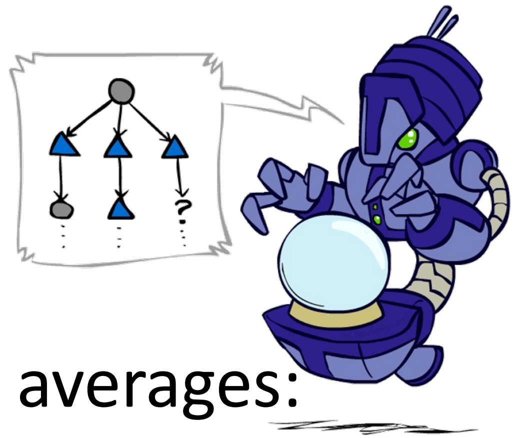
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]$$



  - This approach fully exploited the connections between the states
  - Unfortunately, we need T and R to do it!

- Key question: how can we do this update to V without knowing T and R?
  - In other words, how do we take a weighted average without knowing the weights?

Sample-Based Policy Evaluation?



- We want to improve our estimate of V by computing these averages:

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

- Idea: Take samples of outcomes s' (by doing the action!) and average

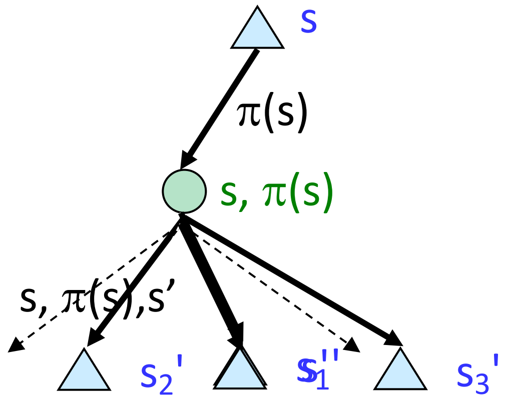
$$sample_1 = R(s,\pi(s),s_1') + \gamma V_k^{\pi}(s_1')$$

$$sample_2 = R(s,\pi(s),s_2') + \gamma V_k^{\pi}(s_2')$$

$$\ldots$$

$$sample_n = R(s,\pi(s),s_n') + \gamma V_k^{\pi}(s_n')$$

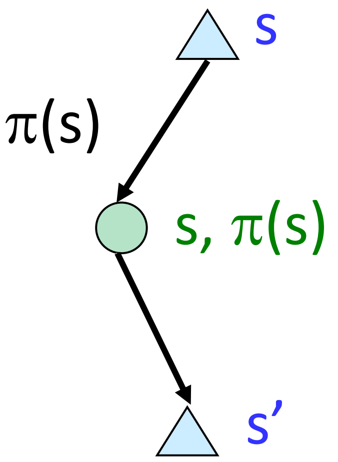
$$V_{k+1}^{\pi}(s) \leftarrow \frac{1}{n}\sum_i sample_i$$



Almost! But we can't rewind time to get sample after sample from state s

Temporal Difference Value Learning

- 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

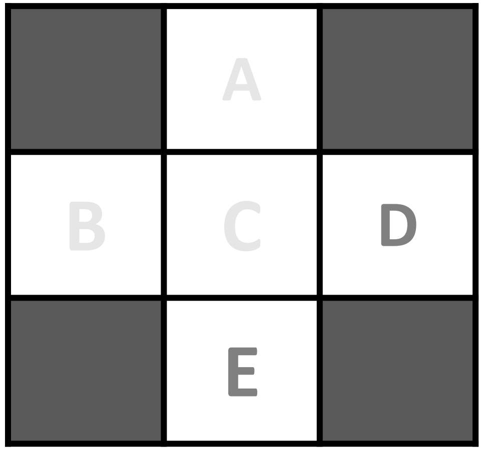
Sample of V(s): $sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$

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

Same update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(sample - V^{\pi}(s))$

Example: Temporal Difference Value Learning

States

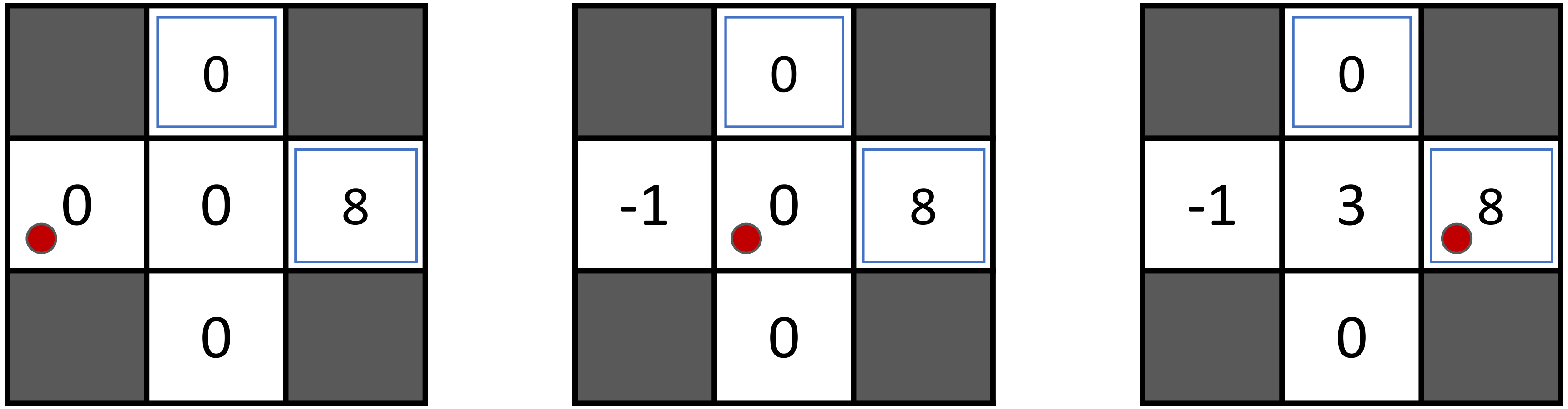


Assume: $\gamma = 1$, $\alpha = 1/2$

Observed Transitions

B, east, C, -2

C, east, D, -2



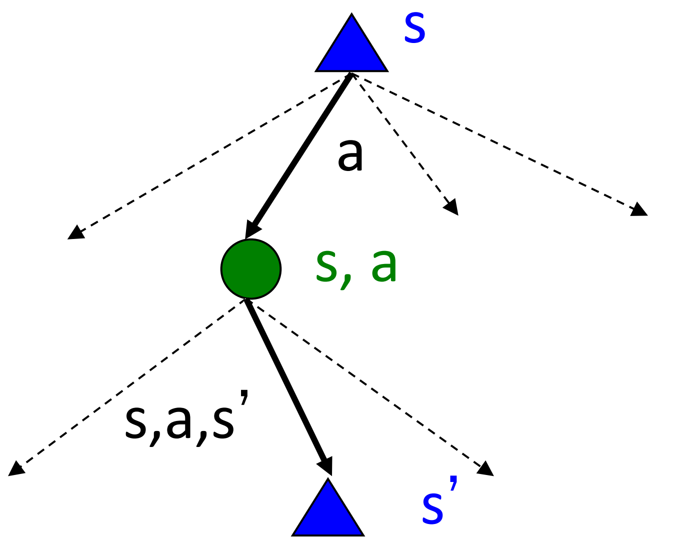
$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

Problems with TD Value Learning

- TD value leaning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
- However, if we want to turn values into a (new) policy, we're sunk:

$$\pi(s) = \arg\max_a Q(s,a)$$

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



- Idea: learn Q-values, not values
- Makes action selection model-free too!

Reinforcement Learning -- Overview

- Passive Reinforcement Learning (= how to learn from experiences)
  - Model-based Passive RL
    - Learn the MDP model from experiences, then solve the MDP
  - Model-free Passive RL
    - Forego learning the MDP model, directly learn V or Q:
      - Value learning – learns value of a fixed policy; 2 approaches: Direct Evaluation & TD Learning
      - Q learning – learns Q values of the optimal policy (uses a Q version of TD Learning)
- Active Reinforcement Learning (= agent also needs to decide how to collect experiences)
  - Key challenges:
    - How to efficiently explore?
    - How to trade off exploration <> exploitation
  - Applies to both model-based and model-free.
    we'll cover only in context of Q-learning

Q-Value Iteration

- Value iteration: find successive (depth-limited) values
  - Start with $V_0(s) = 0$, which we know is right
  - Given V_k, calculate the depth k+1 values for all states:

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

- But Q-values are more useful, so compute them instead
  - Start with $Q_0(s,a) = 0$, which we know is right
  - Given Q_k, calculate the depth k+1 q-values for all q-states:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

Q-Learning

- Q-Learning: sample-based Q-value iteration

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

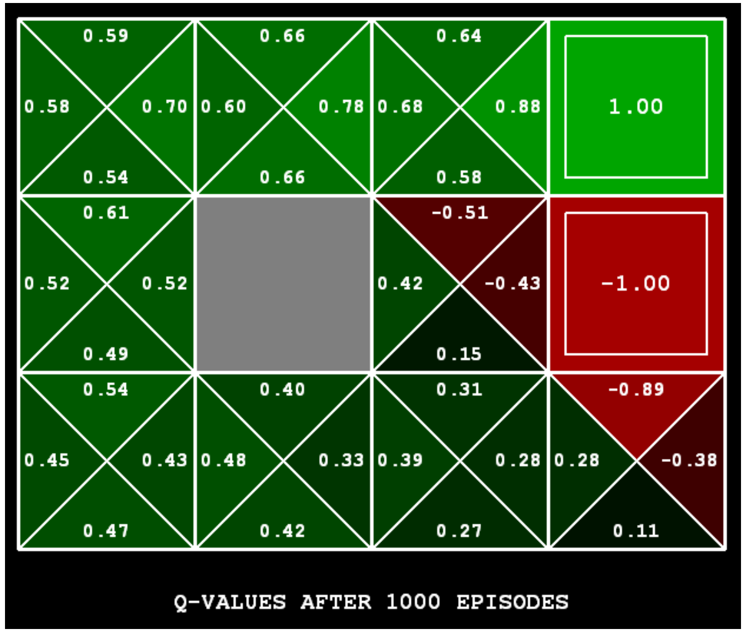
- Learn Q(s,a) values as you go
  - Receive a sample (s,a,s',r)
  - Consider your old estimate: $Q(s,a)$
  - Consider your new sample estimate:

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

no longer policy evaluation!

  - Incorporate the new estimate into a running average:

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



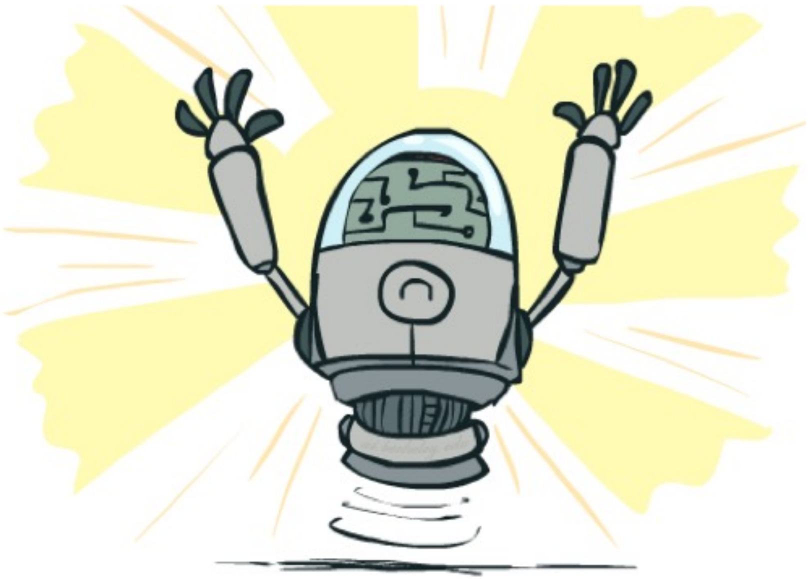
[Demo: Q-learning – gridworld (L10D2)]
[Demo: Q-learning – crawler (L10D3)]

Q-Learning Properties

- Amazing result: 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 (!)

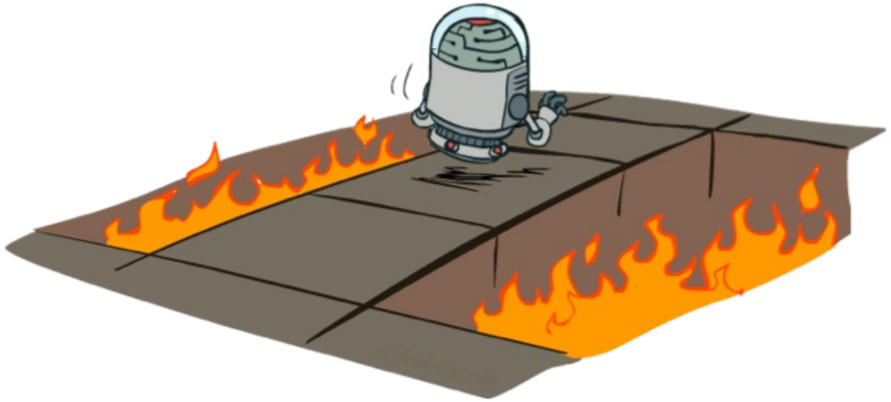


Reinforcement Learning -- Overview

- Passive Reinforcement Learning (= how to learn from experiences)
  - Model-based Passive RL
    - Learn the MDP model from experiences, then solve the MDP
  - Model-free Passive RL
    - Forego learning the MDP model, directly learn V or Q:
      - Value learning – learns value of a fixed policy; 2 approaches: Direct Evaluation & TD Learning
      - Q learning – learns Q values of the optimal policy (uses a Q version of TD Learning)
- Active Reinforcement Learning (= agent also needs to decide how to collect experiences)
  - Key challenges:
    - How to efficiently explore?
    - How to trade off exploration <> exploitation
  - Applies to both model-based and model-free.
    we'll cover only in context of Q-learning

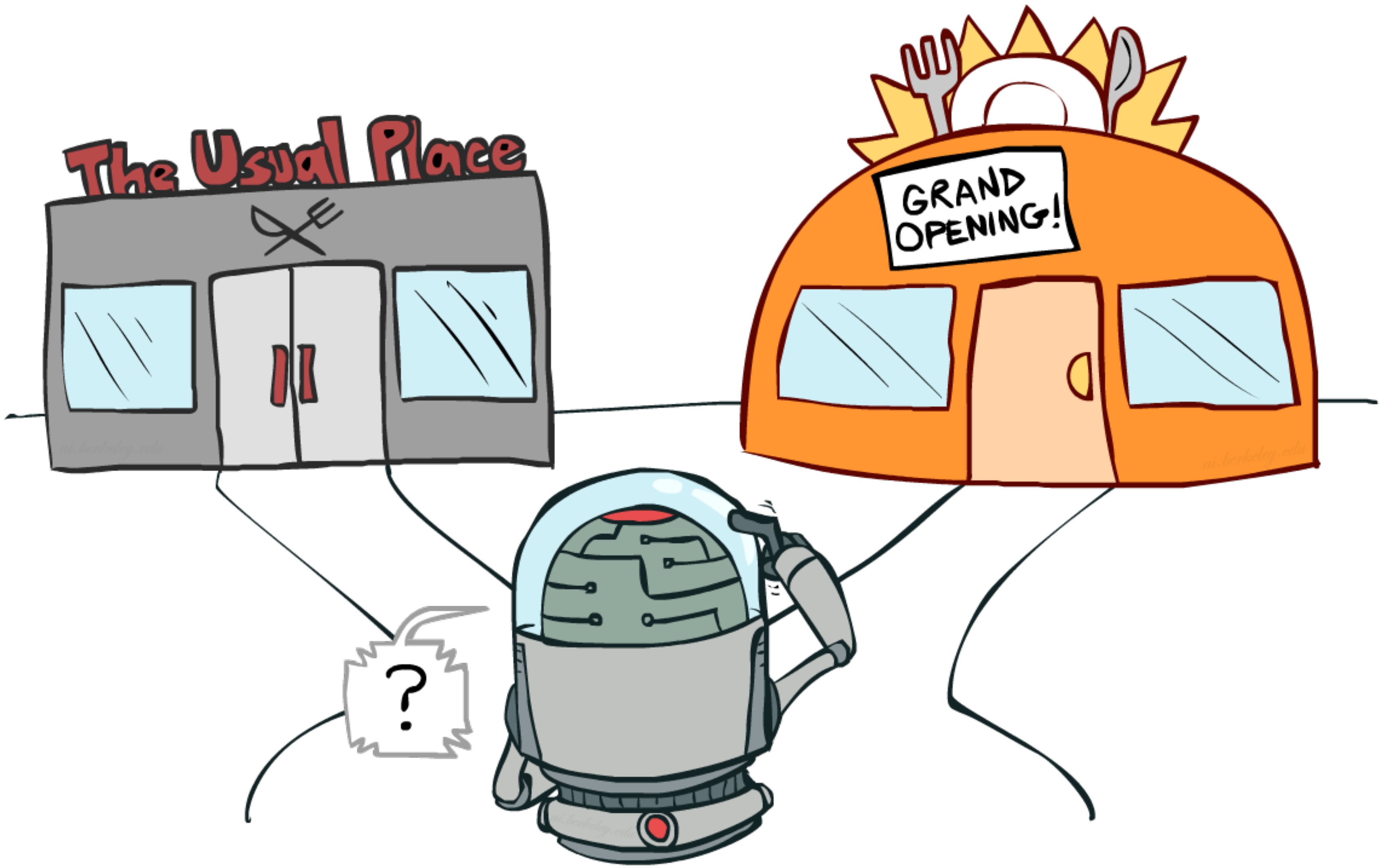
Active Reinforcement Learning

- Full reinforcement learning: optimal policies (like value iteration)
  - You don't know the transitions T(s,a,s')
  - You don't know the rewards R(s,a,s')
  - You choose the actions now
  - Goal: learn the optimal policy / values



- In this case:
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens…

Exploration vs. Exploitation



How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
    - With (small) probability ε, act randomly
    - With (large) probability 1-ε, act on current policy
  - Problems with random actions?
    - You do eventually explore the space, but keep thrashing around once learning is done
    - One solution: lower ε over time
    - Another solution: exploration functions

[Demo: Q-learning – manual exploration – bridge grid (L10D5)]
[Demo: Q-learning – epsilon-greedy -- crawler (L10D3)]

Exploration Functions

A commonly used 'exploration function' is $f(u,n) = u + c\sqrt{\log(1/\delta)/n}$, which is derived by Chernoff-Hoeffding inequality and δ is confidence level

- When to explore?
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is not (yet) established, eventually stop exploring
- Exploration function
  - Takes a value estimate u and a visit count n, and returns an optimistic utility, e.g. $f(u,n) = u + k/n$

    Regular Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} Q(s',a')$

  - Modified Q-Update: $Q(s,a) \leftarrow_\alpha R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$

  - Action selection: Use $a \leftarrow \text{argmax}_a\, Q(s,a)$
  - Note: this propagates the "bonus" back to states that lead to unknown states as well!

[Demo: exploration – Q-learning – crawler – exploration function (L10D4)]

The Story So Far: MDPs and RL

Known MDP: Offline Solution

Goal	Technique
Compute V*, Q*, π*	Value / policy iteration
Evaluate a fixed policy π	Policy evaluation

Unknown MDP: Model-Based

Goal	Technique
Compute V*, Q*, π*	VI/PI on approx. MDP
Evaluate a fixed policy π	PE on approx. MDP

Unknown MDP: Model-Free

Goal	Technique
Compute V*, Q*, π*	Q-learning
Evaluate a fixed policy π	Value Learning

Multi-armed Bandits

Setting: Finite-armed stochastic bandits

items/products/movies/news/…

CTR/profit/…

- There are L arms
  - Each arm a has an unknown reward distribution v_a with unknown mean $\alpha(a)$
  - The best arm is $a^* = \mathrm{argmax}_a \alpha(a)$

- At each time t
  - The learning agent selects an arm a_t
  - Observes the reward $X_{a_t,t} \sim v_{a_t}$

bandit feedback

Objective

- Maximize the expected cumulative reward in T rounds

$$\mathbb{E}\left[\sum_{t=1}^{T} \alpha(a_t)\right]$$

- Minimize the regret in T rounds

$$R(T) = T \cdot \alpha(a^*) - \mathbb{E}\left[\sum_{t=1}^{T} \alpha(a_t)\right]$$

- Balance the trade-off between exploration and exploitation
  - Exploitation: Select arms that yield good results so far
  - Exploration: Select arms that have not been tried much before
- Smaller order of T in $R(T)$ is better

UCB – Upper confidence bound [Auer et al.(2002)]

- With high probability

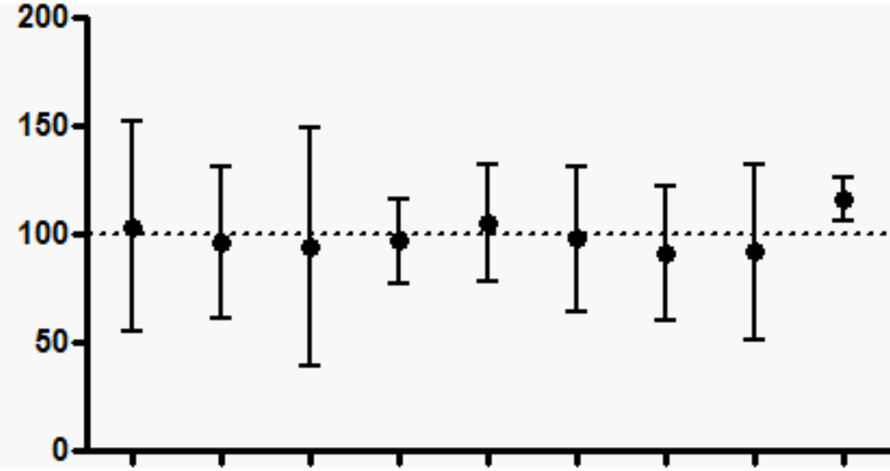
$$\alpha_a \epsilon \left[\hat{\alpha}_a(t) - \sqrt{\frac{2\log t}{T_a(t)}}, \hat{\alpha}_a(t) + \sqrt{\frac{2\log t}{T_a(t)}}\right]$$

sample mean

round t

Hoeffding's inequality

selection times of arm a till round t



- Principle: optimism in face of uncertainty
- UCB policy:

$$a_t = \text{argmax}_a\ \hat{\alpha}_a + \sqrt{\frac{2\log t}{T_a(t)}}$$

exploration

exploitation

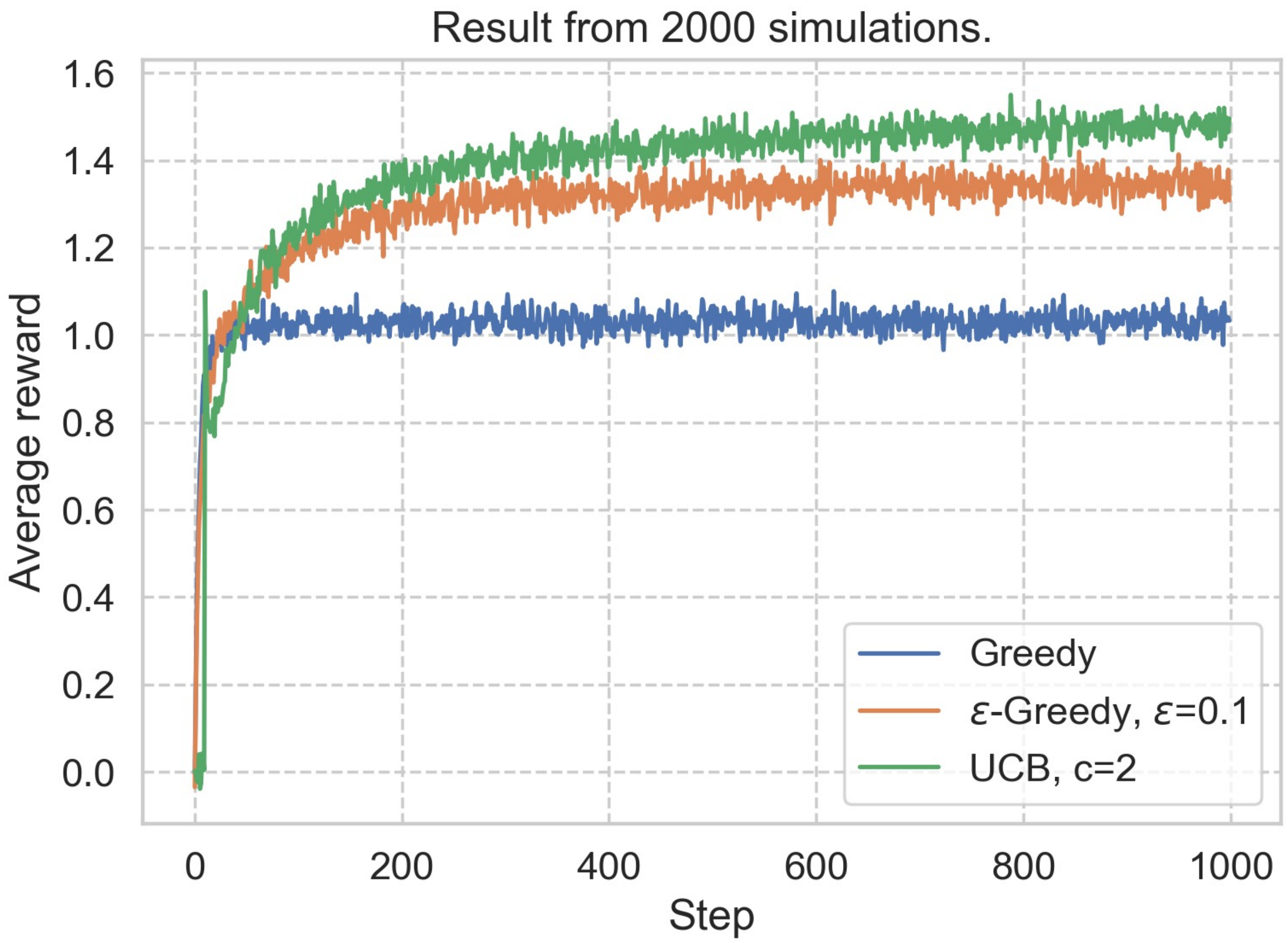
UCB – Upper confidence bound 2

- Regret

$$R(T) = O\left(\frac{L}{\Delta}\log T\right)$$

- Proof sketch
  - Under good event (w/ high probability)
  - If arm a is pulled, then
$$\alpha(a^*) \leq \mathrm{UCB}_{a^*} \leq \mathrm{UCB}_a \leq \alpha(a) + 2\ \mathrm{radius}_a$$
  - $\Longrightarrow \sqrt{\frac{2\log t}{T_a(t)}} = \mathrm{radius}_a \geq \frac{\alpha(a^*) - \alpha(a)}{2}$
  - $\Longrightarrow T_a(t) \leq \frac{8\log t}{\Delta_a^2}$

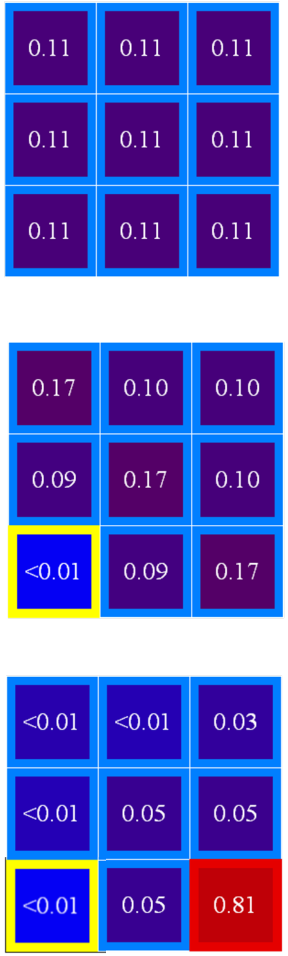
UCB – Upper confidence bound 3



Bayes Nets: Probabilistic Models

Uncertainty

- General situation:
  - **Observed variables (evidence)**: Agent knows certain things about the state of the world (e.g., sensor readings or symptoms)
  - **Unobserved variables**: Agent needs to reason about other aspects (e.g. where an object is or what disease is present)
  - **Model**: Agent knows something about how the known variables relate to the unknown variables
- Probabilistic reasoning gives us a framework for managing our beliefs and knowledge



Probabilistic Inference

- Probabilistic inference: compute a desired probability from other known probabilities (e.g. conditional from joint)

- We generally compute conditional probabilities
  - P(on time | no reported accidents) = 0.90
  - These represent the agent's *beliefs* given the evidence

- Probabilities change with new evidence:
  - P(on time | no accidents, 5 a.m.) = 0.95
  - P(on time | no accidents, 5 a.m., raining) = 0.80
  - Observing new evidence causes *beliefs to be updated*

Inference by Enumeration

- General case:
  - Evidence variables: $E_1 \dots E_k = e_1 \dots e_k$
  - Query* variable: Q
  - Hidden variables: $H_1 \dots H_r$

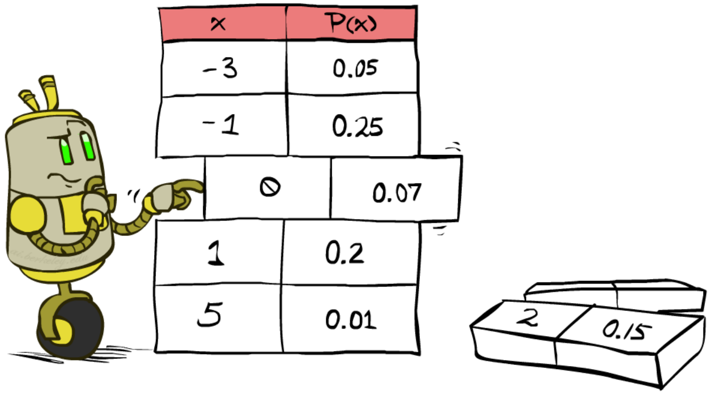
$\Big\} \; X_1, X_2, \dots X_n$ *All variables*

- We want:

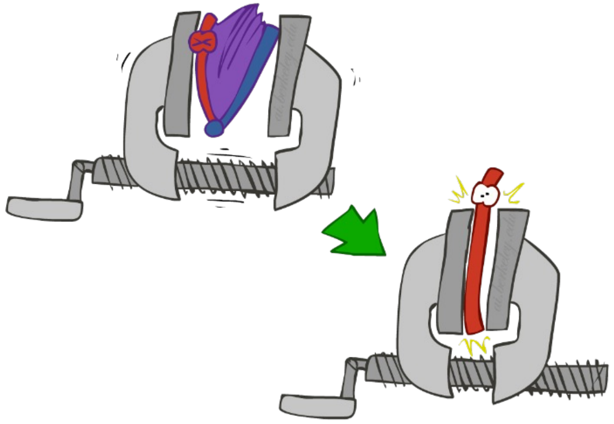
** Works fine with multiple query variables, too*

$$P(Q|e_1 \dots e_k)$$

- Step 1: Select the entries consistent with the evidence



- Step 2: Sum out H to get joint of Query and evidence



$$P(Q, e_1 \dots e_k) = \sum_{h_1 \dots h_r} \underbrace{P(Q, h_1 \dots h_r, e_1 \dots e_k)}_{X_1, X_2, \dots X_n}$$

- Step 3: Normalize

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

$$Z = \sum_q P(Q, e_1 \cdots e_k)$$

$$P(Q|e_1 \cdots e_k) = \frac{1}{Z} P(Q, e_1 \cdots e_k)$$

Answer Any Query from Joint Distributions

- Two tools to go from joint to query
- Joint: $P(H_1, H_2, Q, E)$
- Query: $P(Q \mid e)$

1. Definition of conditional probability

$$P(Q|e) = \frac{P(Q,e)}{P(e)}$$

2. Law of total probability (marginalization, summing out)

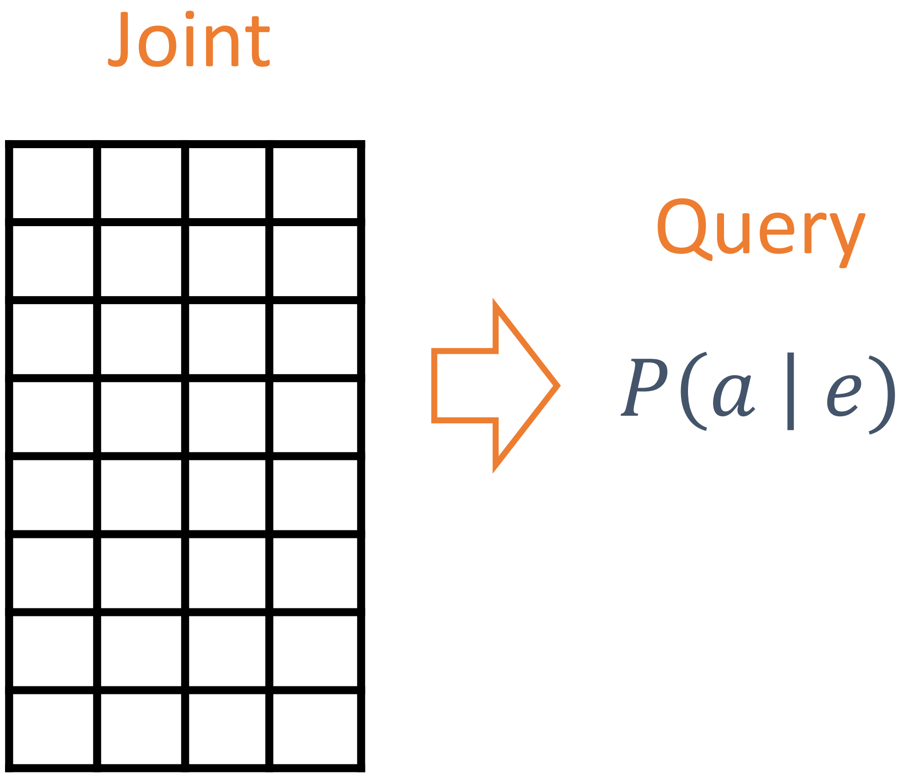
$$P(Q,e) = \sum_{h_1} \sum_{h_2} P(h_1, h_2, Q, e)$$

$$P(e) = \sum_{q} \sum_{h_1} \sum_{h_2} P(h_1, h_2, q, e)$$

Only need to compute $P(Q, e)$ then normalize

Answer Any Query from Joint Distributions

- Joint distributions are the best!
- Problems with joints
  - We aren't given the joint table
  - Usually some set of conditional probability tables
- Problems with inference by enumeration
  - Worst-case time complexity $O(d^n)$
  - Space complexity $O(d^n)$ to store the joint distribution

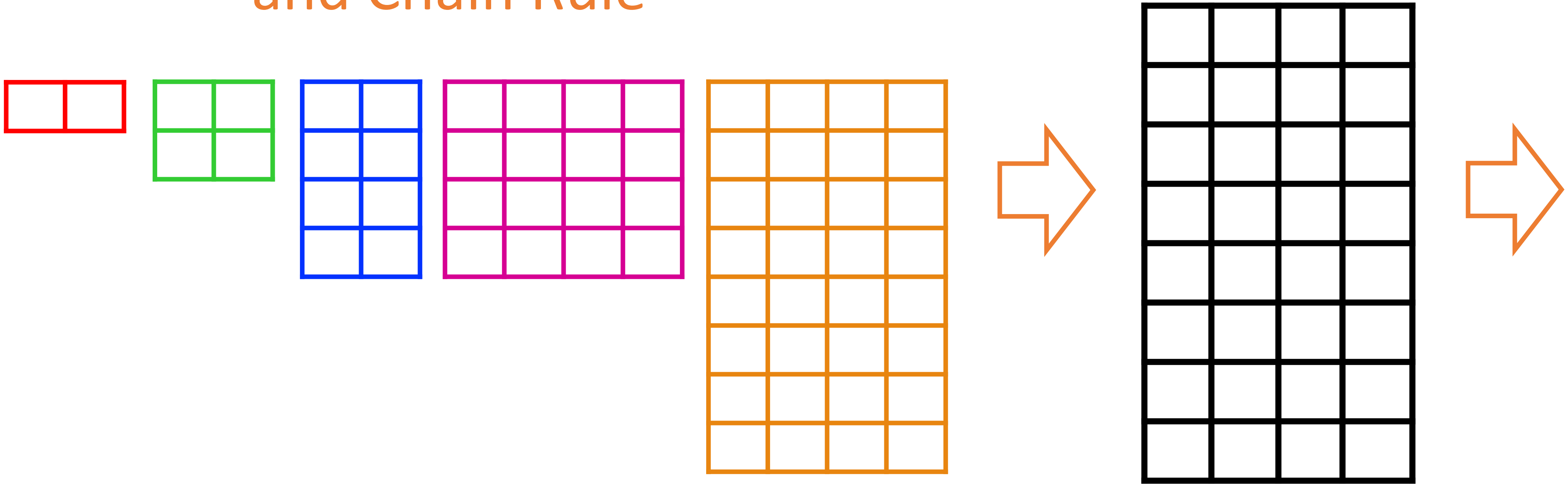


Build Joint Distribution Using Chain Rule

Conditional Probability Tables and Chain Rule

Joint

Query

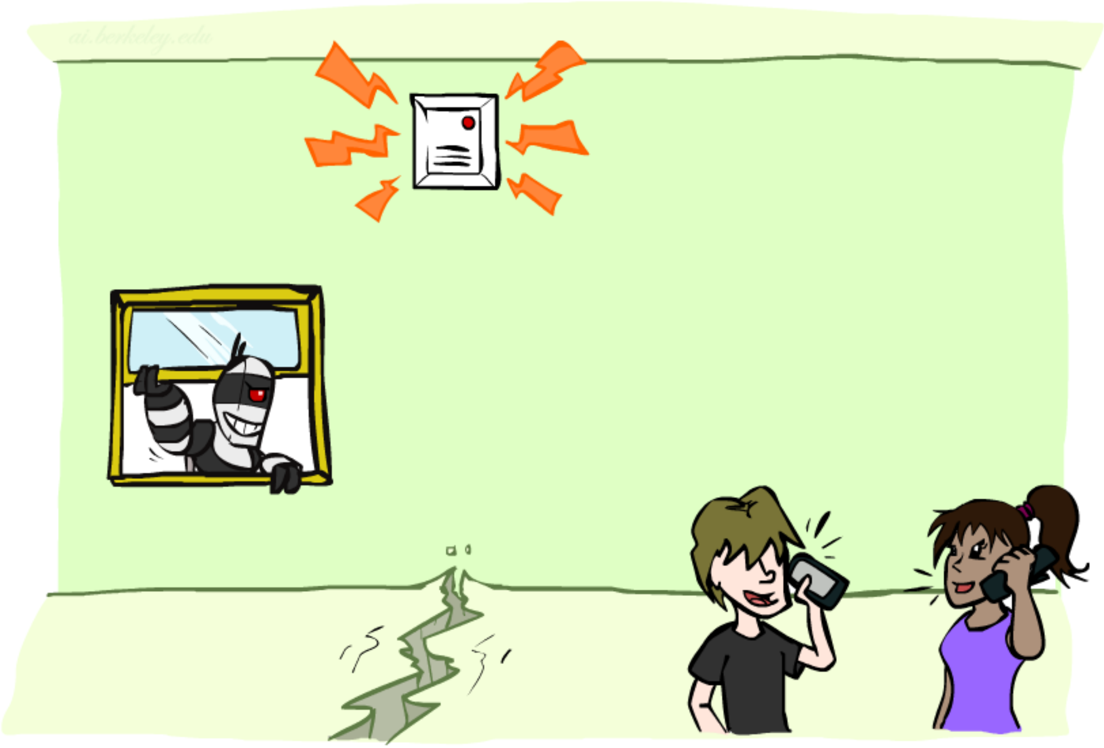


$P(a \mid e)$

$P(A)\ P(B|A)\ P(C|A,B)\ P(D|A,B,C)\ P(E|A,B,C,D)$

Quiz

- Variables
  - B: Burglary
  - A: Alarm goes off
  - M: Mary calls
  - J: John calls
  - E: Earthquake!



How many different ways can we write the chain rule?

A. 1

B. 5

C. $5 \text{ } choose \text{ } 5$

D. $5!$

E. 5^5

Answer Any Query from Condition Probability Tables

- Bayes' rule as an example
- Given: $P(E|Q),\ P(Q)$ Query: $P(Q \mid e)$

1. Construct the joint distribution
   1. Product Rule or Chain Rule

$$P(E,Q) = P(E|Q)P(Q)$$

2. Answer query from joint
   1. Definition of conditional probability

$$P(Q \mid e) = \frac{P(e,Q)}{P(e)}$$

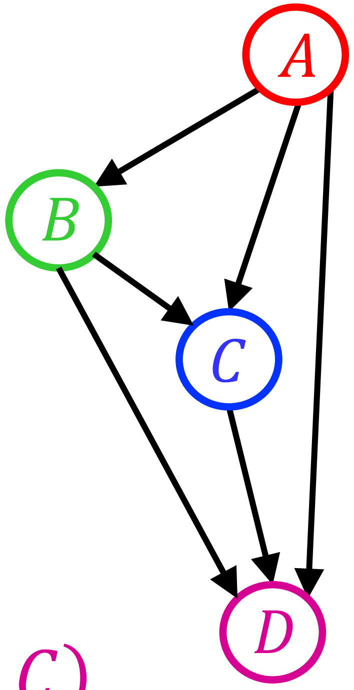
   2. Law of total probability (marginalization, summing out)

$$P(Q \mid e) = \frac{P(e,Q)}{\sum_q P(e,q)}$$

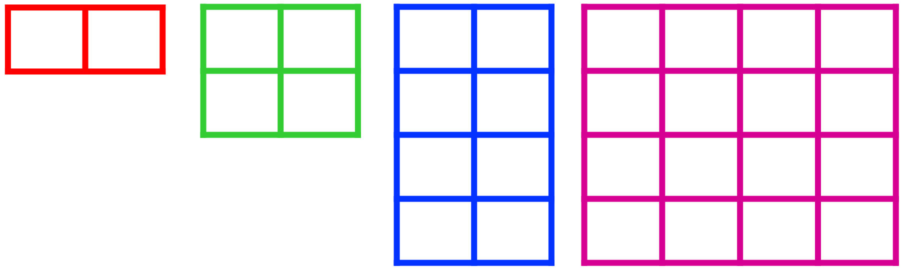
Only need to compute $P(e,Q)$ then normalize

Bayesian Networks

Bayes net



- One node per random variable, DAG
- One conditional probability table (CPT) per node: P(node | *Parents*(node))

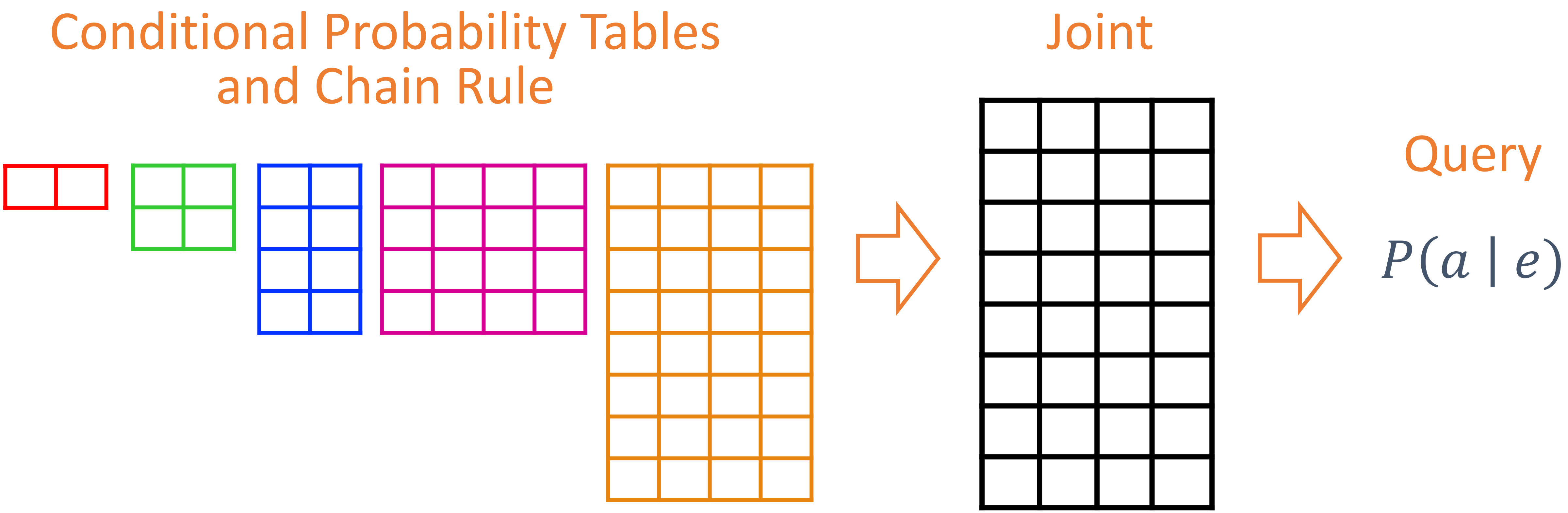


$$P(A, B, C, D) = P(A)\, P(B|A)\, P(C|A, B)\, P(D|A, B, C)$$

Encode joint distributions as product of conditional distributions on each variable

$$P(X_1, \ldots, X_N) = \prod_i P(X_i|\, Parents(X_i))$$

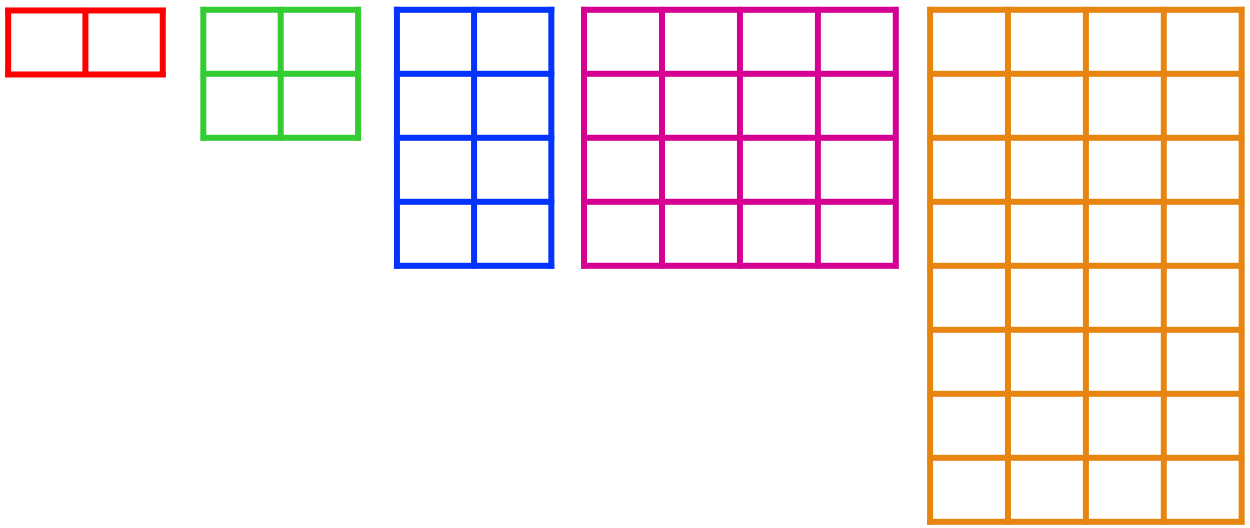
Answer Any Query from Condition Probability Tables



$P(A)\ P(B|A)\ P(C|A,B)\ P(D|A,B,C)\ P(E|A,B,C,D)$

Answer Any Query from Condition Probability Tables 2

Conditional Probability Tables and Chain Rule

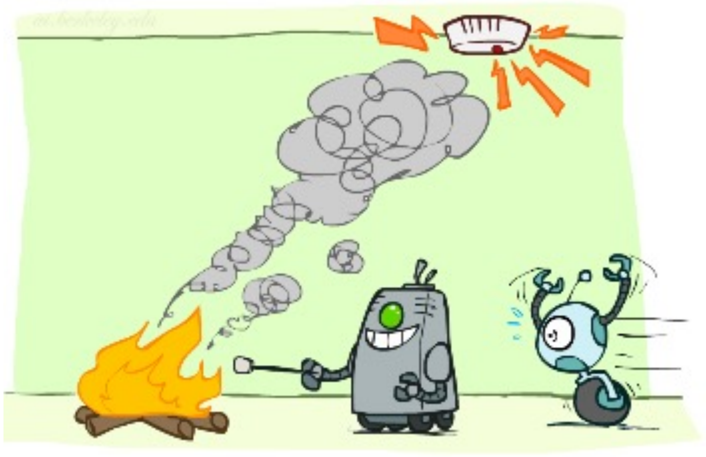


$P(A)\ P(B|A)\ P(C|A,B)\ P(D|A,B,C)\ P(E|A,B,C,D)$

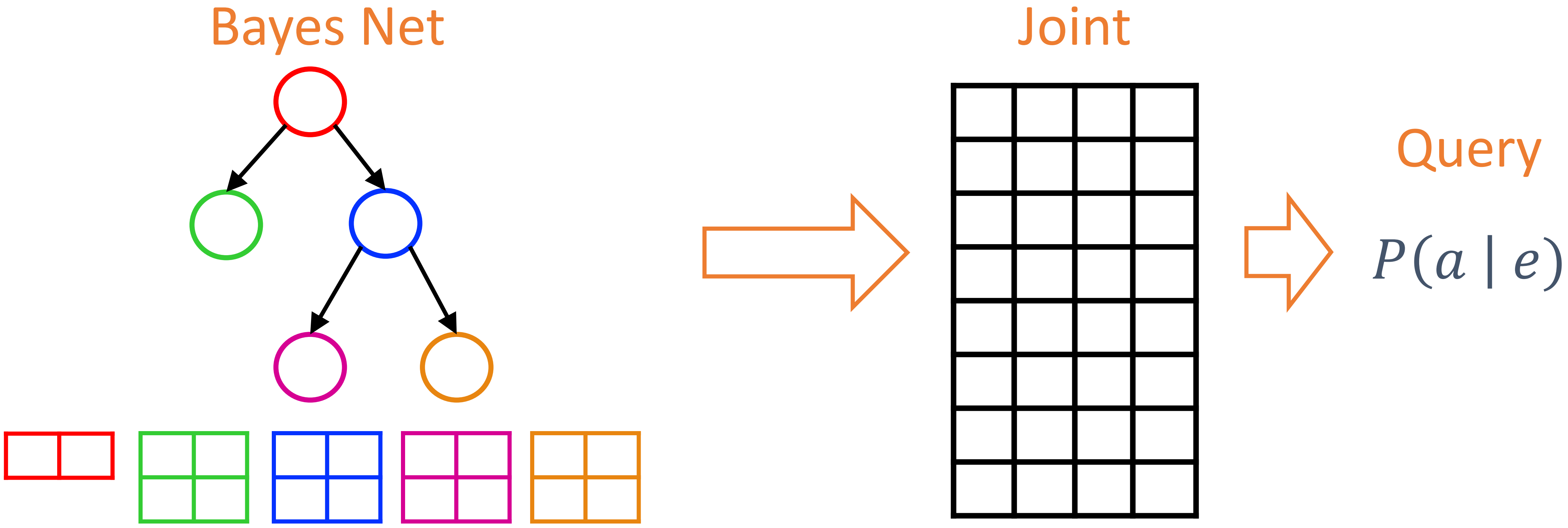
- Problems
  - Huge
    - n variables with d values
    - d^n entries
  - We aren't given the right tables

Do We Need the Full Chain Rule?

- Binary random variables
  - Fire
  - Smoke
  - Alarm



Answer Any Query from Condition Probability Tables

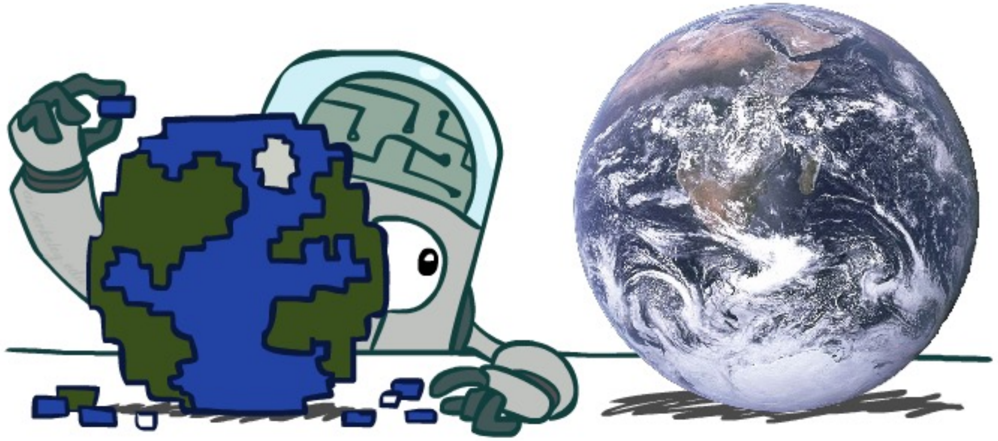


$P(A)$ $P(B|A)$ $P(C|A)$ $P(D|C)$ $P(E|C)$

$$P(X_1, \dots, X_N) = \prod_i P(X_i|\, Parents(X_i))$$

Probabilistic Models

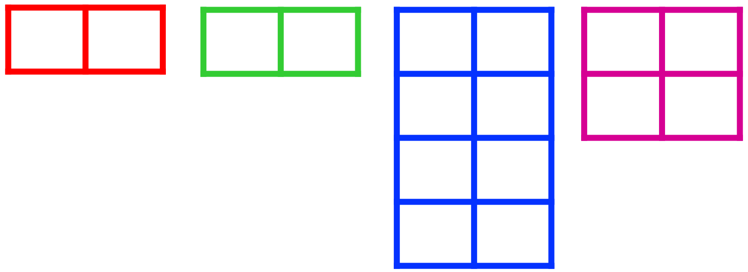
- Models describe how (a portion of) the world works
- Models are always simplifications
  - May not account for every variable
  - May not account for all interactions between variables
  - "All models are wrong; but some are useful."
    – George E. P. Box
- What do we do with probabilistic models?
  - We (or our agents) need to reason about unknown variables, given evidence
  - Example: explanation (diagnostic reasoning)
  - Example: prediction (causal reasoning)
  - Example: value of information

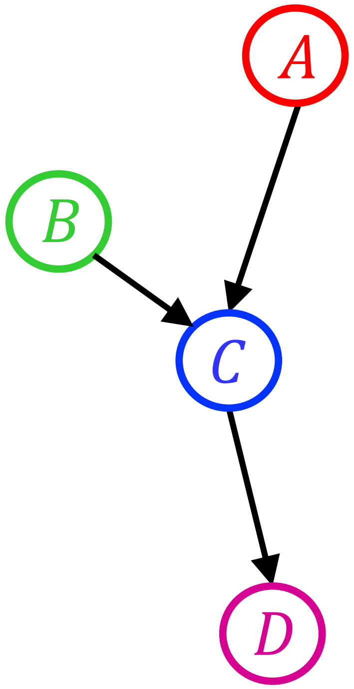


(General) Bayesian Networks

Bayes net

- One node per random variable, DAG
- One conditional probability table (CPT) per node: P(node | *Parents*(node))





$$P(A, B, C, D) = P(A)\, P(B)\, P(C|A, B)\, P(D|C)$$

Encode joint distributions as product of conditional distributions on each variable

$$P(X_1, \dots, X_N) = \prod_i P(X_i |\, Parents(X_i))$$

Conditional Independence

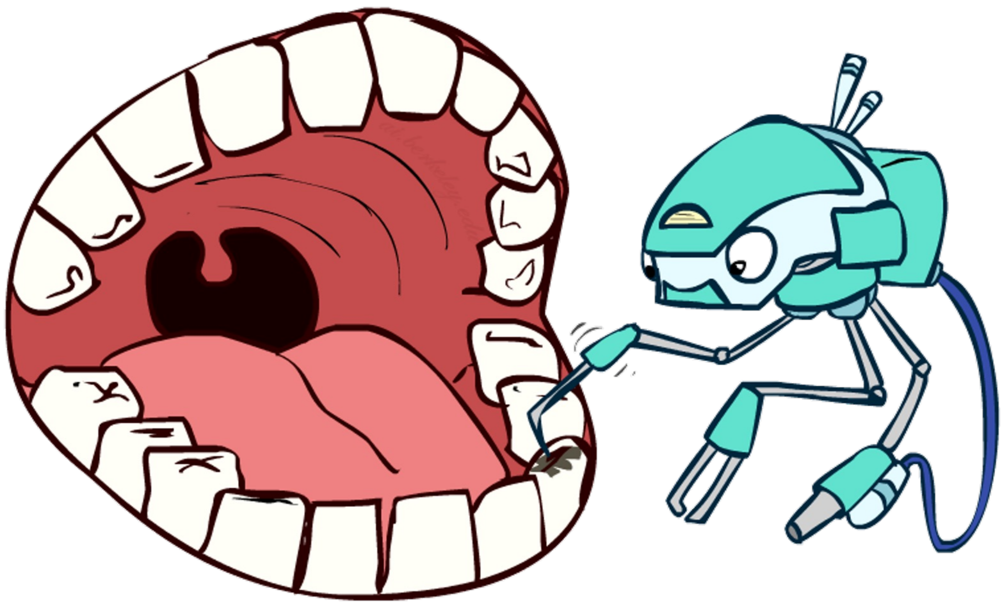
- P(Toothache, Cavity, Catch)

- If I have a cavity, the probability that the probe catches in it doesn't depend on whether I have a toothache:
  - P(+catch | +toothache, +cavity) = P(+catch | +cavity)

- The same independence holds if I don't have a cavity:
  - P(+catch | +toothache, -cavity) = P(+catch| -cavity)

- Catch is *conditionally independent* of Toothache given Cavity:
  - P(Catch | Toothache, Cavity) = P(Catch | Cavity)

- Equivalent statements:
  - P(Toothache | Catch , Cavity) = P(Toothache | Cavity)
  - P(Toothache, Catch | Cavity) = P(Toothache | Cavity) P(Catch | Cavity)
  - One can be derived from the other easily



Conditional Independence (cont.)

- Unconditional (absolute) independence very rare (why?)
- *Conditional independence* is our most basic and robust form of knowledge about uncertain environments.
- X is conditionally independent of Y given Z $X \perp\!\!\!\perp Y | Z$

  if and only if:

$$\forall x, y, z : P(x, y|z) = P(x|z)P(y|z)$$

  or, equivalently, if and only if

$$\forall x, y, z : P(x|z, y) = P(x|z)$$

$$P(x|z,y) = \frac{P(x,z,y)}{P(z,y)}$$

$$= \frac{P(x,y|z)P(z)}{P(y|z)P(z)}$$

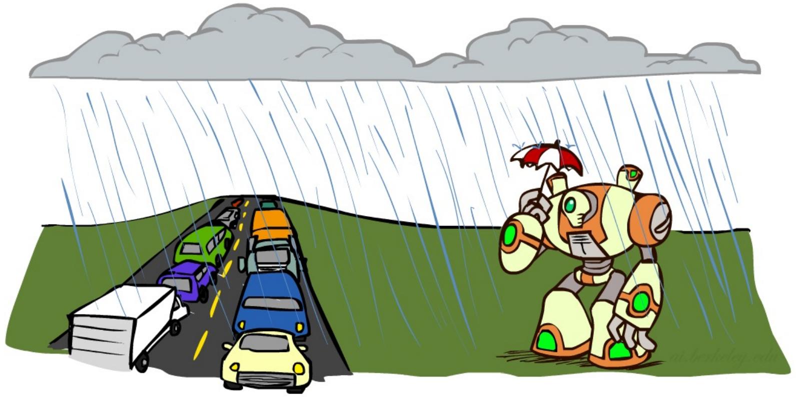
$$= \frac{P(x|z)P(y|z)P(z)}{P(y|z)P(z)}$$

Conditional Independence and the Chain Rule

- Chain rule: $P(X_1, X_2, \dots X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1, X_2)\dots$

- Trivial decomposition:

$$P(\text{Traffic}, \text{Rain}, \text{Umbrella}) =$$
$$P(\text{Rain})P(\text{Traffic}|\text{Rain})P(\text{Umbrella}|\text{Rain}, \text{Traffic})$$



- With assumption of conditional independence:

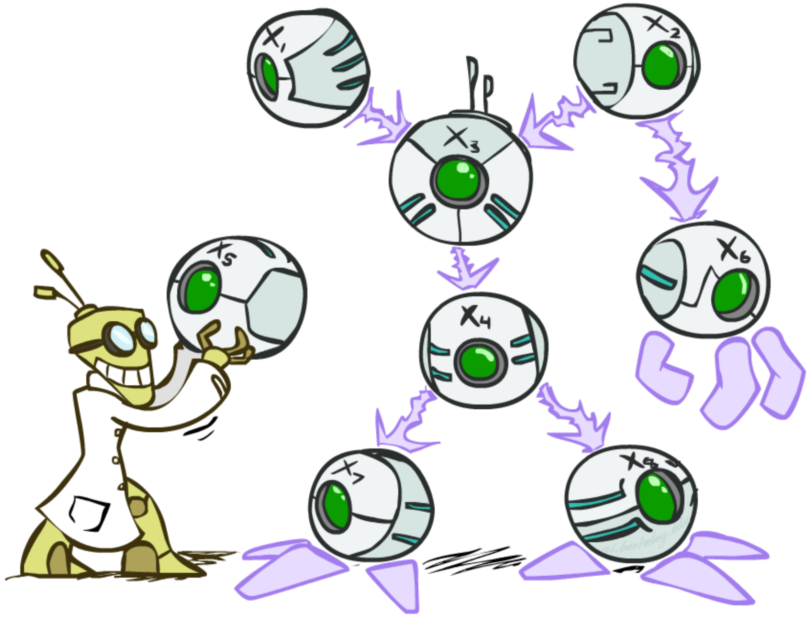
$$P(\text{Traffic}, \text{Rain}, \text{Umbrella}) =$$
$$P(\text{Rain})P(\text{Traffic}|\text{Rain})P(\text{Umbrella}|\text{Rain})$$

- Bayes'nets / graphical models help us express conditional independence assumptions

Bayes' Nets: Big Picture

- Two problems with using full joint distribution tables as our probabilistic models:
  - Unless there are only a few variables, the joint is WAY too big to represent explicitly
  - Hard to learn (estimate) anything empirically about more than a few variables at a time

- Bayes' nets: a technique for describing complex joint distributions (models) using simple, local distributions (conditional probabilities)
  - More properly called graphical models
  - We describe how variables locally interact
  - Local interactions chain together to give global, indirect interactions
  - We first look at some examples

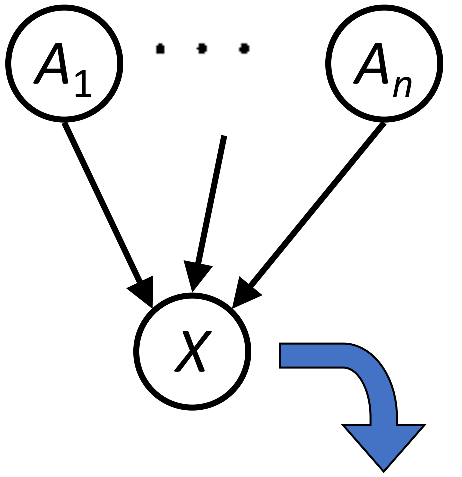


Bayes’ Net Semantics

- A set of nodes, one per variable X
- A directed, acyclic graph
- A conditional distribution for each node
  - A collection of distributions over X, one for each combination of parents’ values

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

  - CPT: conditional probability table
  - Description of a noisy “causal” process



$$P(X|A_1 \ldots A_n)$$

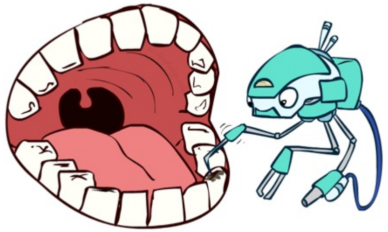
A Bayes net = Topology (graph) + Local Conditional Probabilities

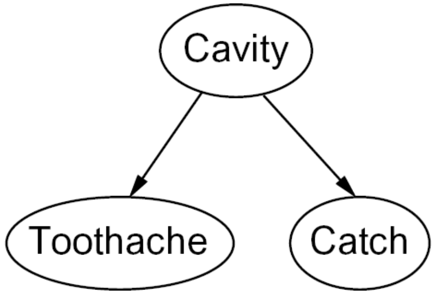
Probabilities in BNs

- Bayes' nets *implicitly* encode joint distributions
  - As a product of local conditional distributions
  - To see what probability a BN gives to a full assignment, multiply all the relevant conditionals together:

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

  - Example:





$$P(+\mathit{cavity},\ +\mathit{catch},\ \text{-}\mathit{toothache})$$

=P(-toothache|+cavity)P(+catch|+cavity)P(+cavity)

Probabilities in BNs 2

- Why are we guaranteed that setting

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

results in a proper joint distribution?

- Chain rule (valid for all distributions): $P(x_1, x_2, \ldots x_n) = \prod_{i=1}^{n} P(x_i | x_1 \ldots x_{i-1})$

- Assume conditional independences: $P(x_i | x_1, \ldots x_{i-1}) = P(x_i | parents(X_i))$

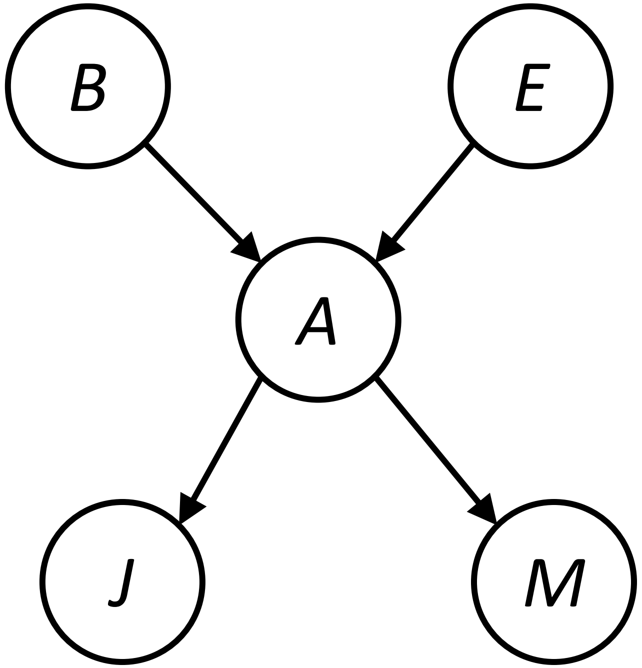
→ Consequence:

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

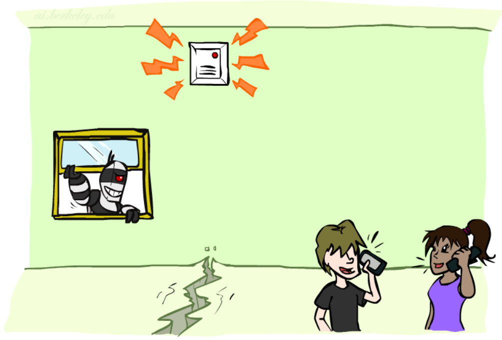
- Not every BN can represent every joint distribution
  - The topology enforces certain conditional independencies

Example: Alarm Network

B	P(B)
+b	0.001
-b	0.999



E	P(E)
+e	0.002
-e	0.998



A	J	P(J|A)
+a	+j	0.9
+a	-j	0.1
-a	+j	0.05
-a	-j	0.95

A	M	P(M|A)
+a	+m	0.7
+a	-m	0.3
-a	+m	0.01
-a	-m	0.99

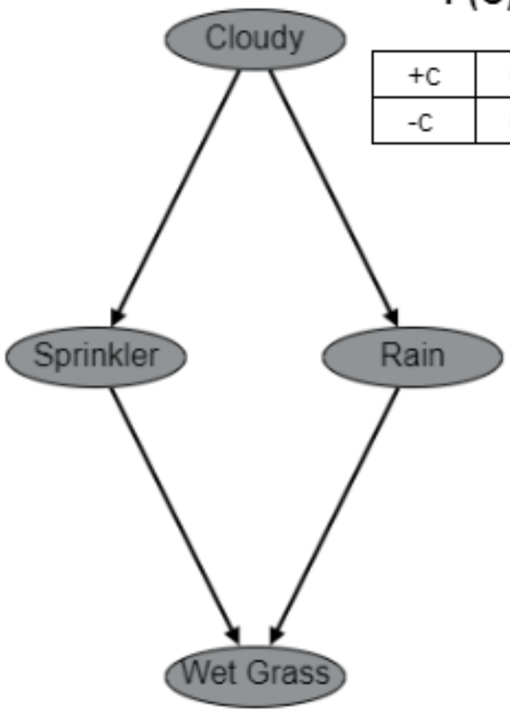
B	E	A	P(A|B,E)
+b	+e	+a	0.95
+b	+e	-a	0.05
+b	-e	+a	0.94
+b	-e	-a	0.06
-b	+e	+a	0.29
-b	+e	-a	0.71
-b	-e	+a	0.001
-b	-e	-a	0.999

$$P(+b, -e, +a, -j, +m) =$$

Quiz

- Compute $P(-c, +s, -r, +w)$

A. 0.0
B. 0.0004
C. 0.001
D. 0.036
E. 0.18
F. 0.198
G. 0.324



P(C)

+c	0.5
-c	0.5

P(S | C)

+c	+s	0.1
	-s	0.9
-c	+s	0.5
	-s	0.5

P(R | C)

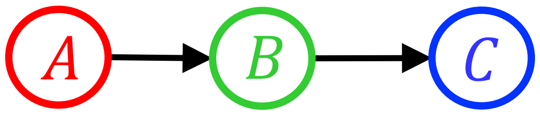
+c	+r	0.8
	-r	0.2
-c	+r	0.2
	-r	0.8

P(W | S, R)

+s	+r	+w	0.99
		-w	0.01
	-r	+w	0.9
		-w	0.1
-s	+r	+w	0.9
		-w	0.1
	-r	+w	0.99
		-w	0.01

Conditional Independence Semantics 2

- For the following Bayes nets, write the joint $P(A, B, C)$
  1. Using the chain rule (with top-down order A,B,C)
  2. Using Bayes net semantics (product of CPTs)



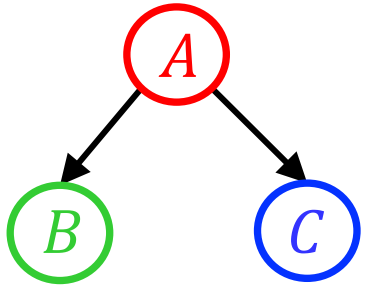
$P(A)\ P(B|A)\ P(C|A,B)$

$P(A)\ P(B|A)\ P(C|B)$

Assumption:

$P(C|A,B) = P(C|B)$

C is independent from A given B



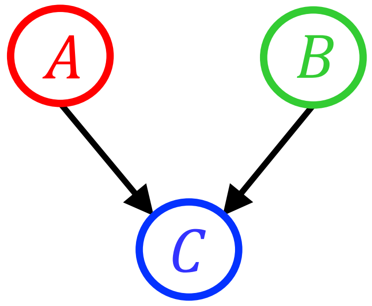
$P(A)\ P(B|A)\ P(C|A,B)$

$P(A)\ P(B|A)\ P(C|A)$

Assumption:

$P(C|A,B) = P(C|A)$

C is independent from B given A



$P(A)\ P(B|A)\ P(C|A,B)$

$P(A)\ P(B)\ P(C|A,B)$

Assumption:

$P(B|A) = P(B)$

A is independent from B given { }

Causal Chains

- This configuration is a "causal chain"

X: Low pressure Y: Rain Z: Traffic

$$P(x, y, z) = P(x)P(y|x)P(z|y)$$

- Guaranteed X independent of Z ?
- *No!*
  - One example set of CPTs for which X is not independent of Z is sufficient to show this independence is not guaranteed.
  - Example:
    - Low pressure causes rain causes traffic, high pressure causes no rain causes no traffic
    - In numbers:

      P(+y | +x) = 1, P(-y | - x) = 1,
      P(+z | +y) = 1, P(-z | -y) = 1

Causal Chains 2

- This configuration is a "causal chain"

X: Low pressure Y: Rain Z: Traffic

$$P(x, y, z) = P(x)P(y|x)P(z|y)$$

- Guaranteed X independent of Z given Y?

$$P(z|x, y) = \frac{P(x, y, z)}{P(x, y)}$$

$$= \frac{P(x)P(y|x)P(z|y)}{P(x)P(y|x)}$$

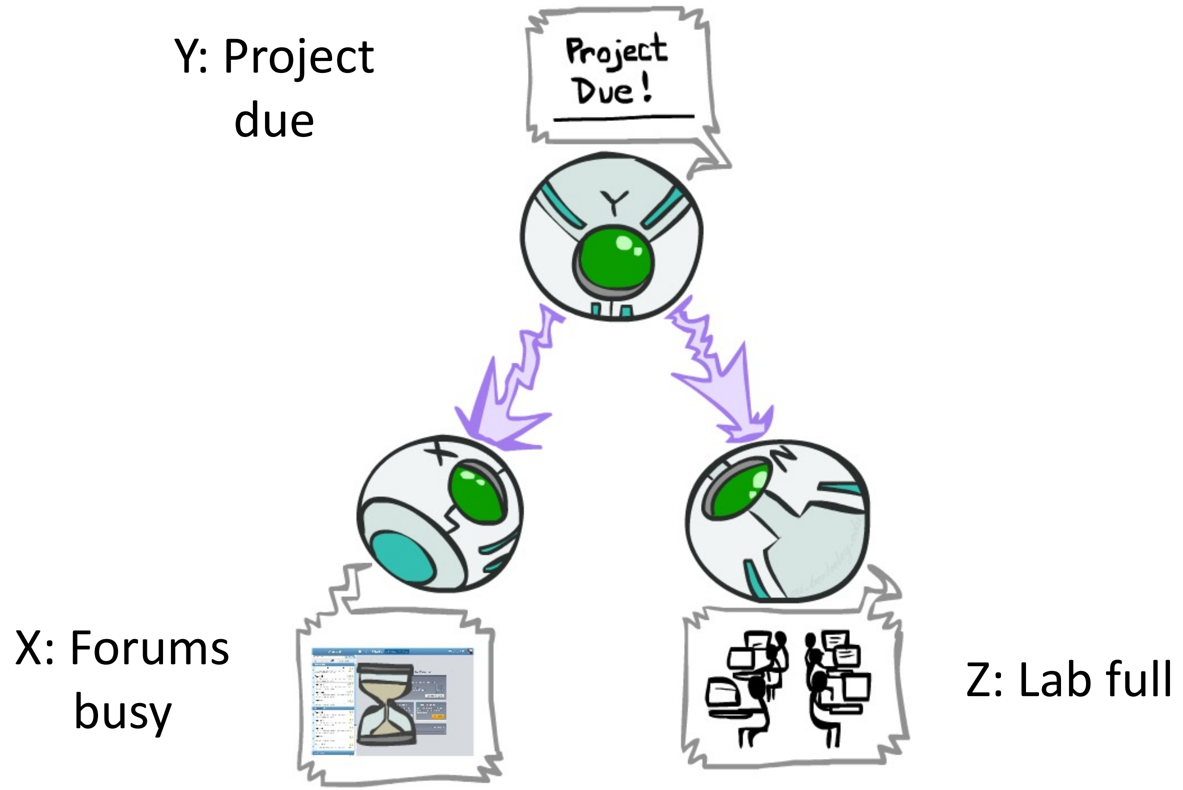
$$= P(z|y)$$

Yes!

- Evidence along the chain "blocks" the influence

Common Causes

- This configuration is a "common cause"



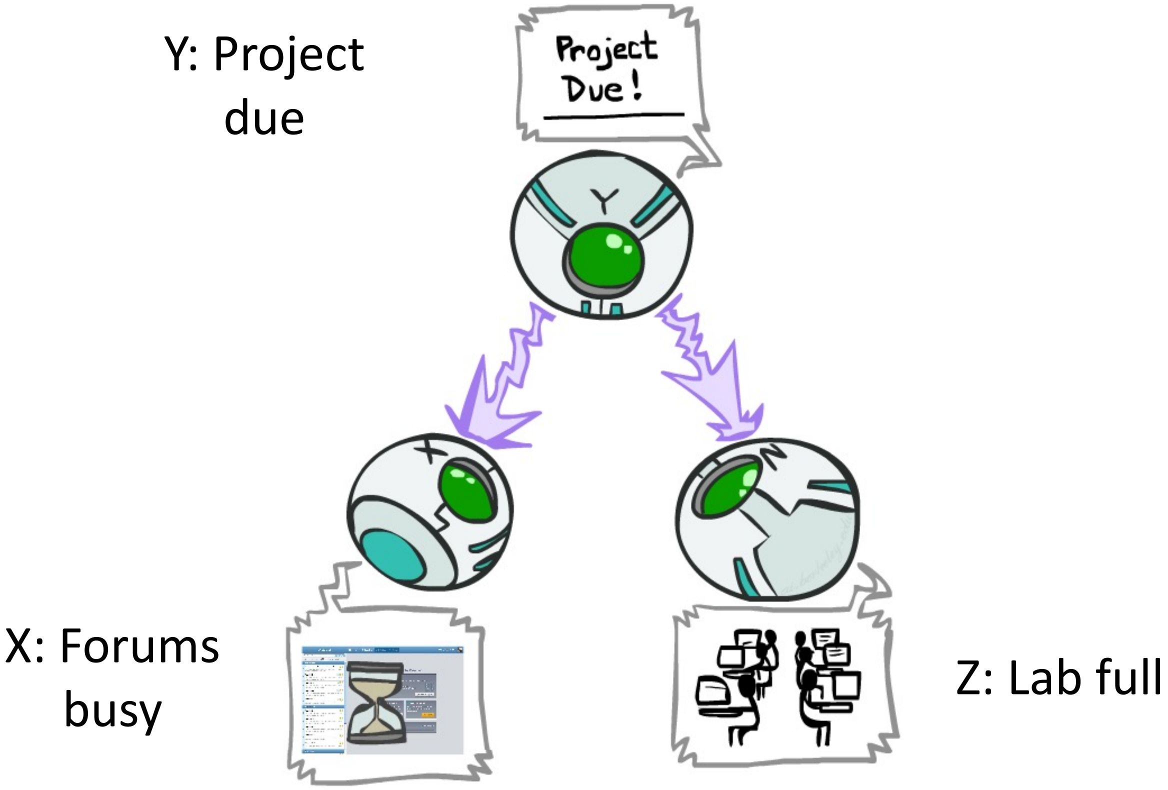
$$P(x,y,z) = P(y)P(x|y)P(z|y)$$

- Guaranteed X independent of Z ?
- *No!*
  - One example set of CPTs for which X is not independent of Z is sufficient to show this independence is not guaranteed.
  - Example:
    - Project due causes both forums busy and lab full
    - In numbers:

P(+x | +y) = 1, P(-x | -y) = 1,
P(+z | +y) = 1, P(-z | -y) = 1

Common Cause 2

- This configuration is a "common cause"



$$P(x, y, z) = P(y)P(x|y)P(z|y)$$

- Guaranteed X and Z independent given Y?

$$P(z|x, y) = \frac{P(x, y, z)}{P(x, y)}$$

$$= \frac{P(y)P(x|y)P(z|y)}{P(y)P(x|y)}$$

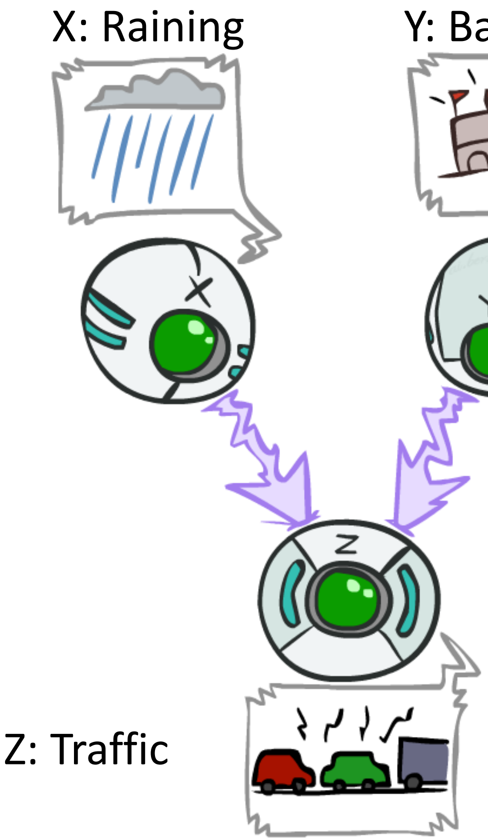
$$= P(z|y)$$

Yes!

- Observing the cause blocks influence between effects

Common Effect

- Last configuration: two causes of one effect (v-structures)

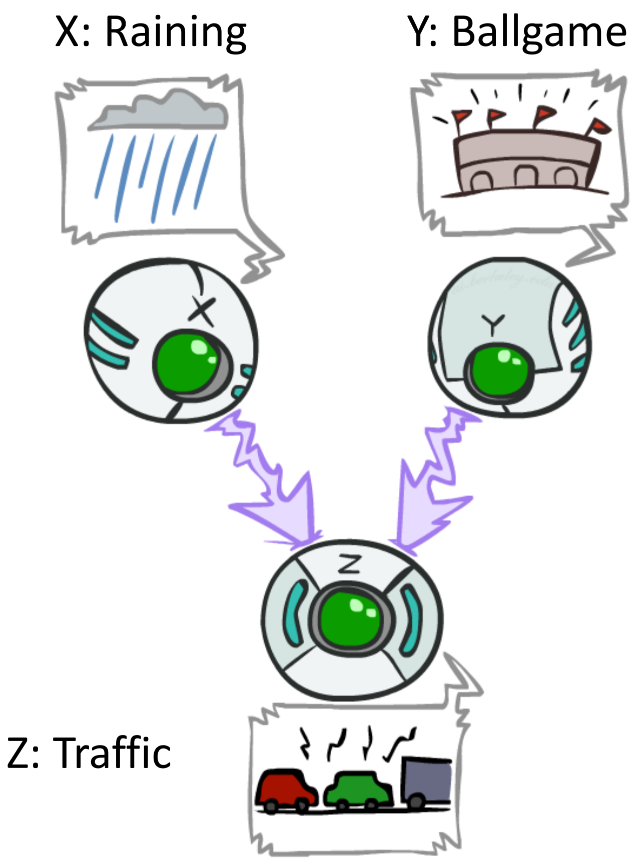


- Are X and Y independent?
  - *Yes*: the ballgame and the rain cause traffic, but they are not correlated
- Proof:

$$P(x,y) = \sum_z P(x,y,z)$$
$$= \sum_z P(x)P(y)P(z|x,y)$$
$$= P(x)P(y)\sum_z P(z|x,y)$$
$$= P(x)P(y)$$

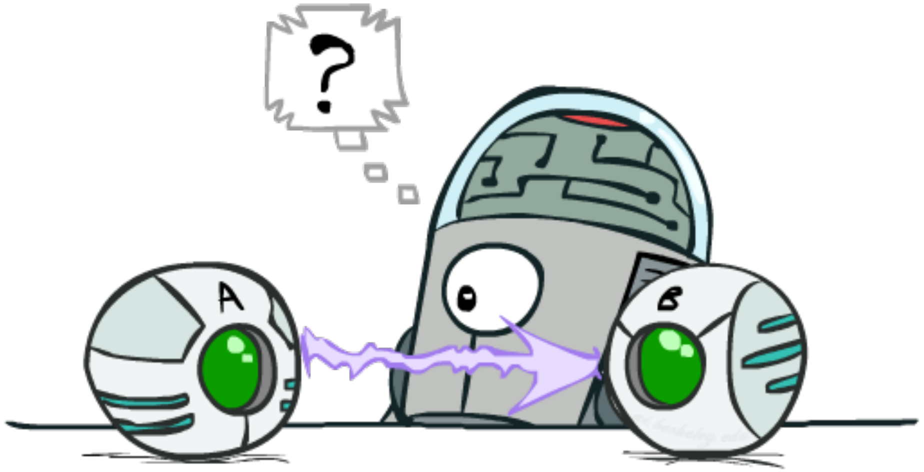
Common Effect 2

- Last configuration: two causes of one effect (v-structures)



- Are X and Y independent?
  - *Yes*: the ballgame and the rain cause traffic, but they are not correlated
  - (Proved previously)
- Are X and Y independent given Z?
  - *No*: seeing traffic puts the rain and the ballgame in competition as explanation.
- This is backwards from the other cases
  - Observing an effect activates influence between possible causes

Causality?



- When Bayes' nets reflect the true causal patterns:
  - Often simpler (nodes have fewer parents)
  - Often easier to think about
  - Often easier to elicit from experts
- BNs need not actually be causal
  - Sometimes no causal net exists over the domain (especially if variables are missing)
  - E.g. consider the variables *Traffic* and *Drips*
  - End up with arrows that reflect correlation, not causation
- What do the arrows really mean?
  - Topology may happen to encode causal structure
  - Topology really encodes conditional independence

$$P(x_i|x_1, \ldots x_{i-1}) = P(x_i|\text{parents}(X_i))$$

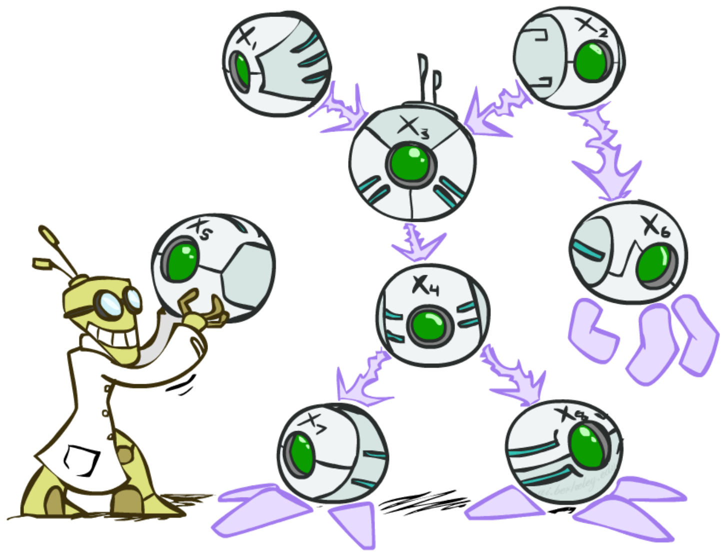
Bayes Net Semantics

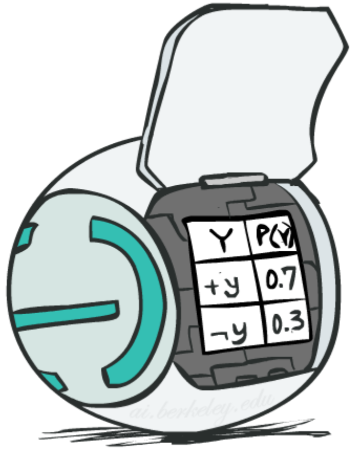
- A directed, acyclic graph, one node per random variable
- A conditional probability table (CPT) for each node
  - A collection of distributions over X, one for each combination of parents' values

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

- Bayes' nets implicitly encode joint distributions
  - As a product of local conditional distributions
  - To see what probability a BN gives to a full assignment, multiply all the relevant conditionals together:

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





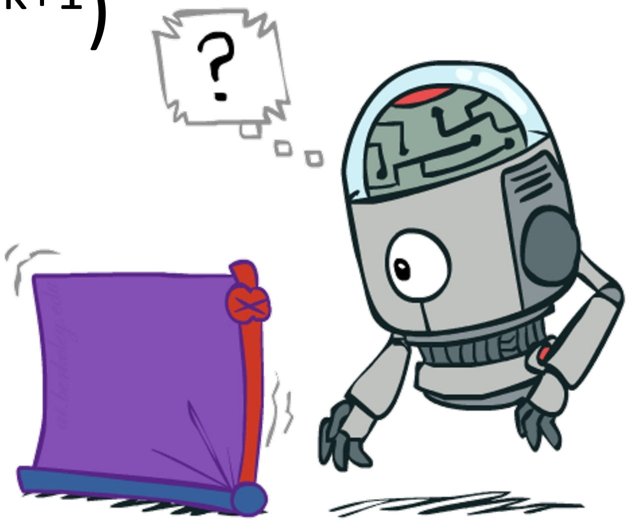
Size of a Bayes Net

- How big is a joint distribution over N Boolean variables?

  2^N

- How big is an N-node net if nodes have up to k parents?

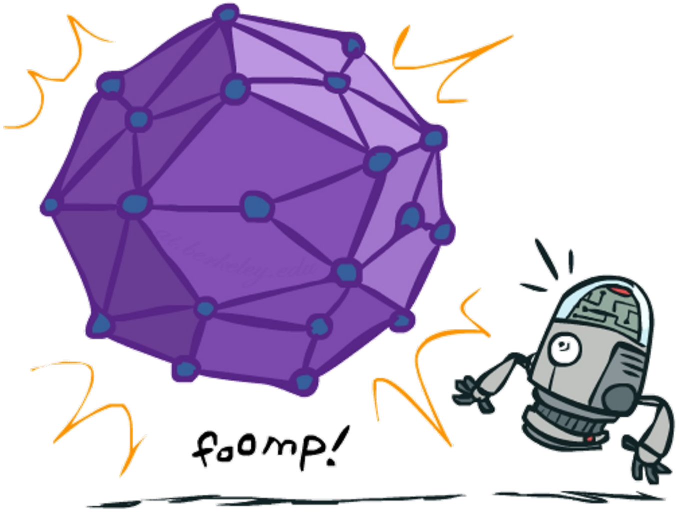
  $O(N * 2^{k+1})$



- Both give you the power to calculate

$$P(X_1, X_2, \ldots X_n)$$

- BNs: Huge space savings!
- Also easier to elicit local CPTs
- Also faster to answer queries

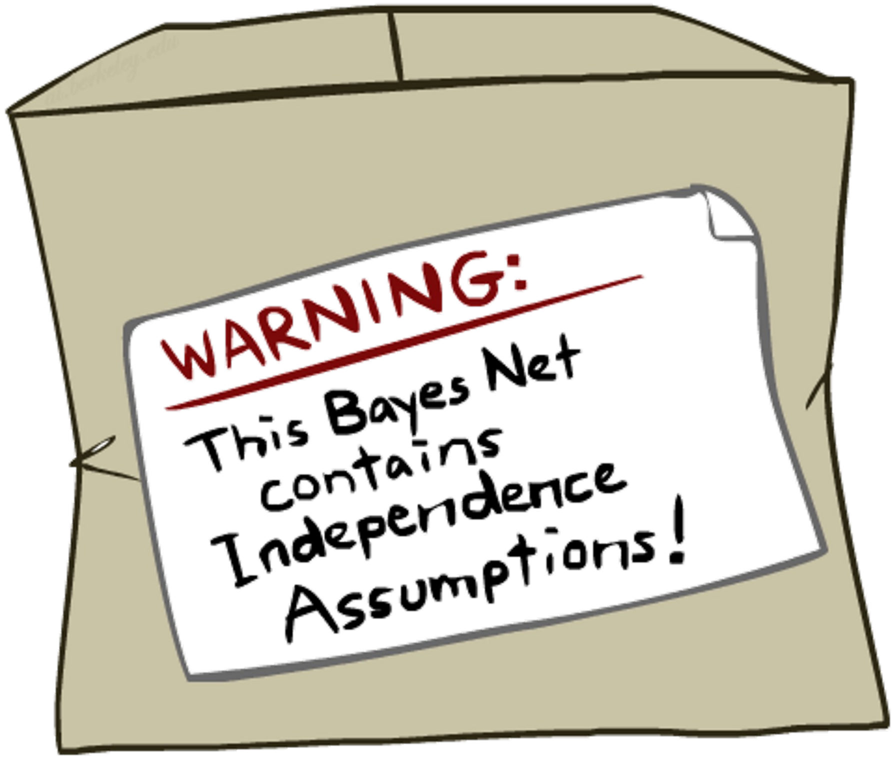


Bayes Nets: Assumptions

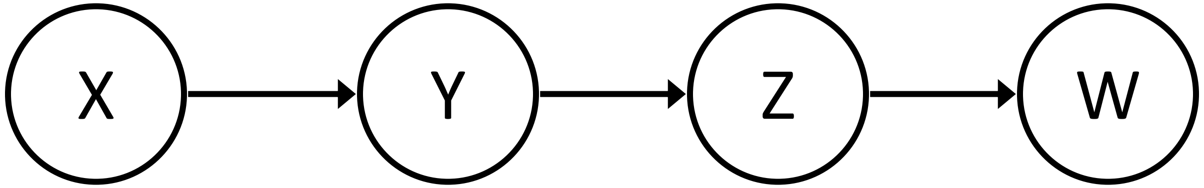
- Assumptions we are required to make to define the Bayes net when given the graph:

$$P(x_i|x_1 \cdots x_{i-1}) = P(x_i|parents(X_i))$$

- Beyond those "chain rule → Bayes net" conditional independence assumptions
  - Often additional conditional independences
  - They can be read off the graph
- Important for modeling: understand assumptions made when choosing a Bayes net graph



Example



- Conditional independence assumptions directly from simplifications in chain rule:

$$P(x,y,z,w) = P(x)P(y|x)P(z|x,y)P(w|x,y,z)$$
$$= P(x)P(y|x)P(z|y)P(w|z)$$

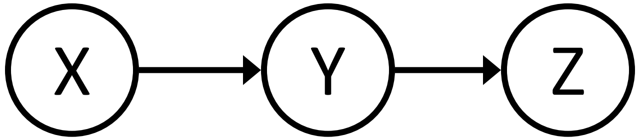
$$X \perp\!\!\!\perp Z|Y \qquad W \perp\!\!\!\perp \{X,Y\}|Z$$

- Additional implied conditional independence assumptions?

$$W \perp\!\!\!\perp X|Y$$ How?

Independence in a BN

- Important question about a BN:
  - Are two nodes independent given certain evidence?
  - If yes, can prove using algebra (tedious in general)
  - If no, can prove with a counter example
  - Example:



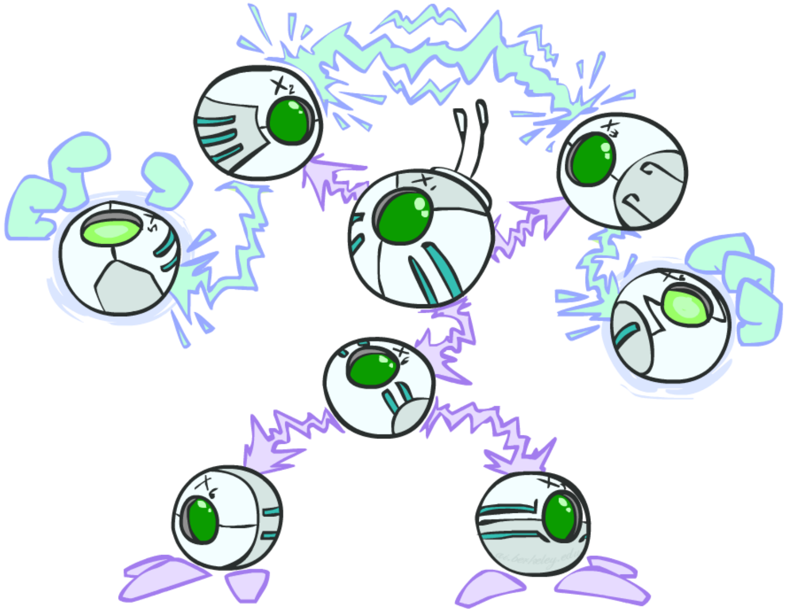
  - Question: are X and Z necessarily independent?
    - Answer: no. Example: low pressure causes rain, which causes traffic.
    - X can influence Z, Z can influence X (via Y)
    - Addendum: they *could* be independent: how?

The General Case

- General question: in a given BN, are two variables independent (given evidence)?

- Solution: analyze the graph

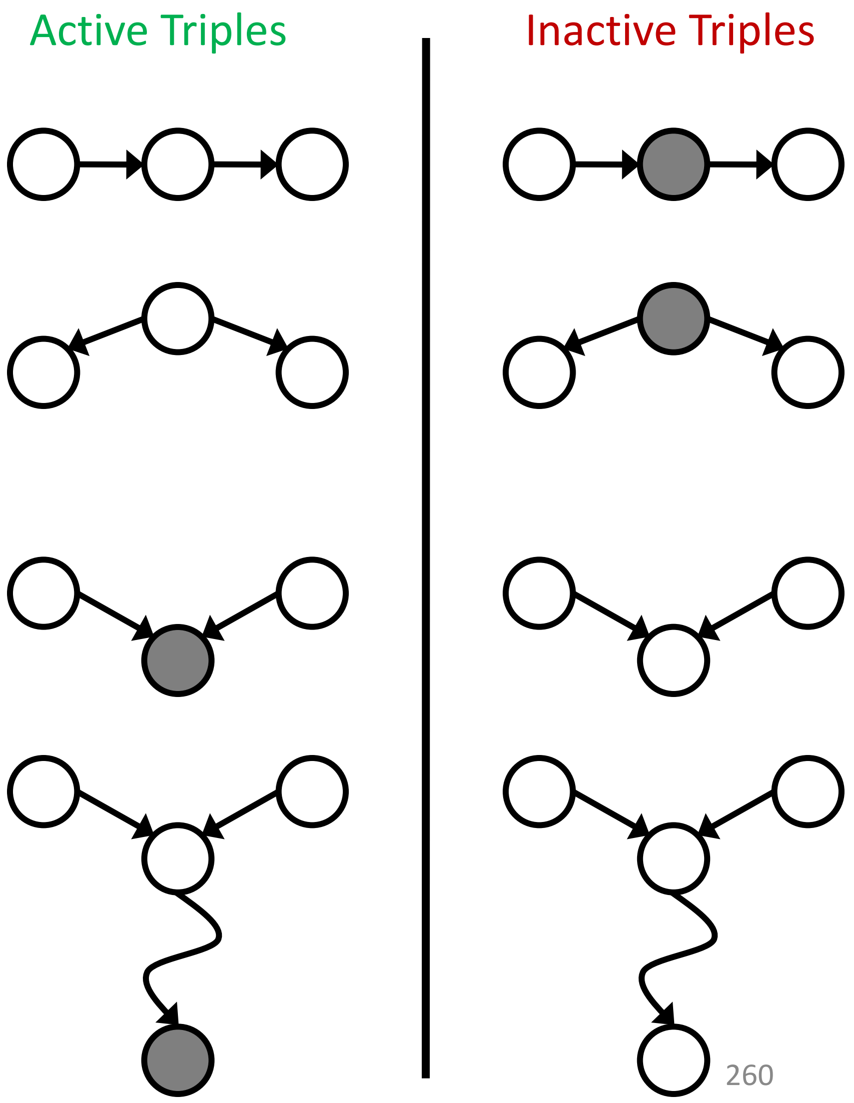
- Any complex example can be broken
  into repetitions of the three canonical cases



Bayes Ball

- Question: Are X and Y conditionally independent given evidence variables {Z}?

1. Shade in Z
2. Drop a ball at X
3. The ball can pass through any *active* path and is blocked by any *inactive* path (ball can move either direction on an edge)
4. If the ball reaches Y, then X and Y are NOT conditionally independent given Z

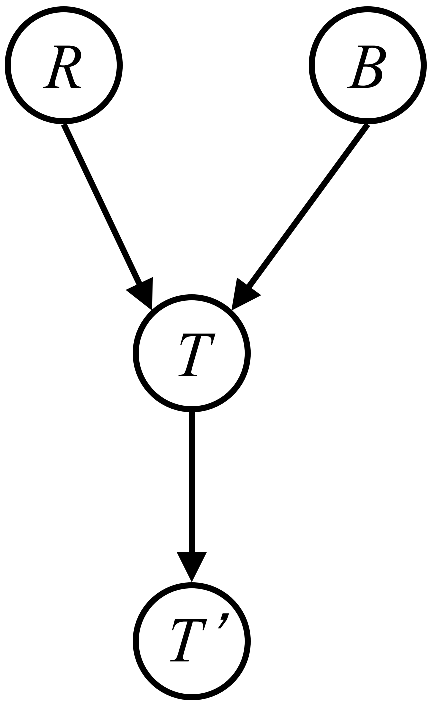


Example

$R \perp\!\!\!\perp B$ *Yes*

$R \perp\!\!\!\perp B | T$

$R \perp\!\!\!\perp B | T'$



Example 2

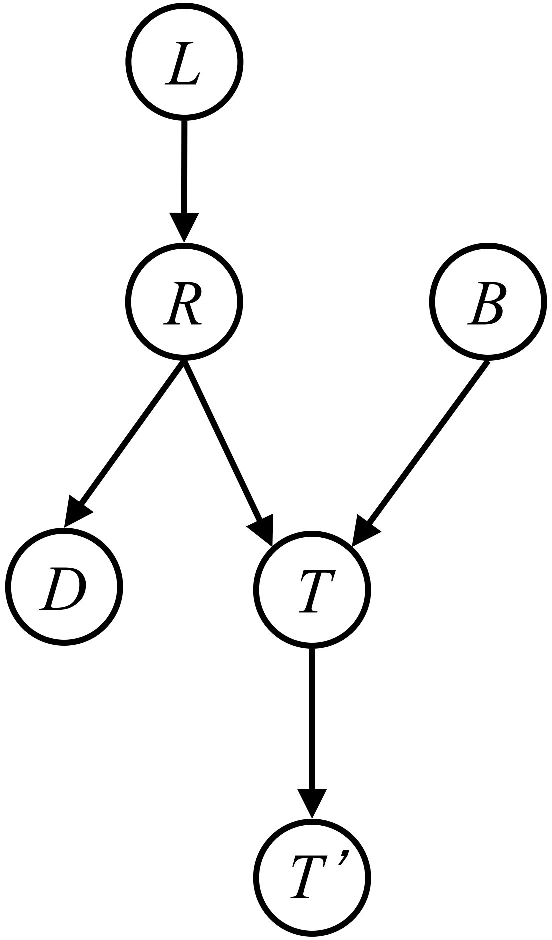
$L \perp\!\!\!\perp T' | T$ *Yes*

$L \perp\!\!\!\perp B$ *Yes*

$L \perp\!\!\!\perp B | T$

$L \perp\!\!\!\perp B | T'$

$L \perp\!\!\!\perp B | T, R$ *Yes*



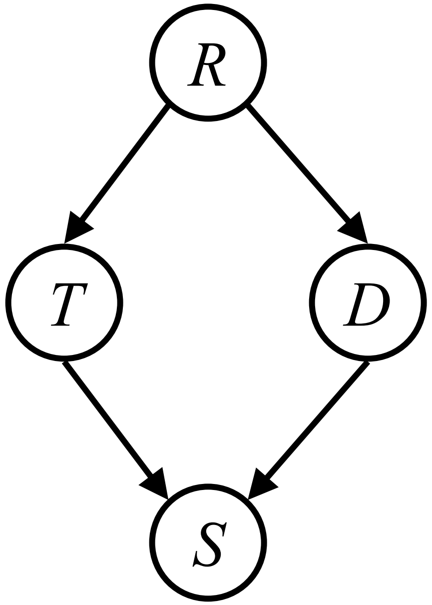
Example 3

- Variables:
  - R: Raining
  - T: Traffic
  - D: Roof drips
  - S: I'm sad
- Questions:

$$T \perp\!\!\!\perp D$$

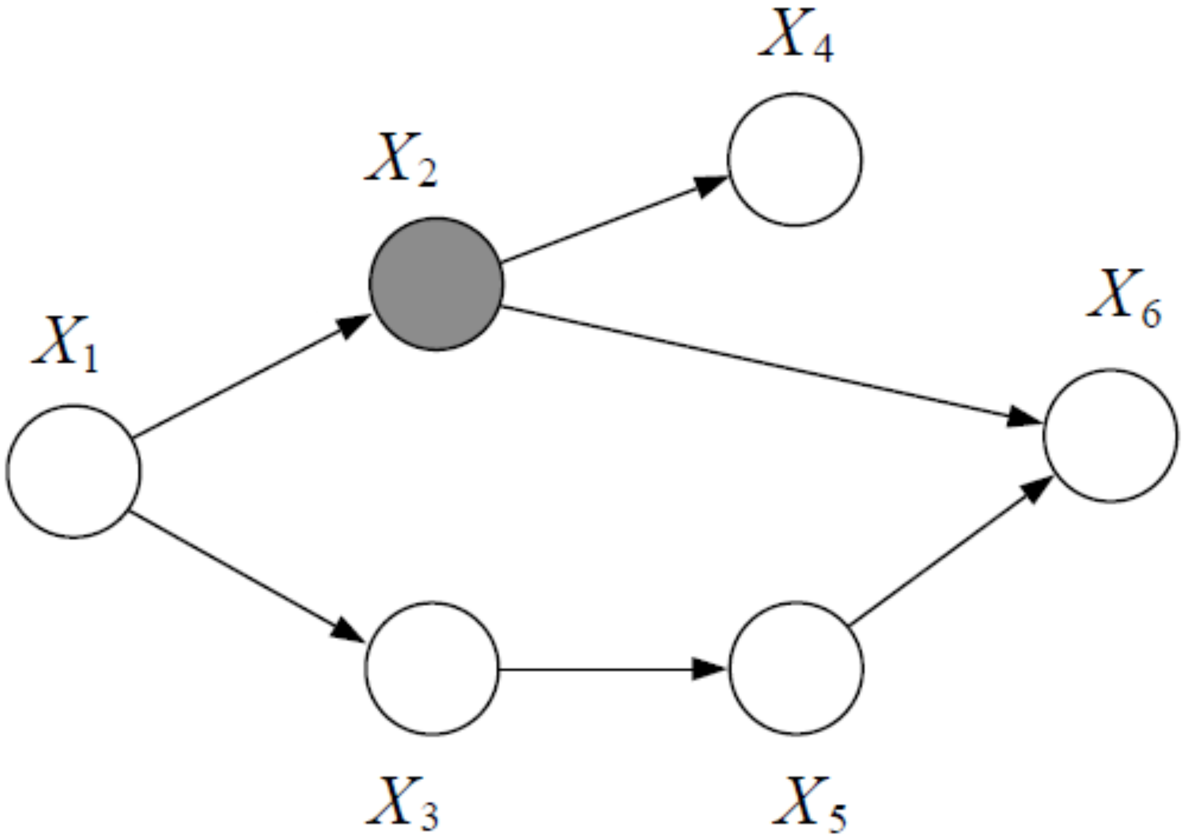
$$T \perp\!\!\!\perp D | R$$ *Yes*

$$T \perp\!\!\!\perp D | R, S$$



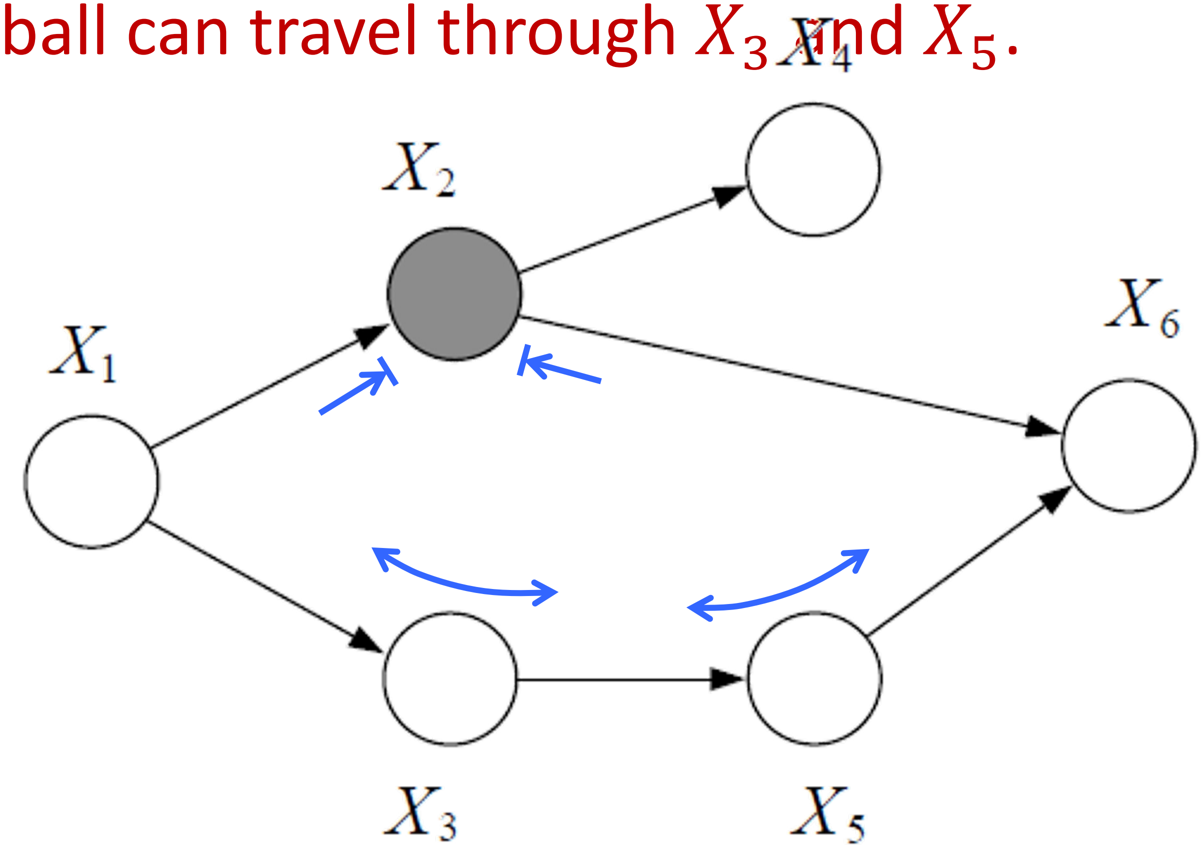
Quiz

- Is X_1 independent from X_6 given X_2?



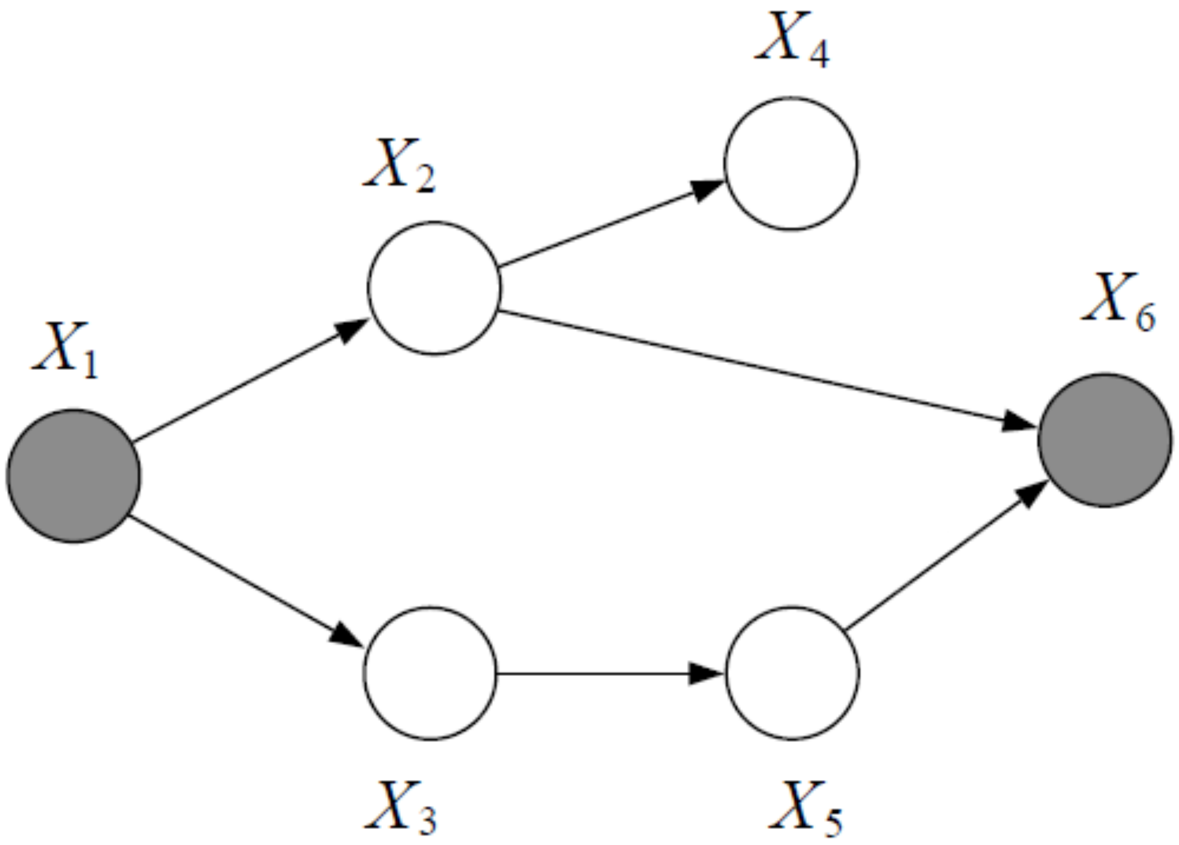
Quiz (cont.)

- Is X_1 independent from X_6 given X_2?
- No, the Bayes ball can travel through X_3 and X_5.



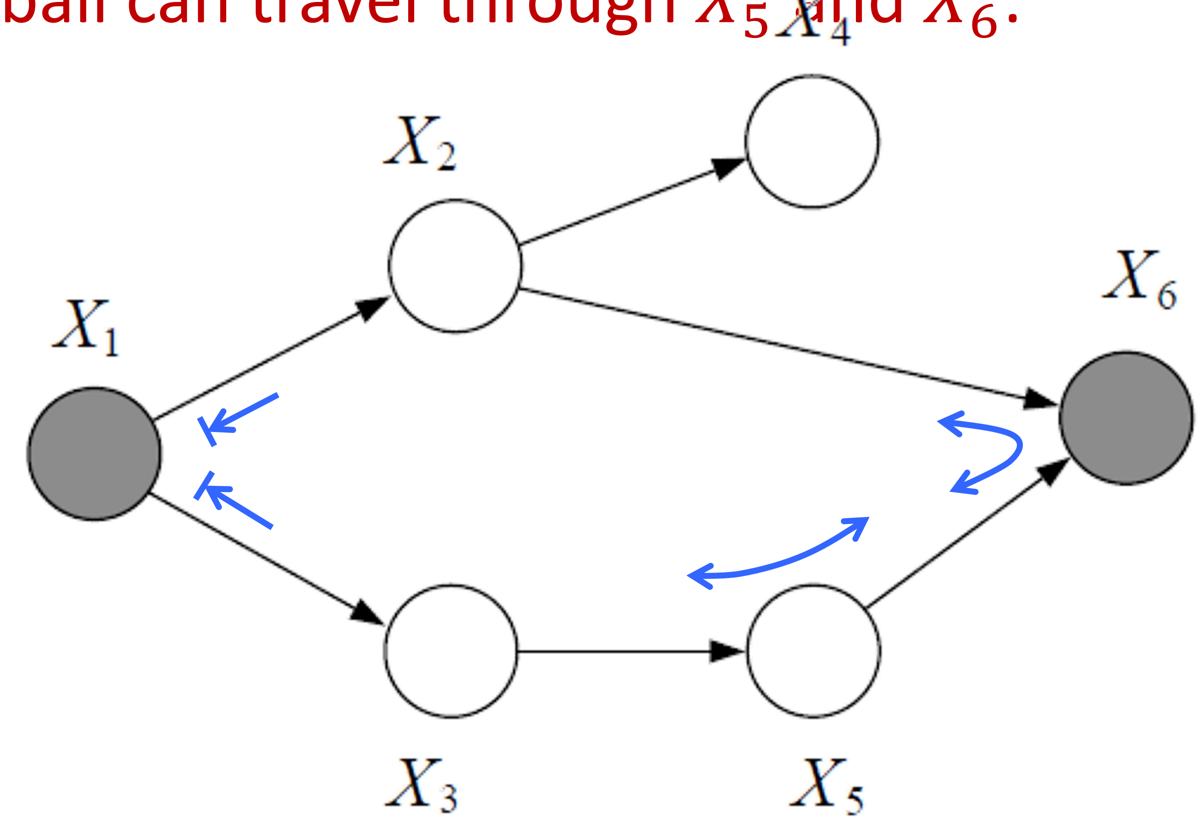
Quiz 2

- Is X_2 independent from X_3 given X_1 and X_6?



Quiz 2 (cont.)

- Is X_2 independent from X_3 given X_1 and X_6?
- No, the Bayes ball can travel through X_5 and X_6.



Bayes Nets: Inference

Queries

- What is the probability of *this* given what I know?

$$P(q \mid e) = \frac{P(q,\ e)}{P(e)} = \frac{\sum_{h_1} \sum_{h_2} P(q,\ h_1,\ h_2,\ e)}{P(e)}$$

- What are the probabilities of all the possible outcomes (given what I know)?

$$P(Q \mid e) = \frac{P(Q,\ e)}{P(e)} = \frac{\sum_{h_1} \sum_{h_2} P(Q,\ h_1,\ h_2,\ e)}{P(e)}$$

- Which outcome is the most likely outcome (given what I know)?

$$\text{argmax}_{q \in Q} P(q \mid e) = \text{argmax}_{q \in Q} \frac{P(q,\ e)}{P(e)}$$

$$= \text{argmax}_{q \in Q} \frac{\sum_{h_1} \sum_{h_2} P(q,\ h_1,\ h_2,\ e)}{P(e)}$$

Inference by Enumeration in Joint Distributions

- General case:
  - Evidence variables: $E_1 \dots E_k = e_1 \dots e_k$
  - Query* variable: Q
  - Hidden variables: $H_1 \dots H_r$

$X_1, X_2, \dots X_n$

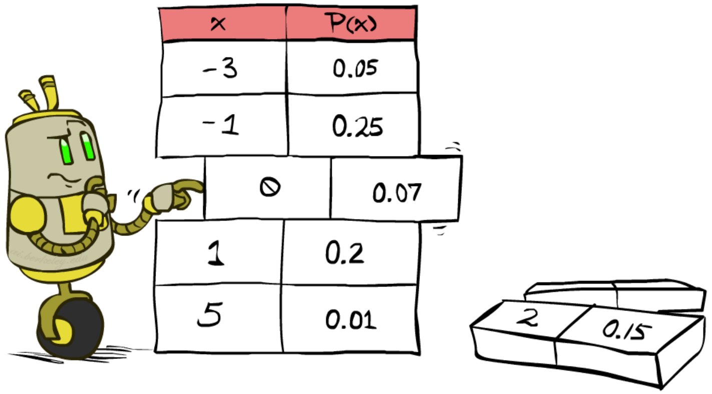
All variables

- We want:

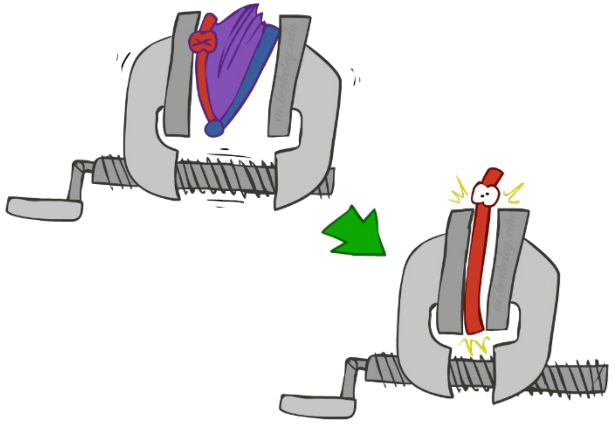
** Works fine with multiple query variables, too*

$$P(Q|e_1 \dots e_k)$$

- Step 1: Select the entries consistent with the evidence



- Step 2: Sum out H to get joint of Query and evidence



$$P(Q, e_1 \dots e_k) = \sum_{h_1 \dots h_r} \underbrace{P(Q, h_1 \dots h_r, e_1 \dots e_k)}_{X_1, X_2, \dots X_n}$$

- Step 3: Normalize

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

$$Z = \sum_q P(Q, e_1 \cdots e_k)$$

$$P(Q|e_1 \cdots e_k) = \frac{1}{Z} P(Q, e_1 \cdots e_k)$$

Inference by Enumeration: Procedural Outline

- Track objects called factors
- Initial factors are local CPTs (one per node)

$P(R)$

+r	0.1
-r	0.9

$P(T|R)$

+r	+t	0.8
+r	-t	0.2
-r	+t	0.1
-r	-t	0.9

$P(L|T)$

+t	+l	0.3
+t	-l	0.7
-t	+l	0.1
-t	-l	0.9

- Any known values are selected
  - E.g. if we know $L = +\ell$, the initial factors are

$P(R)$

+r	0.1
-r	0.9

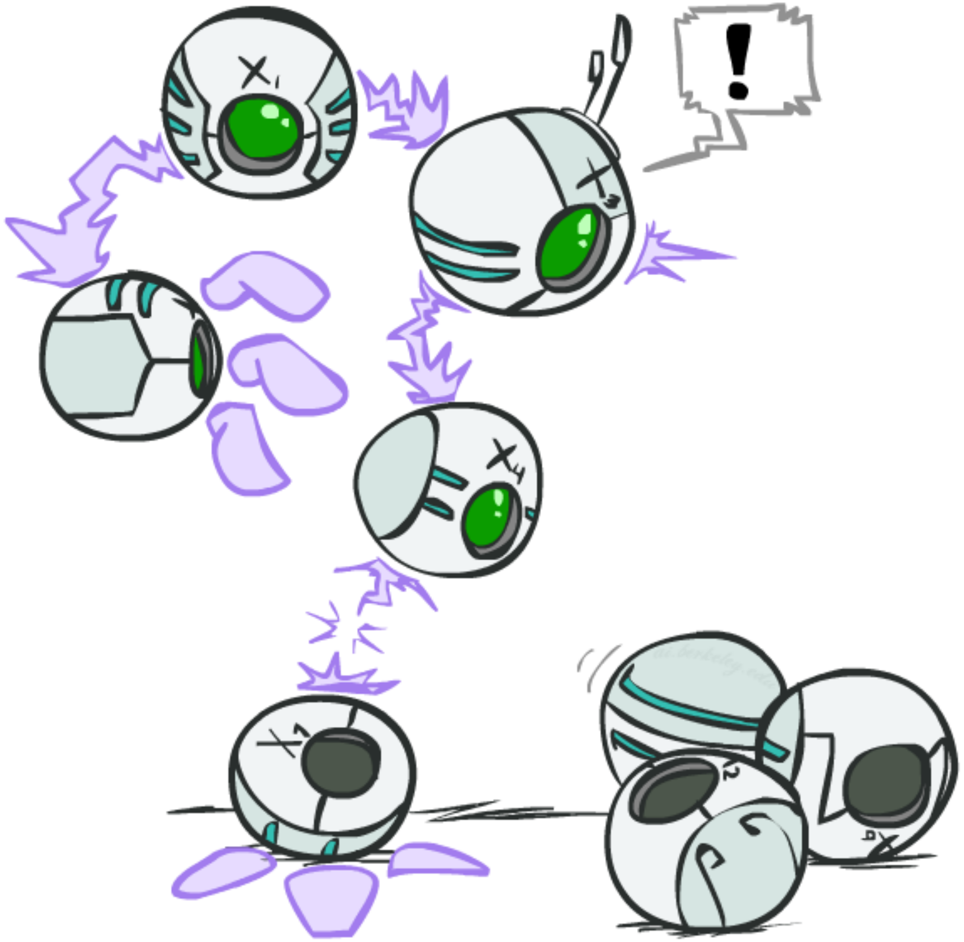
$P(T|R)$

+r	+t	0.8
+r	-t	0.2
-r	+t	0.1
-r	-t	0.9

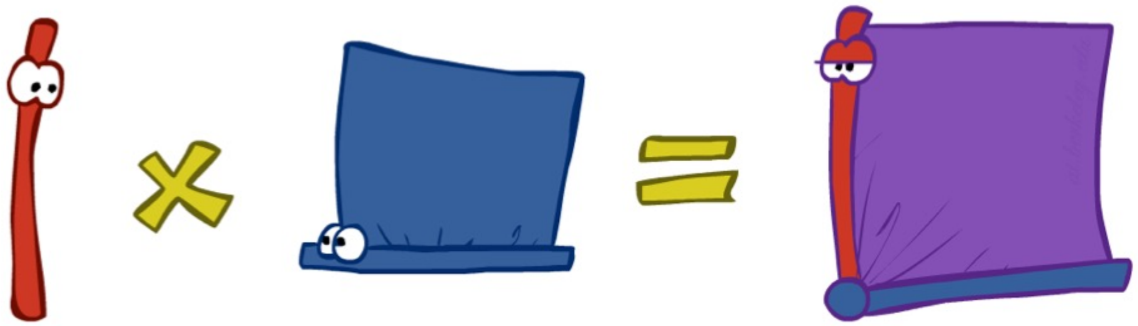
$P(+\ell|T)$

+t	+l	0.3
-t	+l	0.1

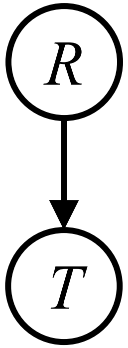
- Procedure: Join all factors, then sum out all hidden variables



Operation 1: Join Factors



- First basic operation: joining factors
- Combining factors:
  - Just like a database join
  - Get all factors over the joining variable
  - Build a new factor over the union of the variables involved
- Example: Join on R



$P(R)$

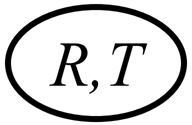
+r	0.1
-r	0.9

$\times$ $P(T|R)$

+r	+t	0.8
+r	-t	0.2
-r	+t	0.1
-r	-t	0.9

$\rightarrow$ $P(R,T)$

+r	+t	0.08
+r	-t	0.02
-r	+t	0.09
-r	-t	0.81



- Computation for each entry: pointwise products $\forall r, t : \quad P(r,t) = P(r) \cdot P(t|r)$

Operation 2: Eliminate

- Second basic operation: marginalization
- Take a factor and sum out a variable
  - Shrinks a factor to a smaller one
  - A projection operation
- Example:

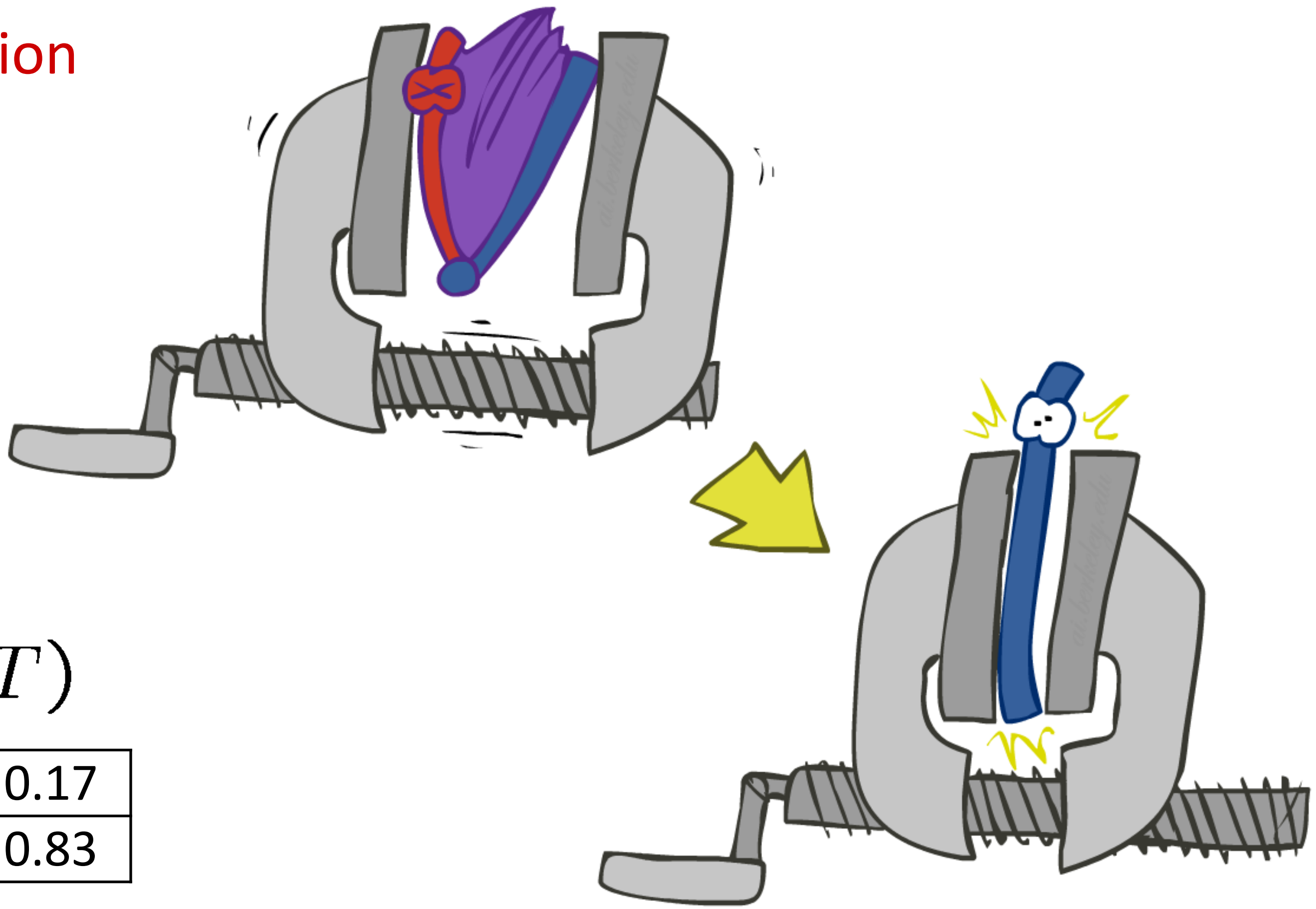
$P(R,T)$

+r	+t	0.08
+r	-t	0.02
-r	+t	0.09
-r	-t	0.81

sum R →

$P(T)$

+t	0.17
-t	0.83



Thus Far: Multiple Join, Multiple Eliminate (= Inference by Enumeration)

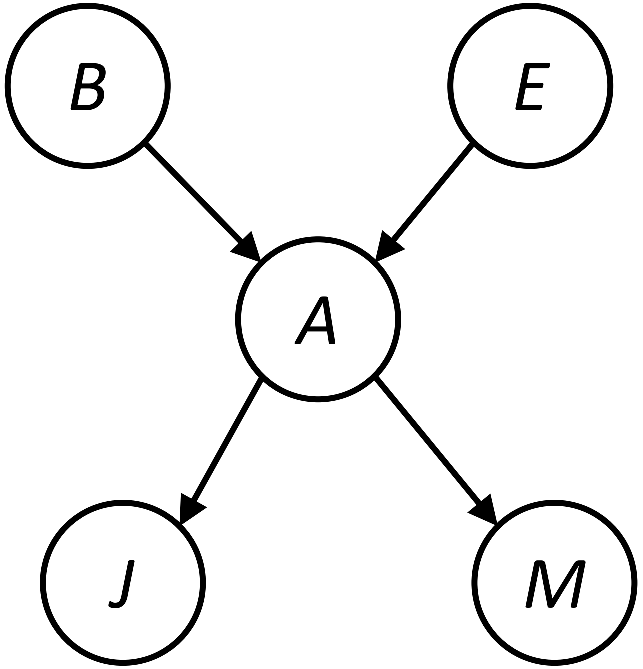
$P(R)$

$P(T|R)$ → $P(R, T, L)$ → $P(L)$

$P(L|T)$

Inference by Enumeration in Bayes Net

- Reminder of inference by enumeration:
  - Any probability of interest can be computed by summing entries from the joint distribution
  - Entries from the joint distribution can be obtained from a BN by multiplying the corresponding conditional probabilities



$$
\begin{aligned}
P(B \mid j, m) &= \alpha\, P(B, j, m) \\
&= \alpha \textstyle\sum_{e,a} P(B, e, a, j, m) \\
&= \alpha \textstyle\sum_{e,a} P(B)\, P(e)\, P(a \mid B, e)\, P(j \mid a)\, P(m \mid a)
\end{aligned}
$$

- So inference in Bayes nets means computing sums of products of numbers: sounds easy!!
- Problem: sums of *exponentially many* products!

Can we do better?

- Consider
  - $x_1y_1z_1 + x_1y_1z_2 + x_1y_2z_1 + x_1y_2z_2 + x_2y_1z_1 + x_2y_1z_2 + x_2y_2z_1 + x_2y_2z_2$
  - 16 multiplies, 7 adds
  - Lots of repeated subexpressions!
- Rewrite as
  - $(x_1 + x_2)(y_1 + y_2)(z_1 + z_2)$
  - 2 multiplies, 3 adds

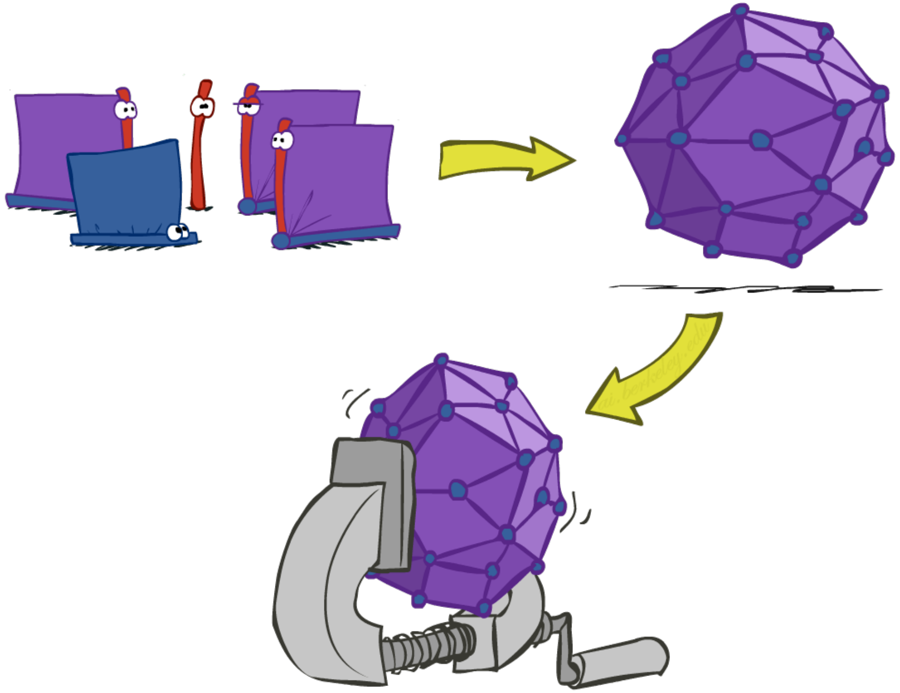
$$\sum_e \sum_a P(B)\,P(e)\,P(a \mid B,e)\,P(j \mid a)\,P(m \mid a)$$
$$= P(B)\,P(+e)\,P(+a \mid B,+e)\,P(j \mid +a)\,P(m \mid +a)$$
$$+ P(B)\,P(-e)\,P(+a \mid B,-e)\,P(j \mid +a)\,P(m \mid +a)$$
$$+ P(B)\,P(+e)\,P(-a \mid B,+e)\,P(j \mid -a)\,P(m \mid -a)$$
$$+ P(B)\,P(-e)\,P(-a \mid B,-e)\,P(j \mid -a)\,P(m \mid -a)$$

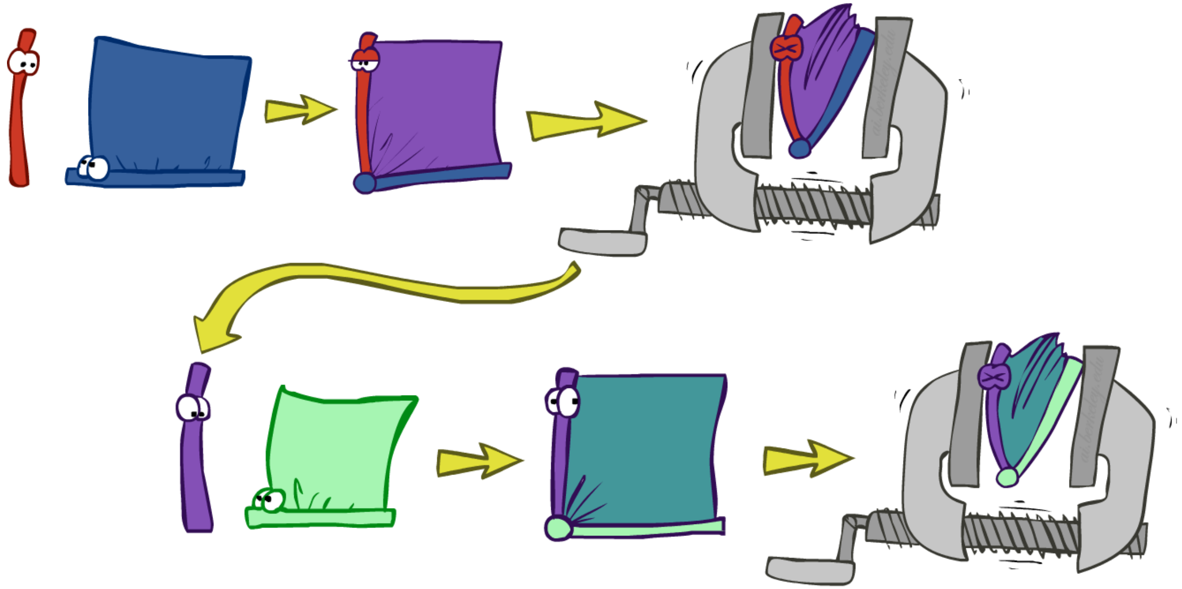
  - Lots of repeated subexpressions!

Inference by Enumeration vs. Variable Elimination

- Why is inference by enumeration so slow?
  - You join up the whole joint distribution before you sum out the hidden variables

- Idea: interleave joining and marginalizing!
  - Called "Variable Elimination"
  - Still NP-hard, but usually much faster than inference by enumeration



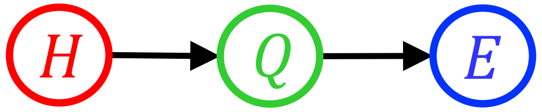


Inference Overview

- Given random variables Q, H, E (query, hidden, evidence)
- We know how to do inference on a joint distribution

$$P(q|e) = \alpha\, P(q,e)$$
$$= \alpha \sum_{h \in \{h_1,h_2\}} P(q,h,e)$$

- We know Bayes nets can break down joint in to CPT factors



$$P(q|e) = \alpha \sum_{h \in \{h_1,h_2\}} P(h)\, P(q|h)\, P(e|q)$$
$$= \alpha\, [P(h_1)\, P(q|h_1)\, P(e|q) + P(h_2)\, P(q|h_2)\, P(e|q)]$$

- But we can be more efficient

$$P(q|e) = \alpha\, P(e|q) \sum_{h \in \{h_1,h_2\}} P(h) P(q|h)$$
$$= \alpha\, P(e|q)\, [P(h_1)P(q|h_1) + P(h_2)P(q|h_2)]$$
$$= \alpha\, P(e|q)\, P(q)$$

- Now just extend to larger Bayes nets and a variety of queries

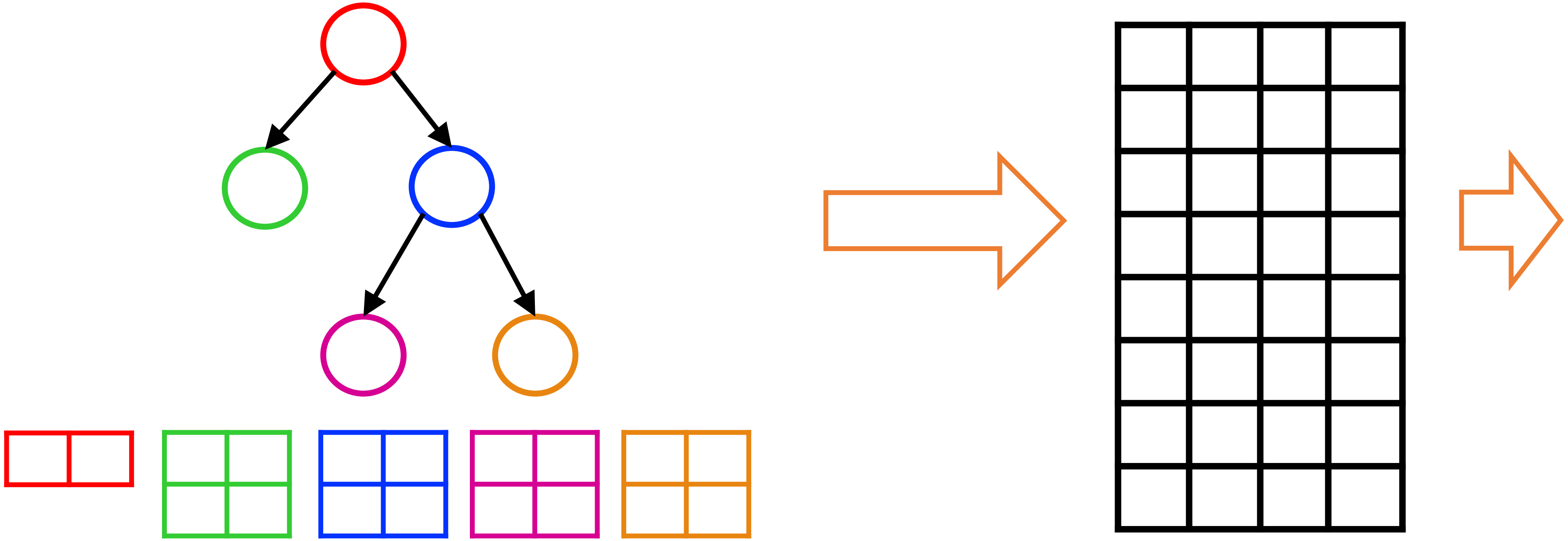
Enumeration

Variable Elimination

Answer Any Query from Bayes Net (Previous)

Bayes Net

Joint



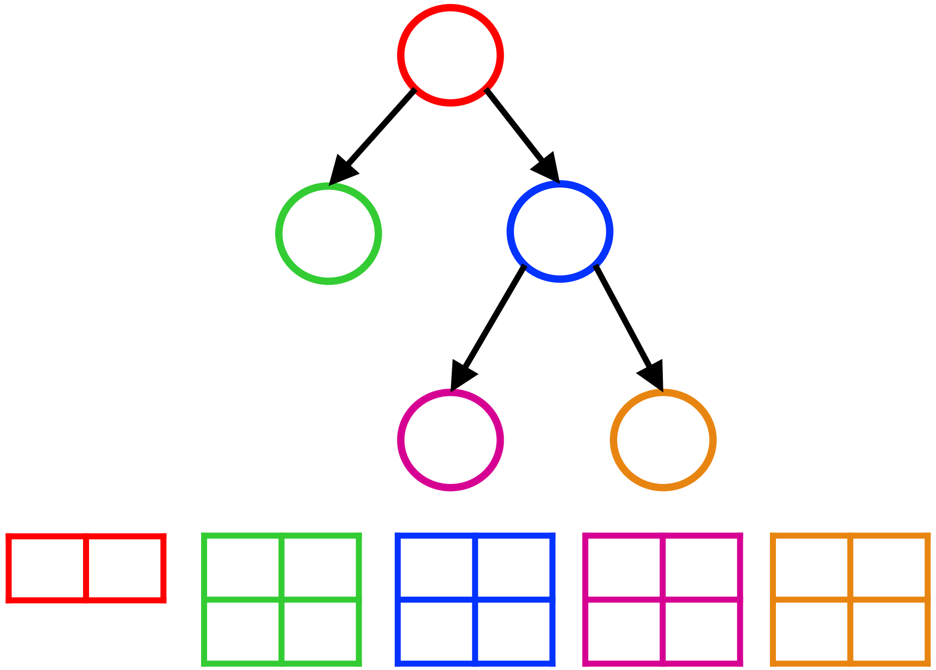
Query

$P(a \mid e)$

$P(A)$ $P(B|A)$ $P(C|A)$ $P(D|C)$ $P(E|C)$

Next: Answer Any Query from Bayes Net

Bayes Net

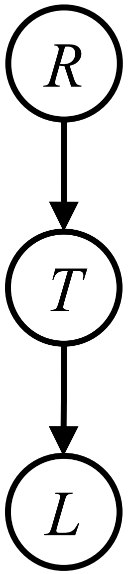


$P(A)$ $P(B|A)$ $P(C|A)$ $P(D|C)$ $P(E|C)$

Query

$P(a \mid e)$

Marginalizing Early! (aka VE)



$P(R)$

+r	0.1
-r	0.9

$P(T|R)$

+r	+t	0.8
+r	-t	0.2
-r	+t	0.1
-r	-t	0.9

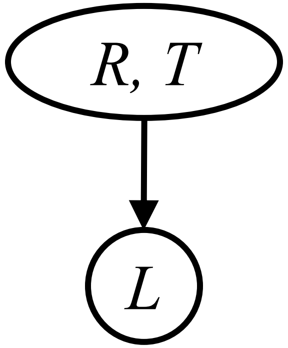
$P(L|T)$

+t	+l	0.3
+t	-l	0.7
-t	+l	0.1
-t	-l	0.9

Join R

$P(R, T)$

+r	+t	0.08
+r	-t	0.02
-r	+t	0.09
-r	-t	0.81



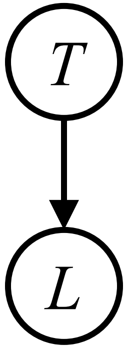
$P(L|T)$

+t	+l	0.3
+t	-l	0.7
-t	+l	0.1
-t	-l	0.9

Sum out R

$P(T)$

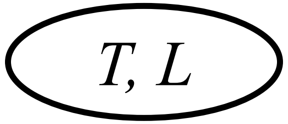
+t	0.17
-t	0.83



$P(L|T)$

+t	+l	0.3
+t	-l	0.7
-t	+l	0.1
-t	-l	0.9

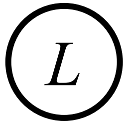
Join T



$P(T, L)$

+t	+l	0.051
+t	-l	0.119
-t	+l	0.083
-t	-l	0.747

Sum out T



$P(L)$

+l	0.134
-l	0.866

Evidence

- If evidence, start with factors that select that evidence
  - No evidence, uses these initial factors:

$P(R)$

+r	0.1
-r	0.9

$P(T|R)$

+r	+t	0.8
+r	-t	0.2
-r	+t	0.1
-r	-t	0.9

$P(L|T)$

+t	+l	0.3
+t	-l	0.7
-t	+l	0.1
-t	-l	0.9

  - Computing $P(L|+r)$, the initial factors become:

$P(+r)$

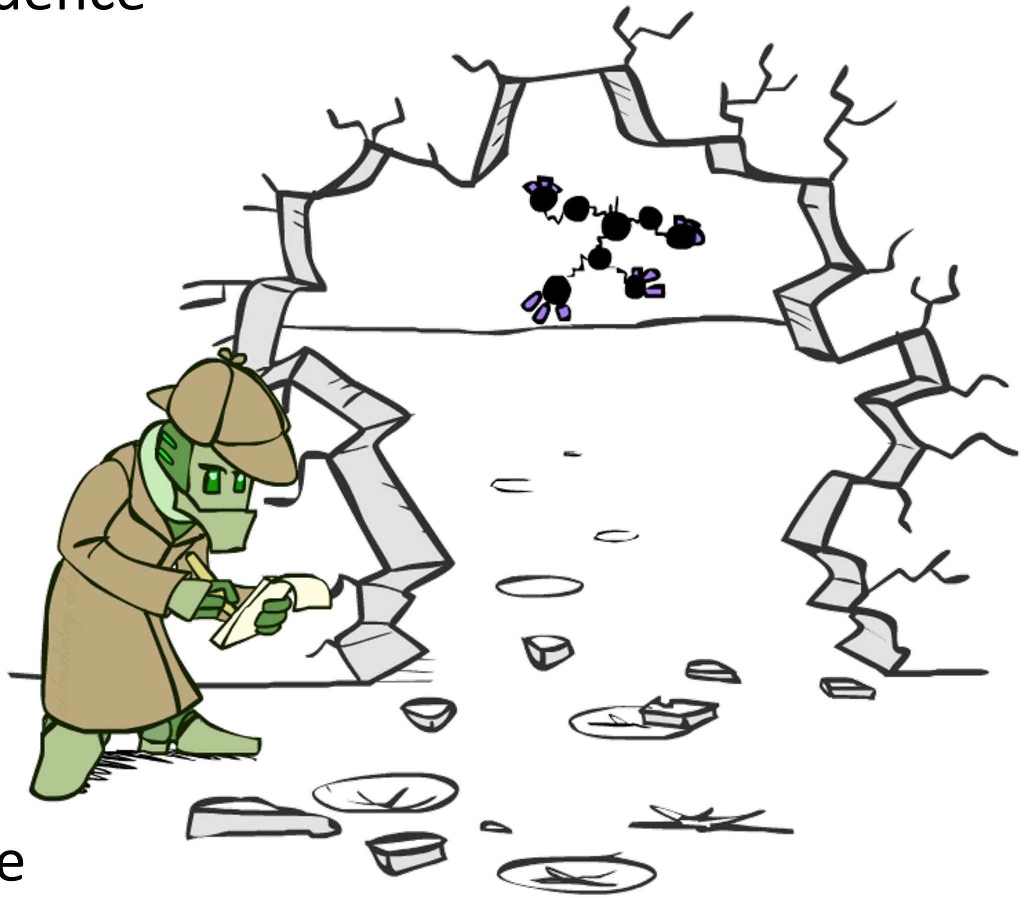
+r	0.1

$P(T|+r)$

+r	+t	0.8
+r	-t	0.2

$P(L|T)$

+t	+l	0.3
+t	-l	0.7
-t	+l	0.1
-t	-l	0.9



- We eliminate all vars other than query + evidence

Evidence II

- Result will be a selected joint of query and evidence
  - E.g. for P(L | +r), we would end up with:

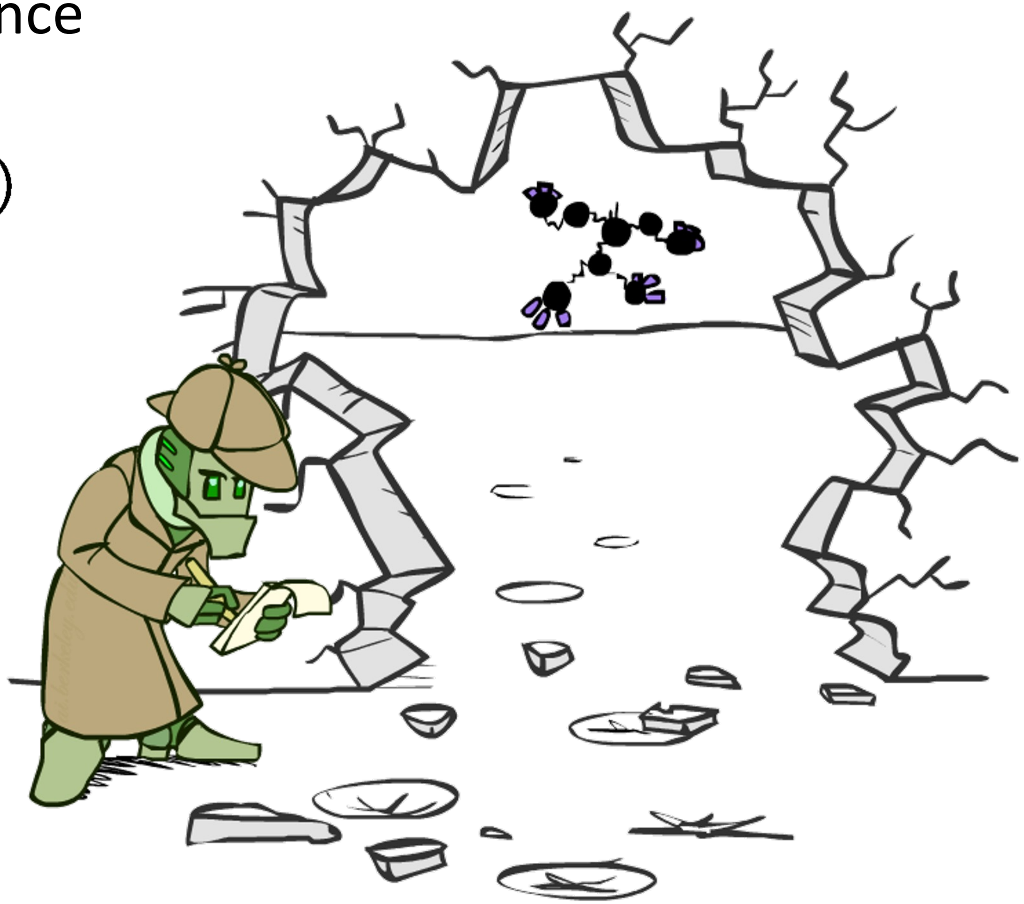
$P(+r, L)$

+r	+l	0.026
+r	-l	0.074

Normalize

$P(L|+r)$

+l	0.26
-l	0.74



- To get our answer, just normalize this!

- That ’s it!

Variable Elimination

- General case:
  - Evidence variables: $E_1 \dots E_k = e_1 \dots e_k$
  - Query* variable: Q
  - Hidden variables: $H_1 \dots H_r$

$X_1, X_2, \dots X_n$

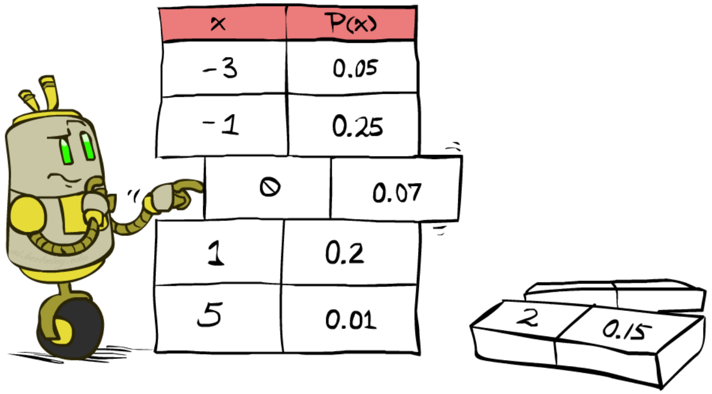
All variables

- We want:

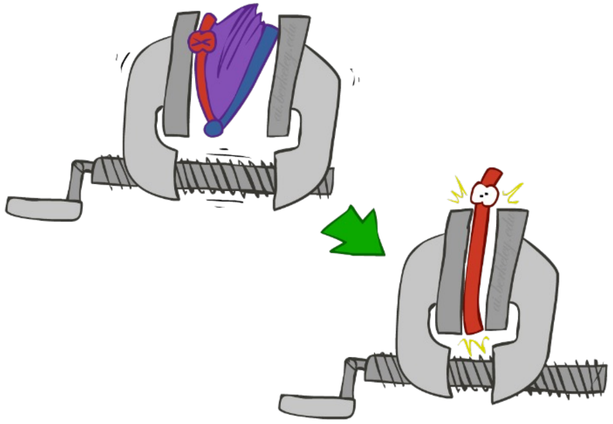
** Works fine with multiple query variables, too*

$$P(Q|e_1 \dots e_k)$$

- Step 1: Select the entries consistent with the evidence



- Step 2: Sum out H to get joint of Query and evidence



$$P(Q, e_1 \dots e_k) = \sum_{h_1 \dots h_r} \underbrace{P(Q, h_1 \dots h_r, e_1 \dots e_k)}_{X_1, X_2, \dots X_n}$$

- Step 3: Normalize

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

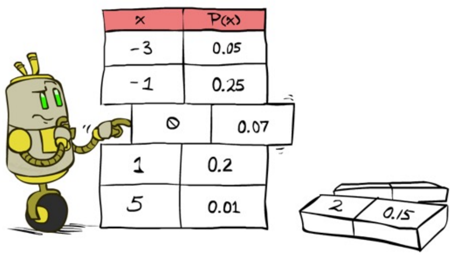
$$Z = \sum_q P(Q, e_1 \cdots e_k)$$

$$P(Q|e_1 \cdots e_k) = \frac{1}{Z} P(Q, e_1 \cdots e_k)$$

- Interleave joining and summing out $X_1, X_2, \dots X_n$

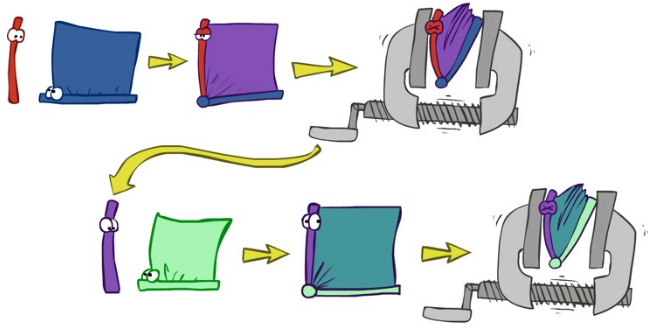
General Variable Elimination

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

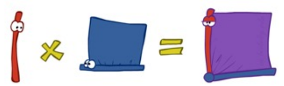


- Start with initial factors:
  - Local CPTs (but instantiated by evidence)

- 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



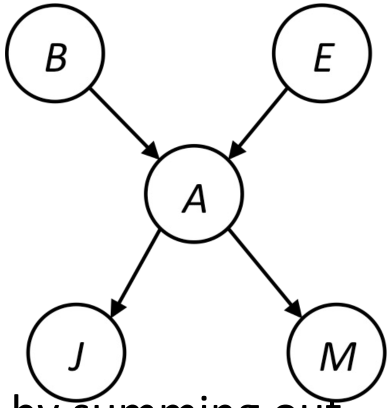
$\times \frac{1}{Z}$

Variable Elimination

```
function VariableElimination(Q , e, bn) returns a distribution over Q
    factors ← [ ]
    for each var in ORDER(bn.vars) do
        factors ← [MAKE-FACTOR(var, e)|factors]
        if var is a hidden variable then
                factors ← SUM-OUT(var,factors)
    return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Example

$$P(B|j,m) \propto P(B,j,m)$$



$P(B)$	$P(E)$	$P(A|B,E)$	$P(j|A)$	$P(m|A)$

$$P(B|j,m) \propto P(B,j,m)$$ marginal can be obtained from joint by summing out

$$= \sum_{e,a} P(B,j,m,e,a)$$ use Bayes' net joint distribution expression

$$= \sum_{e,a} P(B)P(e)P(a|B,e)P(j|a)P(m|a)$$ use x*(y+z) = xy + xz

$$= \sum_{e} P(B)P(e)\sum_{a} P(a|B,e)P(j|a)P(m|a)$$ joining on a, and then summing out gives f_1

$$= \sum_{e} P(B)P(e)f_1(j,m|B,e)$$ use x*(y+z) = xy + xz

$$= P(B)\sum_{e} P(e)f_1(j,m|B,e)$$ joining on e, and then summing out gives f_2

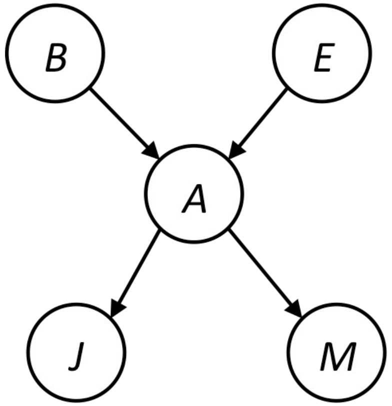
$$= P(B)f_2(j,m|B)$$

All we are doing is exploiting uwy + uwz + uxy + uxz + vwy + vwz + vxy +vxz = (u+v)(w+x)(y+z) to improve computational efficiency!

Example (cont'd)

$$P(B|j,m) \propto P(B,j,m)$$

$P(B)$	$P(E)$	$P(A\|B,E)$	$P(j\|A)$	$P(m\|A)$



Choose A

$P(A|B,E)$

$P(j|A)$

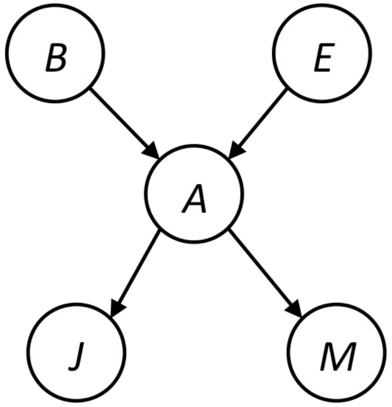
$P(m|A)$

$$\xrightarrow{\times} P(j,m,A|B,E) \xrightarrow{\sum} P(j,m|B,E)$$

$P(B)$	$P(E)$	$P(j,m\|B,E)$

Example (cont'd)

$P(B) \quad P(E) \quad P(j, m|B, E)$



Choose E

$P(E)$

$P(j, m|B, E)$

$\times \Rightarrow P(j, m, E|B) \quad \Sigma \Rightarrow P(j, m|B)$

$P(B) \quad P(j, m|B)$

Finish with B

$P(B)$

$P(j, m|B)$

$\times \Rightarrow P(j, m, B)$ Normalize $\Rightarrow P(B|j, m)$

Another Variable Elimination Example

Query: $P(X_3|Y_1 = y_1, Y_2 = y_2, Y_3 = y_3)$

Start by inserting evidence, which gives the following initial factors:

$$P(Z), P(X_1|Z), P(X_2|Z), P(X_3|Z), P(y_1|X_1), P(y_2|X_2), P(y_3|X_3)$$

Eliminate X_1, this introduces the factor $f_1(y_1|Z) = \sum_{x_1} P(x_1|Z)P(y_1|x_1)$, and we are left with:

$$P(Z), P(X_2|Z), P(X_3|Z), P(y_2|X_2), P(y_3|X_3), f_1(y_1|Z)$$

Eliminate X_2, this introduces the factor $f_2(y_2|Z) = \sum_{x_2} P(x_2|Z)P(y_2|x_2)$, and we are left with:

$$P(Z), P(X_3|Z), P(y_3|X_3), f_1(y_1|Z), f_2(y_2|Z)$$

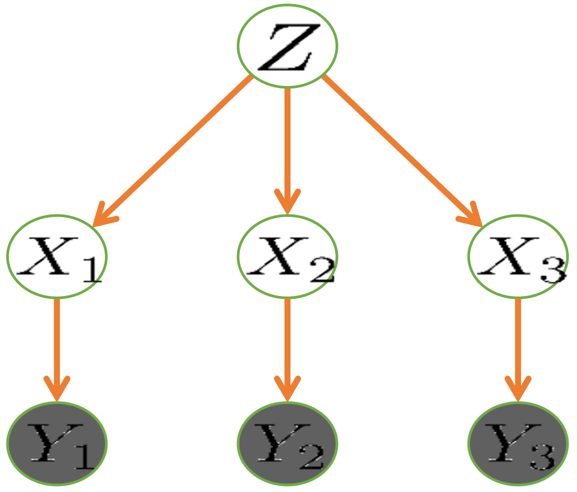
Eliminate Z, this introduces the factor $f_3(y_1, y_2, X_3) = \sum_z P(z)P(X_3|z)f_1(y_1|Z)f_2(y_2|Z)$, and we are left with:

$$P(y_3|X_3), f_3(y_1, y_2, X_3)$$

No hidden variables left. Join the remaining factors to get:

$$f_4(y_1, y_2, y_3, X_3) = P(y_3|X_3)\ \ f_3(y_1, y_2, X_3)$$

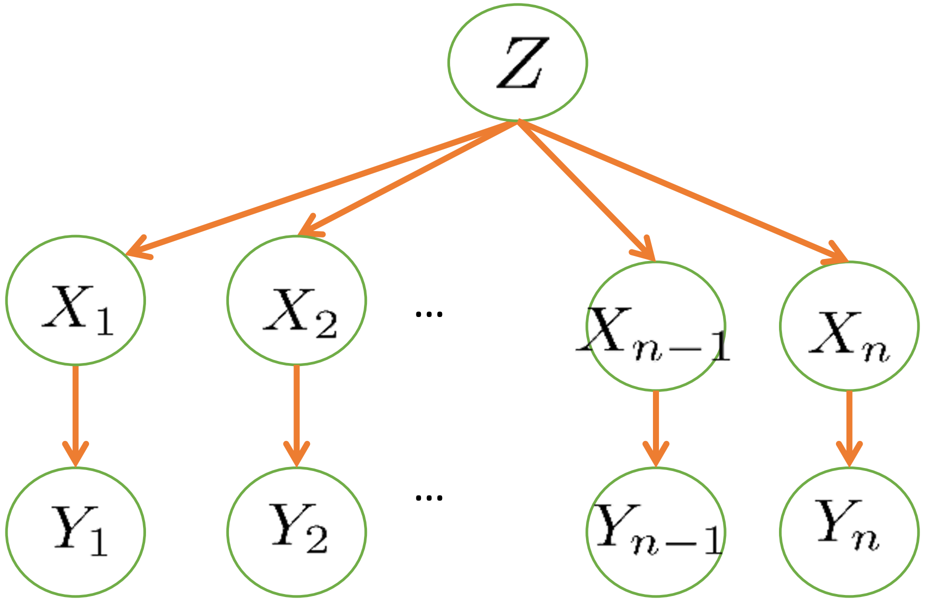
Normalizing over X_3 gives $P(X_3|y_1, y_2, y_3) = f_4(y_1, y_2, y_3, X_3) / \sum_{x_3} f_4(y_1, y_2, y_3, x_3)$



Computational complexity critically depends on the largest factor being generated in this process. Size of factor = number of entries in table. In example above (assuming binary) all factors generated are of size 2 --- as they all only have one variable (Z, Z, and X_3 respectively).

Variable Elimination Ordering

- For the query $P(X_n|y_1,\ldots,y_n)$ work through the following two different orderings as done in previous slide: $Z, X_1, \ldots, X_{n-1}$ and $X_1, \ldots, X_{n-1}, Z$. What is the size of the maximum factor generated for each of the orderings?



- Answer: 2^n versus 2 (assuming binary)

- In general: the ordering can greatly affect efficiency

VE: Computational and Space Complexity

- The computational and space complexity of variable elimination is determined by the largest factor

- The elimination ordering can greatly affect the size of the largest factor
  - E.g., previous slide's example 2^n vs. 2

- Does there always exist an ordering that only results in small factors?
  - No!

Worst Case Complexity?

- CSP:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_2 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_5 \vee x_7) \wedge (x_4 \vee x_5 \vee x_6) \wedge (\neg x_5 \vee x_6 \vee \neg x_7) \wedge (\neg x_5 \vee \neg x_6 \vee x_7)$$

$$P(X_i = 0) = P(X_i = 1) = 0.5$$

$$Y_1 = X_1 \vee X_2 \vee \neg X_3$$

$$\dots$$

$$Y_8 = \neg X_5 \vee X_6 \vee X_7$$

$$Y_{1,2} = Y_1 \wedge Y_2$$

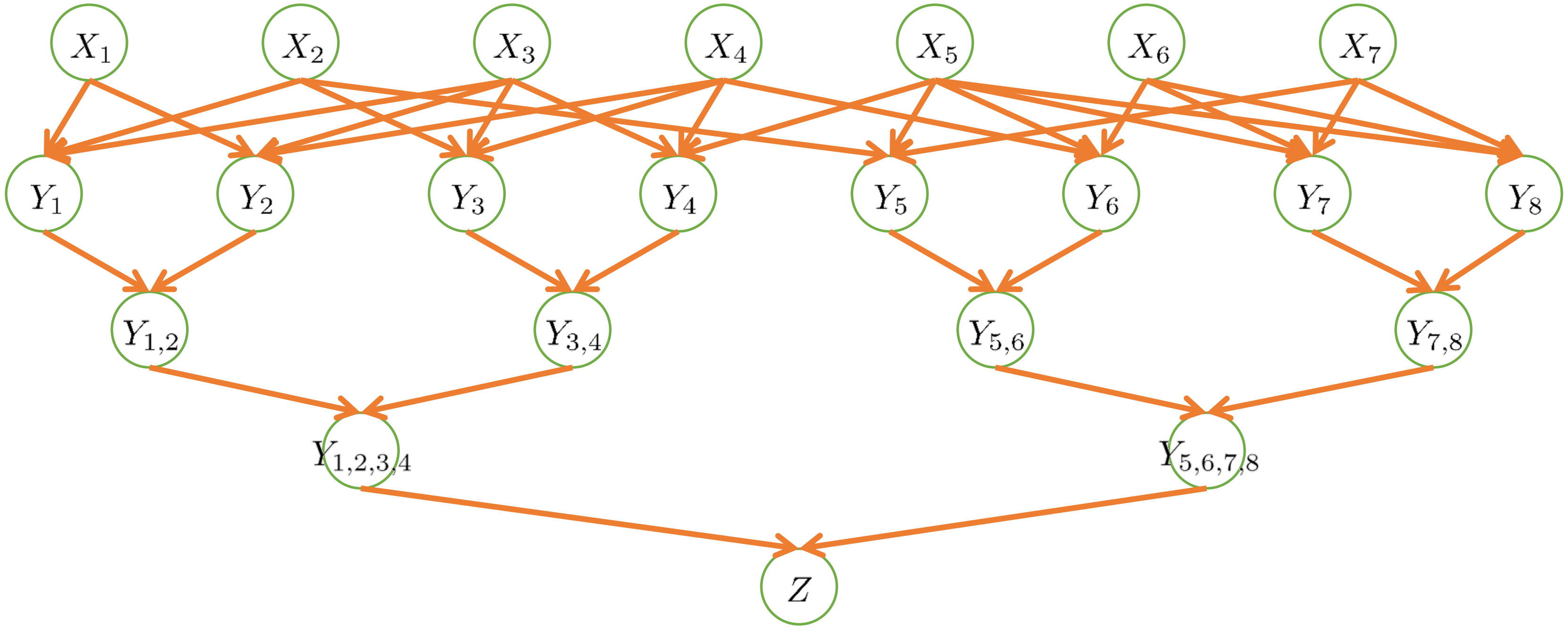
$$\dots$$

$$Y_{7,8} = Y_7 \wedge Y_8$$

$$Y_{1,2,3,4} = Y_{1,2} \wedge Y_{3,4}$$

$$Y_{5,6,7,8} = Y_{5,6} \wedge Y_{7,8}$$

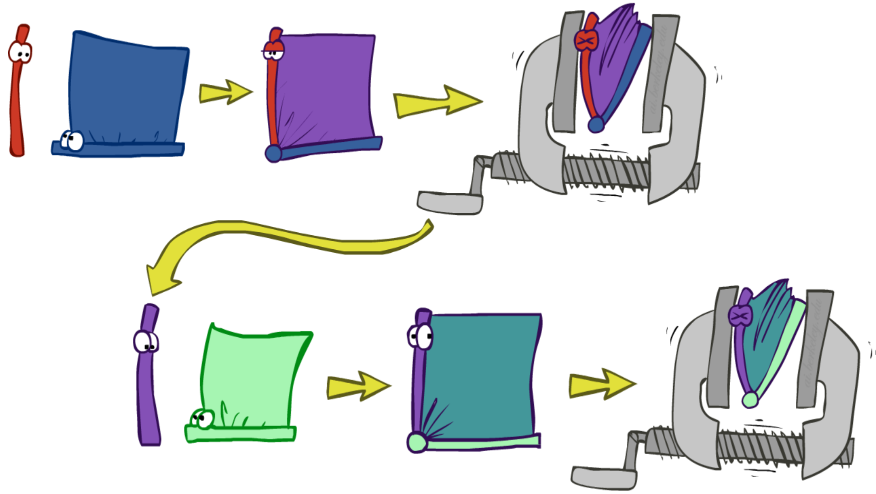
$$Z = Y_{1,2,3,4} \wedge Y_{5,6,7,8}$$



- If we can answer P(z) equal to zero or not, we answered whether the 3-SAT problem has a solution
- Hence inference in Bayes' nets is NP-hard. No known efficient probabilistic inference in general

Variable Elimination: The basic ideas

- Move summations inwards as far as possible
  - $P(B \mid j, m) = \alpha \sum_e \sum_a P(B)\, P(e)\, P(a|B,e)\, P(j|a)\, P(m|a)$
    $= \alpha\, P(B) \sum_e P(e) \sum_a P(a|B,e)\, P(j|a)\, P(m|a)$



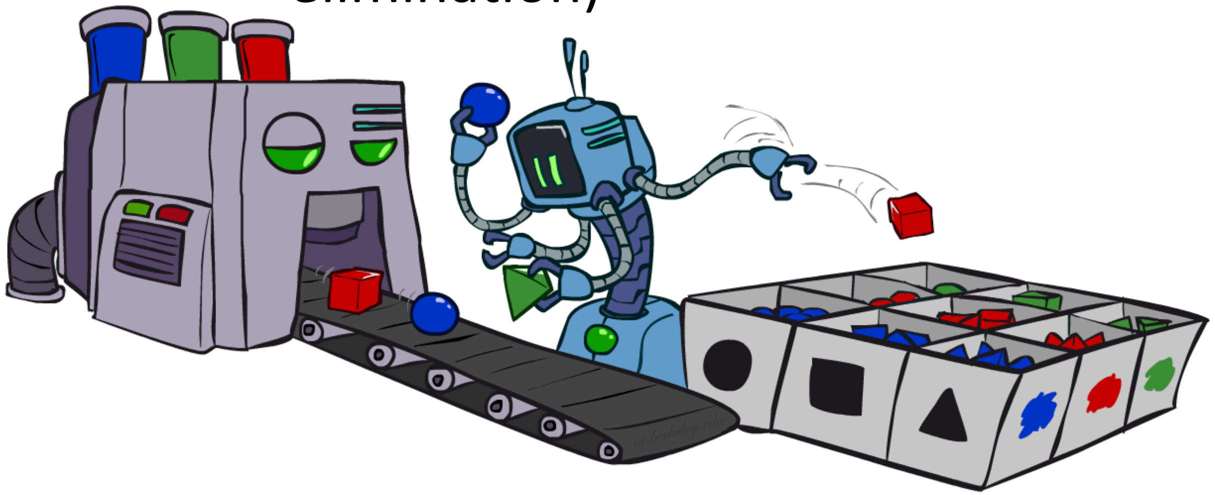
- Do the calculation from the inside out
  - I.e., sum over *a* first, then sum over *e*
  - Problem: $P(a|B,e)$ isn't a single number, it's a bunch of different numbers depending on the values of *B* and *e*
  - Solution: use arrays of numbers (of various dimensions) with appropriate operations on them; these are called factors

Sampling

Sampling

- Sampling is a lot like repeated simulation
  - Predicting the weather, basketball games, …
- Basic idea
  - Draw N samples from a sampling distribution S
  - Compute an approximate posterior probability
  - Show this converges to the true probability P

- Why sample?
  - Learning: get samples from a distribution you don't know
  - Inference: getting a sample is faster than computing the right answer (e.g. with variable elimination)



Sampling 2

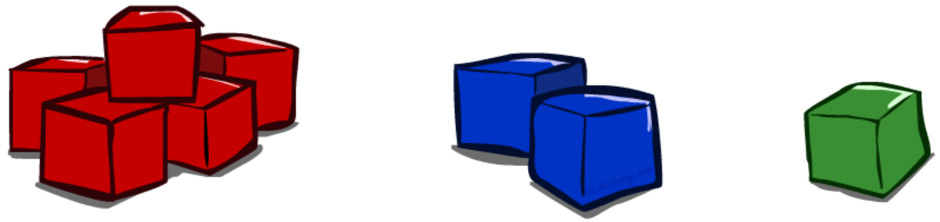
- Sampling from given distribution
  - Step 1: Get sample u from uniform distribution over [0, 1)
    - E.g. random() in python
  - Step 2: Convert this sample u into an outcome for the given distribution by having each target outcome associated with a sub-interval of [0,1) with sub-interval size equal to probability of the outcome

- Example

C	P(C)
red	0.6
green	0.1
blue	0.3

$$0 \leq u < 0.6, \rightarrow C = red$$
$$0.6 \leq u < 0.7, \rightarrow C = green$$
$$0.7 \leq u < 1, \rightarrow C = blue$$

- If random() returns u = 0.83, then our sample is C = blue
- E.g, after sampling 8 times:



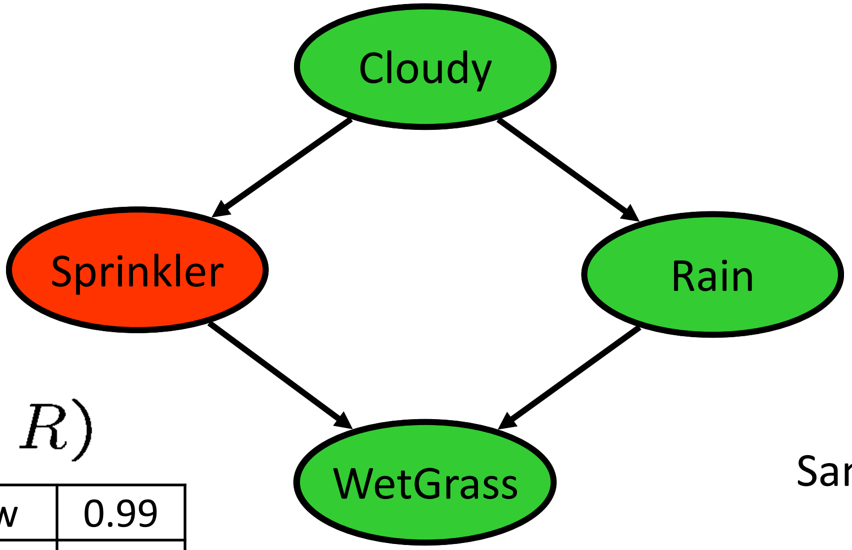
Sampling in Bayes' Nets

- Prior Sampling
- Rejection Sampling
- Likelihood Weighting
- Gibbs Sampling

Prior Sampling: Example

$P(C)$

+c	0.5
-c	0.5



$P(S|C)$

+c	+s	0.1
	-s	0.9
-c	+s	0.5
	-s	0.5

$P(R|C)$

+c	+r	0.8
	-r	0.2
-c	+r	0.2
	-r	0.8

$P(W|S, R)$

+s	+r	+w	0.99
		-w	0.01
	-r	+w	0.90
		-w	0.10
-s	+r	+w	0.90
		-w	0.10
	-r	+w	0.01
		-w	0.99

Samples:

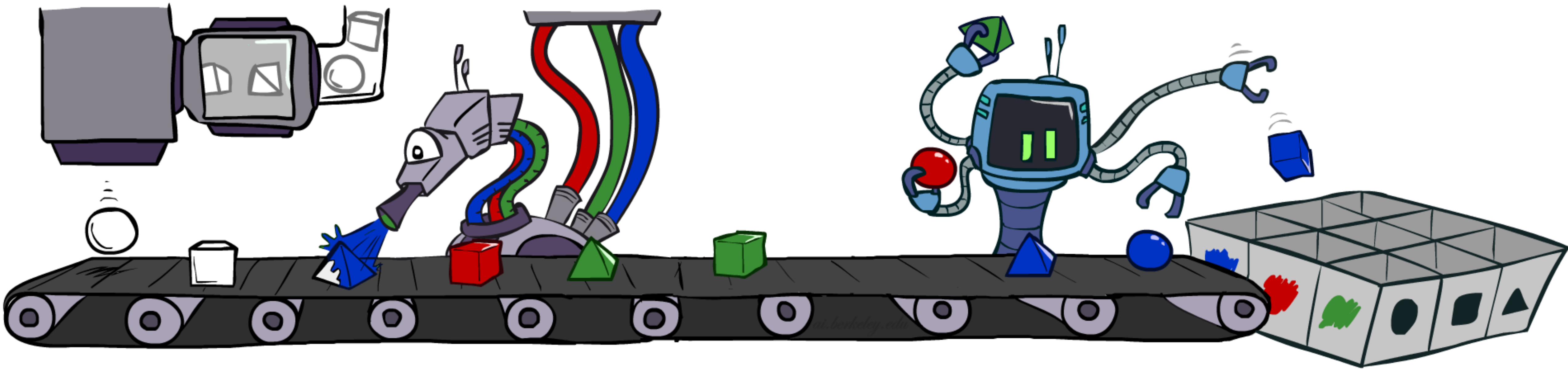
+c, -s, +r, +w

-c, +s, -r, +w

…

Prior Sampling: Algorithm

- For i = 1, 2, …, n in topological order
  - Sample x_i from $P(X_i \mid Parents(X_i))$
- Return $(x_1, x_2, \ldots, x_n)$



Prior Sampling

- This process generates samples with probability:

$$S_{PS}(x_1 \dots x_n) = \prod_{i=1}^{n} P(x_i|\text{Parents}(X_i)) = P(x_1 \dots x_n)$$

- …i.e. the BN's joint probability

- Let the number of samples of an event be $N_{PS}(x_1 \dots x_n)$

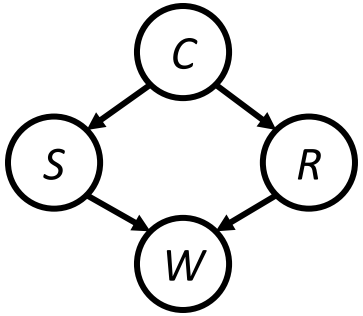
- Then

$$\begin{aligned}\lim_{N\to\infty} \widehat{P}(x_1, \dots, x_n) &= \lim_{N\to\infty} N_{PS}(x_1, \dots, x_n)/N \\ &= S_{PS}(x_1, \dots, x_n) \\ &= P(x_1 \dots x_n)\end{aligned}$$

- i.e., the sampling procedure is consistent

Example

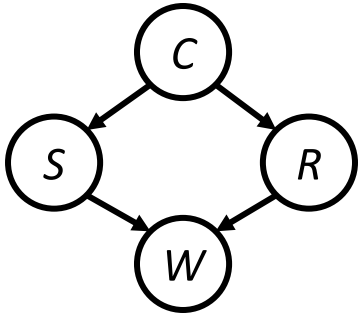
- We'll get a bunch of samples from the BN:
  - +c, -s, +r, +w
  - +c, +s, +r, +w
  - -c, +s, +r, -w
  - +c, -s, +r, +w
  - -c, -s, -r, +w



- If we want to know P(W)
  - We have counts <+w:4, -w:1>
  - Normalize to get P(W) = <+w:0.8, -w:0.2>
  - This will get closer to the true distribution with more samples
  - Can estimate anything else, too
    - P(C | +w)? P(C | +r, +w)?
    - Can also use this to estimate expected value of f(X) - Monte Carlo Estimation
  - What about P(C | -r, -w)?

Rejection Sampling

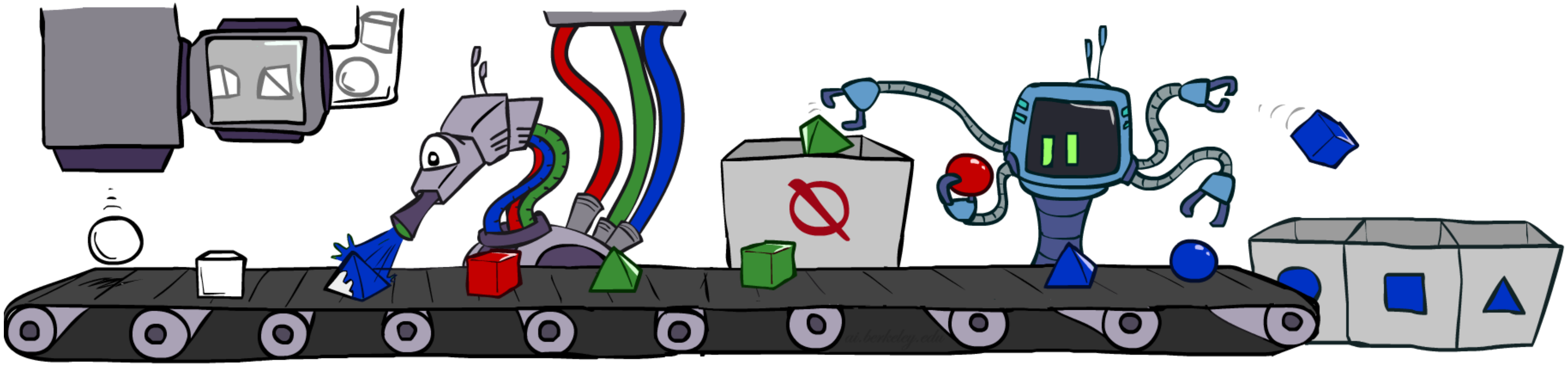
- Let's say we want P(C)
  - Just tally counts of C as we go



- Let's say we want P(C | +s)
  - Same thing: tally C outcomes, but ignore (reject) samples which don't have S=+s
  - This is called rejection sampling
  - We can toss out samples early!
  - It is also consistent for conditional probabilities (i.e., correct in the limit)

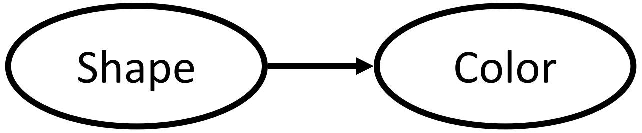
Rejection Sampling: Algorithm

- Input: evidence instantiation
- For i = 1, 2, …, n in topological order
  - Sample x_i from $P(X_i \mid Parents(X_i))$
  - If x_i not consistent with evidence
    - Reject: return – no sample is generated in this cycle
- Return $(x_1, x_2, \ldots, x_n)$

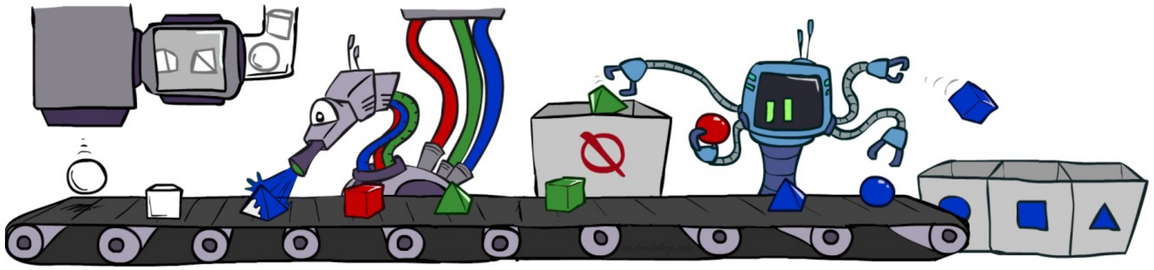


Likelihood Weighting

- Problem with rejection sampling:
  - If evidence is unlikely, rejects lots of samples
  - Consider P(Shape | blue)

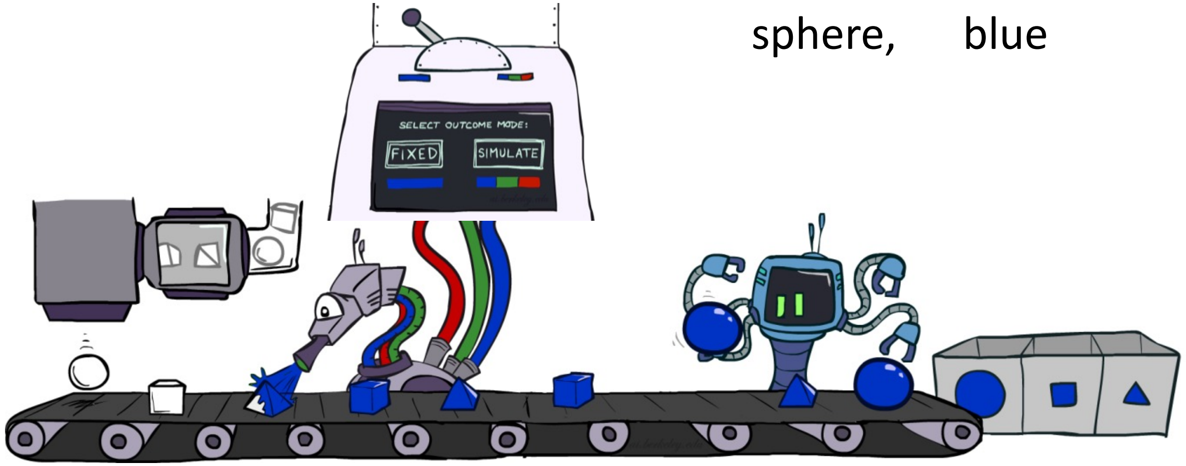


~~pyramid, green~~
~~pyramid, red~~
sphere, blue
~~cube, red~~
~~sphere, green~~



- Idea: fix evidence variables and sample the rest
  - Problem: sample distribution not consistent!
  - Solution: weight by probability of evidence given parents

pyramid, blue
pyramid, blue
sphere, blue
cube, blue
sphere, blue



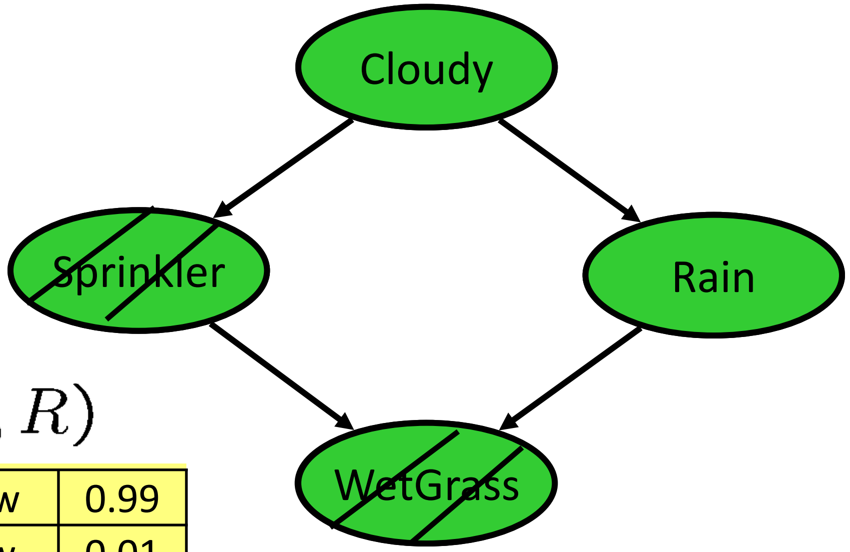
Likelihood Weighting: Example

$P(C)$

+c	0.5
-c	0.5

$P(S|C)$

+c	+s	0.1
	-s	0.9
-c	+s	0.5
	-s	0.5



$P(R|C)$

+c	+r	0.8
	-r	0.2
-c	+r	0.2
	-r	0.8

$P(W|S,R)$

+s	+r	+w	0.99
		-w	0.01
	-r	+w	0.90
		-w	0.10
-s	+r	+w	0.90
		-w	0.10
	-r	+w	0.01
		-w	0.99

Samples:

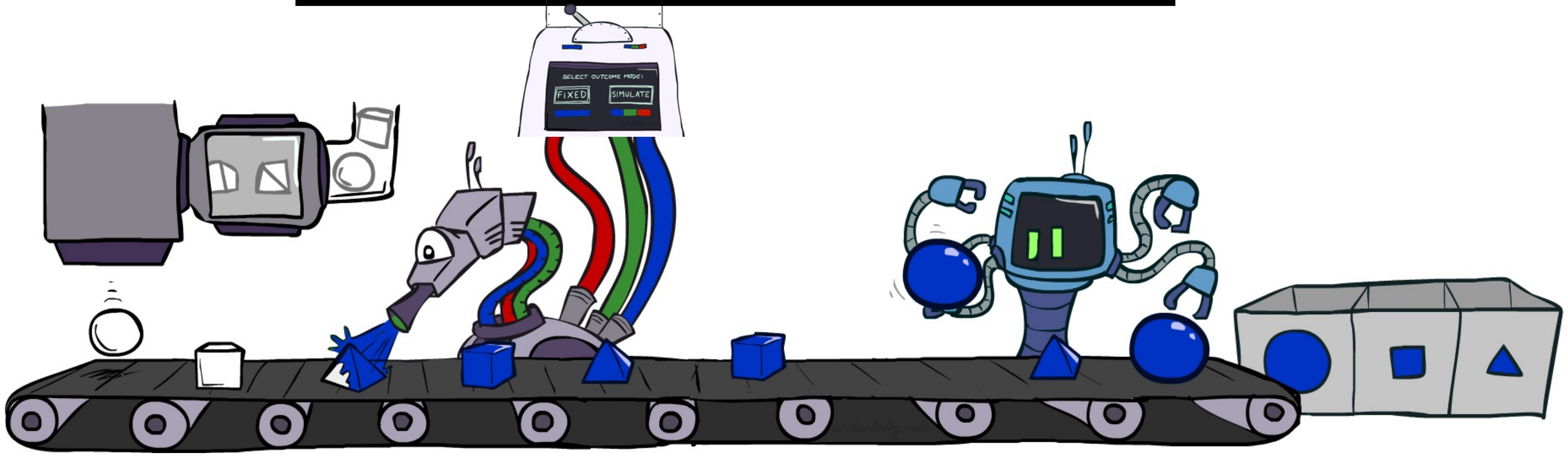
+c, +s, +r, +w   w = 1.0 x

-c, +s, -r, +w   w = 1.0 x 0.5 x 0.90

…

Likelihood Weighting: Algorithm

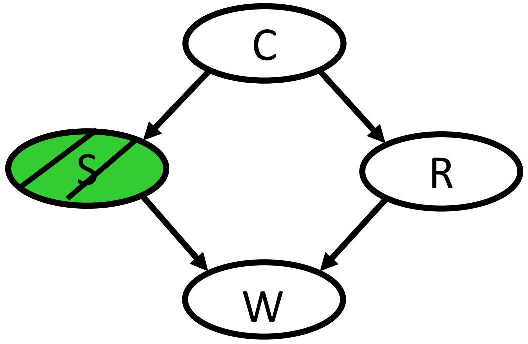
- Input: evidence instantiation
- $w = 1.0$
- for $i = 1, 2, \ldots, n$ in topological order
  - if X_i is an evidence variable
    - X_i = observation x_i for X_i
    - Set $w = w * P(x_i \mid Parents(X_i))$
  - else
    - Sample x_i from $P(X_i \mid Parents(X_i))$
- return $(x_1, x_2, \ldots, x_n)$, w



Likelihood Weighting

- Sampling distribution if z sampled and e fixed evidence

$$S_{WS}(\mathbf{z},\mathbf{e}) = \prod_{i=1}^{l} P(z_i|\text{Parents}(Z_i))$$



- Now, samples have weights

$$w(\mathbf{z},\mathbf{e}) = \prod_{i=1}^{m} P(e_i|\text{Parents}(E_i))$$

- Together, weighted sampling distribution is consistent

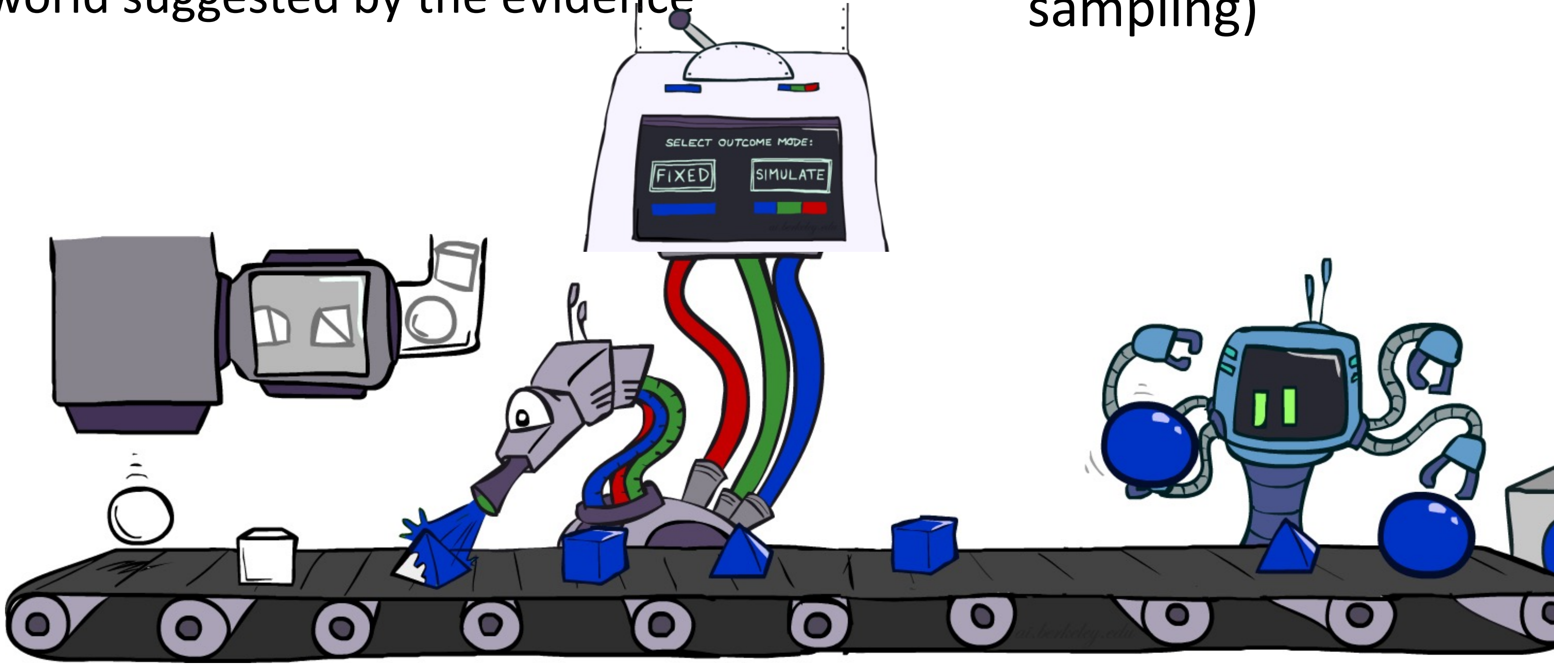
$$S_{\text{WS}}(z,e) \cdot w(z,e) = \prod_{i=1}^{l} P(z_i|\text{Parents}(z_i)) \prod_{i=1}^{m} P(e_i|\text{Parents}(e_i))$$

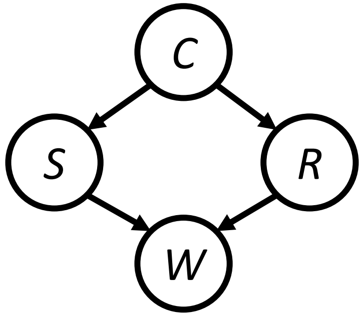
$$= P(\mathbf{z},\mathbf{e})$$

Likelihood Weighting

- Likelihood weighting is helpful
  - We have taken evidence into account as we generate the sample
  - E.g. here, W's value will get picked based on the evidence values of S, R
  - More of our samples will reflect the state of the world suggested by the evidence

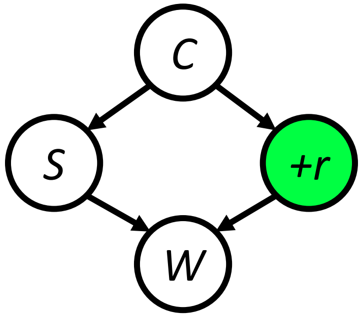
- Likelihood weighting doesn't solve all our problems
  - Evidence influences the choice of downstream variables, but not upstream ones (C isn't more likely to get a value matching the evidence)
- We would like to consider evidence when we sample every variable (leads to Gibbs sampling)



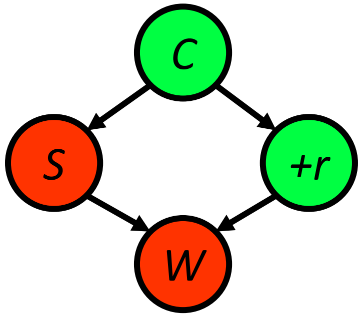


Gibbs Sampling: Example P(S | +r)

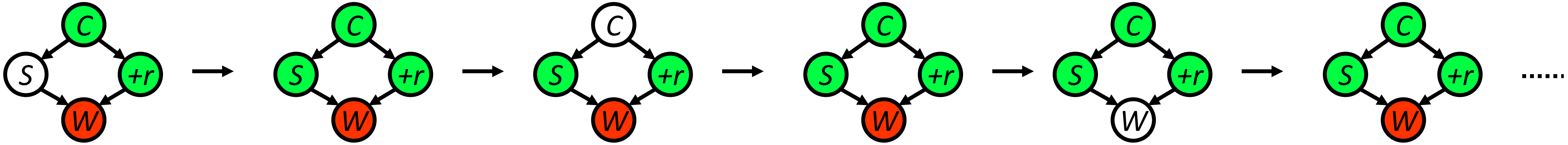
- Step 1: Fix evidence
  - R = +r



- Step 2: Initialize other variables
  - Randomly



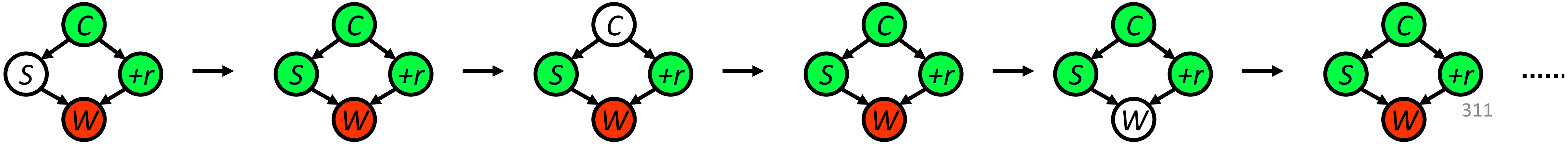
- Steps 3: Repeat
  - Choose a non-evidence variable X
  - Resample X from P(X | all other variables)*



Sample from $P(S|+c,-w,+r)$ Sample from $P(C|+s,-w,+r)$ Sample from $P(W|+s,+c,+r)$

Gibbs Sampling

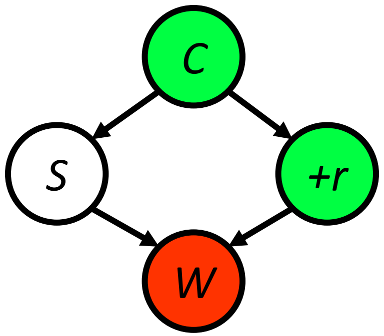
- Procedure
  - Keep track of a full instantiation $x_1, \ldots, x_n$
  - Start with an arbitrary instantiation consistent with the evidence
  - Sample one variable at a time, conditioned on all the rest, but keep evidence fixed
  - Keep repeating this for a long time
- Property
  - In the limit of repeating this infinitely many times the resulting samples come from the correct distribution (i.e. conditioned on evidence)
- Rationale
  - Both upstream and downstream variables condition on evidence
- In contrast:
  - Likelihood weighting only conditions on upstream evidence, and hence weights obtained in likelihood weighting can sometimes be very small
  - Sum of weights over all samples is indicative of how many "effective" samples were obtained, so we want high weight



Resampling of One Variable

- Sample from P(S | +c, +r, -w)

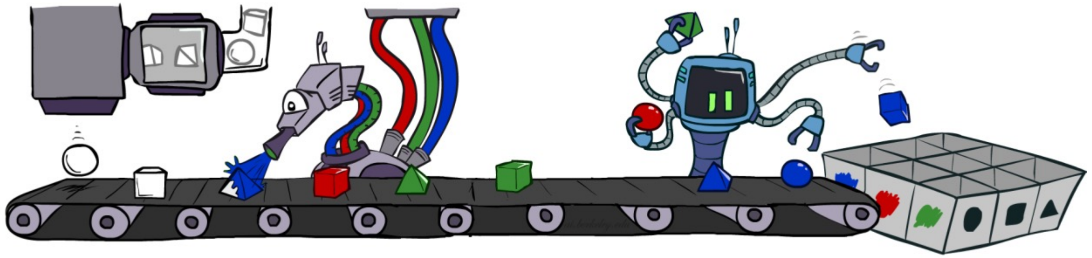
$$
\begin{aligned}
P(S|+c,+r,-w) &= \frac{P(S,+c,+r,-w)}{P(+c,+r,-w)} \\
&= \frac{P(S,+c,+r,-w)}{\sum_s P(s,+c,+r,-w)} \\
&= \frac{P(+c)P(S|+c)P(+r|+c)P(-w|S,+r)}{\sum_s P(+c)P(s|+c)P(+r|+c)P(-w|s,+r)} \\
&= \frac{P(+c)P(S|+c)P(+r|+c)P(-w|S,+r)}{P(+c)P(+r|+c)\sum_s P(s|+c)P(-w|s,+r)} \\
&= \frac{P(S|+c)P(-w|S,+r)}{\sum_s P(s|+c)P(-w|s,+r)}
\end{aligned}
$$



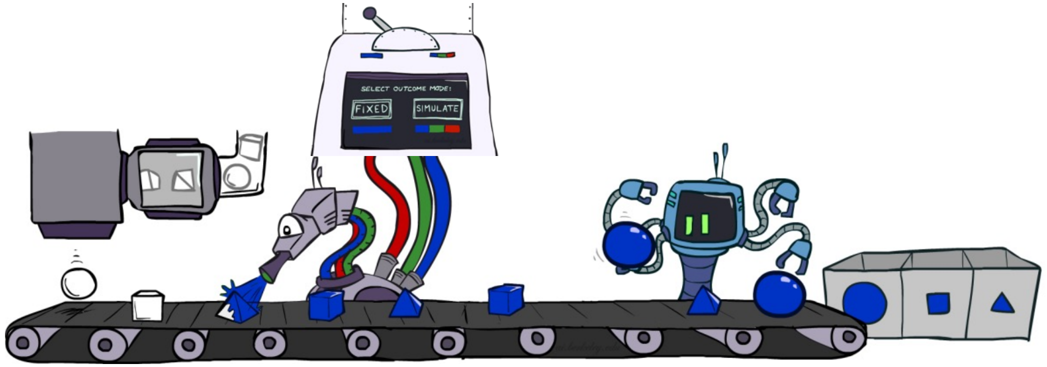
- Many things cancel out – only CPTs with S remain!
- More generally: only CPTs that have resampled variable need to be considered, and joined together

Bayes' Net Sampling Summary

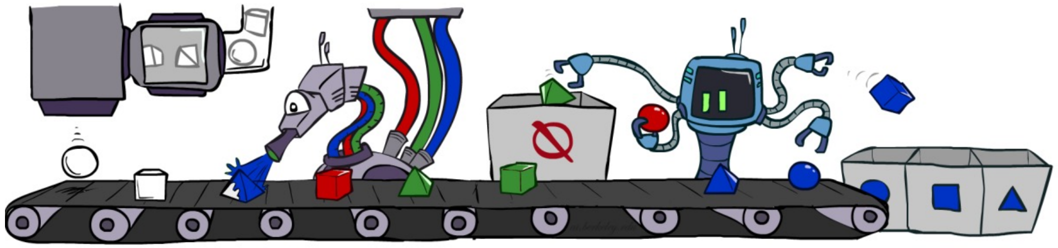
- Prior Sampling P(Q)



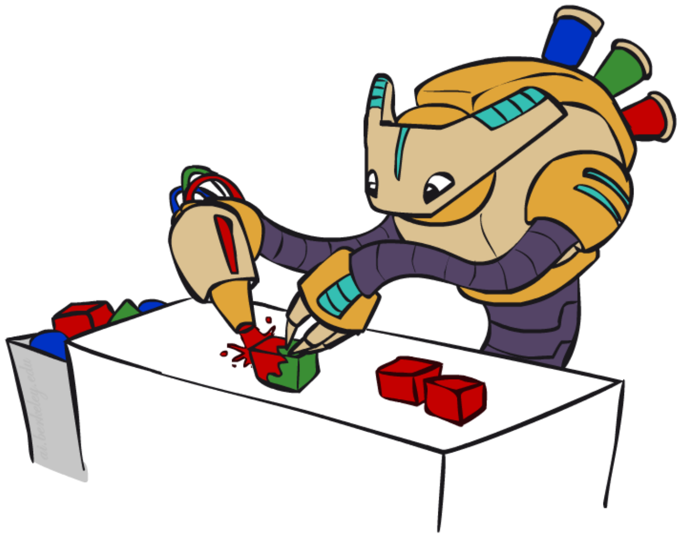
- Likelihood Weighting P(Q|e)



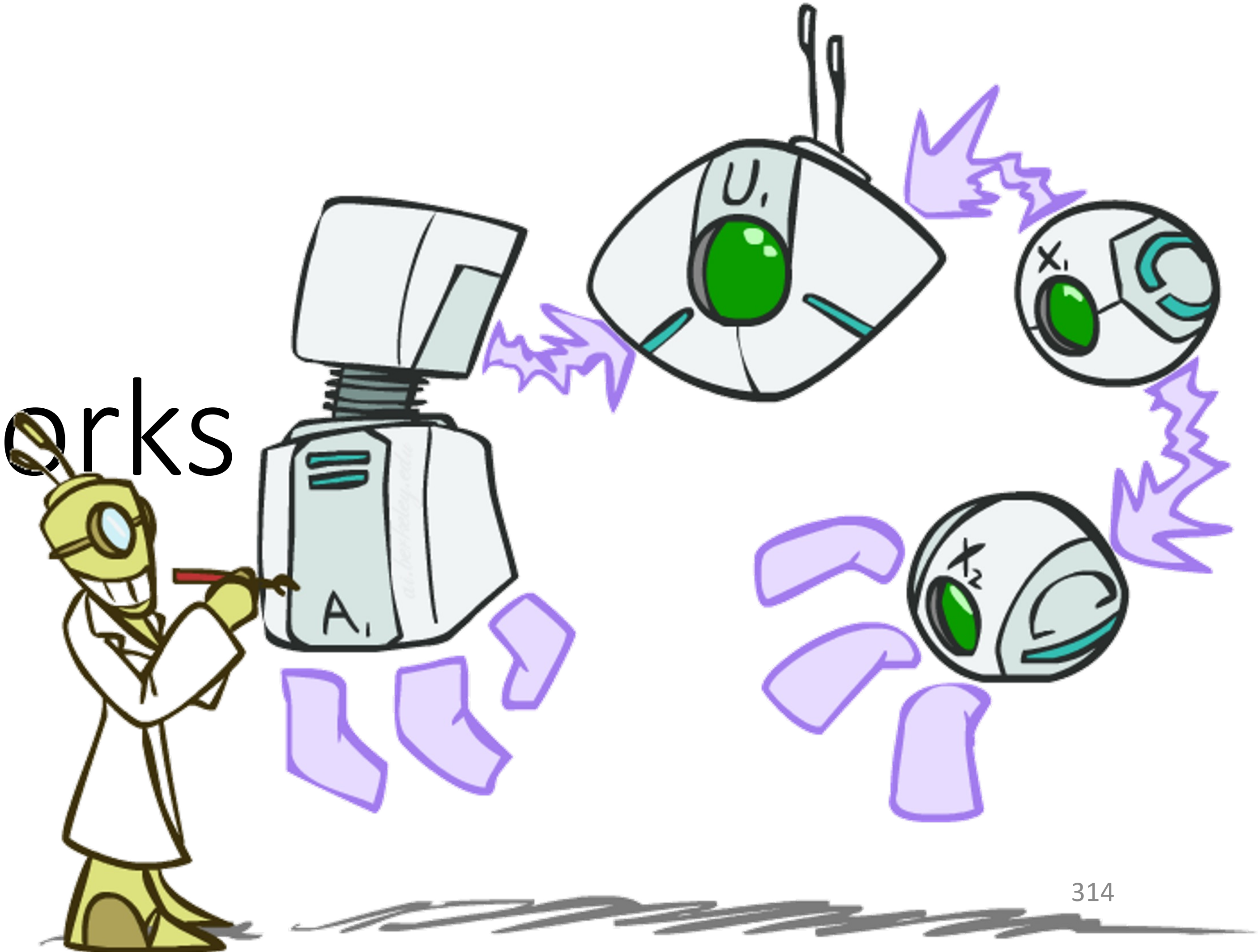
- Rejection Sampling P(Q|e)



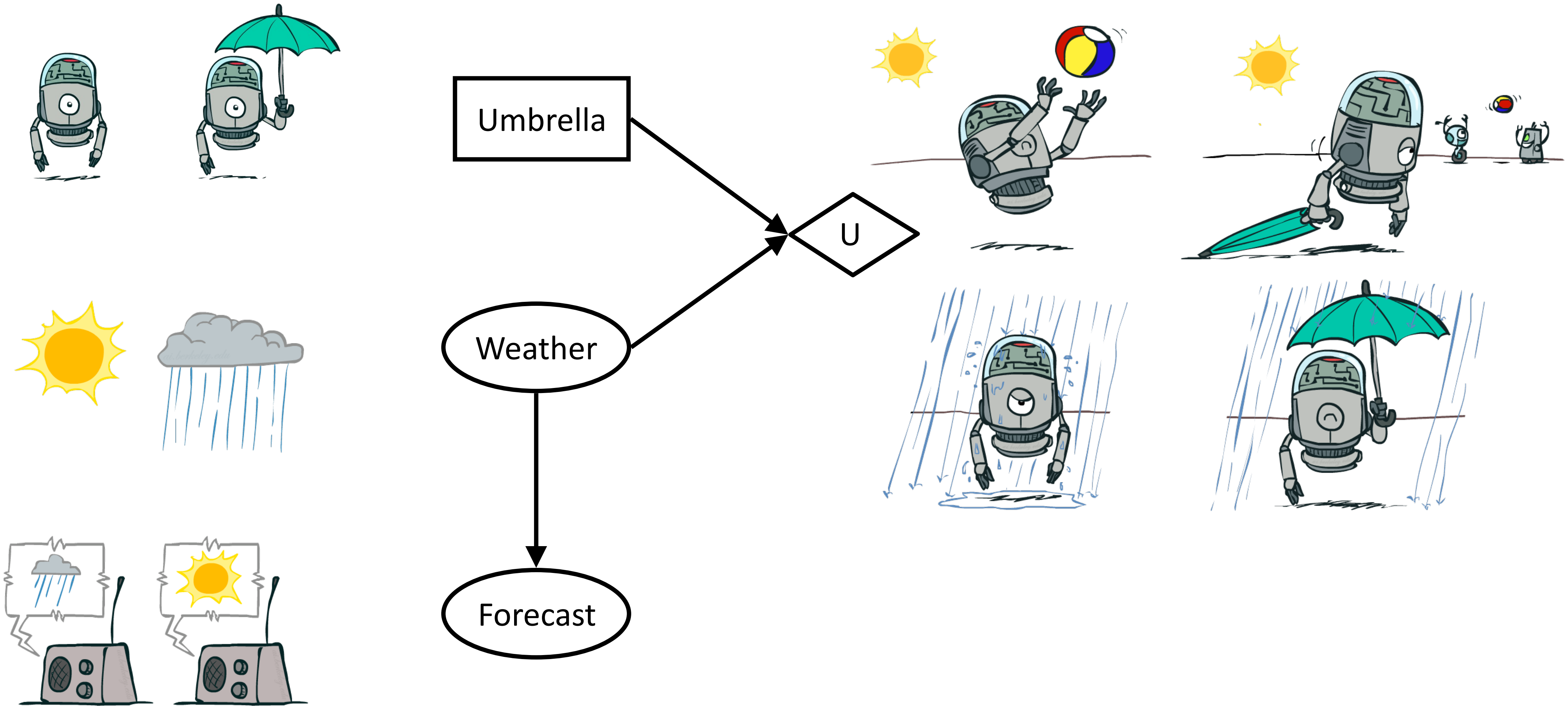
- Gibbs Sampling P(Q|e)



Decision Networks



Decision Networks

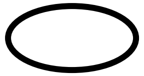


Decision Networks 2

- **MEU: choose the action which maximizes the expected utility given the evidence**

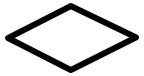
- Can directly operationalize this with decision networks
  - Bayes nets with nodes for utility and actions
  - Lets us calculate the expected utility for each action

- New node types:

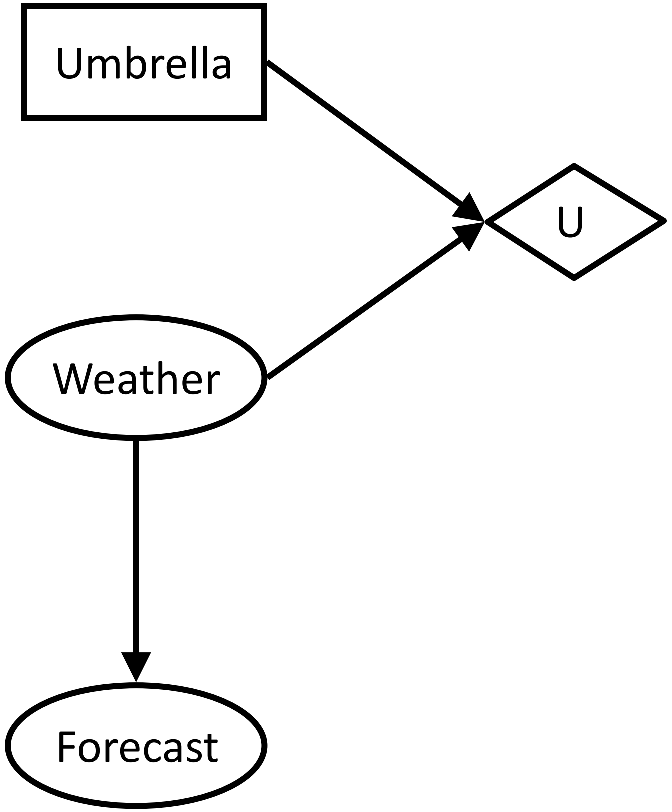


  - Chance nodes (just like BNs)

  - Actions (rectangles, cannot have parents, act as observed evidence)

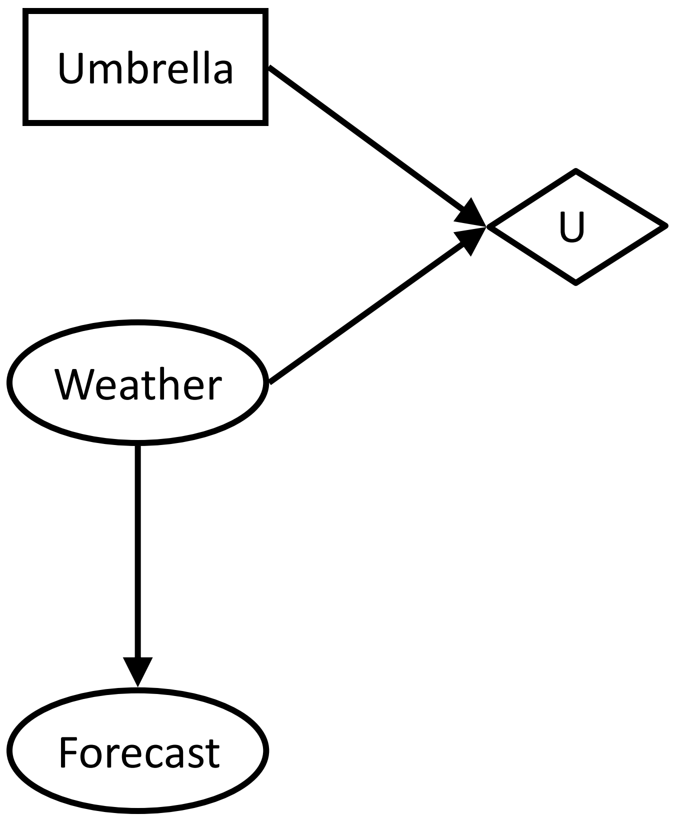


  - Utility node (diamond, depends on action and chance nodes)



Decision Networks 3

- Action selection
  - Instantiate all evidence
  - Set action node(s) each possible way
  - Calculate posterior for all parents of utility node, given the evidence
  - Calculate expected utility for each action
  - Choose maximizing action



Maximum Expected Utility

Umbrella = leave

$$\text{EU}(\text{leave}) = \sum_w P(w)U(\text{leave}, w)$$

$$= 0.7 \cdot 100 + 0.3 \cdot 0 = 70$$

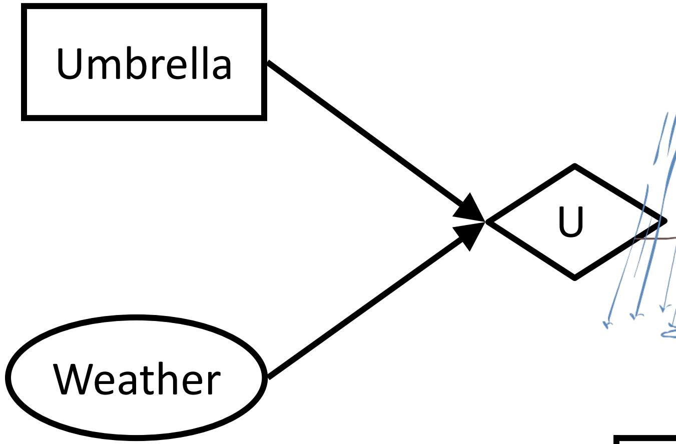
Umbrella = take

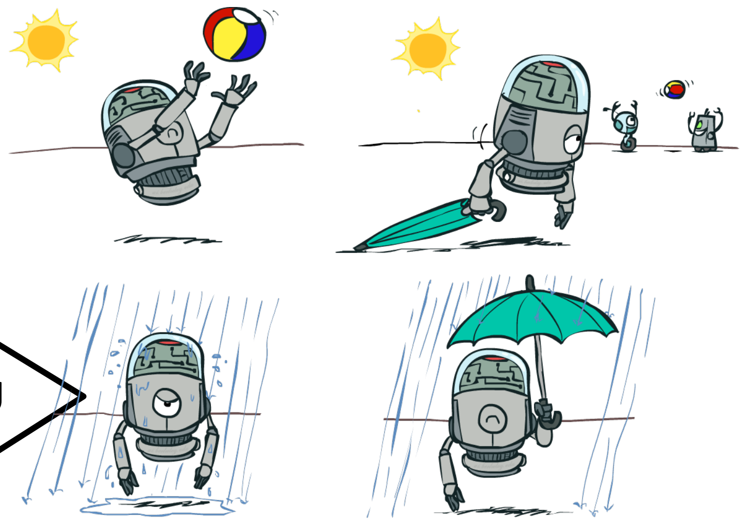
$$\text{EU}(\text{take}) = \sum_w P(w)U(\text{take}, w)$$

$$= 0.7 \cdot 20 + 0.3 \cdot 70 = 35$$

Optimal decision = leave

$$\text{MEU}(\emptyset) = \max_a \text{EU}(a) = 70$$

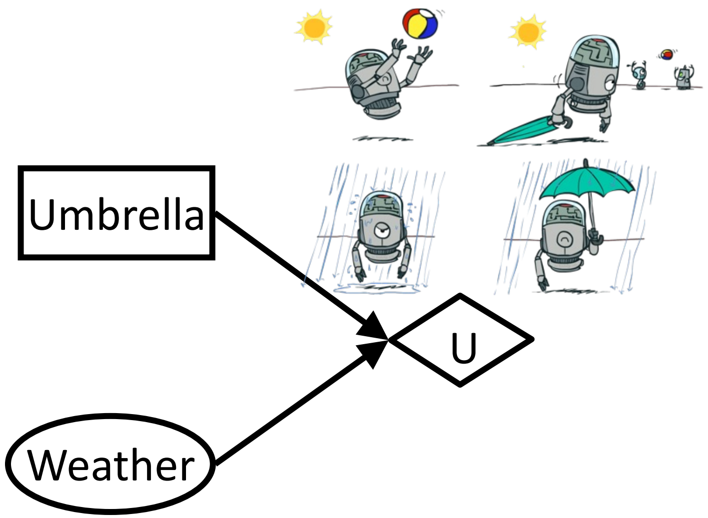


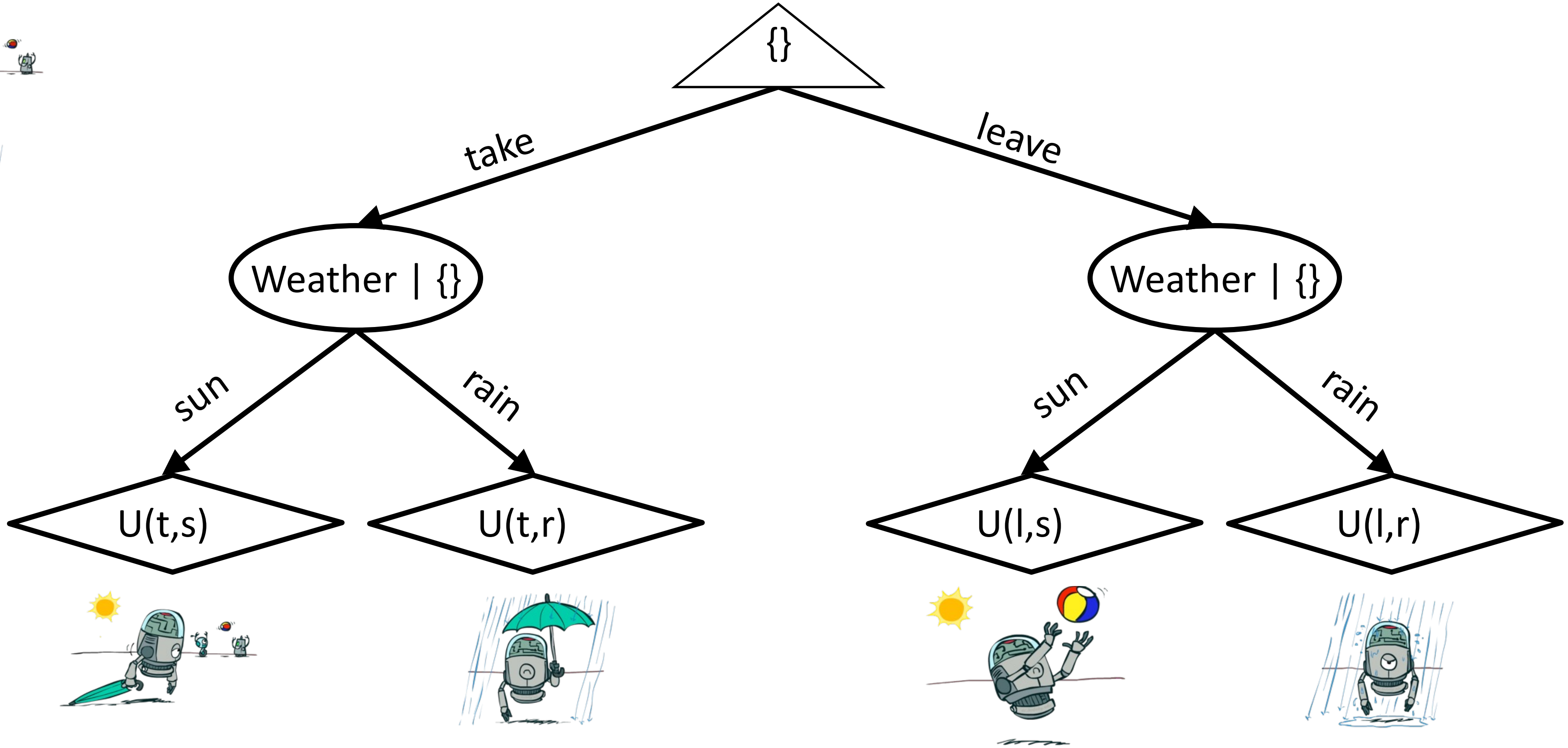


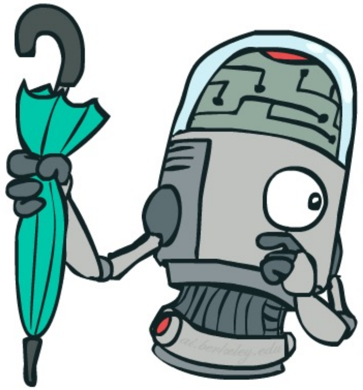
W	P(W)
sun	0.7
rain	0.3

A	W	U(A,W)
leave	sun	100
leave	rain	0
take	sun	20
take	rain	70

Decisions as Outcome Trees







- Almost exactly like expectimax / MDPs
- What's changed?

Maximum Expected Utility

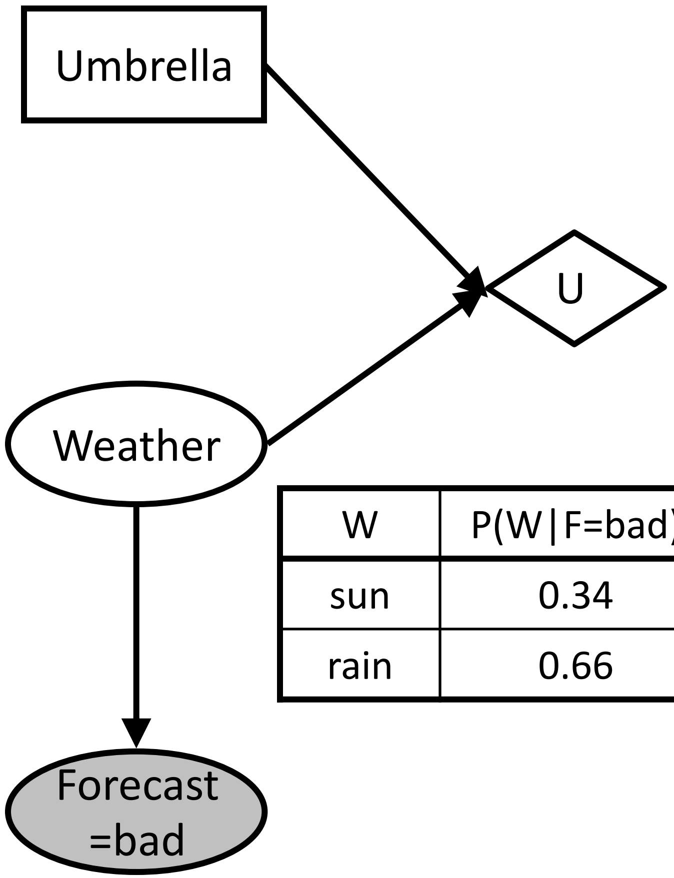
Umbrella = leave

$$\mathrm{EU}(\mathrm{leave}|\mathrm{bad}) = \sum_w P(w|\mathrm{bad})U(\mathrm{leave}, w)$$

$$P(W) \qquad P(F|W)$$

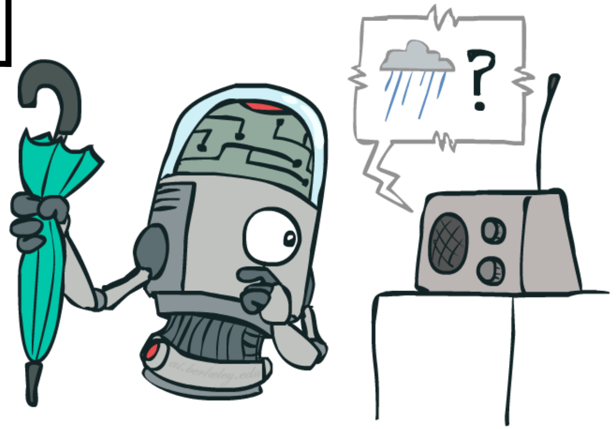
$$P(W|F) = \frac{P(W,F)}{\sum_w P(w,F)}$$

$$= \frac{P(F|W)P(W)}{\sum_w P(F|w)P(w)}$$



A	W	U(A,W)
leave	sun	100
leave	rain	0
take	sun	20
take	rain	70

W	P(W|F=bad)
sun	0.34
rain	0.66



Maximum Expected Utility 2

Umbrella = leave

$$\mathrm{EU}(\mathrm{leave}|\mathrm{bad}) = \sum_w P(w|\mathrm{bad})U(\mathrm{leave}, w)$$

$$= 0.34 \cdot 100 + 0.66 \cdot 0 = 34$$

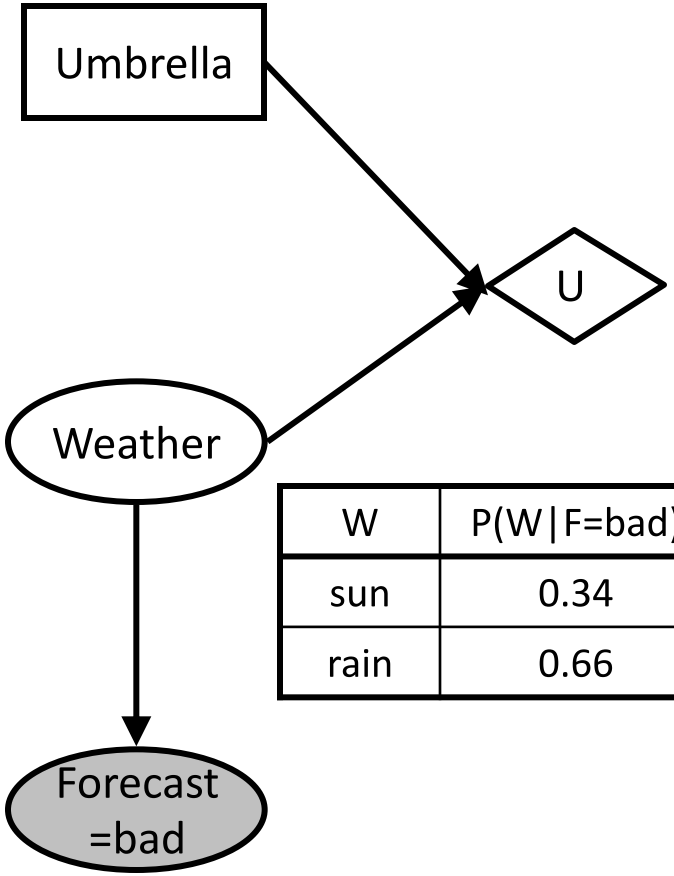
Umbrella = take

$$\mathrm{EU}(\mathrm{take}|\mathrm{bad}) = \sum_w P(w|\mathrm{bad})U(\mathrm{take}, w)$$

$$= 0.34 \cdot 20 + 0.66 \cdot 70 = 53$$

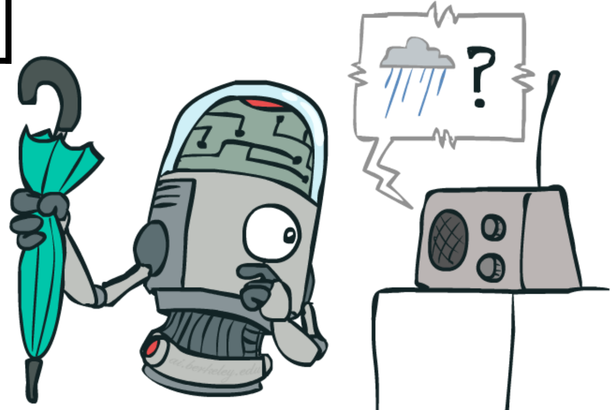
Optimal decision = take

$$\mathrm{MEU}(F = \mathrm{bad}) = \max_a \mathrm{EU}(a|\mathrm{bad}) = 53$$

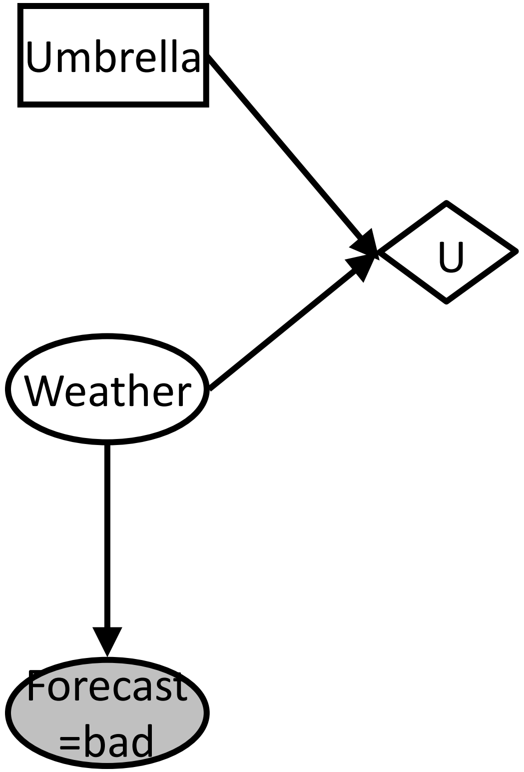


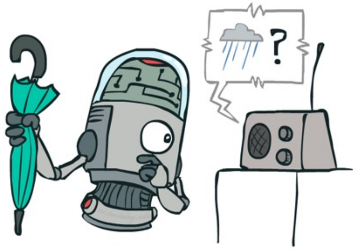
A	W	U(A,W)
leave	sun	100
leave	rain	0
take	sun	20
take	rain	70

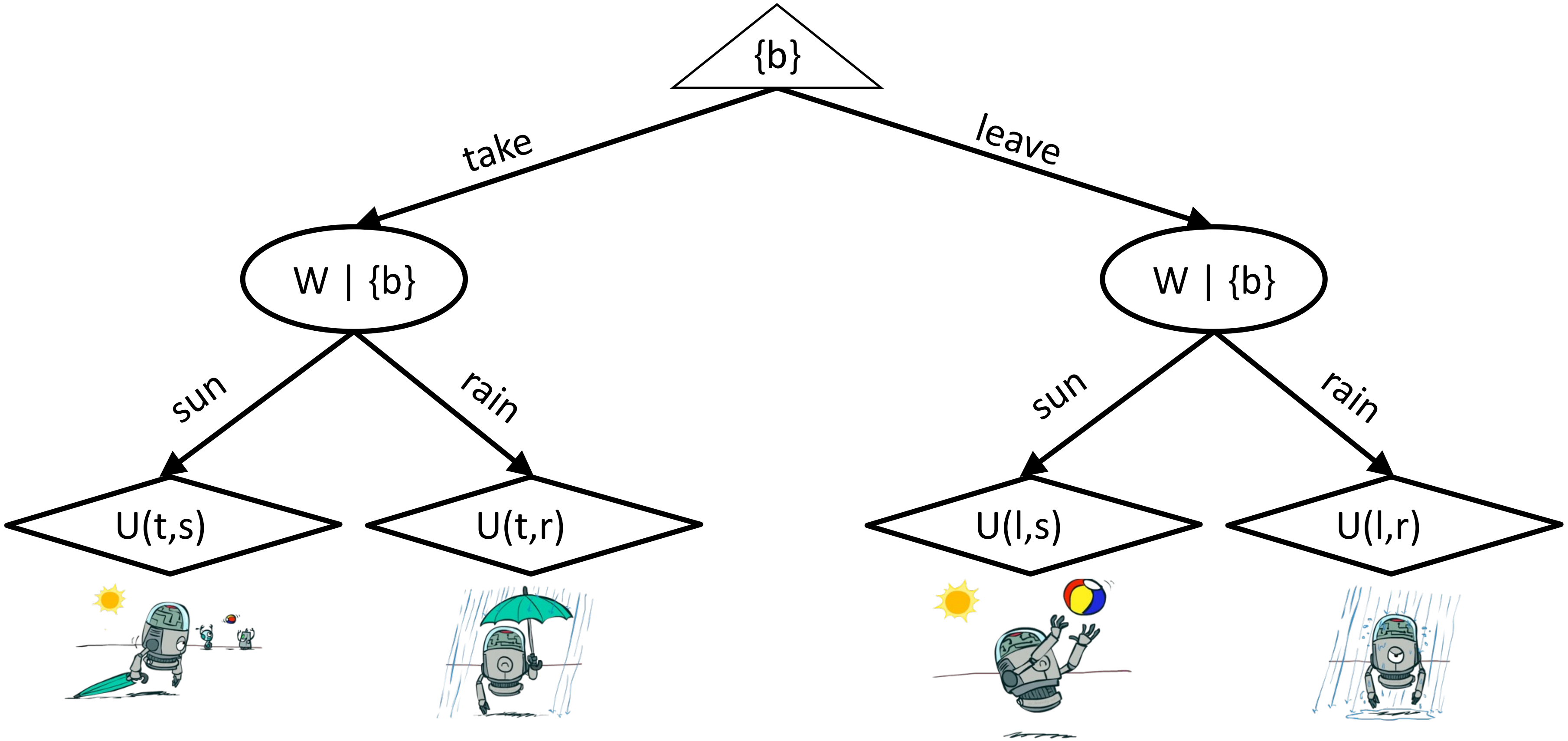
W	P(W|F=bad)
sun	0.34
rain	0.66



Decisions as Outcome Trees



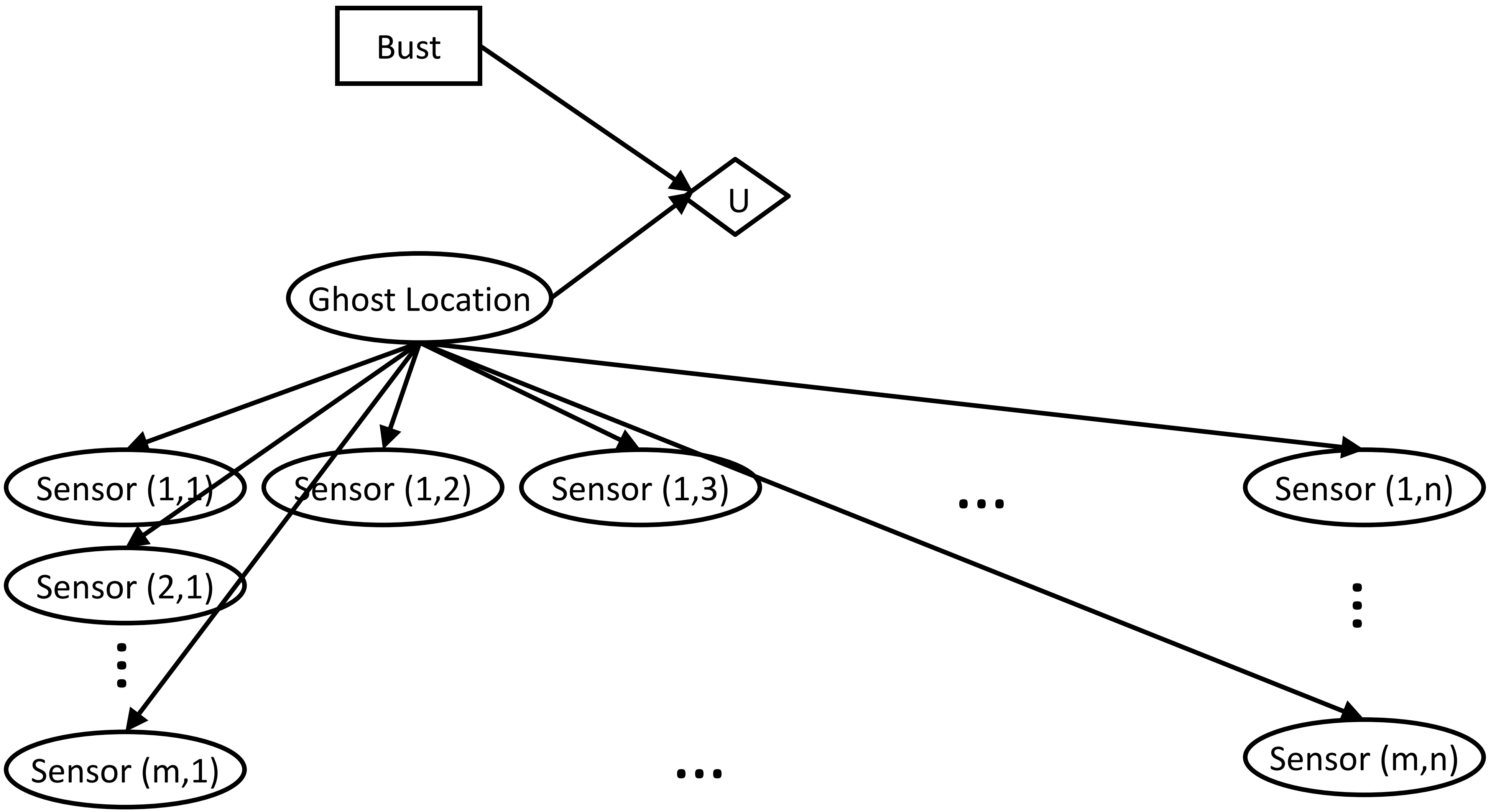




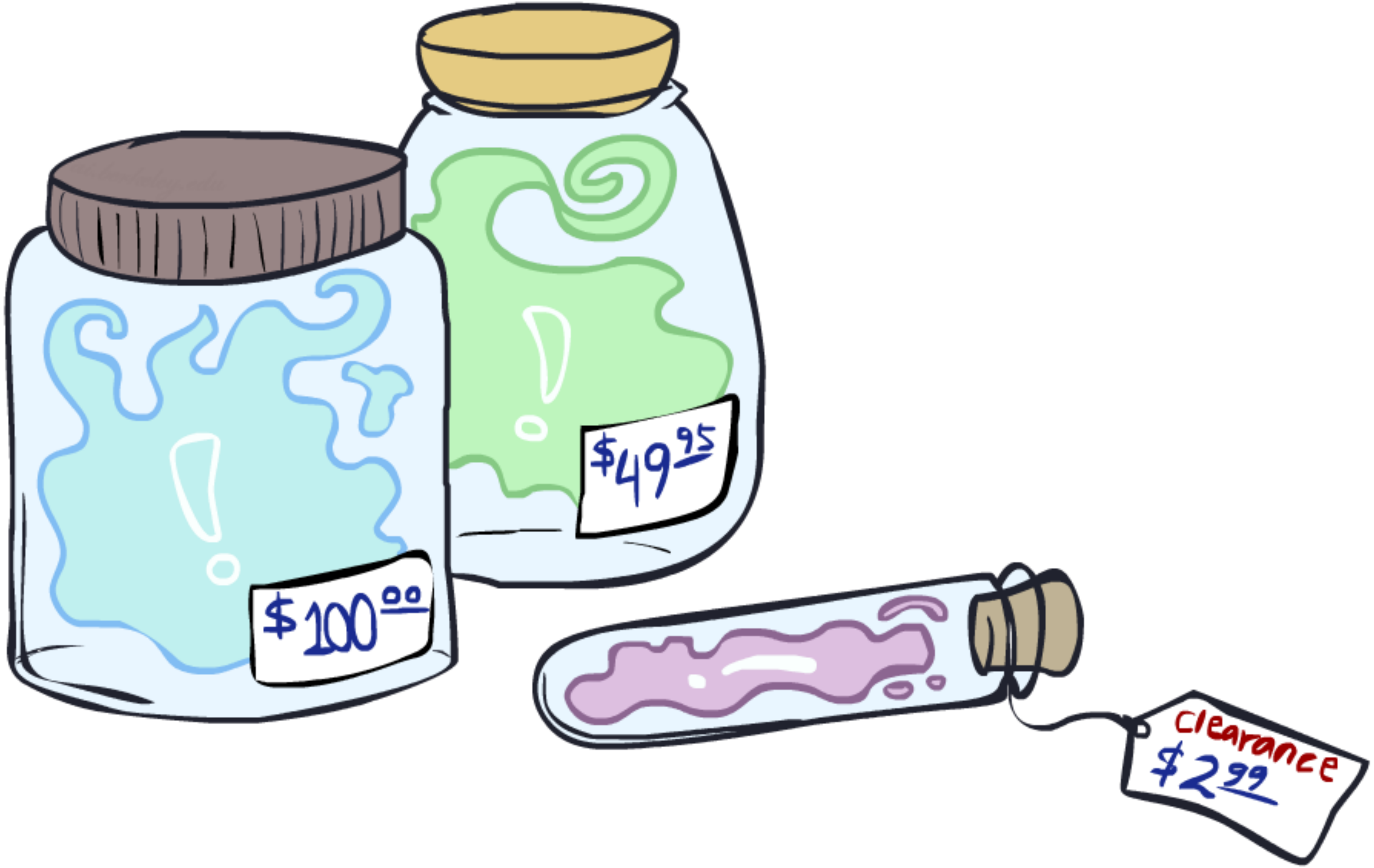
Video of Demo Ghostbusters with Probability

Ghostbusters Decision Network

Demo: Ghostbusters with probability



Value of Information



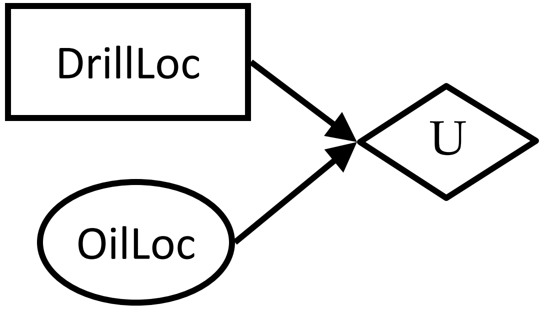
Value of Information

- Idea: compute value of acquiring evidence
  - Can be done directly from decision network

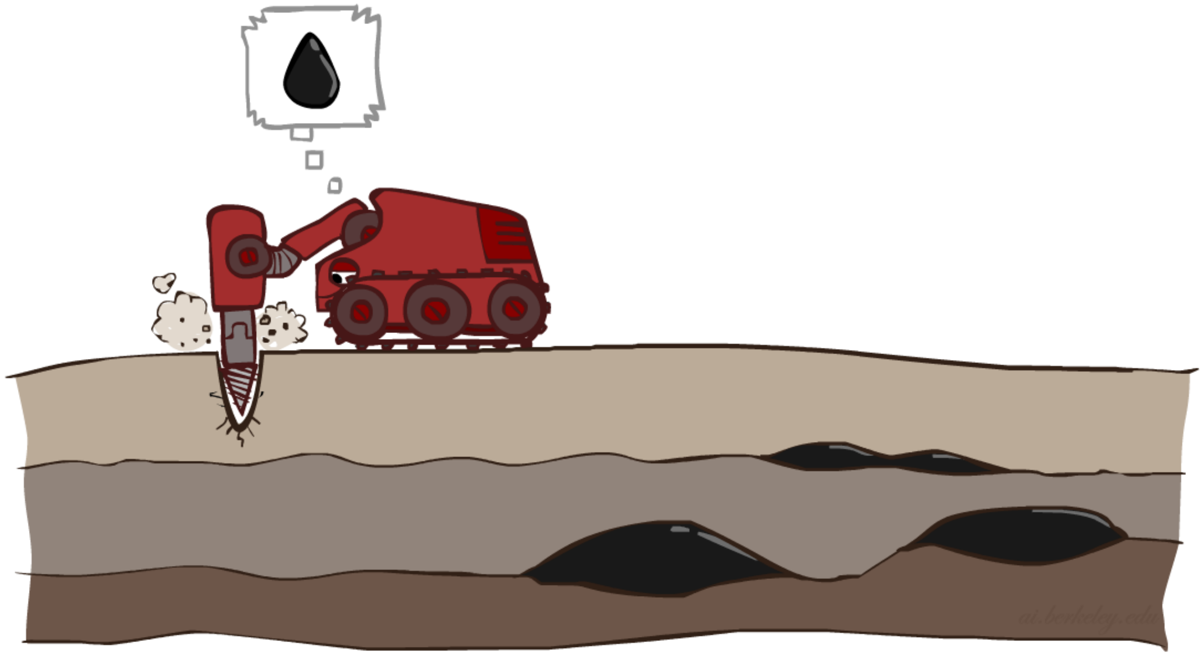
- Example: buying oil drilling rights
  - Two blocks A and B, exactly one has oil, worth k
  - You can drill in one location
  - Prior probabilities 0.5 each, & mutually exclusive
  - Drilling in either A or B has EU = k/2, MEU = k/2

- Question: what's the value of information of O?
  - Value of knowing which of A or B has oil
  - Value is expected gain in MEU from new info
  - Survey may say "oil in a" or "oil in b," prob 0.5 each
  - If we know OilLoc, MEU is k (either way)
  - Gain in MEU from knowing OilLoc?
  - VPI(OilLoc) = k/2
  - Fair price of information: k/2

O	P
a	1/2
b	1/2



D	O	U
a	a	k
a	b	0
b	a	0
b	b	k



Value of Perfect Information

MEU with no evidence

$$\text{MEU}(\phi) = \max_a \text{EU}(a) = 70$$

MEU if forecast is bad

$$\text{MEU}(F = \text{bad}) = \max_a \text{EU}(a|\text{bad}) = 53$$

MEU if forecast is good

$$\text{MEU}(F = \text{good}) = \max_a \text{EU}(a|\text{good}) = 95$$

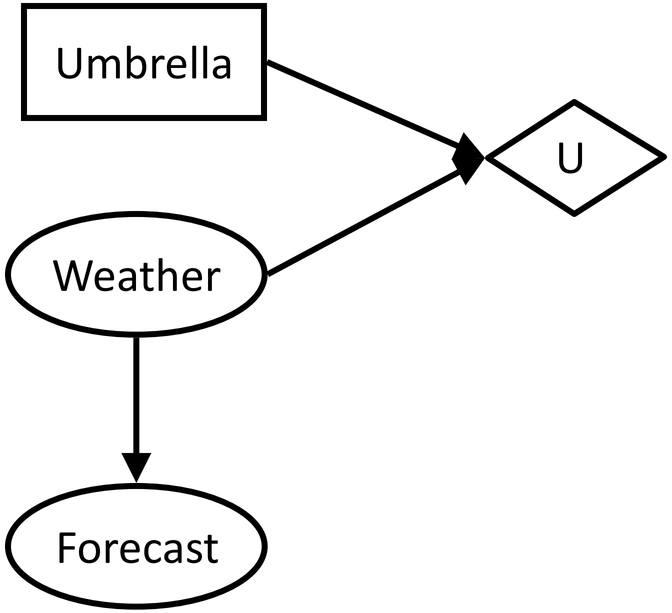
Forecast distribution

F	P(F)
good	0.59
bad	0.41

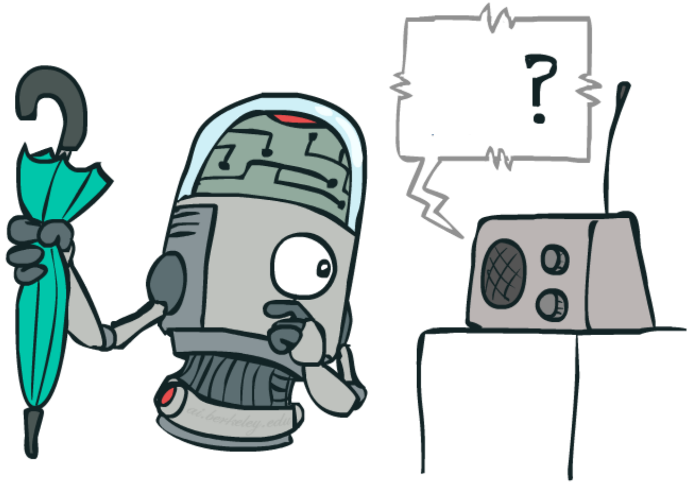
$$0.59 \cdot (95) + 0.41 \cdot (53) - 70$$

$$77.8 - 70 = 7.8$$

$$\text{VPI}(E'|e) = \left(\sum_{e'} P(e'|e)\text{MEU}(e, e')\right) - \text{MEU}(e)$$



A	W	U
leave	sun	100
leave	rain	0
take	sun	20
take	rain	70



Value of Information

- Assume we have evidence E=e. Value if we act now:

$$\text{MEU}(e) = \max_a \sum_s P(s|e)\, U(s,a)$$

- Assume we see that E' = e'. Value if we act then:

$$\text{MEU}(e,e') = \max_a \sum_s P(s|e,e')\, U(s,a)$$

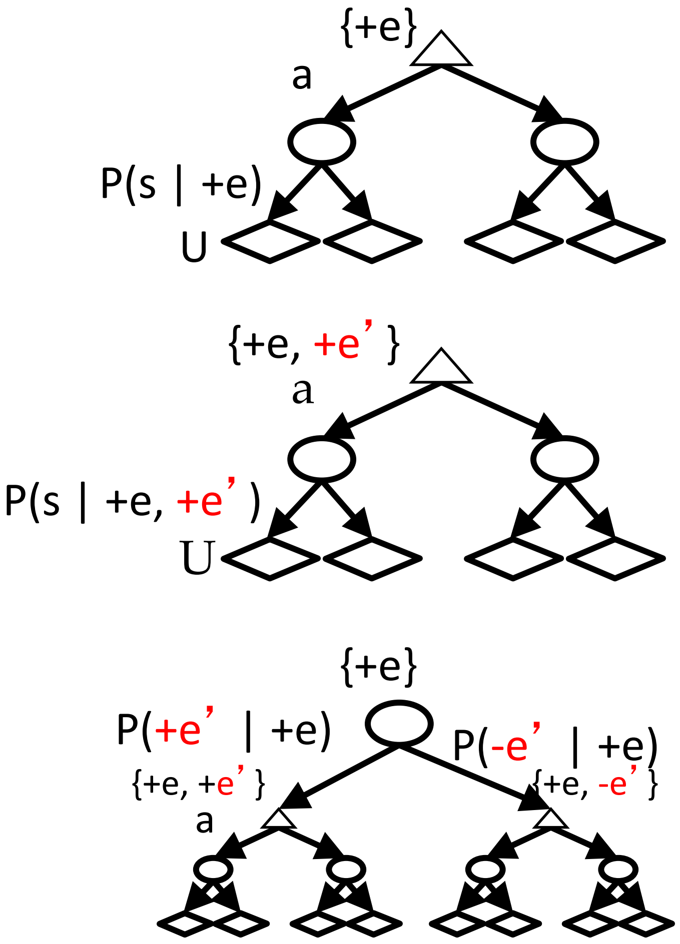
- BUT E' is a random variable whose value is unknown, so we don't know what e' will be

- Expected value if E' is revealed and then we act:

$$\text{MEU}(e,E') = \sum_{e'} P(e'|e)\text{MEU}(e,e')$$

- Value of information: how much MEU goes up by revealing E' first then acting, over acting now:

$$\text{VPI}(E'|e) = \text{MEU}(e,E') - \text{MEU}(e)$$



Value of Information 2

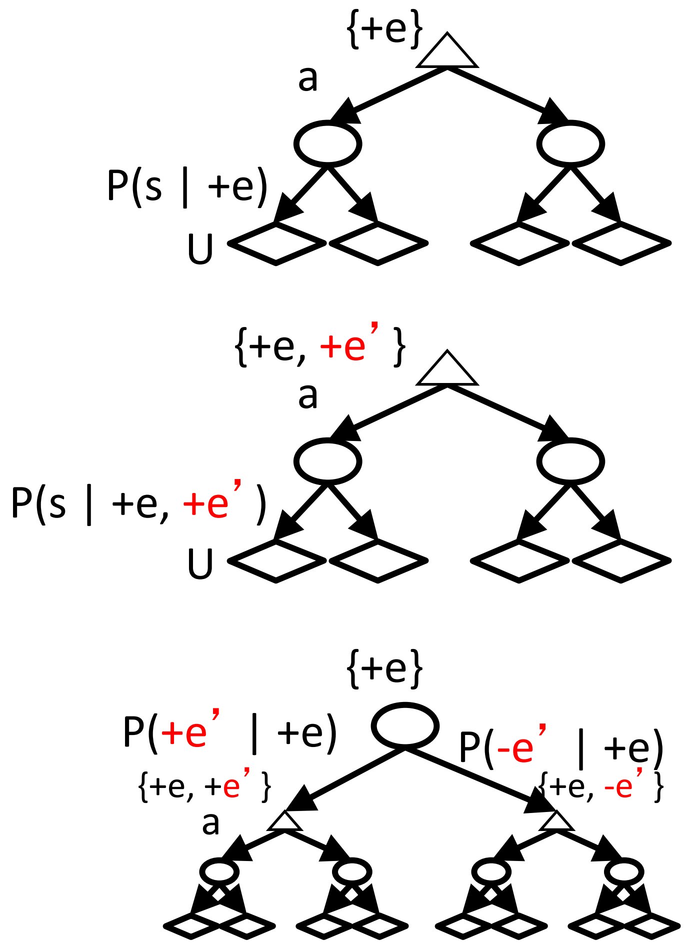
$$\text{MEU}(e, E') = \sum_{e'} P(e'|e)\text{MEU}(e, e')$$

$$= \sum_{e'} P(e'|e) \max_a \sum_s P(s|e, e')U(s, a)$$

$$\text{MEU}(e) = \max_a \sum_s P(s|e)\, U(s, a)$$

$$= \max_a \sum_{e'} \sum_s P(s, e'|e)U(s, a)$$

$$= \max_a \sum_{e'} P(e|e') \sum_s P(s|e, e')U(s, a)$$



VPI Properties

- Nonnegative

$$\forall E', e : \text{VPI}(E'|e) \geq 0$$

- Nonadditive

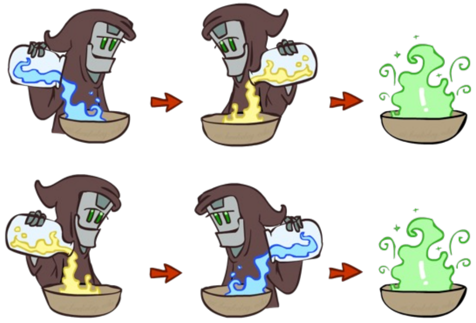
  (think of observing E_i twice)

$$\text{VPI}(E_j, E_k|e) \neq \text{VPI}(E_j|e) + \text{VPI}(E_k|e)$$

- Order-independent

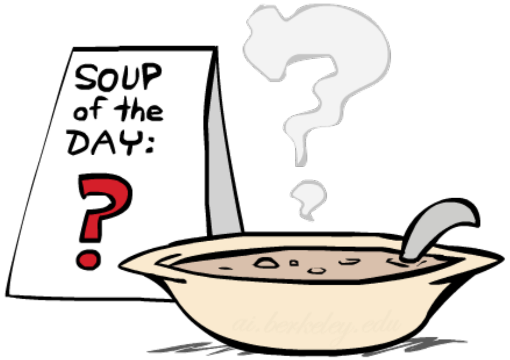
$$\text{VPI}(E_j, E_k|e) = \text{VPI}(E_j|e) + \text{VPI}(E_k|e, E_j)$$

$$= \text{VPI}(E_k|e) + \text{VPI}(E_j|e, E_k)$$

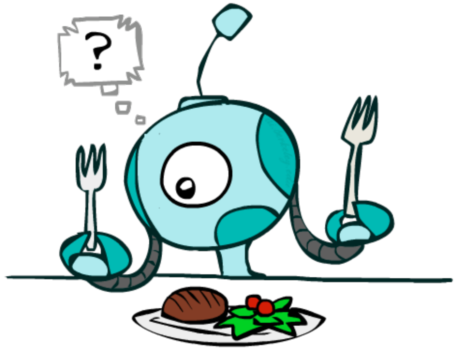


Quick VPI Questions

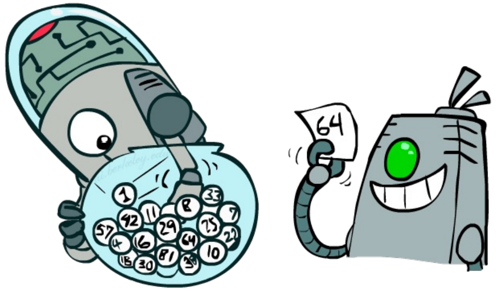
- The soup of the day is either clam chowder or split pea, but you wouldn't order either one. What's the value of knowing which it is?



- There are two kinds of plastic forks at a picnic. One kind is slightly sturdier. What's the value of knowing which?



- You're playing the lottery. The prize will be $0 or $100. You can play any number between 1 and 100 (chance of winning is 1%). What is the value of knowing the winning number?

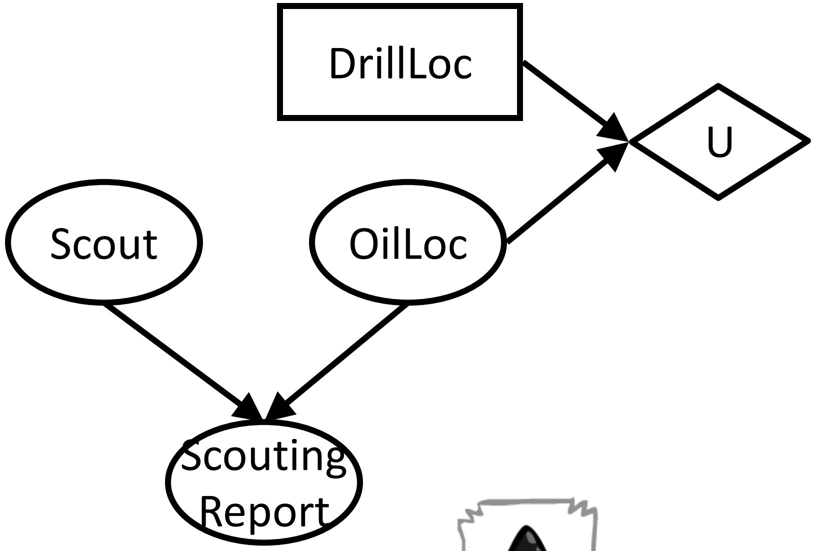


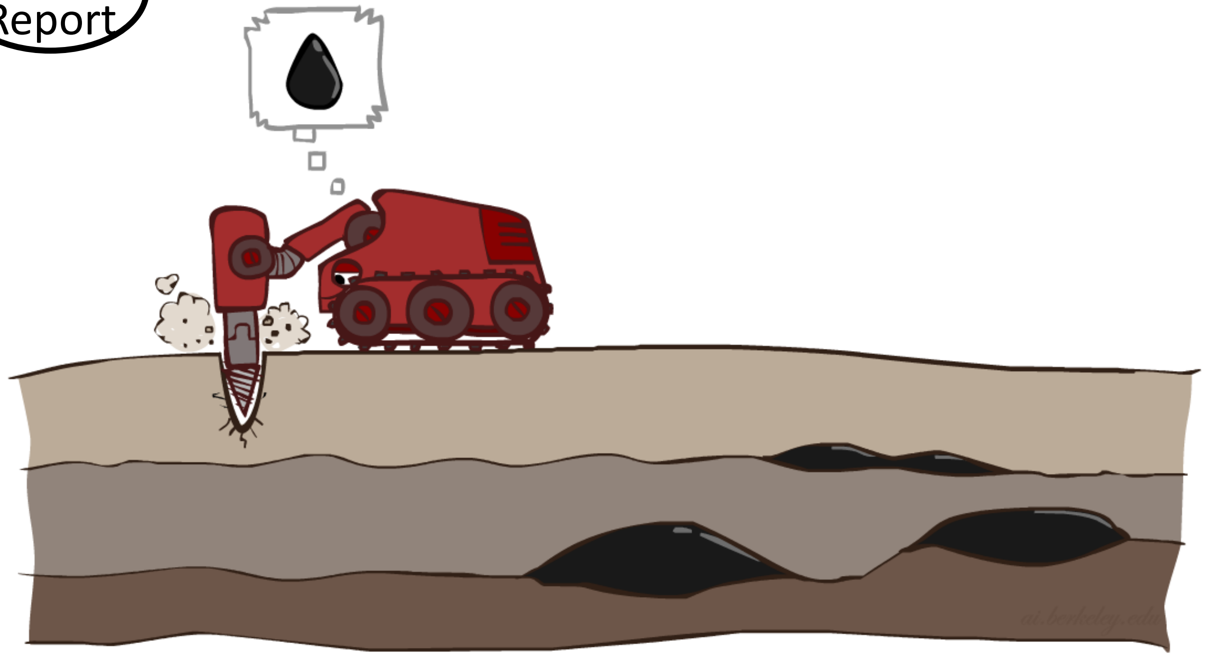
Value of Imperfect Information?

- No such thing
- Information corresponds to the observation of a node in the decision network
- If data is “noisy” that just means we don’t observe the original variable, but another variable which is a noisy version of the original one

VPI Question

- VPI(OilLoc) ?
- VPI(ScoutingReport) ?
- VPI(Scout) ?
- VPI(Scout | ScoutingReport) ?





- Generally:
  If Parents(U) $\perp\!\!\!\perp$ Z | CurrentEvidence
  Then VPI(Z | CurrentEvidence) = 0