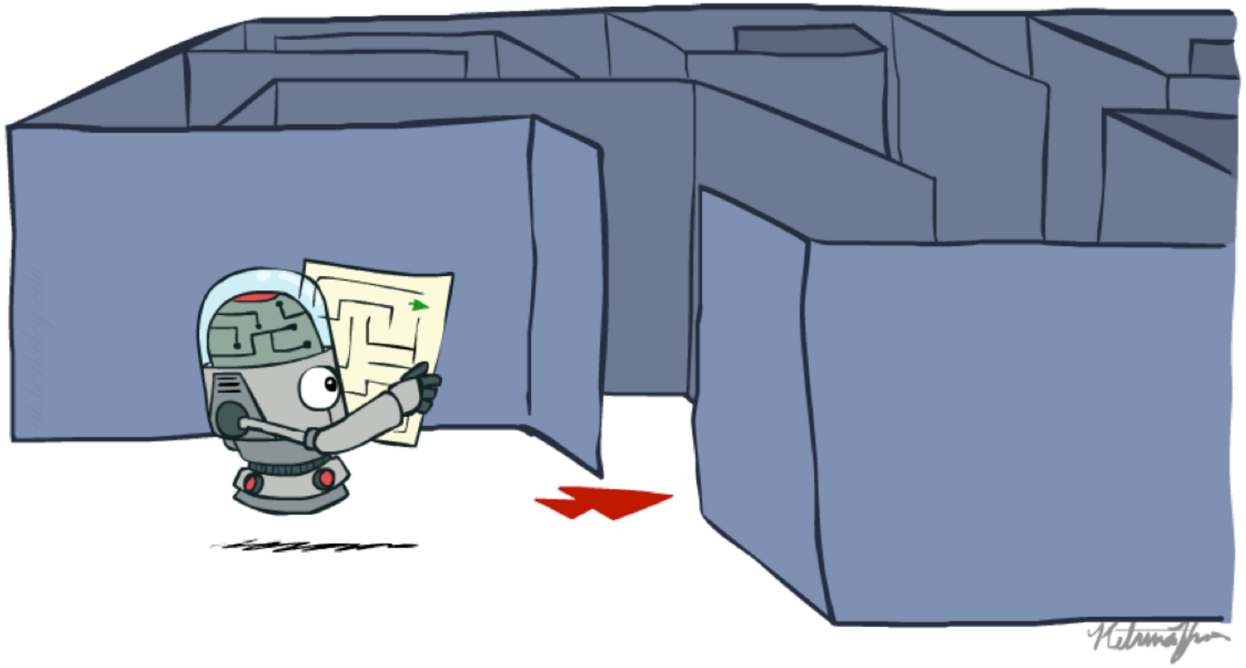
CS 188: Artificial Intelligence

Search



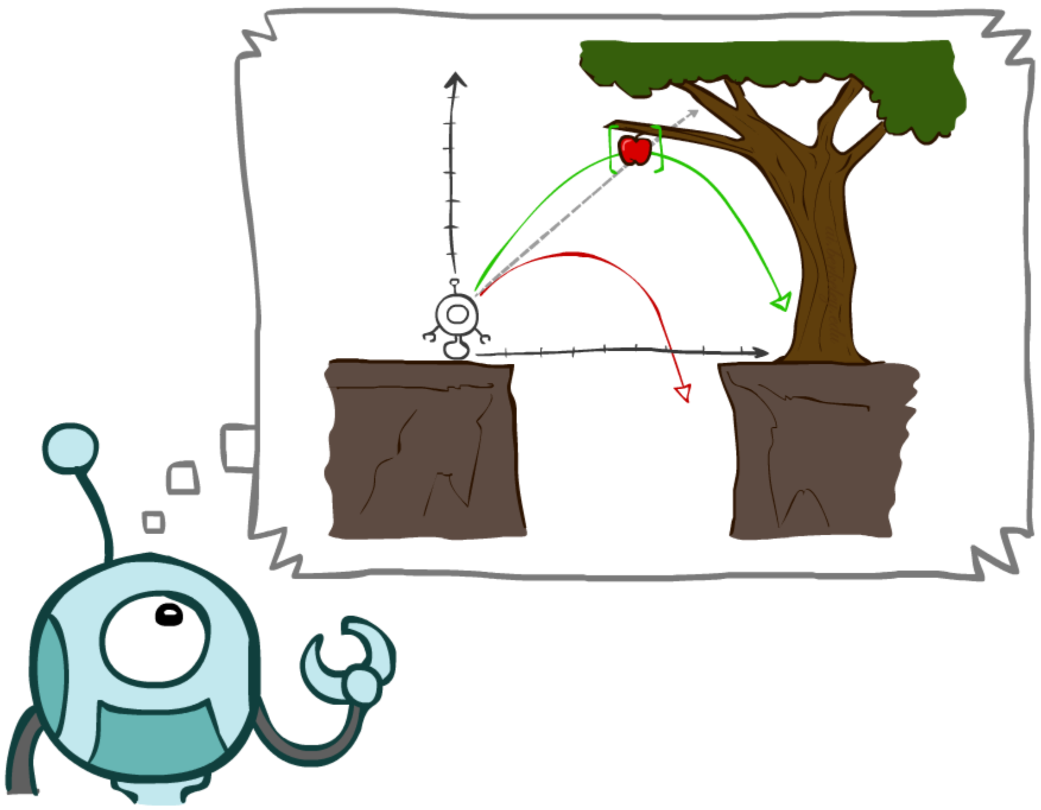
Instructors: Angela Liu and Yanlai Yang

University of California, Berkeley

[slides adapted from Dan Klein, Pieter Abbeel, Stuart Russel, Dawn Song]

Today

- Finish discussion of agents and environments
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search



Environment types

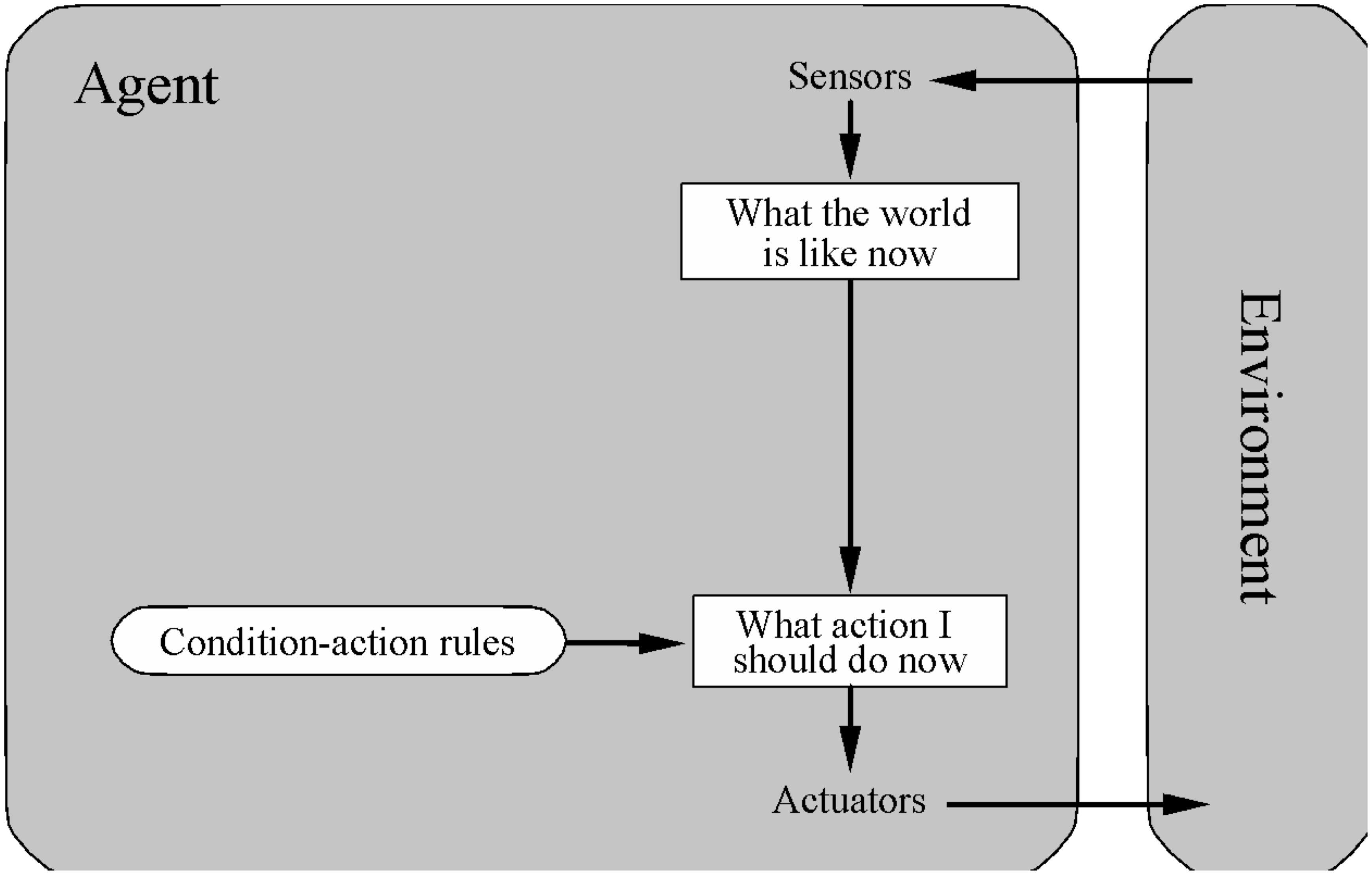
	Pacman	Backgammon	Diagnosis	Taxi
Fully or partially observable				
Single-agent or multiagent				
Deterministic or stochastic				
Static or dynamic				
Discrete or continuous				
Known physics?				
Known perf. measure?				

Agent design

The environment type largely determines the agent design

- ***Partially observable*** => agent requires ***memory*** (internal state)
- ***Stochastic*** => agent may have to prepare for ***contingencies***
- ***Multi-agent*** => agent may need to behave ***randomly***
- ***Static*** => agent has time to compute a rational decision
- ***Continuous time*** => continuously operating ***controller***
- ***Unknown physics*** => need for ***exploration***
- ***Unknown perf. measure*** => observe/interact with ***human principal***

Simple reflex agents

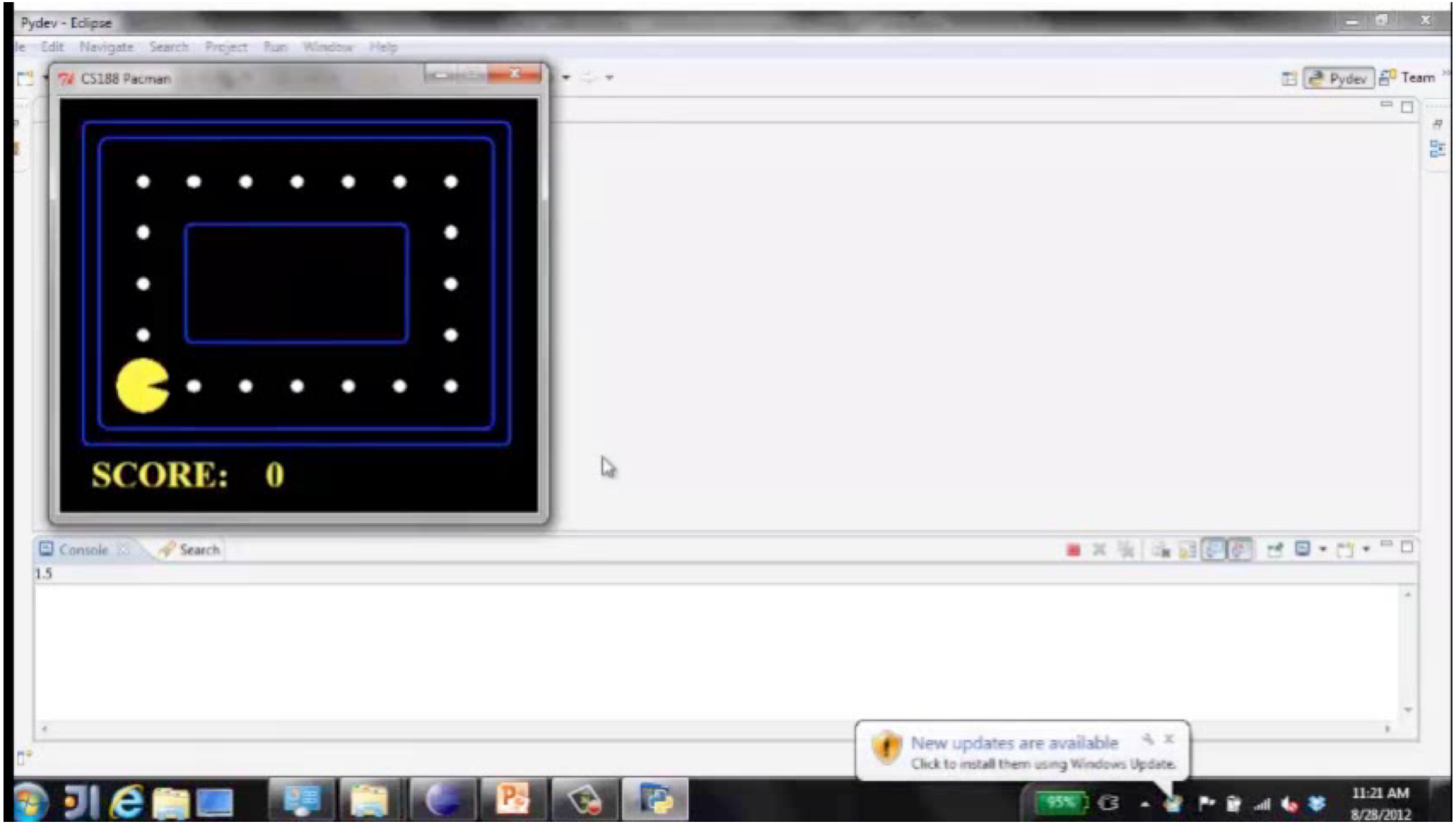


Pacman ***agent program*** in Python

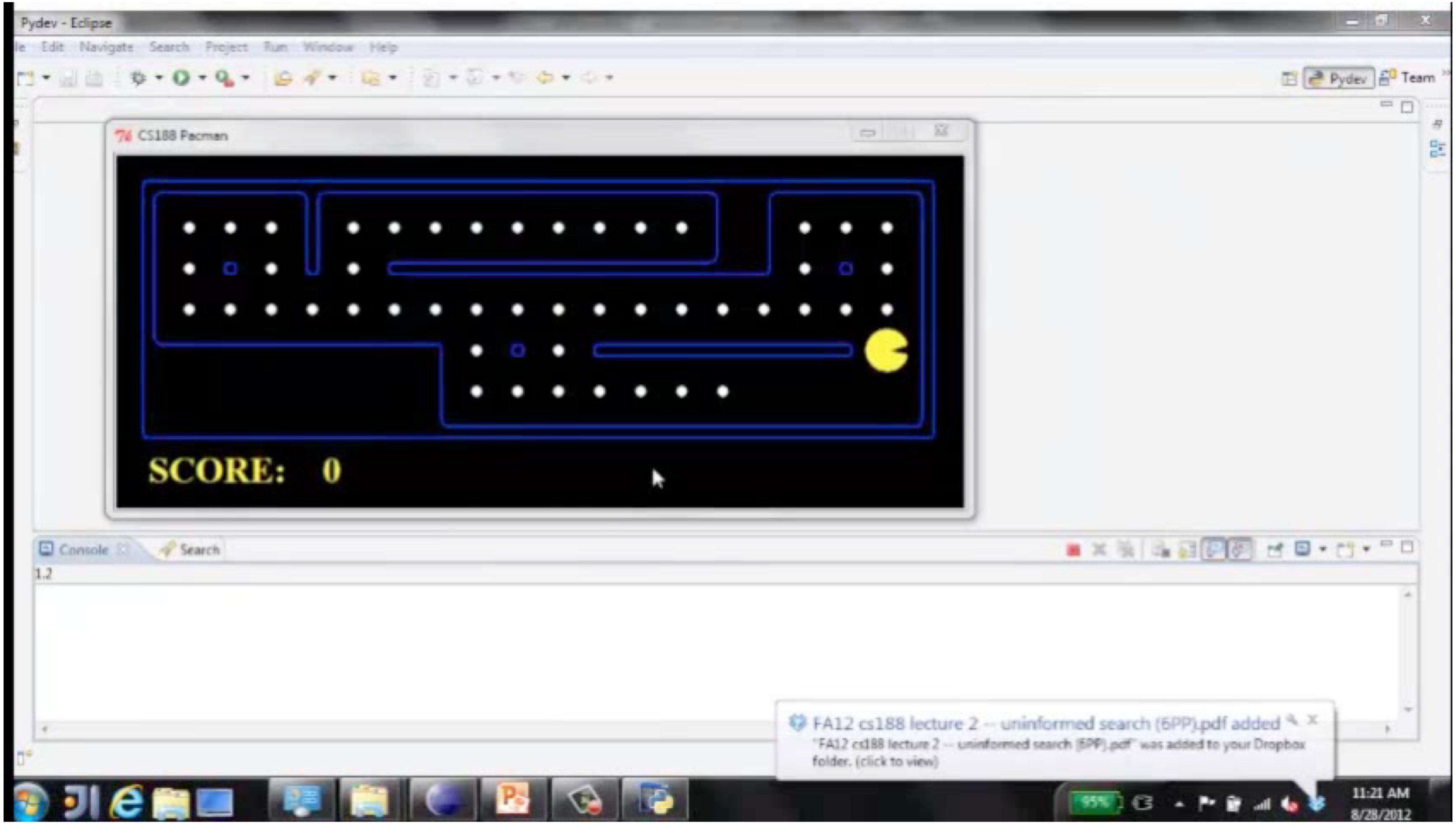
```
class GoWestAgent(Agent):

  def getAction(self, percept):
    if Directions.WEST in percept.getLegalPacmanActions():
      return Directions.WEST
    else:
      return Directions.STOP
```

Eat adjacent dot, if any



Eat adjacent dot, if any

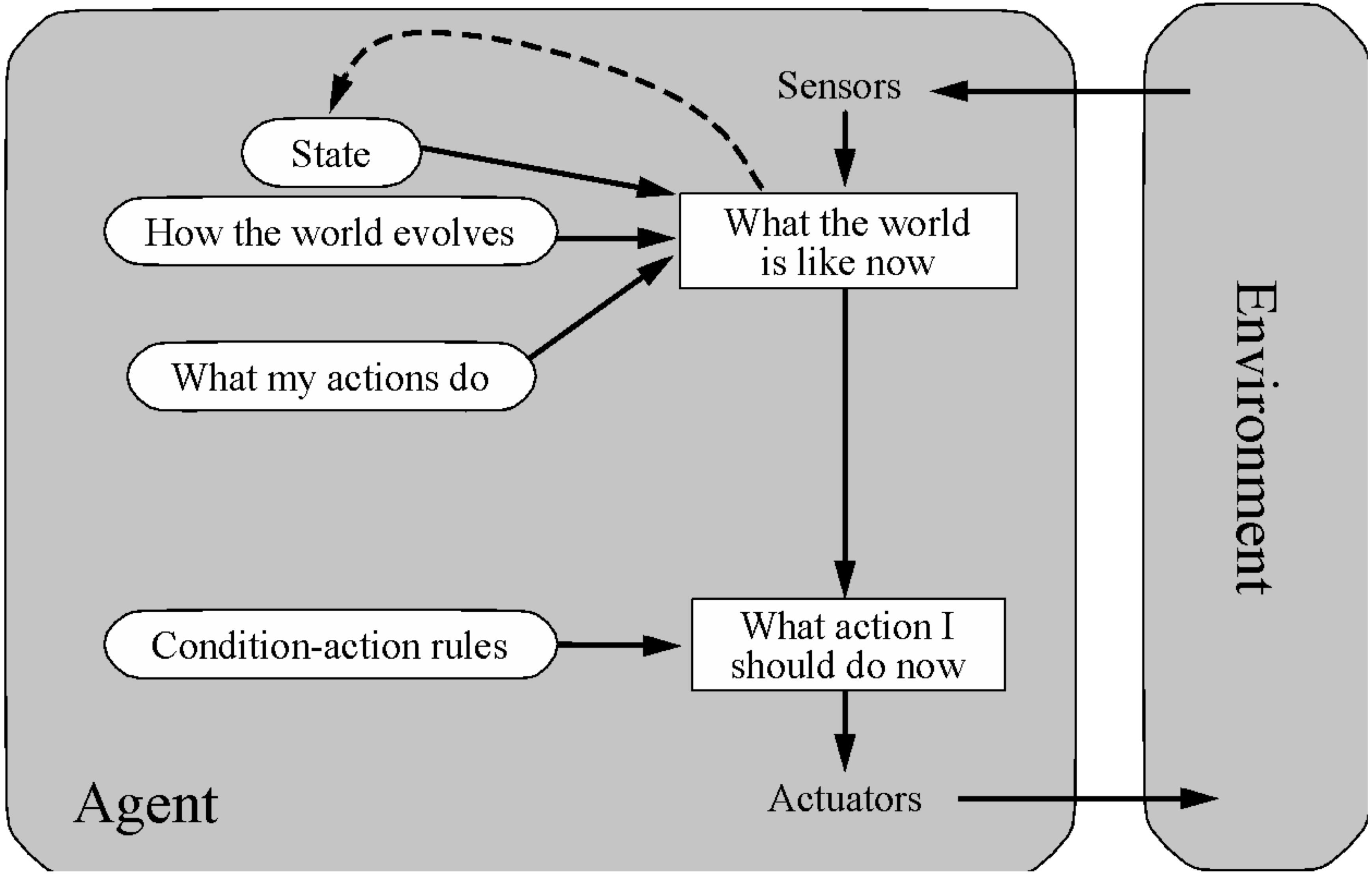


Pacman agent contd.

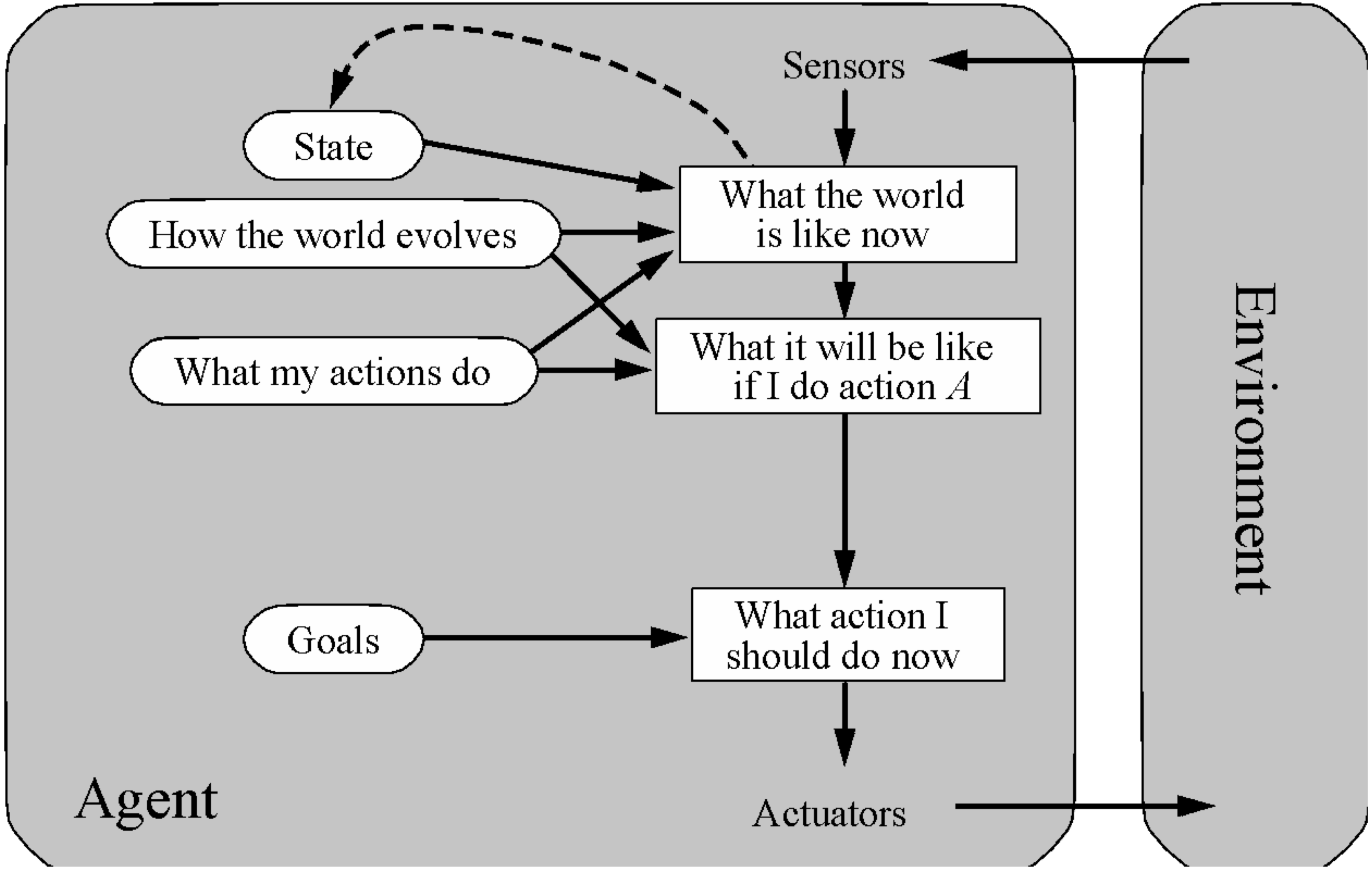
Can we (in principle) extend this reflex agent to behave well in all standard Pacman environments?

- No – Pacman is not quite fully observable (power pellet duration)
- Otherwise, yes – we can (*in principle*) make a lookup table.....
- *How large would it be?*

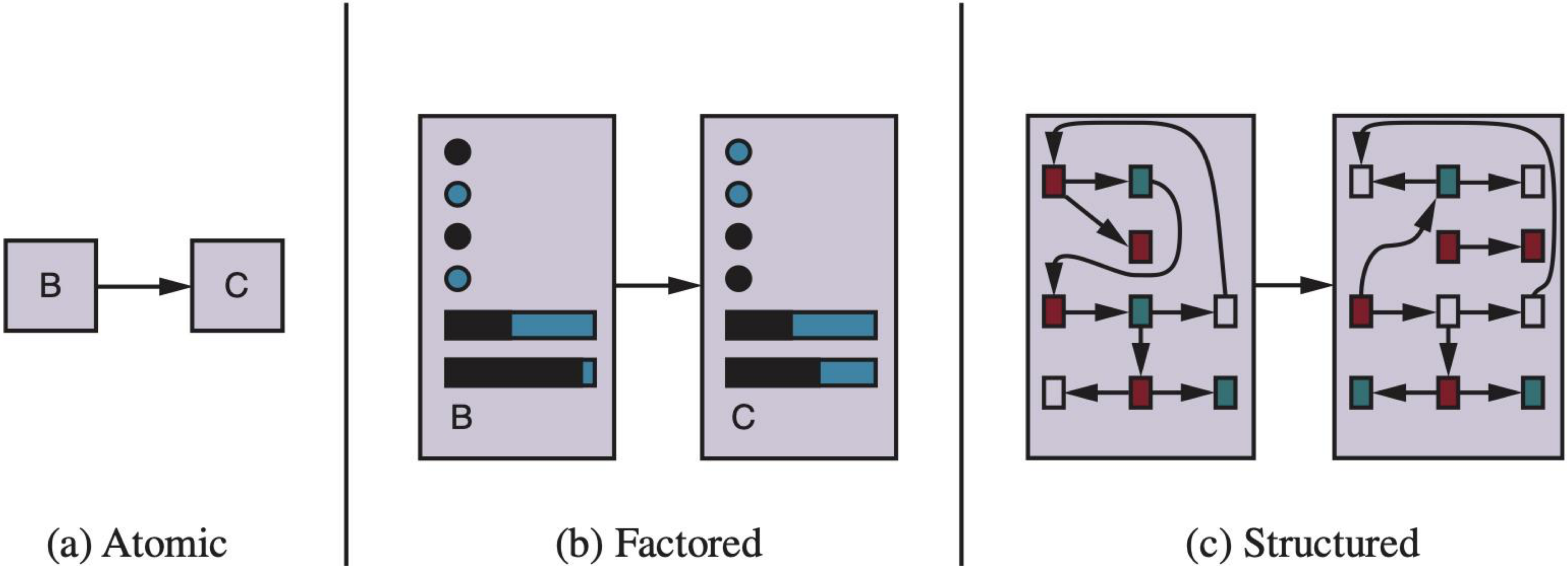
Model-based agents



Goal-based agents

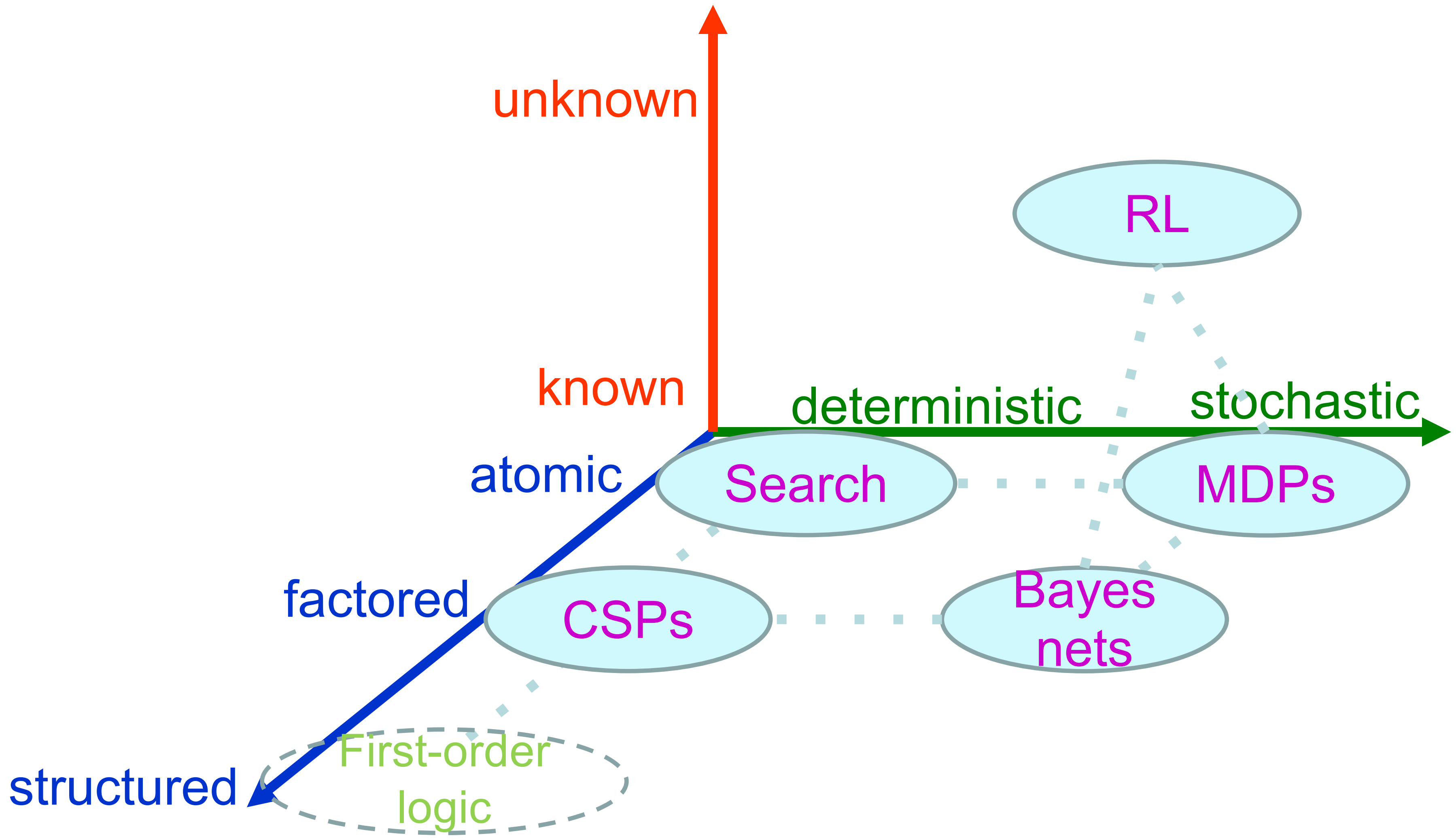


Spectrum of representations



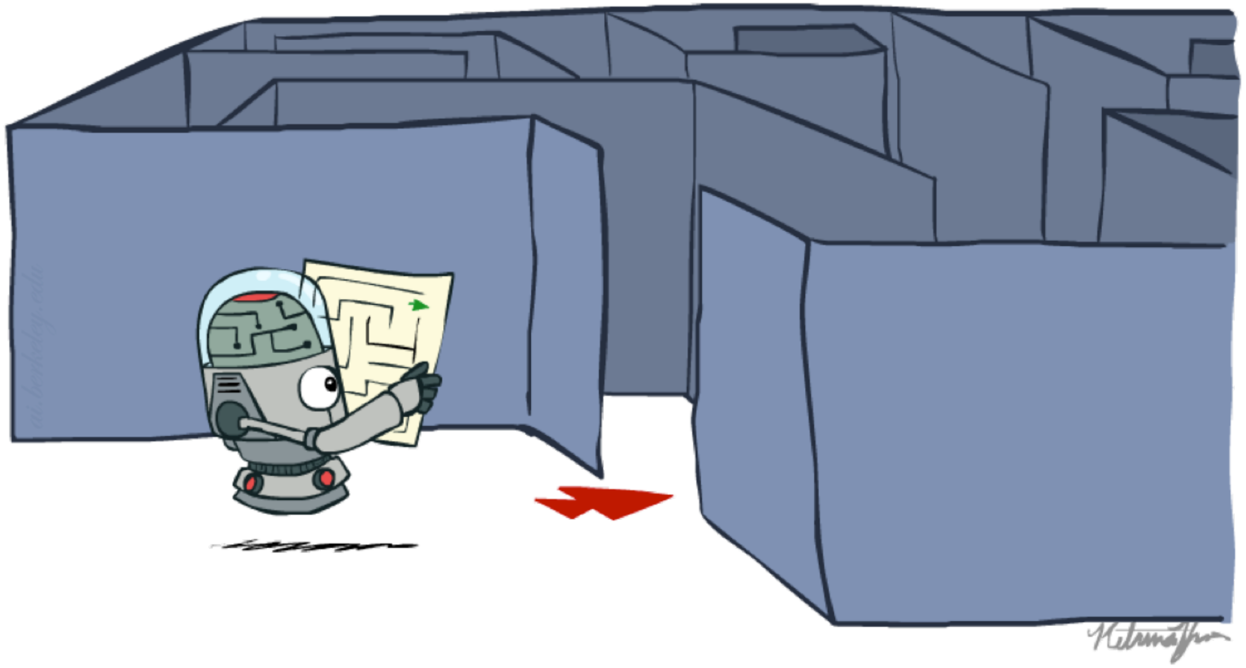
(a) Atomic (b) Factored (c) Structured

Outline of the course



CS 188: Artificial Intelligence

Search



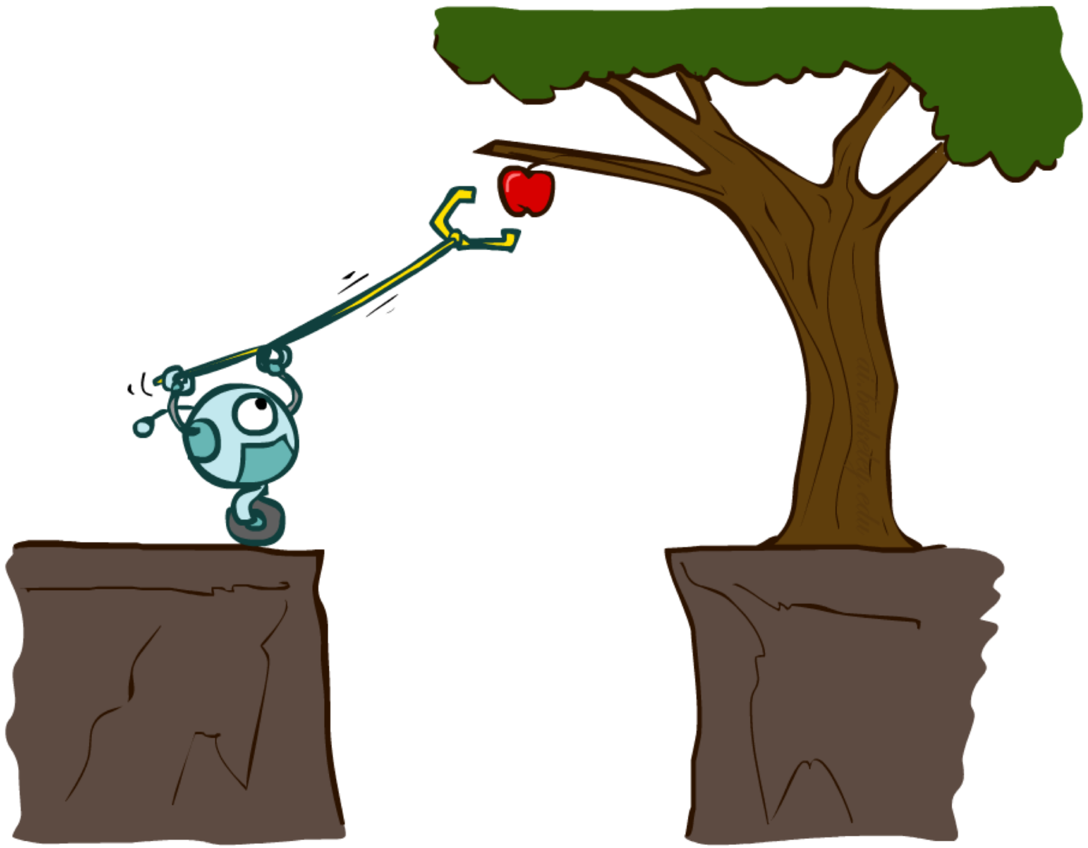
Instructors: Angela Liu and Yanlai Yang

University of California, Berkeley

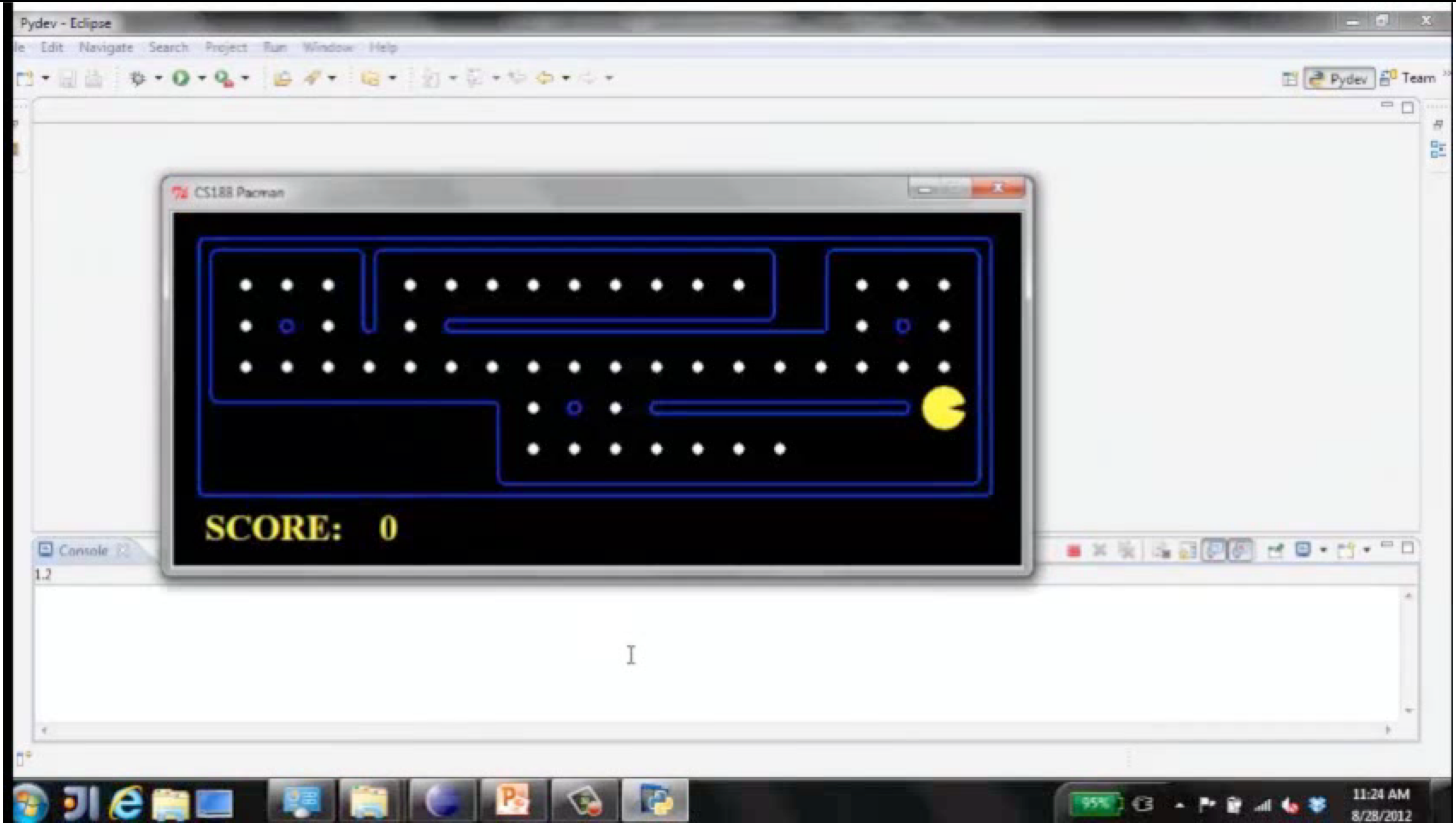
[slides adapted from Dan Klein, Pieter Abbeel, Stuart Russel, Dawn Song]

Planning Agents

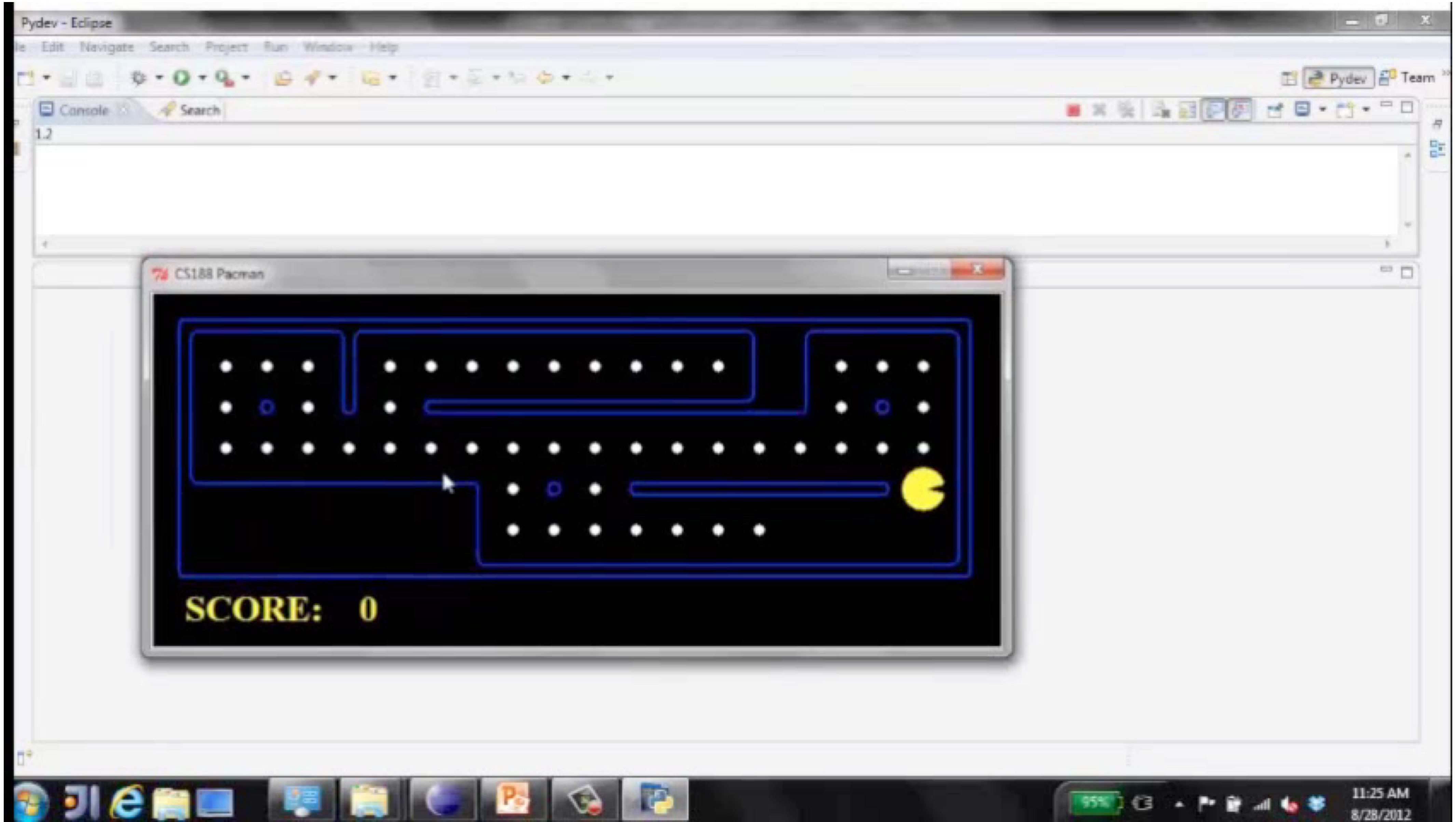
- Planning agents decide based on evaluating future action sequences
- Search algorithms typically assume
  - Known, deterministic transition model
  - Discrete states and actions
  - Fully observable
  - Atomic representation
- Usually have a definite goal
- Optimal: Achieve goal at least cost



Move to nearest dot and eat it



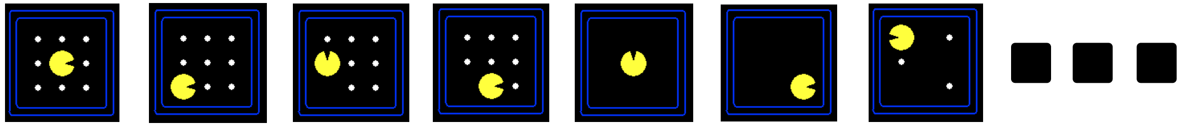
Precompute optimal plan, execute it



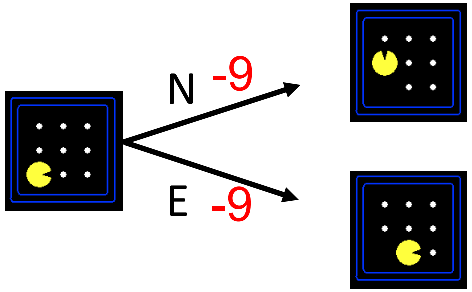
Search Problems

Search Problems

- A search problem consists of:

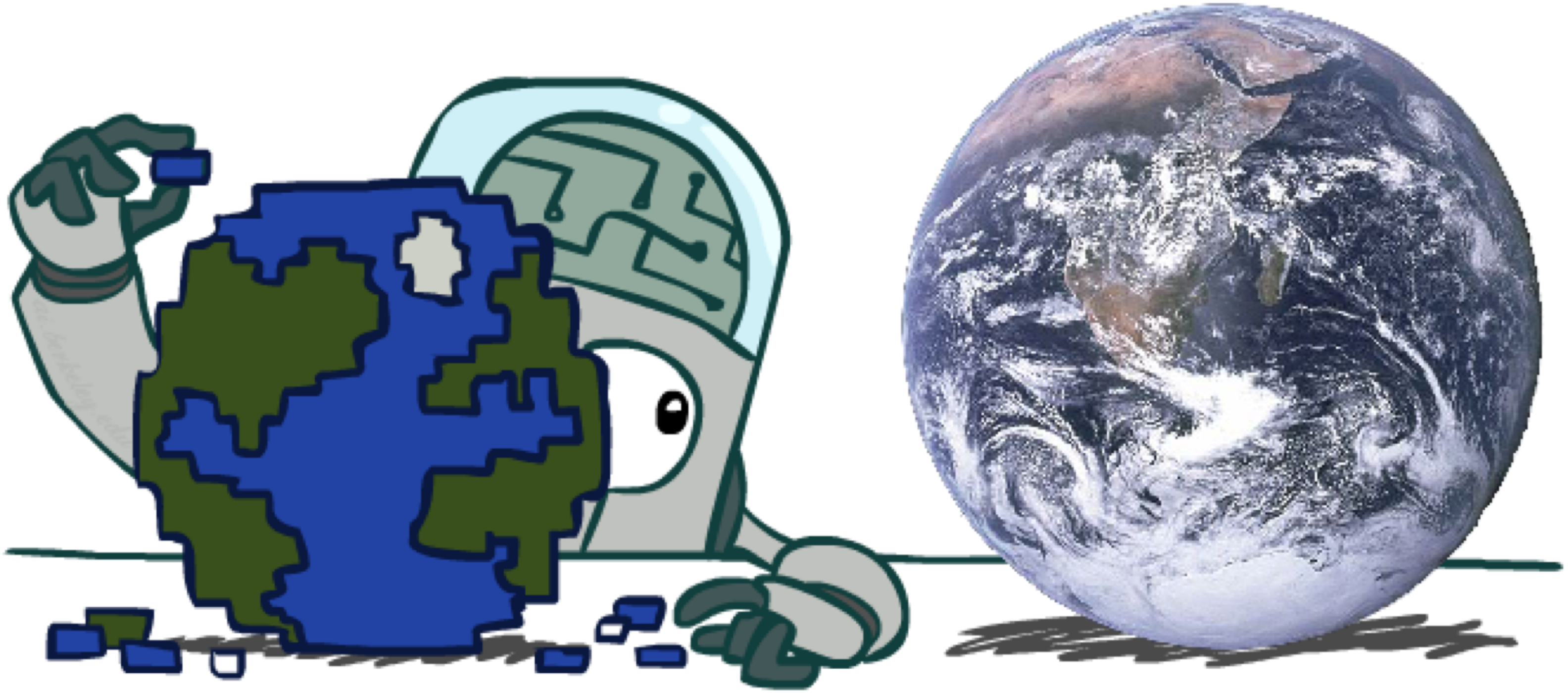


  - A state space $\mathcal{S}$
  - An initial state s_0
  - Actions $\mathcal{A}(s)$ in each state
  - Transition model $Result(s,a)$
  - A goal test $G(s)$
    - s has no dots left
  - Action cost $c(s,a,s')$
    - +1 per step; -10 food; -500 win; +500 die; -200 eat ghost

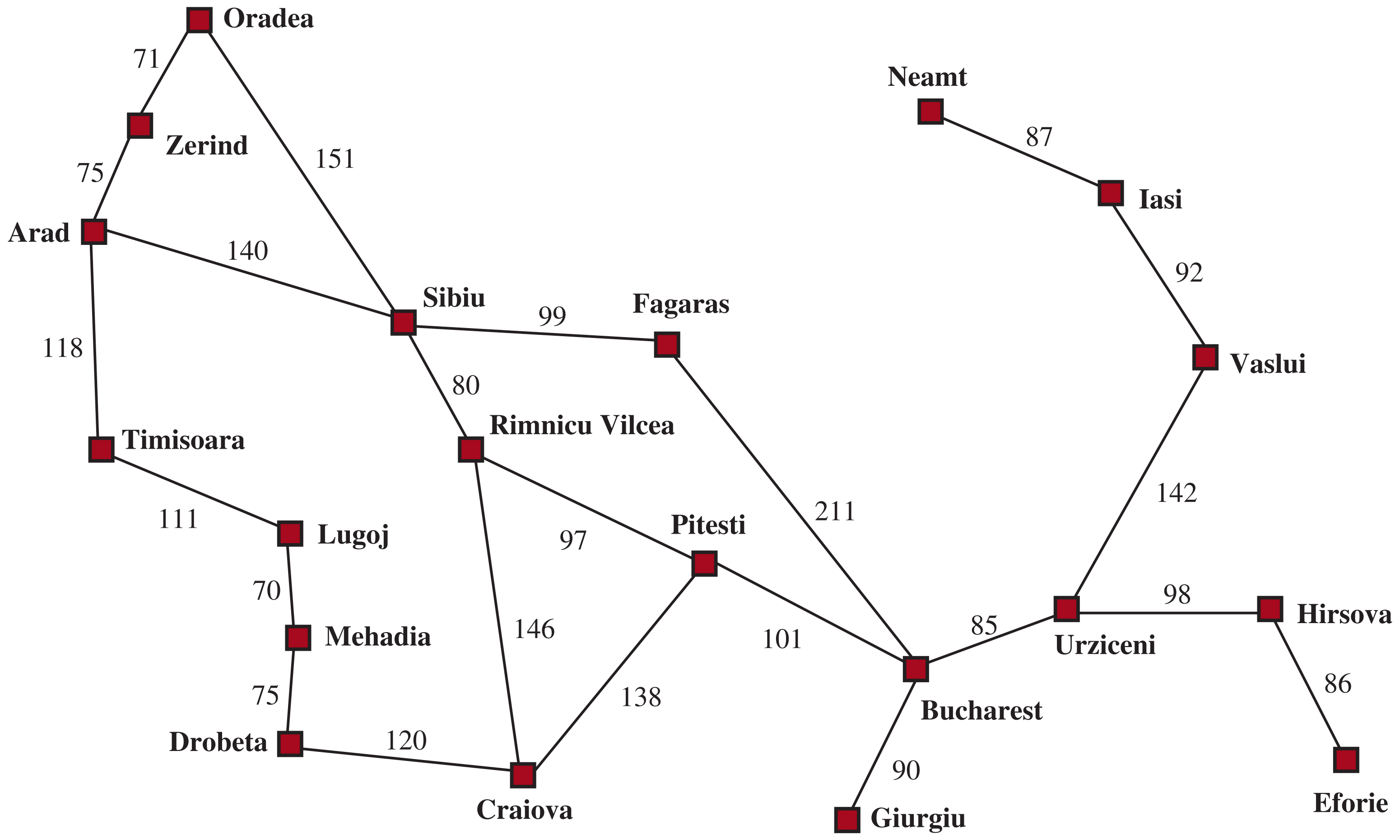


- A solution is an action sequence that reaches a goal state
- An optimal solution has least cost among all solutions

Search Problems Are Models



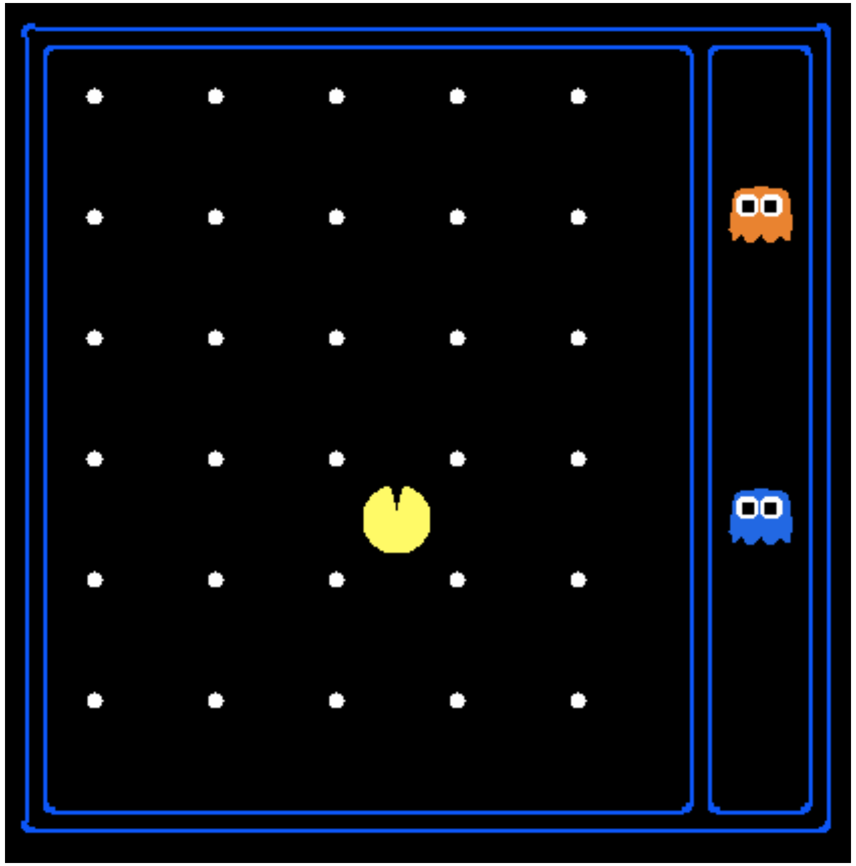
Example: Traveling in Romania



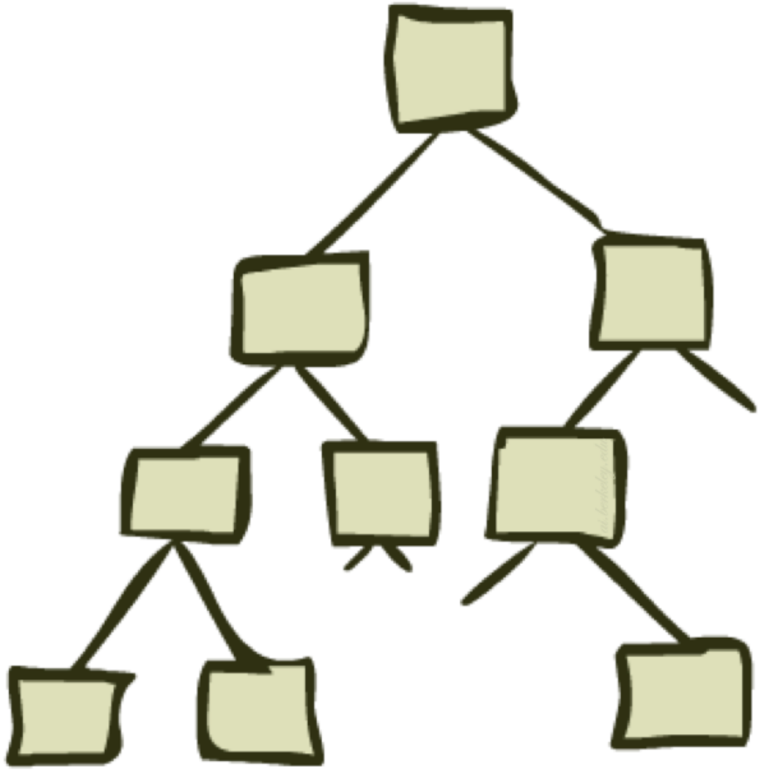
- State space:
  - Cities
- Initial state:
  - Arad
- Actions:
  - Go to adjacent city
- Transition model:
  - Reach adjacent city
- Goal test:
  - *s* = Bucharest?
- Action cost:
  - Road distance from *s* to *s'*
- Solution?

State Space Sizes

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW
- How many
  - World states?
    $120x(2^{30})x(12^{2})x4$
  - States for pathing?
    120
  - States for eat-all-dots?
    $120x(2^{30})$

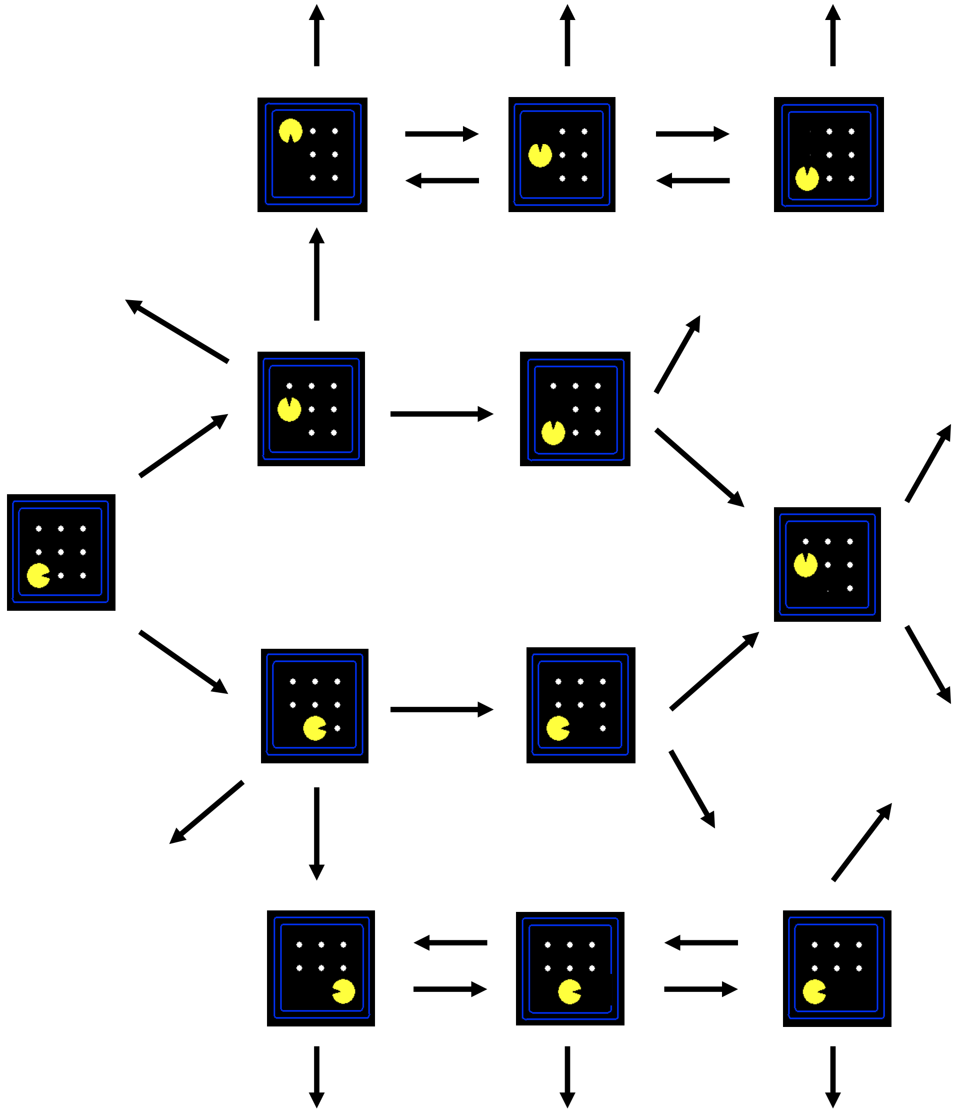


State Space Graphs and Search Trees



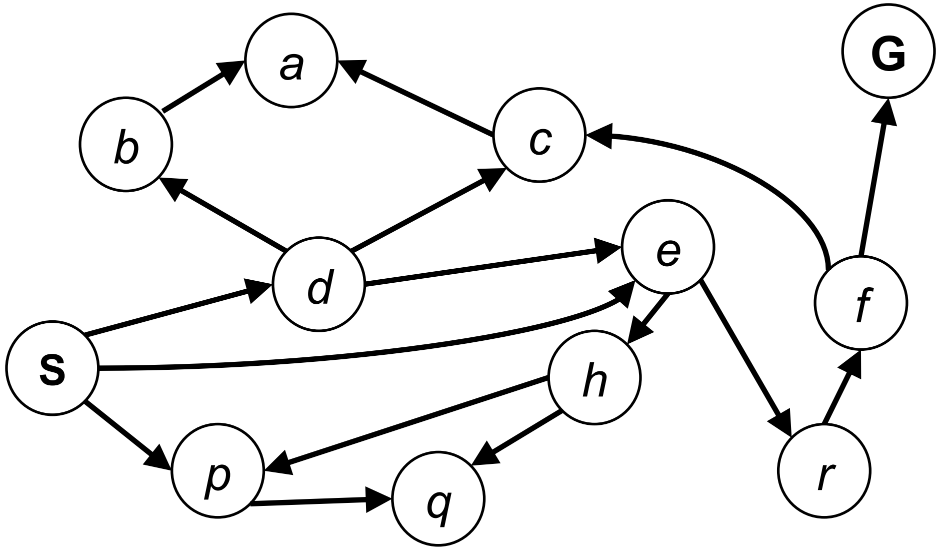
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent transitions (labeled with actions)
  - 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



State Space Graphs

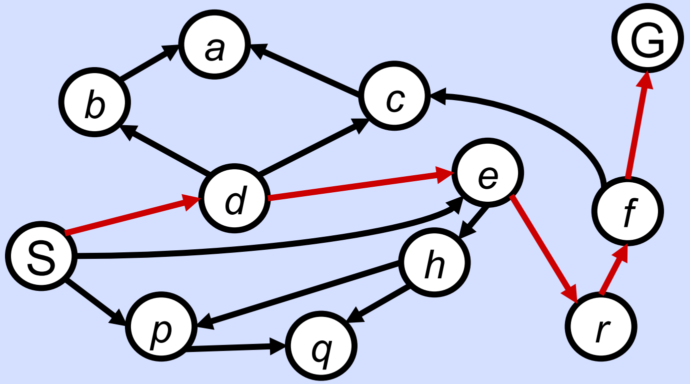
- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



Tiny state space graph for a tiny search problem

State Space Graphs vs. Search Trees

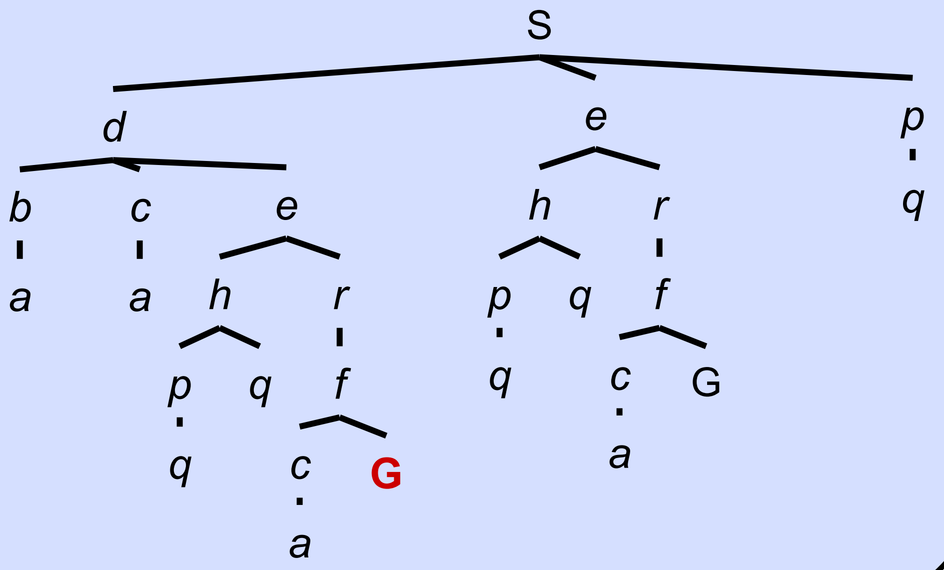
State Space Graph



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

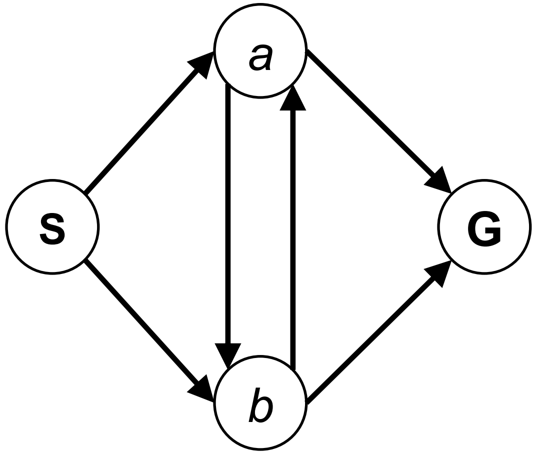
We construct the tree on demand – and we construct as little as possible.

Search Tree



Quiz: State Space Graphs vs. Search Trees

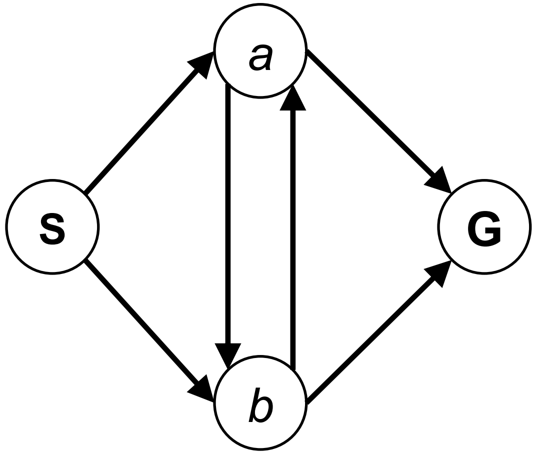
Consider this 4-state graph:



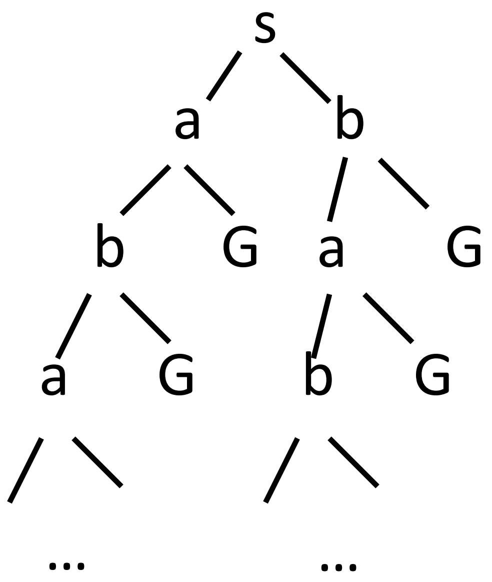
How big is its search tree (from S)?

Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:



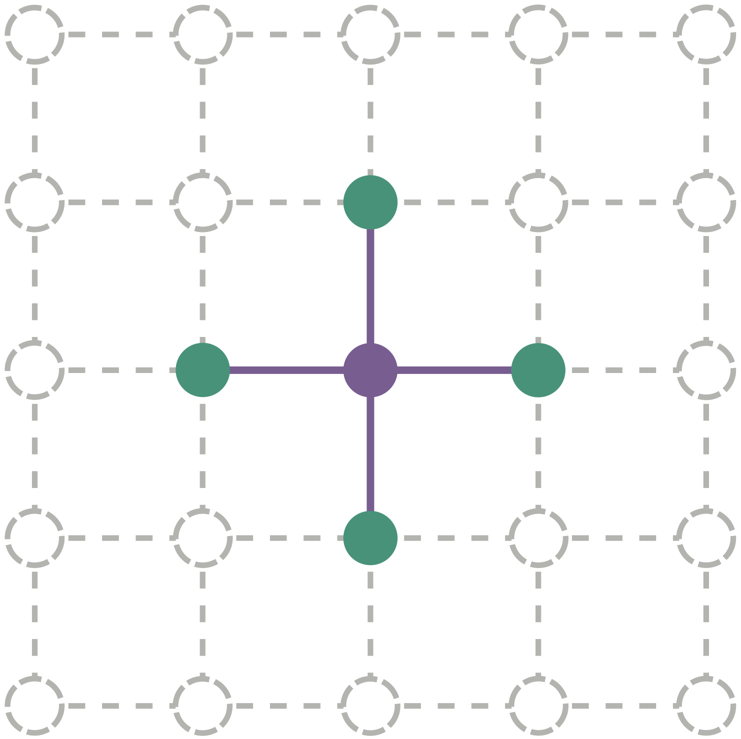
How big is its search tree (from S)?



Important: Those who don’t know history are doomed to repeat it!

Quiz: State Space Graphs vs. Search Trees

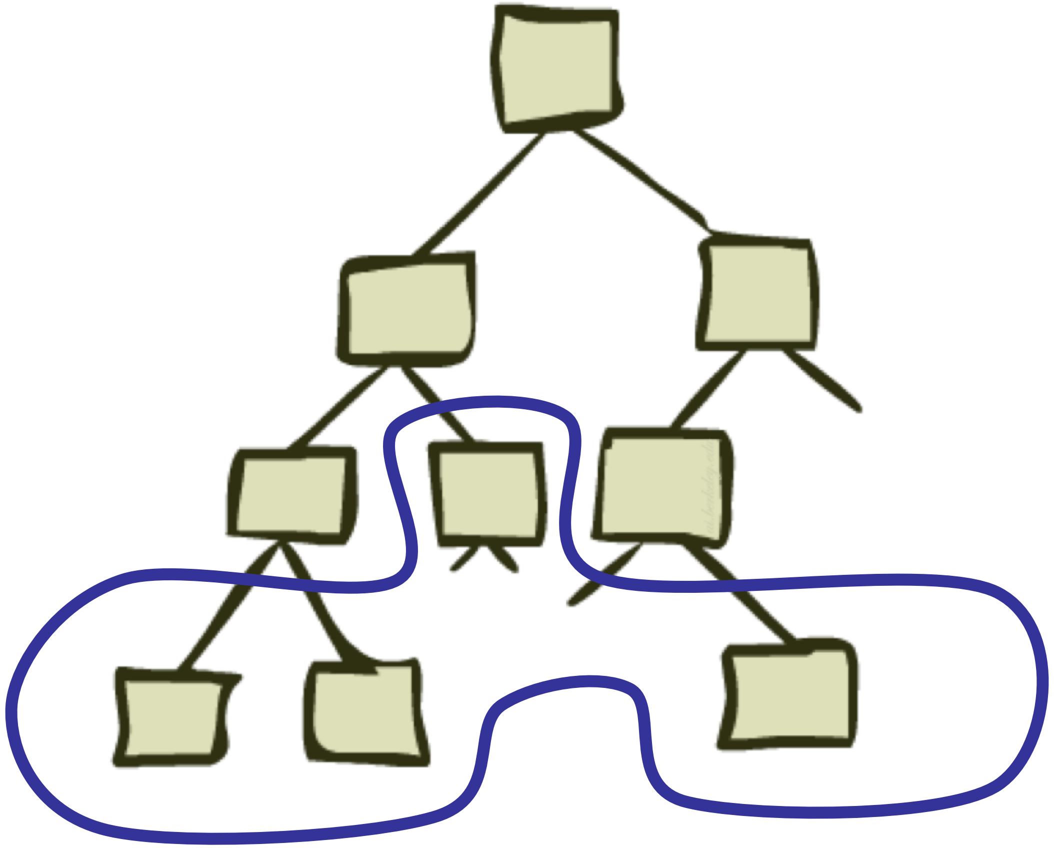
Consider a rectangular grid:



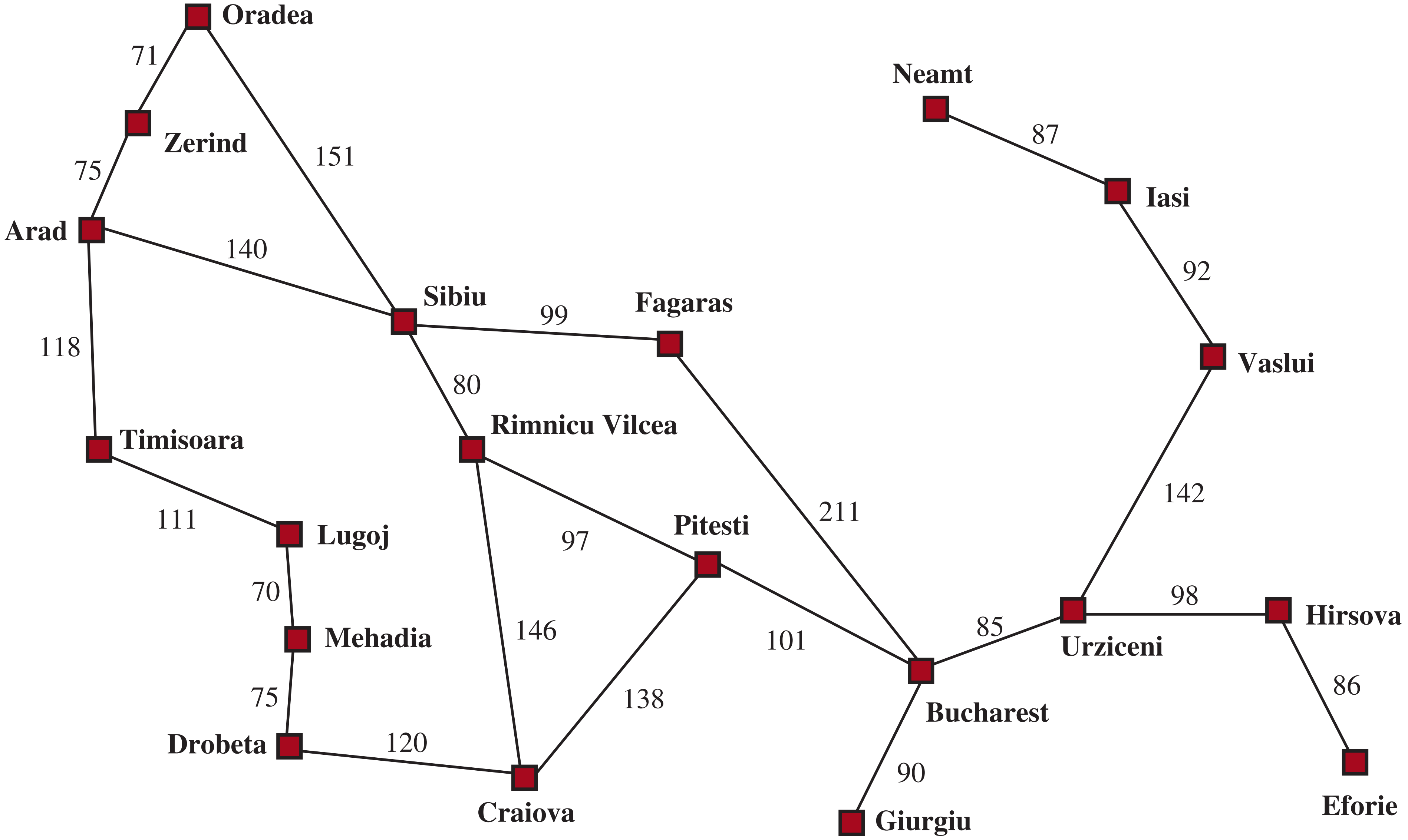
How many states within *d* steps of start?

How many states in search tree of depth *d*?

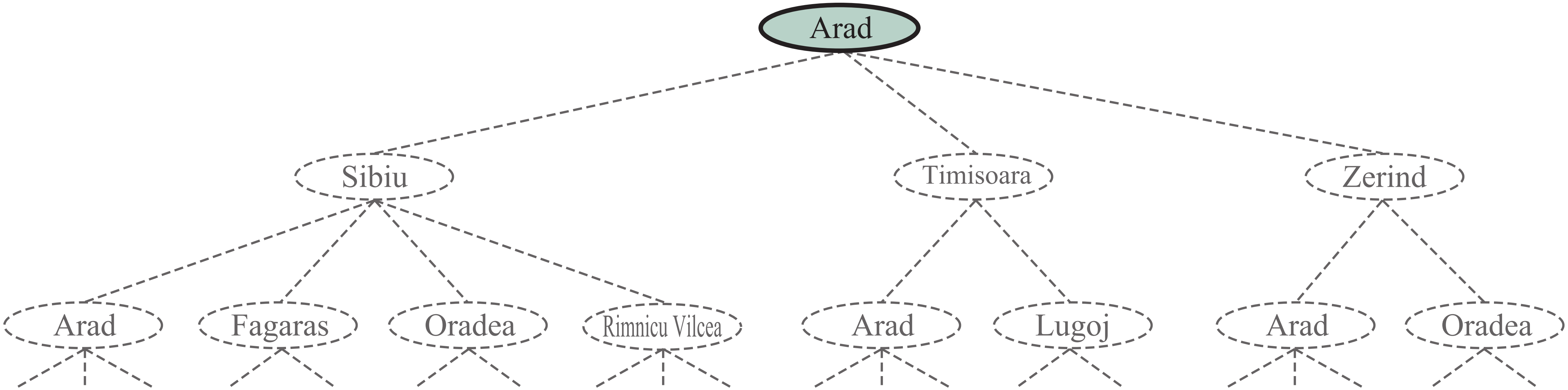
Tree Search



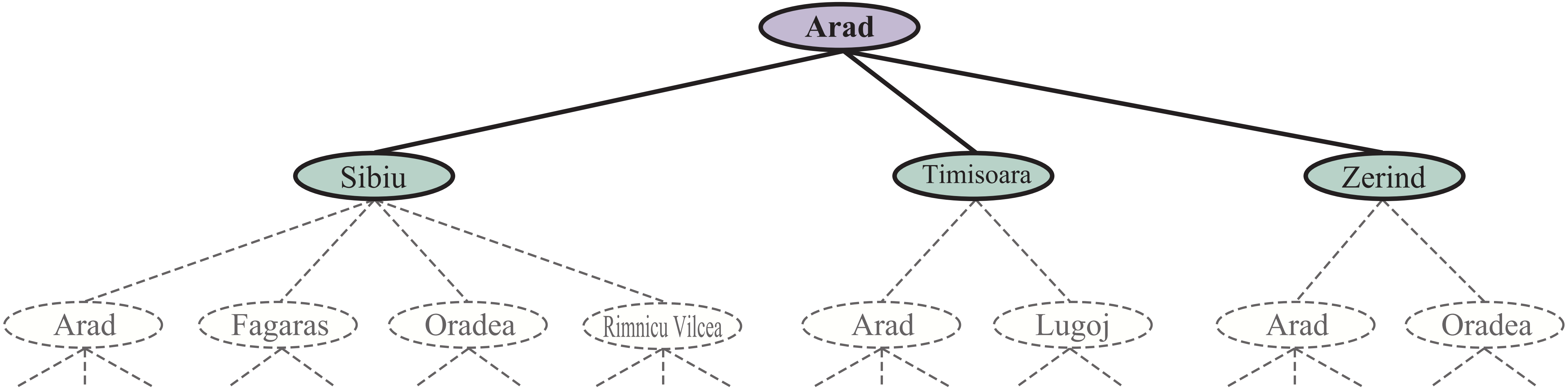
Search Example: Romania



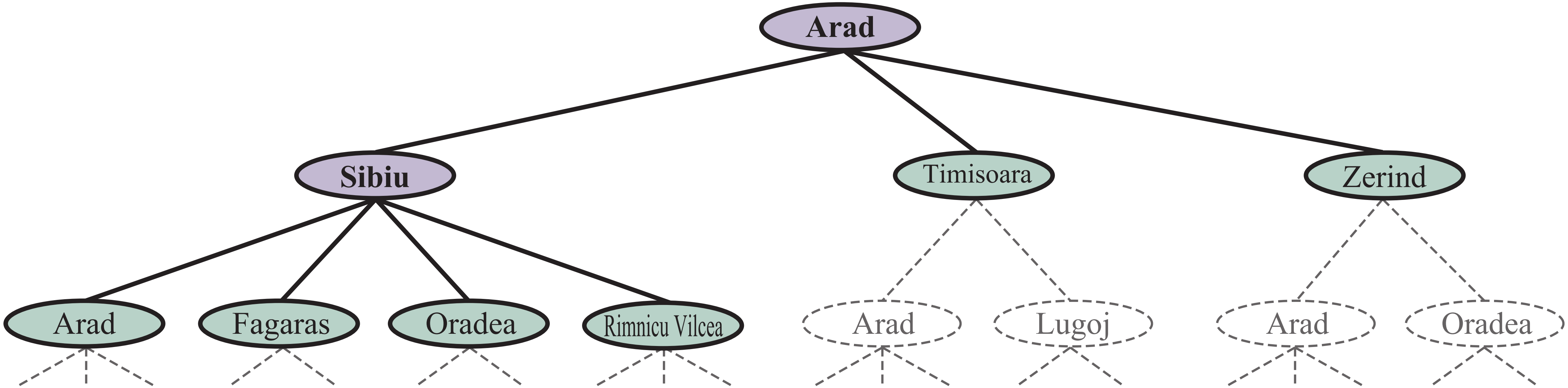
Creating the search tree



Creating the search tree



Creating the search tree



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
```

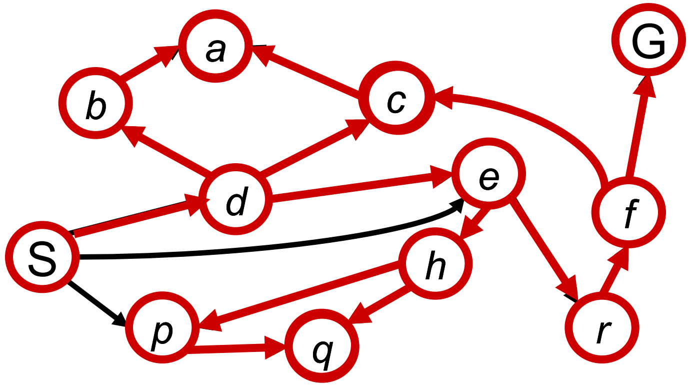
- Main variations:
  - Which leaf node to expand next
  - Whether to check for repeated states
  - Data structures for frontier, expanded nodes

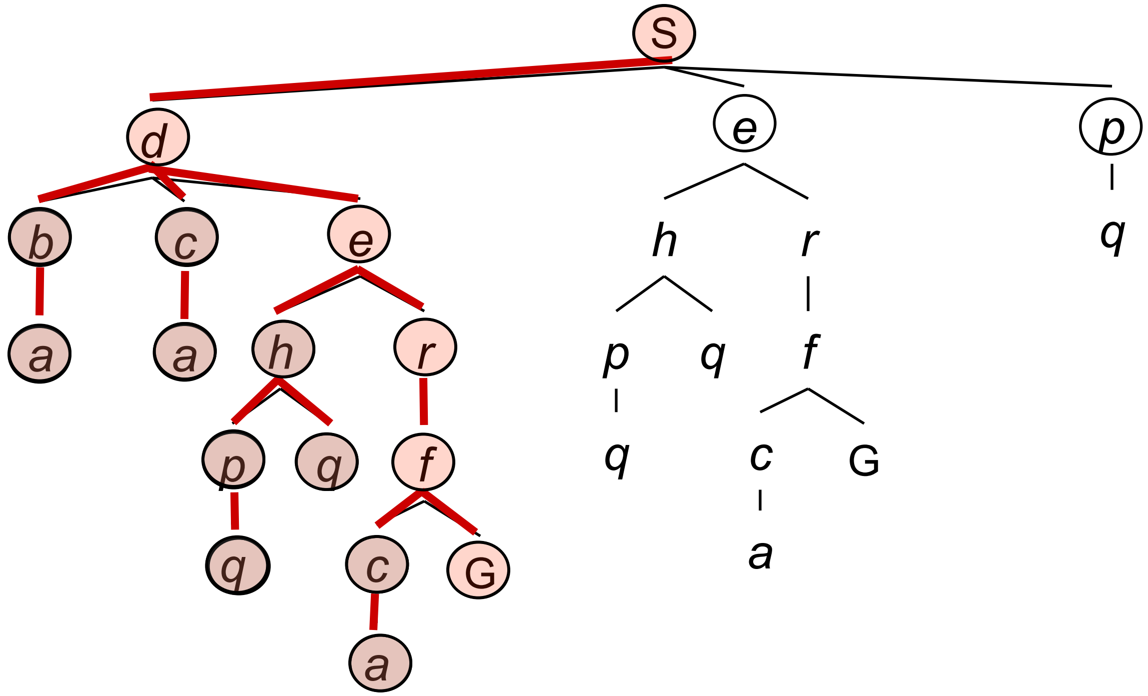
Depth-First Search

Depth-First Search

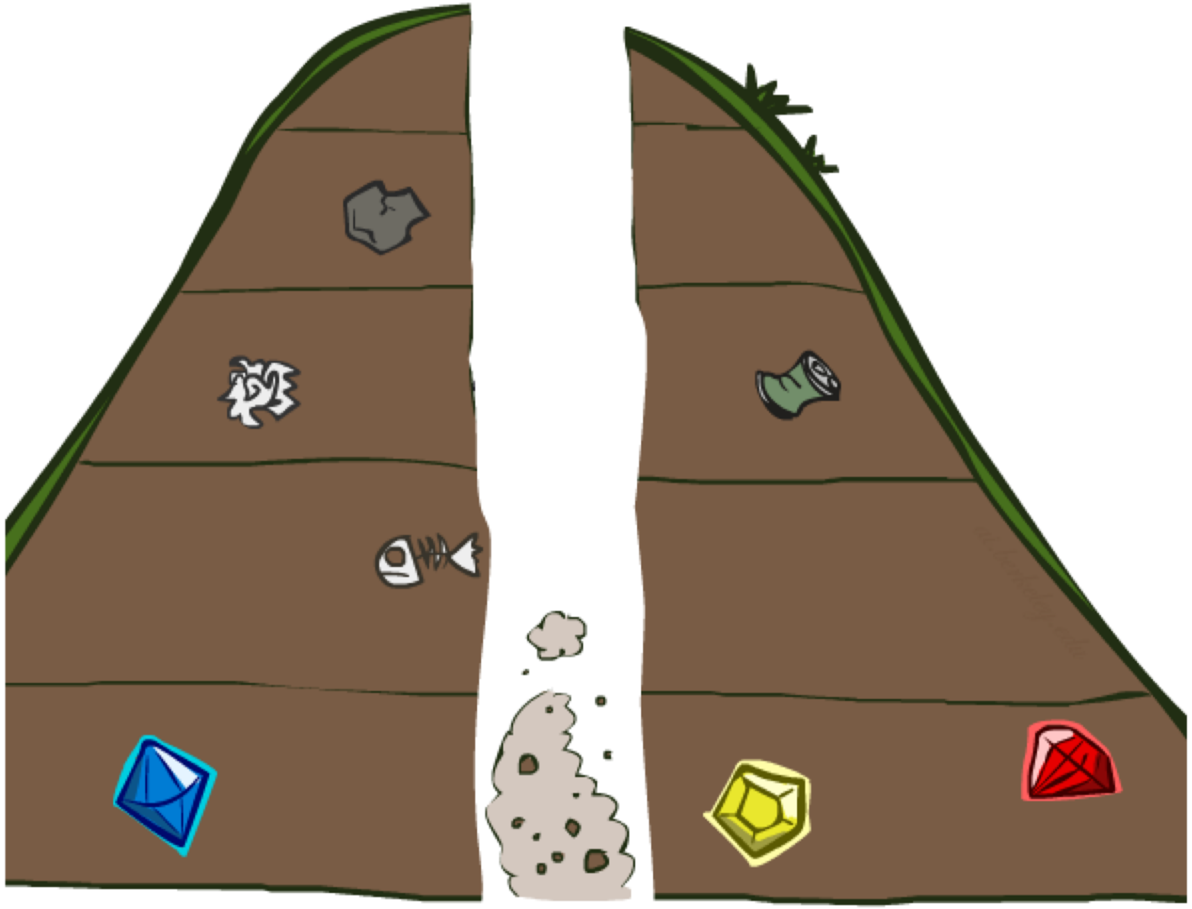
Strategy: expand a deepest node first

Implementation: Frontier is a LIFO stack





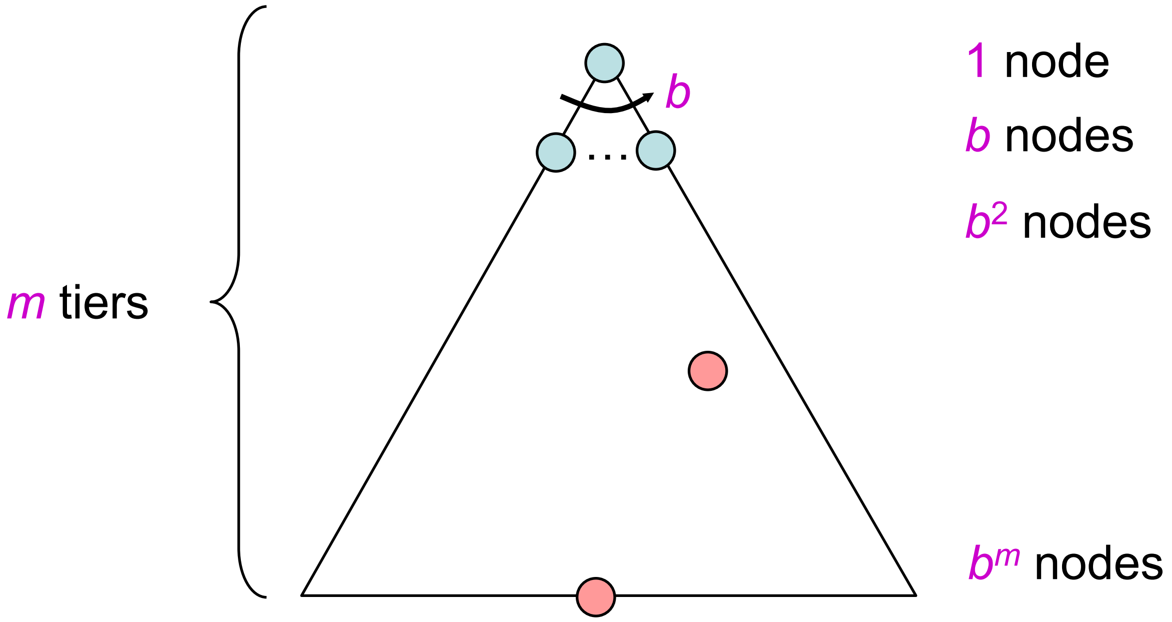
Search Algorithm Properties



Search Algorithm Properties

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

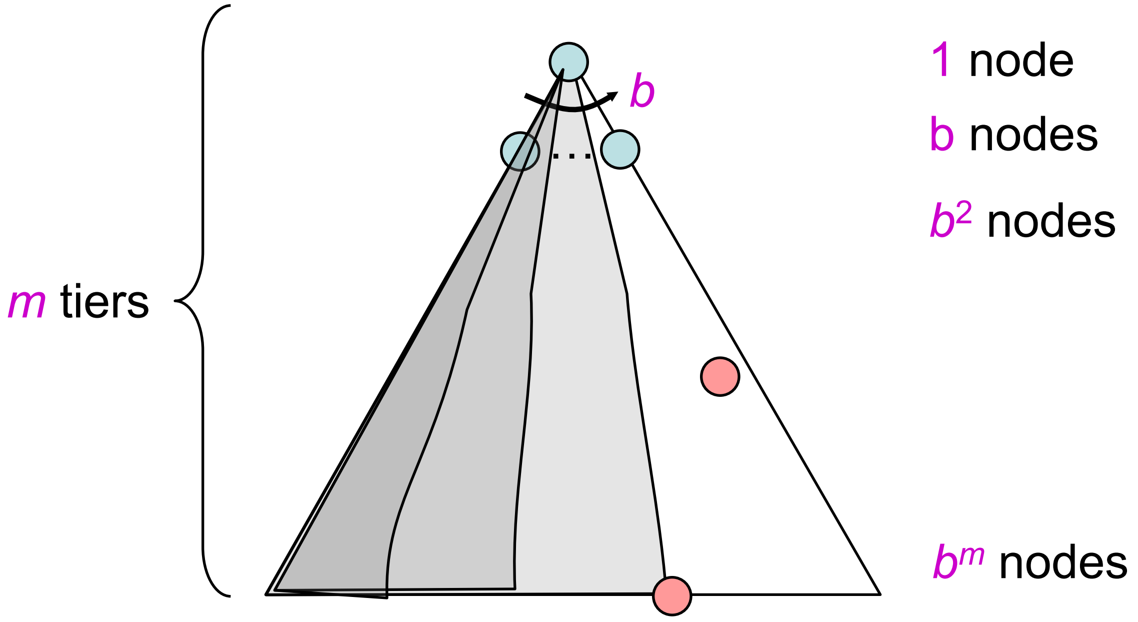
- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths



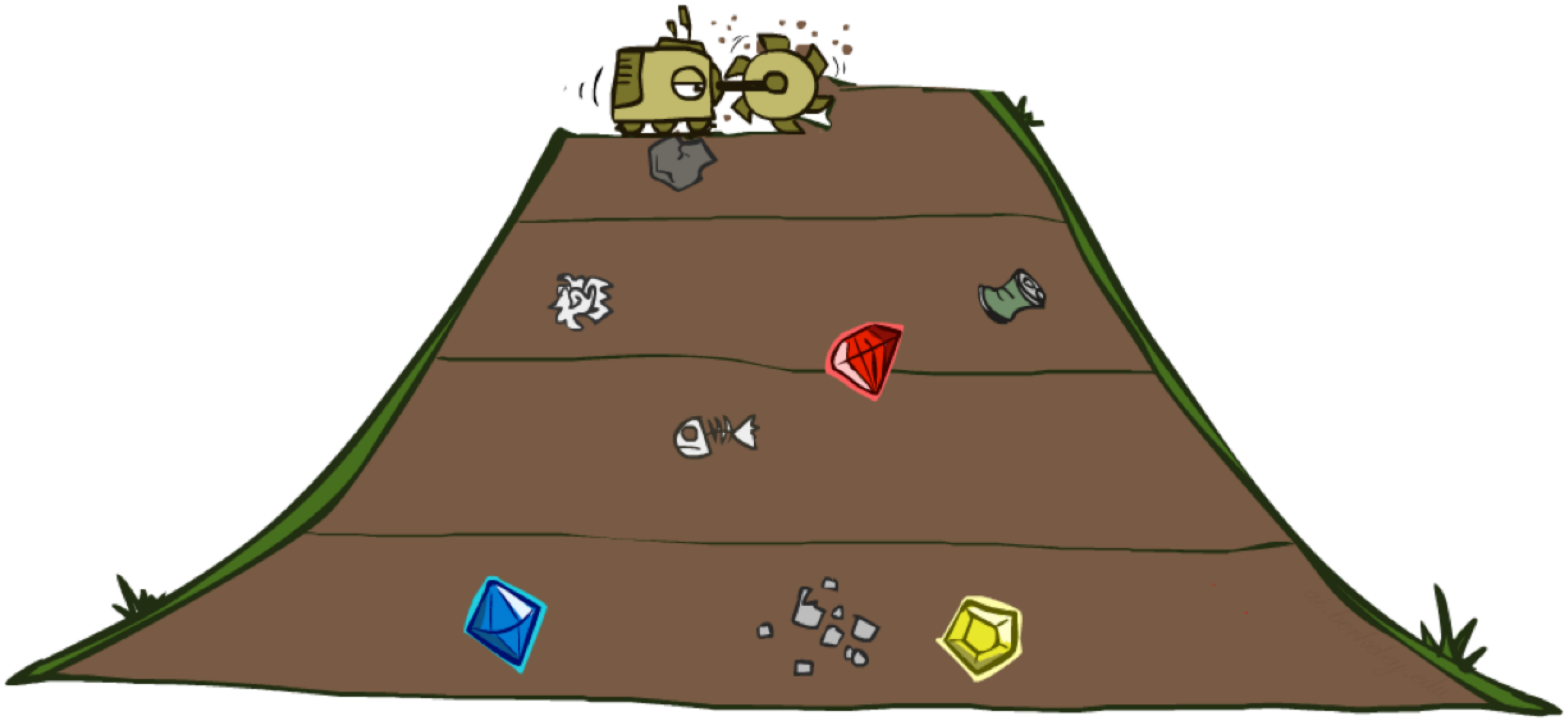
- Number of nodes in entire tree?
  - $1 + b + b^2 + ... b^m = O(b^m)$

Depth-First Search (DFS) Properties

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



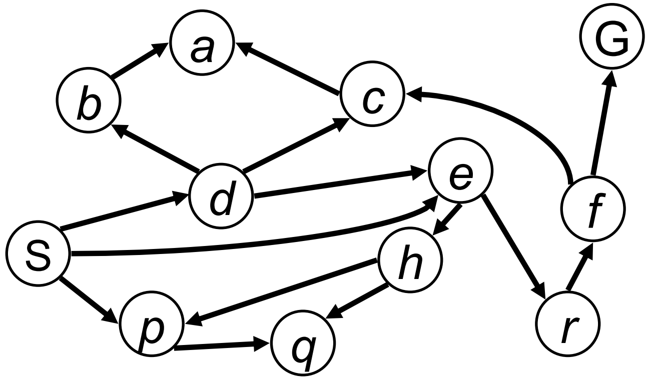
Breadth-First Search



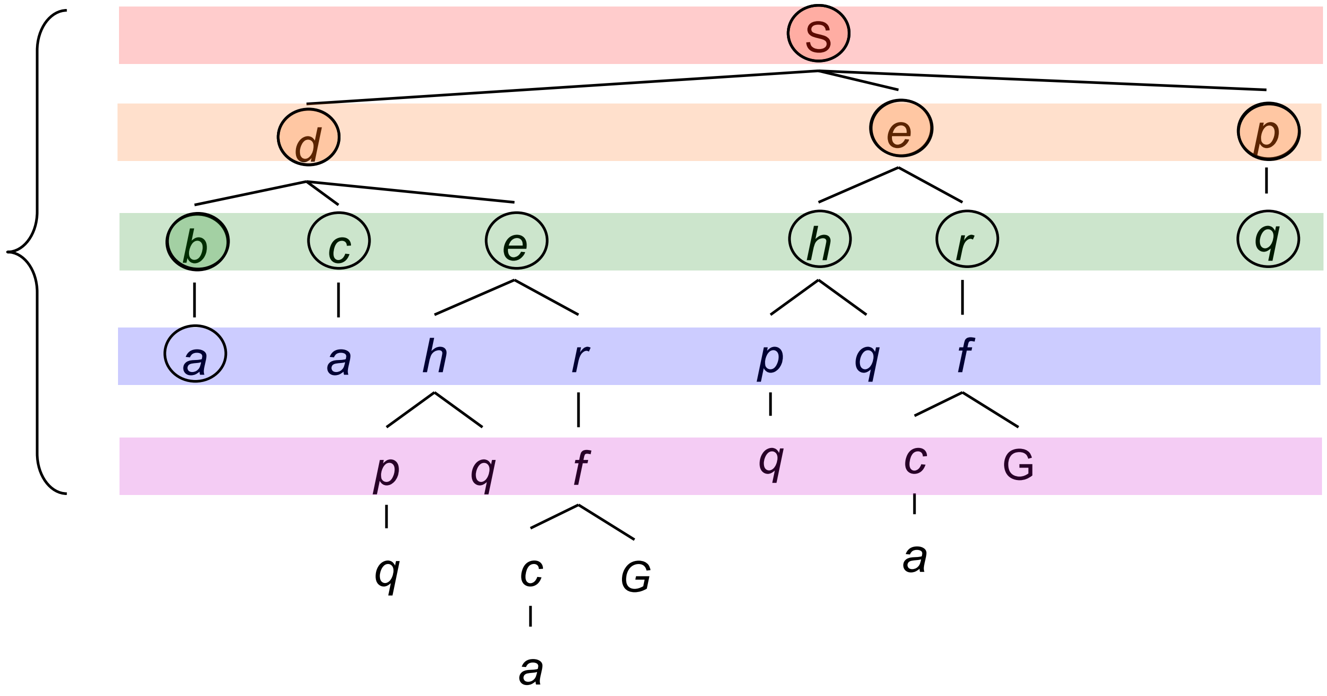
Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Frontier is a FIFO queue

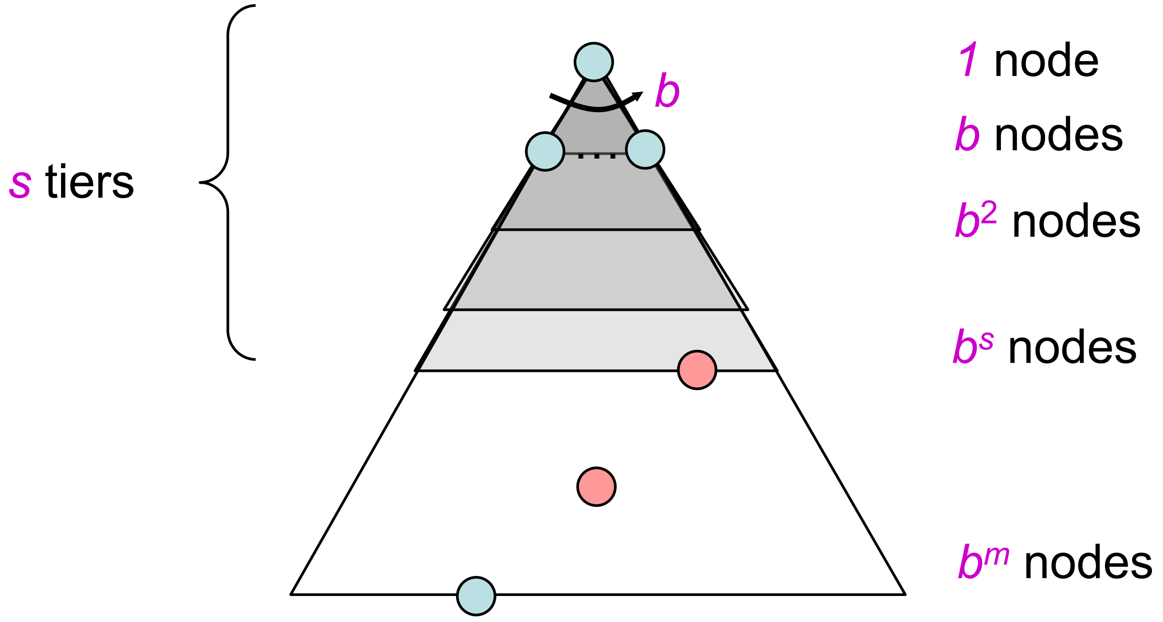


Search Tiers

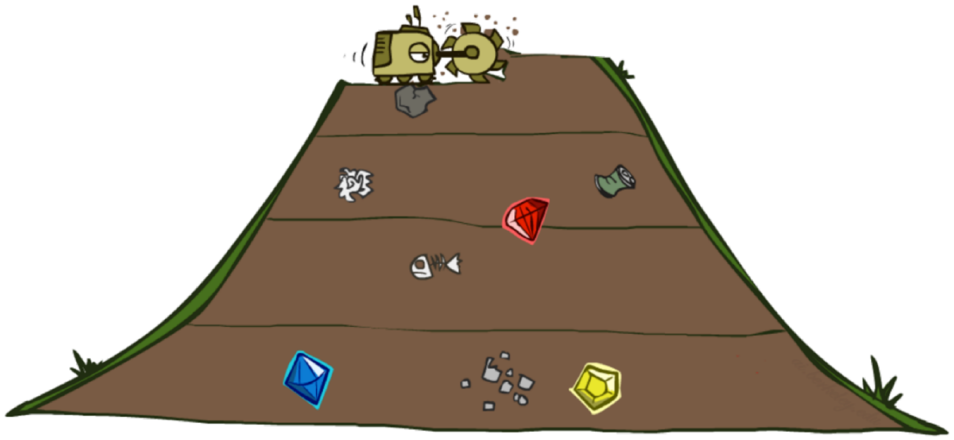


Breadth-First Search (BFS) Properties

- 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 frontier take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - If costs are equal (e.g., 1)



Quiz: DFS vs BFS



Quiz: DFS vs BFS

- When will BFS outperform DFS?

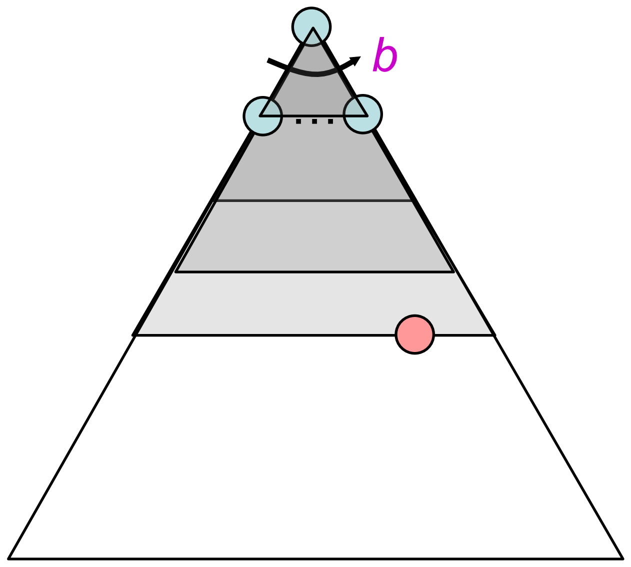
- When will DFS outperform BFS?

[Demo: dfs/bfs maze water (L2D6)]

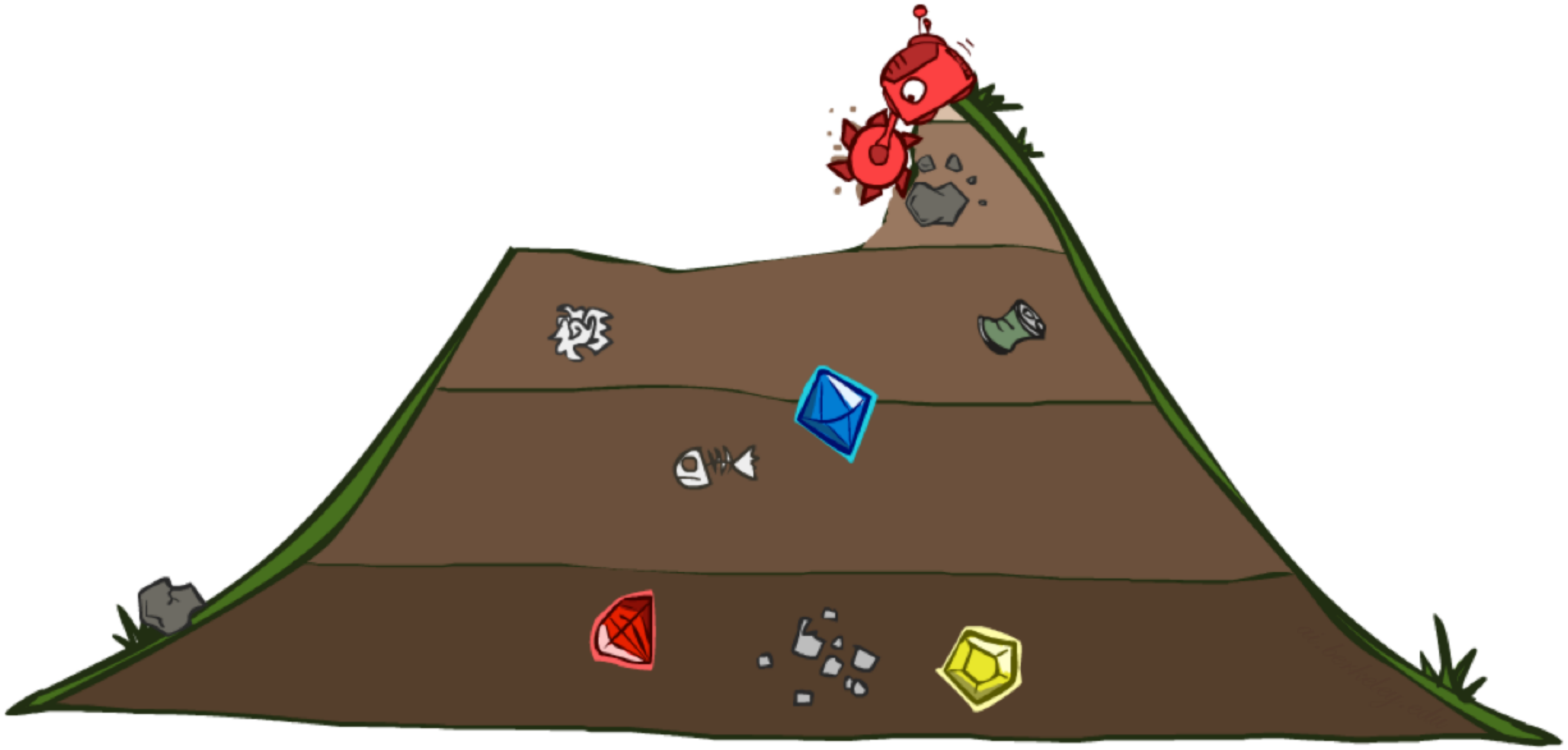
Iterative Deepening

- 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!
  - Extra work is $O(b^{s-1})$



Uniform Cost Search

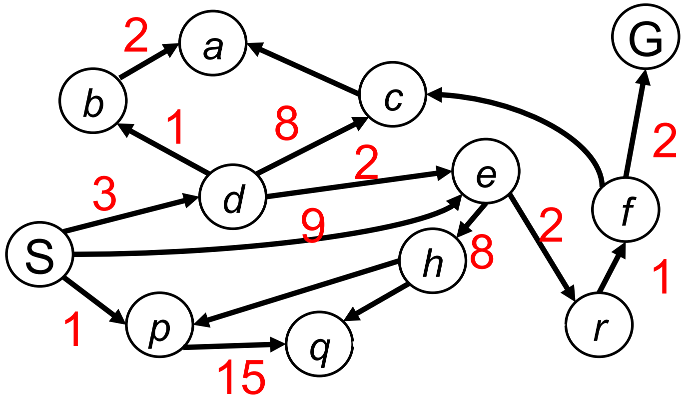


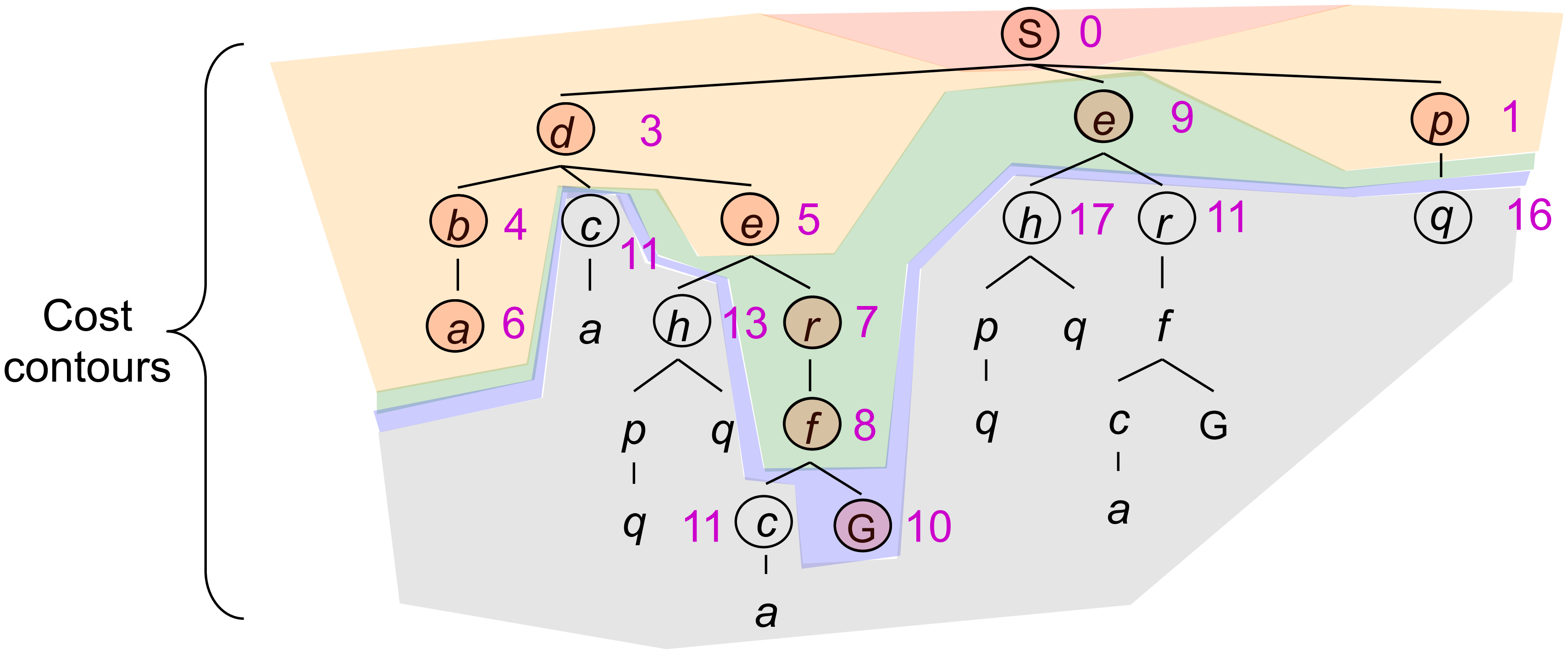
Uniform Cost Search

g(n) = cost from root to n

Strategy: expand lowest g(n)

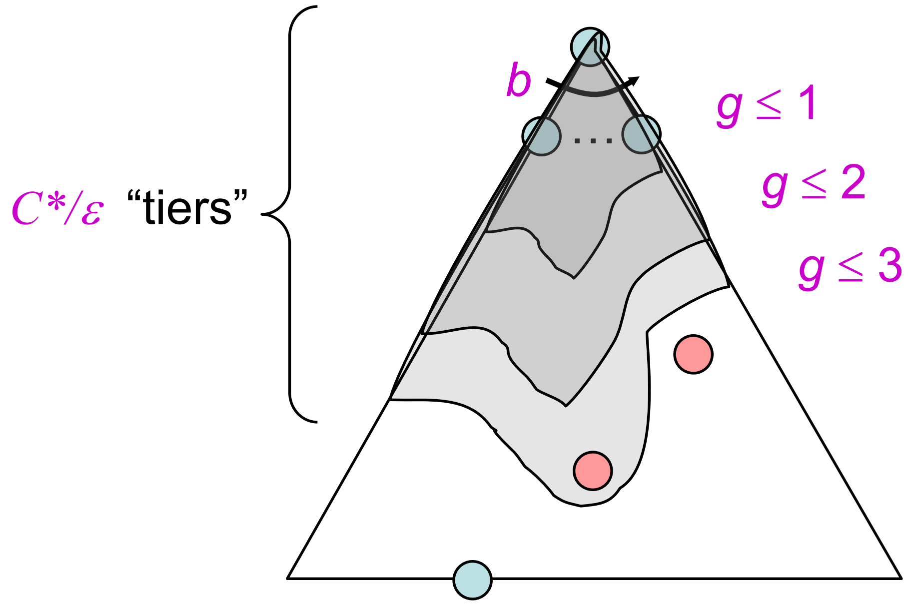
Frontier is a priority queue sorted by g(n)





Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Expands 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 frontier take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming C^* is finite and $\varepsilon > 0$, yes!
- Is it optimal?
  - Yes! (Proof next lecture via A*)



Summary

- Assume known, discrete, observable, deterministic, atomic
- Search problems defined by $\mathcal{S}$, s_0, $\mathcal{A}(s)$, $Result(s,a)$, $G(s)$, $c(s,a,s')$
- Search algorithms find action sequences that reach goal states
  - Optimal => minimum-cost
- Search algorithm properties:
  - Depth-first: incomplete, suboptimal, space-efficient
  - Breadth-first: complete, (sub)optimal, space-prohibitive
  - Iterative deepening: complete, (sub)optimal, space-efficient
  - Uniform-cost: complete, optimal, space-prohibitive

Bonus Search Algo Summary

Search	Frontier	Completeness	Optimality	Time	Space
DFS (Depth-First)	stack	tree search – no (cycle) graph search – yes (finite) / no (infinite)	no	$O(b^m)$	$O(bm)$
BFS (Breadth-First)	queue	yes	no (except when all edge costs same)	$O(b^s)$	$O(b^s)$
Iterative Deepening (BFS result w/ modified DFS algo)	stack (same as DFS)	yes (same as BFS)	no (same as BFS)	$O(b^s)$ (same as BFS)	$O(bs)$ (same as DFS but w/ shortest solution length)
UCS (Uniform Cost)	heap-based PQ (backward cost)	yes (assuming positive edge costs and $\epsilon > 0$)	yes (assuming positive edge costs and $\epsilon > 0$)	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$

b = branching factor (assume finite)

m = max depth of search tree

s = smallest depth of solution (assume finite)

C^* = cost of optimal solution (assume finite)

ϵ = minum cost between 2 nodes