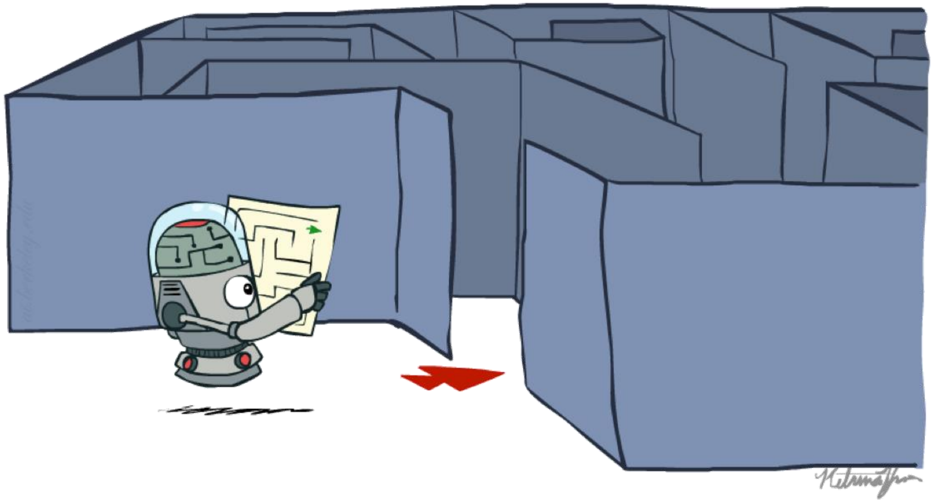
Part 1:
Solving Problems with Search

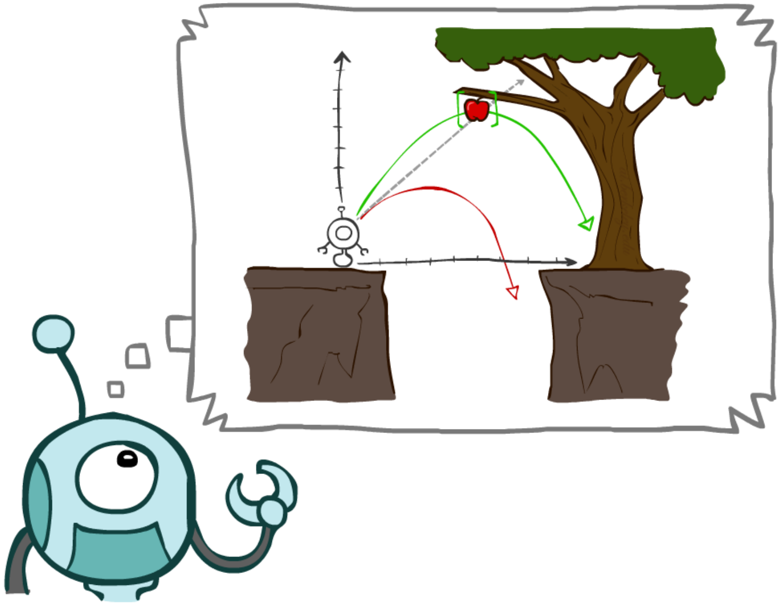
[Aknowledgment: Some Slides adapted from Dan Klein and Pieter Abbeel]

http://ai.berkeley.edu.]

Search

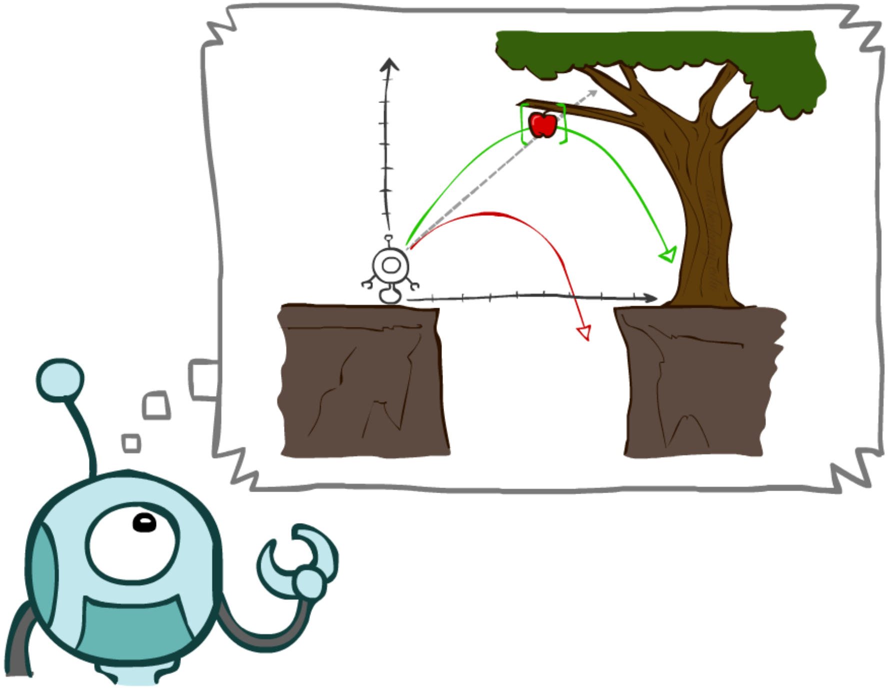


Today



- Agents that Plan Ahead
- Search Problems
- Search Algorithms (Review?)
  - Depth-First Search
  - Breadth-First Search
  - Iterative Deepening

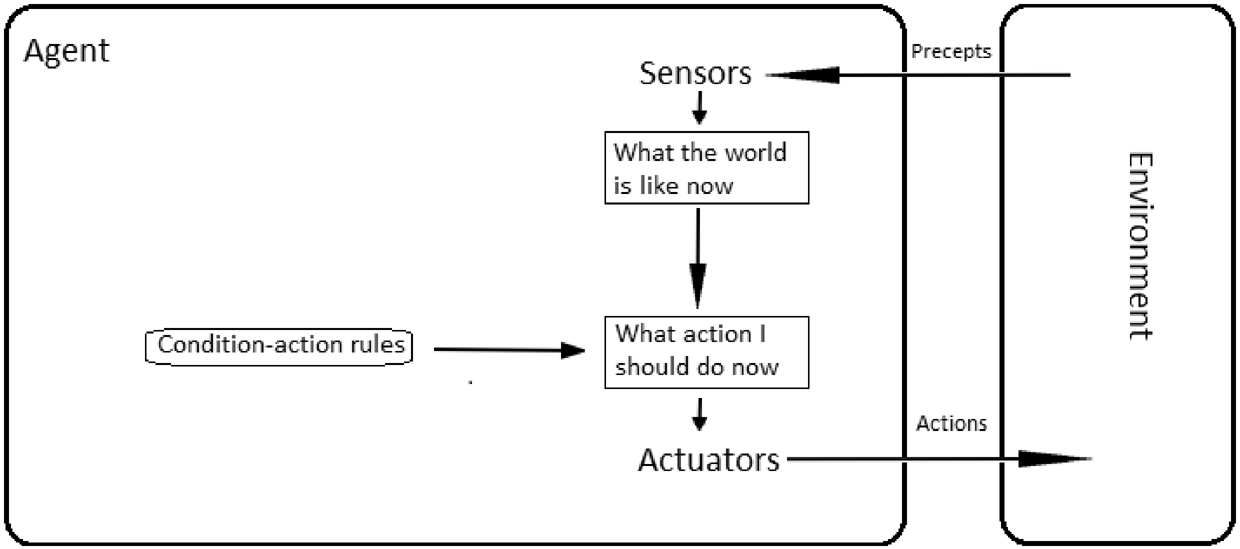
Agents that Plan



Agent types

- Five basic types in order of increasing generality:
  - Table Driven agents
  - **Simple reflex agents (you will make one now)**
  - Model-based reflex agents
  - Goal-based agents
    - Problem-solving agents
  - Utility-based agents
    - Can distinguish between different goals
  - Learning agents

Reflex Agents



- Act only on the basis of the current percept.
- Based on the *condition-action rules*:

  if condition

  then action
- **Infinite loops are common** (can be fixed with some randomization)

Example: Vacuum Agent (R&N, 2.1)

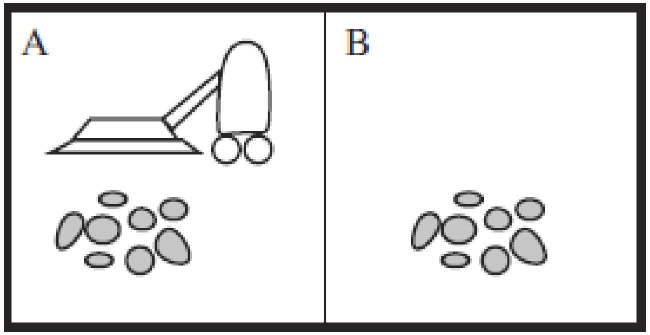


Figure 2.2 A vacuum-cleaner world with just two locations.

Percept sequence	Action
$[A, Clean]$	$Right$
$[A, Dirty]$	$Suck$
$[B, Clean]$	$Left$
$[B, Dirty]$	$Suck$
$[A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Dirty]$	$Suck$
$\vdots$	$\vdots$
$[A, Clean], [A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Clean], [A, Dirty]$	$Suck$
$\vdots$	$\vdots$

Vacuum Reflex agent (Python)

```
def ReflexVacuumAgent():
    "A reflex agent for the two-state vacuum environment.
    def program(percept):
        location, status = percept
        if status == 'Dirty':
            return 'Suck'
        elif location == loc_A:
            return 'Right'
        elif location == loc_B:
            return 'Left'
    return Agent(program)
```

https://github.com/aimacode/aima-python/blob/master/agents.py

Reflex Agent for Pacman

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

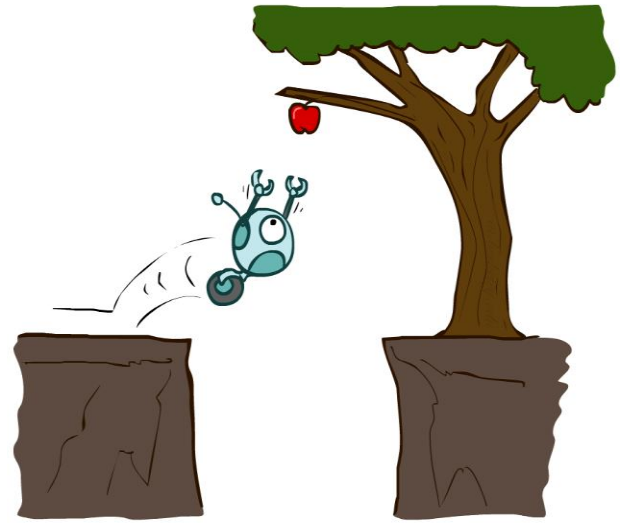
In-Class Exercise: WallFollower Agent

- Step 1: download search.zip from Canvas
- Step 2: find searchAgents.py
- Step 3: Copy code below, rename to new class

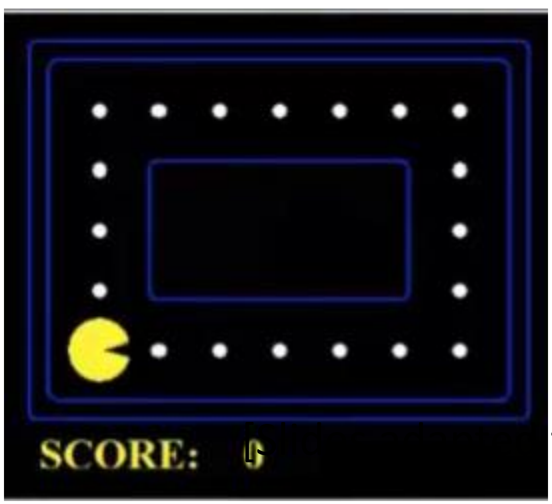
```
class GoWestAgent(Agent): → WallFollower
"An agent that goes West until it can't."
   def getAction(self, state):
   "The agent receives a GameState (defined in pacman.py)."
   if Directions.WEST in state.getLegalPacmanActions():
     return Directions.WEST
    else:
     return Directions.STOP
```

Reflex Agents in General

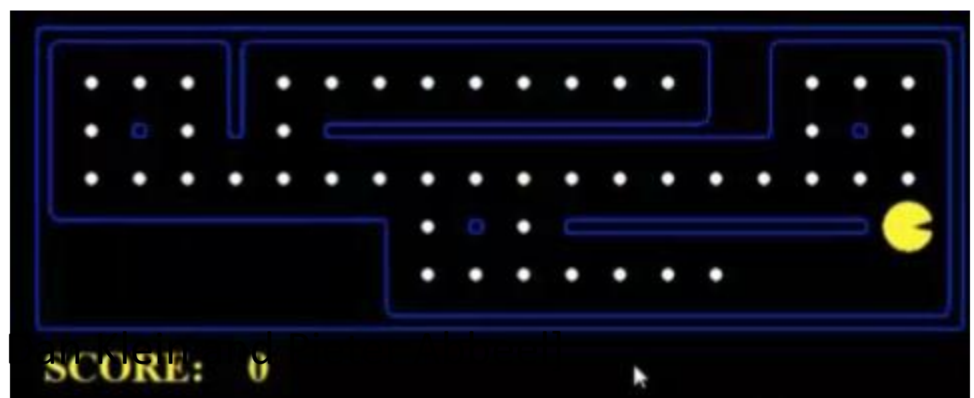
- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS



- Can a reflex agent be rational?



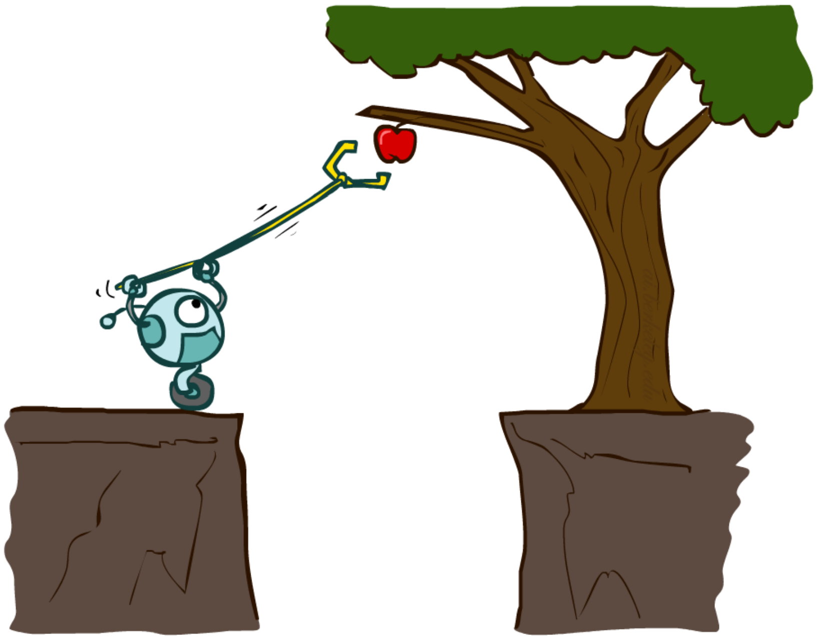
from



http://ai.berkeley.edu.]

Planning Agents

- Planning agents:
  - Ask "what if"
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE
- Optimal vs. complete planning
- Planning vs. replanning



Rational agents

- Performance measure: An objective criterion for success of an agent's behavior, e.g.,
  - Robot driver?
  - Chess-playing program?
  - Spam email classifier?

- Rational Agent: selects actions that is *expected* to **maximize its performance measure,**
  - given percept sequence
  - given agent's built-in knowledge

  - consider: how to maximize expected future performance, with only historical data?

Task Environment

- Before we design an intelligent agent, we must specify its “task environment”:

PEAS:

Performance measure

Environment

Actuators

Sensors

PEAS: Example

- Example: Agent = robot driver in DARPA Challenge
  - Performance measure:
    -
  - Environment:
    -
  - Actuators:
    -
  - Sensors:
    -

Stanley

PEAS: Example

- Example: Agent = robot driver in DARPA Challenge

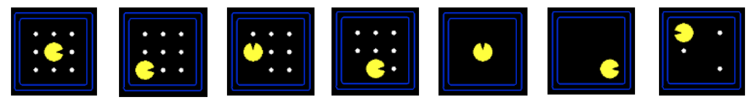
  - Performance measure:
    - Time to complete course
  - Environment:
    - Road, obstacles
  - Actuators:
    - Steering wheel, accelerator, brake, signal, horn
  - Sensors:
    - Optical cameras, lasers, sonar, accelerometer, speedometer, GPS, odometer, engine sensors, …

Goal-Based Agents: Search

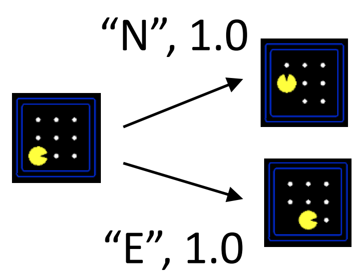
- Five basic types in order of increasing generality:
  - Table Driven agents
  - Simple reflex agents
  - Model-based reflex agents
  - Goal-based agents
    - Problem-solving agents: solve problems by searching for solution
  - Utility-based agents
  - Learning agents

Search Problems

- A search problem consists of:
  - State space



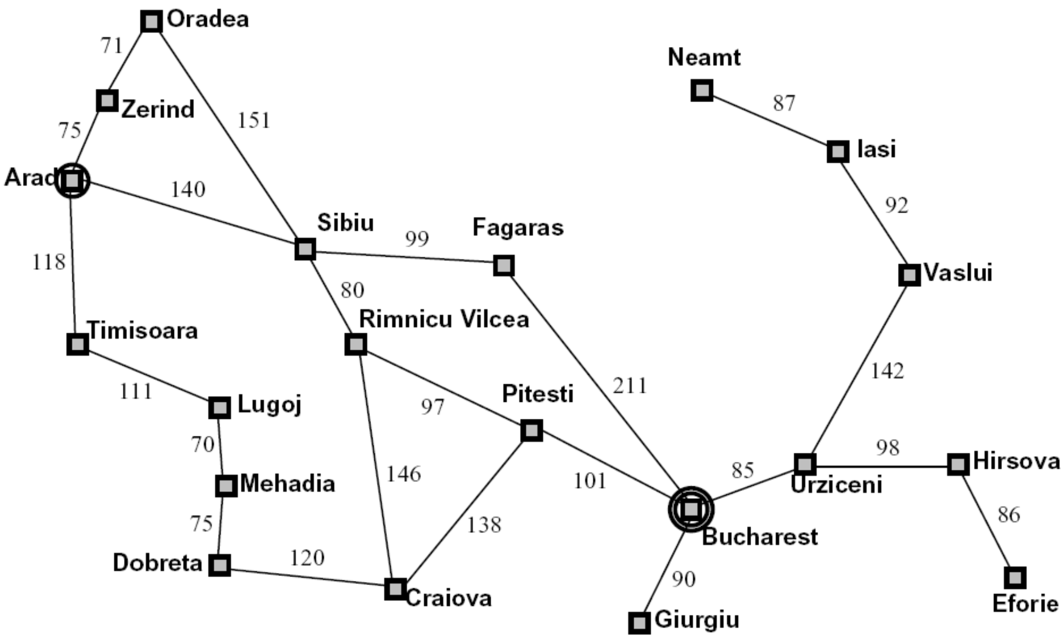
  - Successor function (with actions, costs)



  - Start state and a goal test

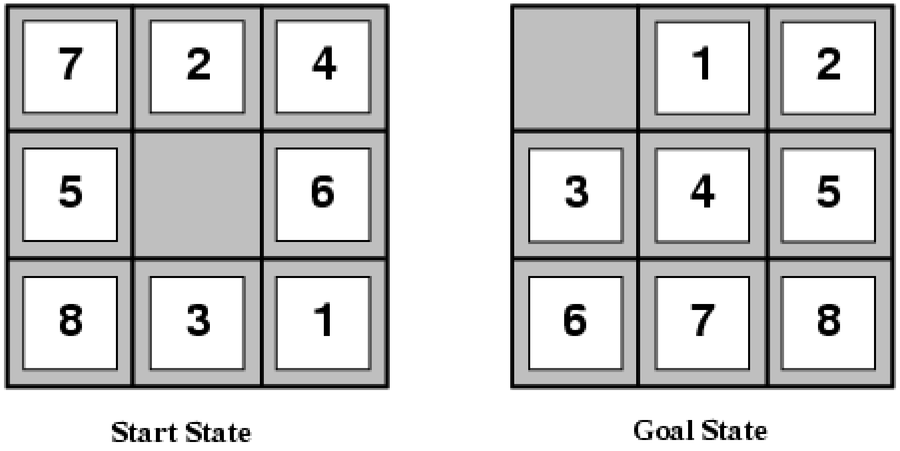
- A **solution** is a sequence of **actions** which transforms the start state to a goal state

Example: Traveling in Romania



- State space:
  -
- Successor function:
  -
- Start state:
  -
- Goal test:
  -
- Solution?

Example: 8-puzzle



- states?
- initial state?
- Successor function?
- goal test?
- solution?

Path finding: Amazon PrimeAir™

- First successful delivery in Cambridge, U.K in December
- Fill in the blank:

  states?

  initial state?

  Successor function?

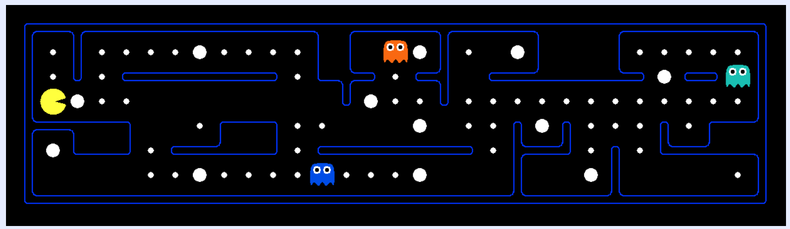
  goal test?

  solution?

https://www.youtube.com/watch?v=vNySOrI2Ny8

How to represent State Space?

The world state includes every last detail of the environment

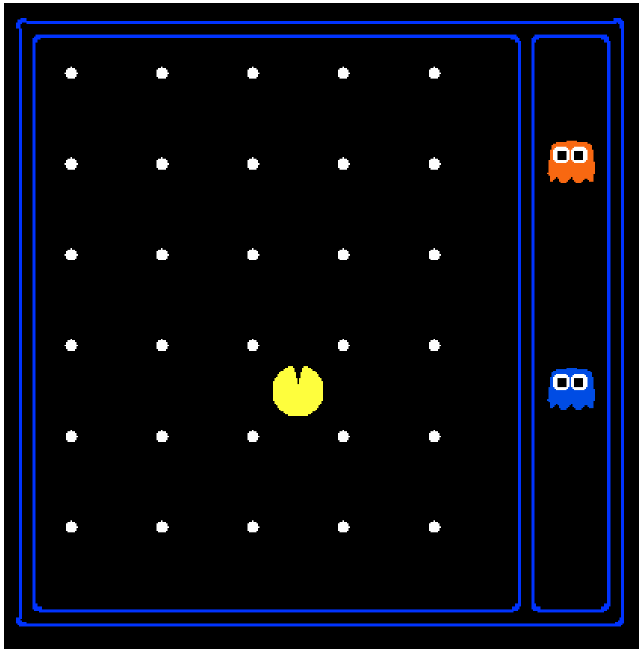


A search state keeps **only** the details needed for planning (abstraction).
Note: *Static info (e.g., walls) do not need to be stored for each state*

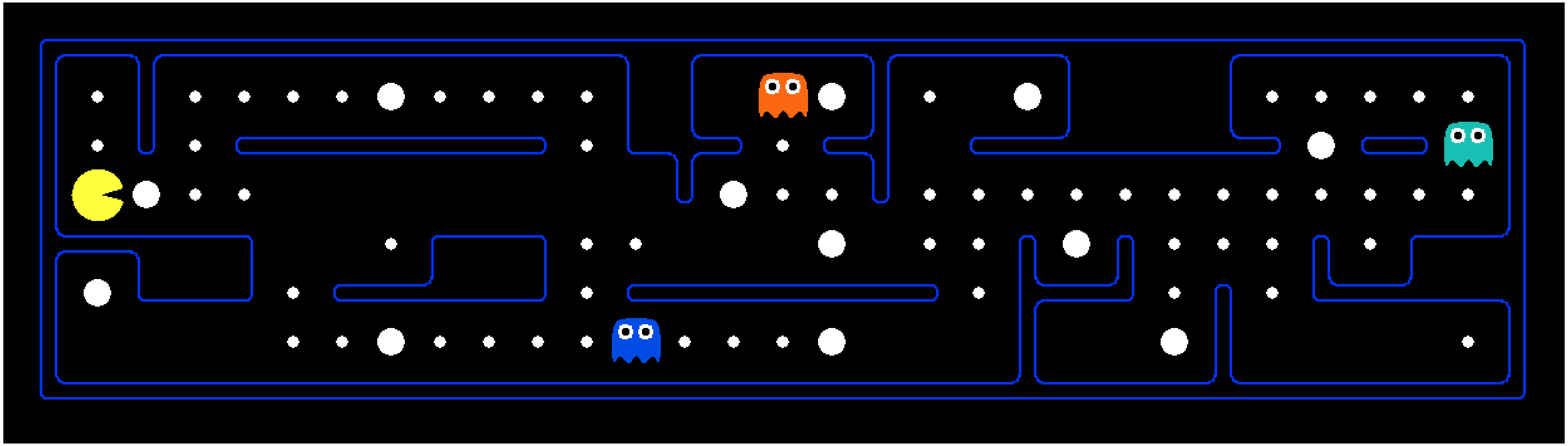
- Problem: Path-Finding
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END
- Problem: Eat-All-Dots
  - States: {(x,y), dots: bool array}
  - Actions: NSEW
  - Successor: update location and possibly dots
  - Goal test: dots all false

State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent direction: NSEW
- How many
  - World states?
    $120 \text{x}(2^{30})\text{x}(12^2)\text{x}4$
  - States for path-finding?
    120
  - States for eat-all-dots?
    $120 \text{x}(2^{30})$



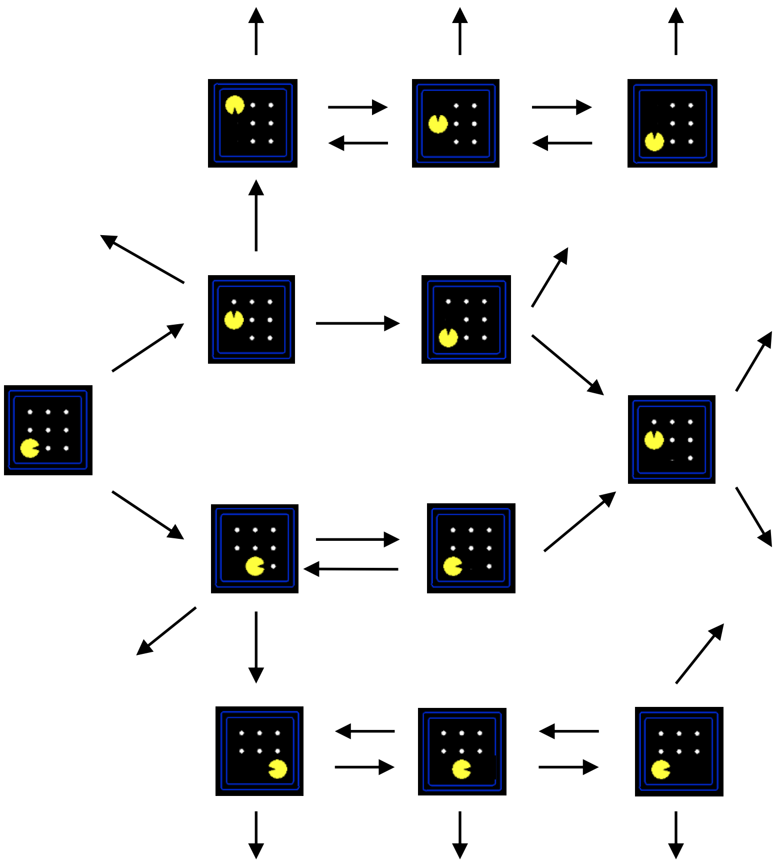
Exercise: Safe Passage



- Problem: eat all dots while keeping the ghosts scared
- What does the state space have to specify?

State Space Graphs: 1

- **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!
  - Careful: if there are loops in this graph, keep track of visited states
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

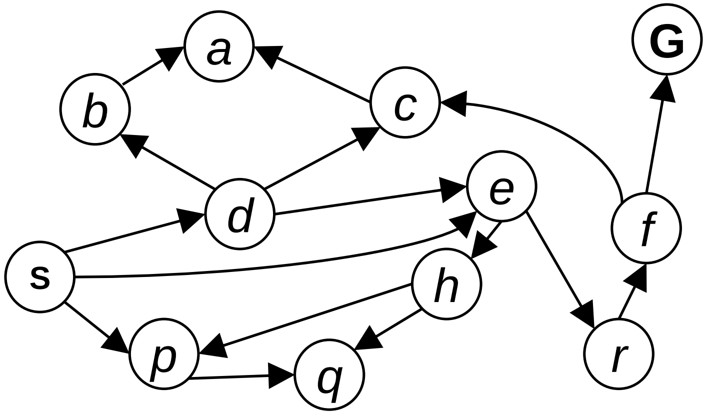


State Space Graphs: 2

- 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)

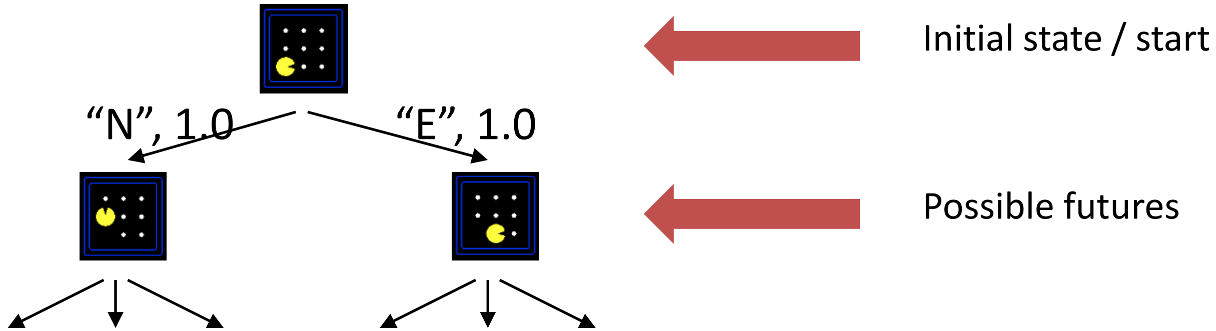
- In a search graph, each state occurs only once!

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



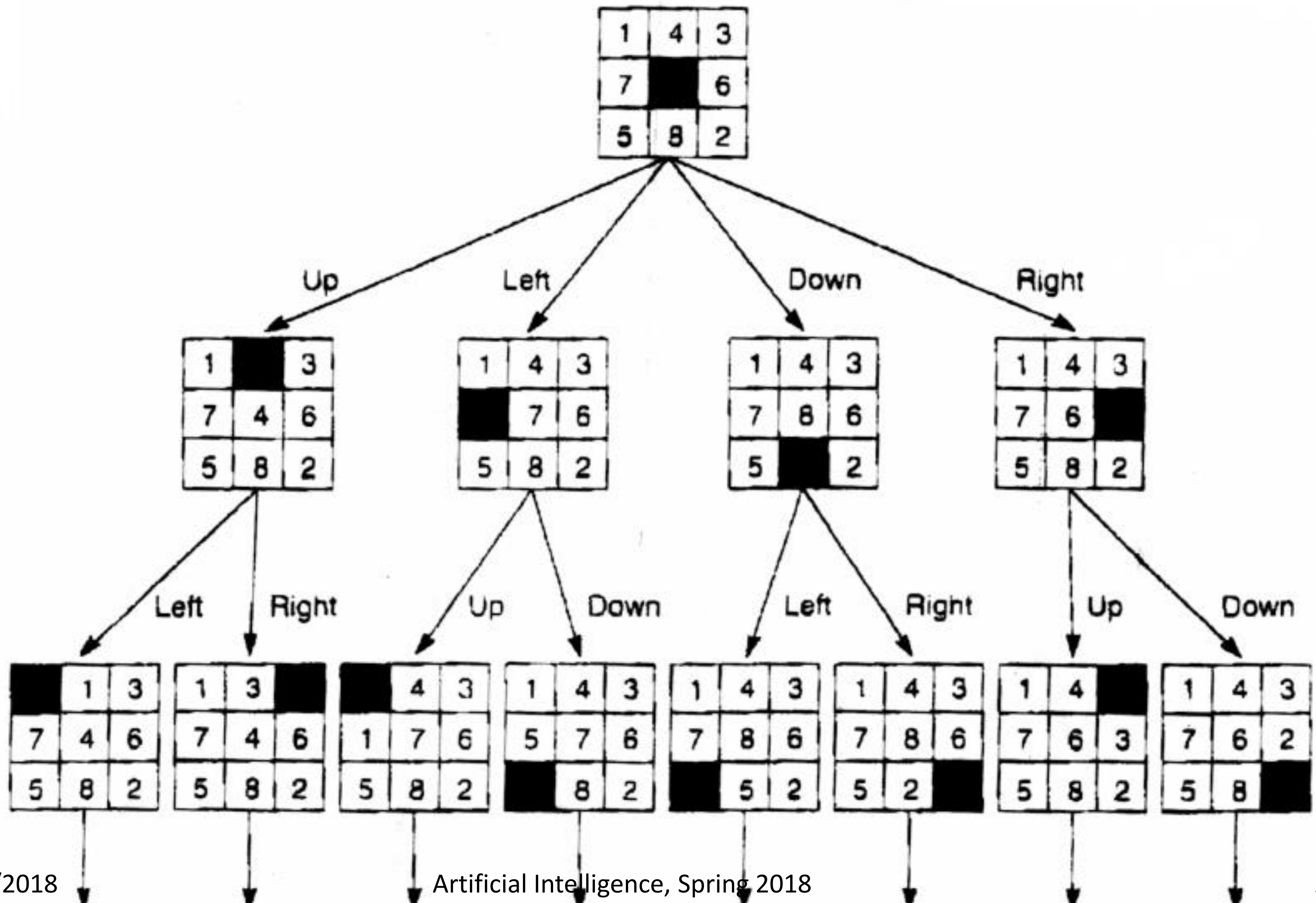
Tiny search graph for a tiny search problem

Search Trees

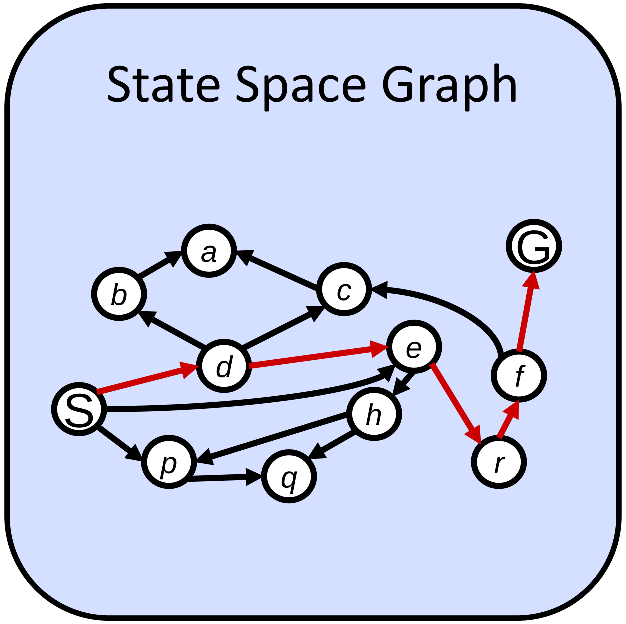


- A search tree:
  - A “what if” tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to **PLANS (paths from root to the state)**
  - For most problems, we can never actually build the whole tree (too large)

Search Tree for 8 puzzle problem

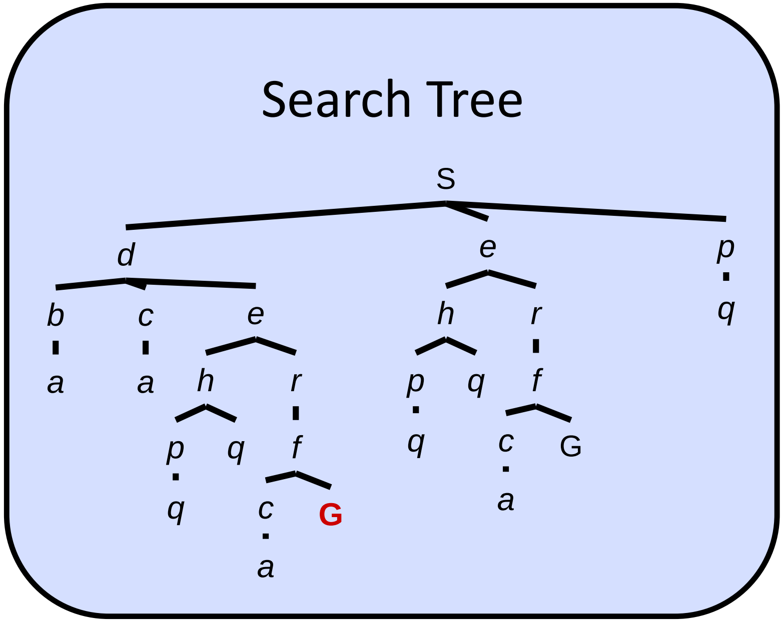


State Space Graphs vs. Search Trees



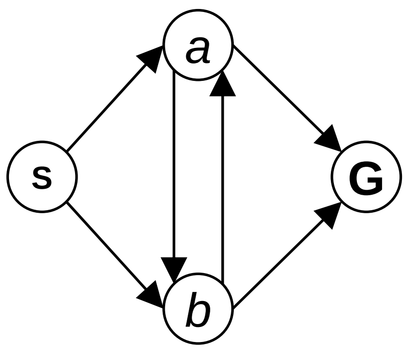
Each NODE in in the search tree represents entire PATH in the state space graph.

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

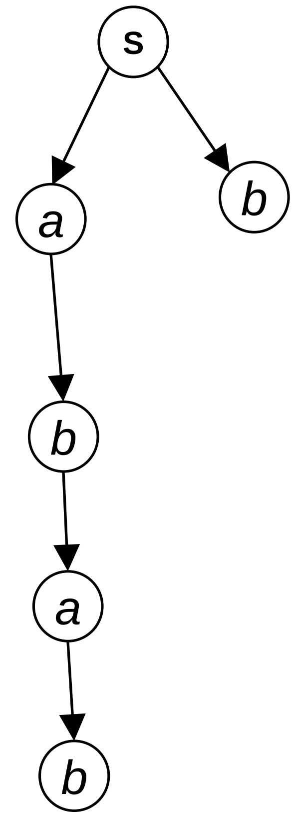


Exercise: State Space Graphs vs. Search Trees

Consider this 4-state graph:

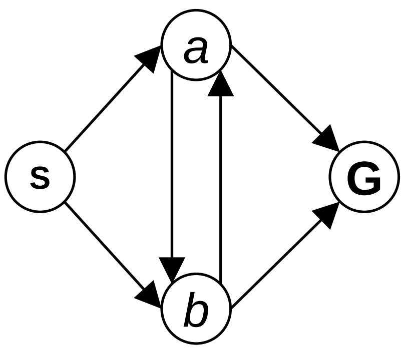


Draw the <u>Search Tree</u>:

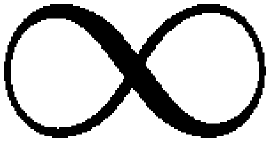


Exercise: State Space Graphs vs. Search Trees

Consider this 4-state graph:

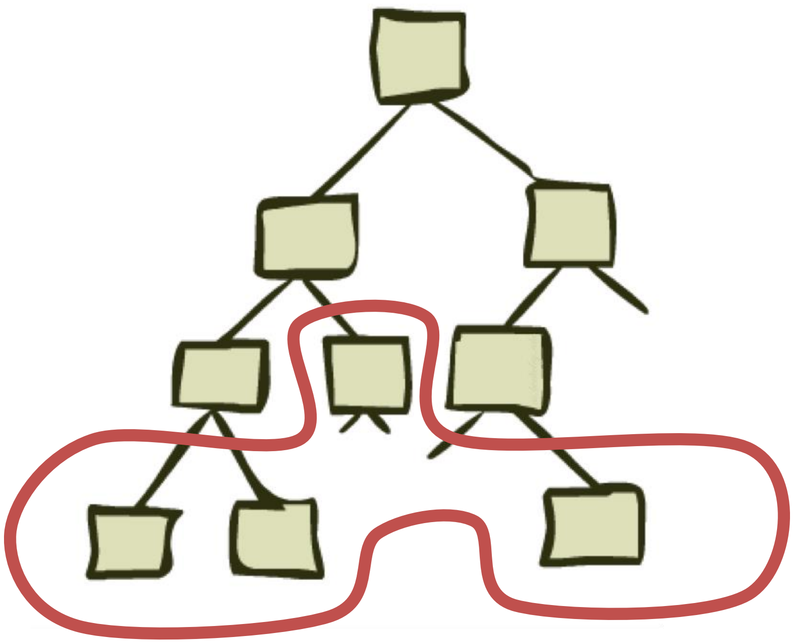


How big is the <u>Search Tree</u>?

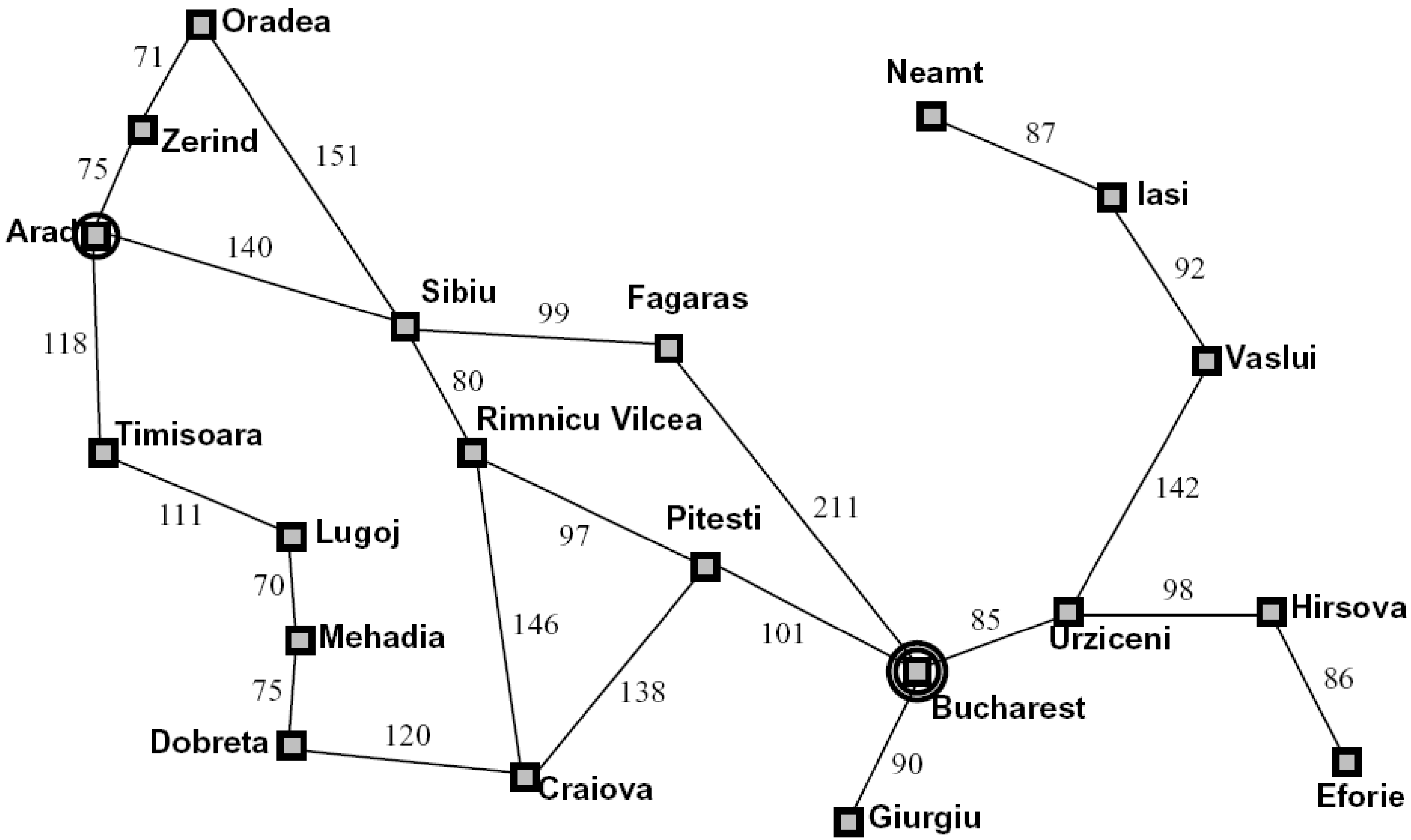


<u>Problem</u>: Lots of repeated structures in the search tree!

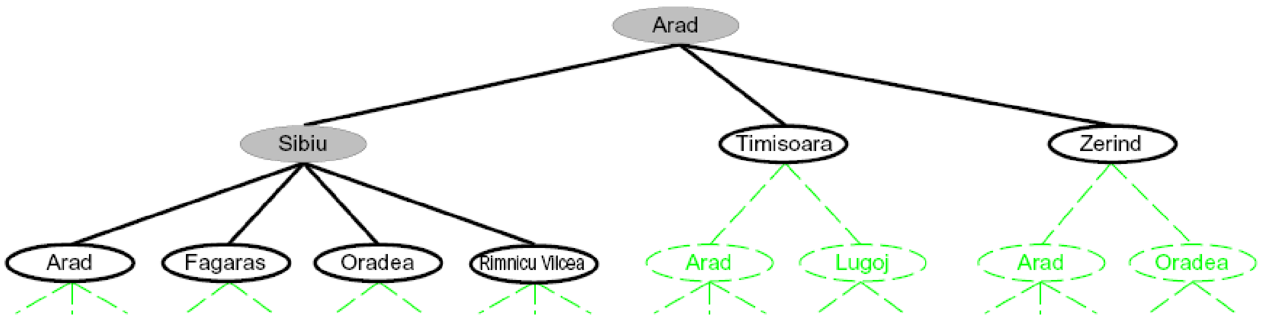
Tree Search



Search Example: Romania



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?

Python Implementation

```
def tree_search(problem, fringe):
    """Search through the successors of a problem to find a goal.
    The argument fringe should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    fringe.append( Node(problem.initial) )

    while fringe:  //a.k.a: fringe.len()>0
        node = fringe.pop()

        if problem.goal_test(node.state):
            return node

        fringe.extend(node.expand(problem))

    return None
```

adapted from https://github.com/aimacode/aima-python/blob/master/search.py

Why Search can be hard

Assuming b=10, 1000 nodes/sec, 100 bytes/node

Depth of Solution	Nodes to Expand	Time	Memory
0	1	1 millisecond	100 bytes
2	111	0.1 seconds	11 kbytes
4	11,111	11 seconds	1 megabyte
8	10^8	31 hours	11 giabytes
12	10^{12}	35 years	111 terabytes

$$P(40) \approx \frac{64!}{32!\,(8!)^2\,(2!)^6} \approx 10^{43}.$$

Sidebar: Search vs. Intuition

Tania Sachdev

- Human chess grandmasters think "only" 3-5 moves ahead (Kasparov occasionally 12-14) but rely on patterns, intuition

- Deep blue and others: <u>exhaustive search</u> for optimal state/solution. Evaluates 100M positions/sec, vs. Kasparov 3 positions/sec

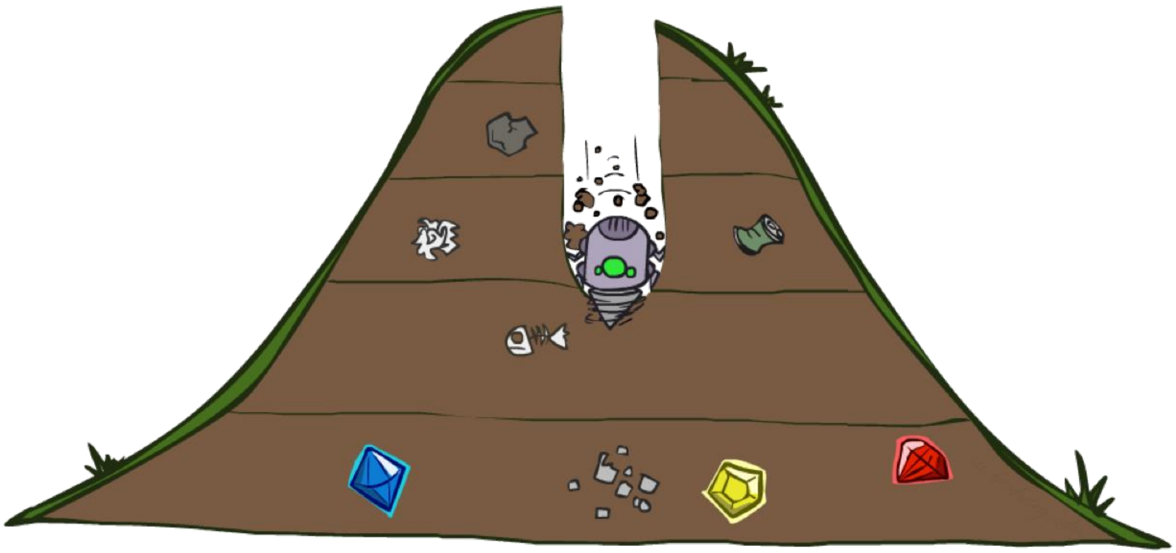
Search Strategies: Notation

- A search strategy is defined by picking the order of node expansion (fringe exploration)

- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - ***b***: maximum branching factor of the search tree
  - ***d***: depth of the least-cost solution
  - ***m***: maximum depth of the state space (may be ∞)

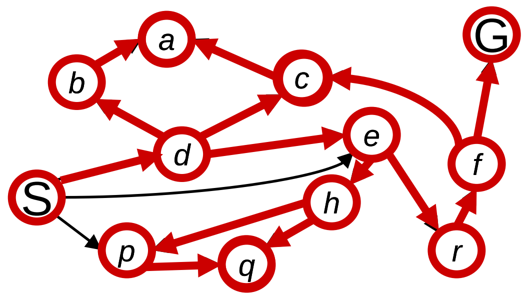
Depth-First Search

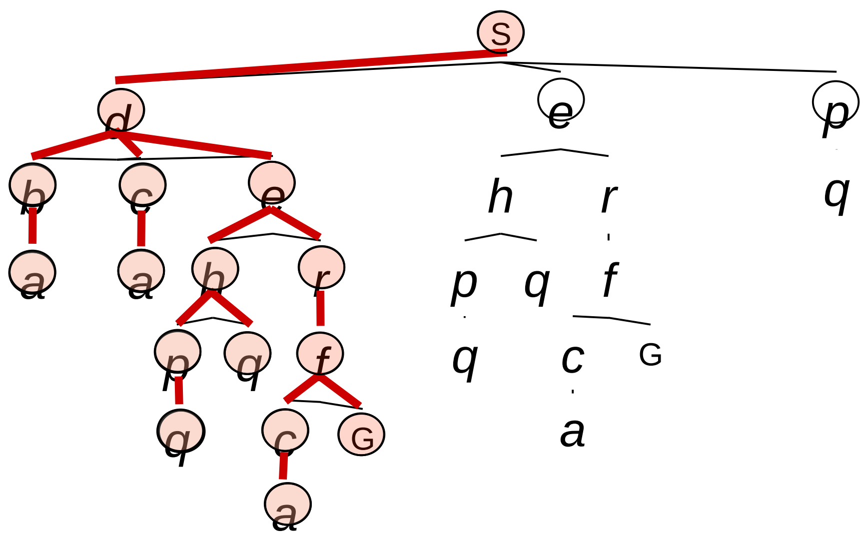


Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack

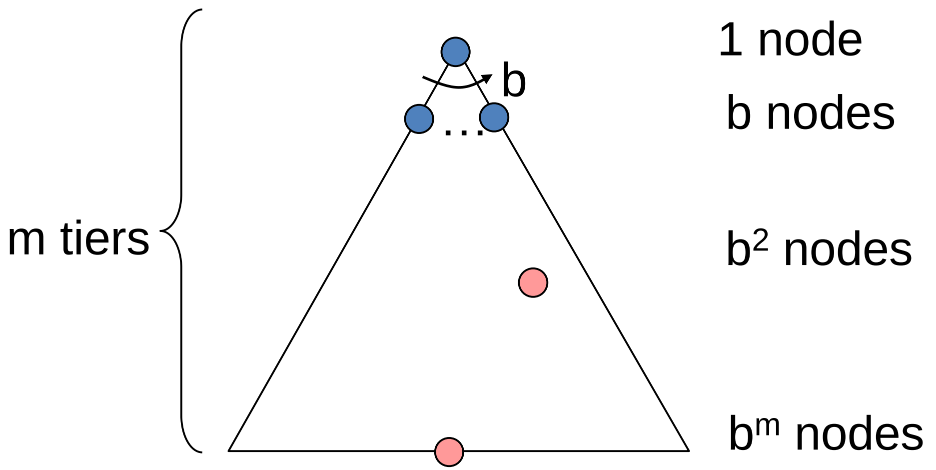




Search Algorithm Properties

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

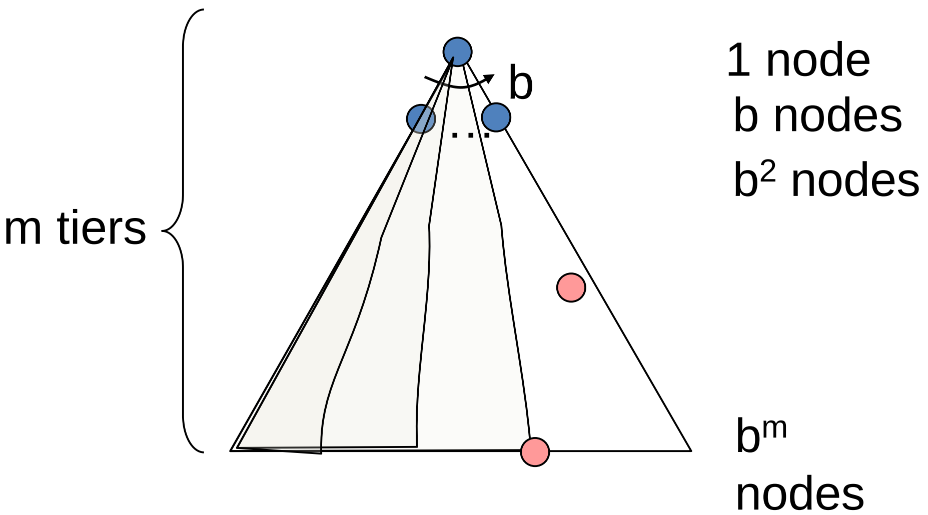
- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths



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

Depth-First Search (DFS) Properties

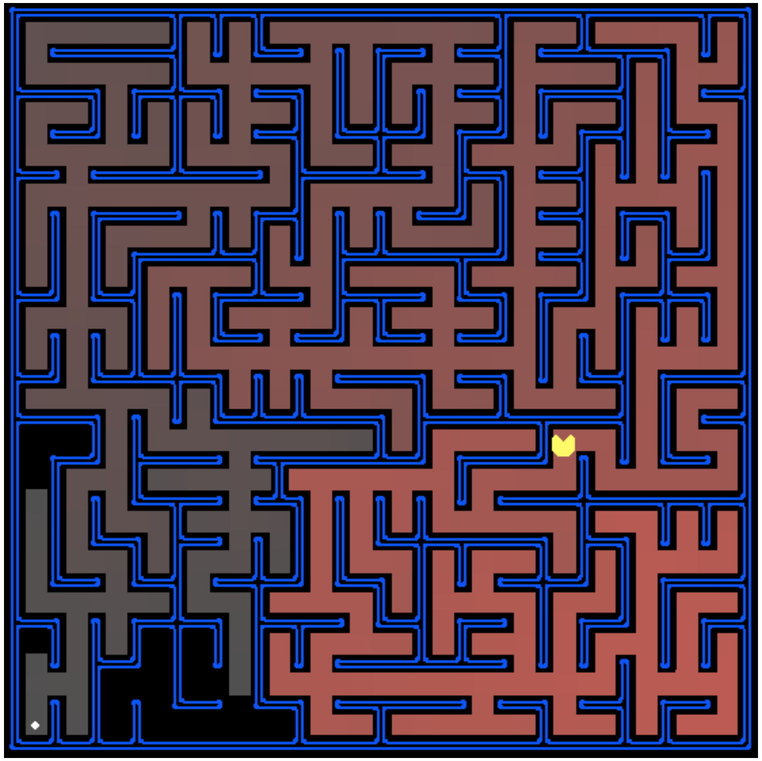
- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so $O(bm)$
- Is it complete?
  - m could be infinite, so yes iff we prevent cycles (more later)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



Project 1: Pacman Search

http://www.mathcs.emory.edu/~eugene/cs425/p1/

Due: Friday February 9th



To Do: Start reading Project 1 code

Files you'll edit:

search.py	Where all of your search algorithms will reside.
searchAgents.py	Where all of your search-based agents will reside.

Files you might want to look at:

pacman.py	The main file that runs Pac-Man games. This file describes a Pac-Man GameState type, which you use in this project.
game.py	The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
util.py	**Useful** data structures for implementing search algorithms