

Introduction to Artificial Intelligence

Search

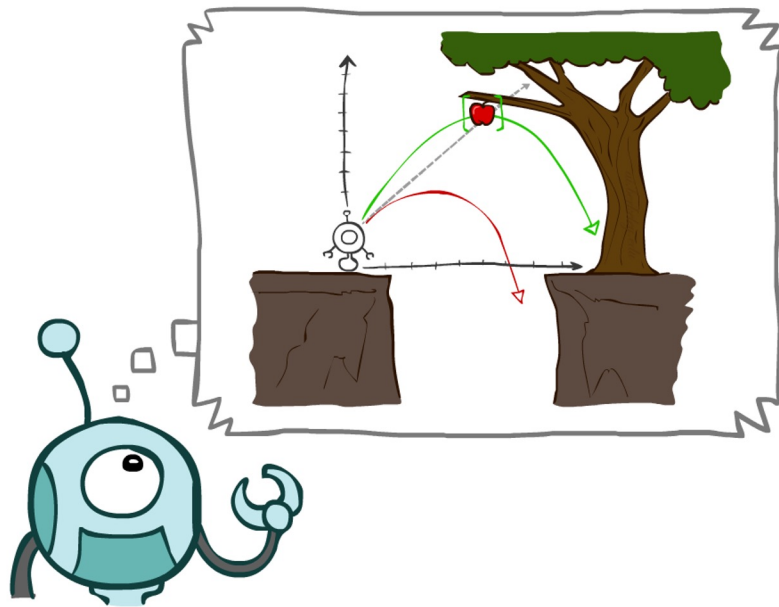


Instructor: Laura Nenzi

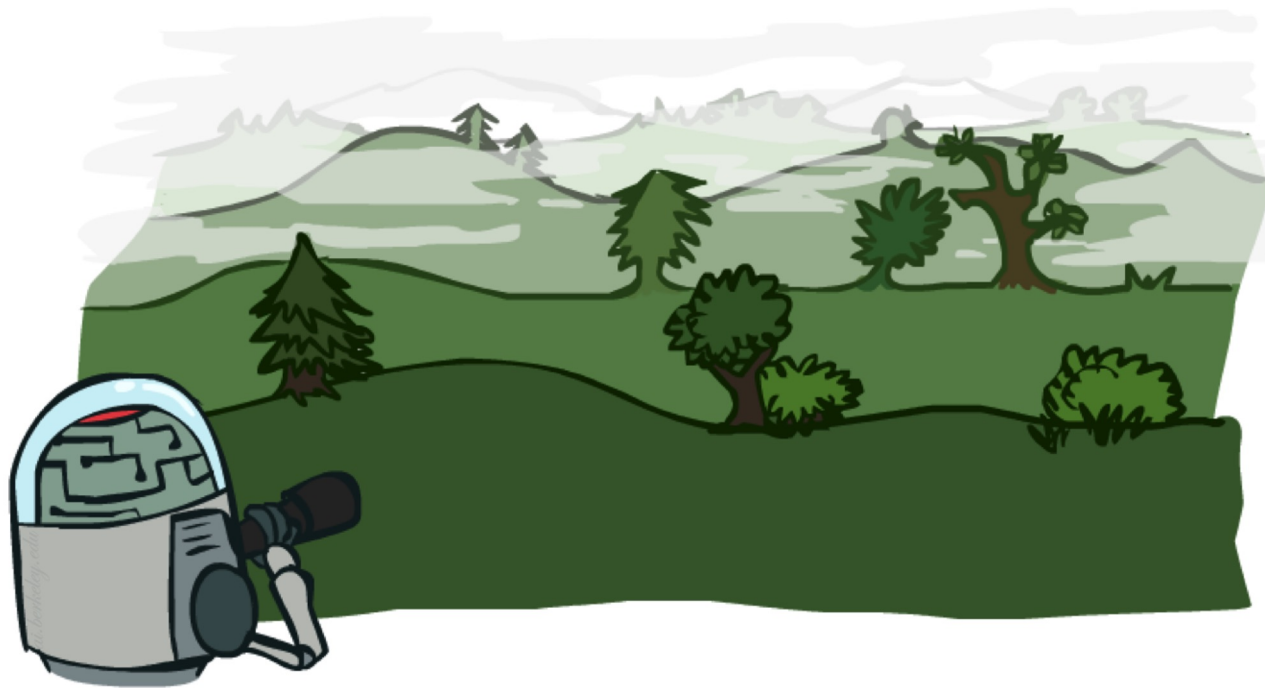
University of Trieste, Italy

Today

- Search Problem
- Uninformed Search Methods
 - Breadth-First Search
 - Depth-First Search
 - Uniform-Cost Search

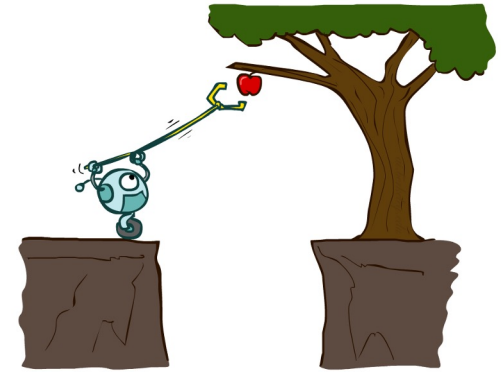


Search Problems

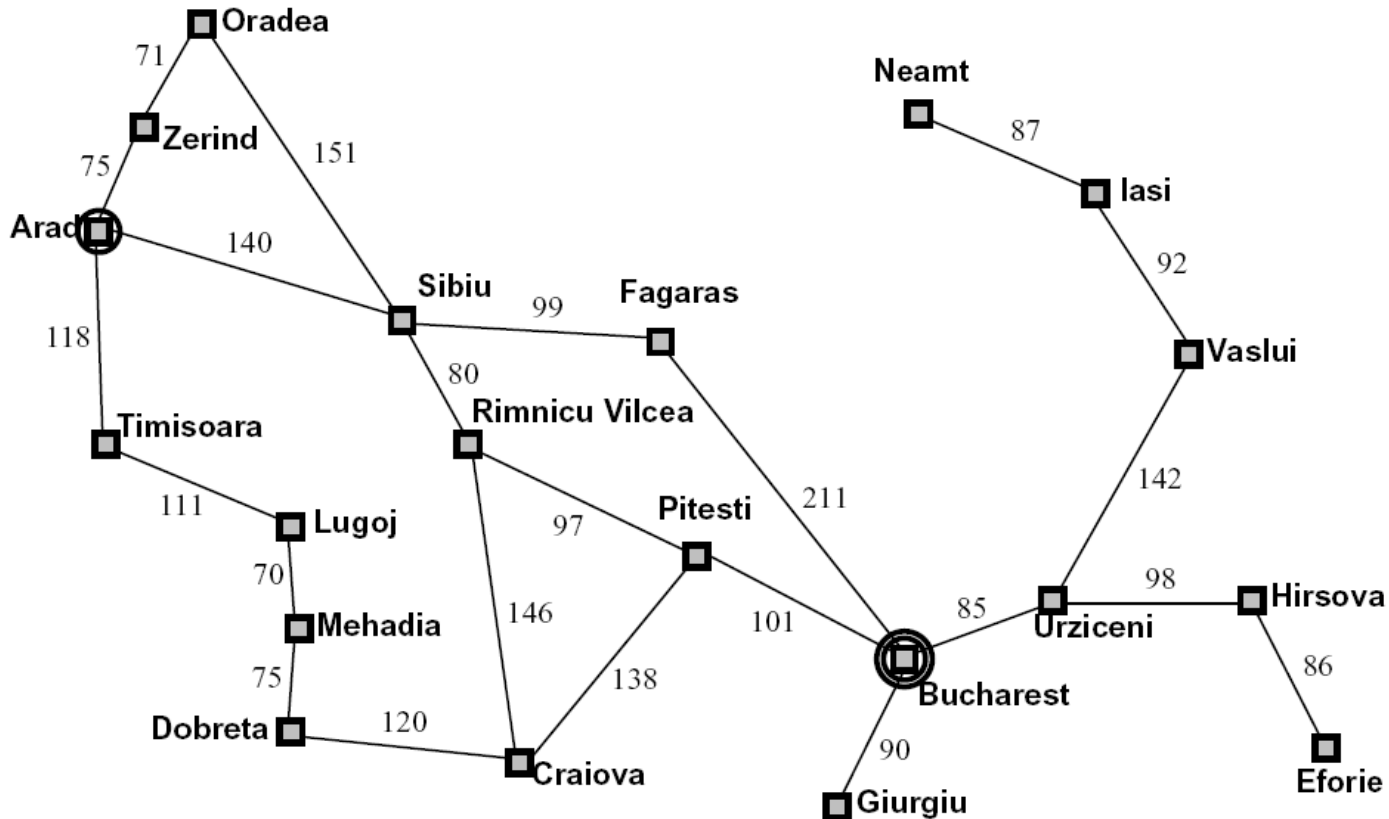


Context

- Agent: goal-based agents with atomic representation
- Environment: episodic, single agent, fully observable, deterministic, static, discrete, and known



Example: Traveling in Romania



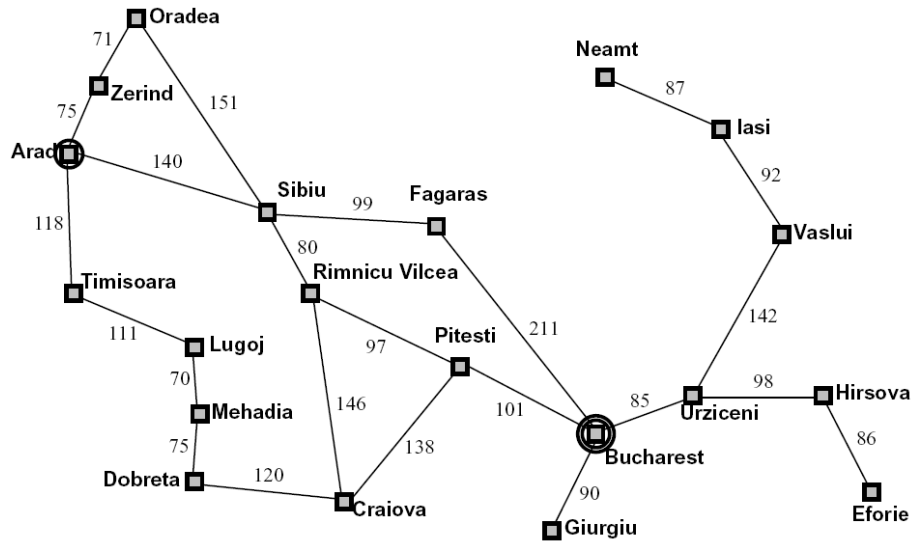
Search Problems Are Models



Search Problems

- A **search problem** consists of:
 - A state space S
 - Actions: $\text{Actions}(s)$
 - A successor(/action cost) function: $c(s, a, s')$ where $a(s) = s'$
 - A initial state and a goal test(/state)
- A **solution** is a path, i.e. a sequence of actions (a plan) which transforms the start state to a goal state
- An **optimal solution** has the lowest path cost among all solutions.

Example: Traveling in Romania

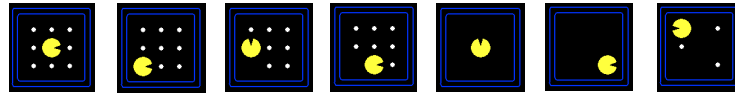


- State space: Cities
- Actions
e.g. $\text{Actions}(\text{Arad}) = \{\text{ToSibiu}, \text{ToTimisoara}, \text{ToZerind}\}$
- Successor function:
 - Roads: Go to adjacent city with cost = distance
e.g. $c(\text{Arad}, \text{ToSibiu}, \text{Sibiu}) = 140$
- Start state:
 - Arad
- Goal state:
 - Bucharest
- Solution?

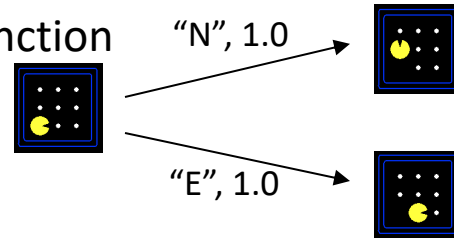
Search Problems

- A **search problem** consists of:

- A state space



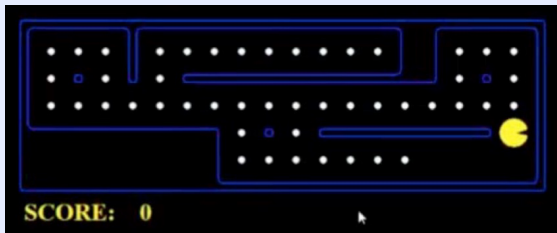
- A successor(/action cost) function
(with actions, costs)



- A initial state and a goal test(/state)
- A **solution** is a path, i.e. a sequence of actions (a plan) which transforms the start state to a goal state

What's in a State Space?

The **world state** includes every last detail of the environment



A **search state** keeps only the details needed for planning (abstraction)

■ Problem: Pathing

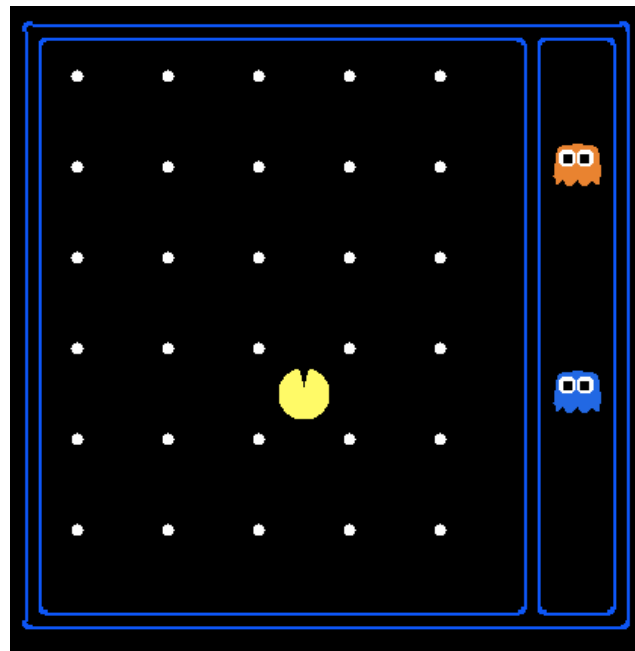
- States: (x,y) location
- Actions: NSEW
- Successor: update location only
- Goal test: is $(x,y)=\text{END}$

■ Problem: Eat-All-Dots

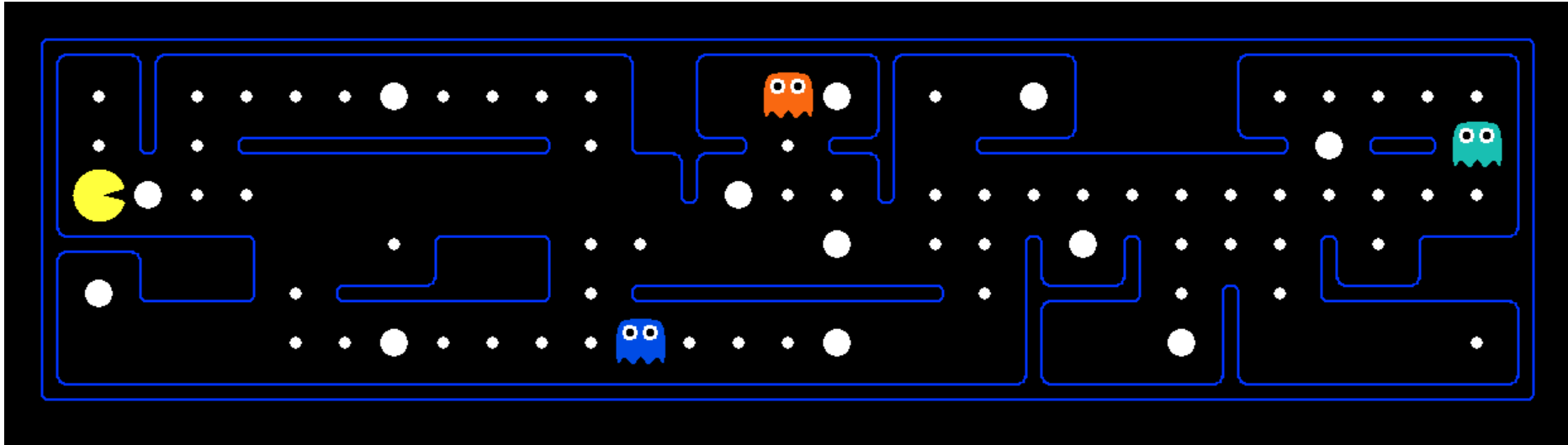
- States: $\{(x,y), \text{dot booleans}\}$
- Actions: NSEW
- Successor: update location and possibly a dot boolean
- Goal test: dots all false

State Space Sizes?

- World state:
 - Agent positions: 120
 - Food count: 30
 - Ghost positions: 12
 - Agent facing: NSEW
- How many
 - World states?
 $120 \times (2^{30} - 1) \times (12^2) \times 4$
 - States for pathing?
120
 - States for eat-all-dots?
 $120 \times (2^{30})$

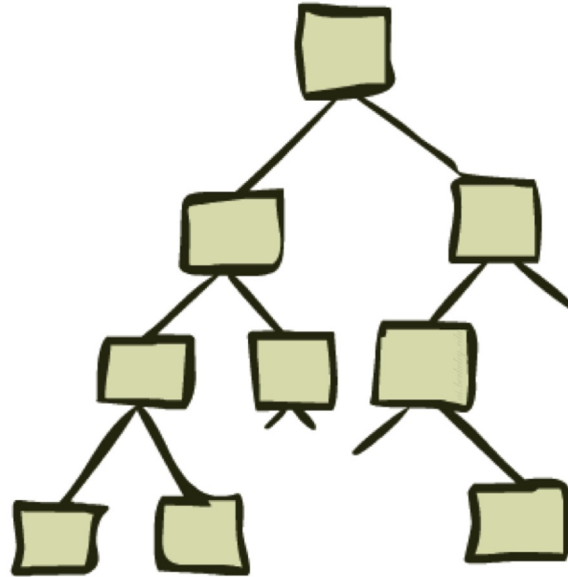


Quiz: Safe Passage



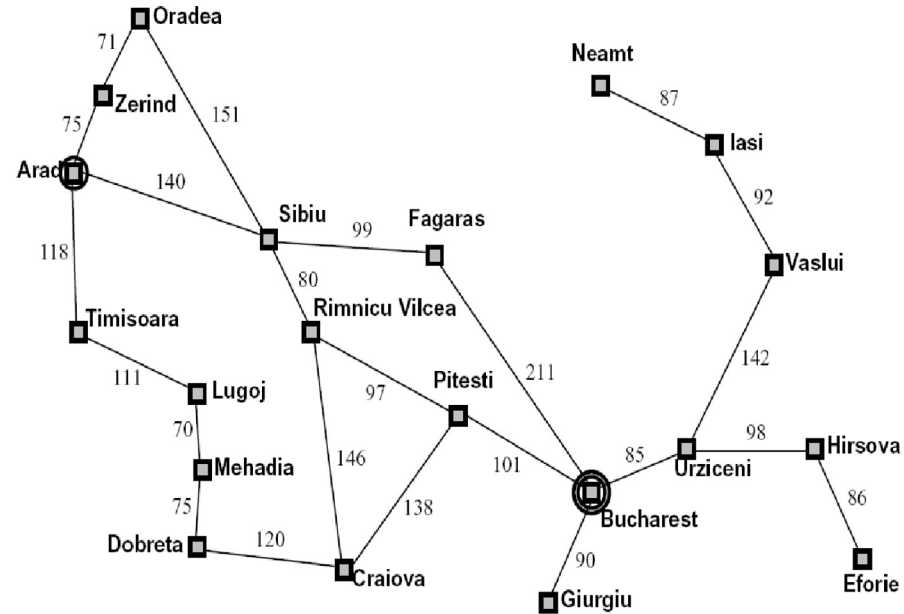
- Problem: eat all dots while keeping the ghosts perma-scared
- What does the state space have to specify?
 - (agent position, dot booleans, power pellet booleans, remaining scared time, ghosts location)

State Space Graphs and Search Trees



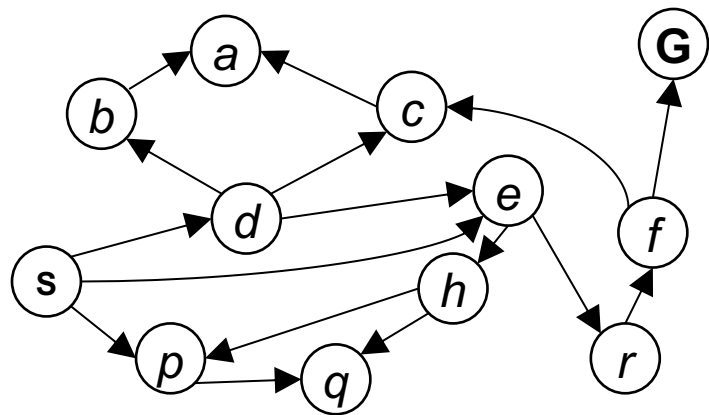
State Space Graphs

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



State Space Graphs

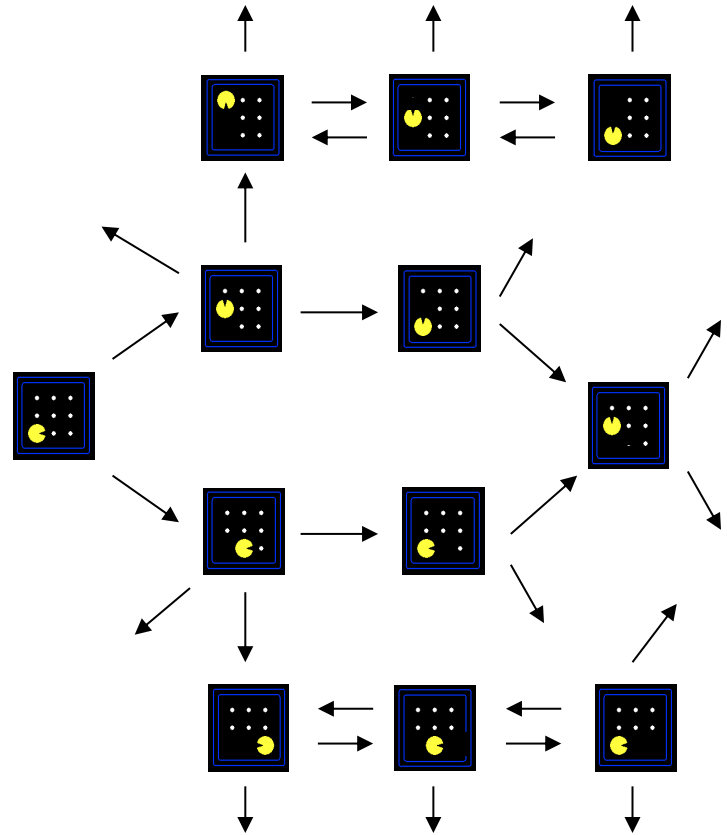
- State space graph: A mathematical representation of a search problem
 - Nodes are (abstracted) world configurations
 - Arcs represent successors (action results)
 - The goal test is a set of goal nodes (maybe only one)



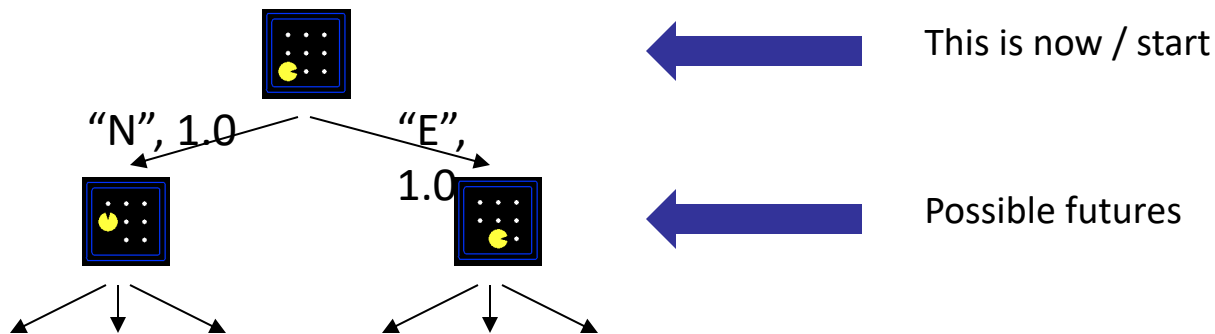
*Tiny state space graph
for a tiny search problem*

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



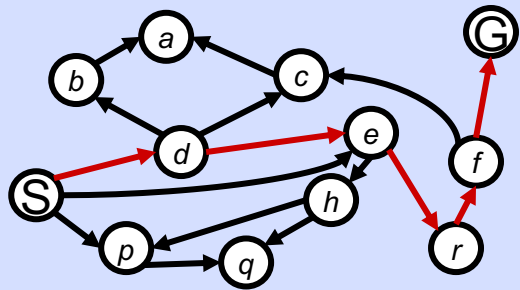
Search Trees



- 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

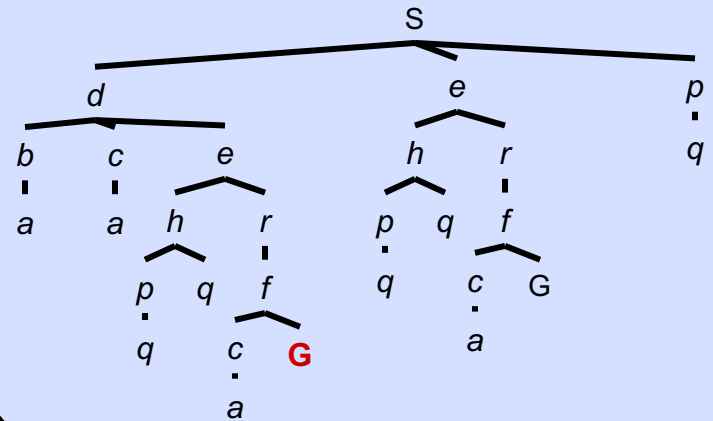
State Space Graph



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

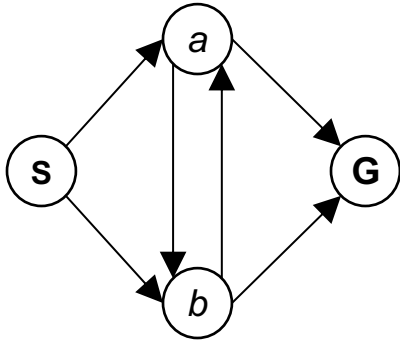
We construct them on demand – and we construct as little as possible.

Search Tree

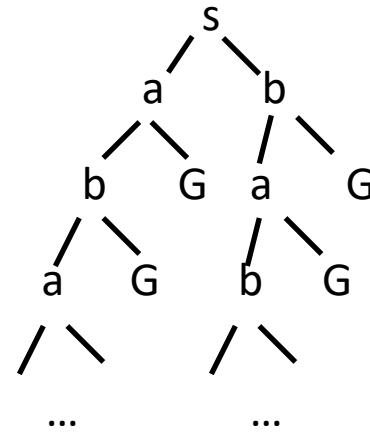


Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:



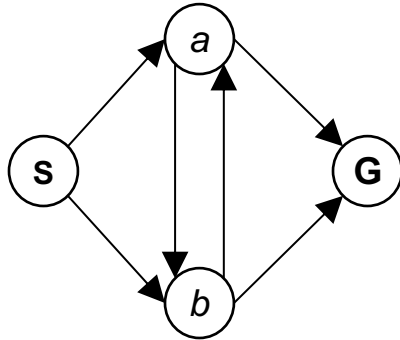
How big is its search tree (from S)?



Important: Lots of repeated structure in the search tree!

Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:

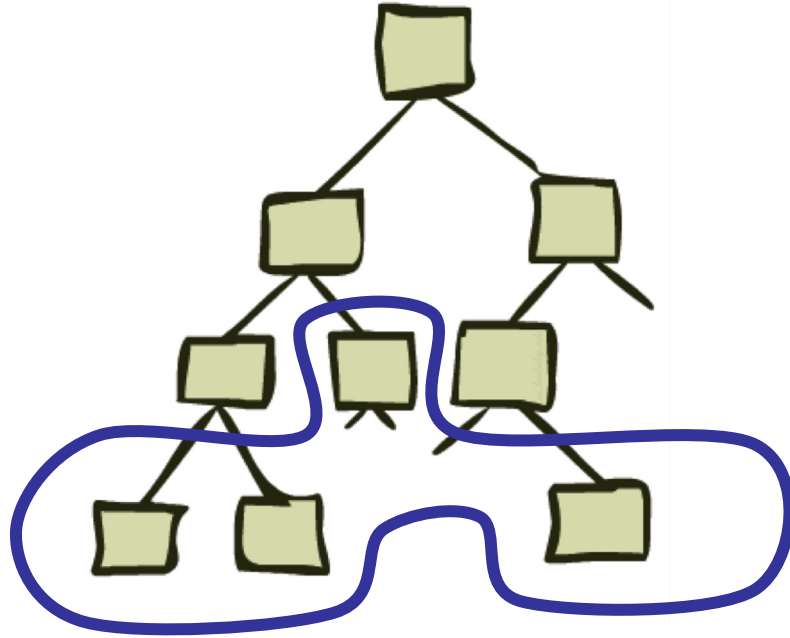


How big is its search tree (from S)?

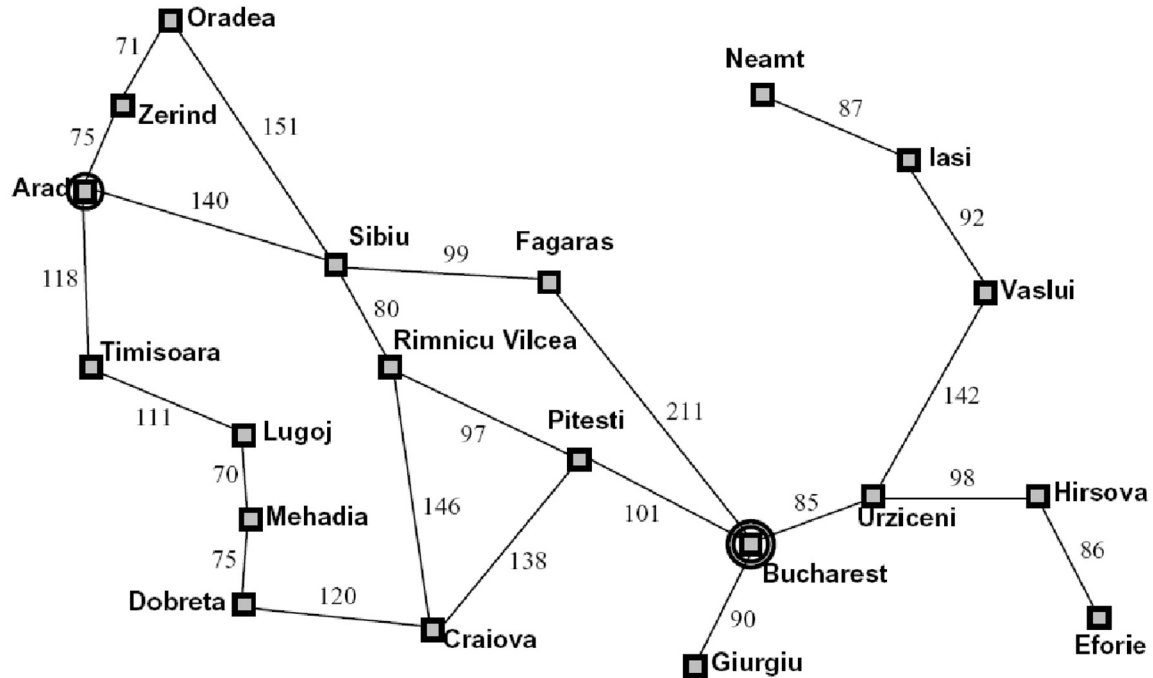


Important: Lots of repeated structure in the search tree!

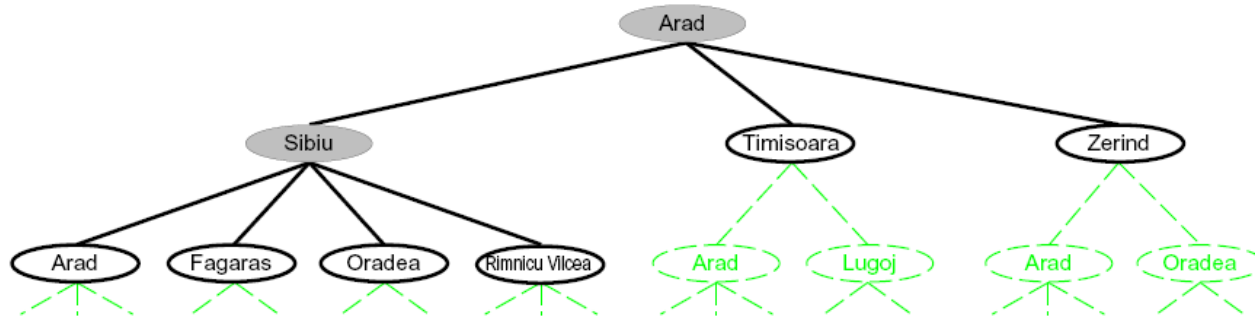
Tree Search



Search Example: Romania



Searching with a Search Tree



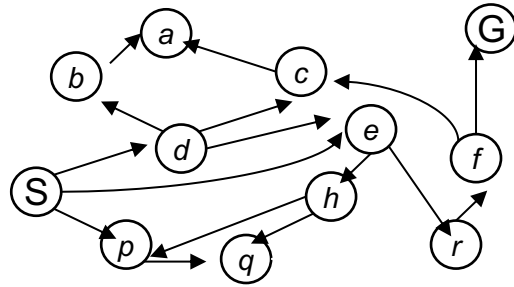
- Search:
 - Expand out potential plans (tree nodes)
 - Maintain a **fringe** of partial plans under consideration
 - Try to expand as few tree nodes as possible

General Tree Search

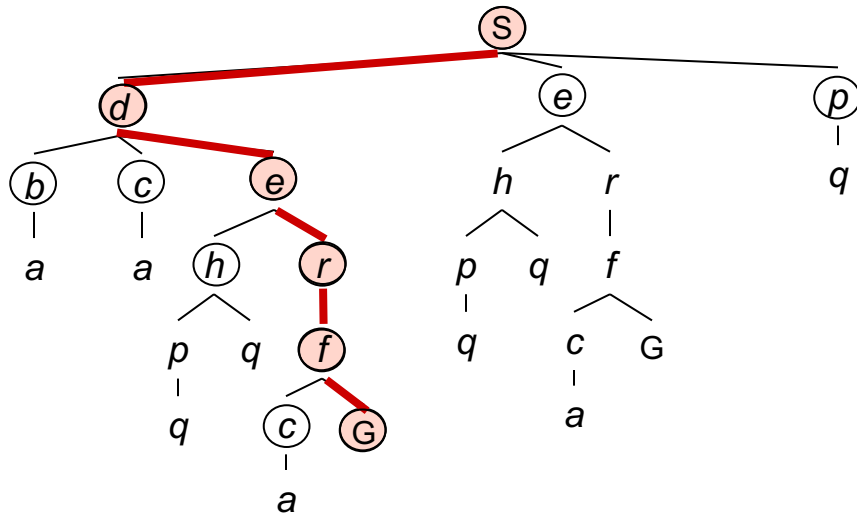
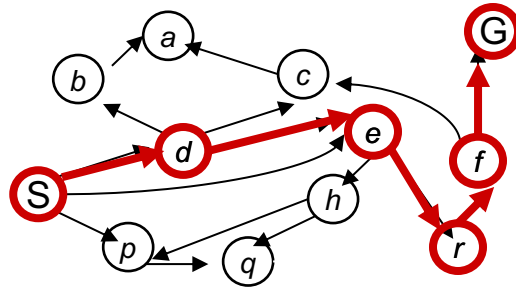
```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal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

Example: Tree Search



Example: Tree Search



- ~~s~~
- ~~s → d~~
- s → e
- s → p
- s → d → b
- s → d → c
- ~~s → d → e~~
- s → d → e → h
- ~~s → d → e → r~~
- ~~s → d → e → r → f~~
- s → d → e → r → f → c
- ~~s → d → e → r → f → G~~

Best-first search

Evaluation function

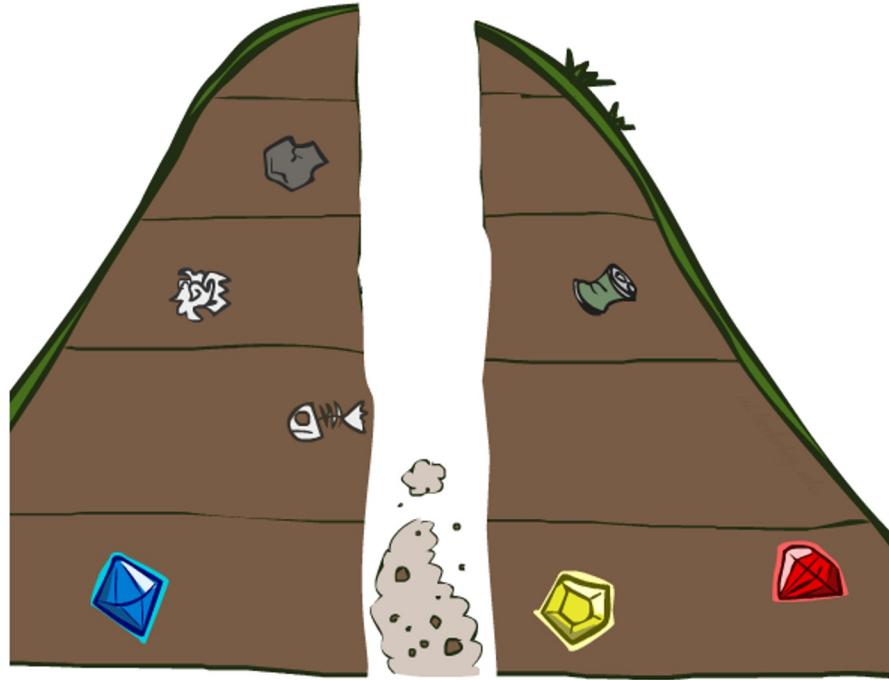
function BEST-FIRST-SEARCH(*problem*, *f*) **returns** a solution node or *failure*
node ← NODE(STATE=*problem*.INITIAL)
frontier ← a priority queue ordered by *f*, with *node* as an element
reached ← a lookup table, with one entry with key *problem*.INITIAL and value *node*
while not IS-EMPTY(*frontier*) **do**
 node ← POP(*frontier*)
 if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
 for each *child* **in** EXPAND(*problem*, *node*) **do**
 s ← *child*.STATE
 if *s* is not in *reached* **or** *child*.PATH-COST < *reached*[*s*].PATH-COST **then**
 reached[*s*] ← *child*
 add *child* to *frontier*
return *failure*

function EXPAND(*problem*, *node*) **yields** nodes
s ← *node*.STATE
for each *action* **in** *problem*.ACTIONS(*s*) **do**
 s' ← *problem*.RESULT(*s*, *action*)
 cost ← *node*.PATH-COST + *problem*.ACTION-COST(*s*, *action*, *s'*)
 yield NODE(STATE=*s'*, PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

Data Structure to store the frontier

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, f (used in best-first search)
- A **FIFO queue** or **first-in-first-out queue** first pops the node that was added to the queue first (used in breadth-first search)
- A **LIFO queue** or **last-in-first-out queue** (also known as a stack) pops first the most recently added node (used in depth-first search)

Search Algorithm Properties

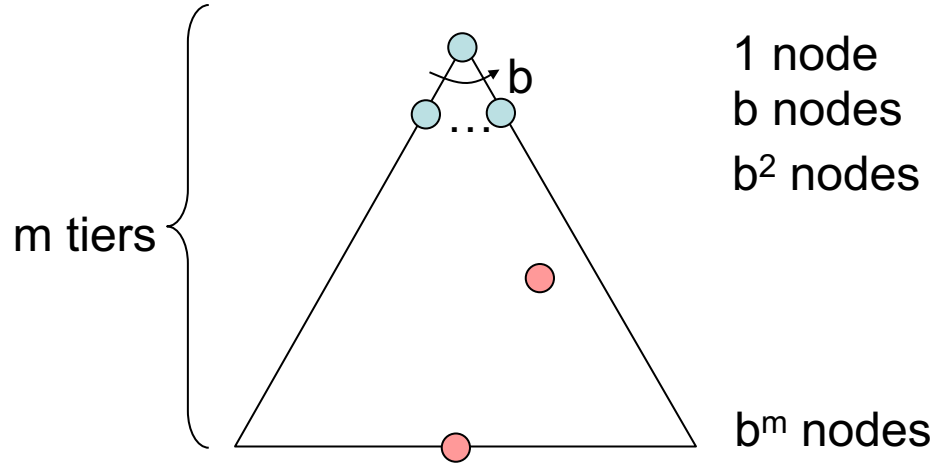


Search Algorithm Properties

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

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

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



Measuring problem-solving performance

- **Completeness:** Is the algorithm guaranteed to find a solution when there is one, and to report failure when there is not?
- **Cost optimality:** Does it find a solution with the lowest path cost of all solutions?
- **Time complexity:** How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.
- **Space complexity:** How much memory is needed to perform the search?