CS 343H: Artificial Intelligence

Lecture 3: Uninformed Search

Tues 1/21/14

Slides courtesy of Dan Klein at UC-Berkeley
Unless otherwise noted

Announcements

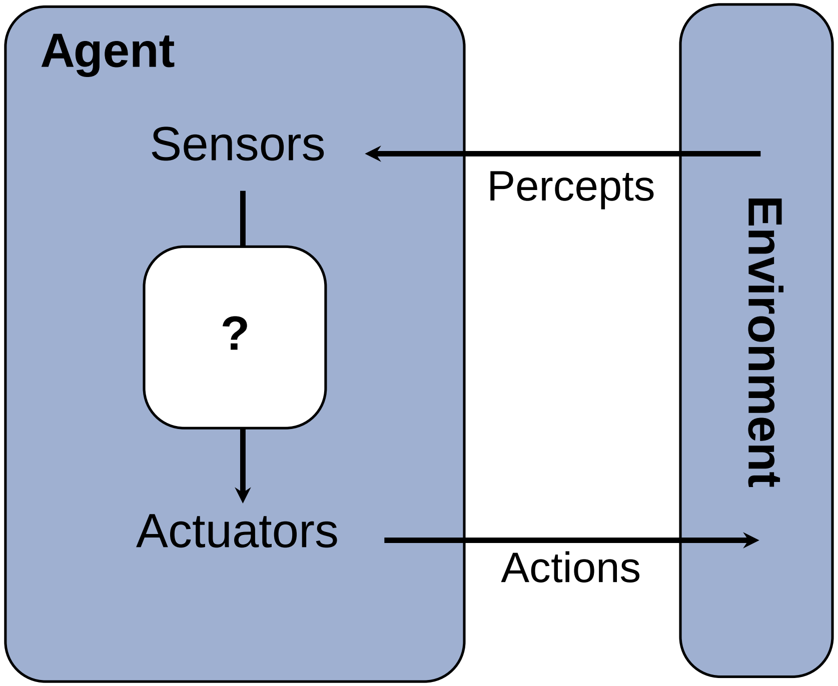
- PS0 due this Thurs 1/23 by 11:59 pm
- All remaining reading response deadlines are firm
- Printing slides before lecture

Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search

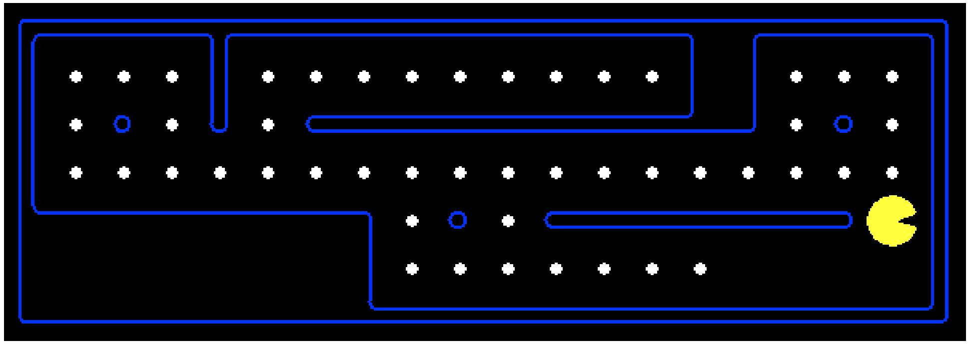
Recall: Rational Agents

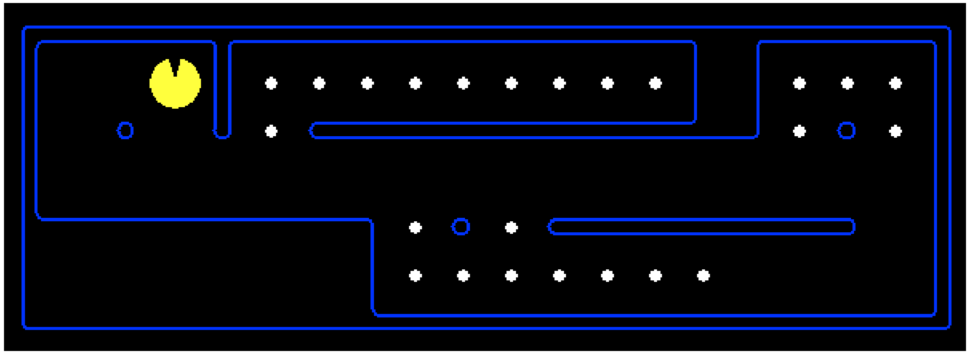
- An **agent** is an entity that *perceives* and *acts*.
- A **rational agent** selects actions that maximize its **utility function**.
- Characteristics of the **percepts, environment,** and **action space** dictate techniques for selecting rational actions.



Reflex Agents

- 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?

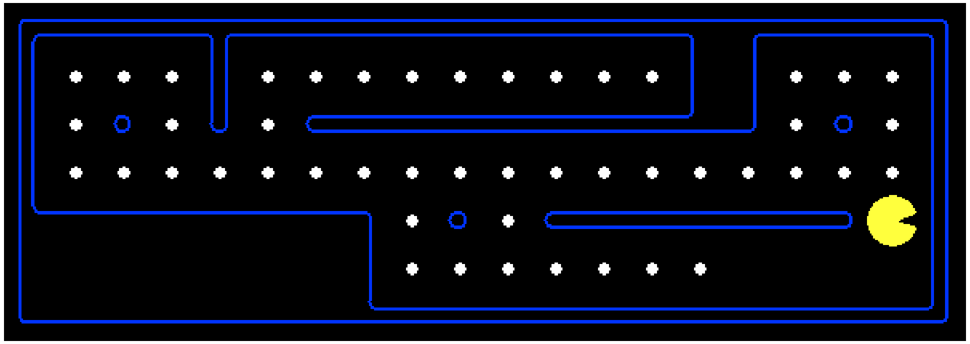


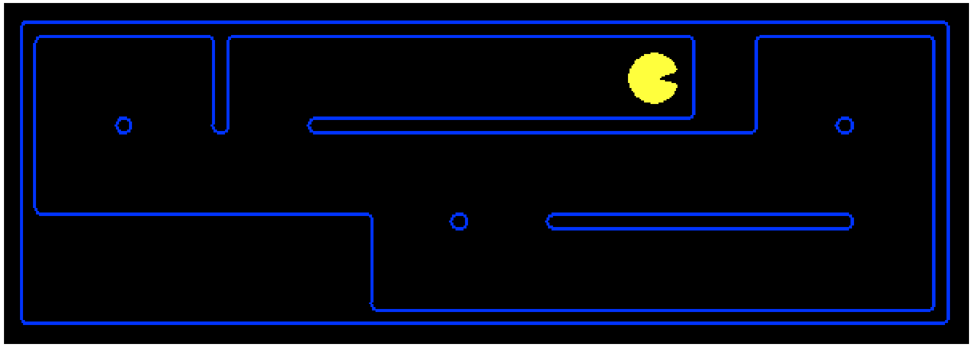


[demo: reflex optimal / loop]

Planning Agents

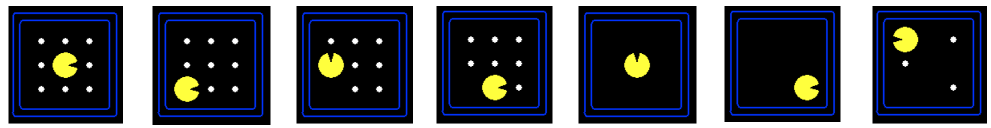
- Plan ahead
- Ask “what if”
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions
- Consider how the world WOULD BE



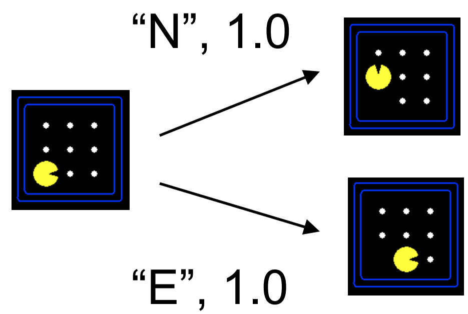


Search Problems

- A search problem consists of:
  - A state space

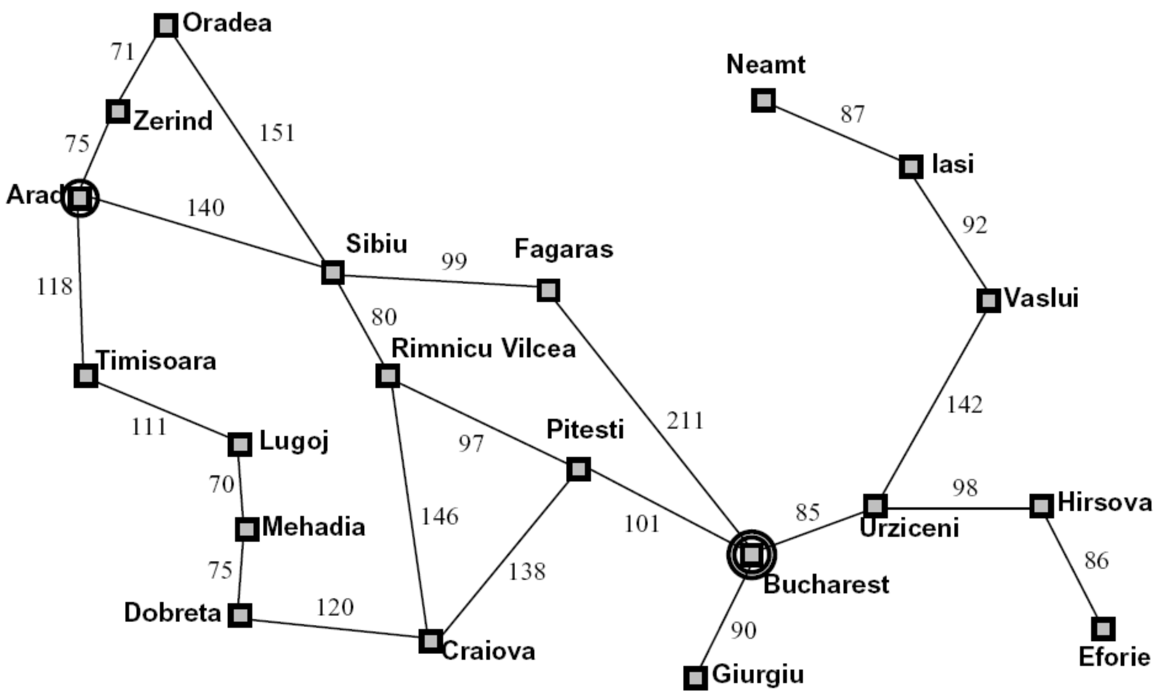


  - A successor function (with actions, costs)



  - 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

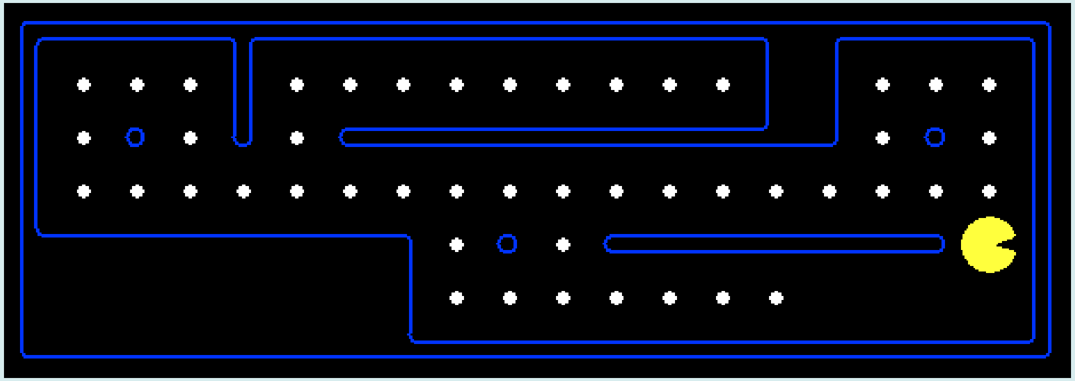
Example: Romania



- State space:
  - Cities
- Successor function:
  - Roads: Go to adj city with cost = dist
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?
- Solution?

What's in a State Space?

The world state specifies every last detail of the environment



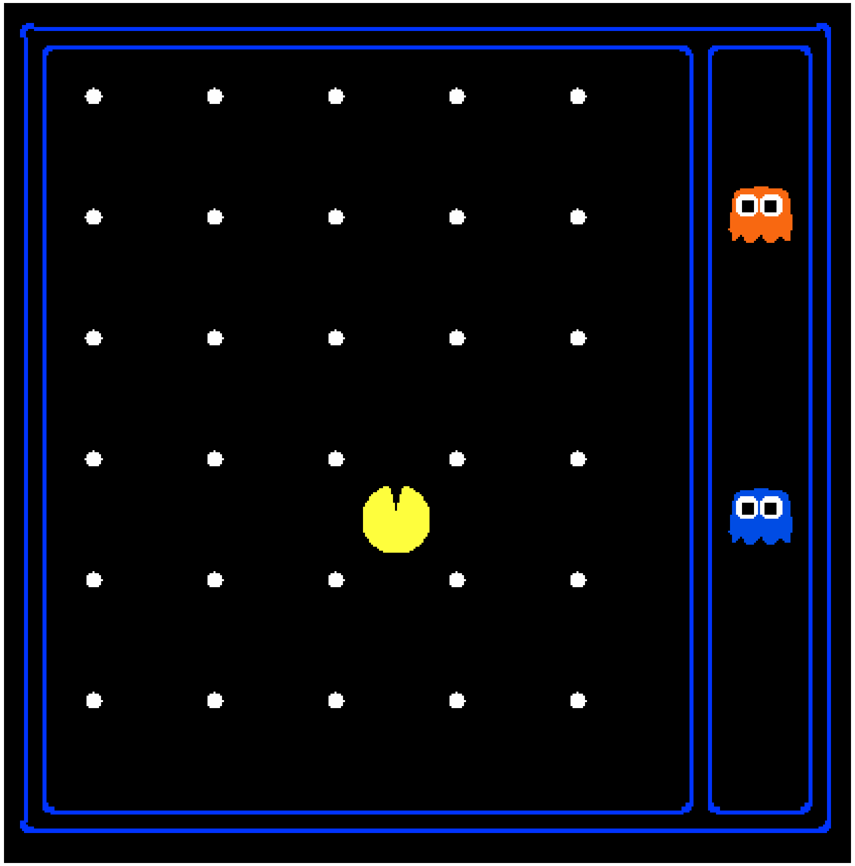
A search state keeps only the details needed (abstraction)

- Problem 1: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

- Problem 2: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

State Space Sizes?

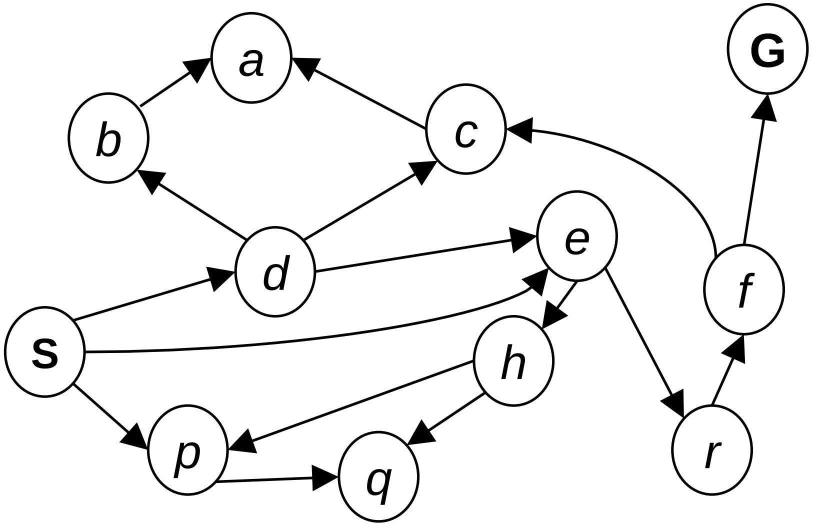
- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW



- How many
  - World states?
    $120x(2^{30})x(12^2)x4$
  - States for pathing?
    120
  - States for eat-all-dots?
    $120x(2^{30})$

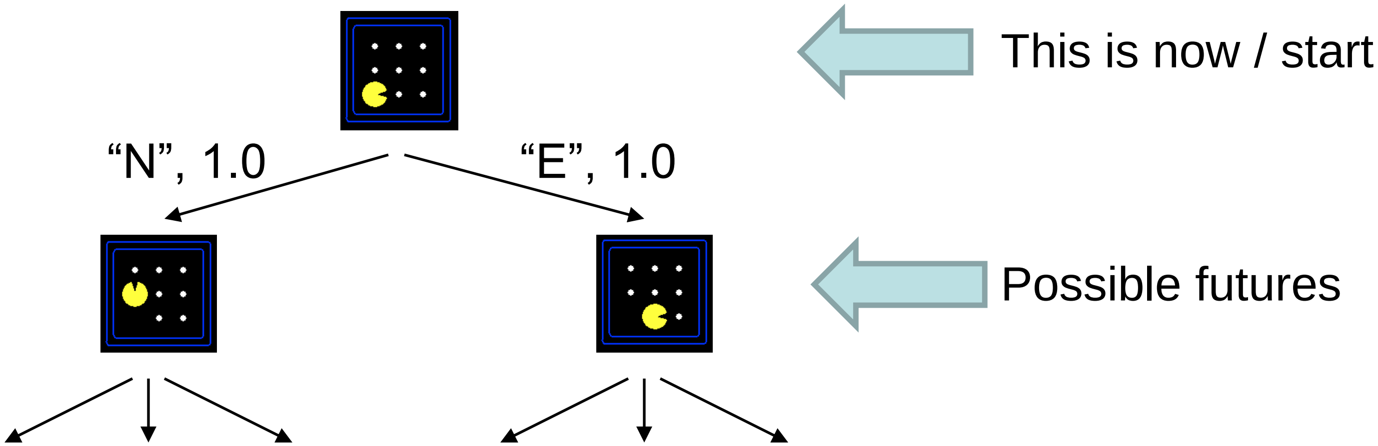
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes: abstracted world configurations
  - Arcs: successors (action results)
  - Goal test is set of goal nodes (maybe only one)
- In a search graph, each state occurs only once!
- We can rarely build this graph in memory (so we don't)



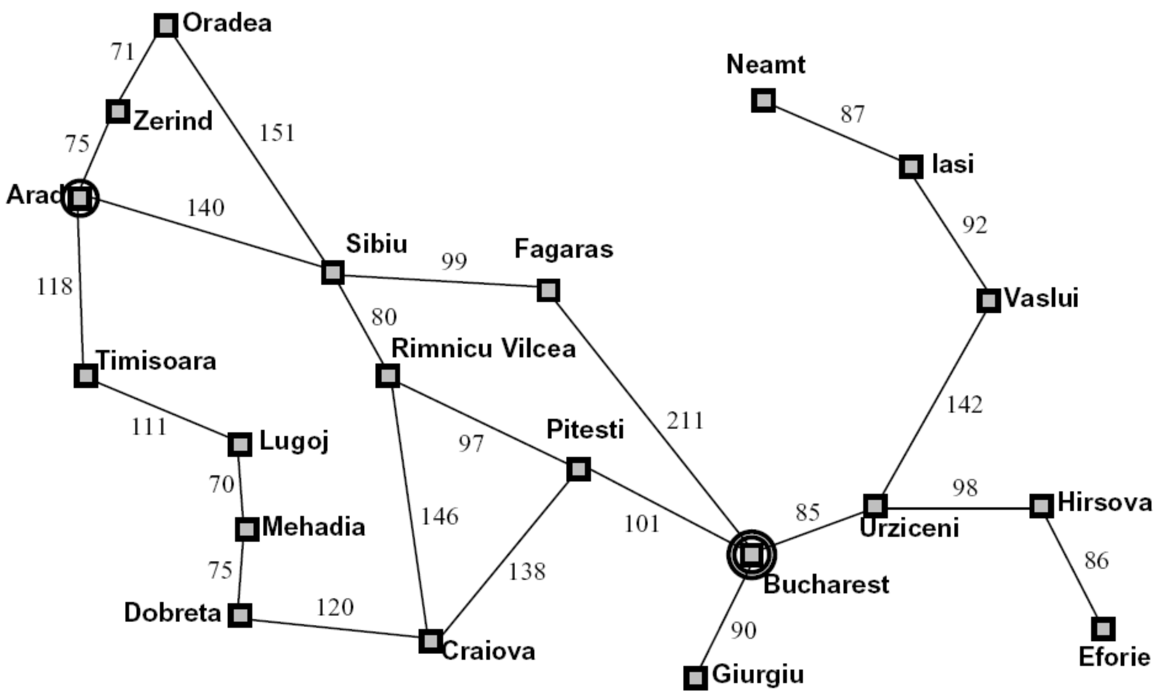
Ridiculously tiny search graph for a tiny search problem

Search Trees

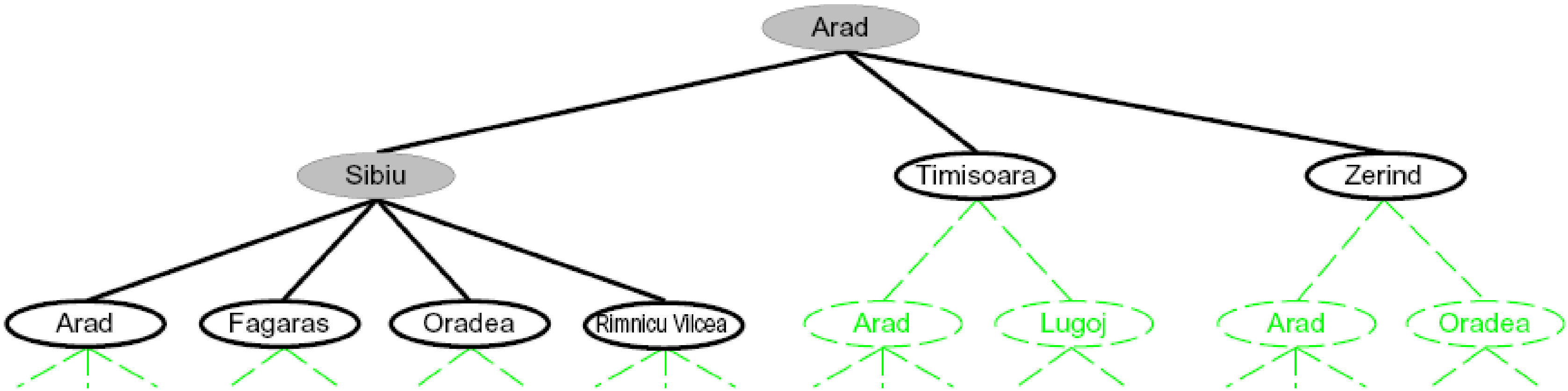


- A search tree:
  - This is a “what if” tree of plans and outcomes
  - Start state at the root node
  - Children correspond to successors
  - Nodes contain states, correspond to PLANS to those states
  - For most problems, we can never actually build the whole tree

Recall: Romania example



Searching with a search tree



- Search:
  - Expand out possible plans
  - Maintain a fringe of unexpanded plans
  - Try to expand as few tree nodes as possible

General Tree Search

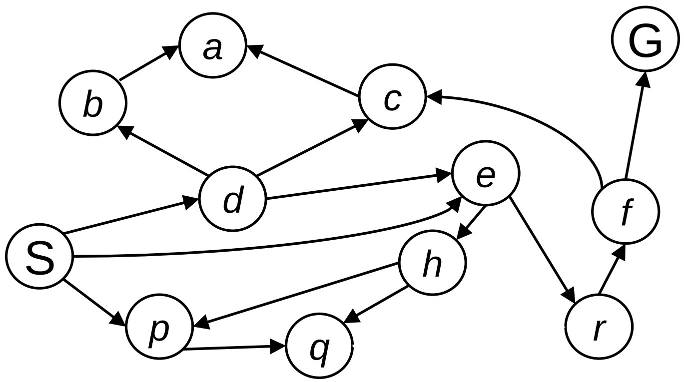
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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

Detailed pseudocode is in the book!

- Main question: which fringe nodes to explore?

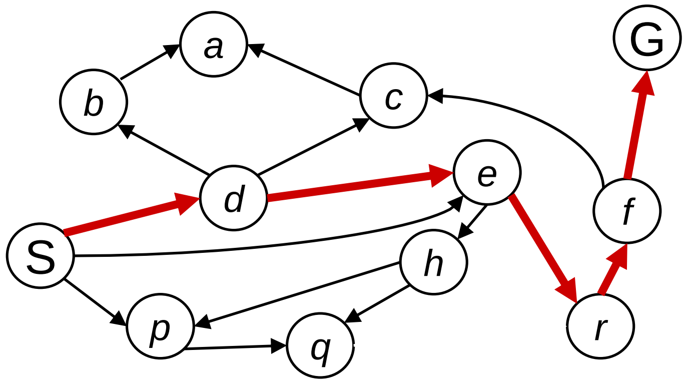
Example: Tree Search



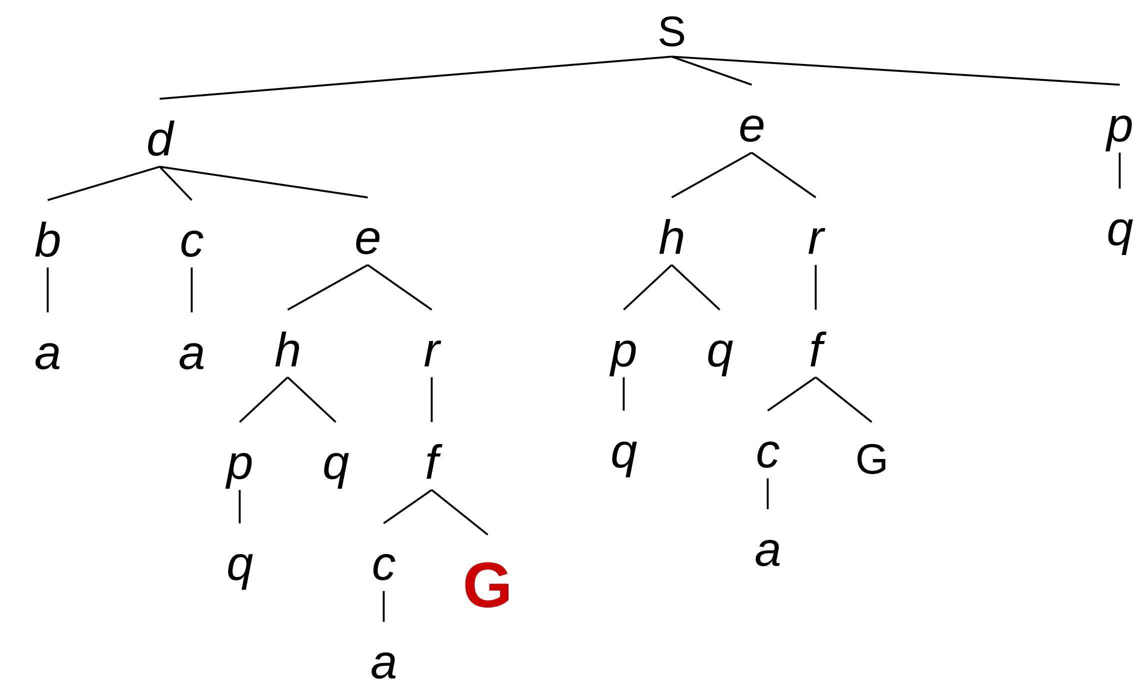
Fringe (potential plans)

Tree

State Graphs vs. Search Trees



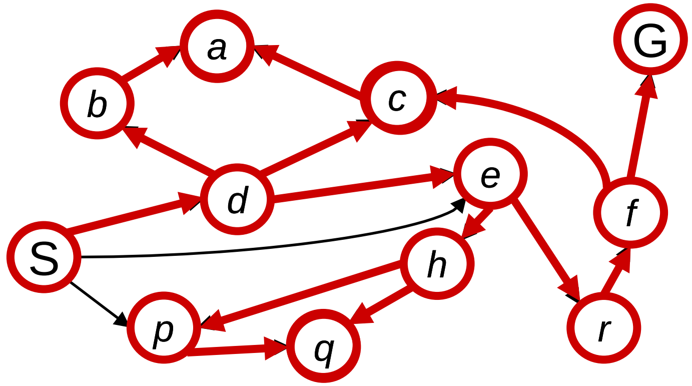
Each NODE in in the search tree is an entire PATH in the problem graph.



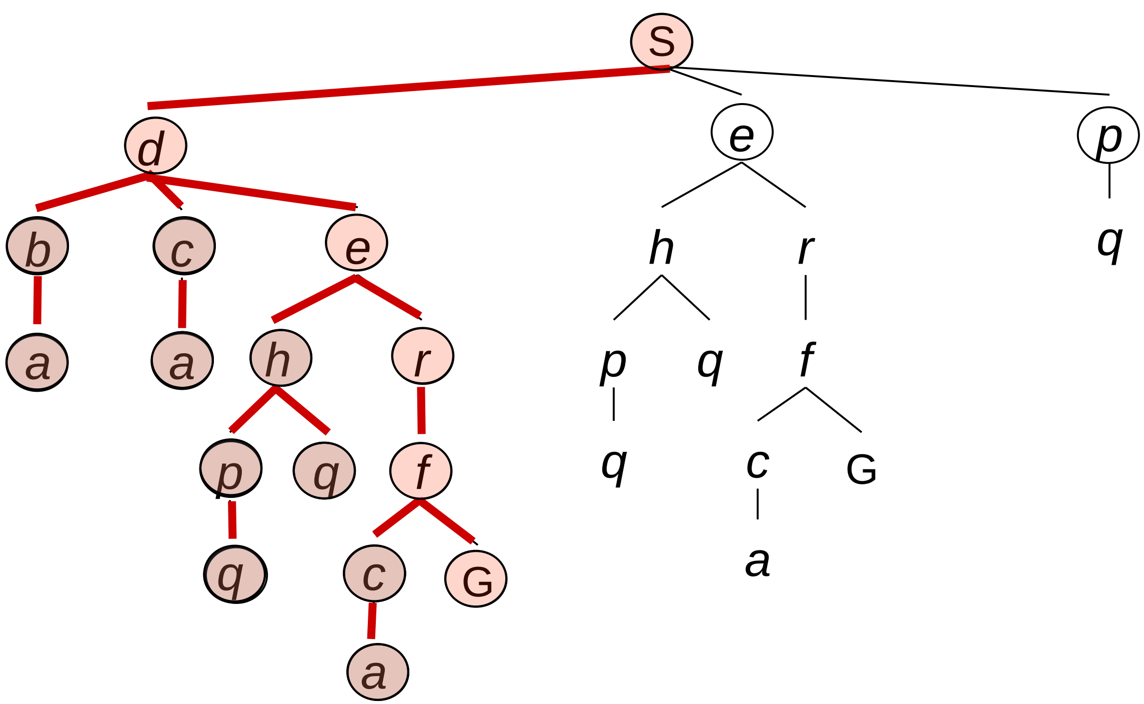
We construct both on demand – and we construct as little as possible.

Depth First Search

Strategy: expand deepest node first



State graph



Search tree

Search Algorithm Properties

Complete? Guaranteed to find a solution if one exists?

Optimal? Guaranteed to find the least cost path?

Time complexity?

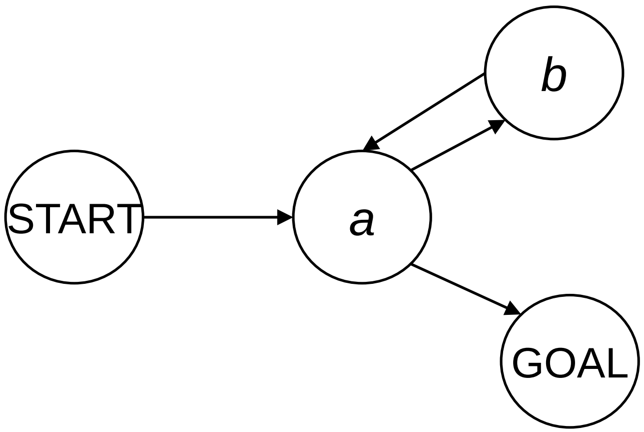
Space complexity?

Variables:

n	Number of states in the problem (huge)
b	The average branching factor *B* (the average number of successors)
$C*$	Cost of least cost solution
s	Depth of the shallowest solution
m	Max depth of the search tree

DFS

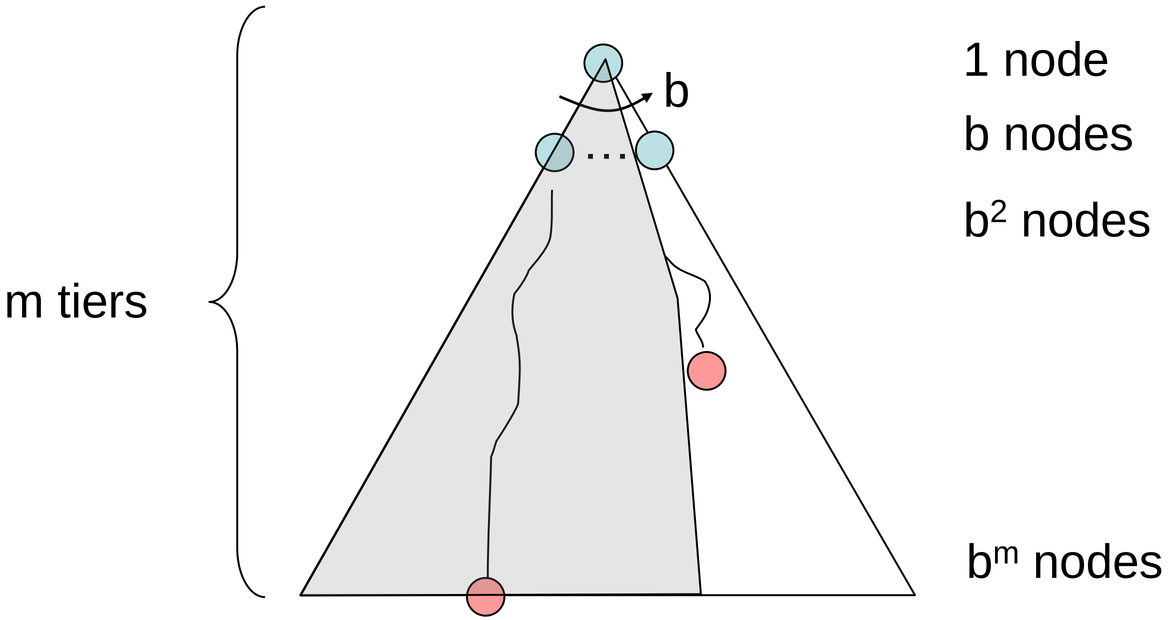
Algorithm		Complete	Optimal	Time	Space
DFS	Depth First Search				



- Infinite paths make DFS incomplete…
- How can we fix this?

DFS

- With cycle checking, DFS is complete.*



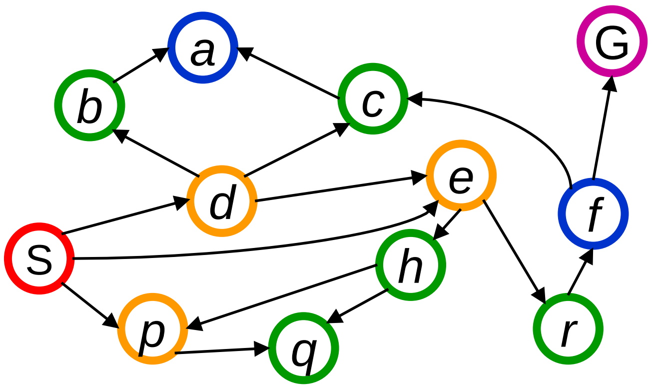
Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking				

- When is DFS optimal?

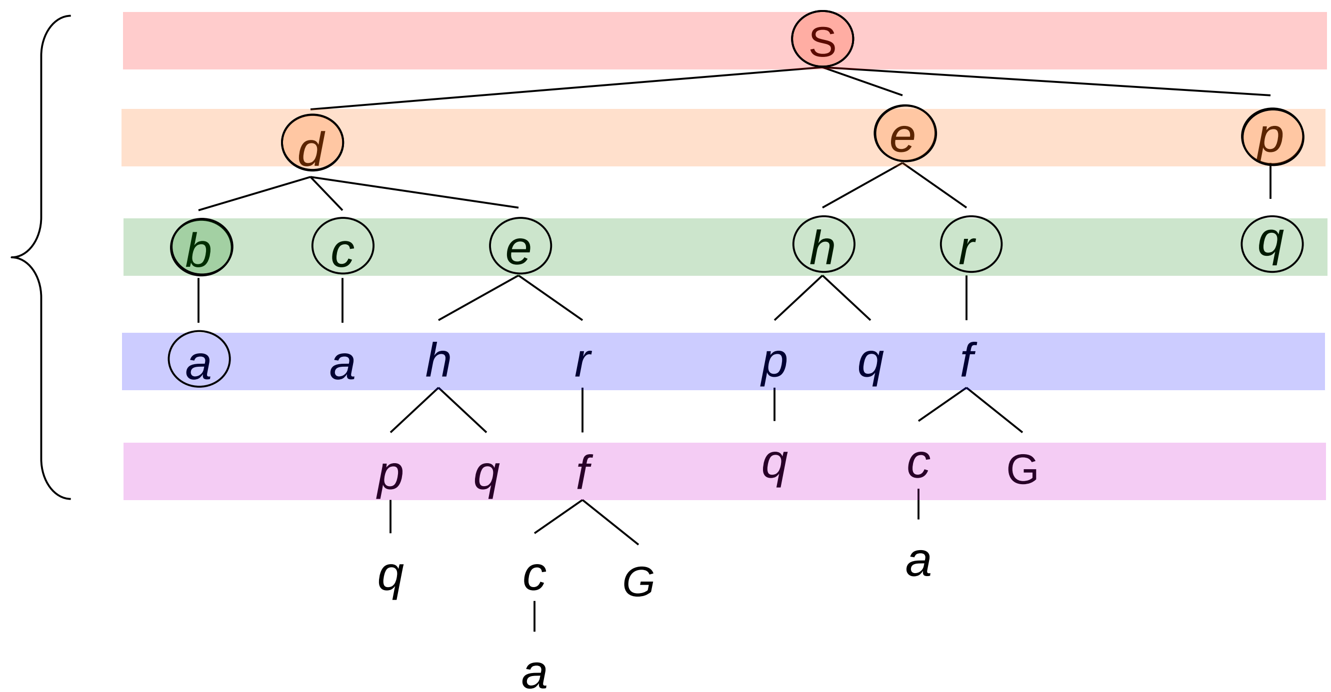
* Or graph search – next lecture.

Breadth First Search

Strategy: expand shallowest node first

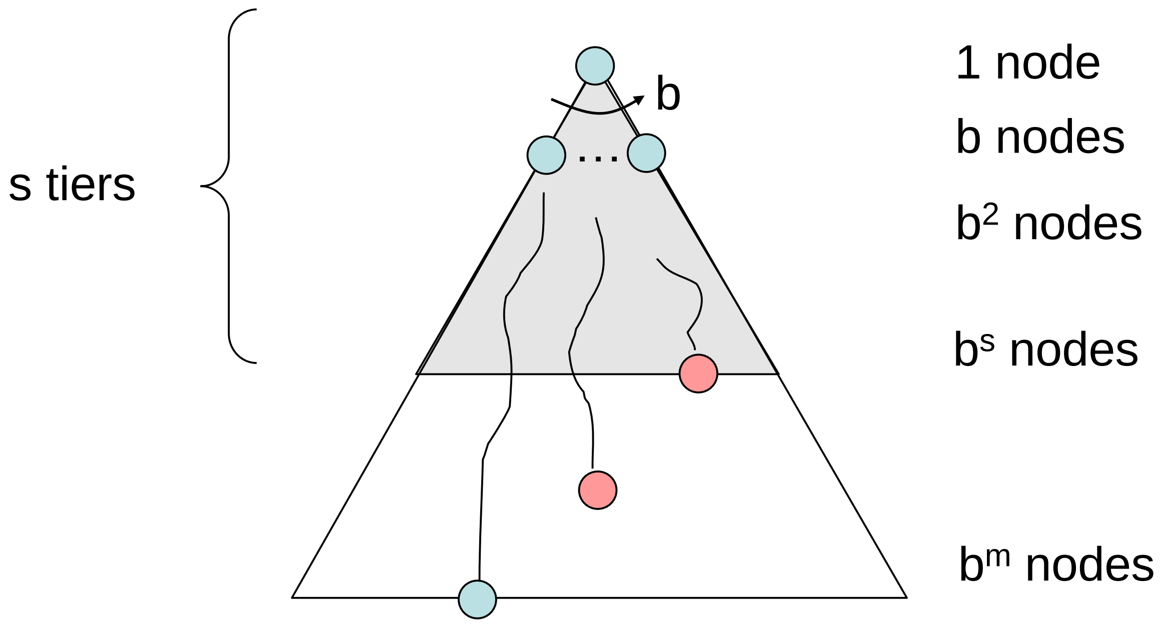


Search Tiers



BFS

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking				
BFS					



- When is BFS optimal?

BFS complexity: concretely

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

- Russell & Norvig

Comparisons

- When will BFS outperform DFS?

- When will DFS outperform BFS?

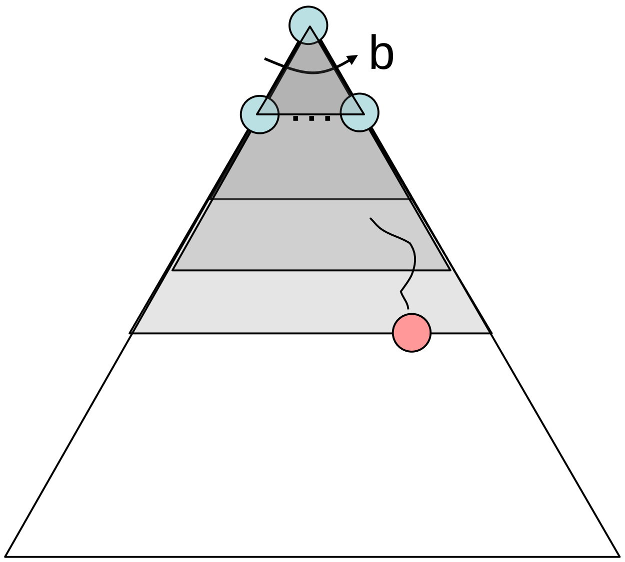
[demo: dfs/bfs]

Iterative Deepening

Iterative deepening: BFS using DFS as a subroutine:

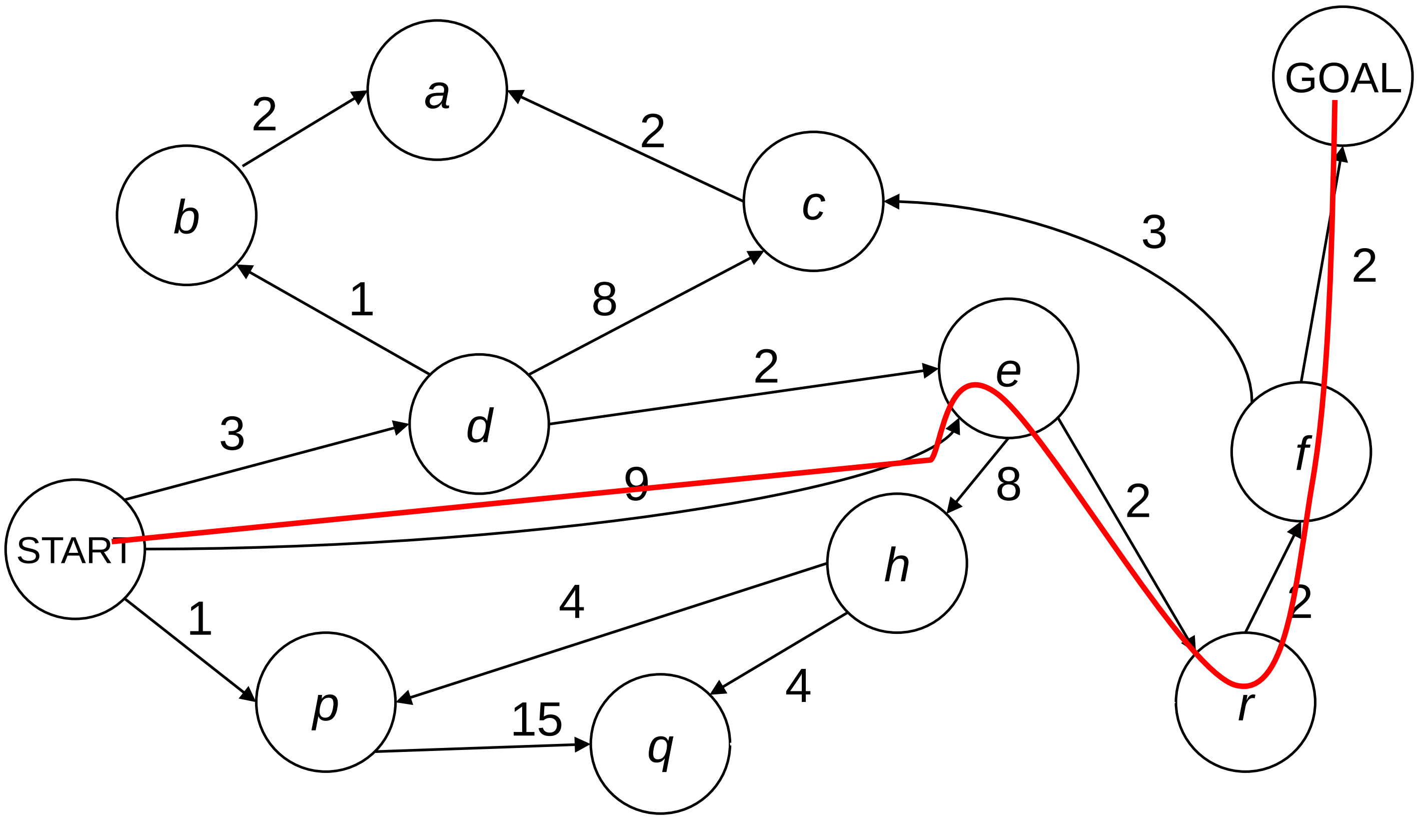
1. Do a DFS which only searches for paths of length 1 or less.
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

….and so on.



Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking				
BFS					
ID					

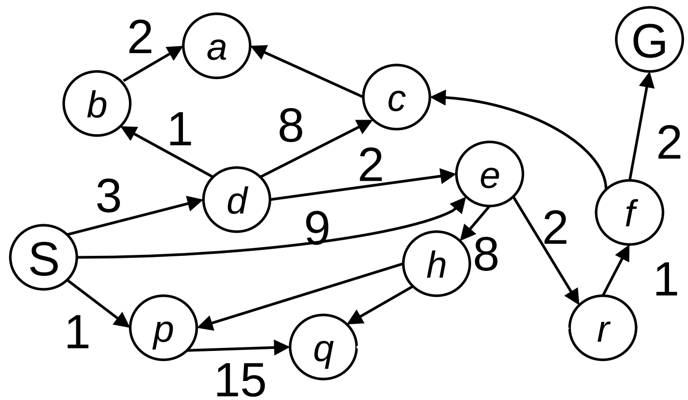
Costs on Actions

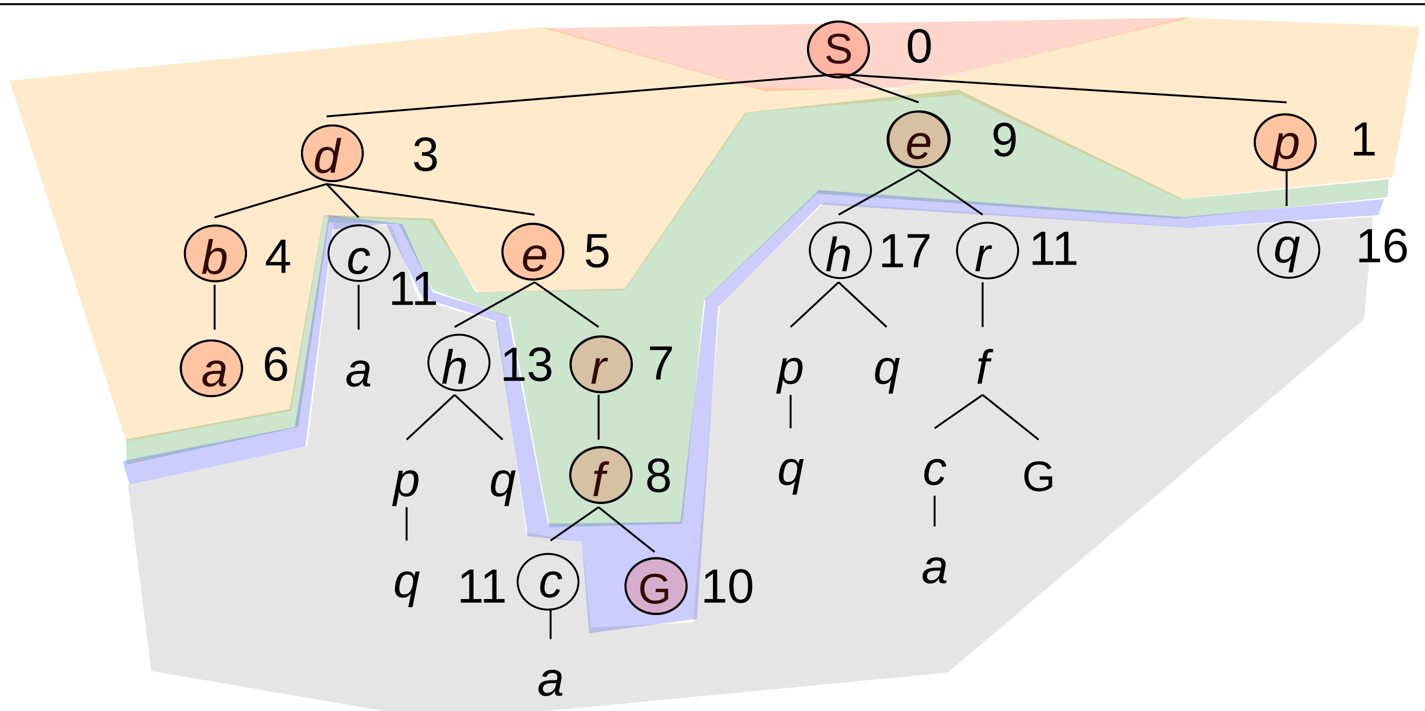


Notice that BFS finds the shortest path in terms of <u>number of transitions</u>. It does not find the least-cost path.

Uniform Cost Search

Expand cheapest node first:





Cost contours

Priority Queue Refresher

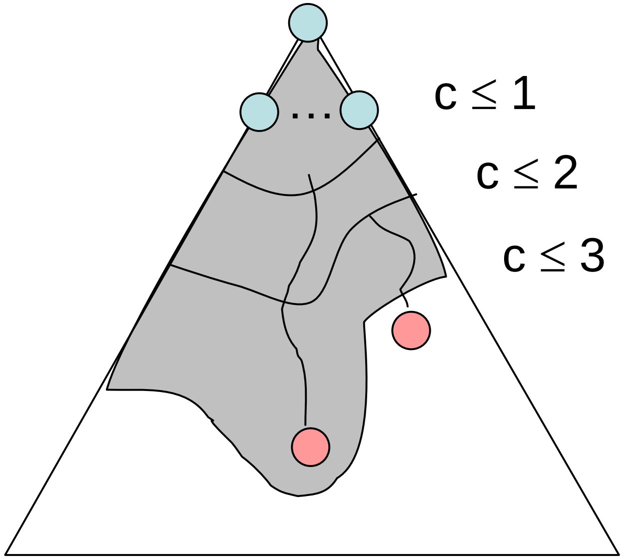
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

pq.push(key, value)	inserts *(key, value)* into the queue.
pq.pop()	returns the key with the lowest value, and removes it from the queue.

- You can decrease a key's priority by pushing it again
- Unlike a regular queue, insertions aren't constant time, usually O(log *n*)
- We'll need priority queues for cost-sensitive search methods

Uniform Cost Search

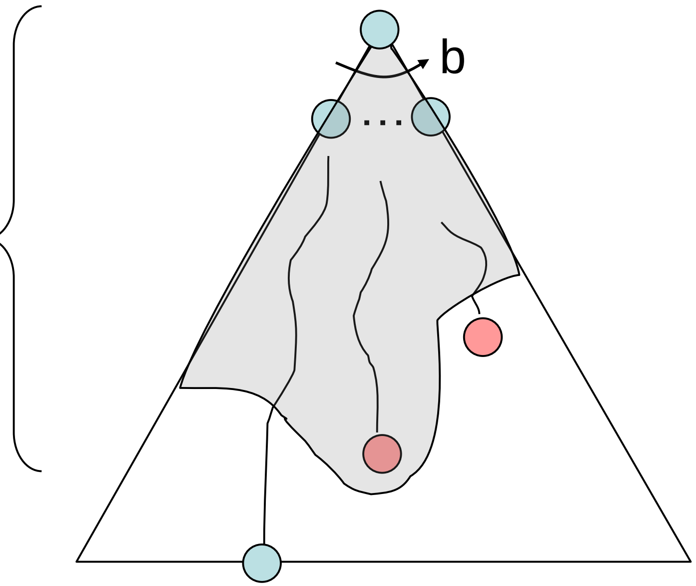
- Remember: explores increasing cost contours



Uniform Cost Search

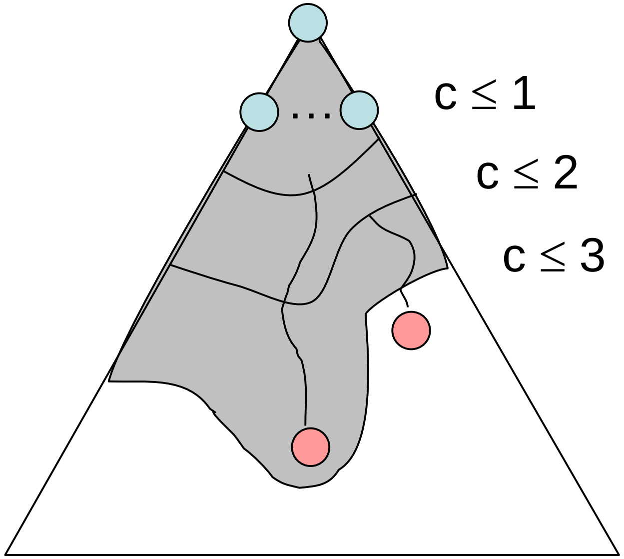
Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking				
BFS					
UCS					

C^*/ε tiers



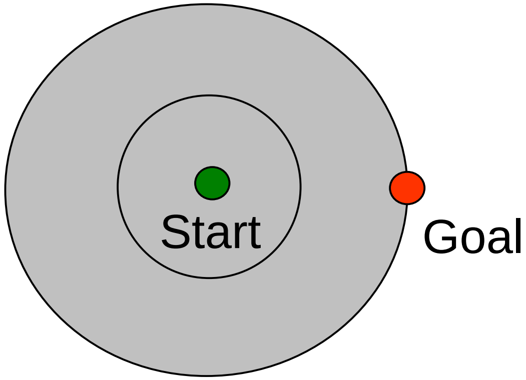
Uniform Cost Issues

- Remember: explores increasing cost contours

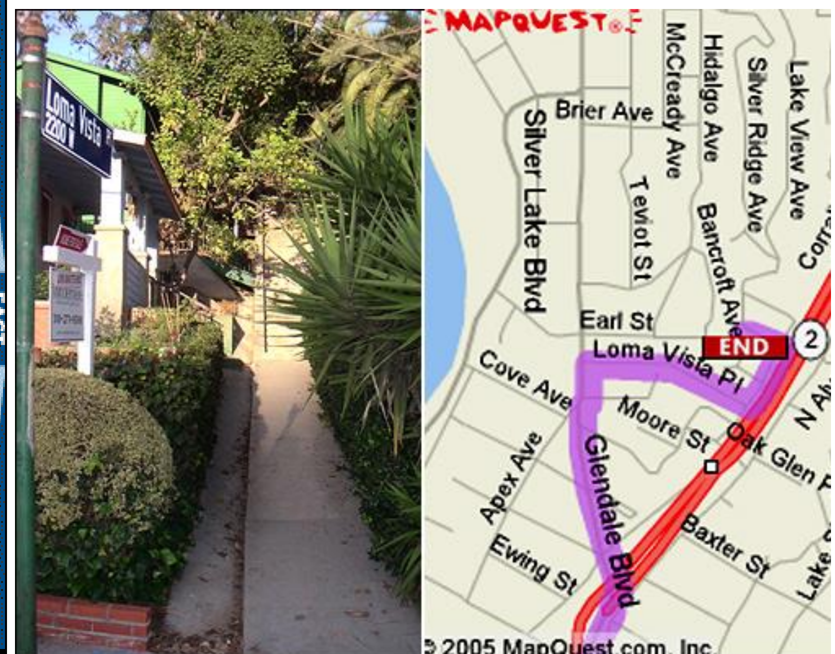


- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every "direction"
  - No information about goal location



Search Gone Wrong?



Summary

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Next time: informed search, A*