CSE 573: Artificial Intelligence

Problem Spaces & Search

Dan Weld

With slides from
Dan Klein, Stuart Russell, Andrew Moore, Luke Zettlemoyer, Dana Nau…

Logistics

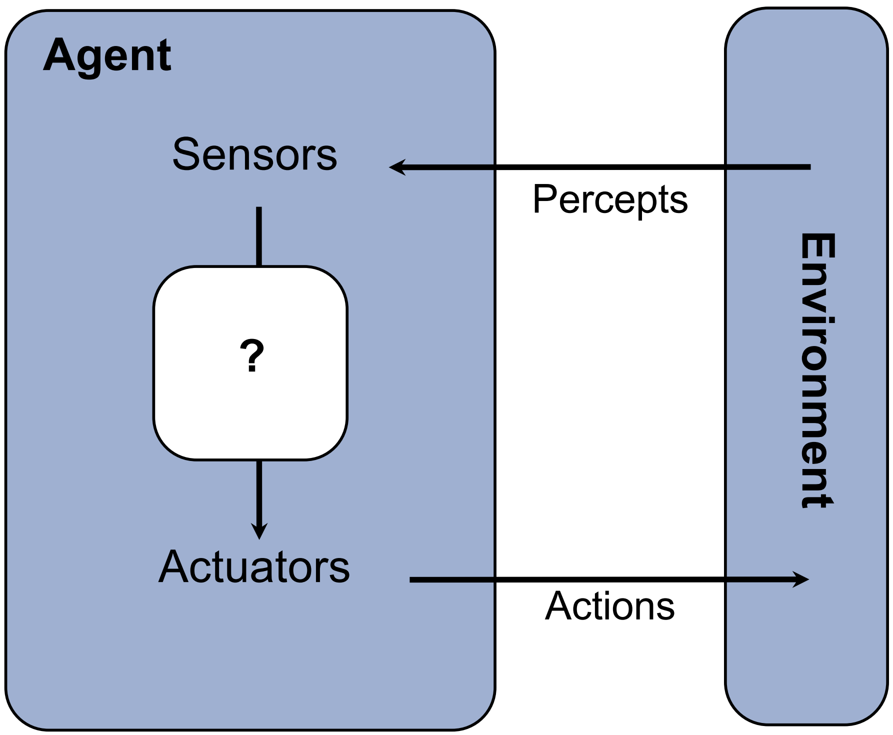
- Read Ch 3
- Form 2-person teams.
  - Post on forum if you want a partner
- Start PS1

Outline

- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Iterative Deepening Search
  - Uniform-Cost Search
- Heuristic Search Methods
- Heuristic Generation

Agent *vs.* Environment

- 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.

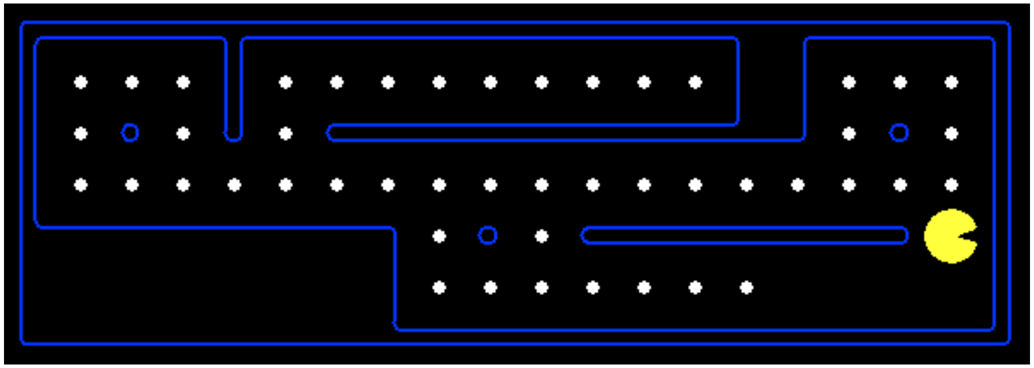


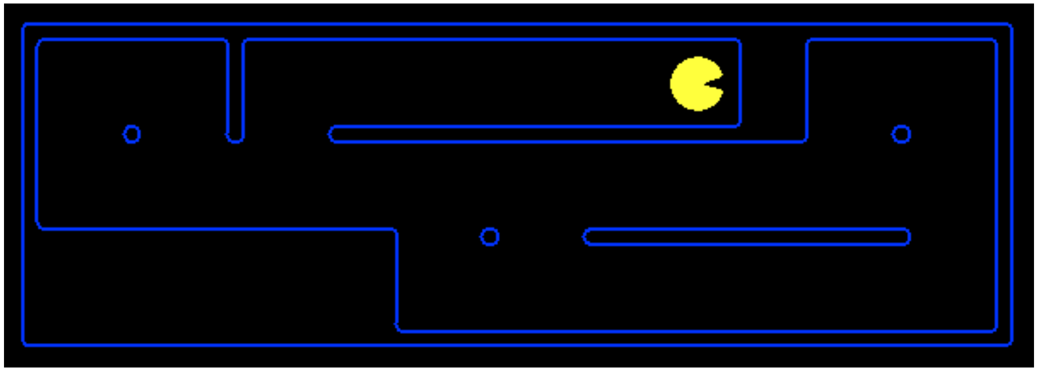
Goal Based Agents

- 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

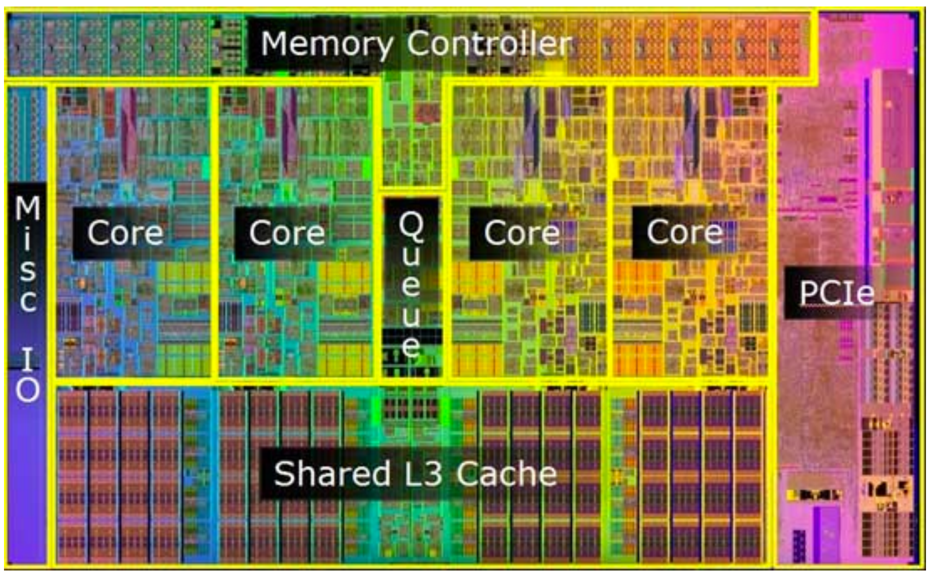
- Act on how the world WOULD BE





Search: It’s not just for Agents

Hardware verification



Planning optimal repair sequences

Search thru a Problem Space (aka State Space)

- Input:
  - Set of states
  - Operators *[and costs]*
  - Start state
  - Goal state *[or test]*

Functions: States → States

Aka "Successor Function"

- Output:
  - Path: start $\Rightarrow$ a state satisfying goal test

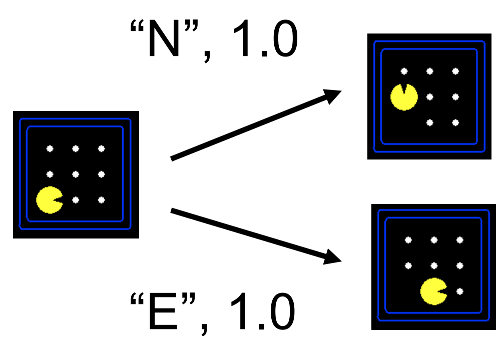
    [May require shortest path]

    [Sometimes just need a state that passes test]

Example: Simplified Pac-Man

- Input:
  - A state space

  - Successor function

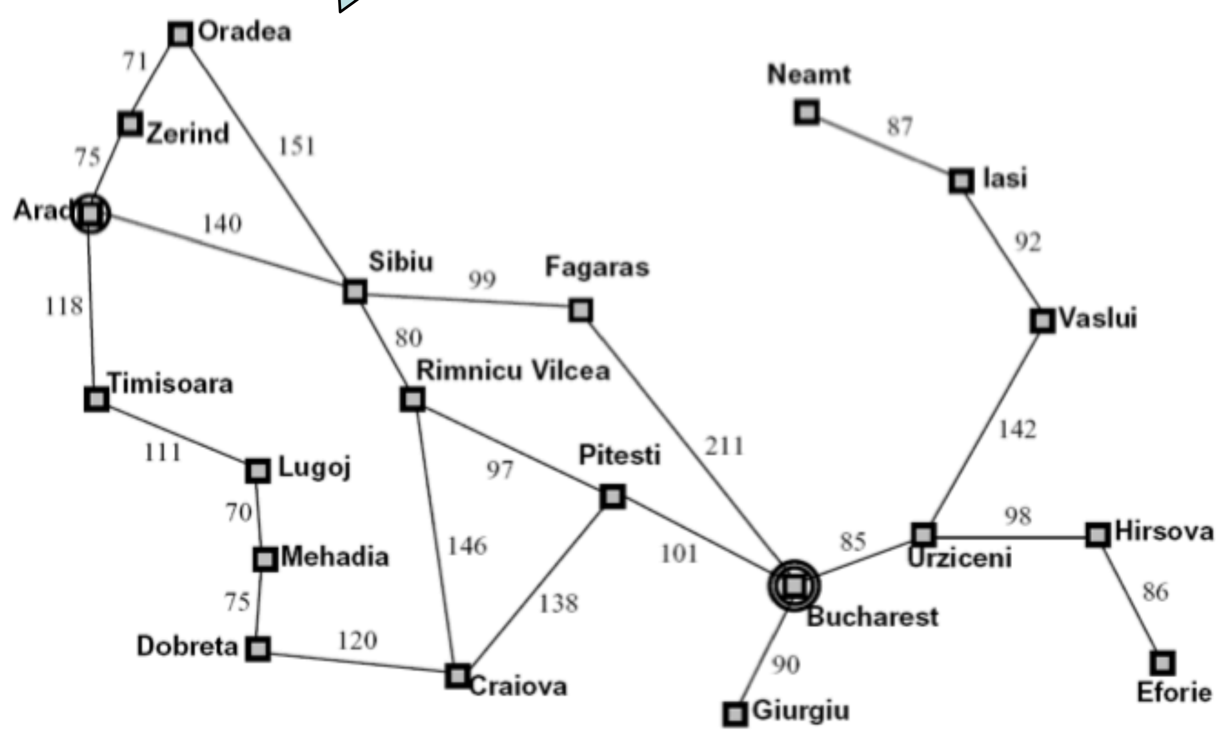


  - A start state
  - A goal test
- Output:

Ex: Route Planning: Arad → Bucharest

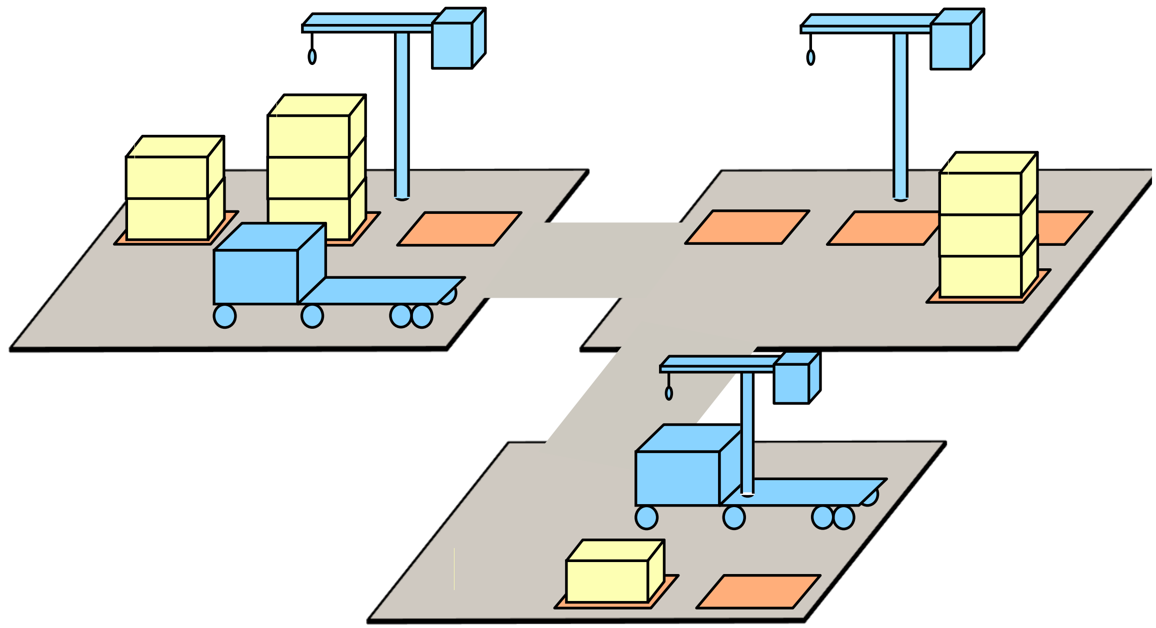
- Input:
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state (test)
- Output:

Different operators may be applicable in different states



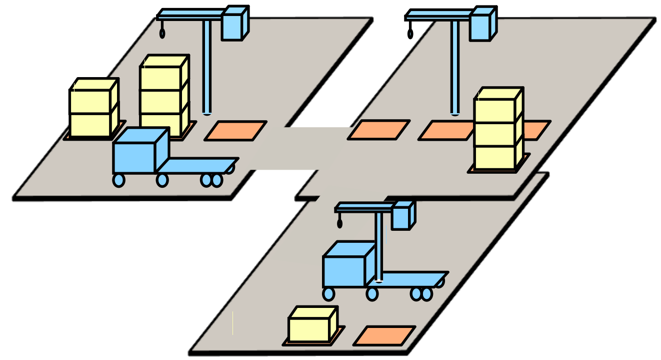
Ex: Dock Worker Robots

- A harbor with several locations
  - e.g., docks, docked ships, storage areas, parking areas
- Containers
  - going to/from ships
- Robot vehicles
  - can move containers
- Cranes
  - can load and unload containers



- Multiple robots can operate at the same time
- Move, load & other actions have **different durations**

Dock Worker 2



Input:

- Set of states

  Partially specified plans

- Operators [and costs]

  Plan modification operators

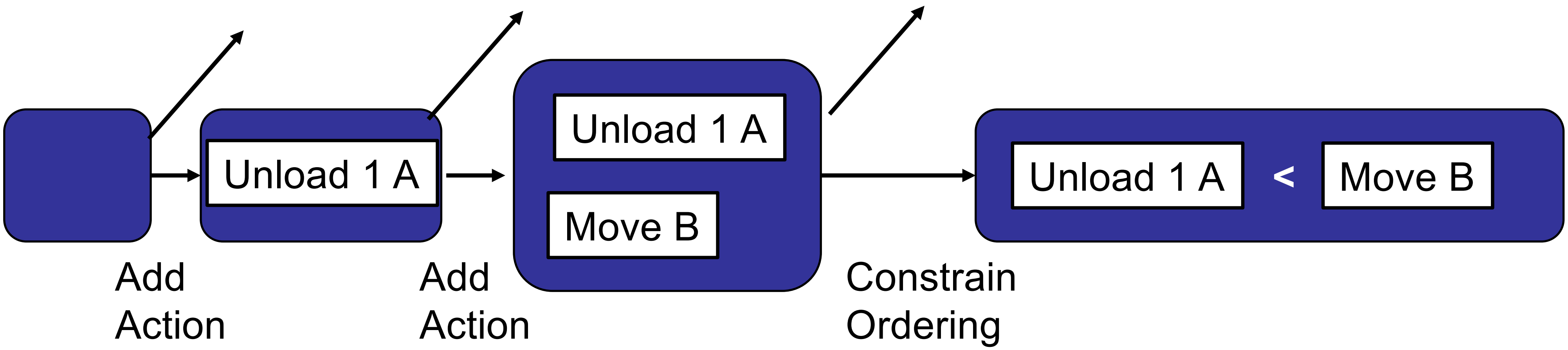
- Start state

  The null plan (no actions)

- Goal test

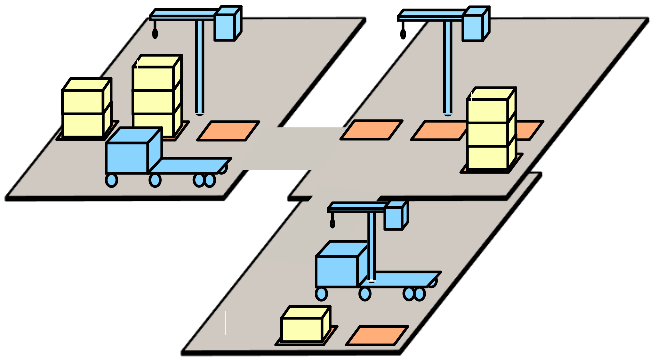
  A plan which provably achieves the desired world configuration

Plan Space



Blue boxes are plans = states in search space
Operators modify plans
Successors(p) = all possible ways of modifying p

Multiple Problem Spaces



Real World

- States of the world (e.g. loading dock configurations)
- Actions (take one world-state to another)

Robot's Head

- **Problem Space 1**
- PS states =
  - models of world states
- Operators =
  - models of actions

- **Problem Space 2**
- PS states =
  - partially spec. plan
- Operators =
  - plan modificat'n ops

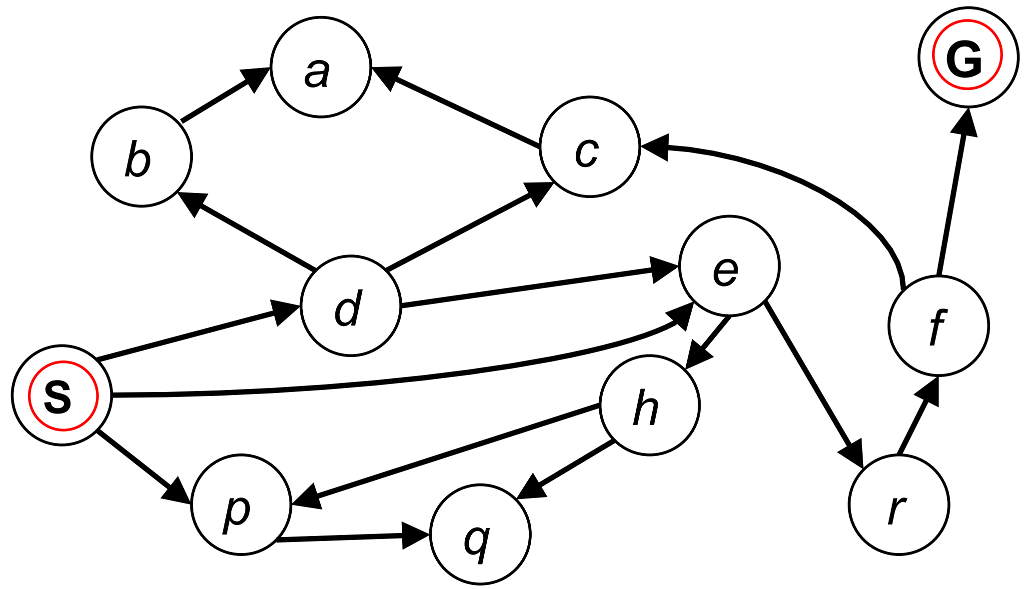
Algebraic Simplification

$$
\begin{aligned}
\partial_r^2 u &= -\left[E' - \frac{l(l+1)}{r^2} - r^2\right] u(r) \\
e^{-2s}\left(\partial_s^2 - \partial_s\right) u(s) &= -\left[E' - l(l+1)e^{-2s} - e^{2s}\right] u(s) \\
e^{-2s}\left[e^{\frac{1}{2}s}\left(e^{-\frac{1}{2}s}u(s)\right)'' - \frac{1}{4}u\right] &= -\left[E' - l(l+1)e^{-2s} - e^{2s}\right] u(s) \\
e^{-2s}\left[e^{\frac{1}{2}s}\left(e^{-\frac{1}{2}s}u(s)\right)''\right] &= -\left[E' - \left(l + \frac{1}{2}\right)^2 e^{-2s} - e^{2s}\right] u(s) \\
v'' &= -e^{2s}\left[E' - \left(l + \frac{1}{2}\right)^2 e^{-2s} - e^{2s}\right] v
\end{aligned}
$$

- Input:
  - Set of states
  - Operators [and costs]
  - Start state
  - Goal state (test)
- Output:

State Space Graphs

- State space graph:
  - Each node is a state
  - The operators are represented by arcs
  - Edges may be labeled with costs

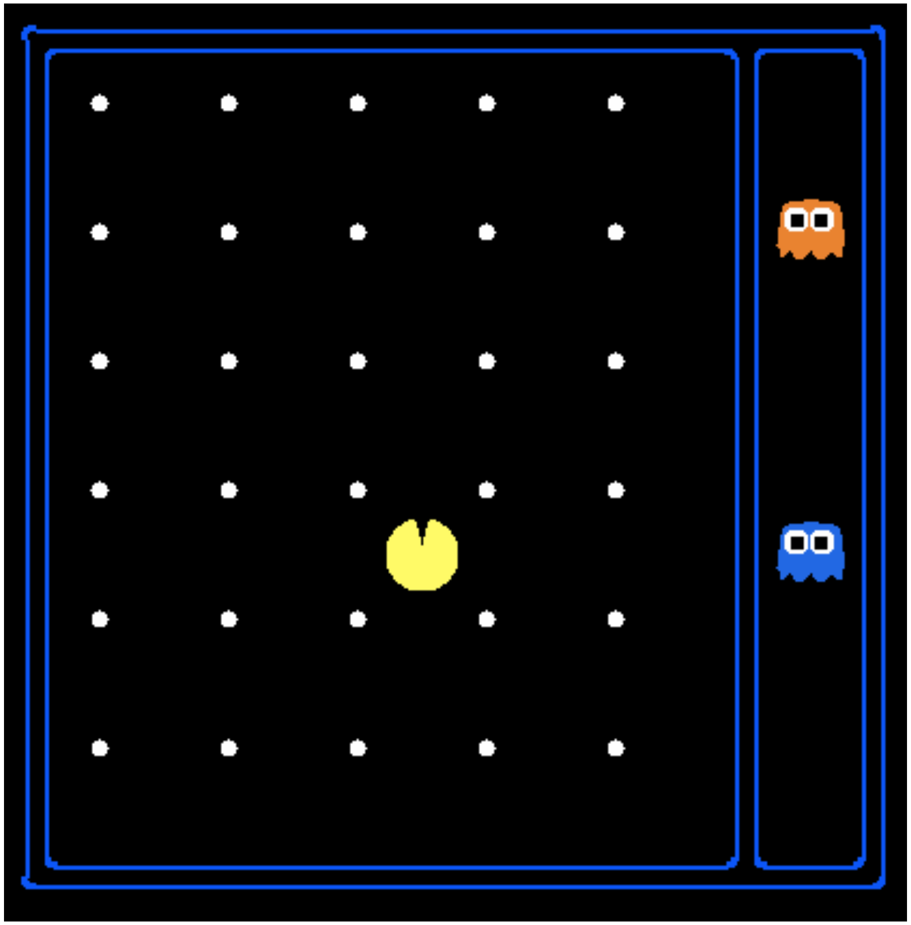


- We can rarely build this graph in memory (so we don't try)

Ridiculously tiny search graph for a tiny search problem

State Space Sizes?

- Search Problem:
  Eat all of the food
- Pacman positions:
  10 x 12 = 120
- Pacman facing:
  up, down, left, right
- Food configurations: 2^{30}
- Ghost1 positions: 12
- Ghost 2 positions: 11

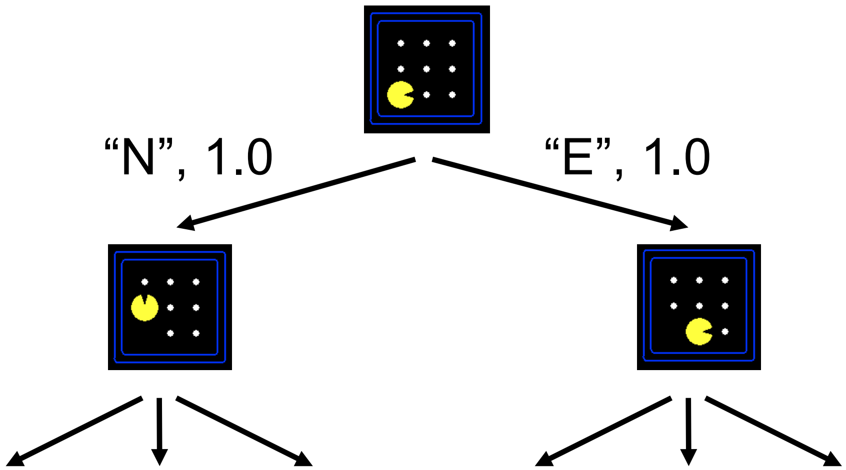


$120 \times 4 \times 2^{30} \times 12 \times 11 = 6.8 \times 10^{13}$

Search Methods

- Blind Search
  - Depth first search
  - Breadth first search
  - Iterative deepening search
  - Uniform cost search
- Local Search
- Informed Search
- Constraint Satisfaction
- Adversary Search

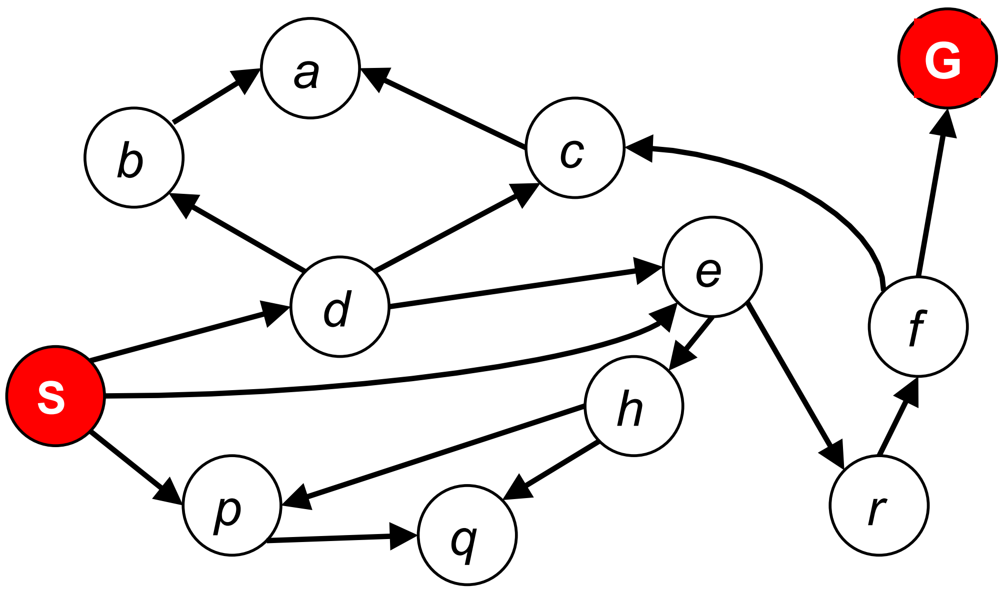
Search Trees



- A search tree:
  - Start state at the root node
  - Children correspond to successors
  - Nodes ***contain*** states, correspond to PLANS to those states
  - Edges are labeled with actions and costs
  - For most problems, we can never actually build the whole tree

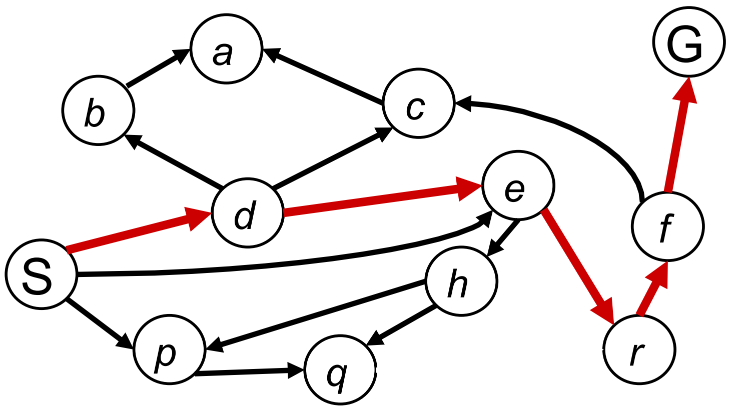
Example: Tree Search

State graph:

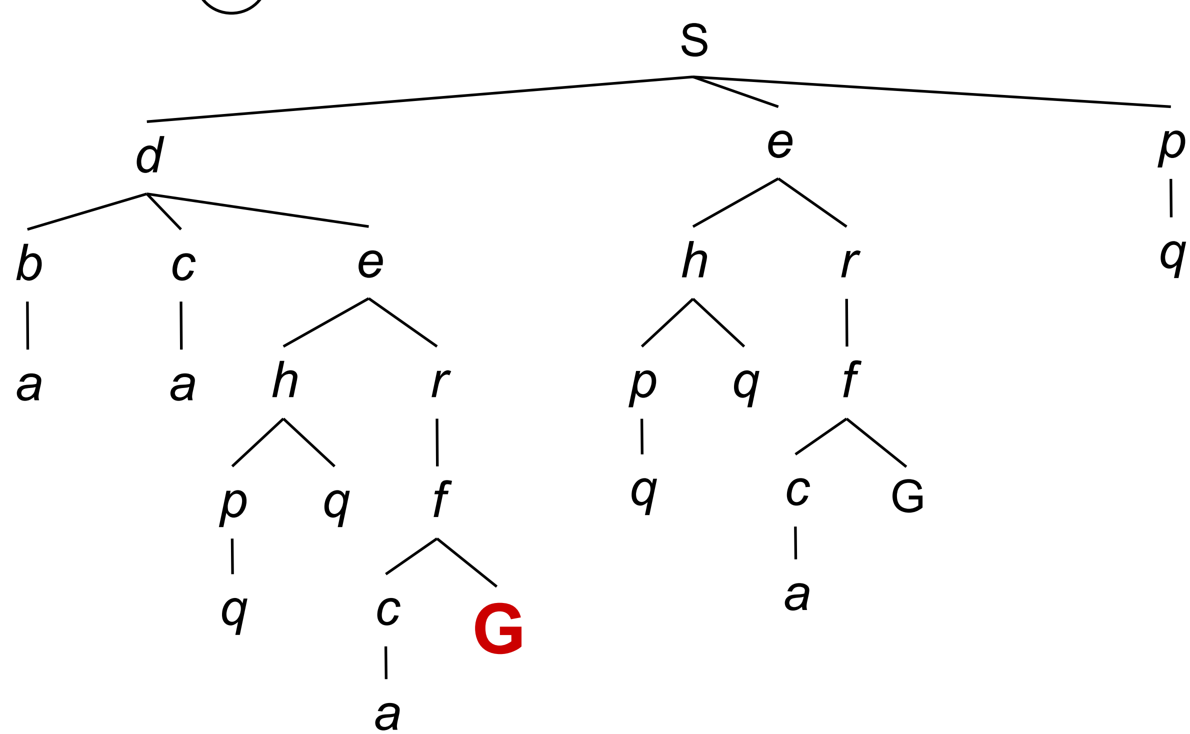


What is the search tree?

State Graphs vs. Search Trees



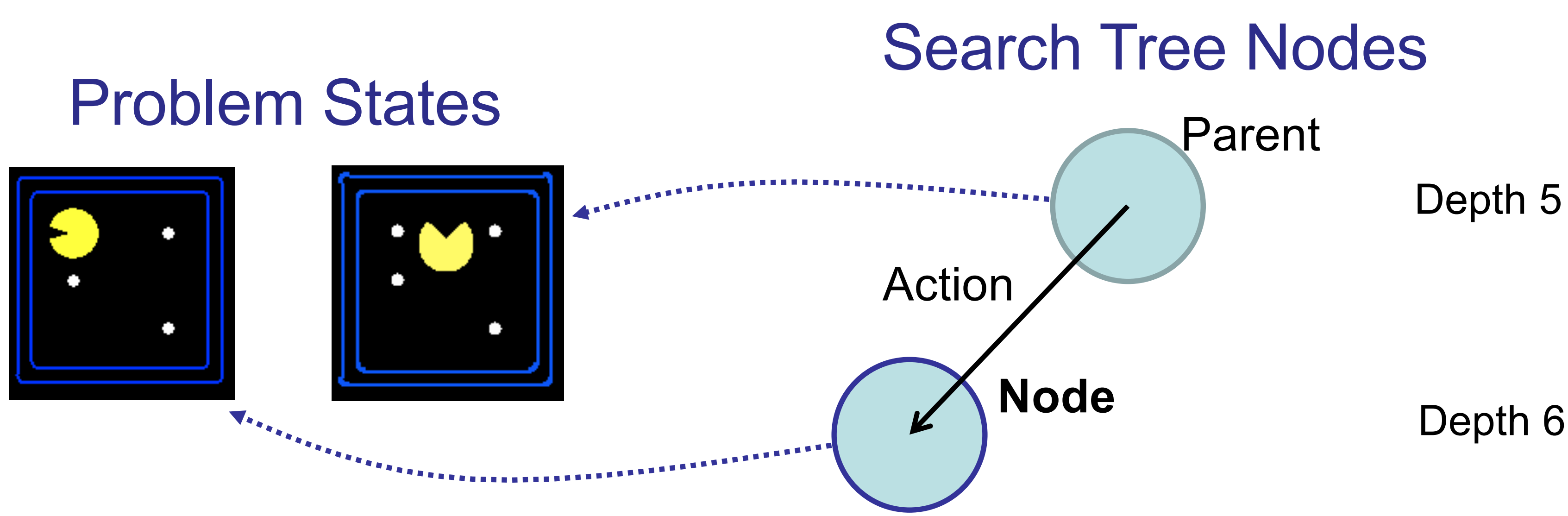
Each NODE in in the search tree denotes an entire PATH in the problem graph.



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

States vs. Nodes

- Vertices in state space graphs are problem states
  - Represent an abstracted state of the world
  - Have successors, can be goal / non-goal, have multiple predecessors
- Vertices in search trees ("Nodes") are plans
  - Contain a problem state and one parent, a path length, a depth & a cost
  - Represent a plan (sequence of actions) which results in the node's state
  - The same problem state may be achieved by multiple search tree nodes



Building Search Trees

- Search:
  - Expand out possible nodes (plans) in the tree
  - Maintain a fringe of unexpanded nodes
  - Try to expand as few 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
```

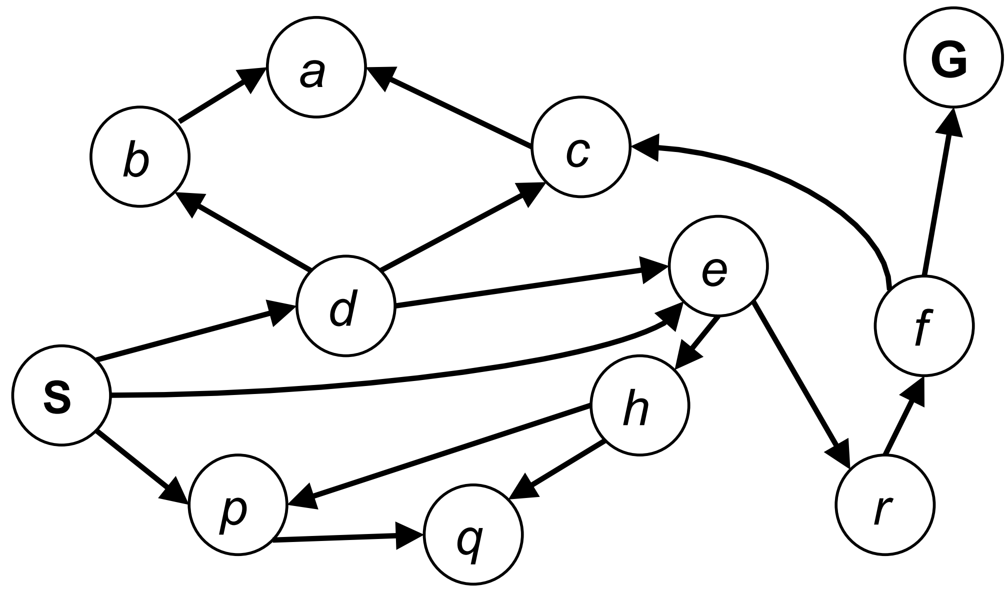
Detailed pseudocode is in the book!

Important ideas:

- Fringe (leaves of tree)
- Expansion (adding successors of a leaf)
- Exploration strategy

which fringe node to expand next?

Review: Depth First Search



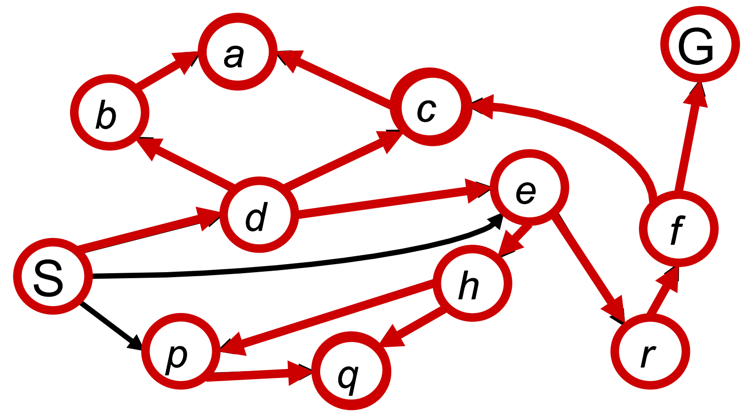
Strategy: *expand* ***deepest*** *node first*

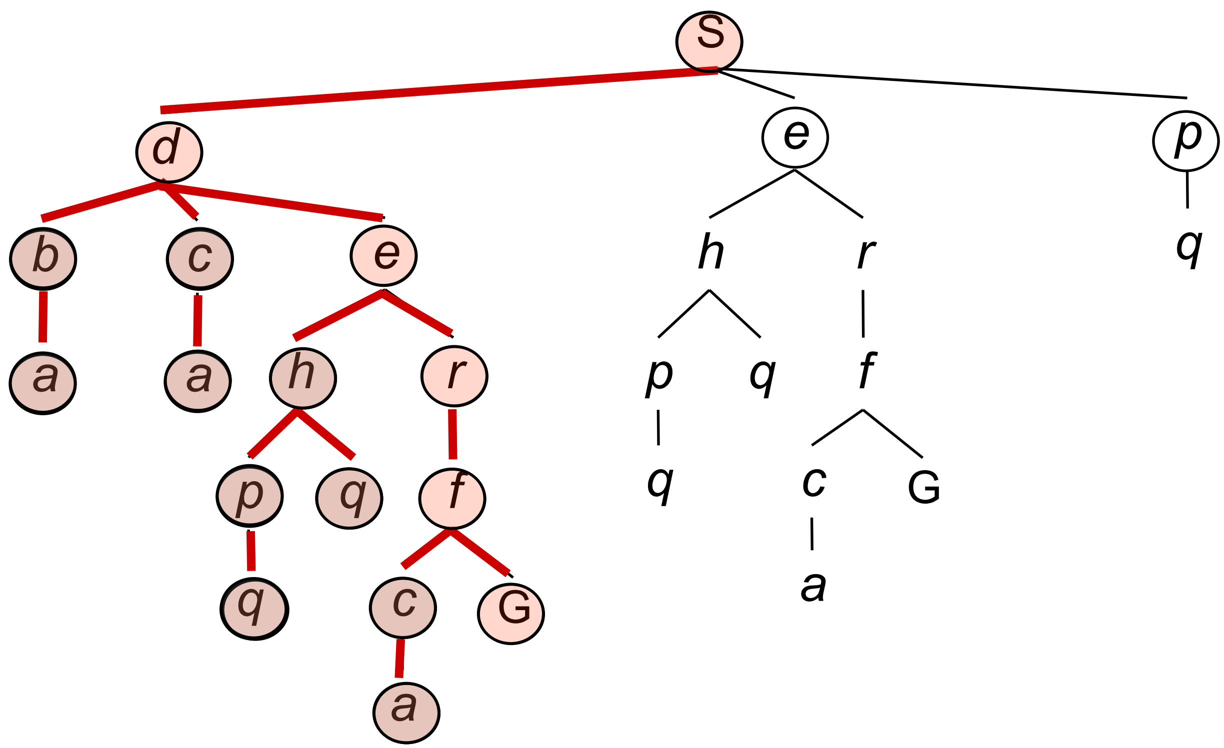
Implementation:
Fringe is a stack - LIFO

Review: Depth First Search

Expansion ordering:

(d,b,a,c,a,e,h,p,q,q,r,f,c,a,G)

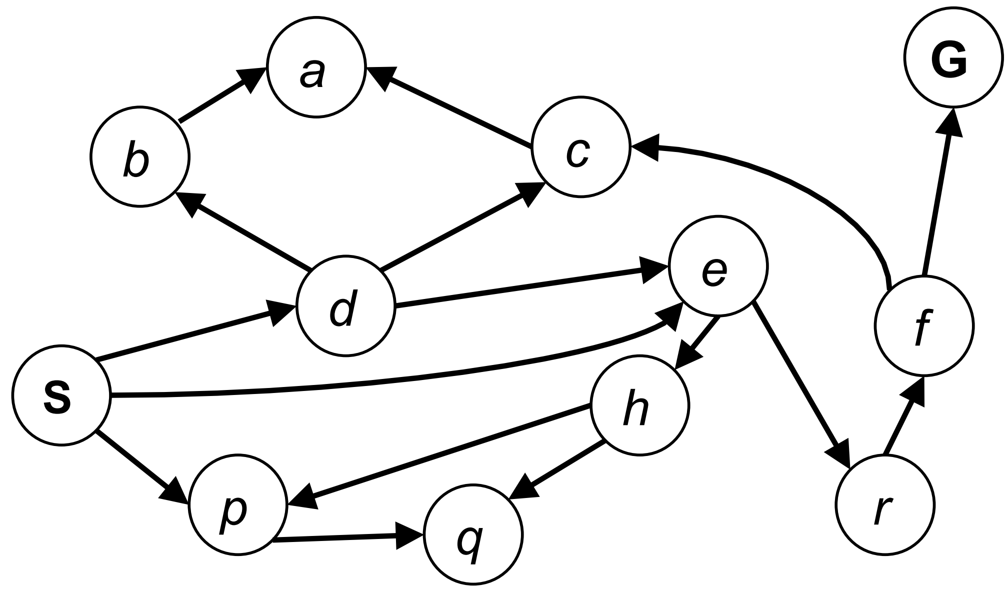




Review: Breadth First Search

Strategy: *expand* ***shallowest*** *node first*

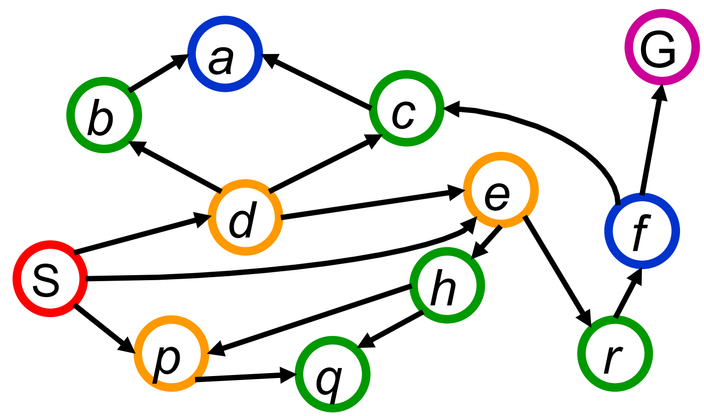
Implementation:
Fringe is a queue - FIFO

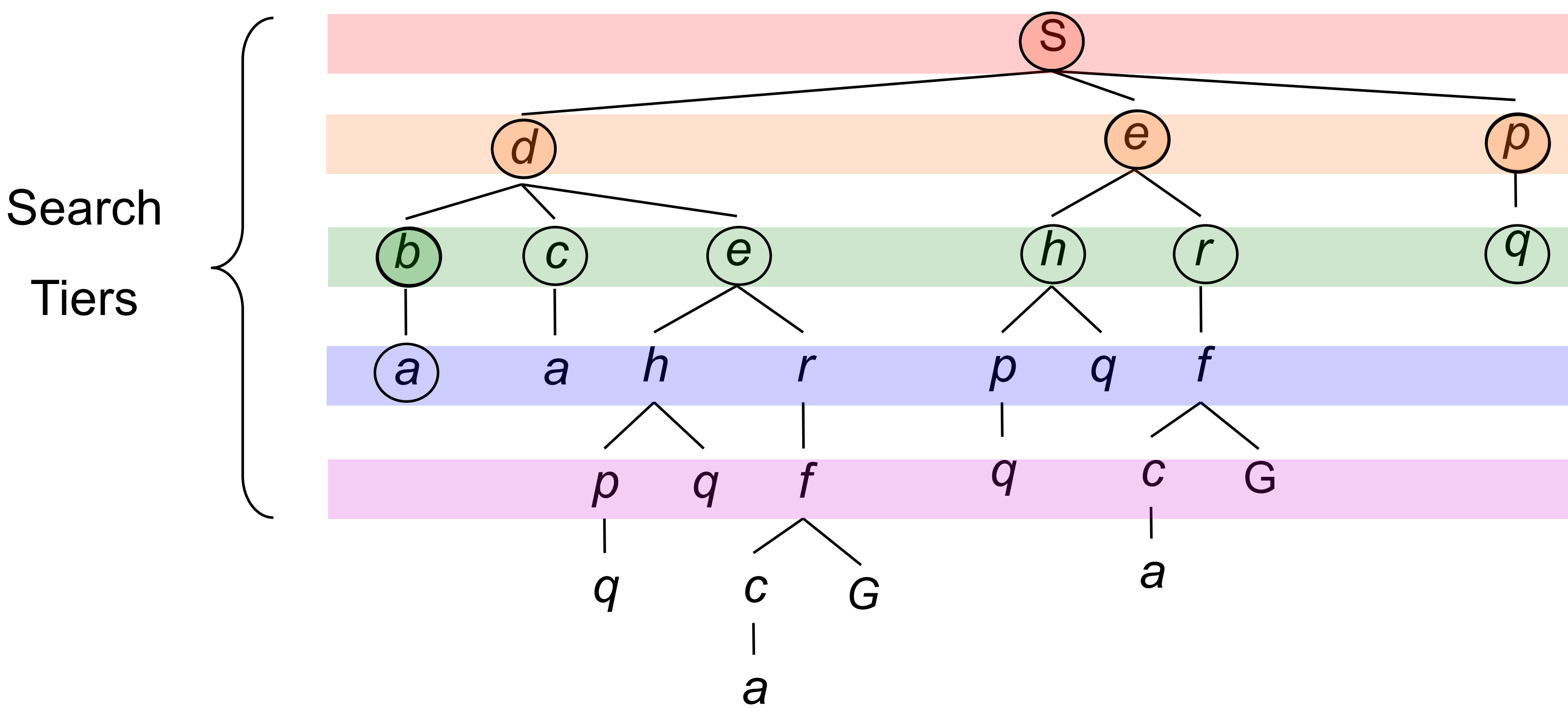


Review: Breadth First Search

Expansion order:

(S,d,e,p,b,c,e,h,r,q,a,a ,h,r,p,q,f,p,q,f,q,c,G)





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:

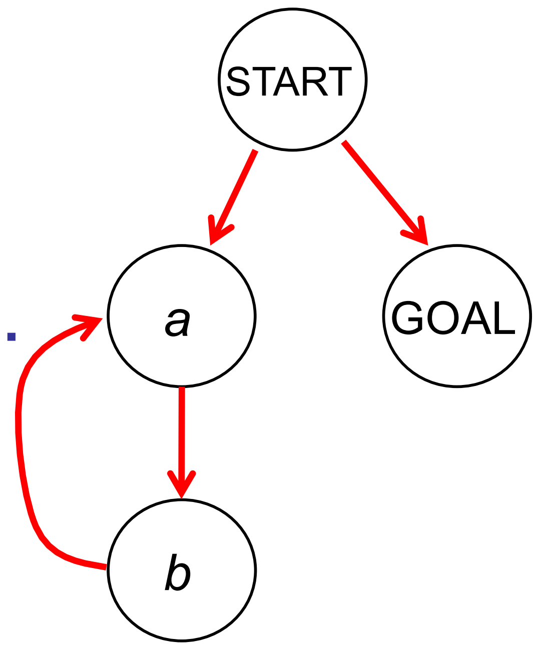
n	Number of states in the problem
b	The maximum branching factor *B* (the maximum number of successors for a state)
C^*	Cost of least cost solution
d	Depth of the shallowest solution
m	Max depth of the search tree

Depth-First Search

Assuming finite tree

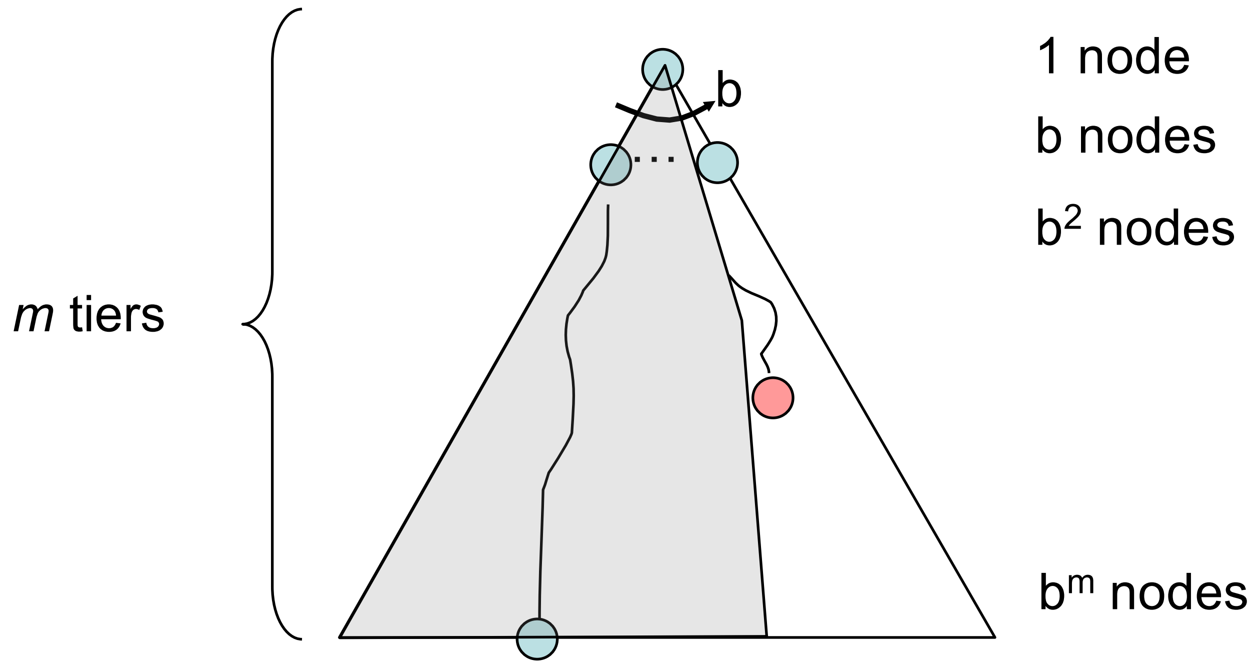
Algorithm		Complete	Optimal	Time	Space
DFS	Depth First Search	No	No	$O(b^m)$	O(b m)

- Infinite paths make DFS incomplete…
  - How can we fix this?
  - Check new nodes against ***path*** from S



d depth of solution
m max depth of tree

DFS Search (w/ cycle checking)

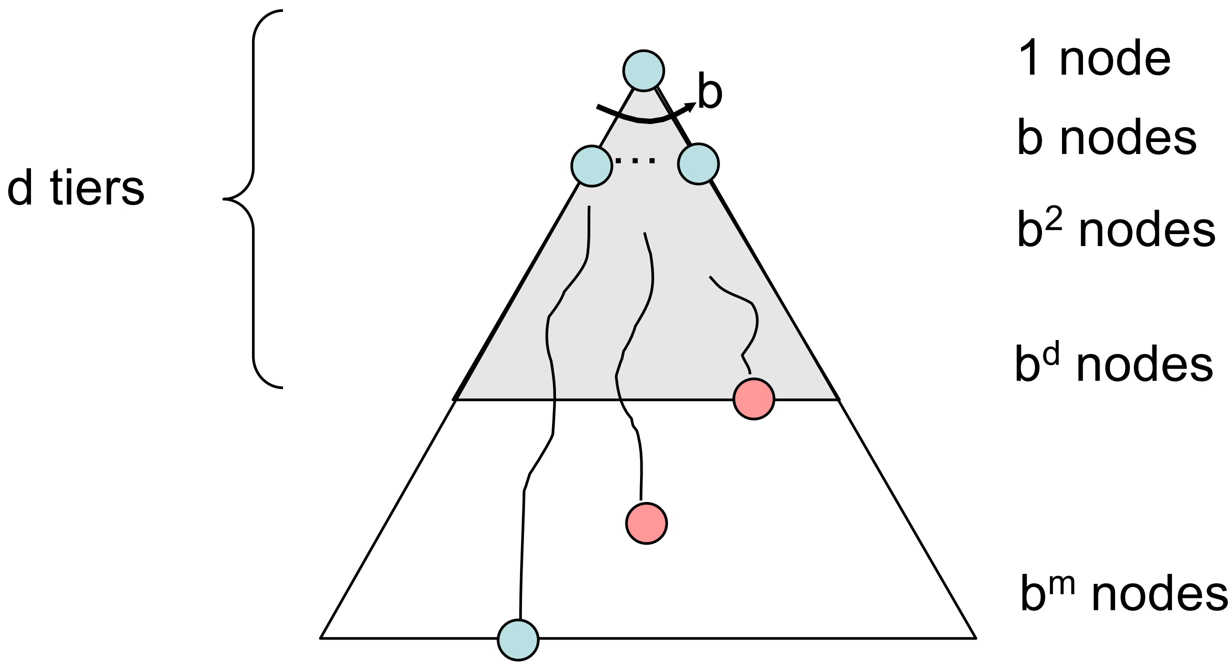


Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y if finite	N	$O(b^m)$	$O(b\ m)$

Only if finite tree

BFS Tree Search

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	N unless finite	N	$O(b^m)$	$O(bm)$
BFS		Y*	Y*	$O(b^d)$	$O(b^d)$



* Assuming finite branching factor

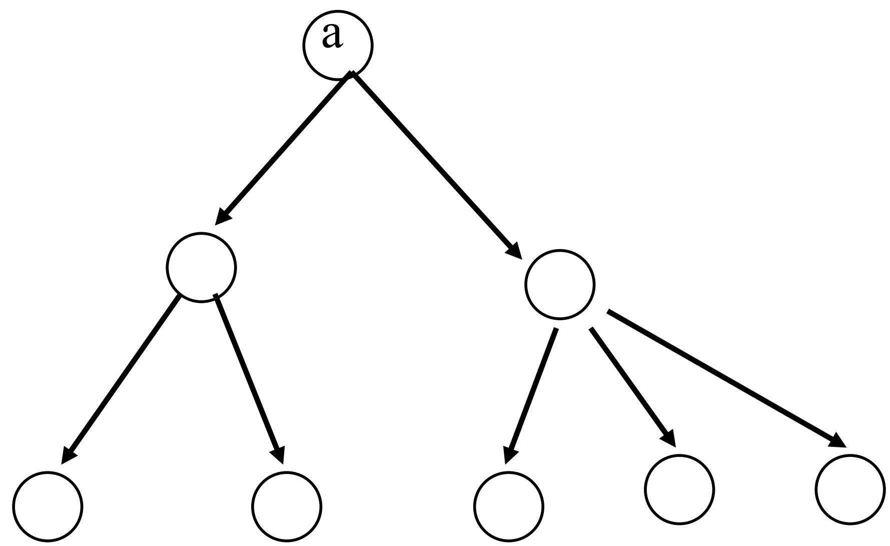
Memory a Limitation?

- Suppose:
  - 4 GHz CPU
  - 32 GB main memory
  - 100 instructions / expansion
  - 5 bytes / node

- 40 M expansions / sec
  - Memory filled in … 3 min

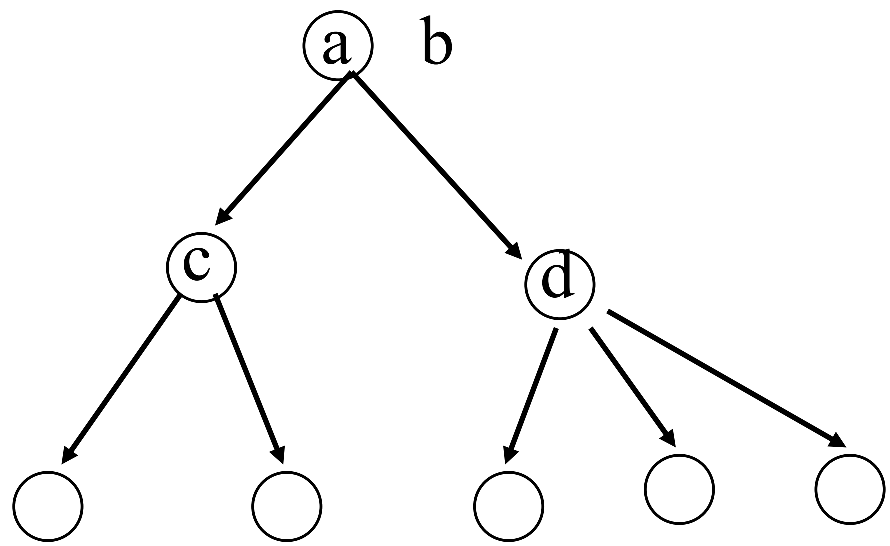
Iterative Deepening Search

- DFS with limit; incrementally grow limit
- Evaluation



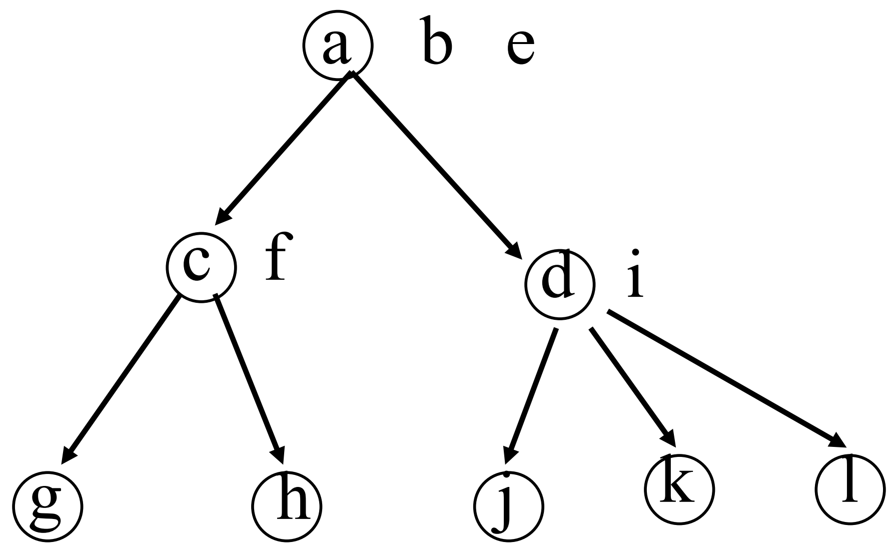
Iterative Deepening Search

- DFS Tree Search with limit; incrementally grow limit
- Evaluation



Iterative Deepening Search

- DFS Tree Search with limit; incrementally grow limit
- Evaluation
  - Complete?
  - Time Complexity?
  - Space Complexity?



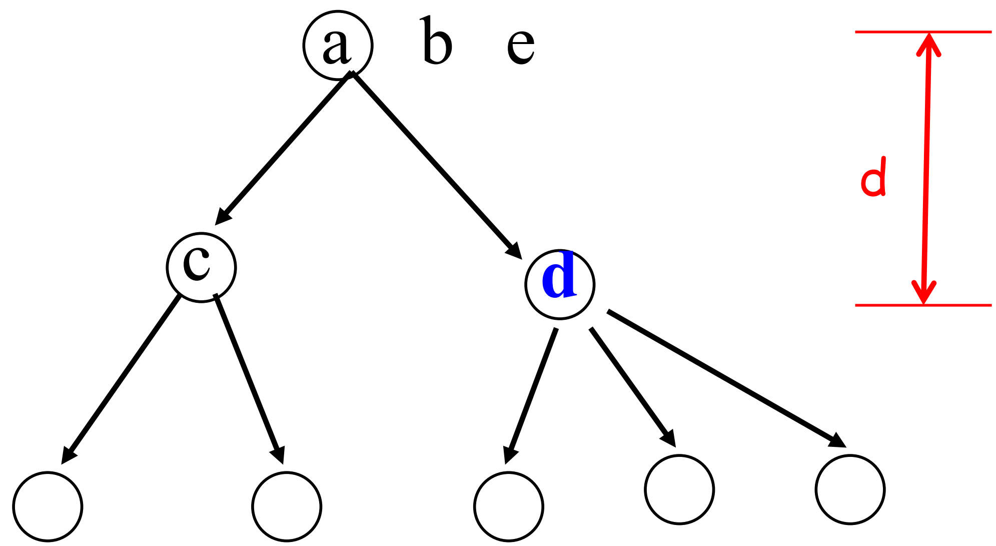
Iterative Deepening Search

- DFS with limit; incrementally grow limit
- Evaluation
  - Complete?

    Yes *
  - Time Complexity?

    $O(b^d)$
  - Space Complexity?

    $O(bd)$



* Assuming branching factor is finite
Important Note: no cycle checking necessary!

Cost of Iterative Deepening

b	ratio ID to DFS
2	3
3	2
5	1.5
10	1.2
25	1.08
100	1.02

Speed

Assuming 10M nodes/sec & sufficient memory

	BFS Nodes	BFS Time		Iter. Deep. Nodes	Iter. Deep. Time
8 Puzzle	10^5	.01 sec		10^5	.01 sec
2x2x2 Rubik's	10^6	.2 sec		10^6	.2 sec
15 Puzzle	10^{13}	6 days	1Mx	10^{17}	20k yrs
3x3x3 Rubik's	10^{19}	68k yrs	8x	10^{20}	574k yrs
24 Puzzle	10^{25}	12B yrs		10^{37}	10^{23} yrs

Why the difference?

Rubik has higher branch factor
15 puzzle has greater depth

# of duplicates

Slide adapted from Richard Korf presentation

Search Methods

- Depth first search (DFS)
- Breadth first search (BFS)
- Iterative deepening depth-first search (IDS)

Blind search

Search Methods

- Depth first search (DFS)
- Breadth first search (BFS)
- Iterative deepening depth-first search (IDS)
- Best first search
- Uniform cost search (UCS)
- Greedy search
- A*
- Iterative Deepening A* (IDA*)
- Beam search
- Hill climbing

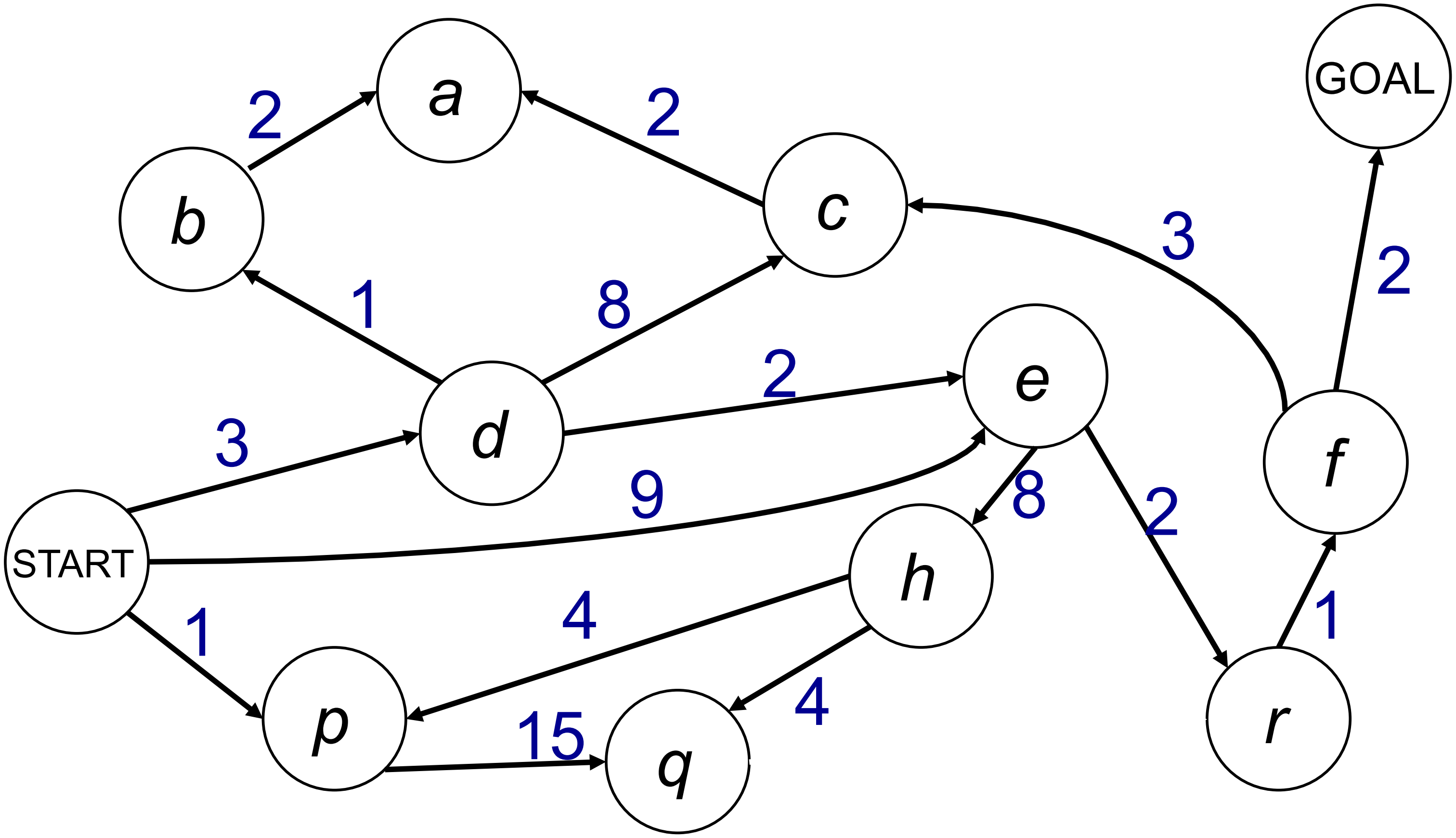
Heuristic search

Blind vs Heuristic Search

- Costs on Actions
- Heuristic Guidance

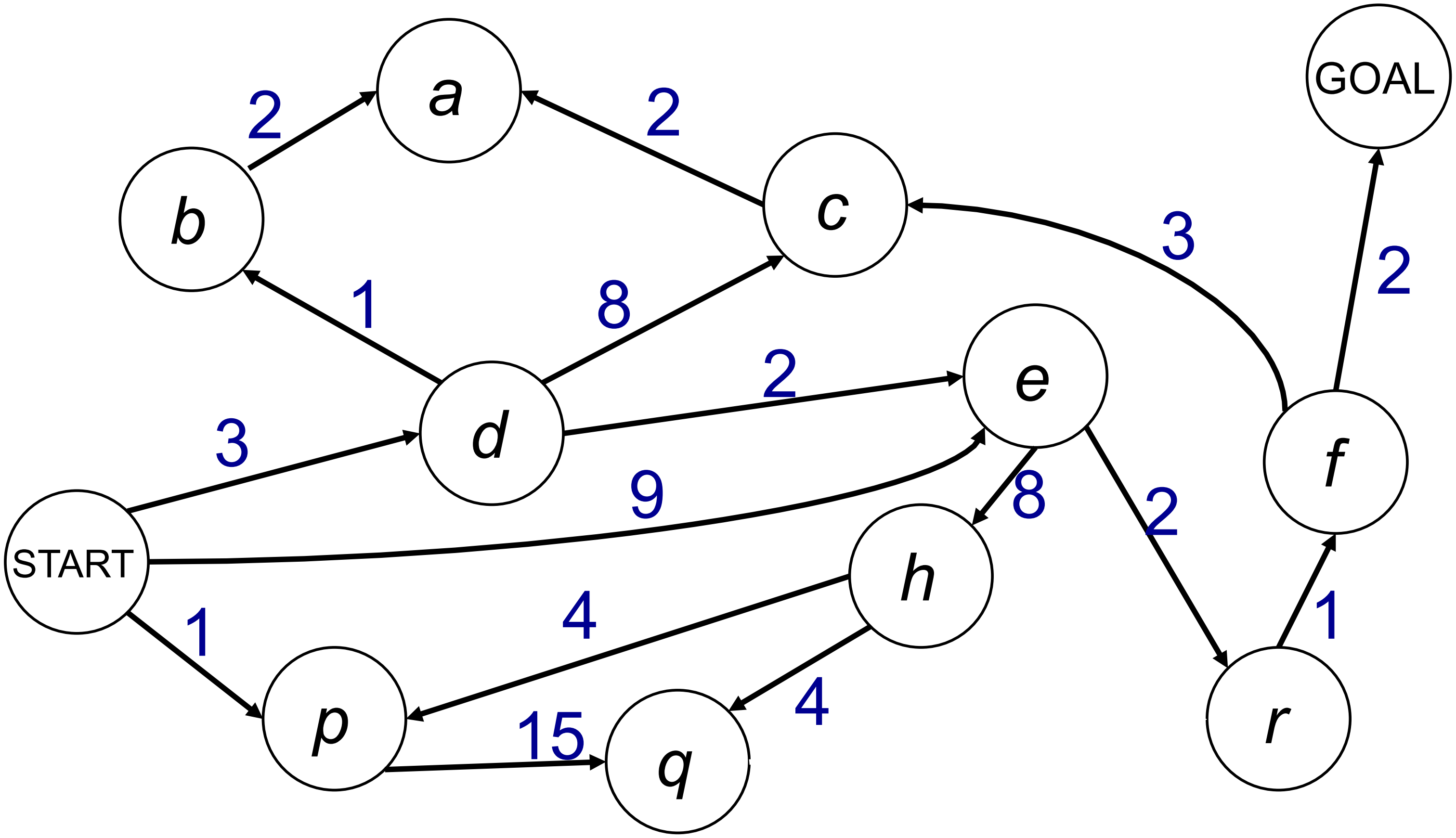
Separable Issues, but usually linked.

Costs on Actions



Objective: Path with smallest overall cost

Costs on Actions



What will BFS return?

… finds the shortest path in terms of number of transitions.

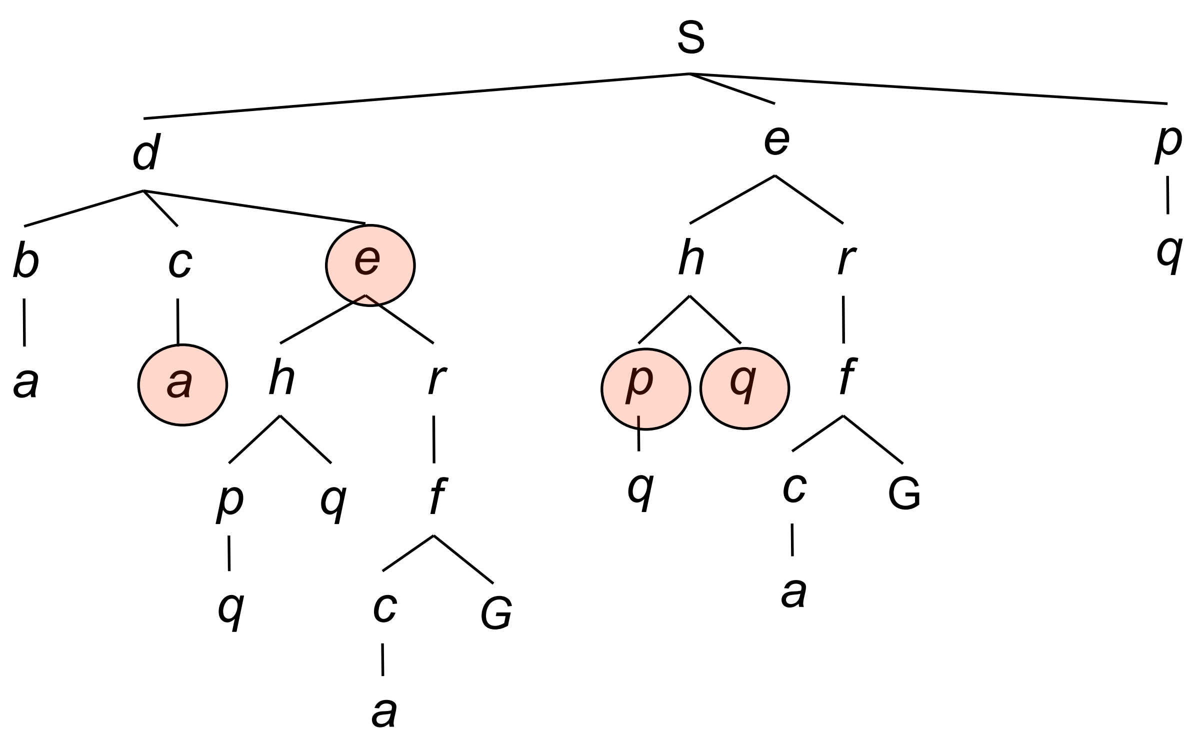
It does ***not*** find the least-cost path.

Best-First Search

- Generalization of breadth-first search
- Fringe = ***Priority*** queue of nodes to be explored
- Cost function f(n) applied to each node

Tree vs Graph Search

In BFS, for example, we shouldn't bother expanding the circled nodes (why?)



Graph Search

- Very simple fix: never expand a state type twice

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure

   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if fringe is empty then return failure
       node ← REMOVE-FRONT(fringe)
       if GOAL-TEST(problem, STATE[node]) then return node
       if STATE[node] is not in closed then
           add STATE[node] to closed
           fringe ← INSERTALL(EXPAND(node, problem), fringe)
   end
```

Some Hints

- On small problems
  - Graph search almost always better than tree search
  - Implement your closed list as a dict or set!
- On many real problems
  - Storage space is a huge concern
  - Graph search impractical

Best-First Search

- Generalization of breadth-first search
- Fringe = ***Priority*** queue of nodes to be explored
- Cost function f(n) applied to each node

```
Add initial state to priority queue
While queue not empty
        Node = head(queue)
        If goal?(node) then return node
        Add new children of node to queue
```

"*expanding the node*" (arrow pointing to "Add *new* children of node to queue")

Old Friends

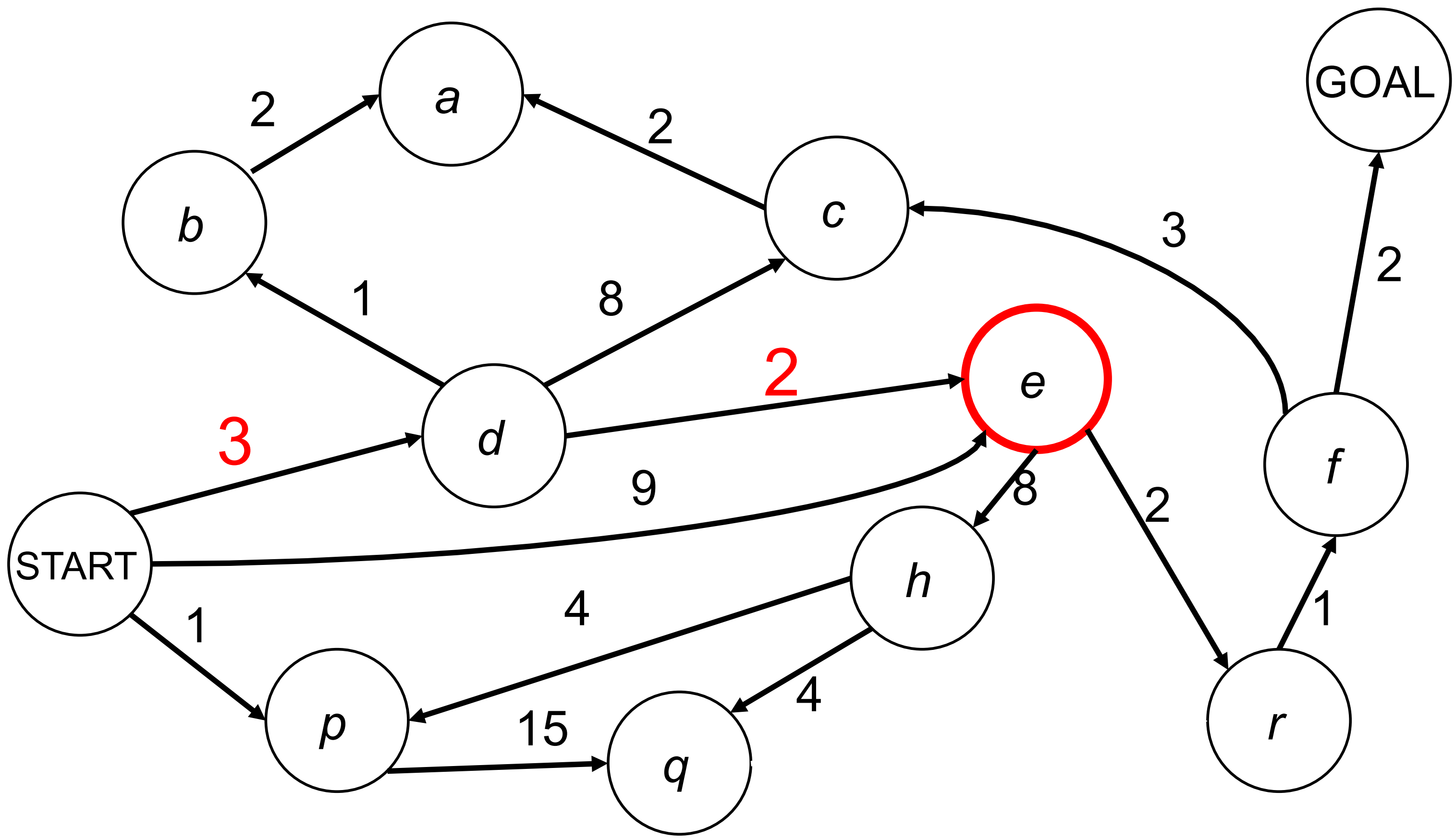
- Breadth First =
  - Best First
  - with f(n) = depth(n)

- Dijkstra’s Algorithm (Uniform cost) =
  - Best First
  - with f(n) = the sum of edge costs from start to n

Uniform Cost Search

Best first, where

f(n) = “cost from **start** to **n**”

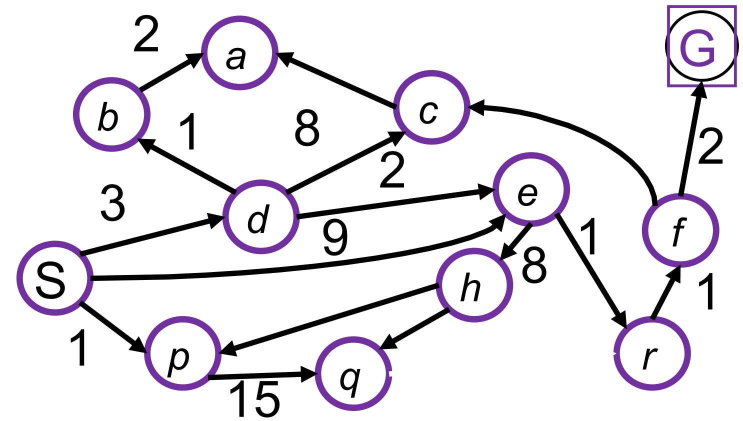


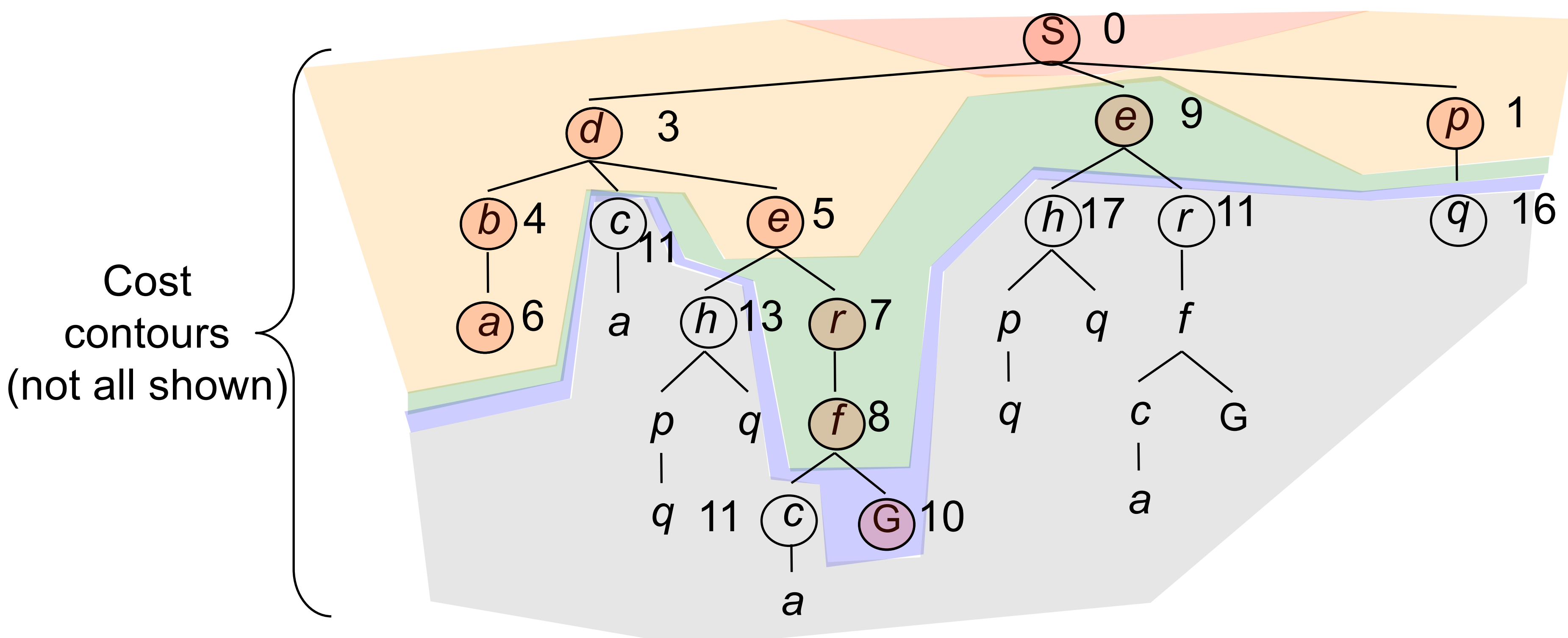
aka “Dijkstra’s Algorithm”

Uniform Cost Search

Expansion order:

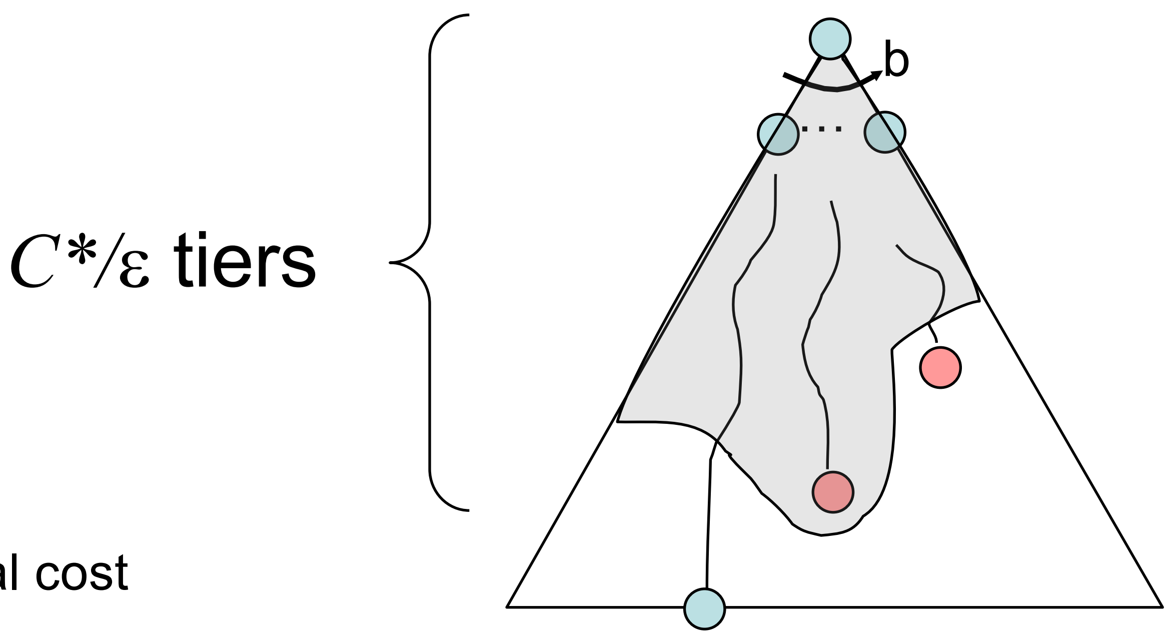
S, p, d, b, e, a, r, f, e, G





Uniform Cost Search

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y if finite	N	$O(b^m)$	$O(bm)$
BFS		Y	Y*	$O(b^d)$	$O(b^d)$
UCS		Y*	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$

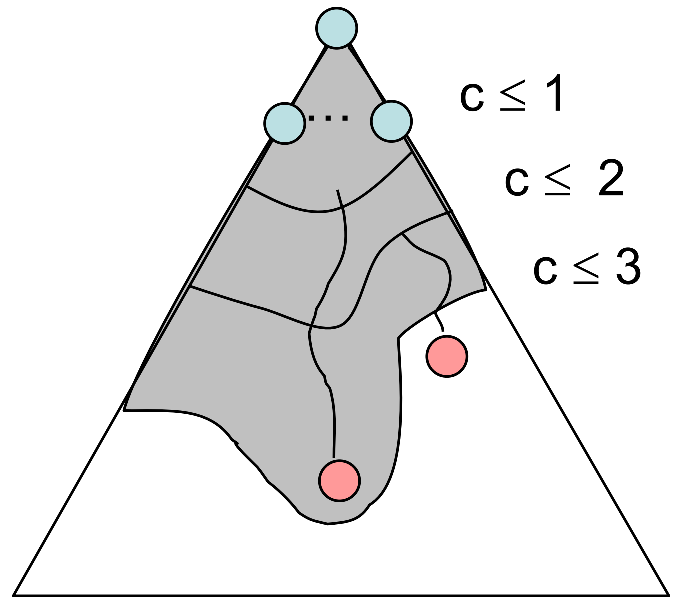


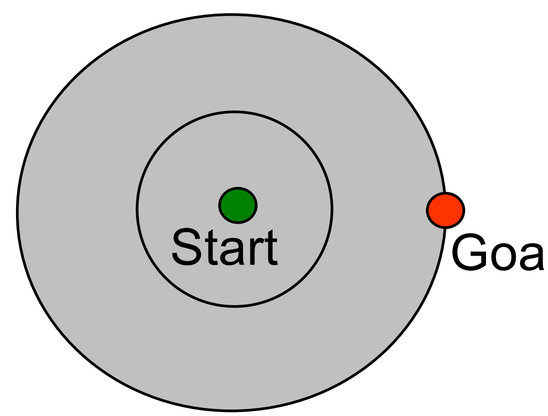
C^* = Optimal cost

ε = Minimum cost of an action

Uniform Cost Issues

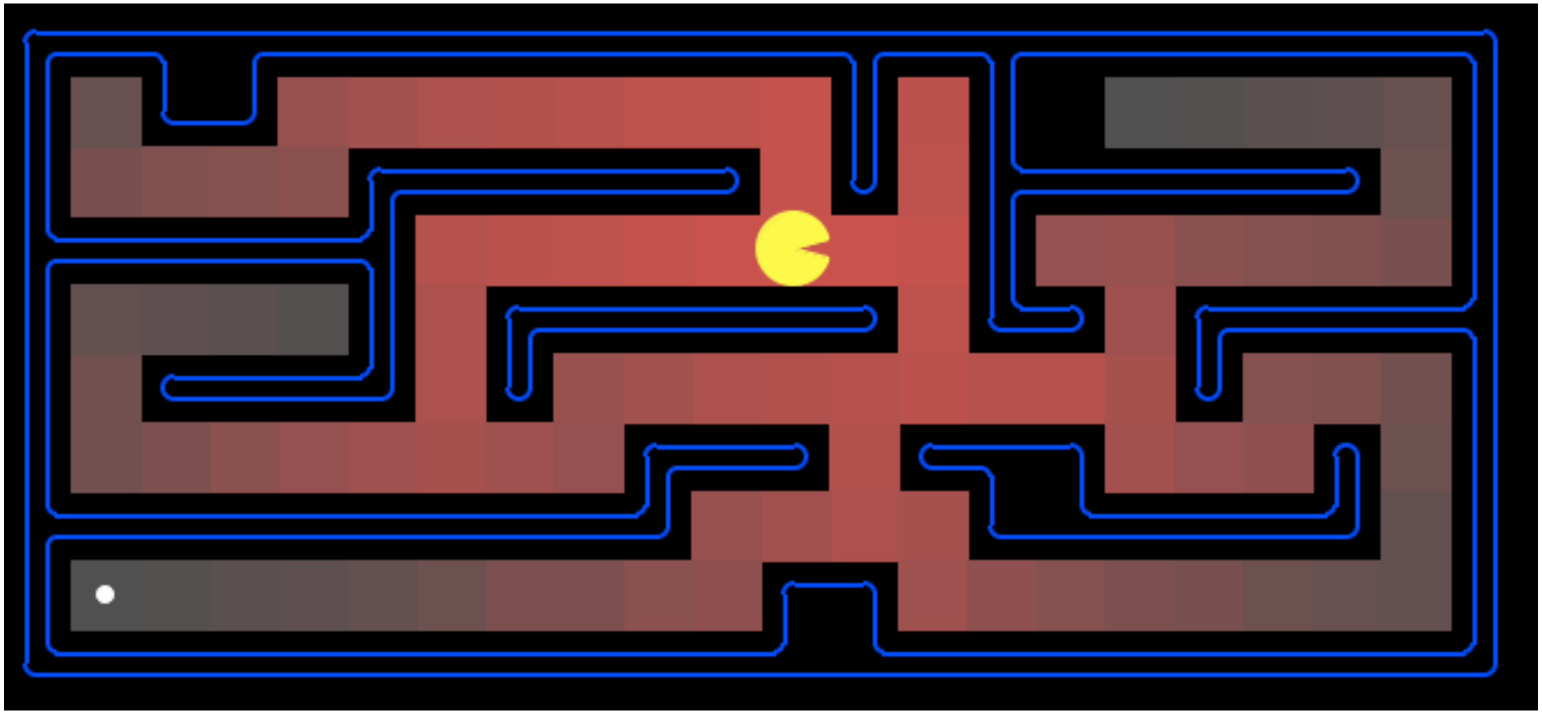
- Remember: explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location





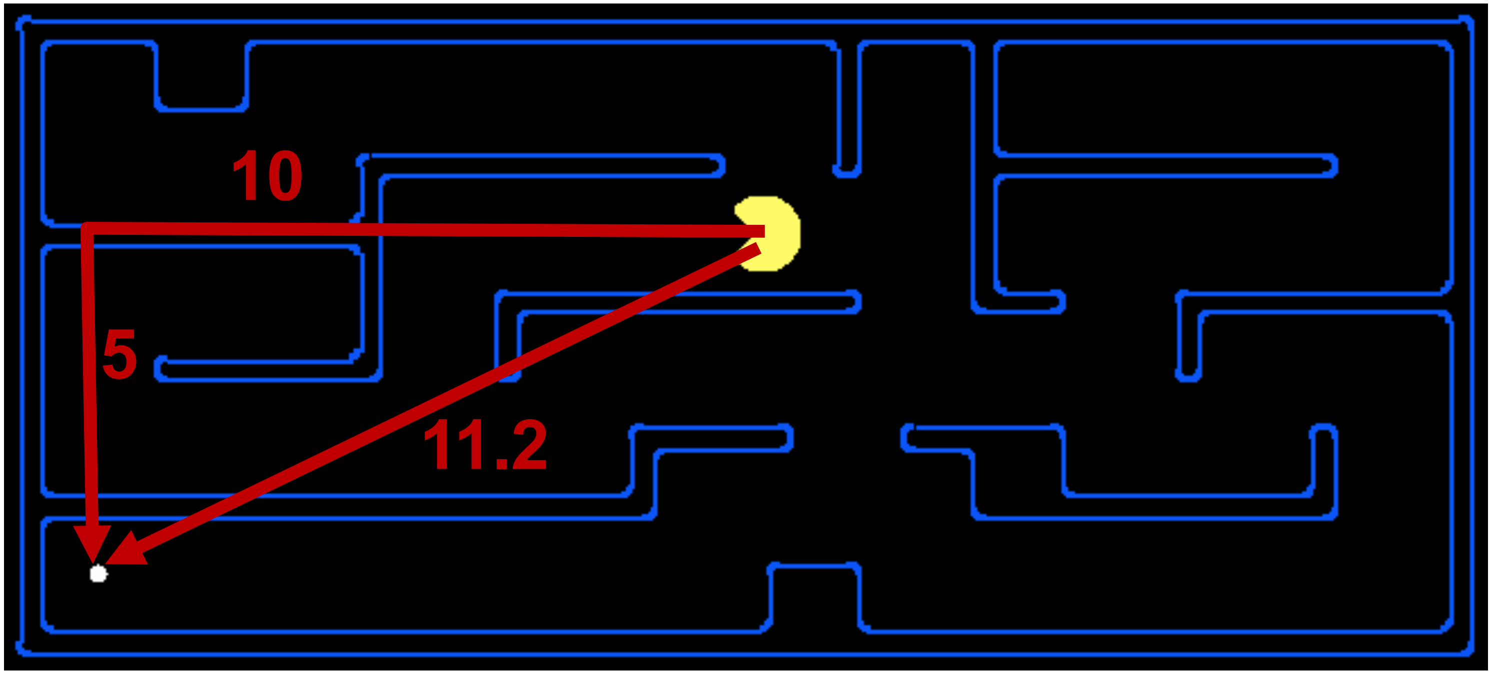
Uniform Cost: Pac-Man

- Cost of 1 for each action
- Explores all of the states, but one



What is a ***Heuristic***?

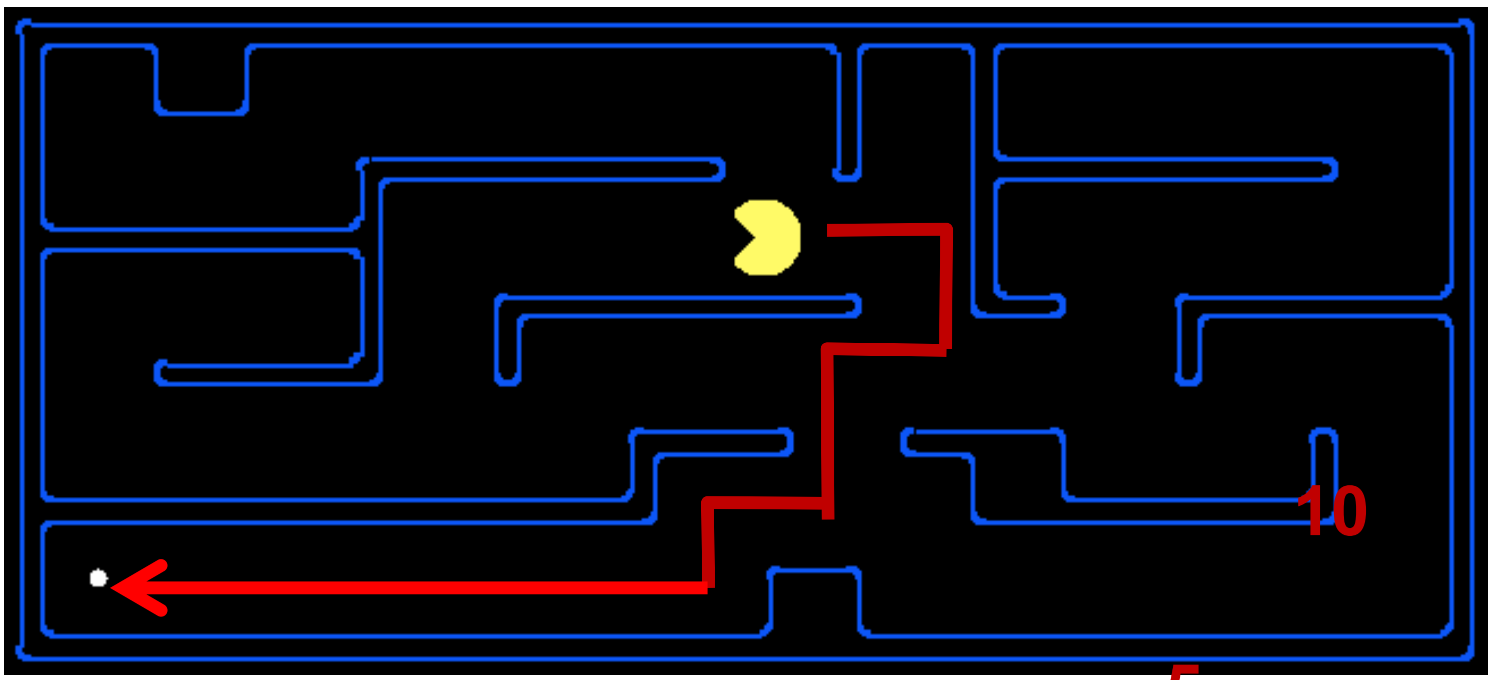
- An ***estimate*** of how close a state is to a goal
- Designed for a particular search problem



- Examples: Manhattan distance: 10+5 = 15
  Euclidean distance: 11.2

What is a ***Heuristic***?

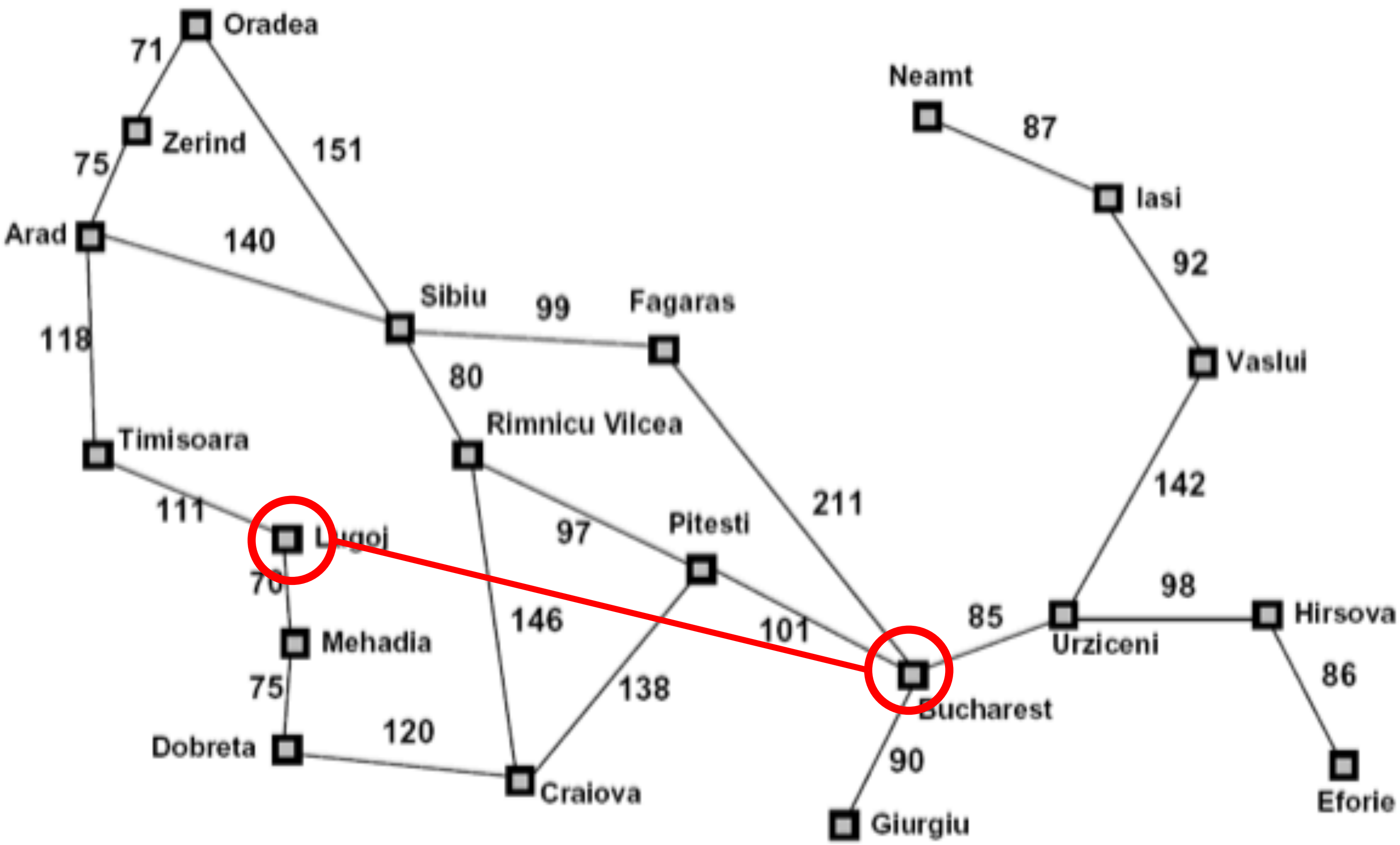
- An ***estimate*** of how close a state is to a goal
- Designed for a particular search problem



- Actual distance to goal: 2+4+2+1+8=

Greedy Search

Best first with f(n) = heuristic estimate of distance to goal

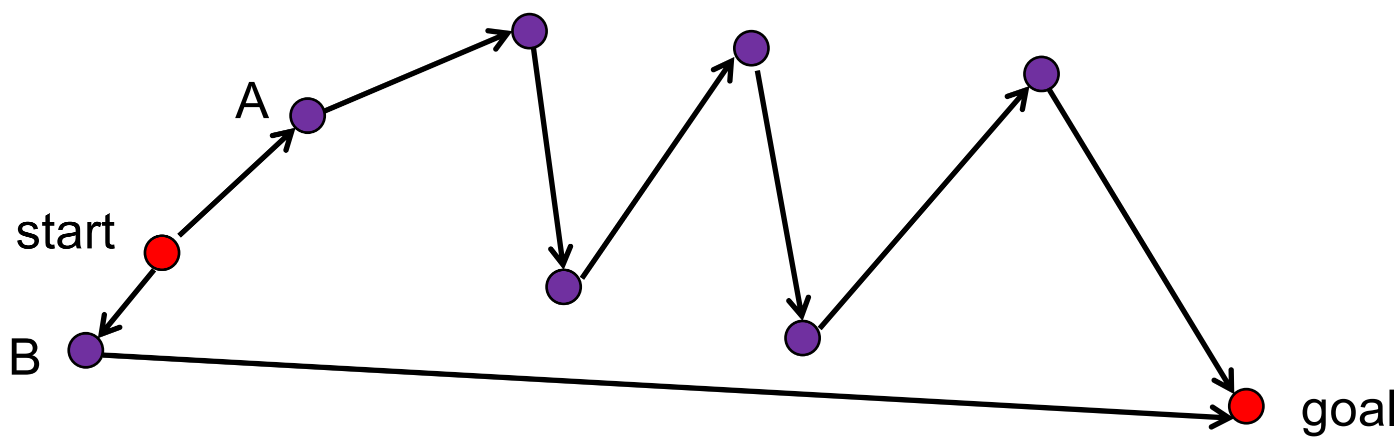


Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Search

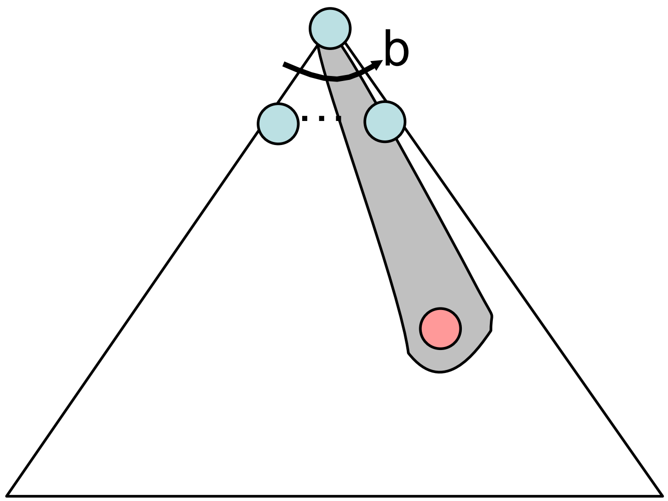
Expand the node that seems closest…

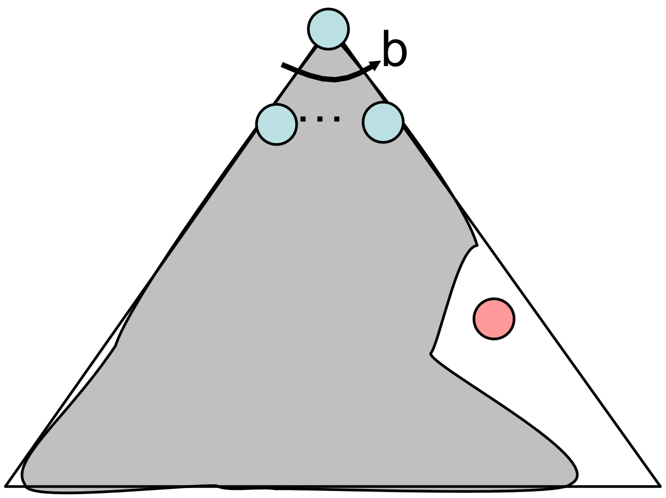


What can go wrong?

Greedy Search

- Common case:
  - Best-first takes you straight to a (suboptimal) goal
- Worst-case: like a badly-guided DFS
  - Can explore everything
  - Can get stuck in loops if no cycle checking
- Like DFS in completeness
  - Complete w/ cycle checking
  - *If* finite # states





A* Search

Hart, Nilsson & Rafael 1968

Best first search with $f(n) = g(n) + h(n)$

- $g(n)$ = sum of costs from start to n
- $h(n)$ = estimate of lowest cost path n $\rightarrow$ goal

  $h(goal) = 0$

Can view as cross-breed:

g(n) ~ uniform cost search

h(n) ~ greedy search

Best of both worlds…