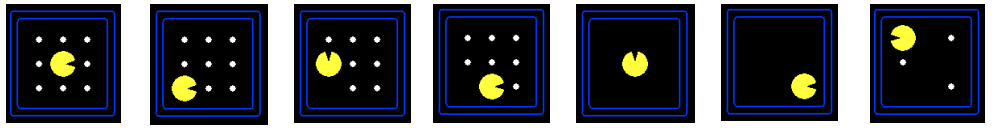
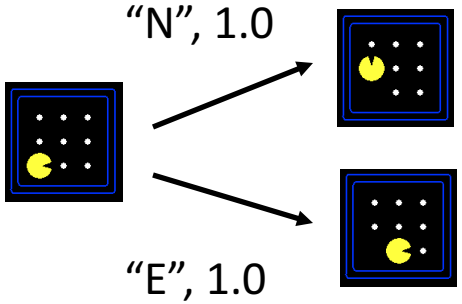
Homework 0 is due today

Search Problem Mechanics

- A search problem consists of:
  - A state space



  - A successor function (with actions, costs)

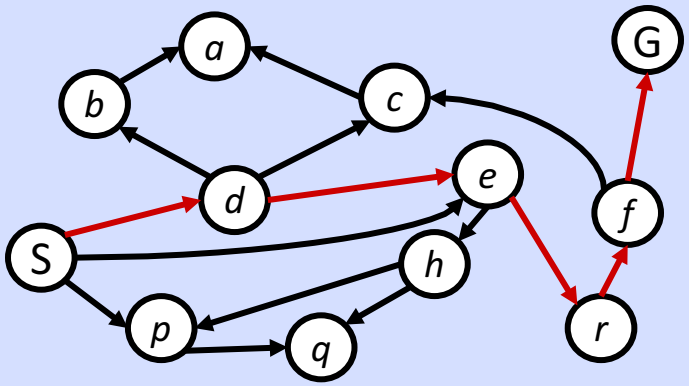


  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

State Space Graphs vs. Search Trees

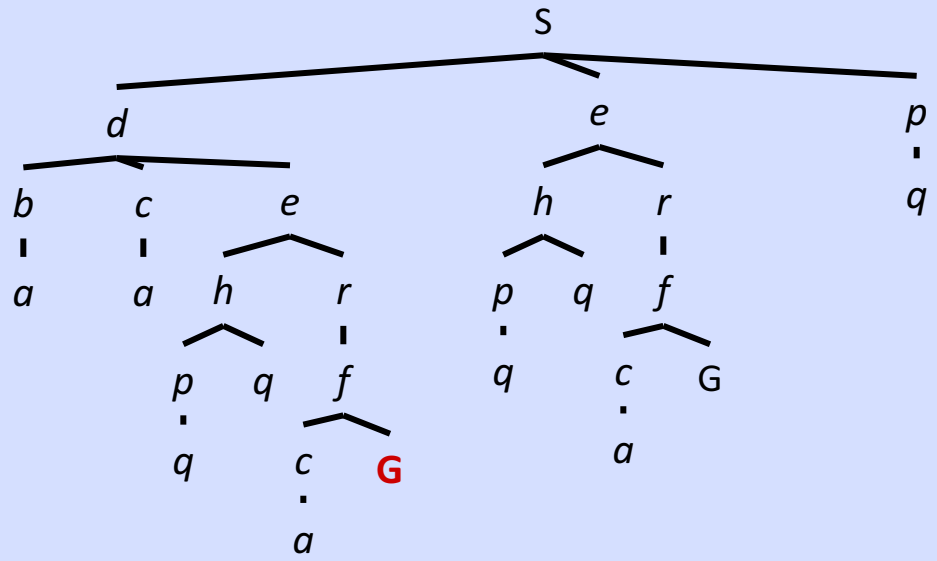
State Space Graph



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

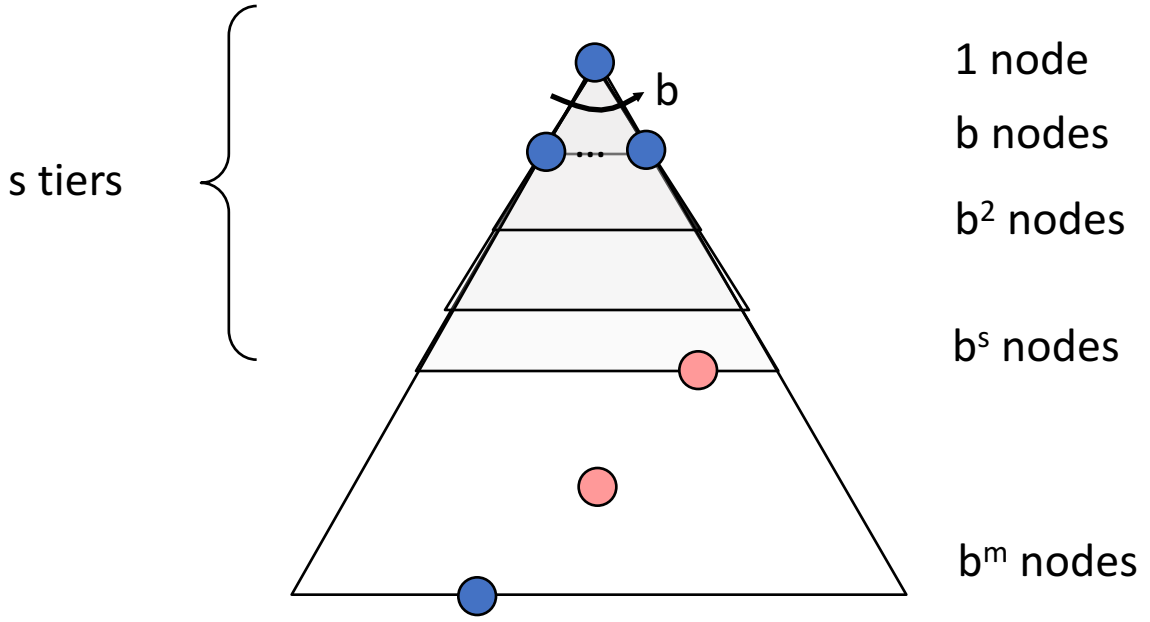
We construct both on demand – and we construct as little as possible.

Search Tree



Breadth-First Search (BFS) Properties

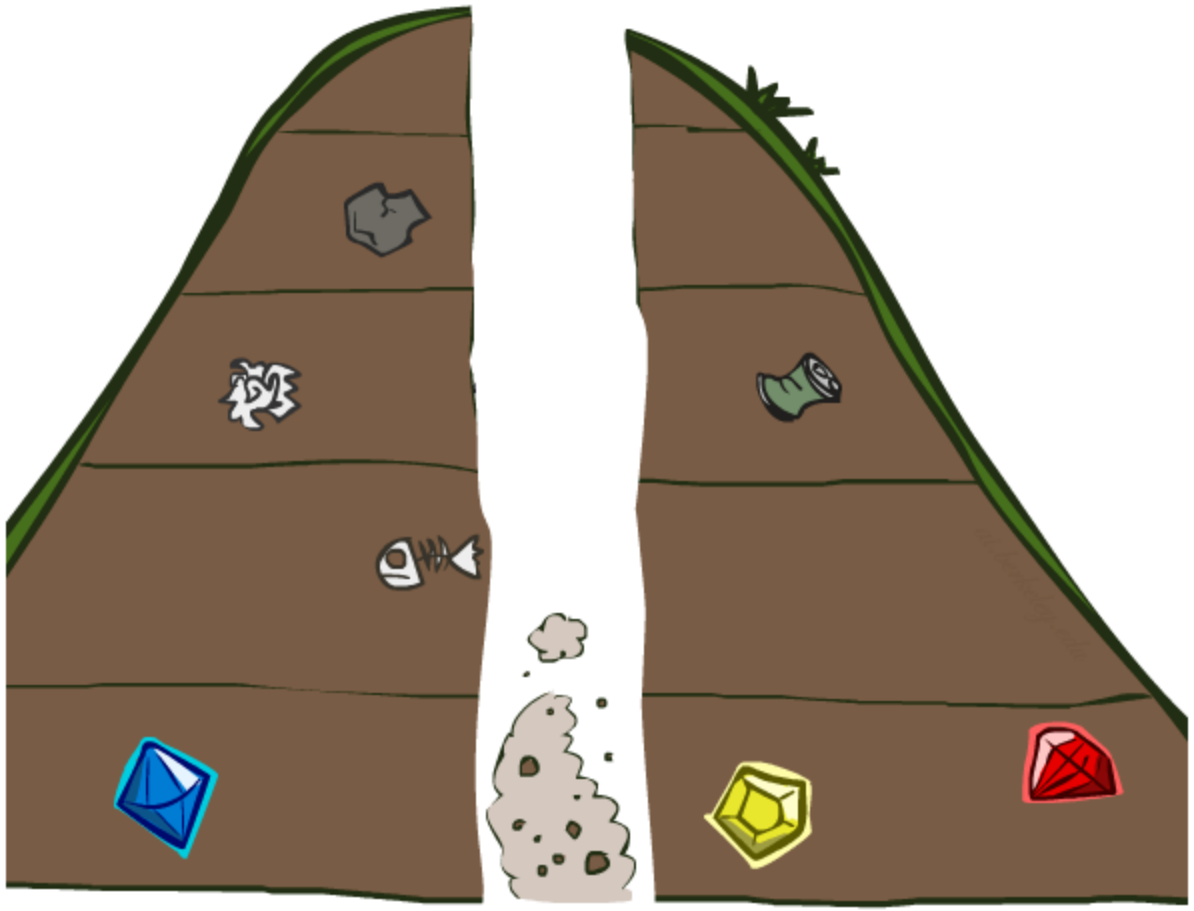
- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1 (more on costs later)



Python code for BFS

```
import collections

def bfs(graph, root):
    seen, queue = set([root]), collections.deque([root])
    while queue:
        vertex = queue.popleft()
        visit(vertex)
        for node in graph[vertex]:
            if node not in seen:
                seen.add(node)
                queue.append(node)

def visit(n):
    print(n)

if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2, 0], 2: []}
    bfs(graph, 0)
```

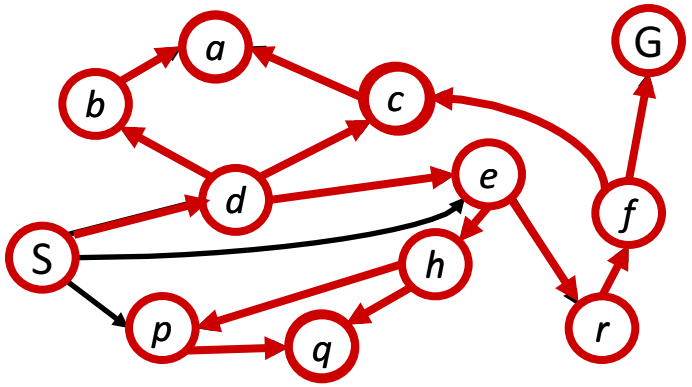
Depth-First Search

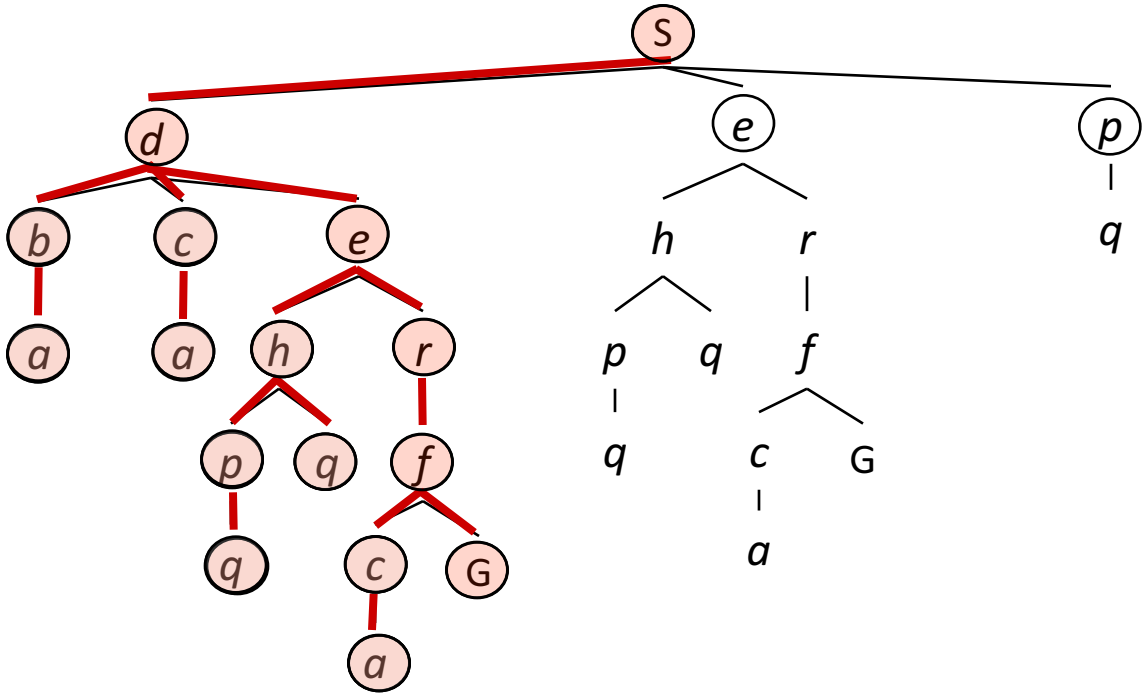


Depth-First Search

Strategy: expand a deepest node first

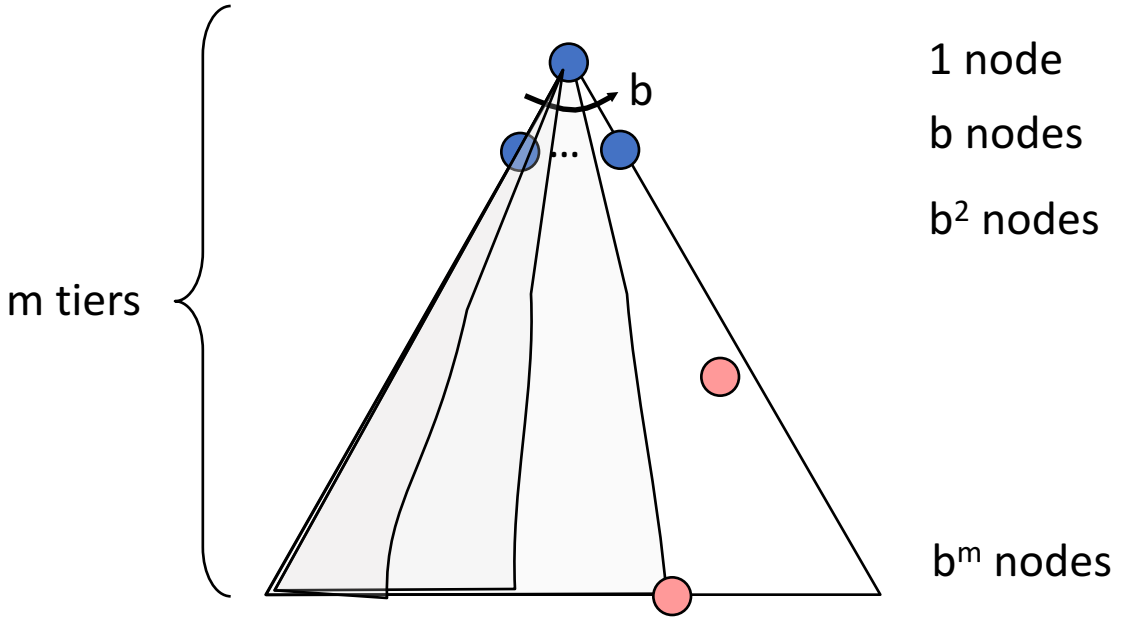
Implementation: Fringe is a LIFO stack





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 only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost



Python code for DFS

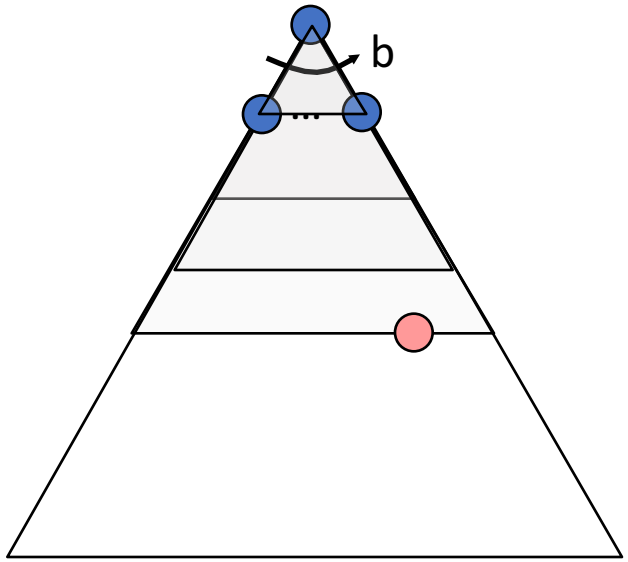
```
def dfs_recursive(graph, vertex, path=[]):
    path += [vertex]

    for neighbor in graph[vertex]:
        if neighbor not in path:
            path = dfs_recursive(graph, neighbor, path)

    return path


adjacency_matrix = {1: [2, 3], 2: [4, 5],
                    3: [5], 4: [6], 5: [6],
                    6: [7], 7: []}

print(dfs_recursive(adjacency_matrix, 1))
# [1, 2, 4, 6, 7, 5, 3]
```

DFS vs BFS

- If you know a solution is not far from the root of the tree, a breadth first search (BFS) might be better.
- If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster.
- If the tree is very wide, a BFS might need too much memory, so it might be completely impractical.
- If solutions are frequent but located deep in the tree, BFS could be impractical.
- If the search tree is very deep you will need to restrict the search depth for depth first search (DFS), anyway (for example with iterative deepening).

Iterative Deepening

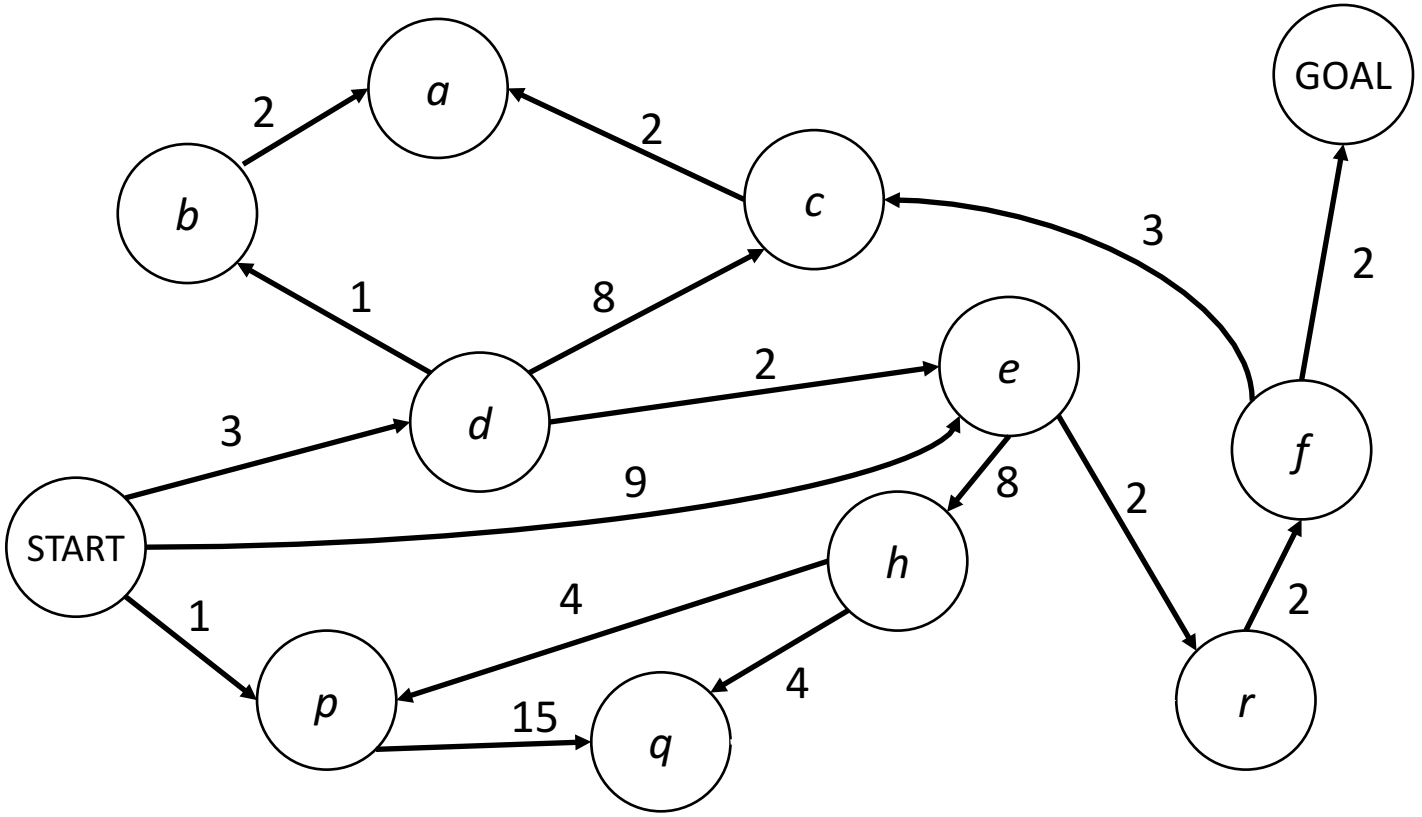
- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!



Python Code for Iterative Deepening

```
# A function to perform a Depth-Limited search
# from given source 'src'
def DLS(src,target,maxDepth):

    if src == target : return True

    # If reached the maximum depth, stop recursing.
    if maxDepth <= 0 : return False

    # Recur for all the vertices adjacent to this vertex
    for i in graph[src]:
            if(DLS(i,target,maxDepth-1)):
                return True
    return False

# IDDFS to search if target is reachable from v.
# It uses recursive DLS()
def IDDFS(src,target, maxDepth):

    # Repeatedly depth-limit search till the
    # maximum depth
    for i in range(maxDepth):
        if (DLS(src, target, i)):
            return True
    return False
```

Cost-Sensitive Search

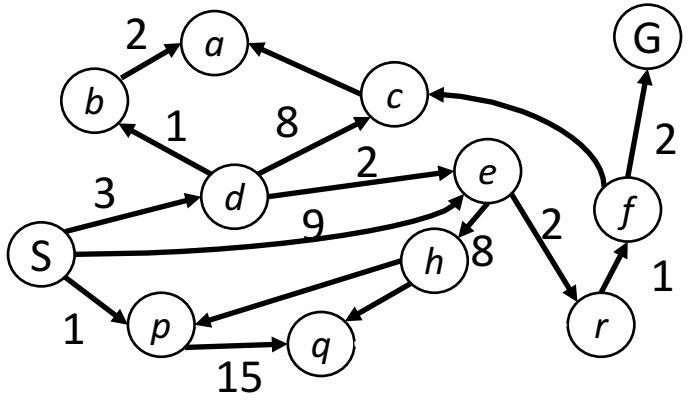


BFS finds the shortest path in terms of number of actions.
It does not find the least-cost path. We will now cover
a similar algorithm which does find the least-cost path.

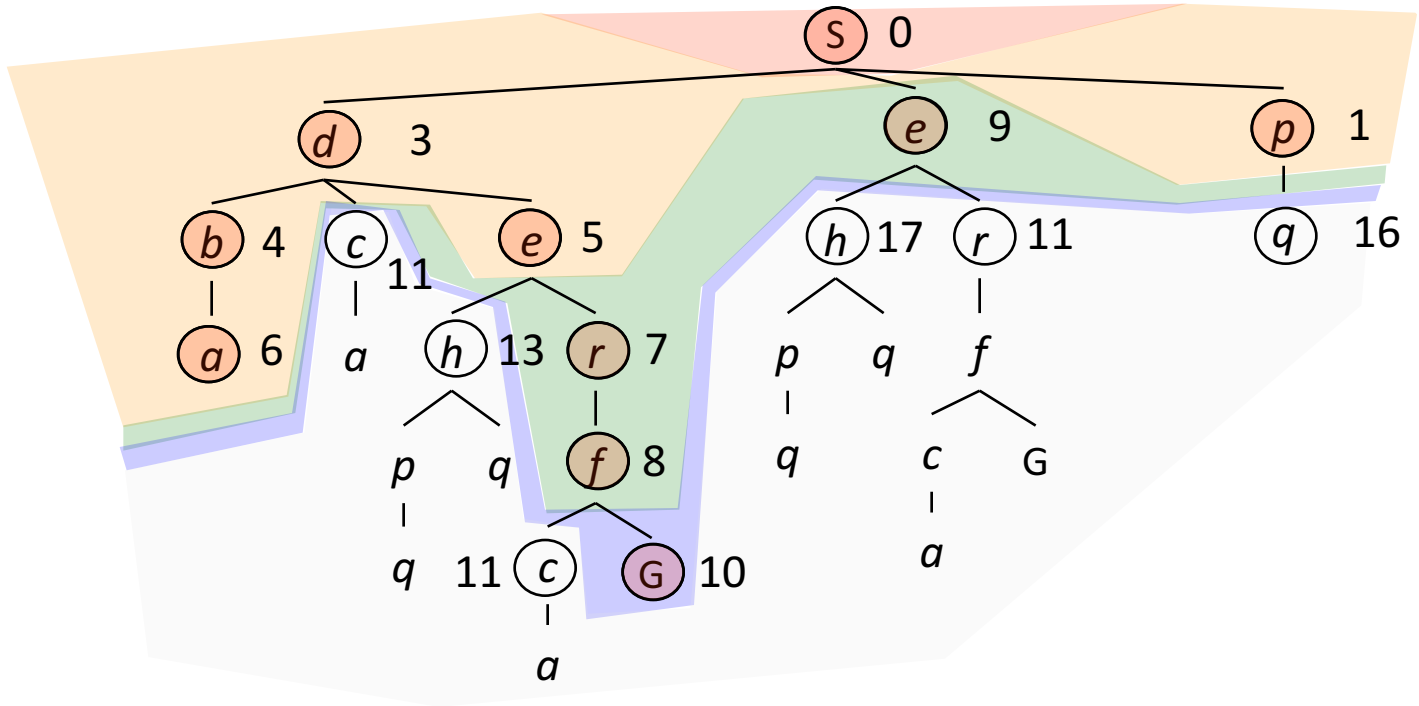
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)

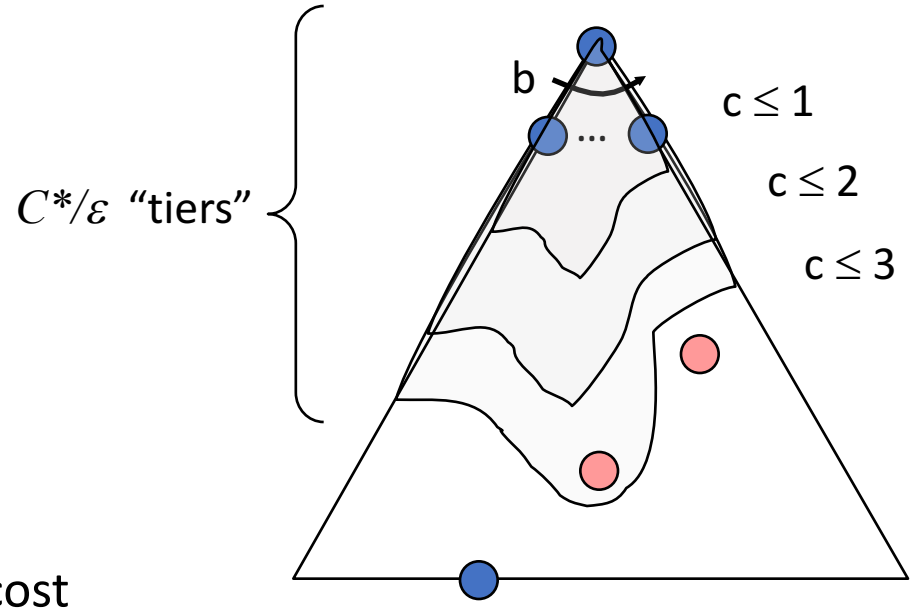


Cost contours



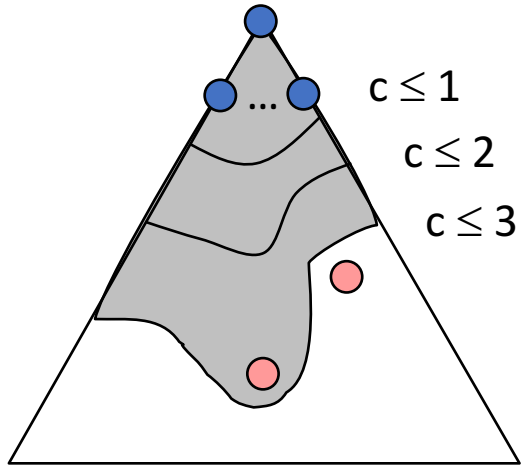
Uniform Cost Search (UCS) Properties

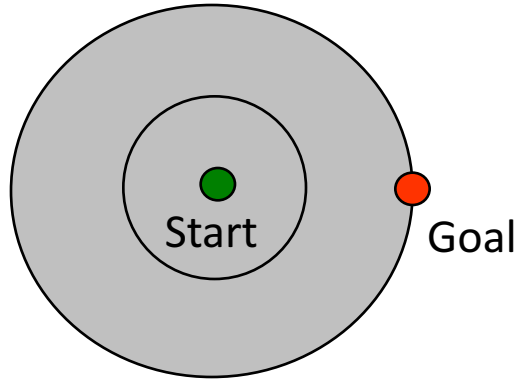
- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (skipping the proof for now)



Uniform Cost Issues

- Remember: UCS explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every “direction”
  - No information about goal location
- We’ll fix that soon!





Next time

Informed search methods