COMS W4701: Artificial Intelligence

Lecture 2a: Uninformed Search

Tony Dear, Ph.D.

Department of Computer Science

School of Engineering and Applied Sciences

Today

- State space graphs and search trees
- Uninformed search: DFS, BFS, UCS
- Variants: Iterative deepening, branch-and-bound

Search Problems

- State space S: Set of descriptions of the agent and environment
- Initial state and one or more goal states

- Goal test $S \rightarrow \{True, False\}$, e.g., $isGoal(s_1) = False$
- Action set for each state, e.g., $Actions(s_1) = \{a_1, a_2, a_3\}$

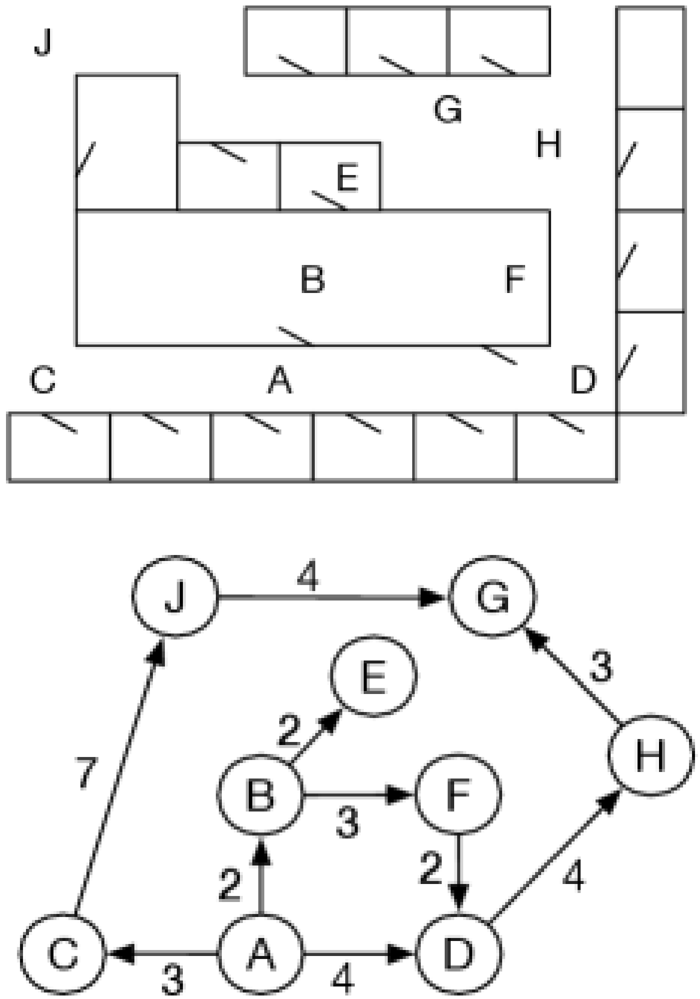
- Transition model (function) $S \times A \rightarrow S$, e.g., $Result(s_1, a_1) = s_2$
- State-action cost function $S \times A \rightarrow \mathbb{R}$, e.g., $Cost(s_1, a_1\ ) = 10$

State Space Graphs

- State space can be *abstracted* as a **directed graph**
- States represented as vertices/nodes

- Transitions represented as edges/arcs
- Costs represented by edge weights

- A solution is a *path* from initial to a goal state
- State spaces can be very large or infinite, so we rarely build or store the full graph in memory

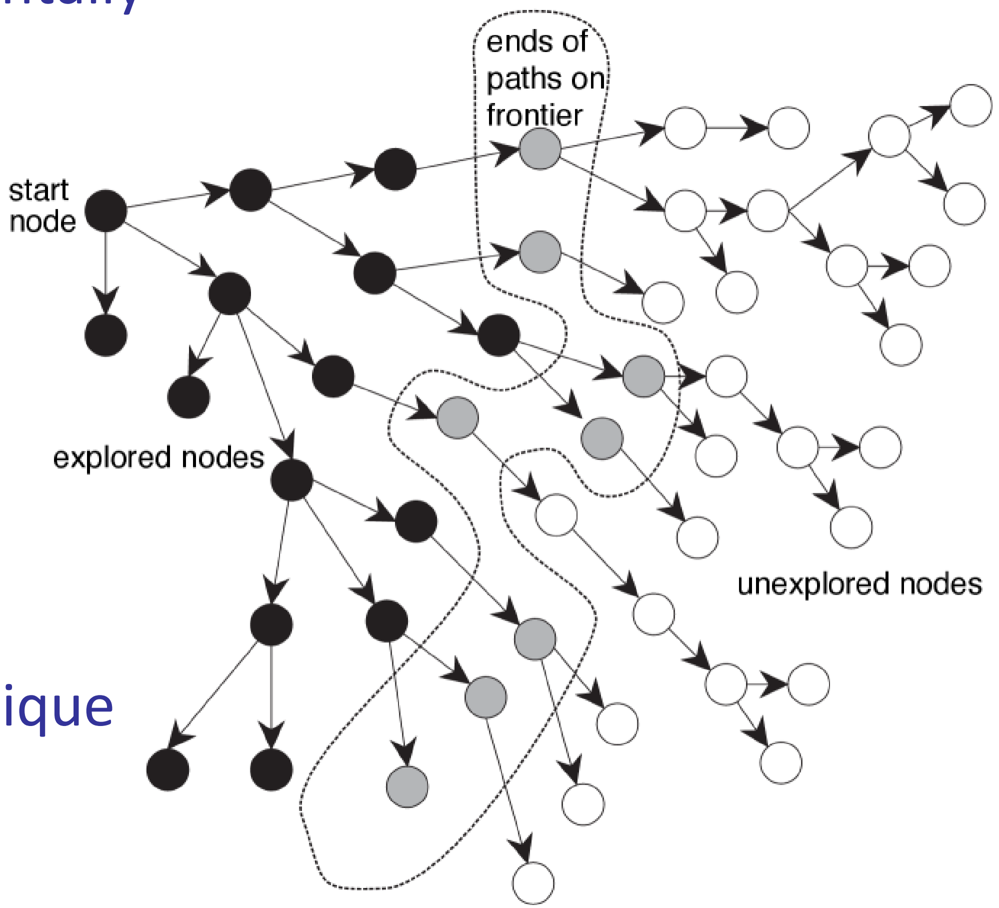


Search Trees

- Most search algorithms explore paths by incrementally constructing a **search tree** from initial state

- Internal nodes have already been *explored*
- Leaf nodes lie in a **frontier** data structure
- Frontier nodes are candidates for *expansion*

- Important: Unexplored nodes are *not* in memory
- For explored nodes, tree structure records the unique path to each one from initial state



Node Expansion

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

- For each node that we explore in the search graph/tree, we *create* a new node for each *successor* and place them all in the frontier
- Can do so by looping over all available actions and looking at the results

- Also compute the *cumulative cost* of the new node—parent node cost + action cost
- Create successor node storing corresponding state, parent node, action, and cost

Search Implementation

- We can now come up with a general *best-first-search* procedure
- Main idea: Repeatedly pop nodes out of and insert new nodes into frontier
- Run until we either find a goal or we run out of nodes to expand

```
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
  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
        add child to frontier
  return failure
```

Initialize current node to contain the initial state, frontier as a queue data structure

“Late” goal test (upon popping from frontier)

Node expansion

Frontier and Reached States

- The *search strategy* determines the ordering for popping frontier nodes
- Different strategies yield different efficiencies and guarantees

- Frontier implementation examples: LIFO (stack), FIFO (queue)
- Priority queue using general **evaluation function** $f(n)$

- We may also want to *prune* the search space to make it smaller
- There may be multiple paths to a state; should check whether a state has already been *reached* before placing into frontier

Search Implementation with Reached

```
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
```

reached may be implemented as a lookup table (dictionary) mapping states to nodes

When adding a node to frontier, also add its state to *reached*

Two scenarios for adding node to frontier:

1. Previously unencountered (not in reached)
2. Already encountered, BUT found cheaper path

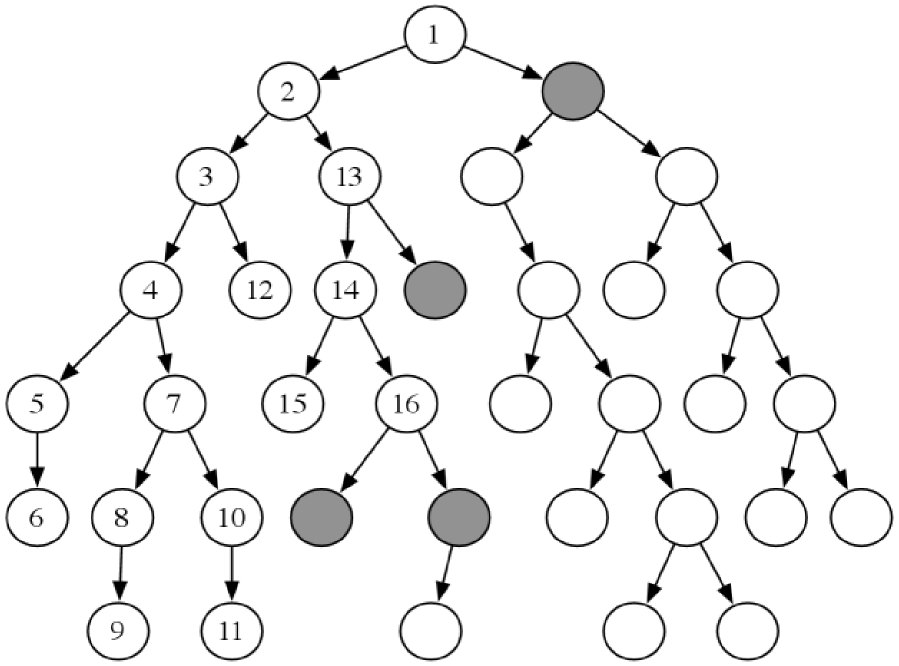
Depth-First Search

- Idea: Expand the *deepest* node in the frontier, disregarding action costs
- Implement frontier as a stack (LIFO) or priority queue where $f(n) = -depth(n)$

- Behavior ends up searching each path to completion
- If no goal, *backtrack* and proceed along a new path

- DFS is not **optimal**: no guarantee of least-cost solution

- DFS is not **complete**: no guarantee of finding a solution if one exists or correctly reporting failure otherwise

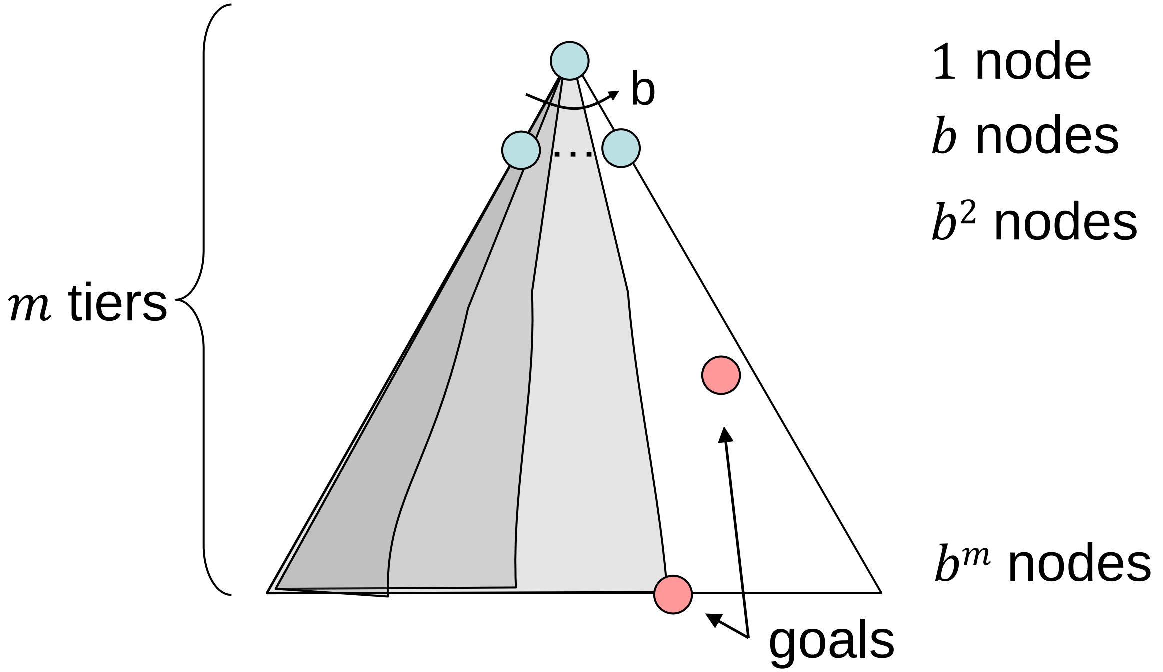


DFS Complexity

- Suppose search tree has *branching factor* b and *max depth* m
- **Time complexity:** (Worst case) number of expanded nodes $O(b^m)$

- **Space complexity**: (Worst case) number of frontier nodes in memory $O(bm)$
  - One node in each of m tiers with b successors each

- Good use cases for DFS when space is restricted, or many goals exist and we just want to find one fast

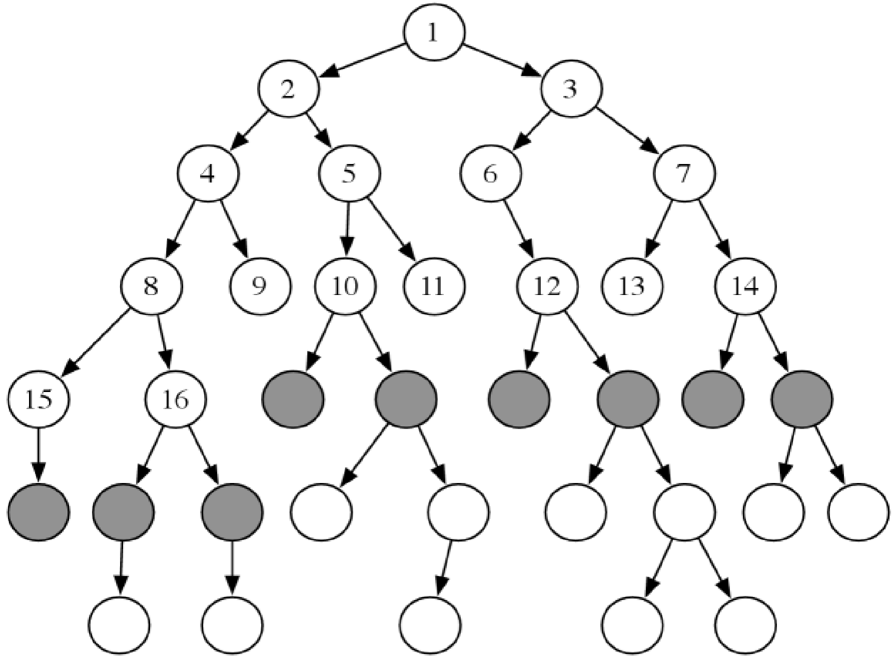


Breadth-First Search

- Idea: Expand the *shallowest* node in the frontier, disregarding action costs
- Implement frontier as a FIFO queue or priority queue where $f(n) = depth(n)$

- BFS is **complete**; will eventually find a goal if one exists
- BFS returns shallowest solution (e.g., at depth d)
- **Optimal** if action costs are uniform

- Worst-case time and space complexities are $O(b^d)$
- Good use cases for BFS when problem / state space are small, or solution is close to start



Comparison: DFS and BFS

- DFS and BFS are both implementations of best-first search
- Frontier nodes are popped out *deepest* and *shallowest* first, respectively

- Both have exponential time complexity; BFS has exponential space complexity while DFS space is linear in search tree depth
- Neither is optimal in general; BFS only guaranteed to return shallowest solution

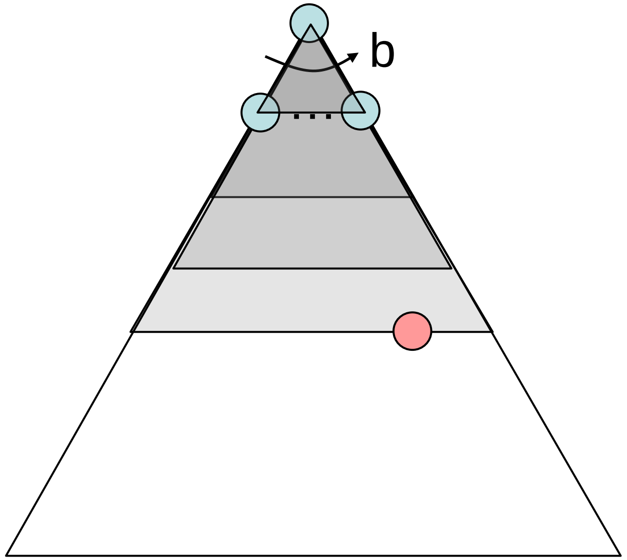
- Small optimization: "Early" goal test
- Check upon *insertion* into frontier

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**
    if *problem*.IS-GOAL(*child*.STATE) **then return** *child*

Iterative Deepening Search

- **Depth-limited DFS**: Prevent DFS from going past a set depth l
- Time complexity $O(b^l)$, space complexity $O(bl)$
- Not complete: Will miss the goal if l is set too small

- **Iterative-deepening:** Iteratively do depth-limited search
  - Try $l = 0$, then $l = 1$, …
- Ends when l reaches shallowest solution at depth d
- Space complexity is $O(bd)$, and strategy is now complete



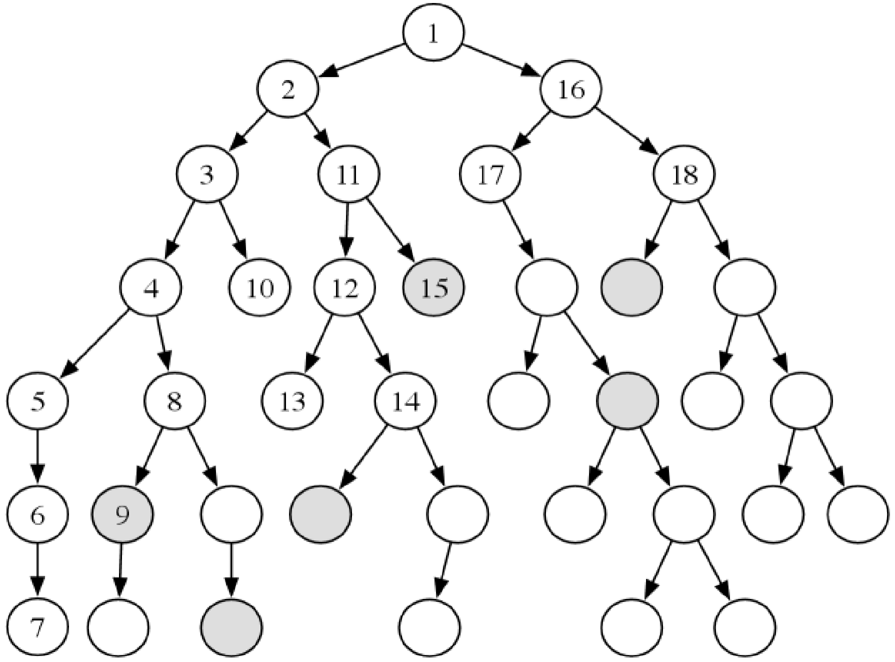
- More nodes expanded than depth-limited DFS, but time complexity is still $O(b^d)$!
- Shallow depths are searched multiple times, but lowest levels still dominate

Branch-and-Bound

- We can also set an *adaptive* depth limit based on best solution found so far
- Best use case when there are multiple solutions and we want to find all of them

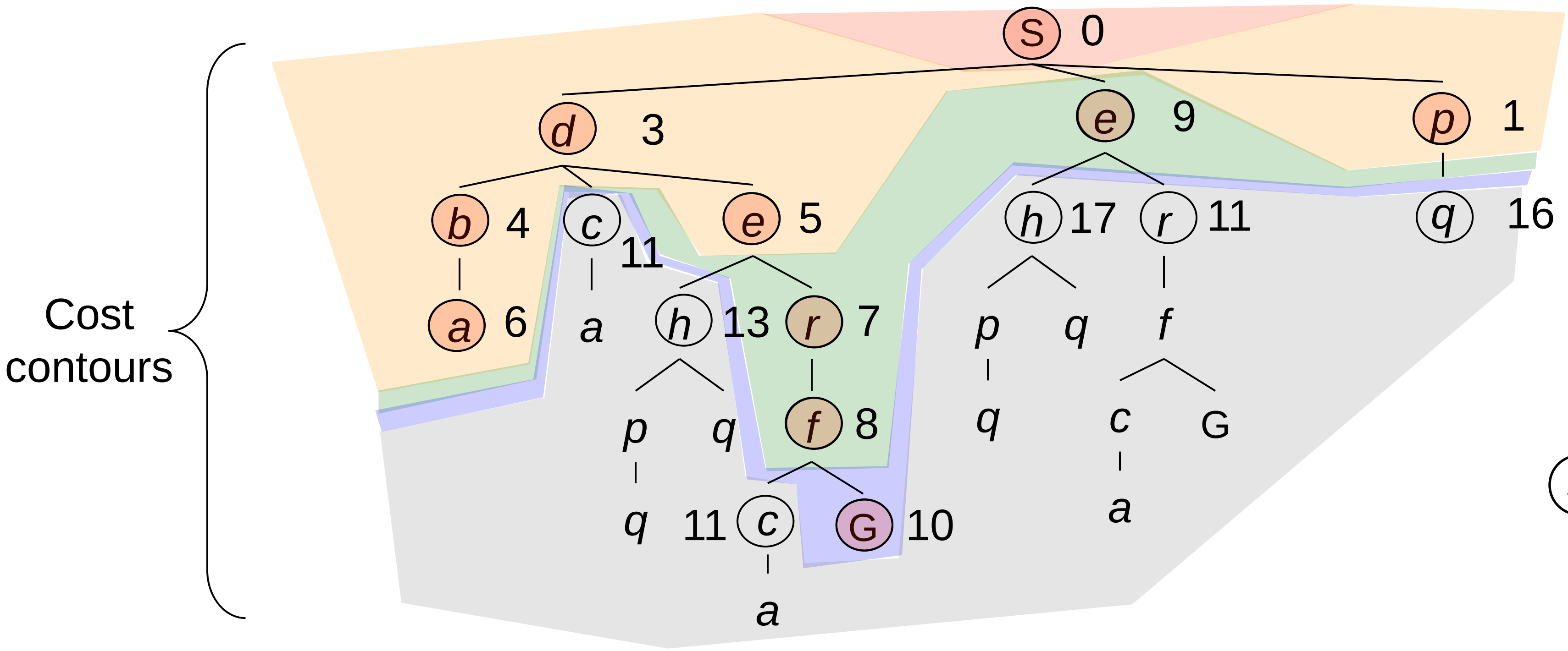
- First run DFS as usual until a goal G is found
- Save the solution and continue DFS, but now with depth or cost limit equal to that of G

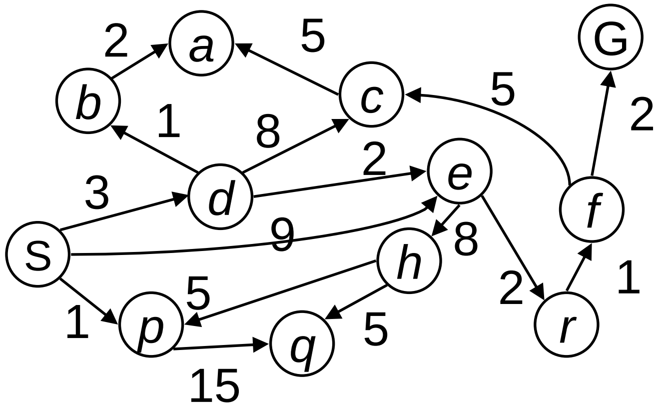
- If we find better solution, save it and update the bound
- Search until frontier is empty and return best solution
- Can also be combined with iterative deepening!



Uniform-Cost Search (Dijkstra)

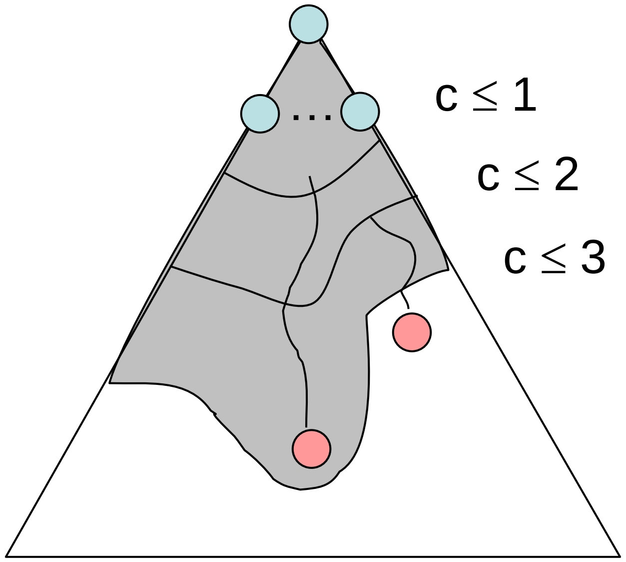
- To find an optimal solution, we expand nodes in order of *cumulative path cost*
- Implement frontier as priority queue with evaluation function $f(n) = cost(n)$
- First goal that is found is guaranteed to be the one with lowest cost

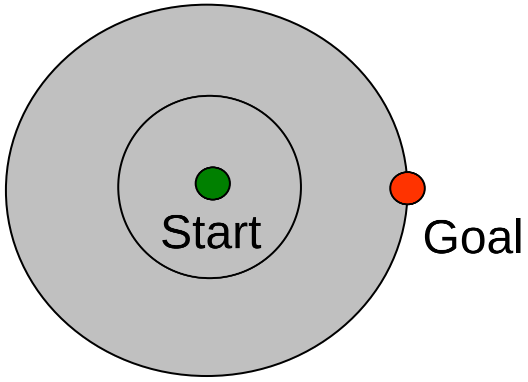




UCS Properties

- Let C^* be the cost of optimal solution
- Let ϵ be lower bound on all possible costs
- $1 + \lfloor C^*/\epsilon \rfloor$ is the max depth to traverse before finding optimal solution
- **Time and space complexity**: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- UCS is both **complete** and **optimal**





Summary

- Problem-solving agents solve search problems to find discrete sequences of states and actions between initial and goal states
- Search problems represented as graphs and/or trees, which we try to represent and search in a systematic and efficient manner
- Various search strategies, each with different tradeoffs in time complexity, space complexity, completeness, optimality
- Algorithms: DFS, BFS, iterative deepening, branch-and-bound, UCS