Chapter 3: Solving Problems by Searching

UNIFORMED SEARCH

THESE SLIDES ARE ADOPTED FROM BERKELEY COURSE MATERIALS AND RUSSELL AND NORVIG TEXTBOOK

Chapter 2 Summary

- Agents interact with environments through actuators and sensors
  - The agent function describes what the agent does in all circumstances
  - The agent program calculates the agent function

- A perfectly rational agent maximizes expected performance

- PEAS descriptions define task environments

- Environments are categorized along several dimensions:
  - Observable? Deterministic? Episodic? Static? Discrete? Single-agent?

Problem-Solving Agents

- Goal-based agents versus reflex agents
  - Reflex agents base their actions on a direct mapping from states to actions.
  - Goal-based agents consider the future actions and their outcomes to achieve a certain goal.
- Problem-solving agent is a goal-based agent that consider atomic representation for the world states.

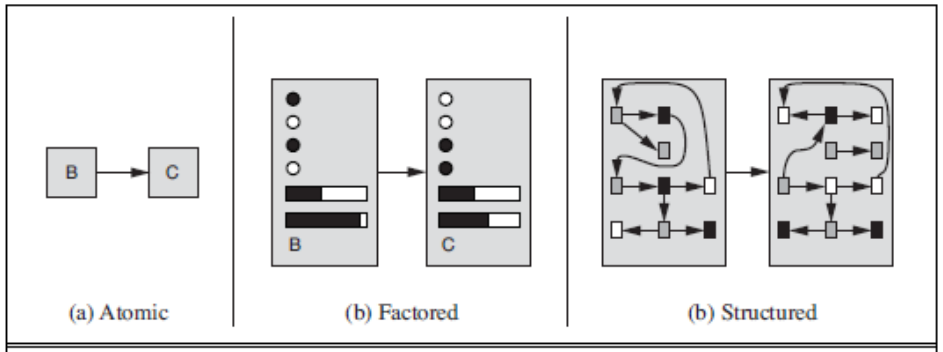


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

Problem-Solving Agents

- Goal formulation.
- Problem formulation.
- We assume observable, deterministic, discrete, and known environments.
- The solution to a search problem would be a sequence of actions.
- Then, the agent can execute the solution.
- "Formulate- Search- Execute" agent.

Problem-Solving Agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq); seq ← REST(seq)
    return action
```

This is the hard part!

This offline problem solving!

Solution is executed “eyes closed.

Problem formulation

- The goal of this agent is to travel from Arad to Bucharest.
- **Initial state**

The state where the agent starts. In(Arad)

- **Actions(s)**

Possible actions available to the agent at state s. From Arad: Go(Sibiu), Go(Zerind), Go(Timisoara)

- **Transition model (successor function):** A description of what each action does.

Results(s,a) returns the state resulting from applying action a at state s.

Result(In(Arad),Go(Zerind))=In(Zerind)

Goal test: Determines if a certain state is a goal state. In(Bucharest)

Path cost function: Assigns a cost to a path. Summation of step costs of actions along the path.

Step cost: c(s,a,s')

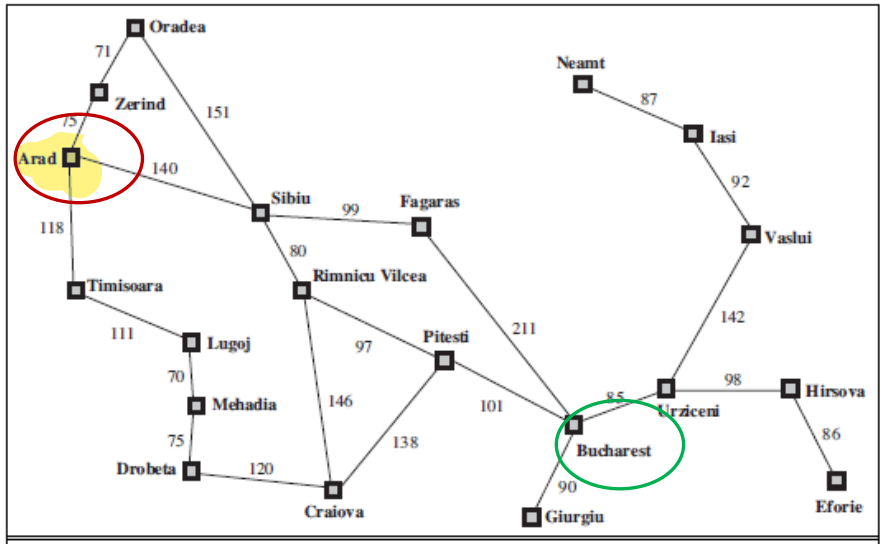


Figure 3.2 A simplified road map of part of Romania.

Problem formulation (cont.)

- **Solution**

A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function.

- **Optimal Solution**

The **optimal solution** has the lowest path cost among all solutions.

Abstraction

- Abstraction is removing details from a representation.
- For the trip from Arad to Bucharest, compare the previous problem formulation to the real world states.
- The real world state includes so many things: the traveling companions, the current radio program, the scenery out of the window, the distance to the next rest stop, the condition of the road, the weather, and so on.

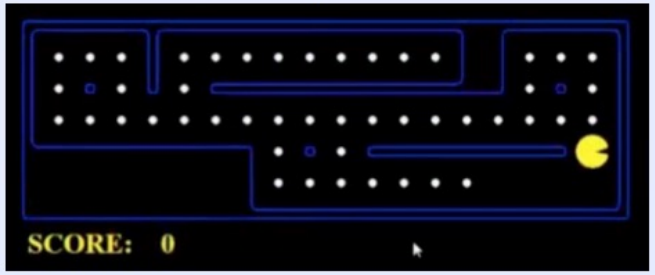
- Abstract the actions also!

  Omit all irrelevant actions: turning on the radio, looking out of the window, and so on.

  And do not specify actions at the level of “turn steering wheel to the left by one degree”.

Abstraction- example

The world state includes every last detail of the environment



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

Assume no ghosts for the two problems.

Problem: Pathing

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

Problem: Eat-All-Dots

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

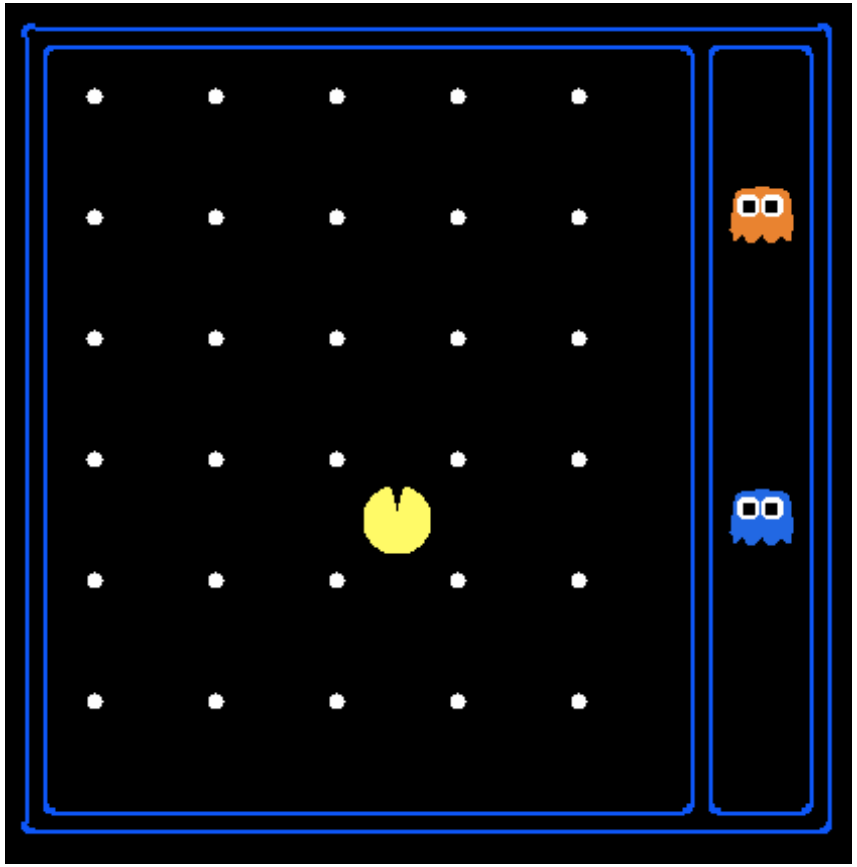
State Space Sizes?

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?
  $120x(2^{30})$



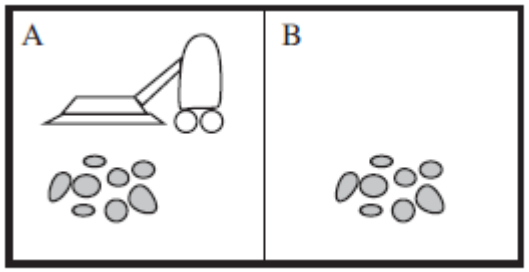
Toy Problems: Vaccum Cleaner

States: The state is determined by both the agent location and the dirt locations.

The agent is in one of two locations, each of which might or might not contain dir

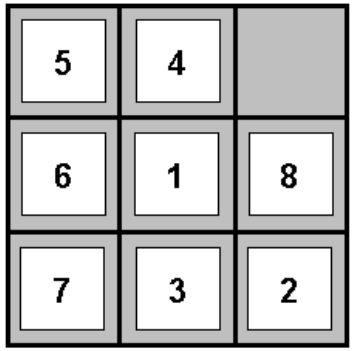
Thus, there are $2 \times 2^2 = 8$ possible world states.

A larger environment with n locations has $n \cdot 2^n$ states.

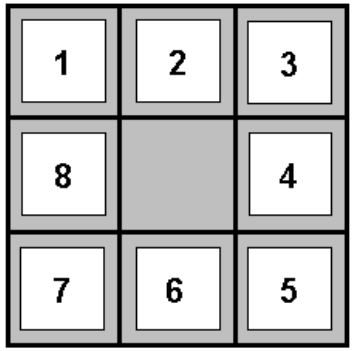


- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*.
- **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Toy Problems: 8-puzzle problem



Start State



Goal State

Initial State: The location of each of the 8 tiles in one of the nine squares

Operators: blank moves (1) Left (2) Right (3) Up (4) Down

Goal Test: state matches the goal configuration

Path cost: each step costs 1, total path cost = no. of steps

Toy Problems: 8-queens problem

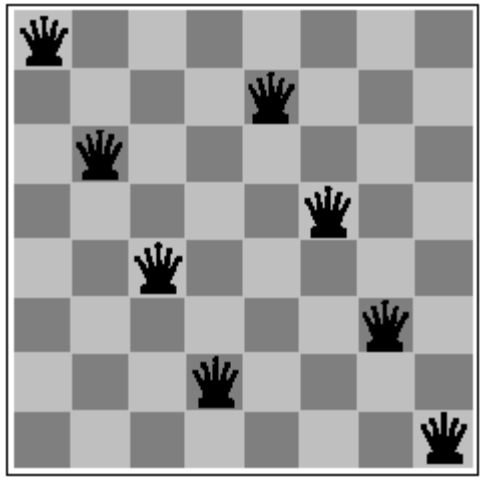


Fig 4.7

Initial State: Any arrangement of 0 to 8 queens on board.

Operators: add a queen to any square.

Goal Test: 8 queens on board, none attacked.

Path cost: not applicable or Zero (because only the final state counts, search cost might be of interest).

Real-world Problems

- **Route Finding** - computer networks, automated travel advisory systems, airline travel planning.
- **VLSI Layout** - A typical VLSI chip can have as many as a million gates, and the positioning and connections of every gate are crucial to the successful operation of the chip.
- **Robot Navigation -** rescue operations
- **Mars Pathfinder** - search for Martians or signs of intelligent lifeforms
- **Time/Exam Tables**

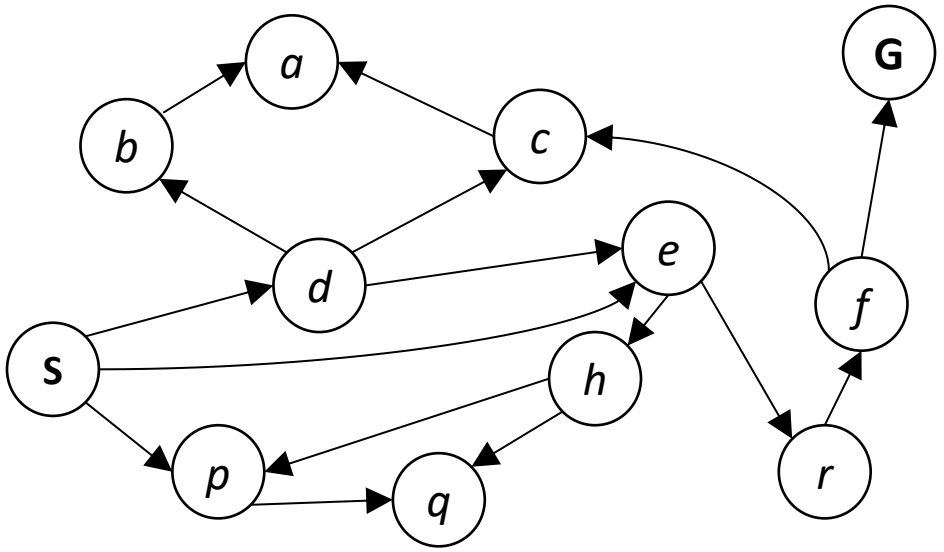
State Space Graphs

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)

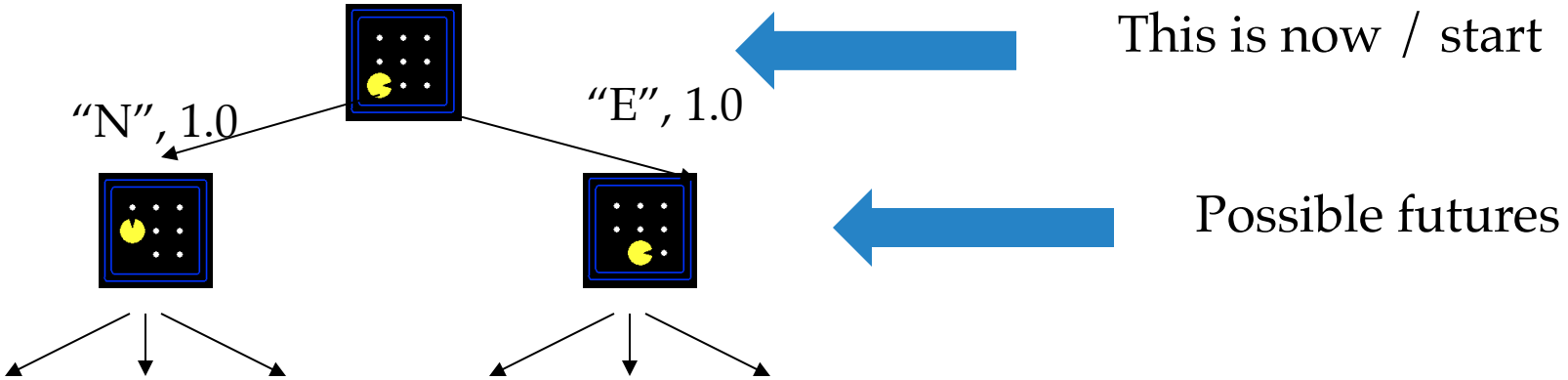
In a search graph, each state occurs only once!

We can rarely build this full graph in memory (it's too big), but it's a useful idea



Tiny search graph for a tiny search problem

Search Trees

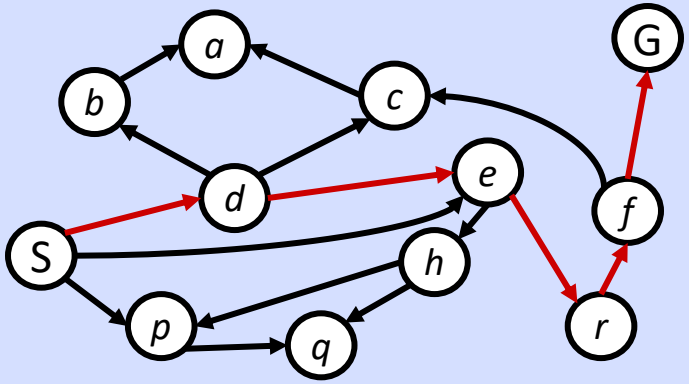


A search tree:

- A "what if" tree of plans and their outcomes
- The initial sate 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

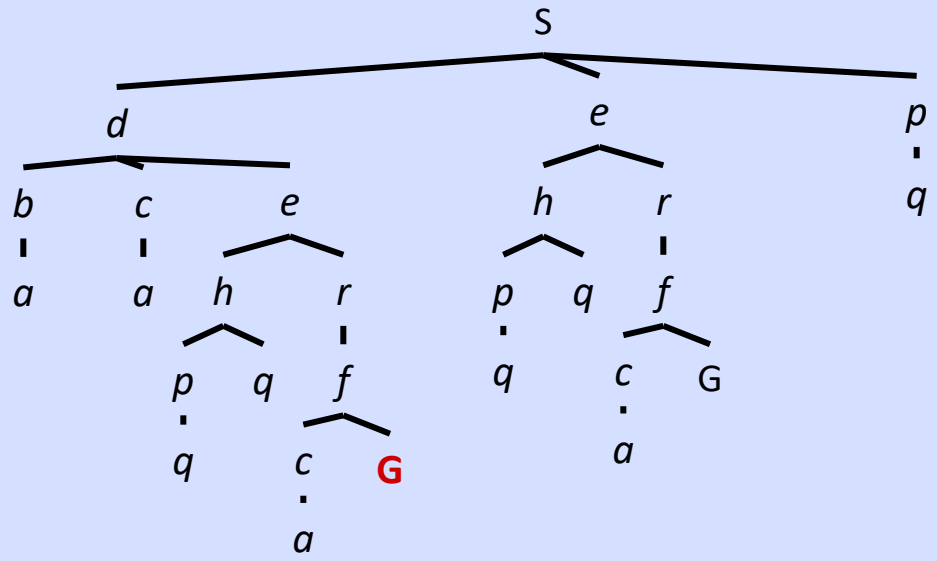
State Space Graph



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

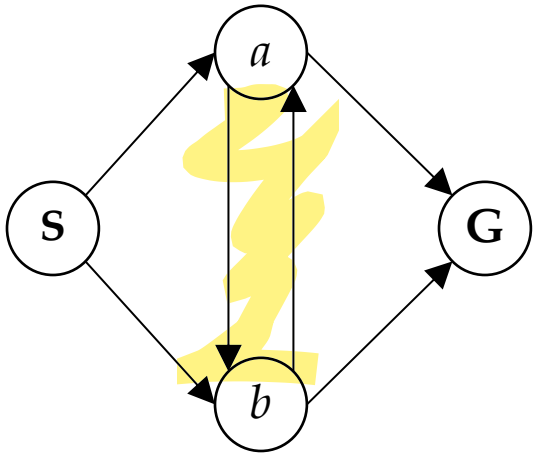
We construct both on demand – and we construct as little as possible.

Search Tree

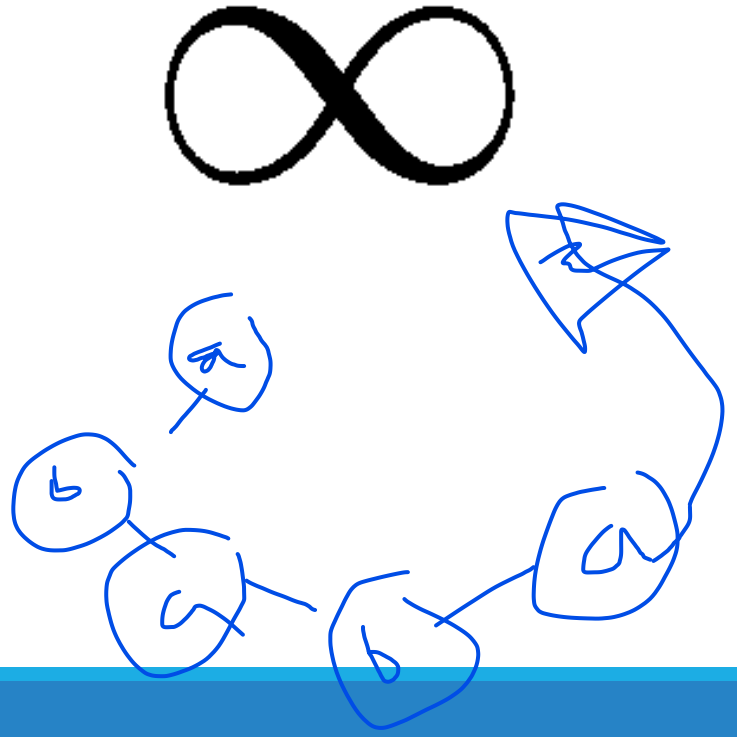


State Space Graphs vs. Search Trees

Consider this 4-state graph:



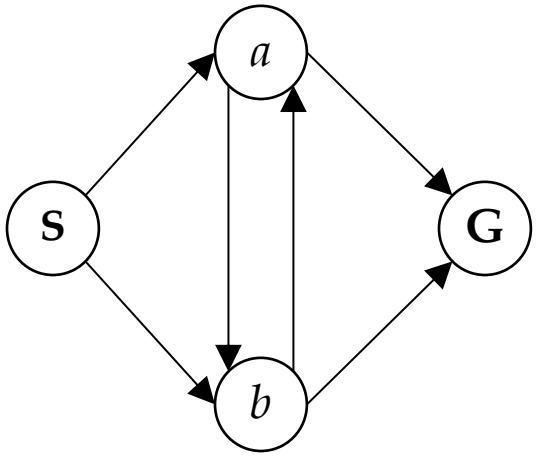
How big is its search tree (from S)?

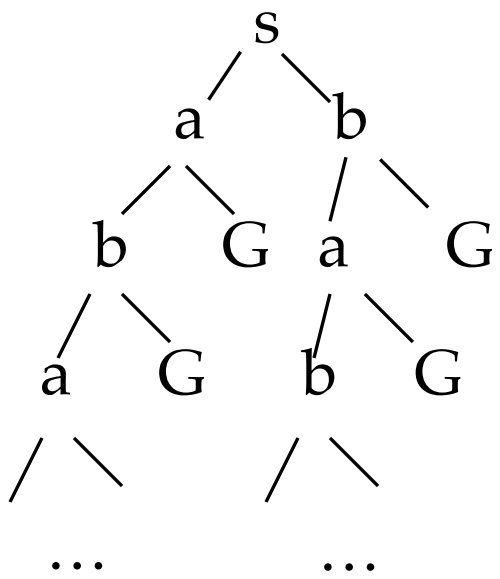


State Space Graphs vs. Search Trees

Consider this 4-state graph:

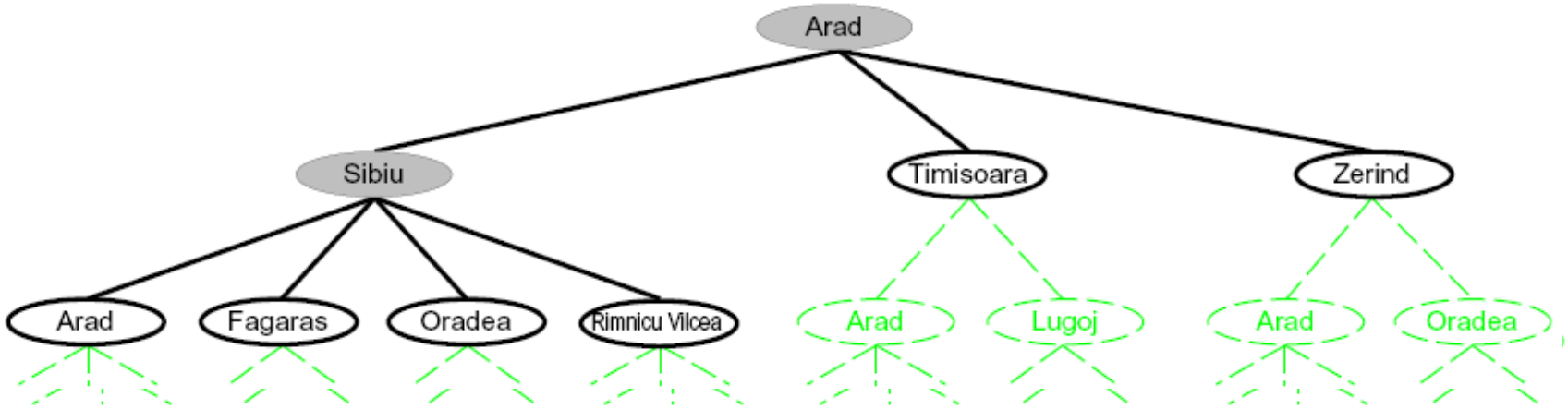
How big is its search tree (from S)?





Important: Lots of repeated structure in the search tree!

Searching with a Search Tree



Search:

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

General Tree Search

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

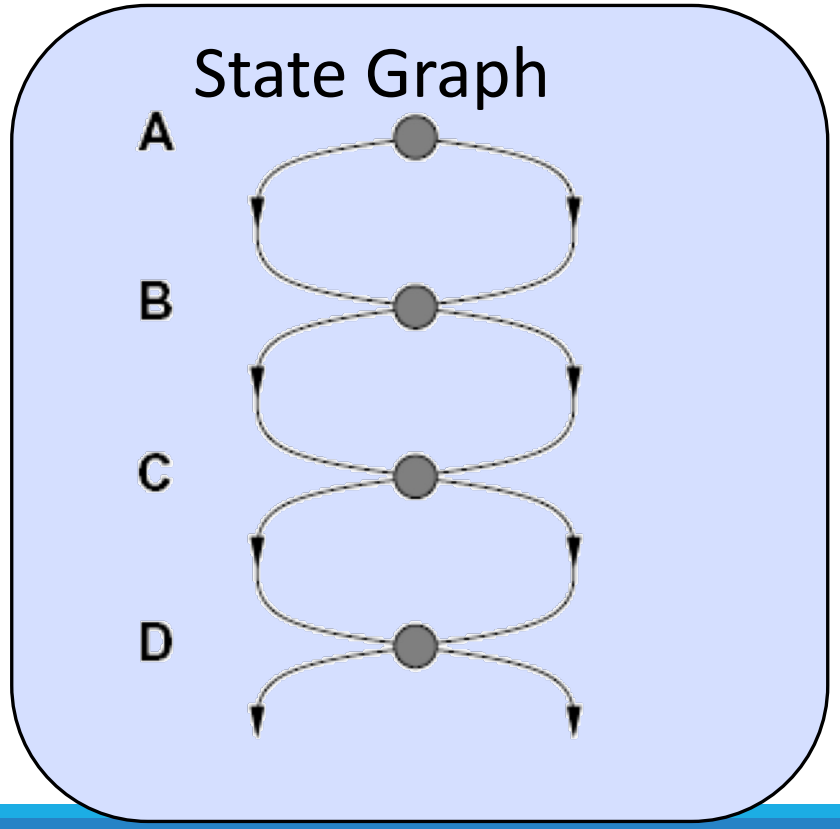
Important ideas:

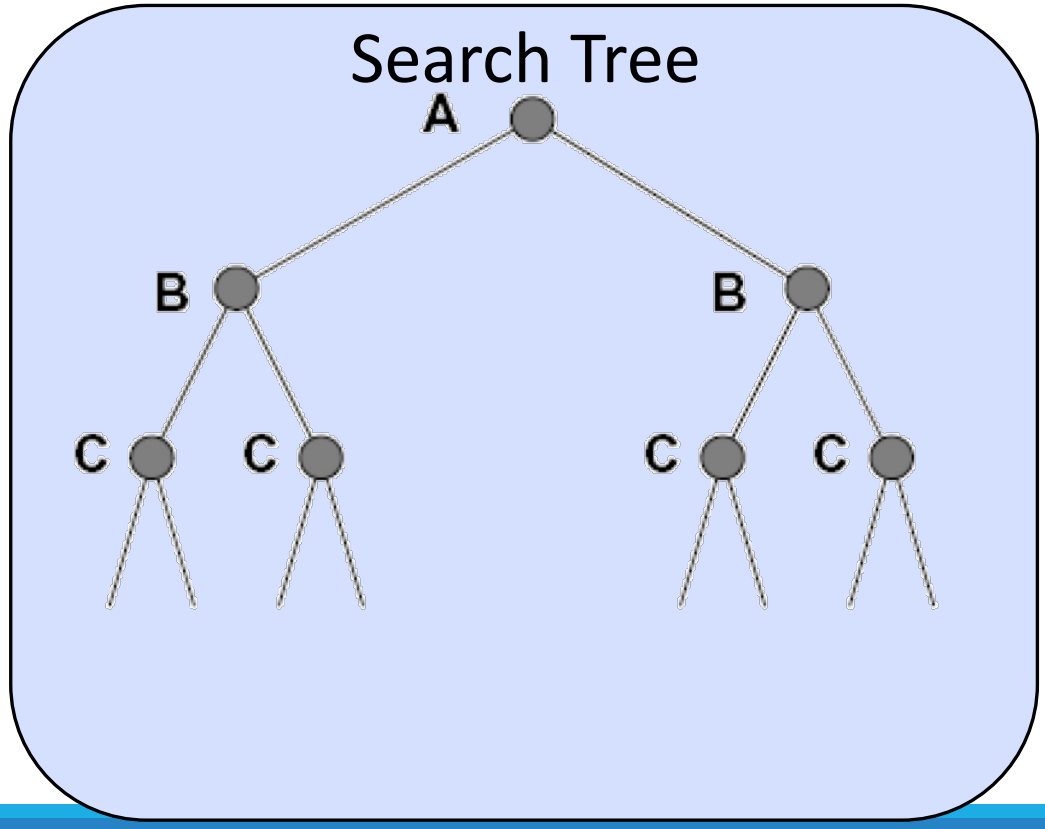
- Frontier: the set of leaf nodes available for expansion at any given point.
- Expansion

Main question: which frontier nodes to expand?

Tree Search: Extra Work!

Failure to detect repeated states can cause exponentially more work.





Graph Search

- Algorithms that forget their history are doomed to repeat it.

- How to avoid exploring redundant paths in Tree Search?

  Add a data structure called the **explored set** to save every expanded node. Newly generated nodes that match previously generated nodes—**ones in the explored set or the frontier**—can be discarded instead of being added to the frontier.

Graph Search (cont.)

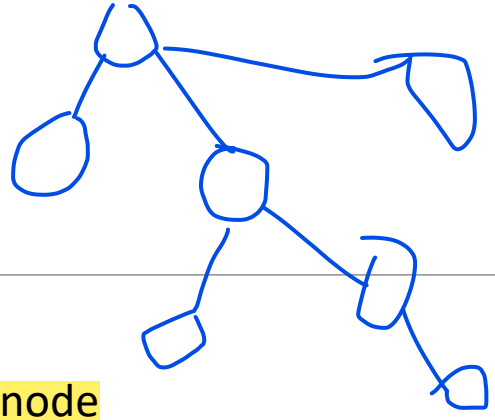
function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Evaluating Search Algorithms

Each of the search strategies are evaluated based on:

- **Completeness**: is the strategy **guaranteed** to find a **solution** when there is one?
- **Time complexity**: how **long** does it take to find a solution?
  (Measured in terms of number of nodes generated during the search.)
- **Space complexity**: how much **memory** does it need to perform the search?
  (Measured in terms of the maximum number of nodes stored in memory.)
- **Optimality**: does the strategy find the **lowest path cost solution** when there are **several solutions**?

Complexity Notation



- The complexity is expressed in terms of three quantities:
  - b: the branching factor or maximum number of successors of any node
  - d: the depth of the shallowest goal node (i.e., the number of steps along the path from the root)
  - m: the maximum length of any path in the state space.

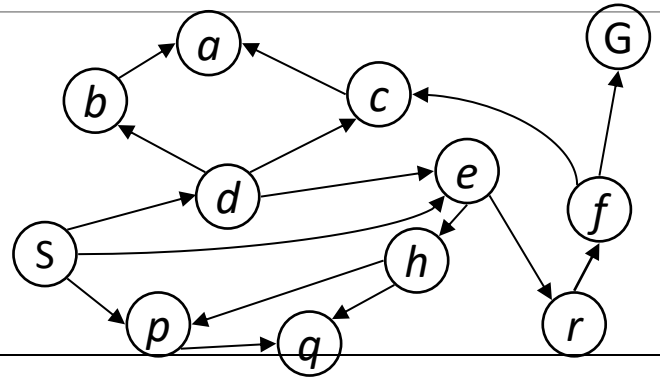
Uninformed Search Strategies

- The uninformed search strategies have no additional information about states beyond that provided in the problem definition
- Thus, they can only distinguish a goal state from a non-goal state, and they cannot tell how far a state from the goal.
- Examples:
  - Breadth-first Search
  - Uniform Cost Search
  - Depth-first Search
  - Depth-limited Search
  - Iterative deepening
  - Bi-directional Search

Breadth-First Search

Strategy: expand the shallowest node first

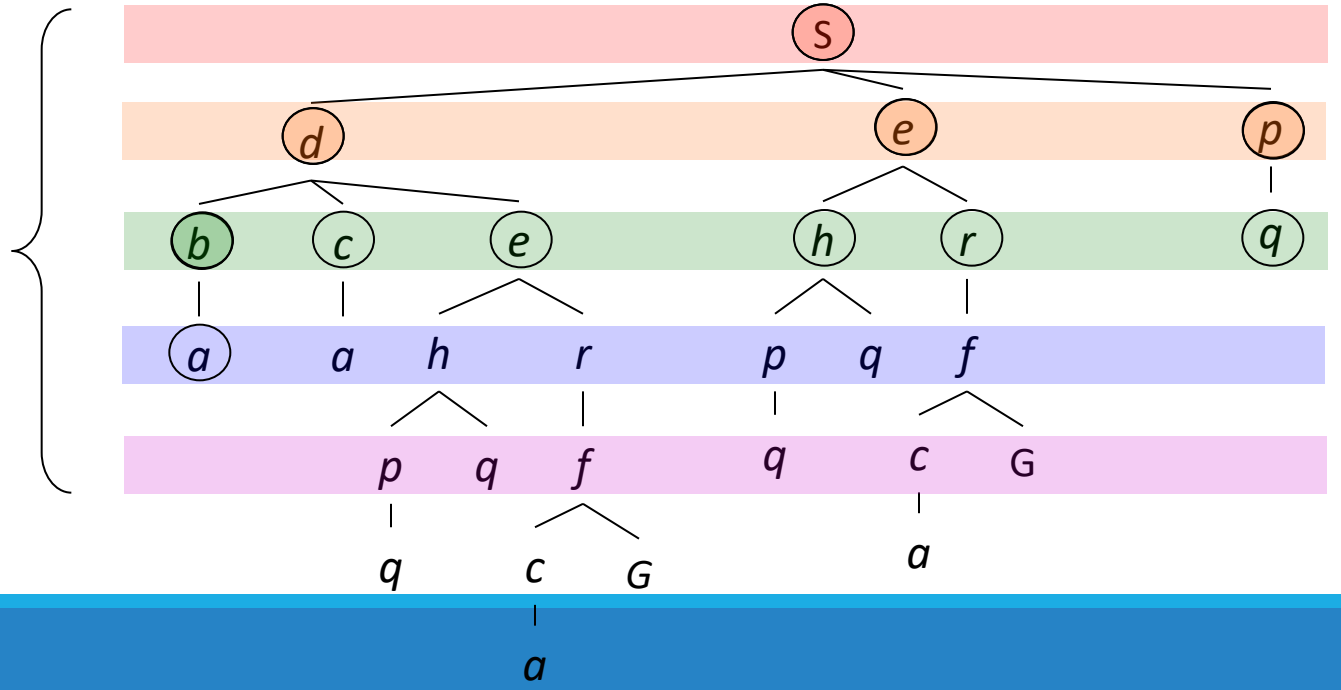
All nodes at a certain level are expanded before the next level nodes.



Frontier Data structure?

FIFO queue

Search Tiers



Breadth-First Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

fa mogard eny ashuf goal node, msh b7otha fl queue, and 3ltoul baro7 a3mlha execution.

Why do we add goal test here not when the node is chosen for expansion?

e7na hena bnkarn 3nd el parent el state bta3t el child, 34an nwfr time, lw tel3 eno el state bta3t el child goal, brg3 mn 3nd el parent, lakn lw kona mbn3mlsh keda, kan lazm anzl lel child level w sa3tha ha3rf 3ndo, fa el complexity hatb2a O(b^(d+1))

Breadth-First Search (BFS) Properties

What nodes does BFS expand?

- Processes all nodes above shallowest solution
- Let depth of shallowest solution be d
- Search takes time $O(b^d)$
- The complexity would be $O(b^{d+1})$ if the goal test is applied when the node is chosen for expansion.

How much space does the frontier take?

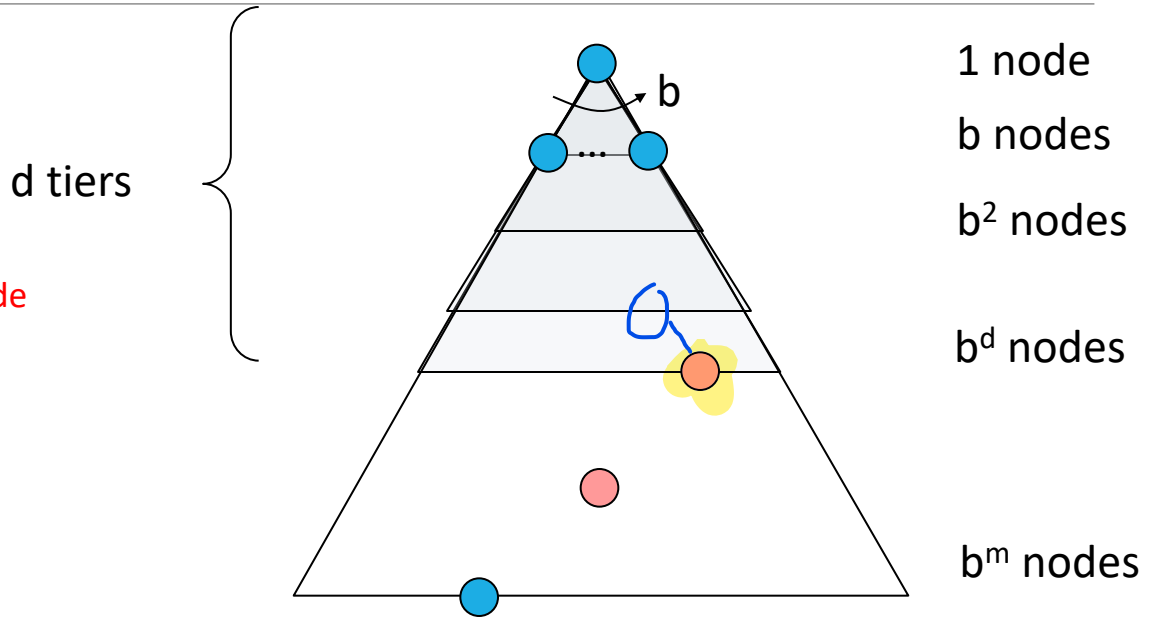
- Has roughly the last tier, so $O(b^d)$

Is it complete?

- d must be finite if a solution exists, so yes!

Is it optimal?

- If the path cost is a nondecreasing function of the depth of the node (for example if all actions have the same cost).



b is the branching factor
m is the maximum depth
solutions at various depths

Breadth-First Search (BFS)

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

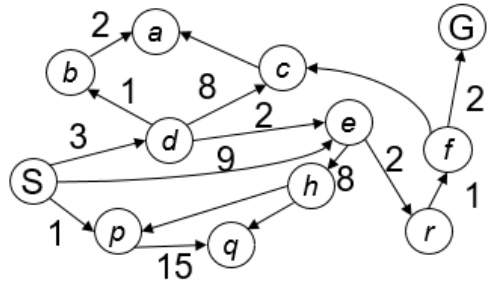
Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

- Memory requirements are a major problem for BFS.
- Time requirement is also a big problem. (check depth 14,16).

Uniform-Cost Search

Strategy: expand the cheapest node first.

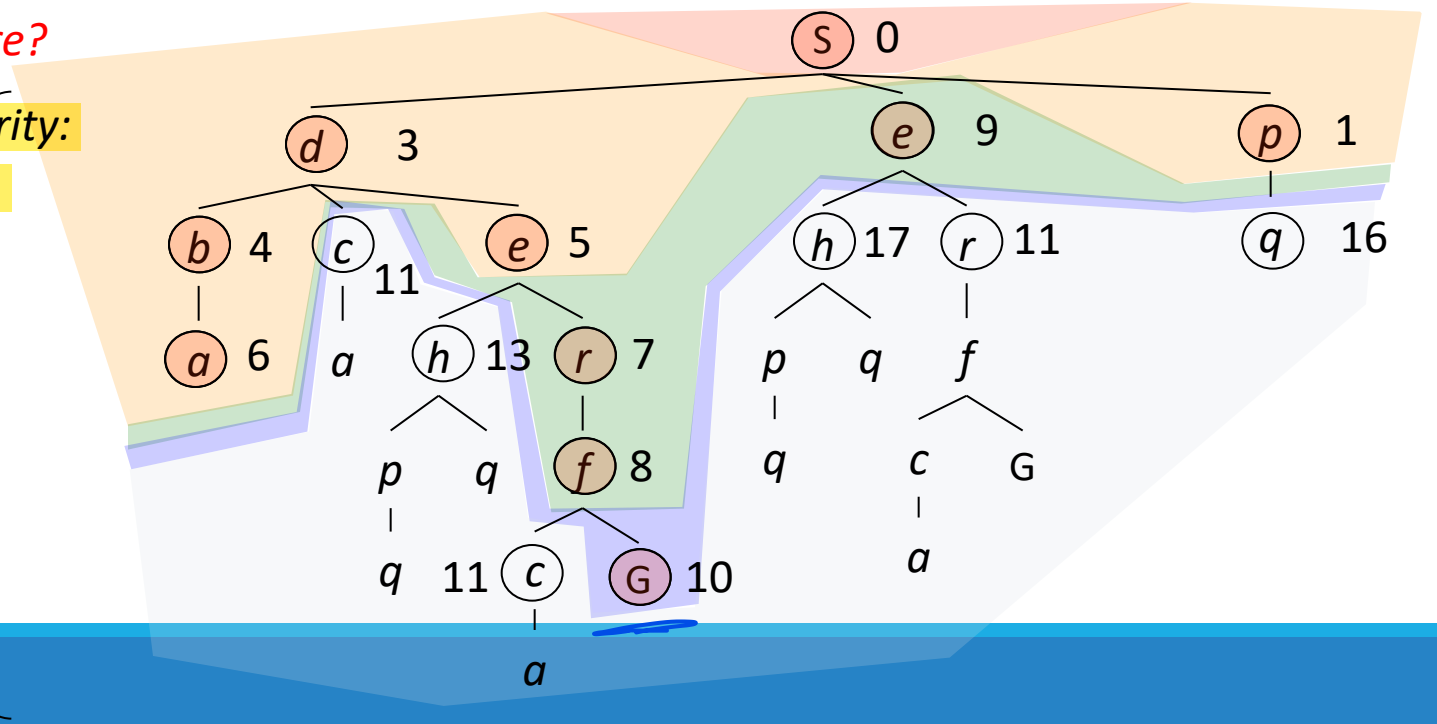
(The lowest path cost)



Frontier Data structure?

A priority queue (priority: cumulative cost (path cost))

Cost contours



Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table

Observe the differences from BFS algorithm!

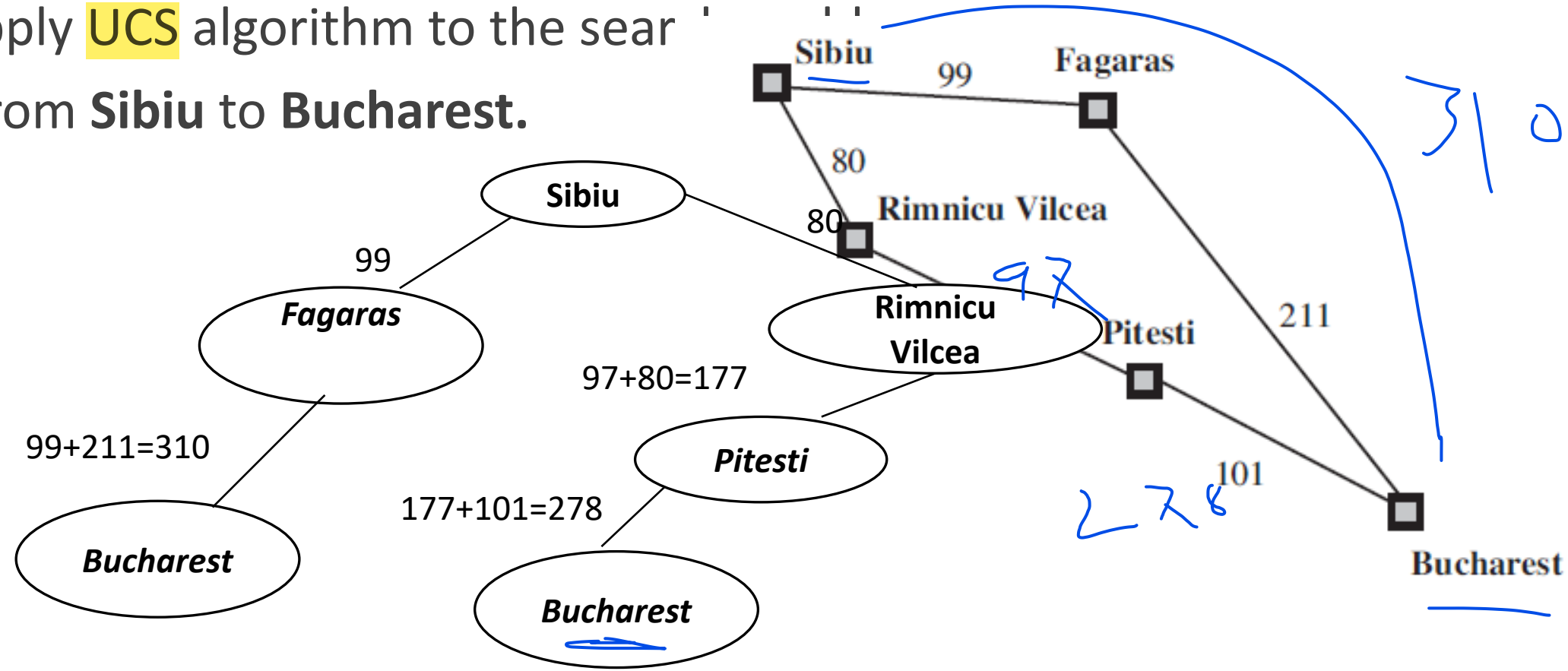
Why goal test is done when the node is expanded?

In case a better path is found to a node in the frontier.

Uniform Cost Search Example

Apply UCS algorithm to the sear

From **Sibiu** to **Bucharest.**



Uniform Cost Search (UCS) Properties

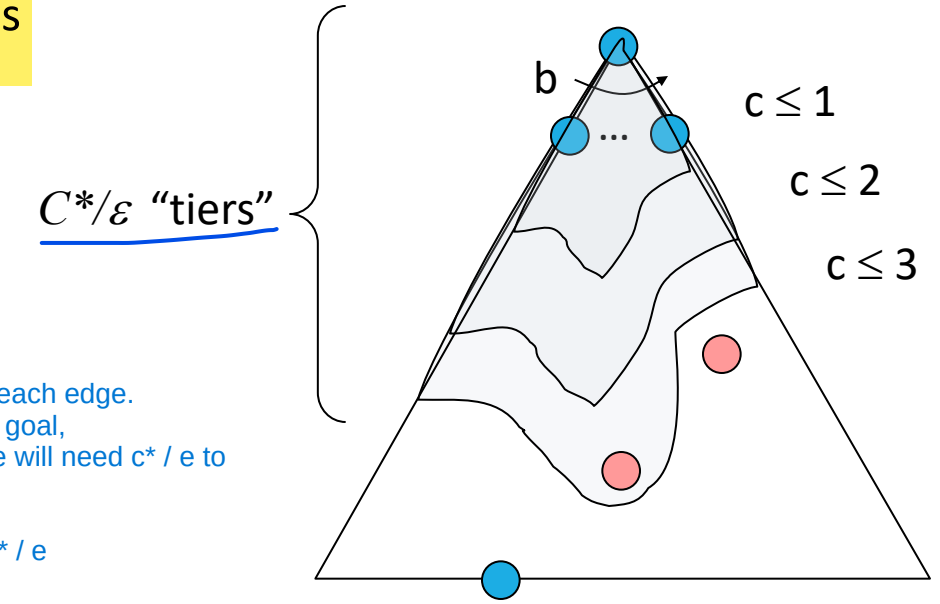
- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.

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 frontier 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!



e is the path cost for each edge.
C* is the depth of the goal,
roughly estimation we will need c* / e to achieve a goal

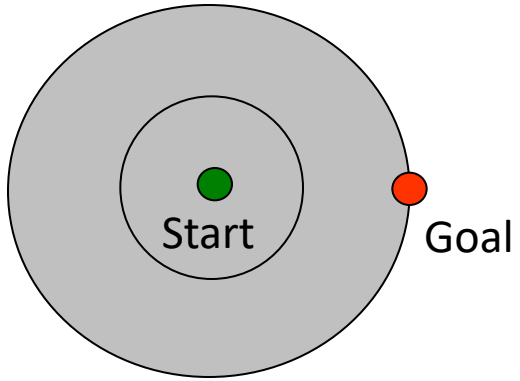
number of levels is C* / e

Uniform Cost Search Issues

UCS is complete and optimal!

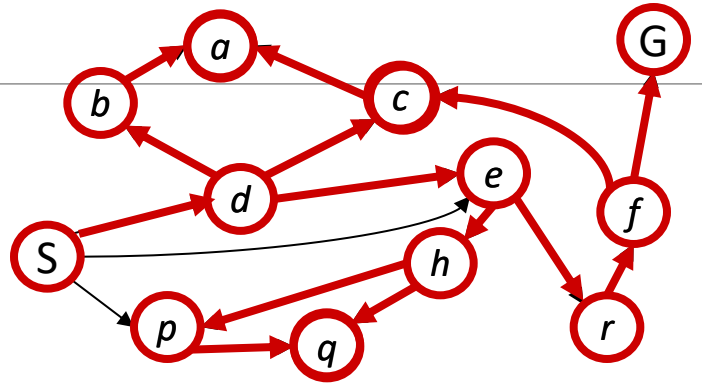
UCS Issues :

- Explores options in every “direction”
- No information about goal location



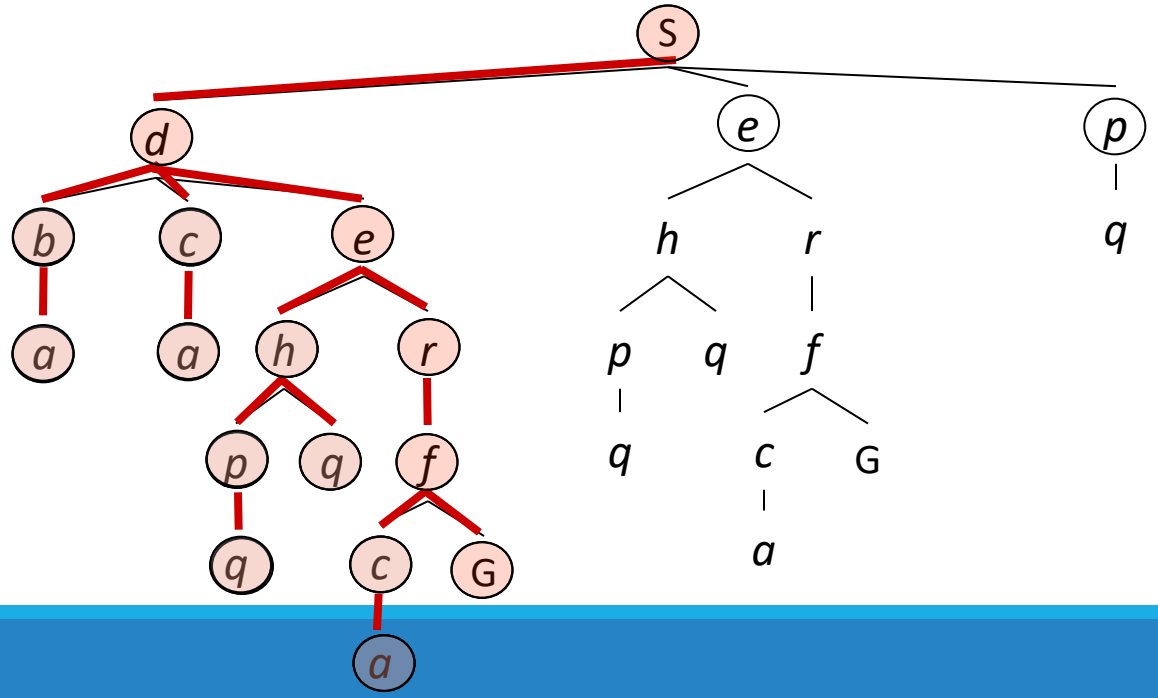
Depth-First Search

Strategy: expand the deepest node first.



Frontier Data structure?

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 frontier take?

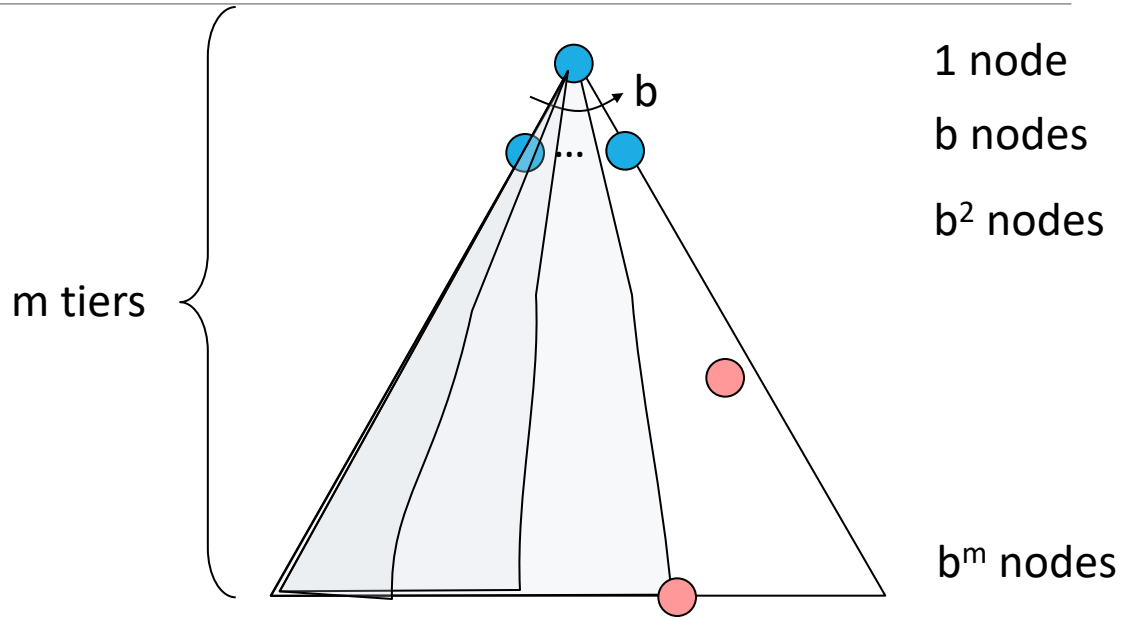
- **Only has siblings on path to root, so O(bm)**

Is it complete?

- The graph search version (no repeated states) is complete in **finite** spaces, but tree search is not complete.

Is it optimal?

- No, it finds the "leftmost" solution, regardless of the cost



Depth-first Search

- Depth-first search main advantage is its space complexity O(bm) for a maximum depth m.

- DFS issues:
  - It can get **stuck** going down the wrong path.
  - Many problems have **very deep** or even **infinite** search trees.
  - DFS should be **avoided** for search trees with **large** or **infinite maximum depths**.

Depth-limited Search

- Depth limited search avoids the pitfalls of DFS by imposing a cutoff on the **maximum depth l.**
- However, if we choose a depth limit that is too small (l<d), then DLS is not even complete.
- The time and space complexity of DLS is similar to DFS.

Completeness: Yes, if *l >= d*

Time complexity: b^l

Space complexity: bl

Optimality: No *(b-branching factor, l-depth limit)*

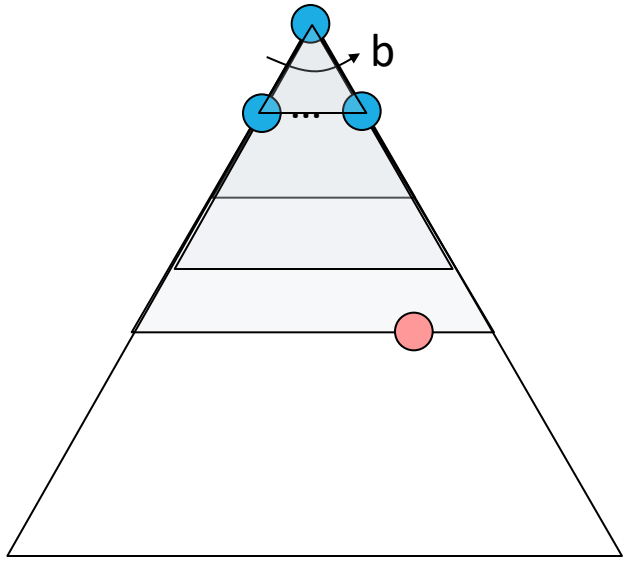
Iterative Deepening

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 deepest level searched.



Iterative Deepening

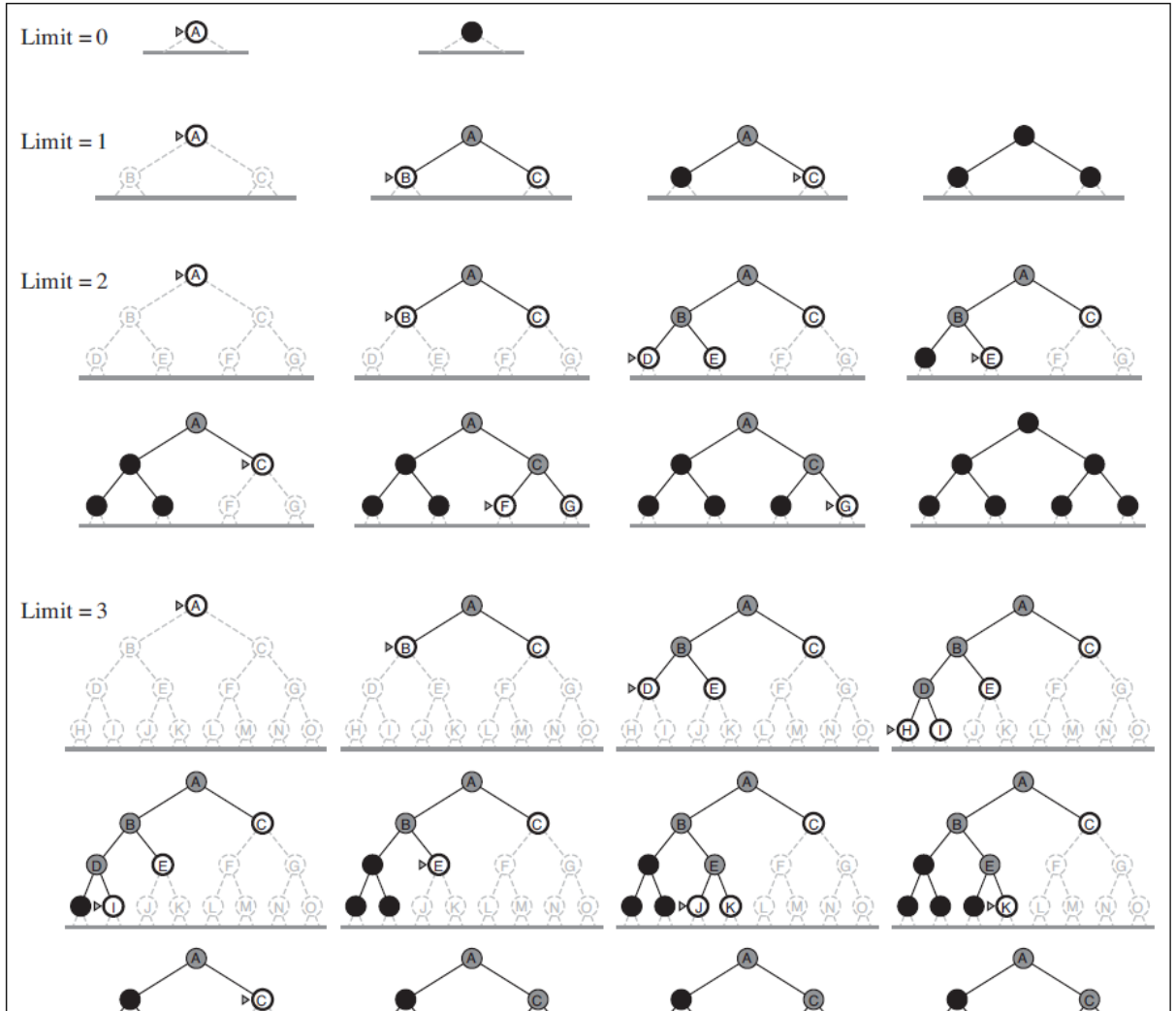
- The **hard part** about depth-limited search is **picking an appropriate depth limit**.

- Iterative deepening is a strategy that resolves the issue of choosing the best depth limit by **trying all possible depth limits**: first depth 0, then depth 1, the depth 2, and so on until a goal is found.

- In effect, it combines the **benefits of DFS and BFS**.

- It is **complete** like BFS and has **modest memory requirements** of DFS.

Iterative Deepening



enta hatdkhol mn hena

w ana hadkhol mn hena

w net2fsh hena.

shaklak fahem ya nosa.

Bi-directional Search

- **Search forward** from the Initial state
- And **search backwards** from the Goal state..
- Stop when two meet in the **middle**.

(The frontiers of the two searches intersect; means a solution is found.)

Completeness: Yes

Time complexity: $b^{d/2}$

Space complexity: $b^{d/2}$

Optimality: Yes

Bi-directional Search Issues

How do we search backward?

- Not all actions are reversible.
- There could be multiple goal states.
- Then construct a new dummy goal state whose immediate predecessors are all the actual goal states.
- The goal can be an abstract description like n-queens search problem (“no queen attacks another queen”, then bidirectional search is difficult to use.

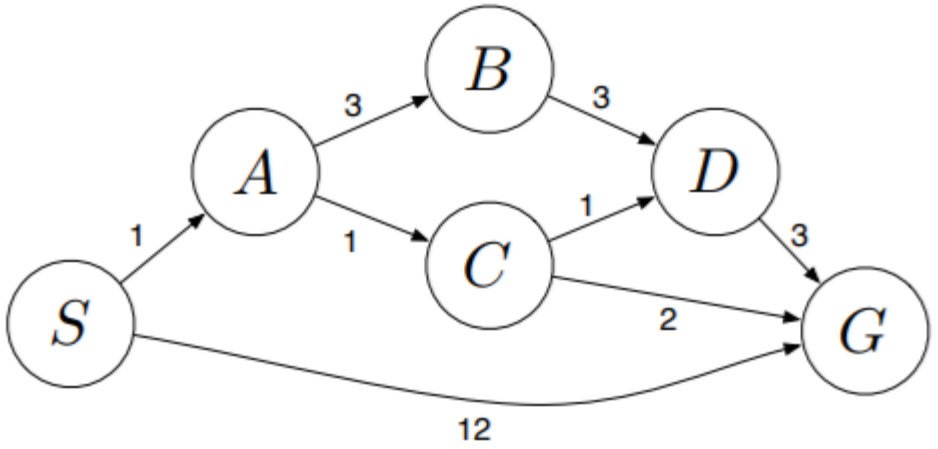
Comparing Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Example

Apply BFS, DFS, and UCS to the following problem:



Summary

- Problem-Solving Agents
- Search Problem Formulation
- Graph Search vs. Tree Search
- Evaluating Search algorithms performance.
- Uninformed Search Strategies