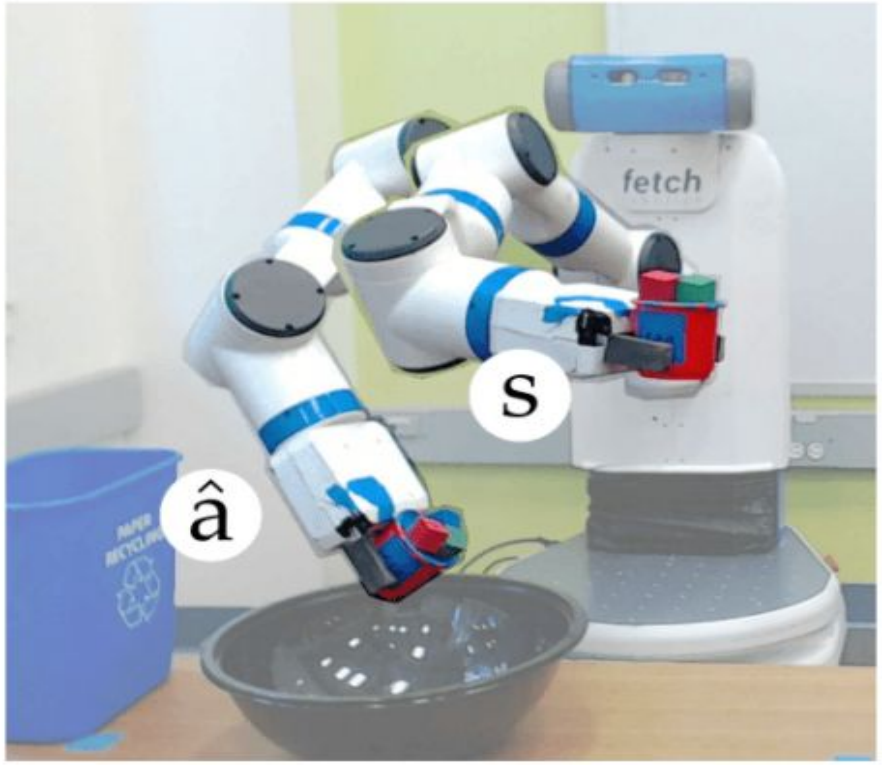
Search

Route Finding

Objective: shortest? fastest? most scenic?

Actions: go straight, turn left, turn right

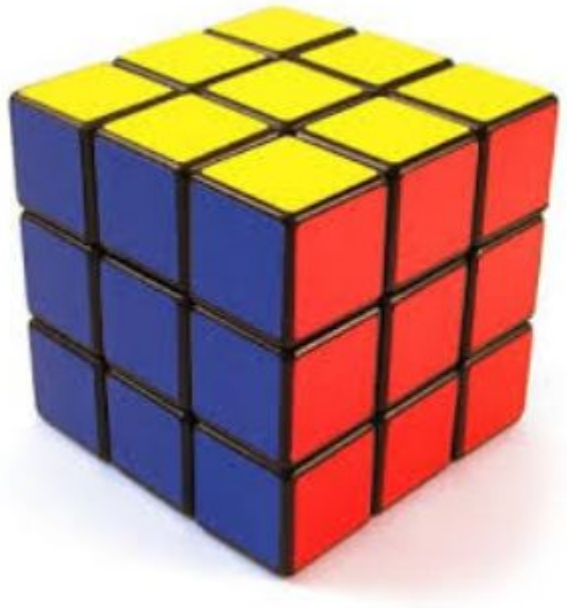
Robot Motion planning



Objective: fastest? most energy efficient? safest? most expressive?

Actions: translate and rotate joints

Solving Puzzles



Objective: reach a certain configuration
Actions: move pieces (e.g., Move12Down)

Machine Translation

la maison bleue

the blue house

Objective: fluent English and preserves meaning

Actions: append single words (e.g., the)

Reflex agent Vs Goal based agent

Classifier (reflex-based models):

Search problem (state-based models)(Goal based Agent)

Key: need to consider future consequences of an action!

Reflex agent Vs Goal based agent

Reflex-based models - Machine Learning Models

(e.g., linear predictors and neural networks) that output either 0 or 1 (for binary classification) or a real number (for regression).

reflex-based models - suitable - sentiment classification
spam filtering etc.,

But applications such as solving puzzles, demand more.

Search problems - Goal based Agent - we build a predictor which takes an input , but will return an entire **action sequence**, not just a single action. .

Problem Solving

- Formulate the problem
- Solve it by search through the state space
- Explicit search trees/search graph

Measures of performance

Completeness

Guaranteed to find a solution when there is one?

Optimality

Finds an optimal solution?

Time

How long does it take to find a solution?

Space

How much memory is needed to conduct the search?

Search Strategies

- Uninformed search / Blind search
- Informed search / Heristic search

Uninformed

Can only generate successors and distinguish goals from non-goals

Informed

Strategies that know whether one non-goal is more promising than another

Uninformed search strategies

- We have no clue whether one non-goal state is better than any other.
- Search is blind.
- We don’t know if your current exploration is likely to be fruitful.

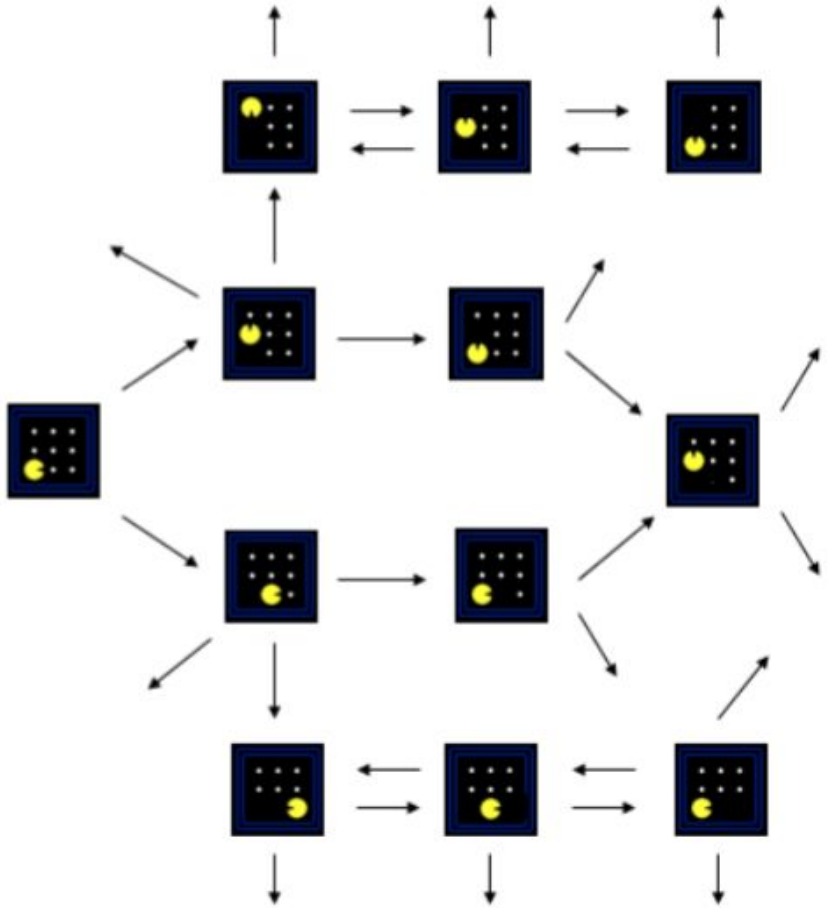
Various blind strategies

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Iterative deepening search

Travelling in Romania

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

State Space Graph

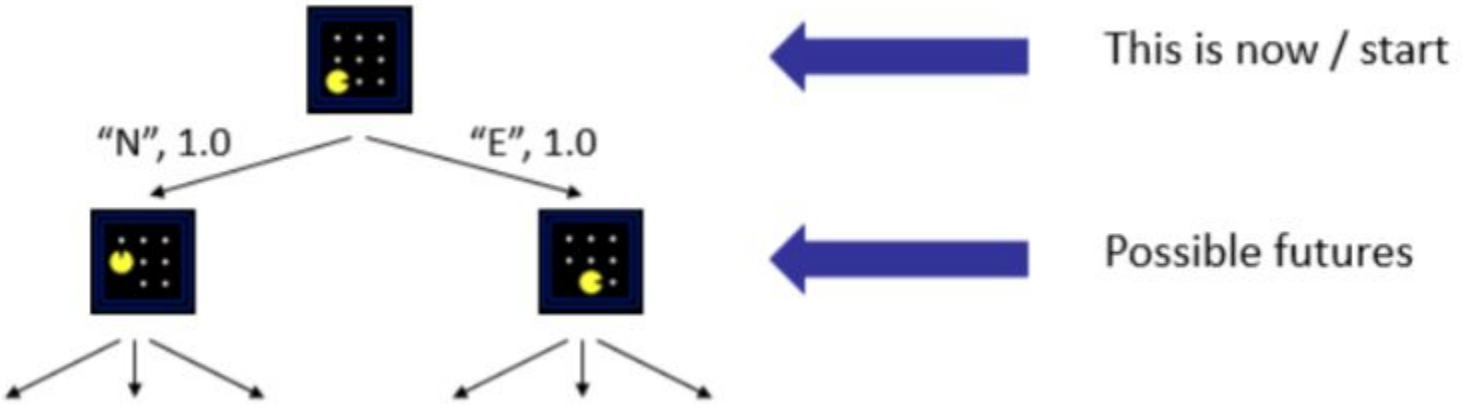


- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

- In a state space graph, each state occurs only once!

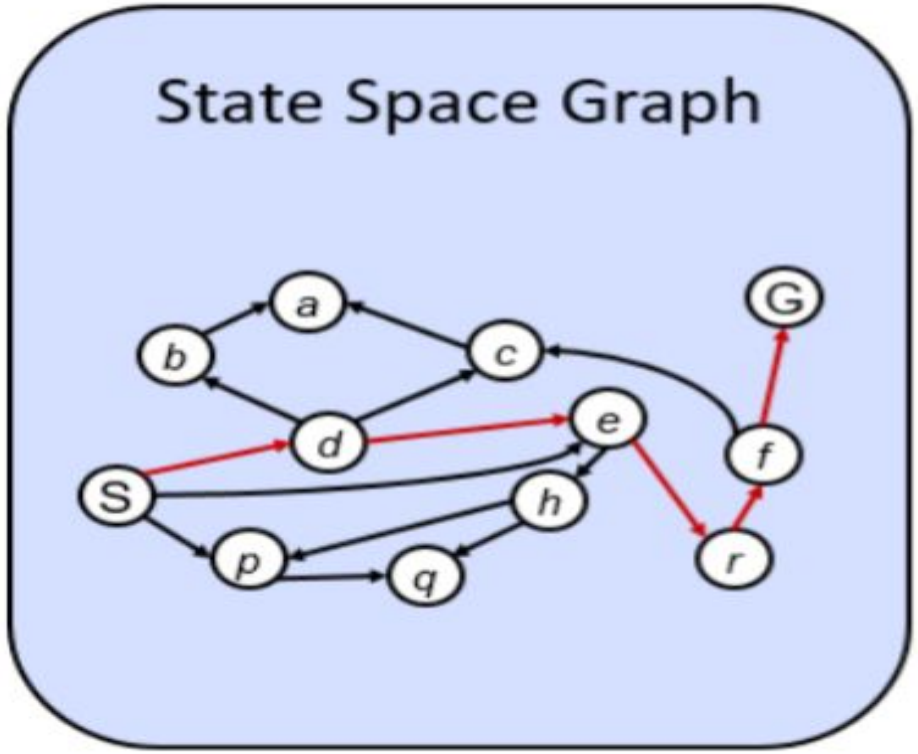
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

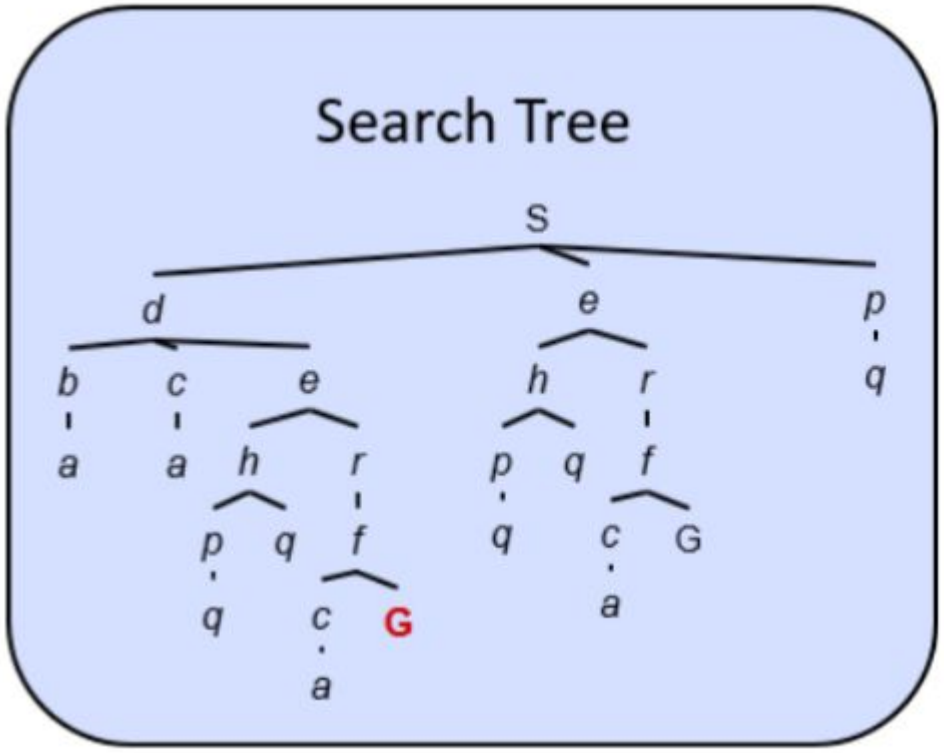
Search Trees

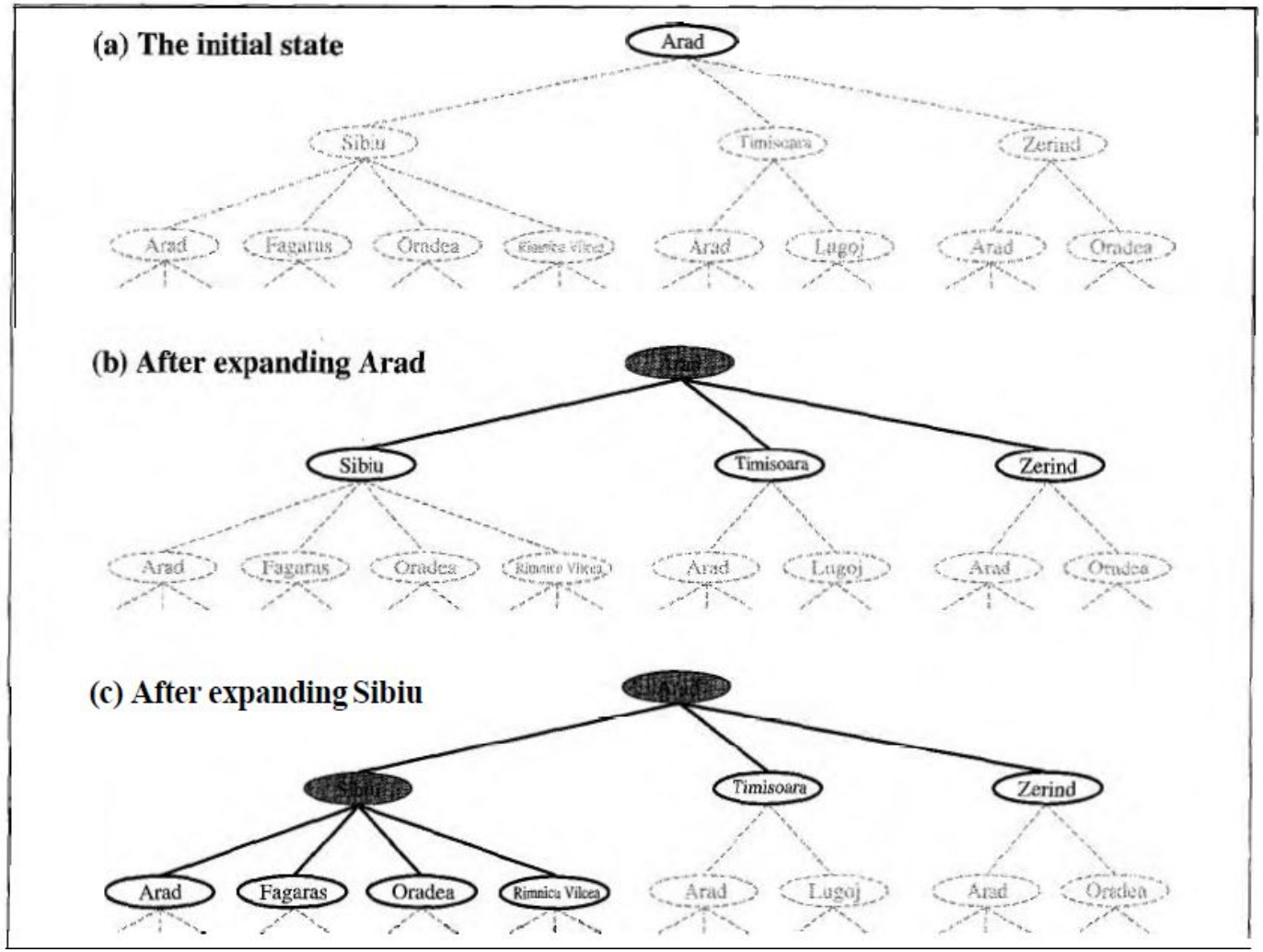


- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

Search Graph Vs Search Tree







Partial search trees for finding a route from Arad to Bucharest.

- Nodes that have been expanded are shaded;
- nodes that have been generated but not yet expanded are outlined in bold;
- nodes that have not yet been generated are shown in faint dashed lines.

General Search Tree algorithm

Function *TREE SEARCH(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

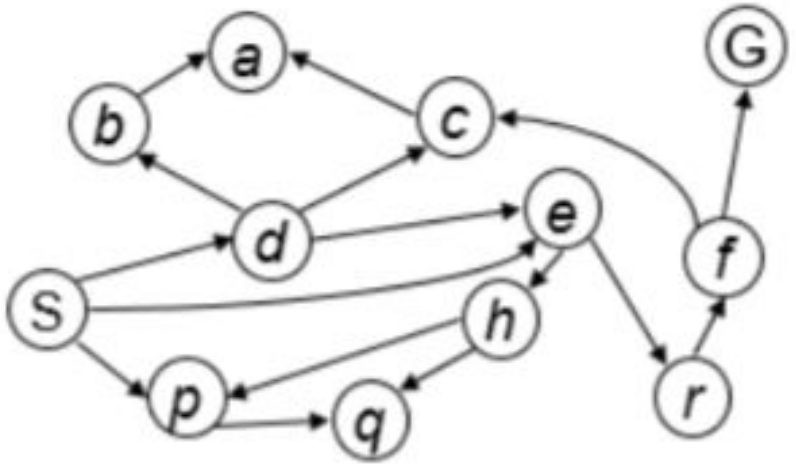
then return the corresponding solution

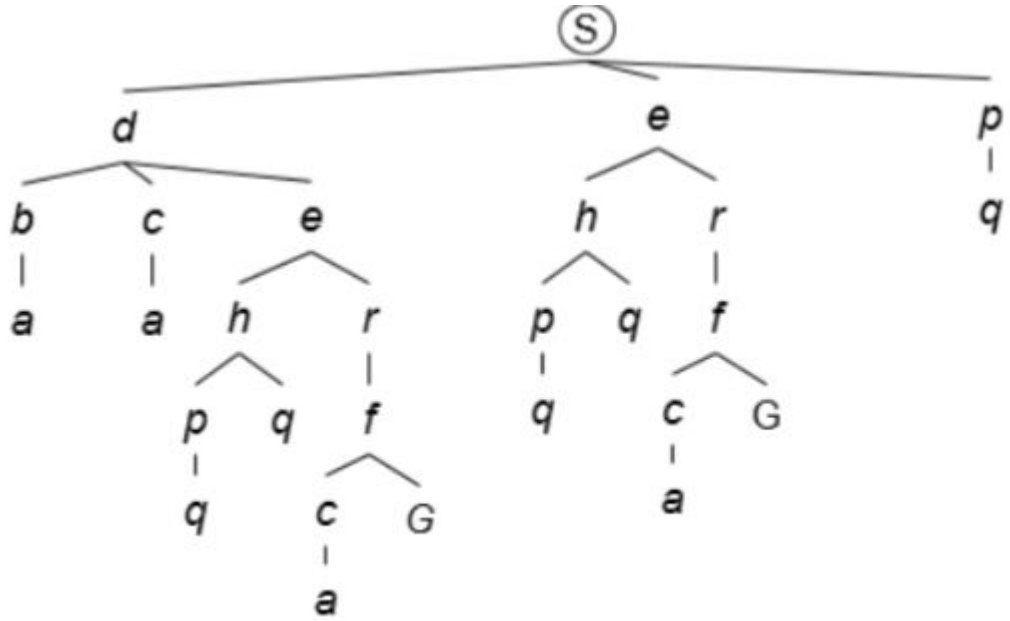
else expand the node and add the resulting nodes to the search tree

Breadth-first Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

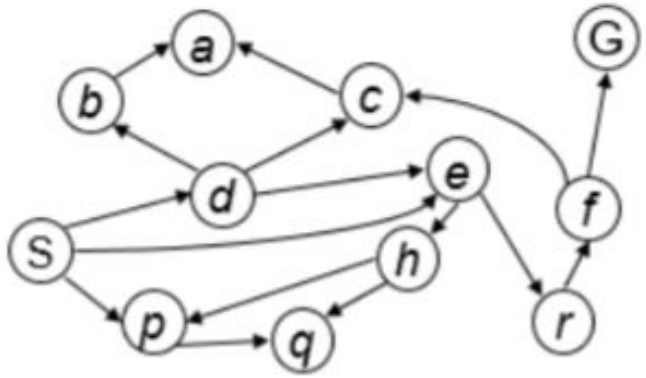


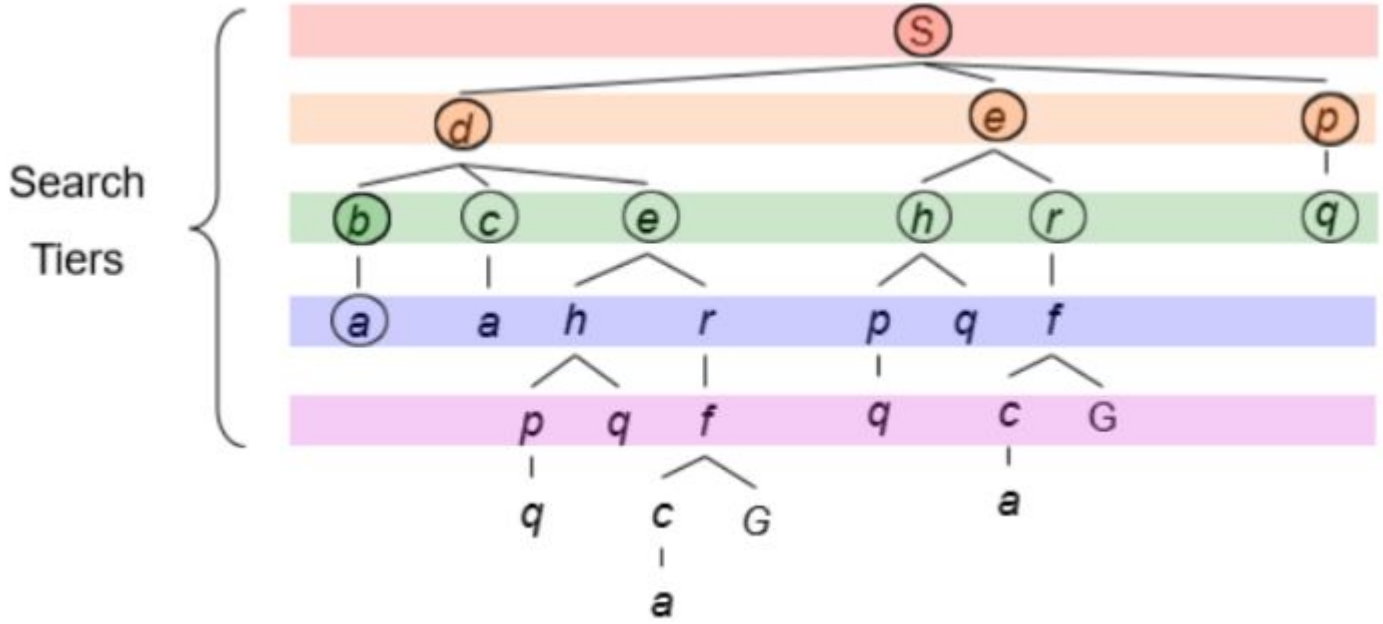


Breadth-first Search

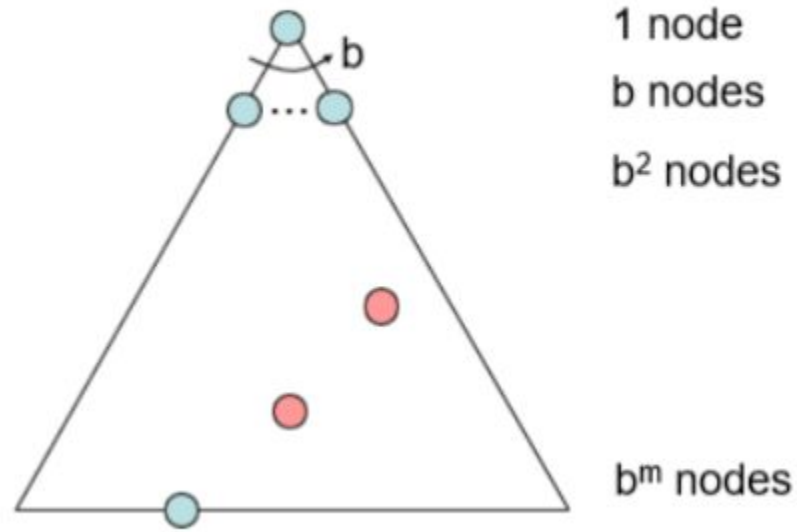
Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

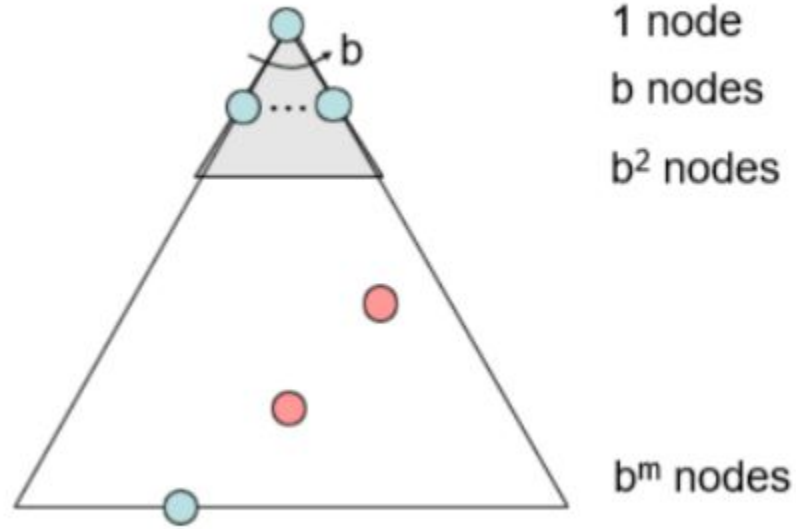




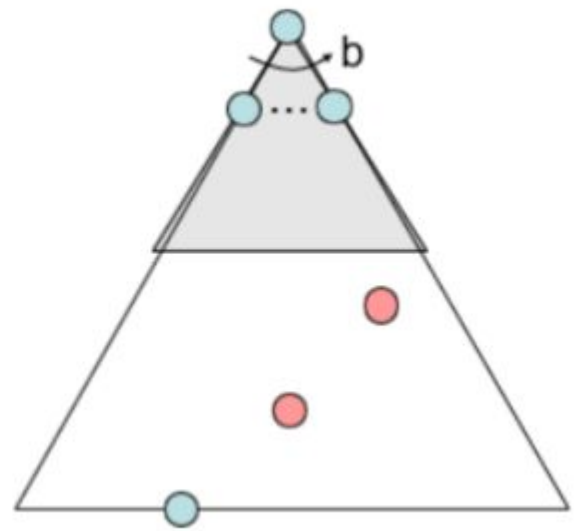
- What nodes does BFS expand?



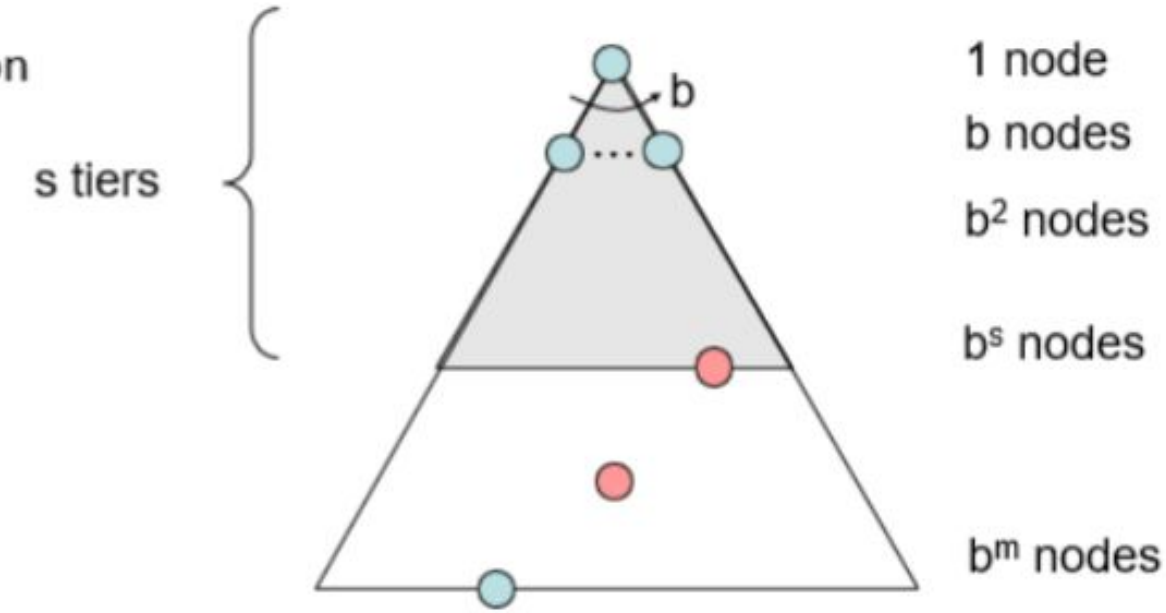
- What nodes does BFS expand?



- What nodes does BFS expand?



- 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)



Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Figure 3.11 Time and memory requirements for breadth-first search. The numbers shown assume branching factor b = 10; 10,000 nodes/second; 1000 bytes/node.

The table assumes that 10,000 nodes can be generated per second and that a node requires 1000 bytes of storage

BFS Graph 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)
```

Uniform Cost Search

BFS finds the ***shallowest*** goal state.

Uniform cost search modifies the BFS by **expanding ONLY the lowest cost node** (as measured by the path cost $g(n)$)

The **cost of a path** must **never decrease** as we traverse the path, ie. no negative cost should in the problem domain

This is done by storing the frontier as a priority queue ordered by g

Uniform Cost Search

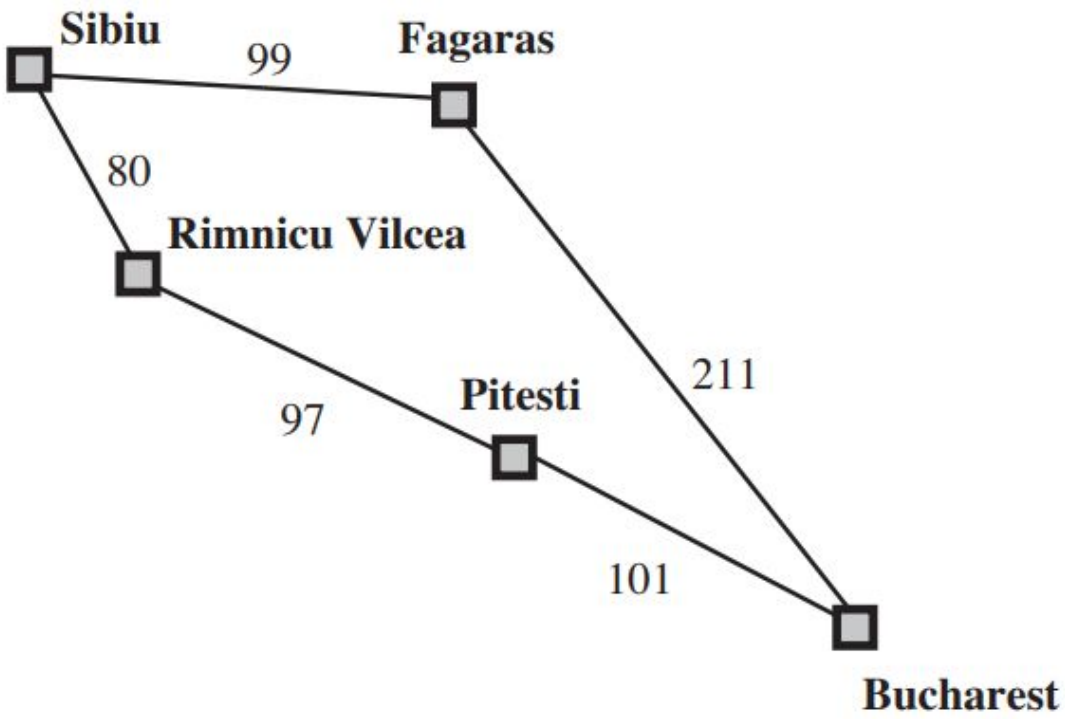
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

    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
```

In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.

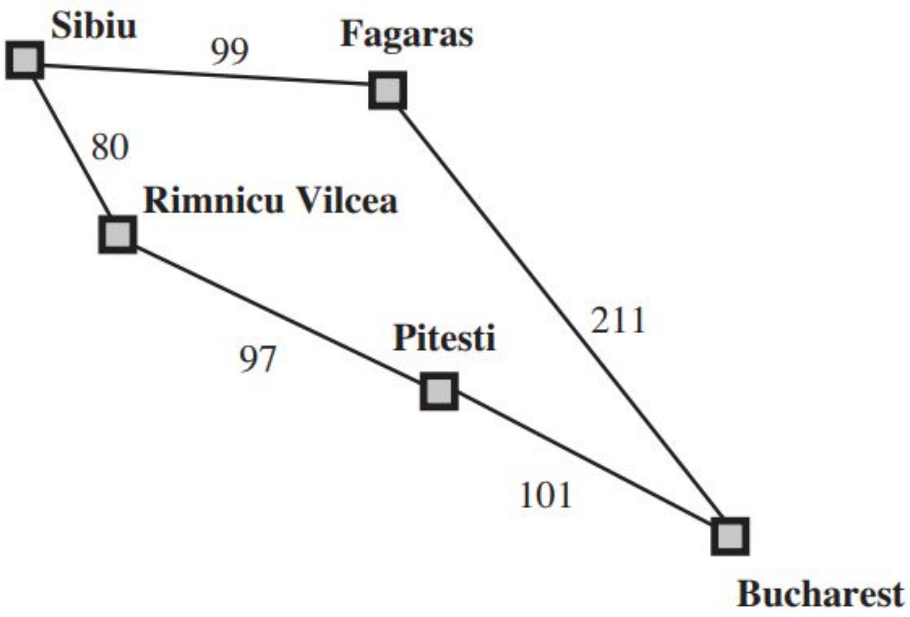
1. The goal test is applied to a node when it is selected for expansion rather than when it is first generated.

2. A test is added in case a better path is found to a node currently on the frontier



Problem is to get from Sibiu to Bucharest

Problem is to get from Sibiu to Bucharest



The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively

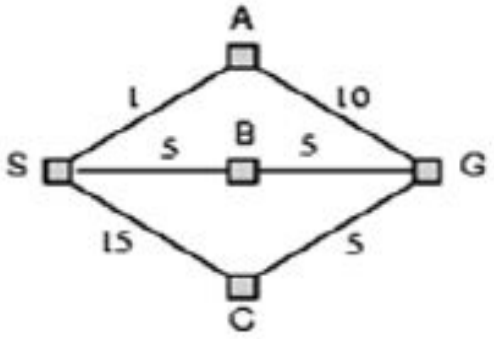
The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177

The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310

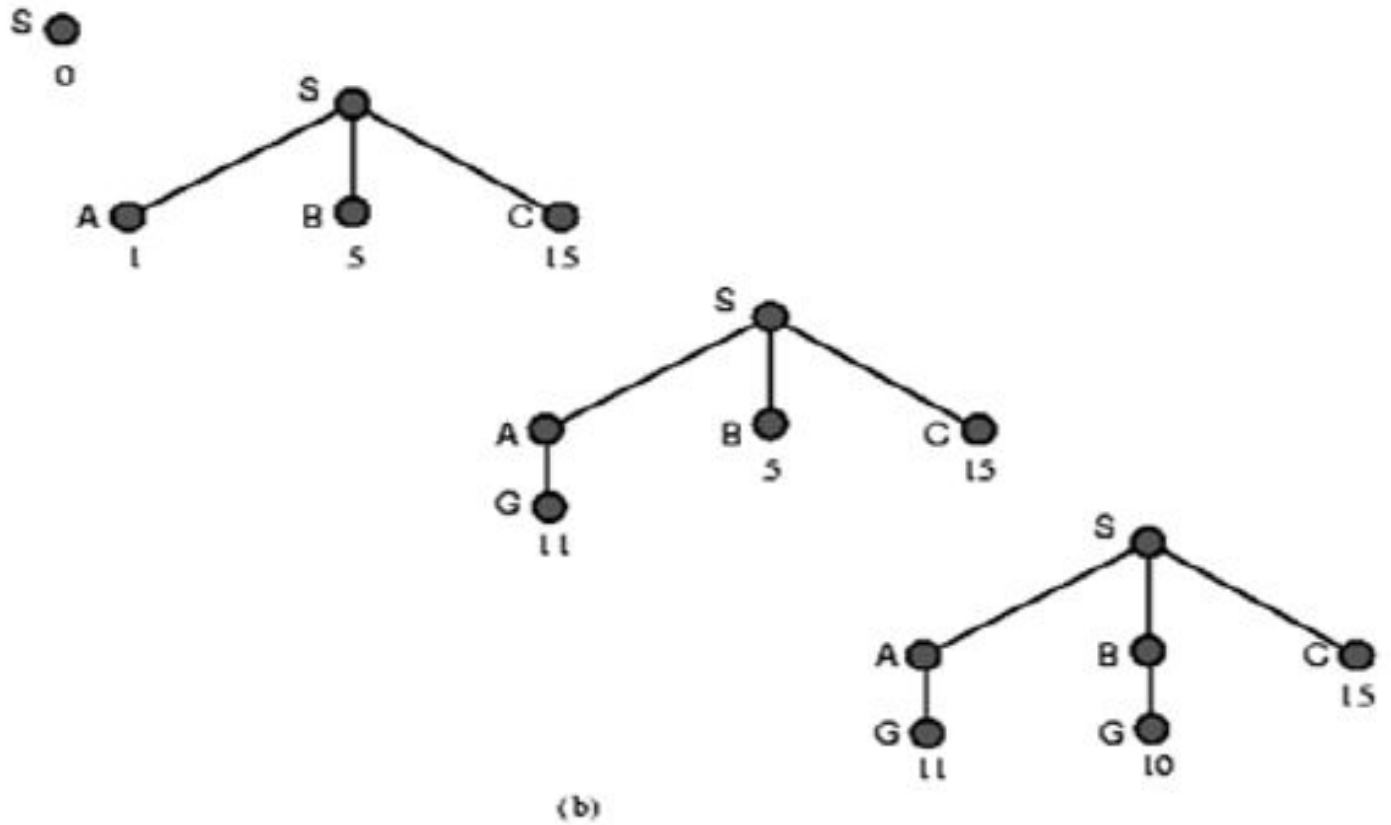
UCS keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+ 97+ 101 = 278.

Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned

Uniform Cost Search

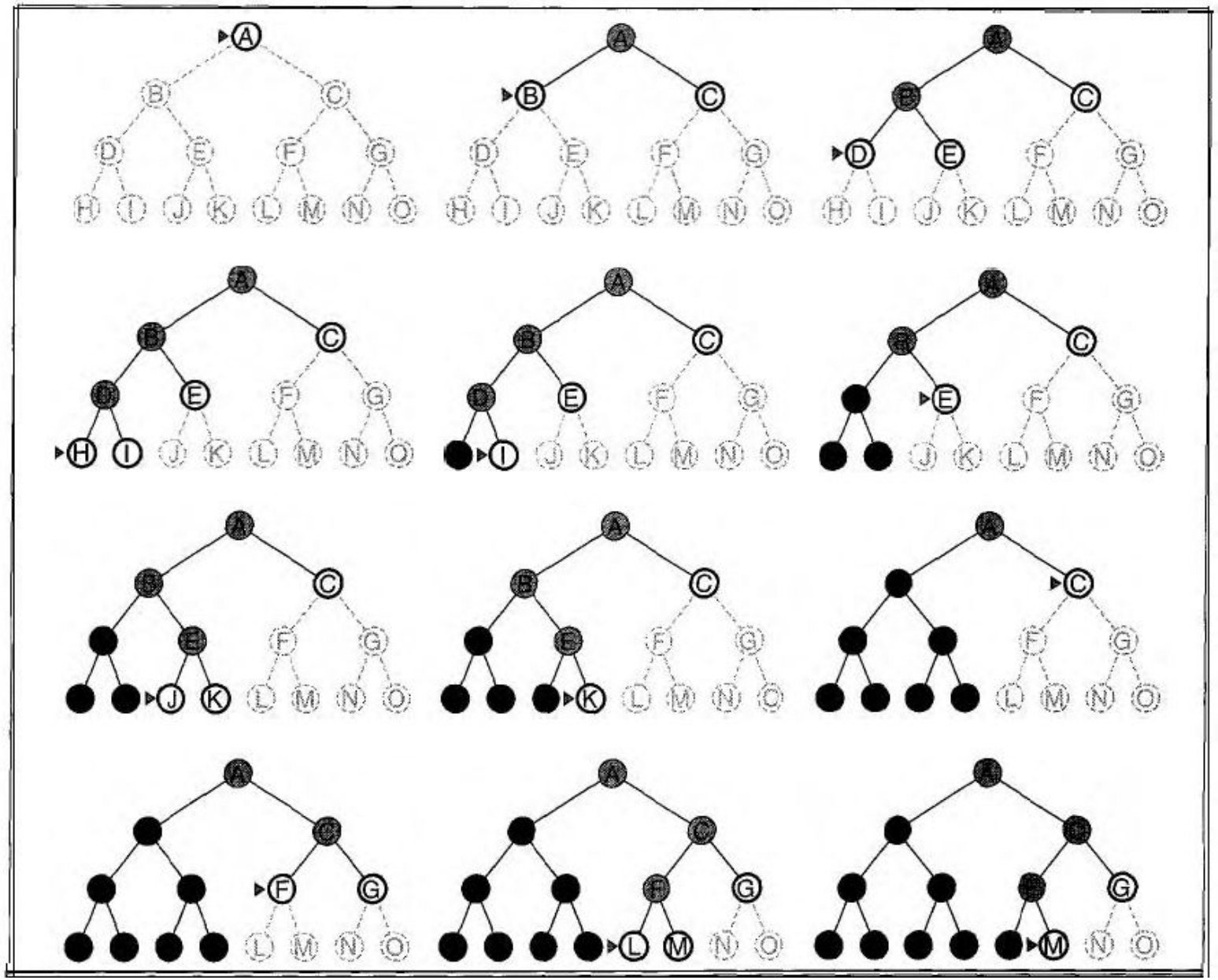


(a)



(b)

Depth-first Search



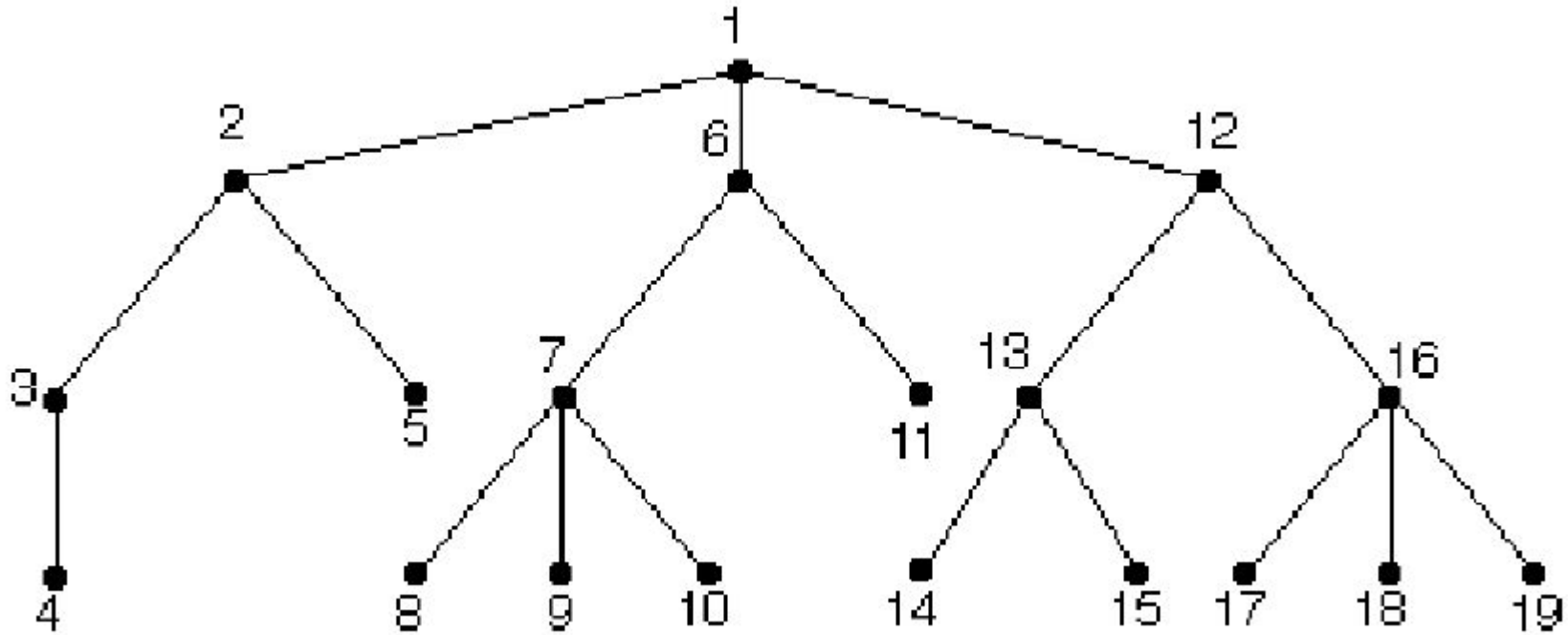
Depth-first search always expands the *deepest* node in the current fringe of the search tree

last-in-first-out (LIFO) -- Stack

Depth-first search with a recursive function

Depth First Search

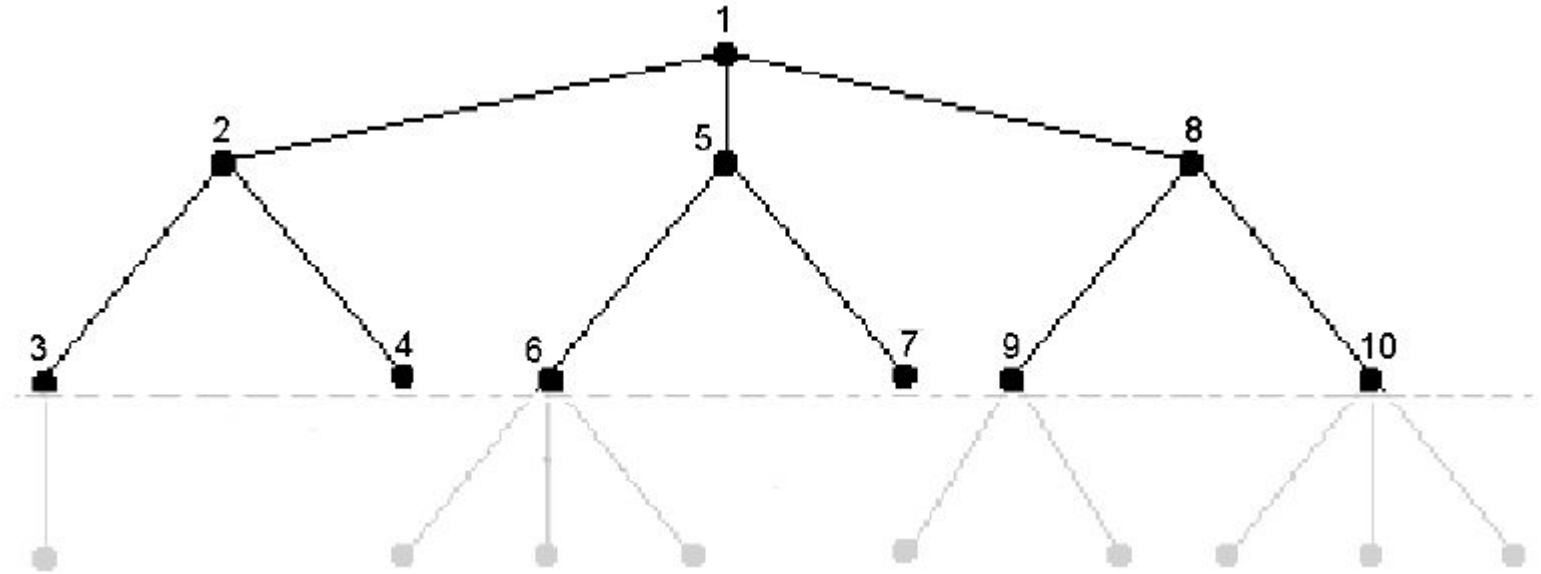
Fringe: is a LIFO queue, new successors go to the front



Depth Limited Search

"**Practical**" DFS

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

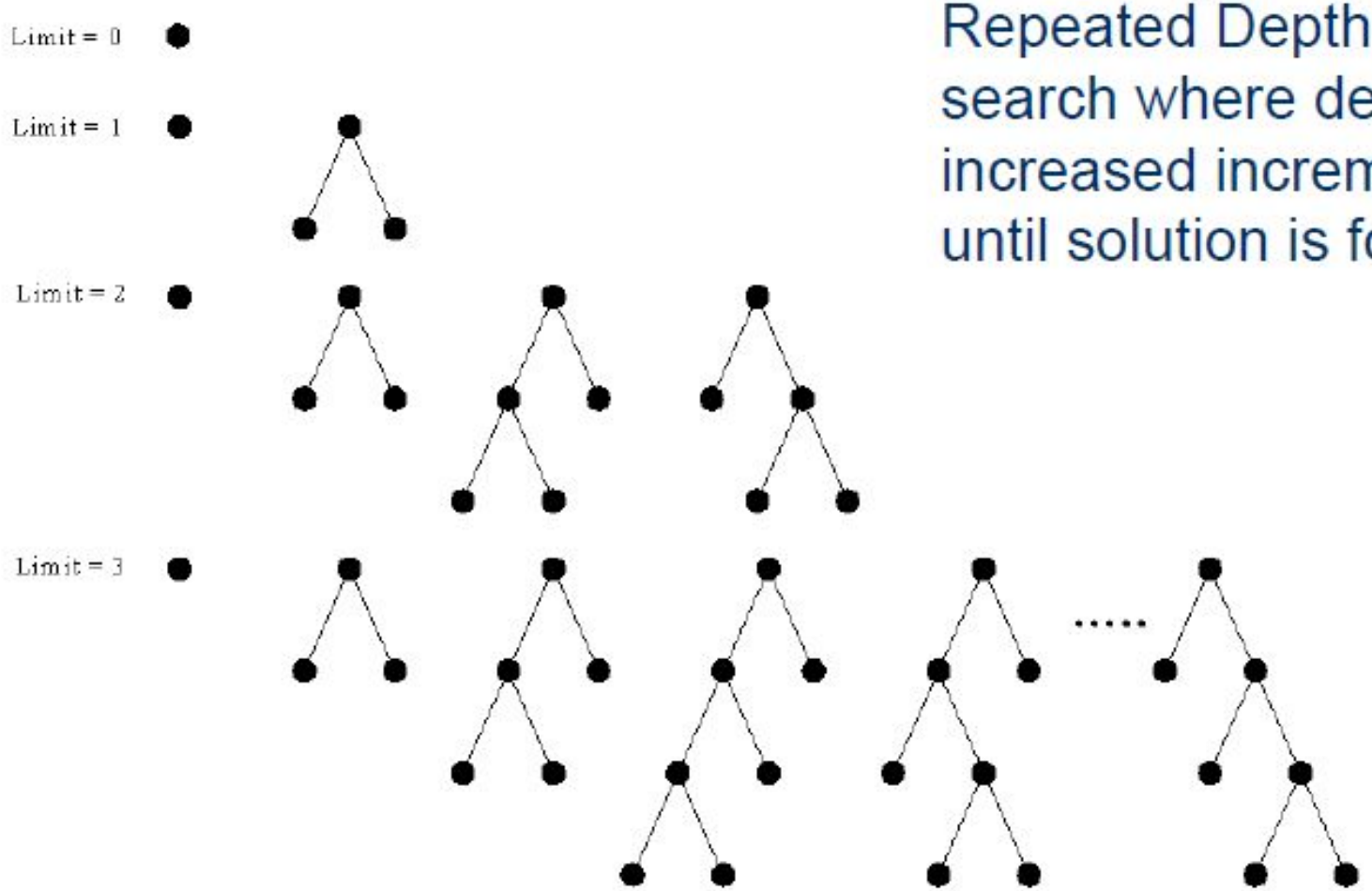


Iterative Deepening Search

The hard part about DLS is picking a good limit.

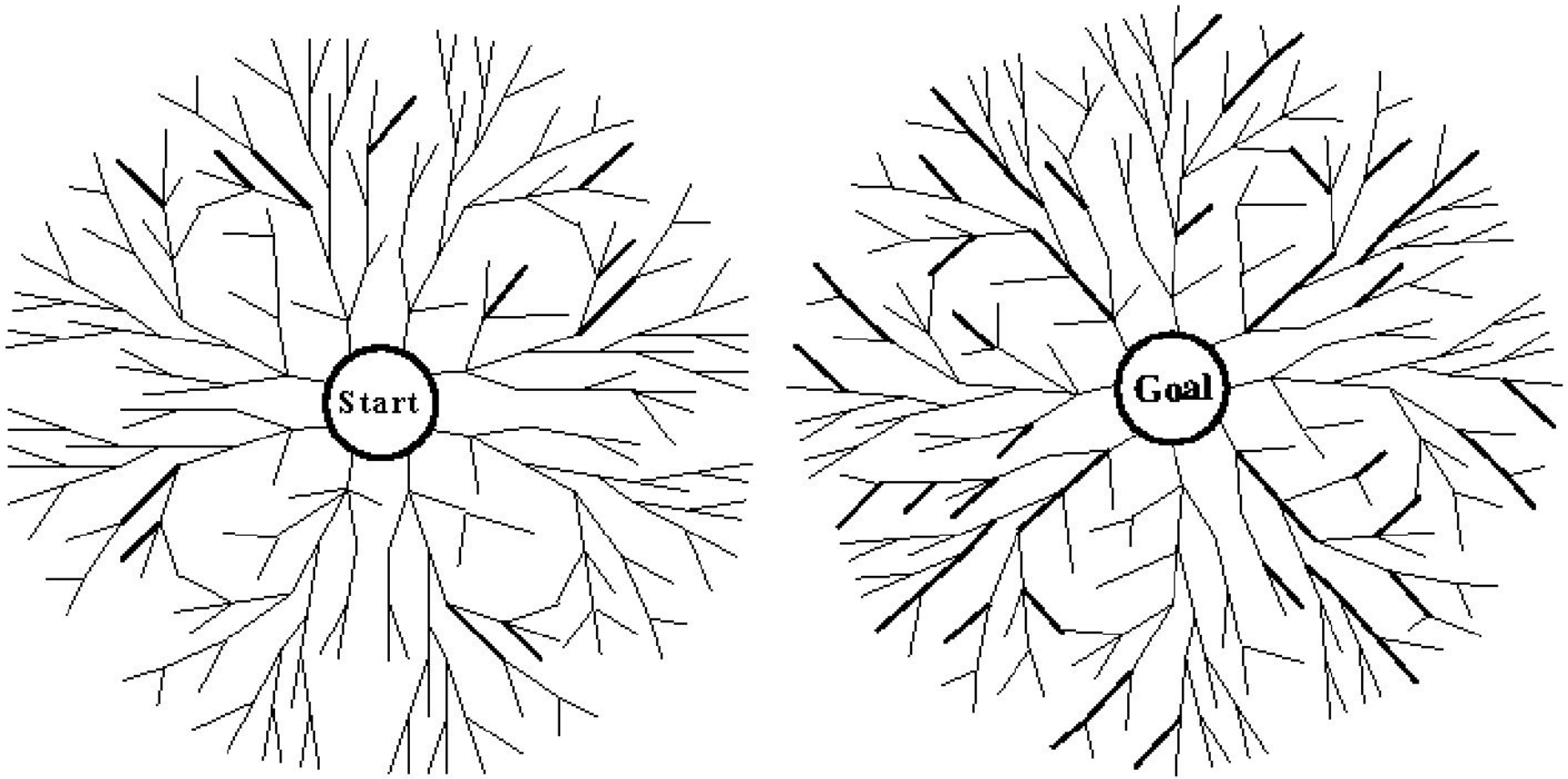
IDS is a strategy that sidesteps the issue of choosing the best depth limit by **trying all possible depth limits**: first depth 0, then depth 1, the depth 2, and so on

Iterative Deepening Search



Repeated Depth-limited search where depth-limit is increased incrementally until solution is found

Bi-directional Search



- **Search forward** from the Initial state
- And **search backwards** from the Goal state.
- Stop when they meet in the middle
- good??; When can you do such a search??
- Each search checks each node before it is expanded to see if it is in the fringe of the other search.

Comparing blind search techniques

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

b - branching factor;
d is the depth of the solution;
m is the maximum depth of the search tree;
l is the depth limit