CS7IS2: Artificial Intelligence

Lecture 1: Intelligent Agents

Ivana.Dusparic@tcd.ie

Problem-solving agents

Definitions - AI

- EU High-level working group on AI
- https://ec.europa.eu/futurium/en/system/files/ged/ai_hleg_definition_of_ai_18_december_1.pdf
- “Artificial intelligence (AI) refers to systems that display intelligent behaviour by analysing their environment and taking actions – with some degree of autonomy – to achieve specific goals.”
- “Artificial intelligence (AI) refers to systems designed by humans that, given a complex goal, act in the physical or digital world by perceiving their environment, interpreting the collected structured or unstructured data, reasoning on the knowledge derived from this data and deciding the best action(s) to take (according to pre-defined parameters) to achieve the given goal. AI systems can also be designed to learn to adapt their behaviour by analysing how the environment is affected by their previous actions. “

Definitions – Rational Agent

RATIONAL AGENT – AI: A Modern Approach book

"For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has"

Task Environment

Performance, **E**nvironment, **A**ctuators, **S**ensors – AI book

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

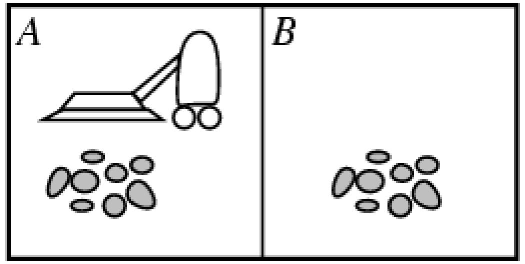
Figure 2.5 Examples of agent types and their PEAS descriptions.

Problem-solving agents

- Remember: intelligent agents maximize their performance measure
  - If a **goal** is given, an agent needs to satisfy it
- First step: goal formulation
  - (including the alignment problem)
- Problem formulation:
  1. Initial state
  2. Possible actions available to the agent
  3. Transition model (description of what each action does, ie what state transition does it result in)
  4. Goal test – determine whether a reached state = goal state
  5. Path cost – numeric cost assigned to each step/path
- **Solution** = action sequence that leads from start state to goal state
- **Optimal solution** = solution with the lowest path cost

Example problems – toy vs real-world

› Vacuum cleaner world

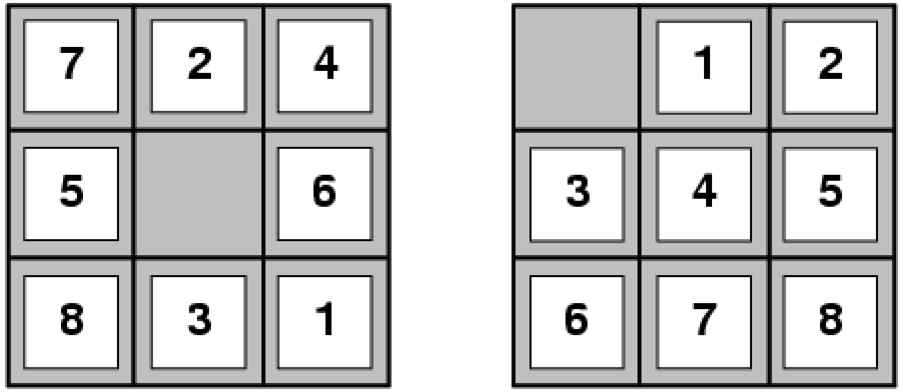


› Percepts: location and contents, e.g., [A, Dirty]

› Actions: Left, Right, Suck, NoOp

8-puzzle

- Given a 3x3 board, 8 numbered tiles and 1 free space
- Goal – given a start state, rearrange numbered tiles into a goal state

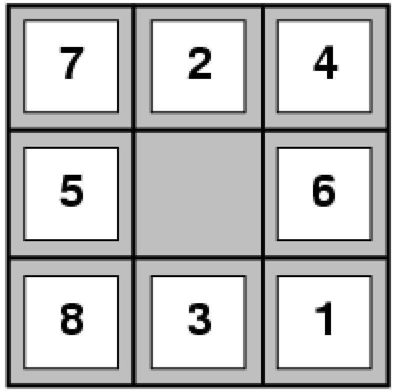


Start State

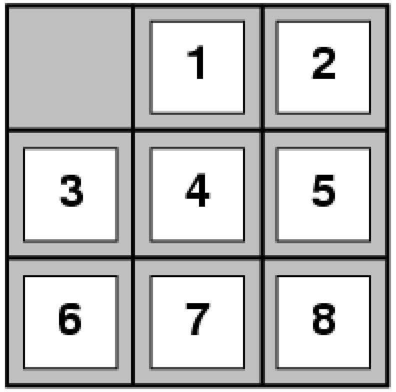
Goal State

8-puzzle

- State: ____________________
- Total number of states = ___
- Actions: __________
- Total number of actions= ___
- Goal test: _____
- Path cost: assume 1 per move



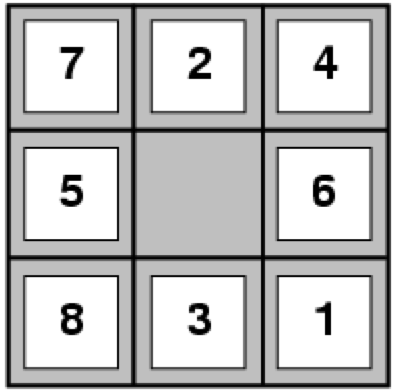
Start State



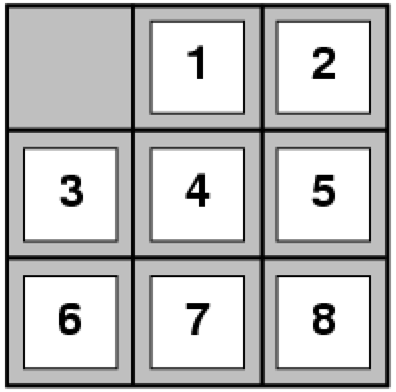
Goal State

8-puzzle

- State: locations of tiles
- Total number of states = 9! = **360k**
- Actions: move blank L, R, U, D
- Total number of actions= 4
- Goal test: goal state
- Path cost: assume 1 per move



Start State



Goal State

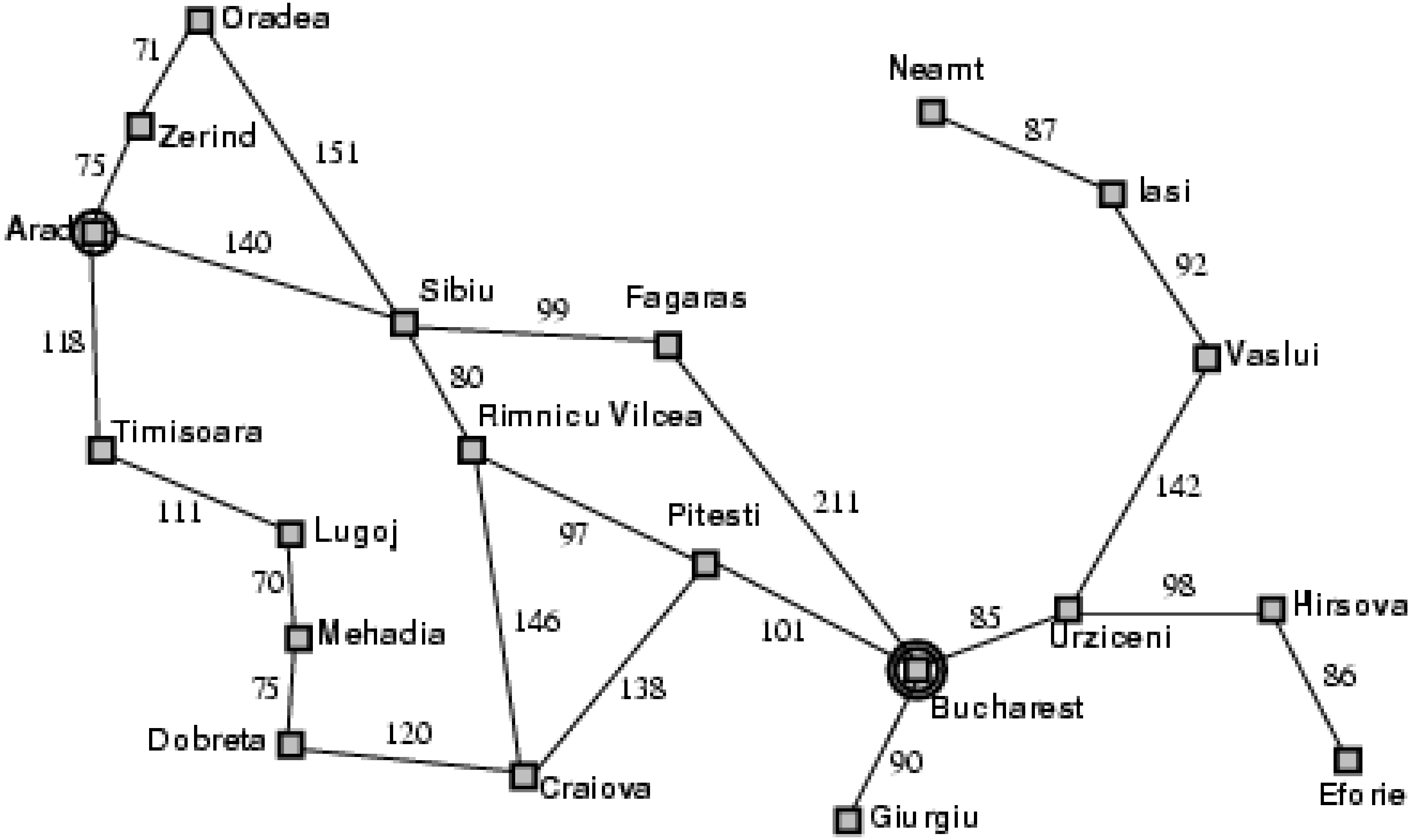
8-puzzle

- Total number of states = 9! = **360k**
- Total number of solvable states = 9!/2

- Knowing if/when the problem is solvable at all!

- Scalability issues
- 15-puzzle (4x4 rather than 3x3)
  - 20,922,789,888,000 states

Travelling in Romania



Travelling in Romania

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal: be in Bucharest
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Types of problems

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence

- Non-observable
  - Agent may have no idea where it is; solution is a sequence

- Nondeterministic and/or partially observable
  - percepts provide new information about current state
  - often interleave search, and execution

Selecting the state space

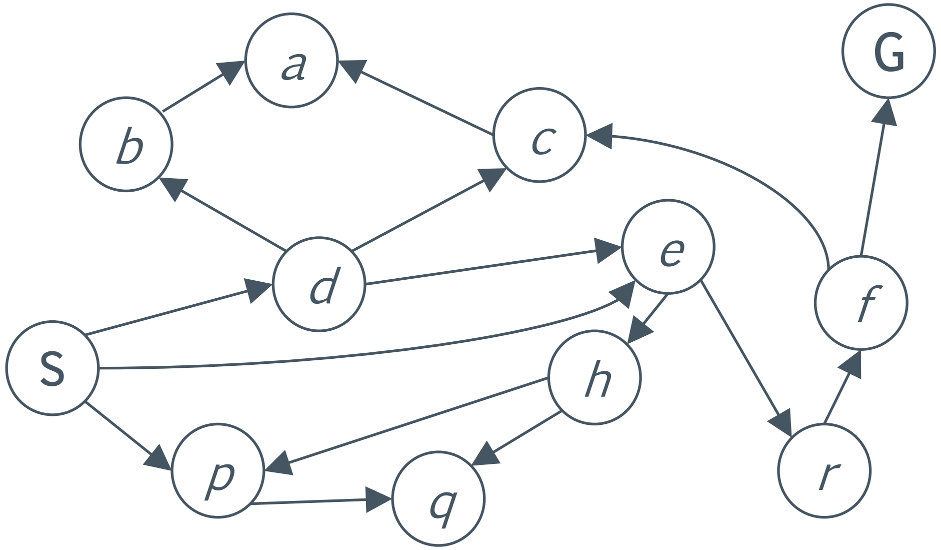
- World = all the details of the environment
- State space = only details relevant to the problem-solving this particular problem
  - Identify relevant information?
  - Observable, non-observable, partially observable?
  - Continuous or discrete?
  - If discrete, what granularity?
    - Too much or too little – state space explosion (curse of dimensionality)

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 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



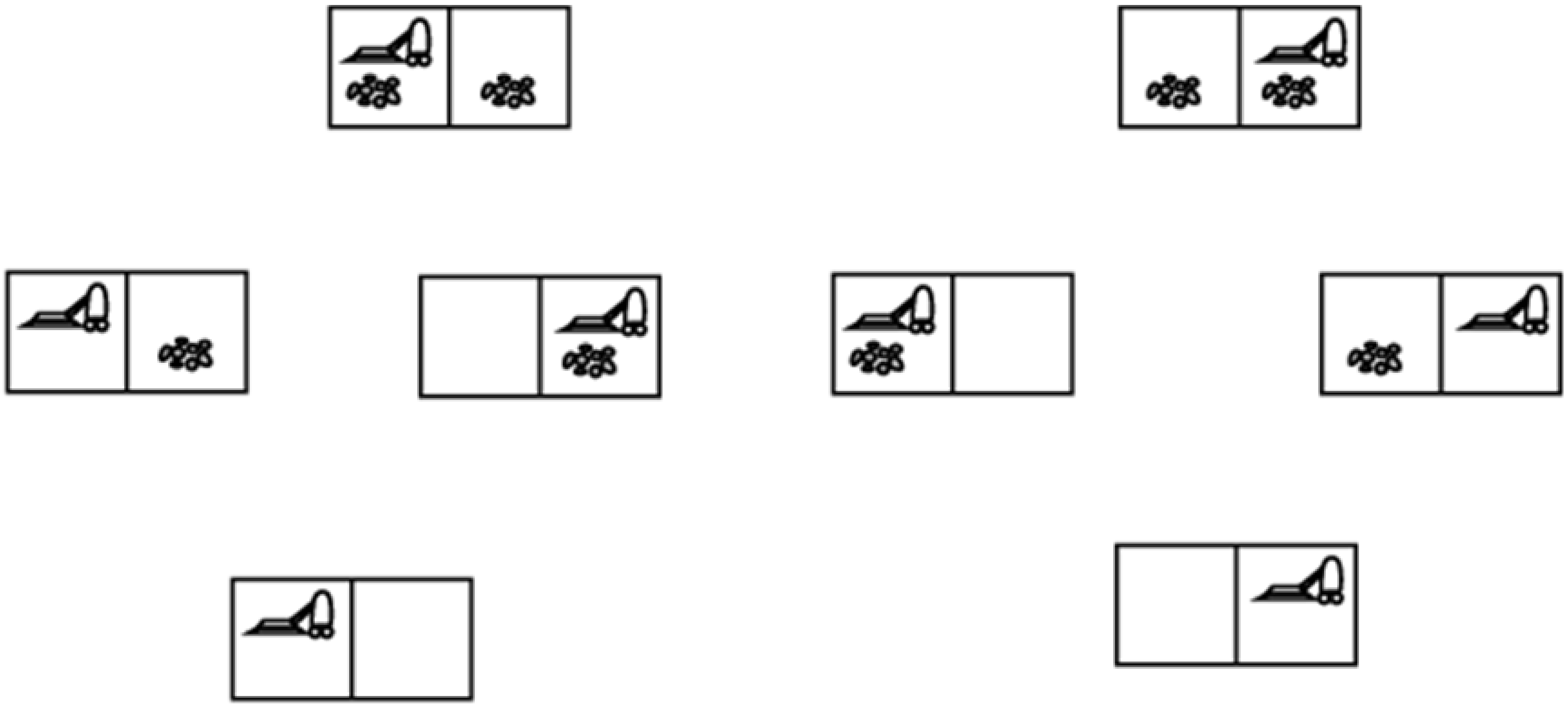
Vacuum cleaner world

- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

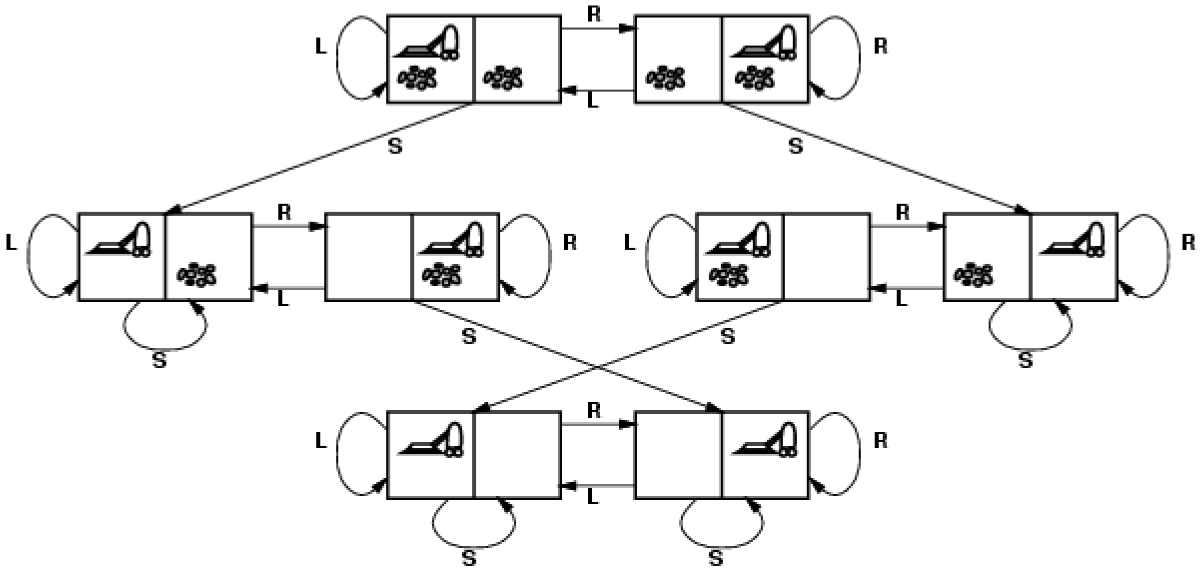
- State transitions

- Initial state needs to be given

Vacuum cleaner world – state transitions



Vacuum cleaner world – solution



Search algorithms

Outline

- Uninformed
- Informed (heuristic)

- Sophisticated search approaches
  - Continuous spaces, nondeterministic actions, uncertain environments

- Adversarial search (games/multi-agent environments)

- Constraint satisfaction problems
- Local search

Outline – Uninformed search

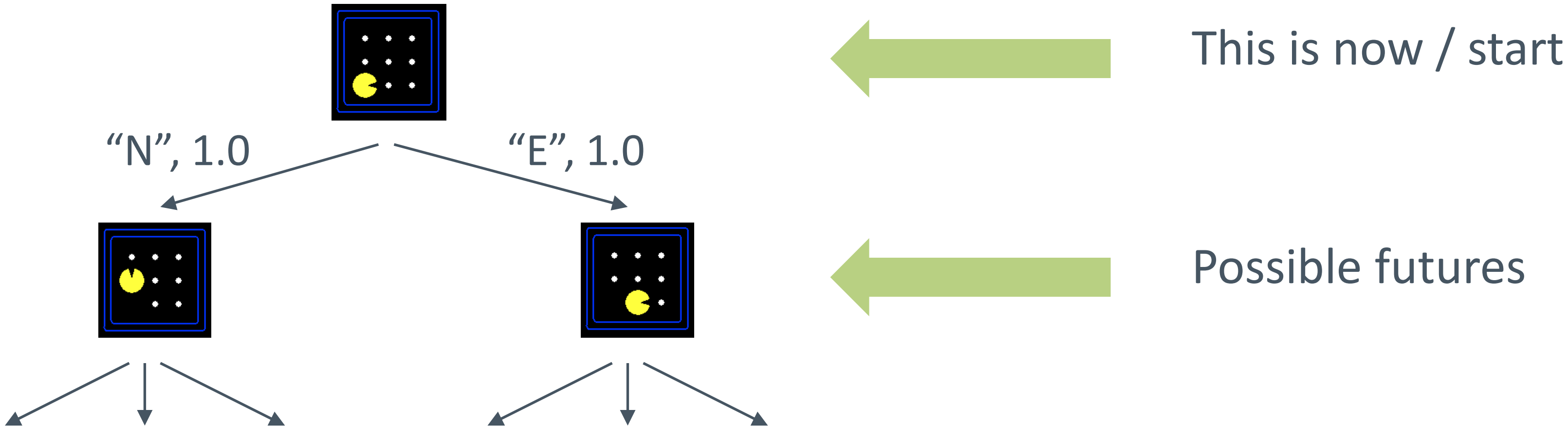
- Uninformed search algorithms:
  - Depth first search (DFS)
  - Breadth first search (BFS)
  - Iterative deepening search (IDS)
  - Uniform cost search (UCS)

Tree-search algorithms

Tree-search algorithms

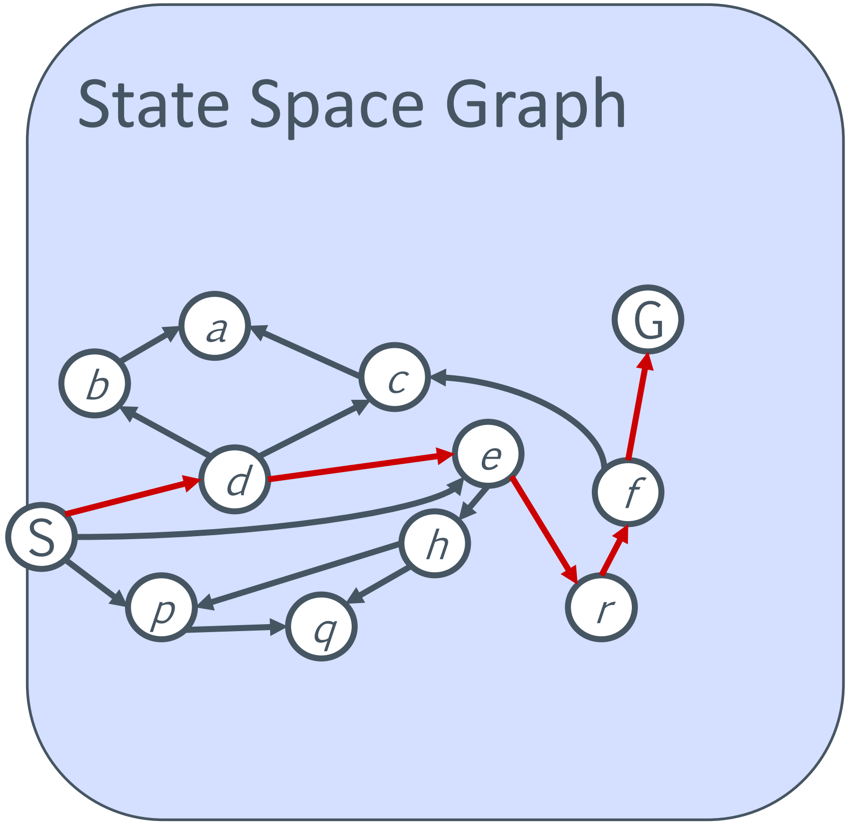
- Branches correspond to actions, nodes correspond to states
- Root of the tree = start state
- **Expand** the current state = apply each (legal) action to the current state, thereby generating new set of states (child nodes of the previous node/state)
- Frontier (fringe) – set of all leaf nodes available for expansion at any given point
- Same basic structure = differ by how they choose which state to expand next – **search strategy**

Search Trees



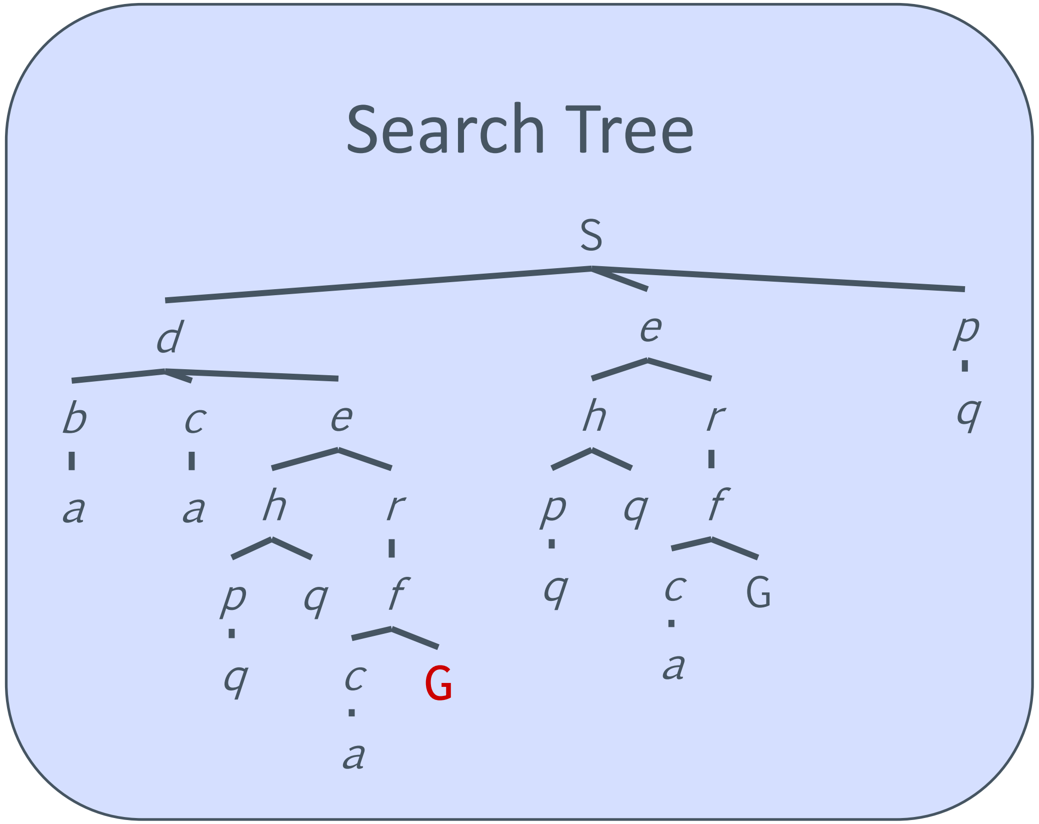
- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

State Space Graphs vs. Search Trees

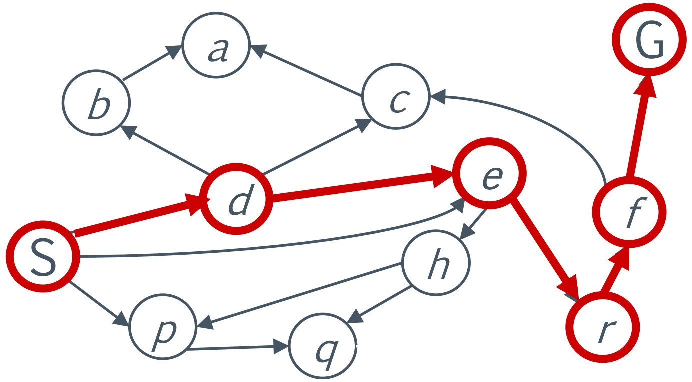


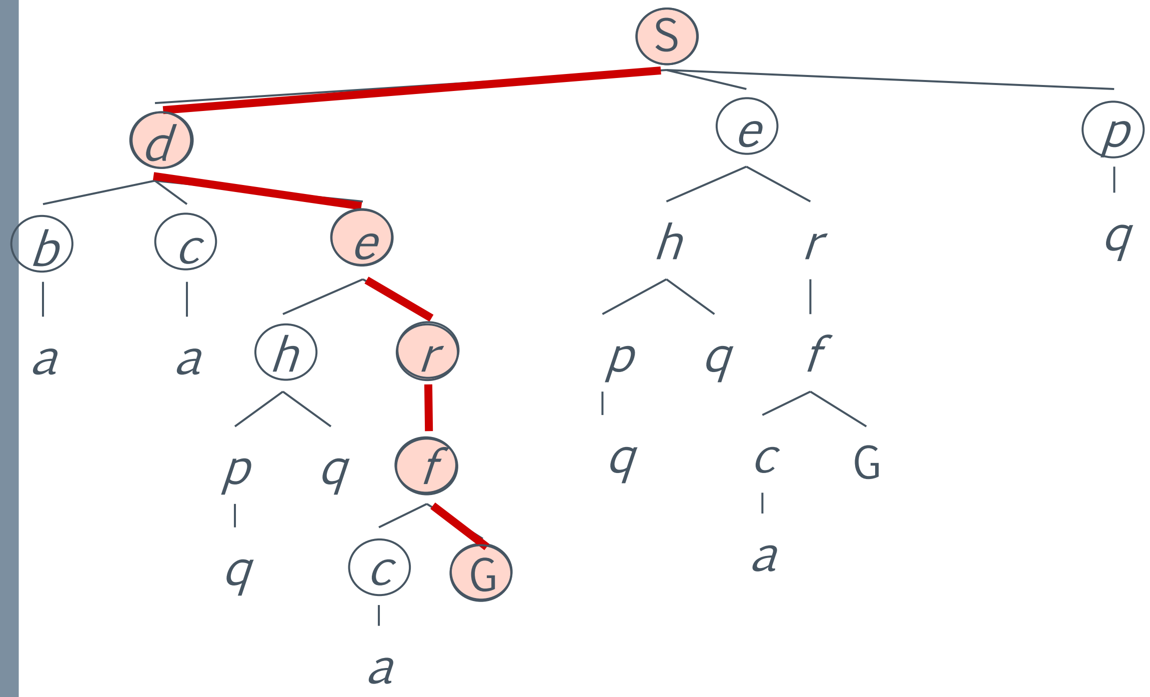
Each NODE in in the search tree is an entire PATH in the state space graph.

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



Example: Tree Search





~~s~~

~~s → d~~

s → e

s → p

s → d → b

s → d → c

~~s → d → e~~

s → d → e → h

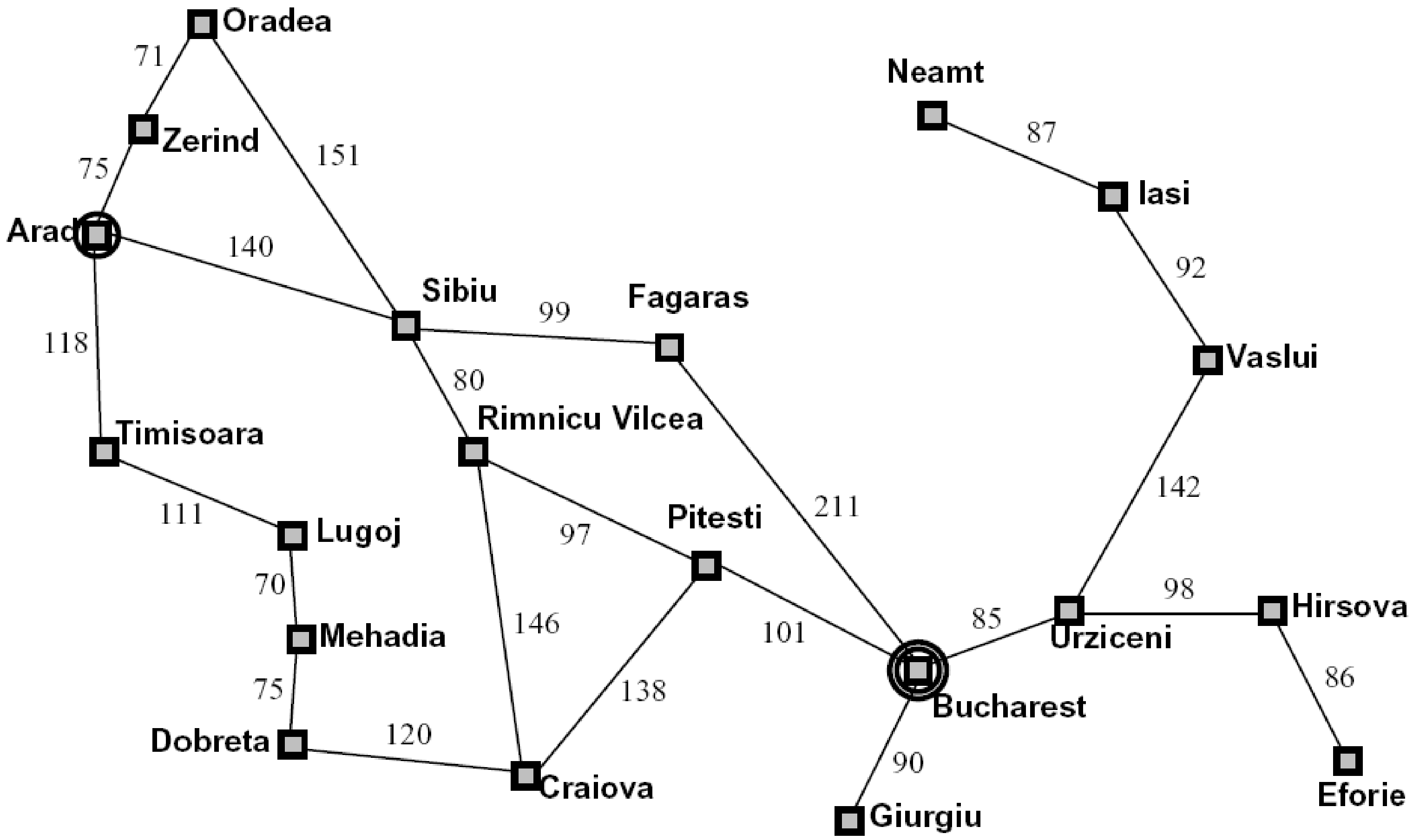
~~s → d → e → r~~

~~s → d → e → r → f~~

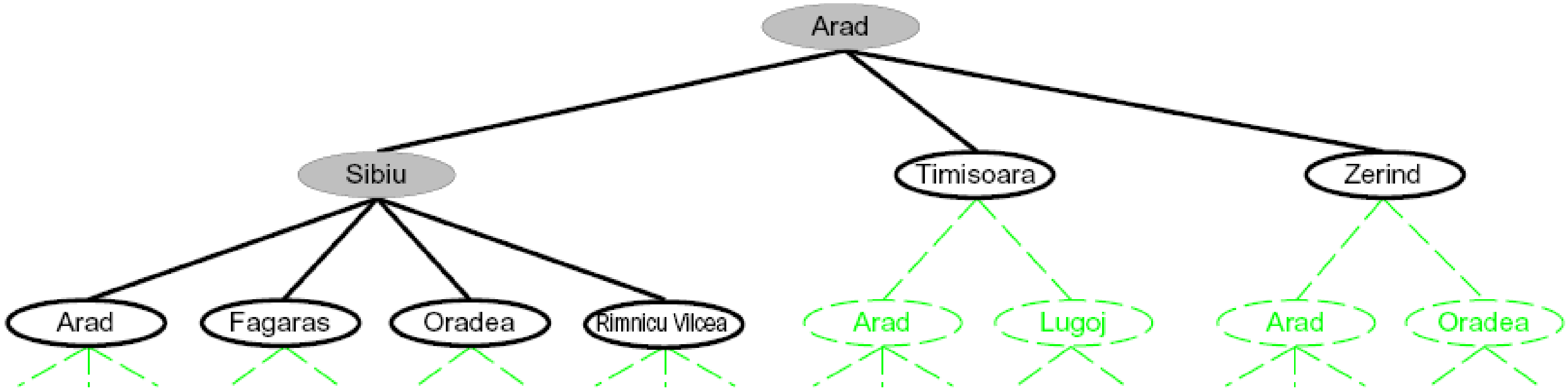
s → d → e → r → f → c

~~s → d → e → r → f → G~~

Search Example: Romania



Searching with a Search Tree



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

General tree-search algorithm

```
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
```

General tree-search algorithm

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    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 SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;   ACTION[s] ← action;   STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Search strategies

- A search strategy is defined by picking the **order of node expansion**
- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity:** number of nodes generated
  - **space complexity:** maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree (max number of successors of any node)
  - *d:* depth of the least-cost solution (shallowest goal node)
  - *m*: maximum depth of the state space (may be ∞)

Uninformed search strategies

› Uninformed search strategies use only the information available in the problem definition

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

Uninformed search design choices

Queue for frontier

- FIFO
- LIFO
- Priority queue

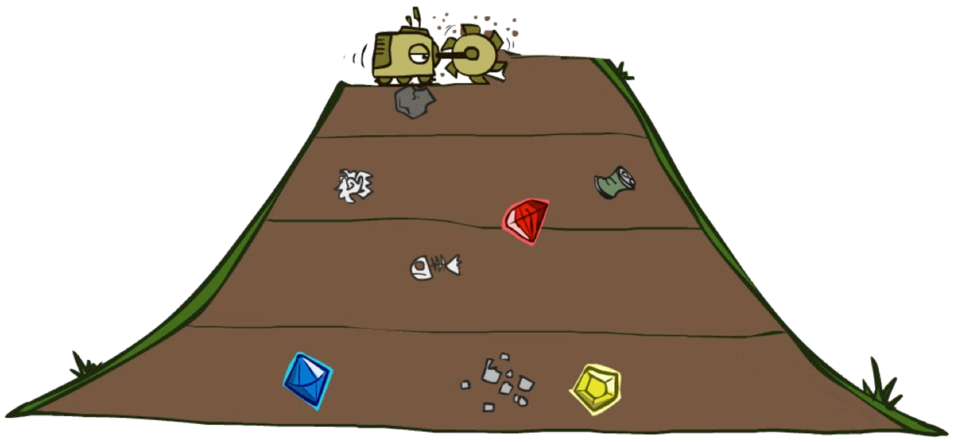
Goal-Test

- Do goal test when a node is inserted or removed from the frontier?

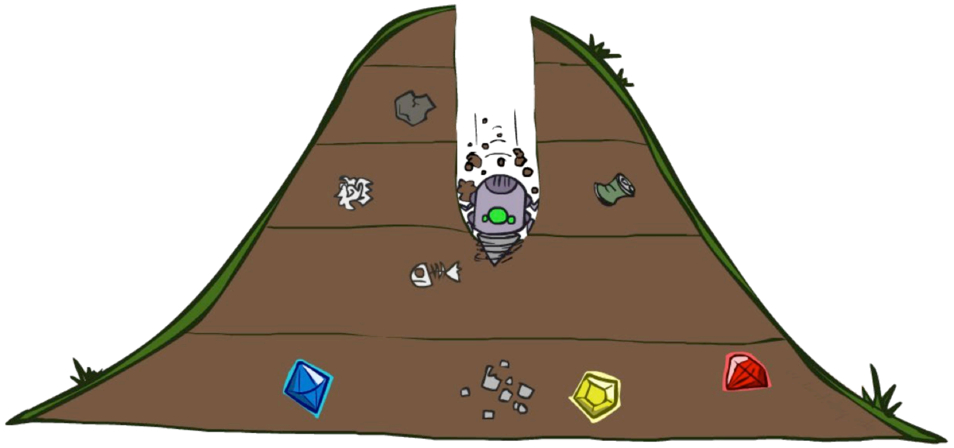
Tree Search vs. Graph Search

Breadth-first search vs Depth-first search

BFS



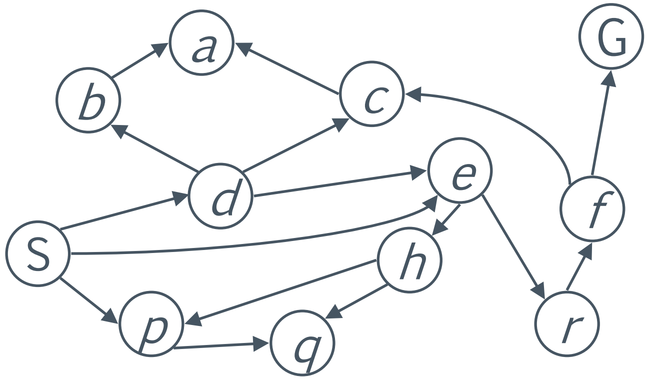
DFS

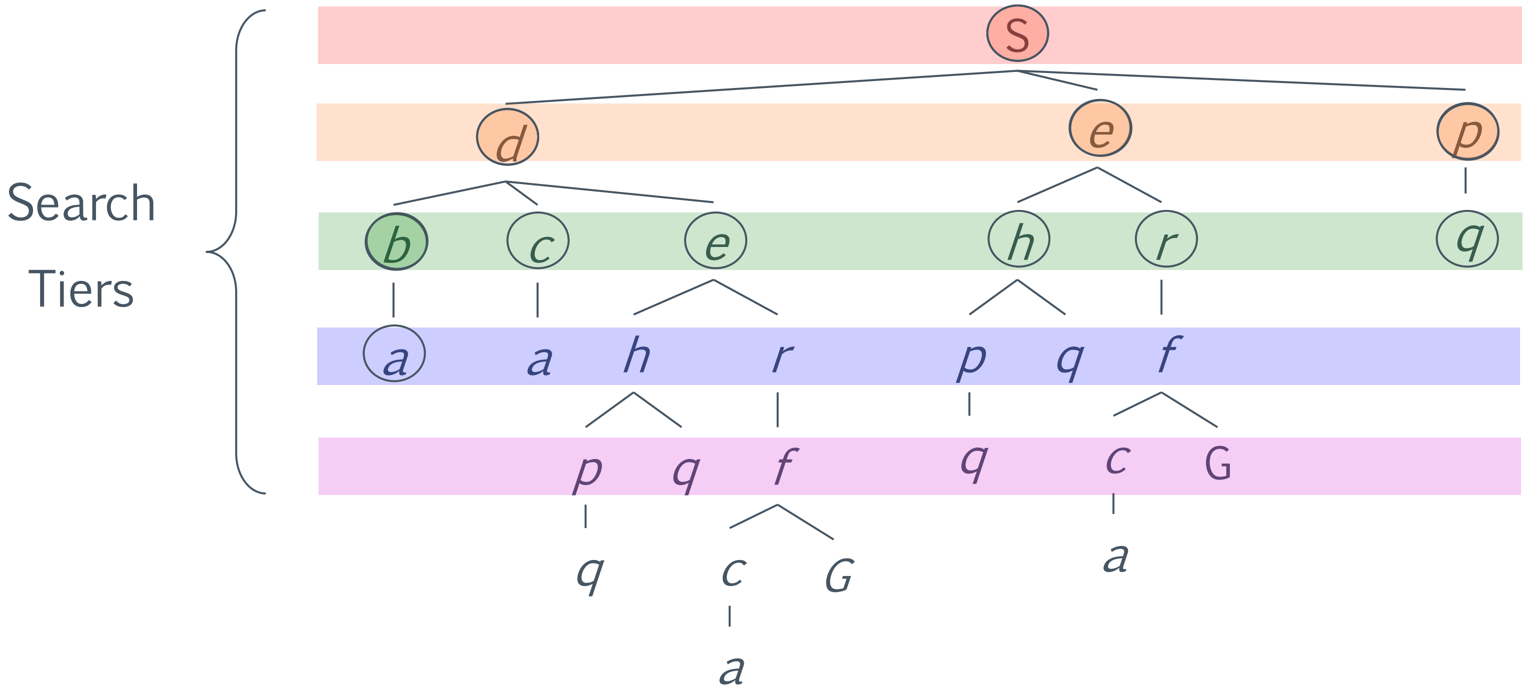


Breadth-First Search

Strategy: expand a shallowest node first

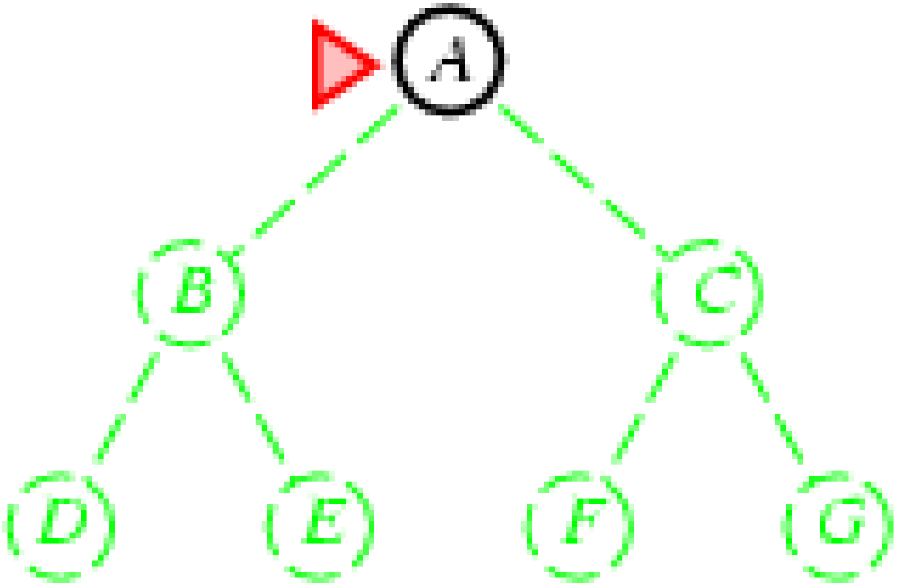
Implementation: Fringe is a FIFO queue





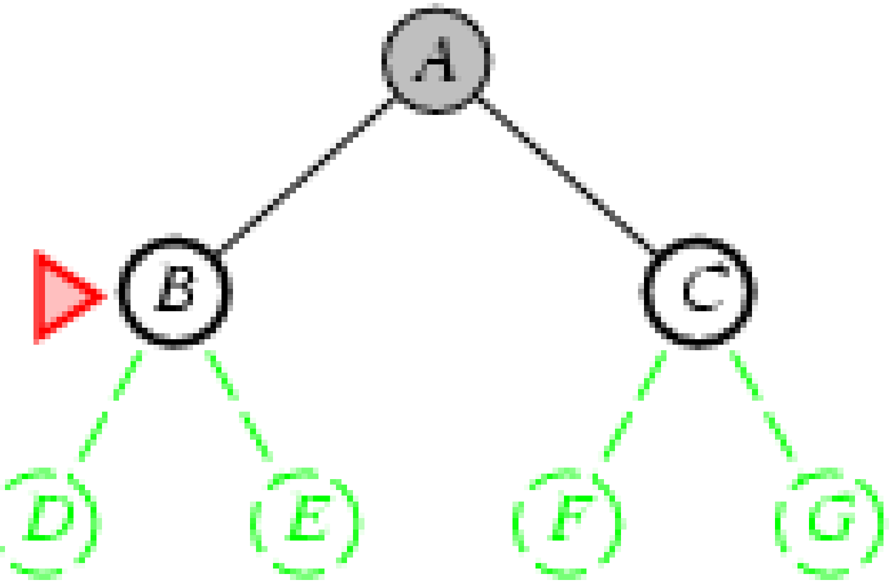
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

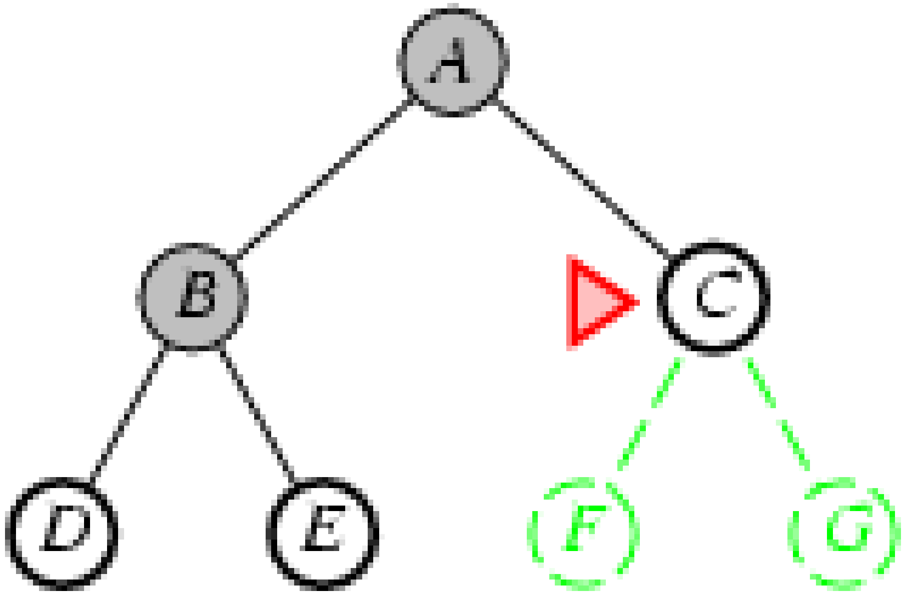


Breadth-first search

› Expand shallowest unexpanded node

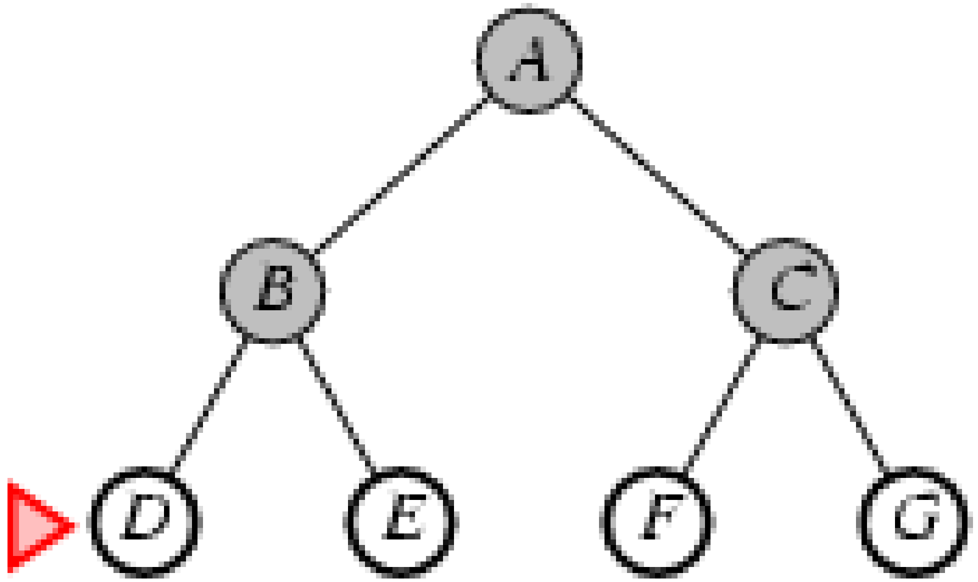
› Implementation:

  – *fringe* is a FIFO queue, i.e., new successors go at end

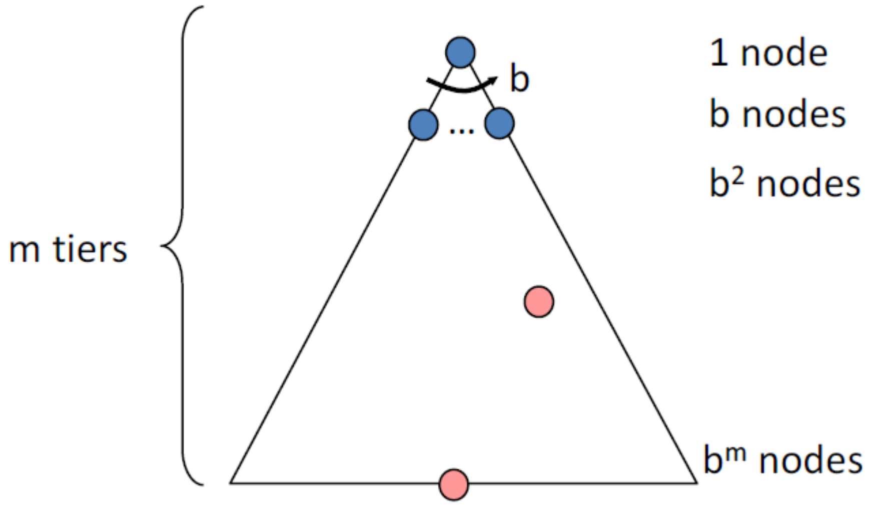


Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search



Complete?

- Yes (if b is finite)

Time?

- $1 + b + b^2 + b^3 + \ldots + b^d$
  $= O(b^d)$ (*) , i.e., exp. in d

Space?

- $O(b^d)$ (keeps every node in memory either fringe or path to fringe)

Optimal?

- Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec

so 24hrs = 8640GB.

(*) The time complexity is $O(b^{d+1})$ if the goal test is applied when the node is expanded rather than generated

Uniform-cost search (cost-sensitive search)

Expand least-cost unexpanded node

- g(n) cost function for a given path to the node n

Implementation:

- Fringe = priority queue ordered by path cost, lowest first

Goal test is applied to a node when it is selected for expansion

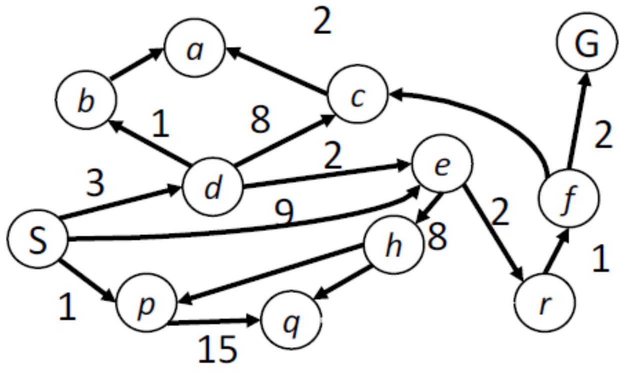
A test is added in case a better path is found to a node currently on the frontier

Equivalent to breadth-first if step costs all equal

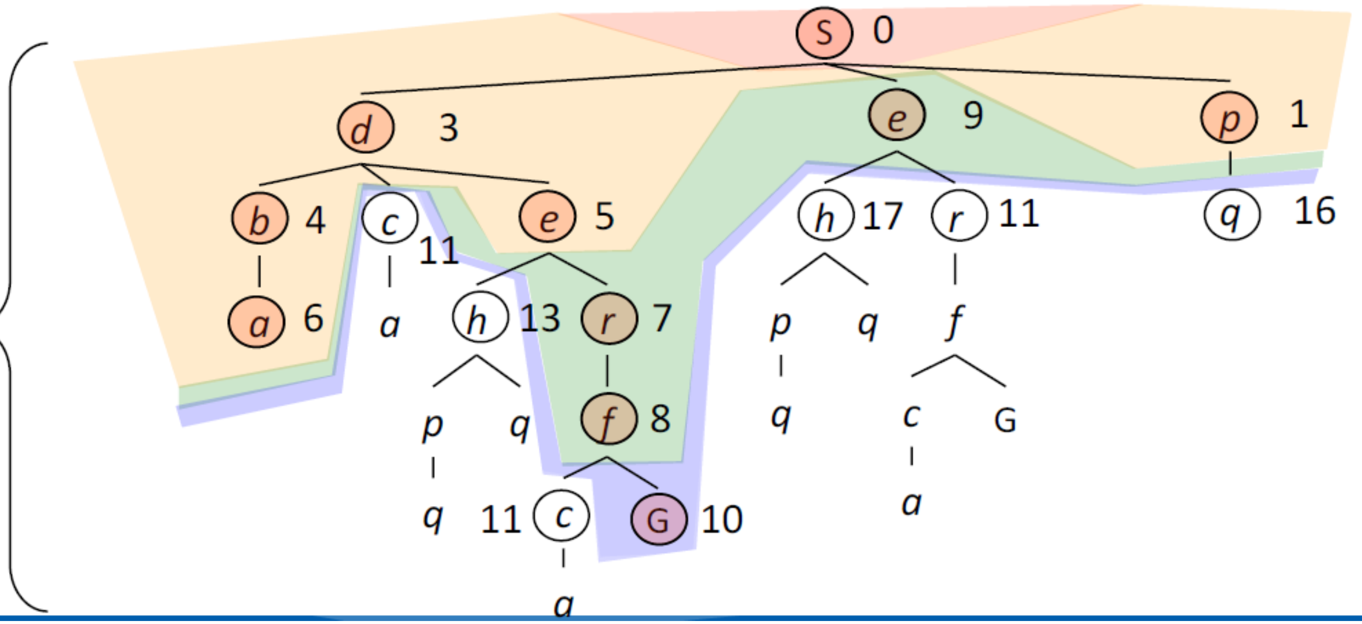
Uniform-cost search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)



Cost contours



Uniform-cost search

Expand least-cost unexpanded node

Implementation:

– Fringe = priority queue ordered by path cost, lowest rst

Equivalent to breadth-first if step costs all equal

Complete?

– Yes, if step cost $\geq \epsilon$

Time?

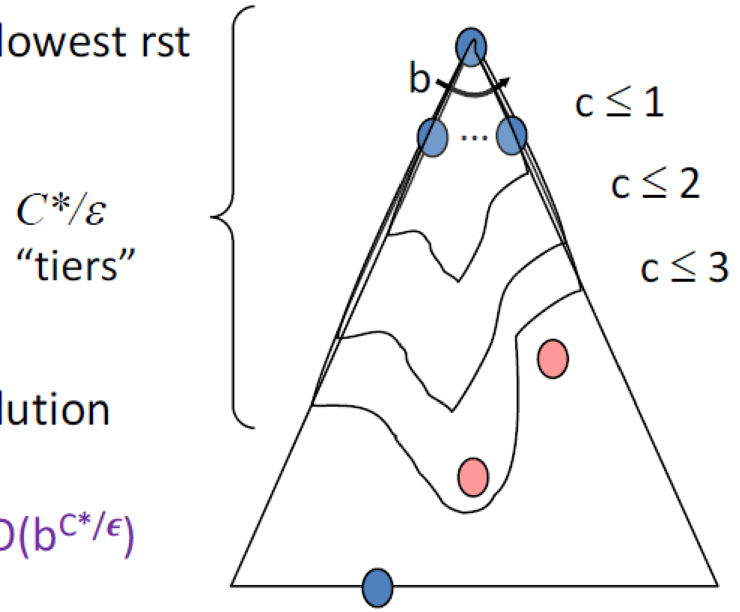
– # of nodes with g ≤ cost of optimal solution, $O(b^{C^*/\epsilon})$ where C^* is the cost of the optimal solution

Space?

– # of nodes with g ≤ cost of optimal solution, $O(b^{C^*/\epsilon})$

Optimal?

– Yes, nodes expanded in increasing order of g(n)

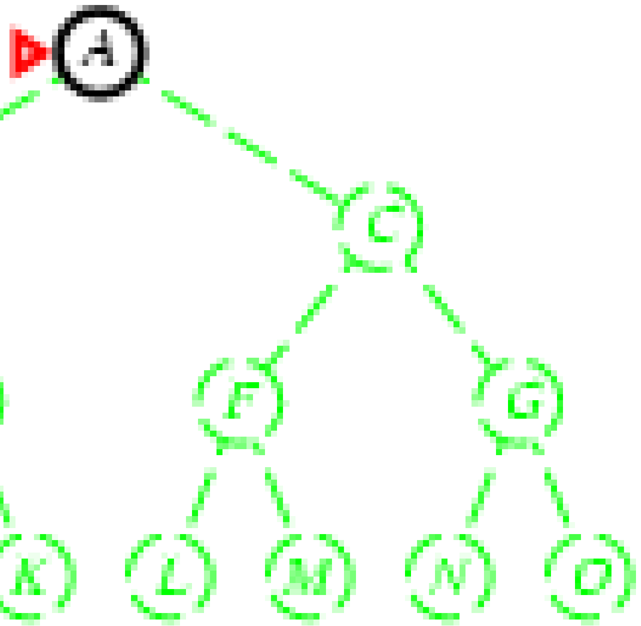


Depth-first search

› Expand deepest unexpanded node

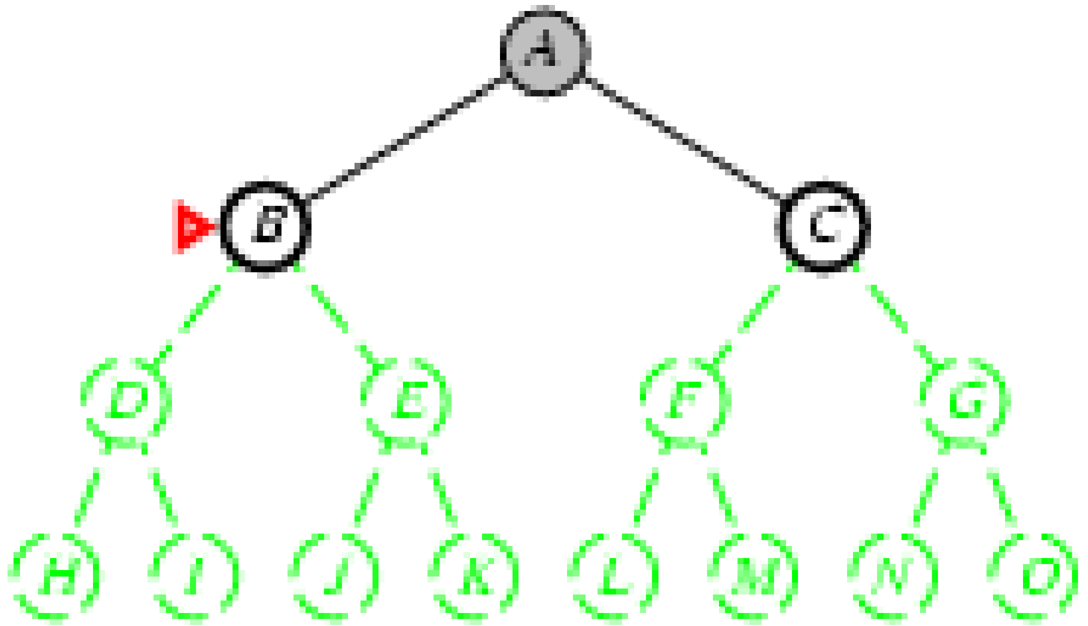
› Implementation:

  – *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- › Expand deepest unexpanded node
- › Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

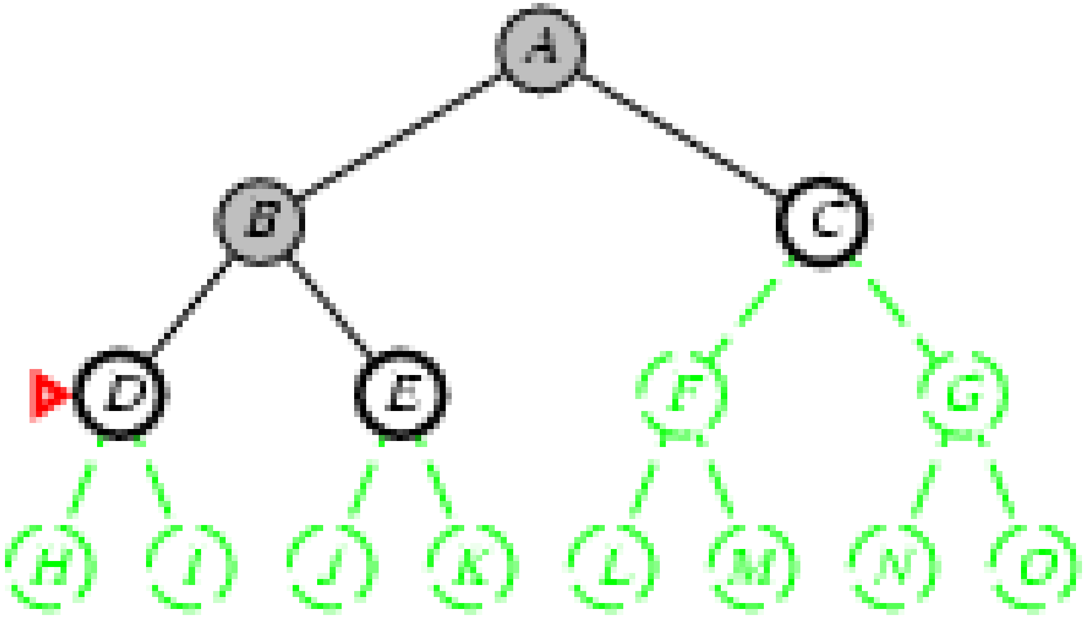


Depth-first search

› Expand deepest unexpanded node

› Implementation:

  – *fringe* = LIFO queue, i.e., put successors at front

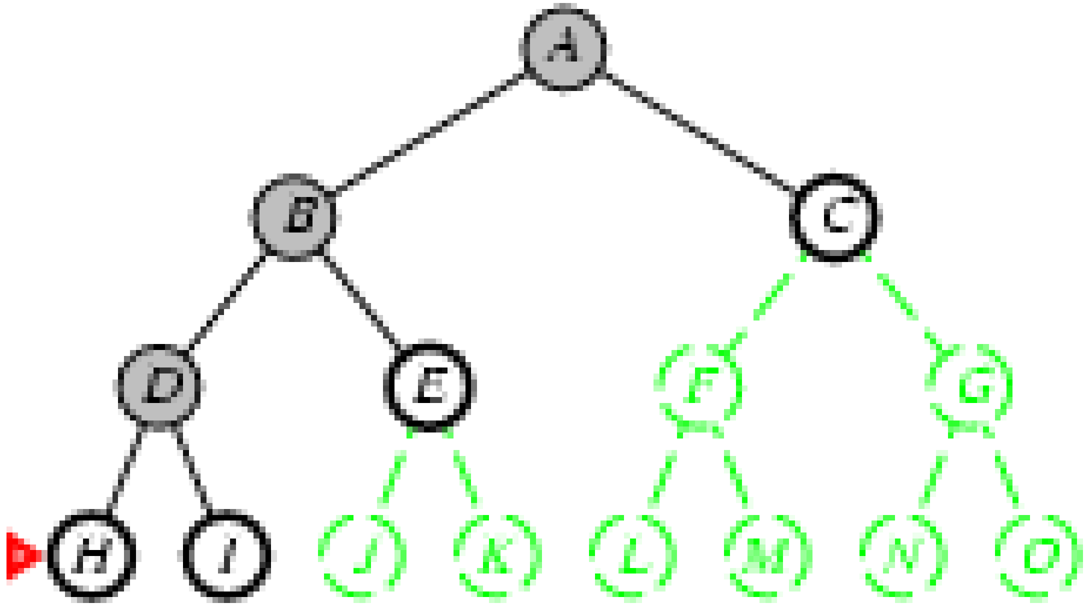


Depth-first search

› Expand deepest unexpanded node

› Implementation:

  – *fringe* = LIFO queue, i.e., put successors at front

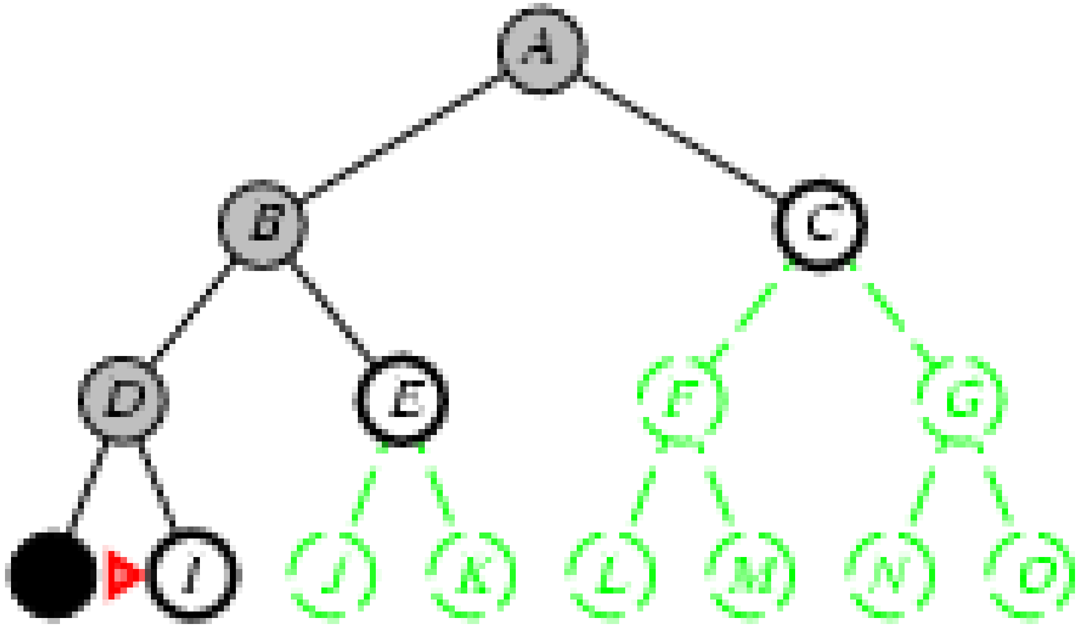


Depth-first search

› Expand deepest unexpanded node

› Implementation:

  – *fringe* = LIFO queue, i.e., put successors at front

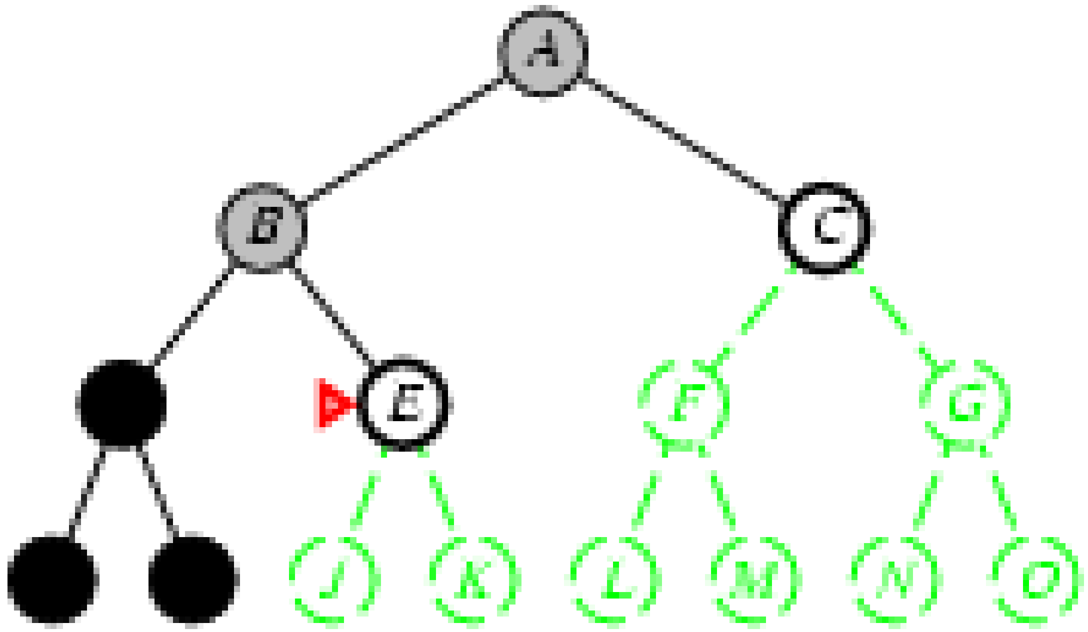


Depth-first search

› Expand deepest unexpanded node

› Implementation:

  – *fringe* = LIFO queue, i.e., put successors at front



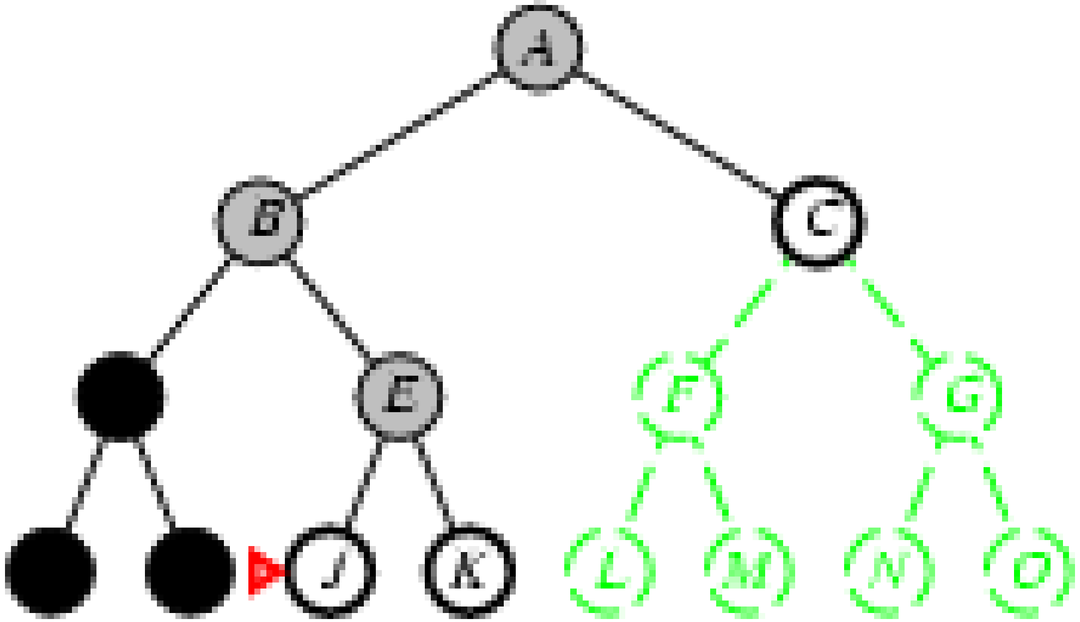
Depth-first search

› Expand deepest unexpanded node

› Implementation:

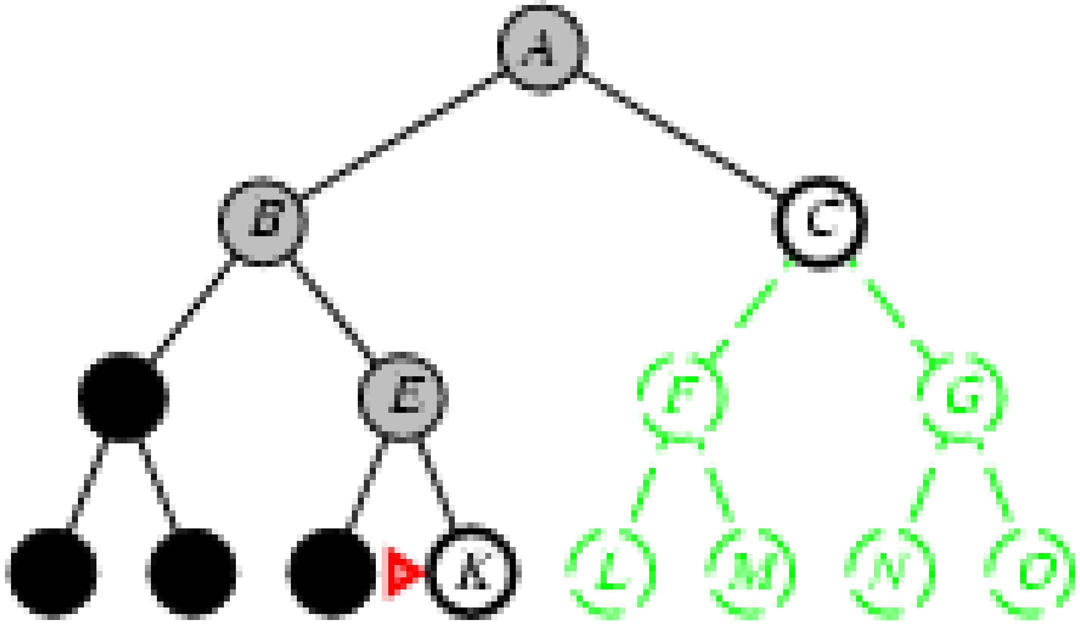
  – *fringe* = LIFO queue, i.e., put successors at front

  –



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

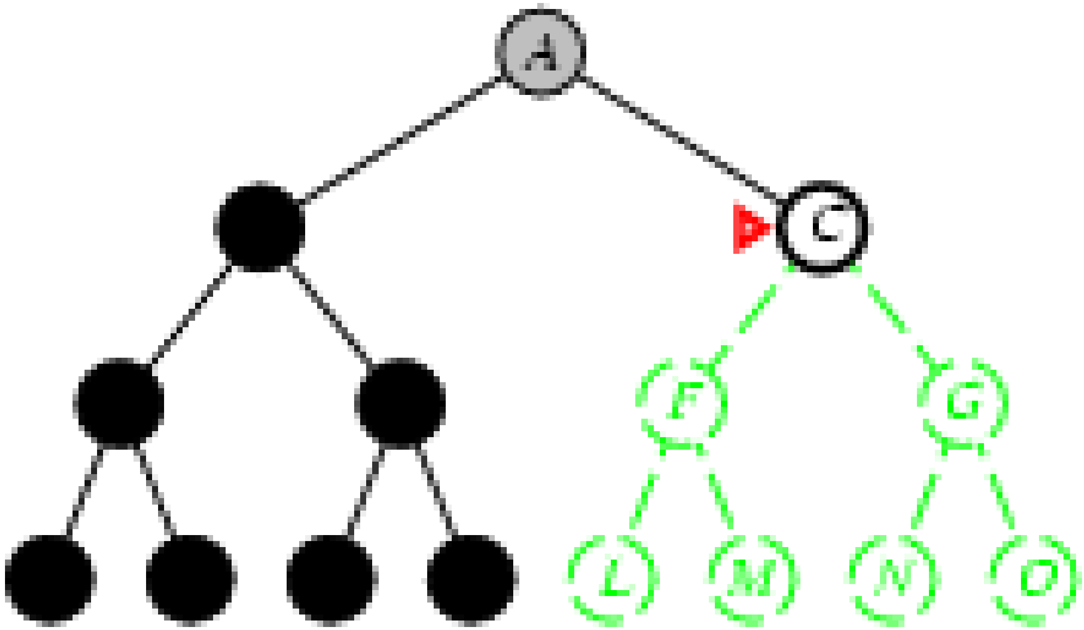


Depth-first search

› Expand deepest unexpanded node

› Implementation:

  – *fringe* = LIFO queue, i.e., put successors at front

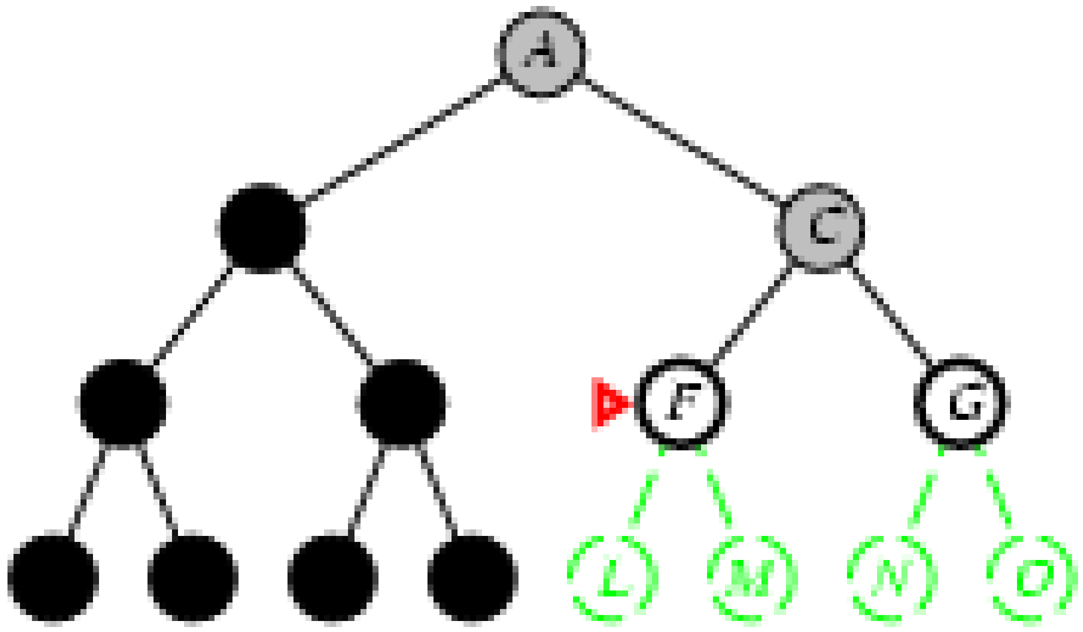


Depth-first search

› Expand deepest unexpanded node

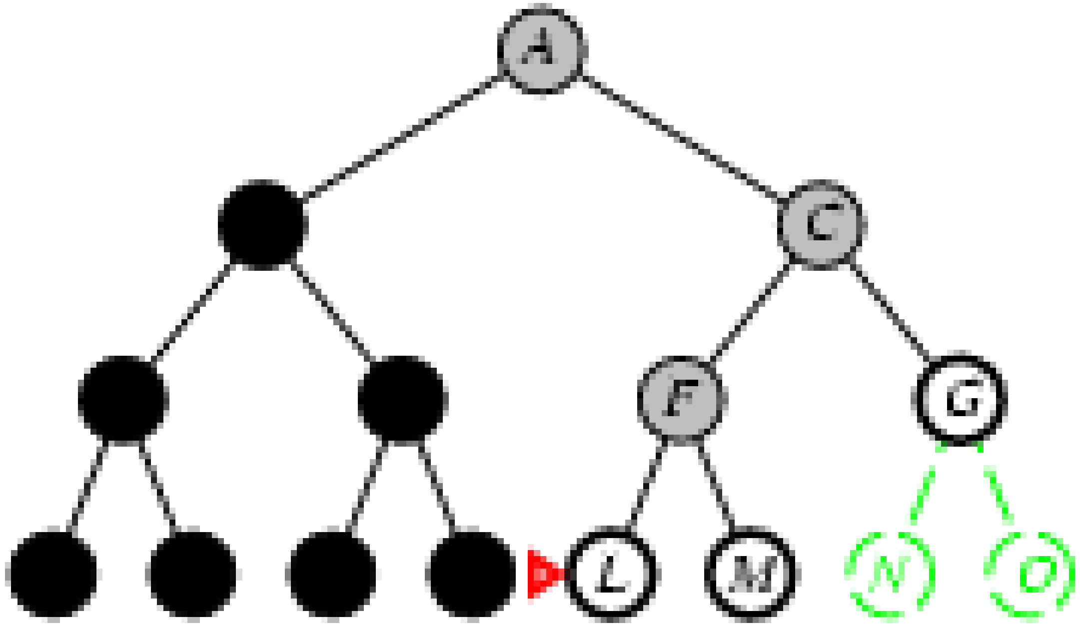
› Implementation:

  – *fringe* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

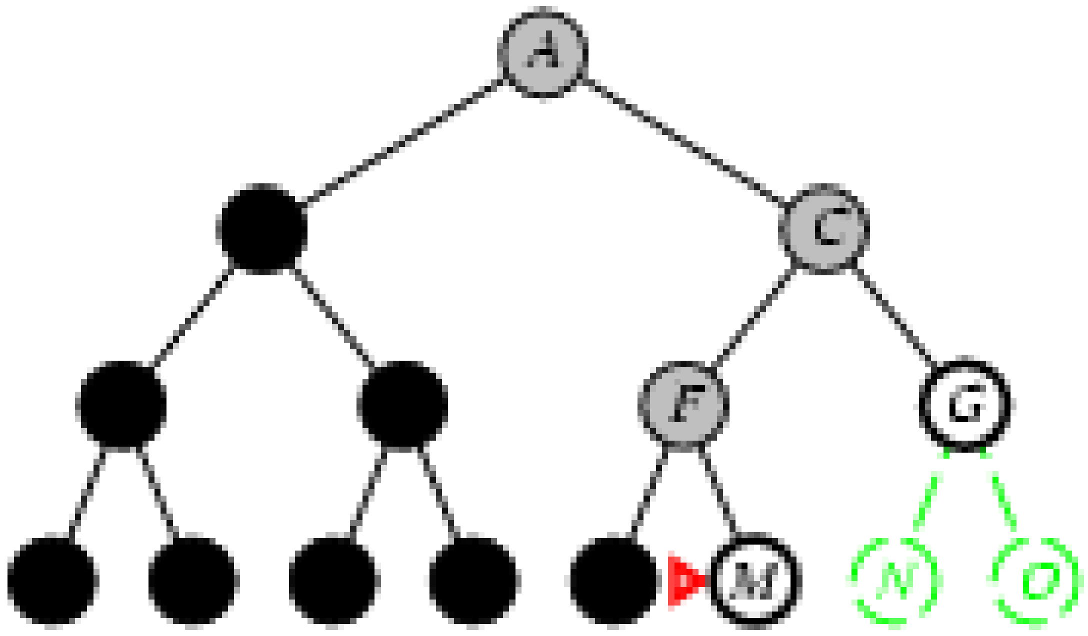


Depth-first search

› Expand deepest unexpanded node

› Implementation:

– *fringe* = LIFO queue, i.e., put successors at front



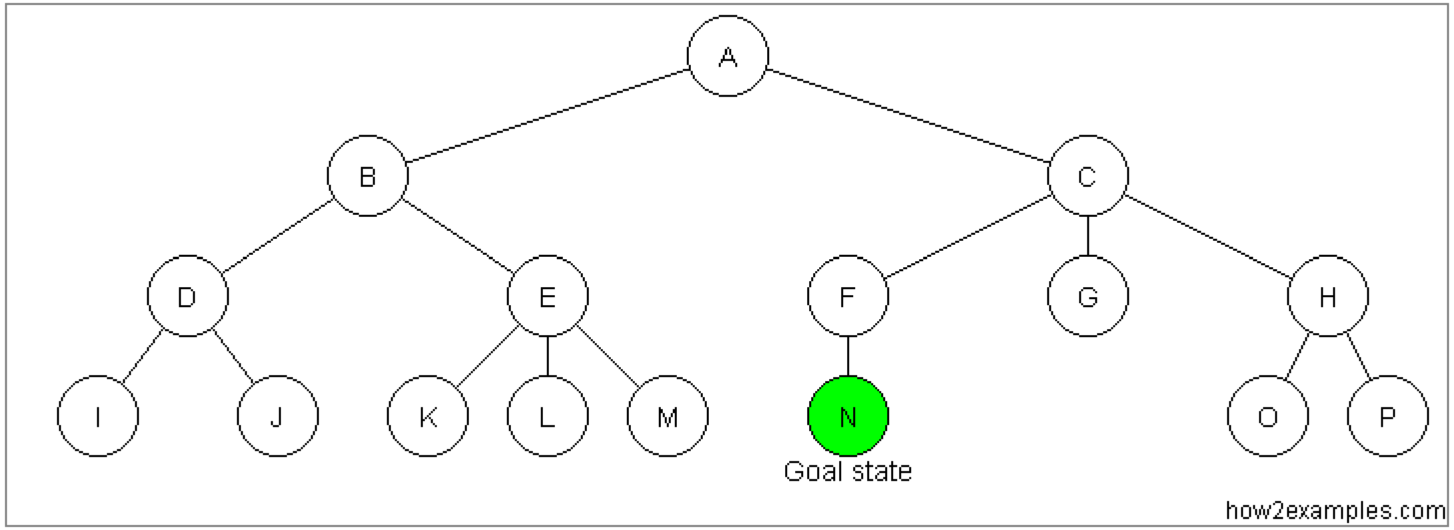
Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    - → complete in finite spaces

- Time? $O(b^m)$: terrible if m is much larger than b
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

DFS vs BFS exercise

- show sequences of nodes visited using BFS and DFS



Depth-limited search

- Limit the depth, and nodes past that depth are treated as if they have no successors
  - Can base it on the knowledge of the problem (domain knowledge)
  - Incomplete

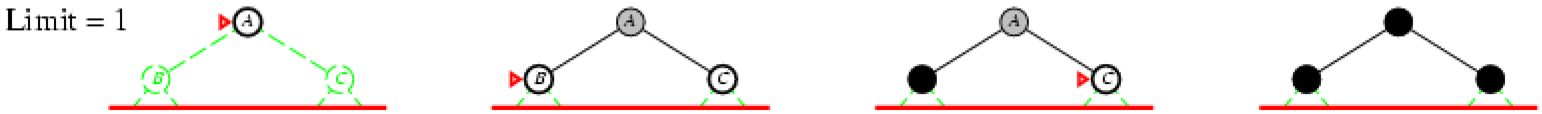
Iterative deepening search

- Find the best depth limit, by gradually increasing the limit (0, 1, 2...) until the goal is found
- Combines the benefits of DFS and BFS

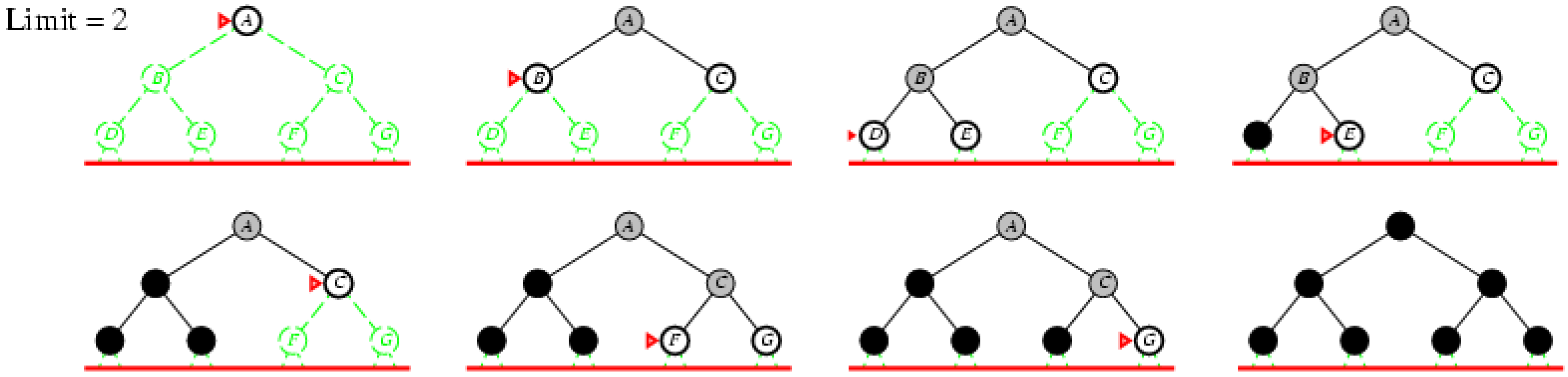
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening search l=0

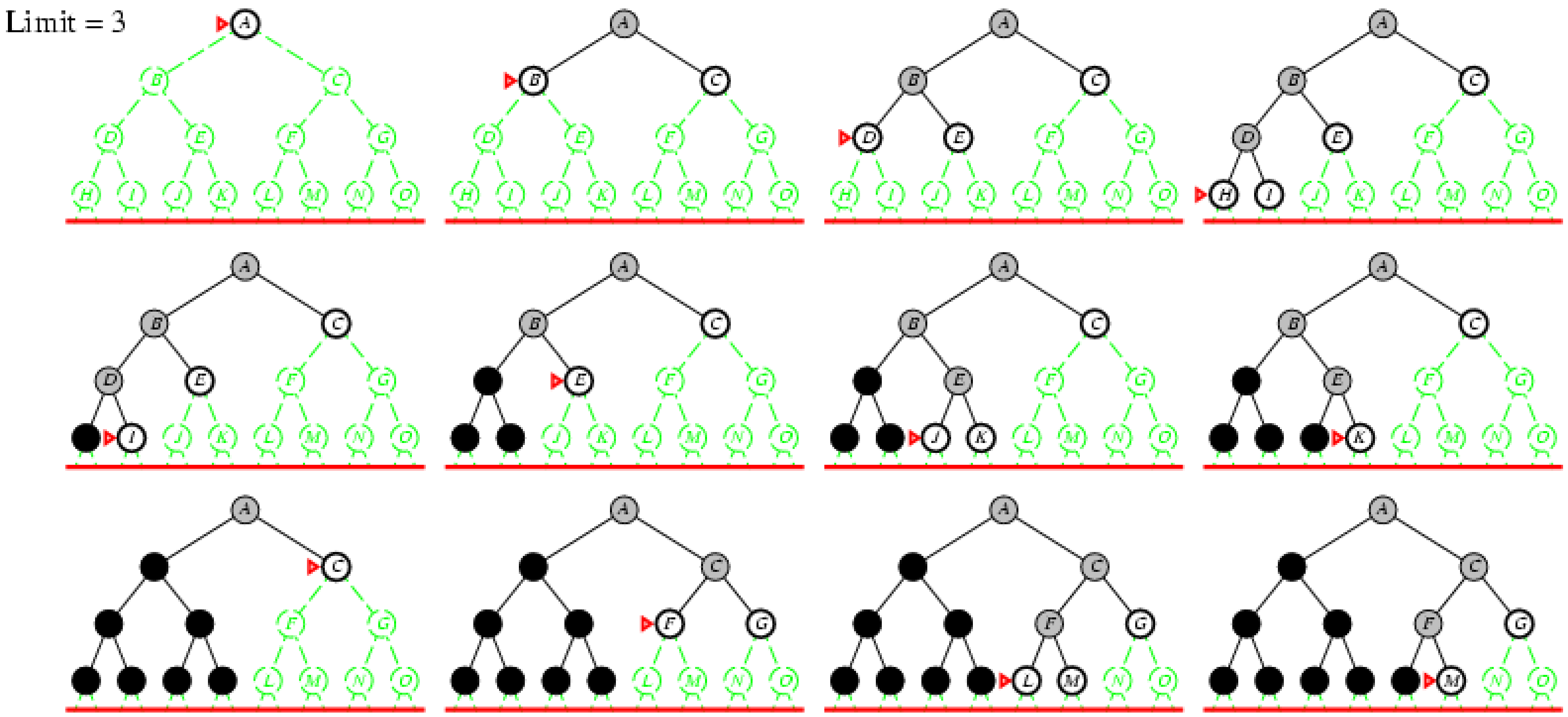
Iterative deepening search $l = 1$



Iterative deepening search l=2



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete?

– Yes

Time?

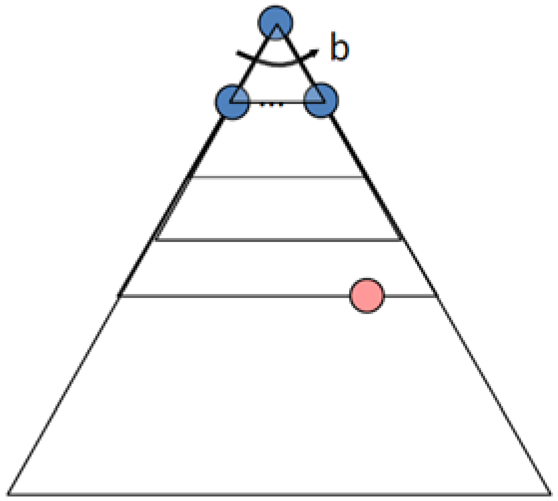
– $(d + 1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?

– $O(bd)$

Optimal?

– Yes, if step cost = 1
  Can be modified to explore uniform-cost tree

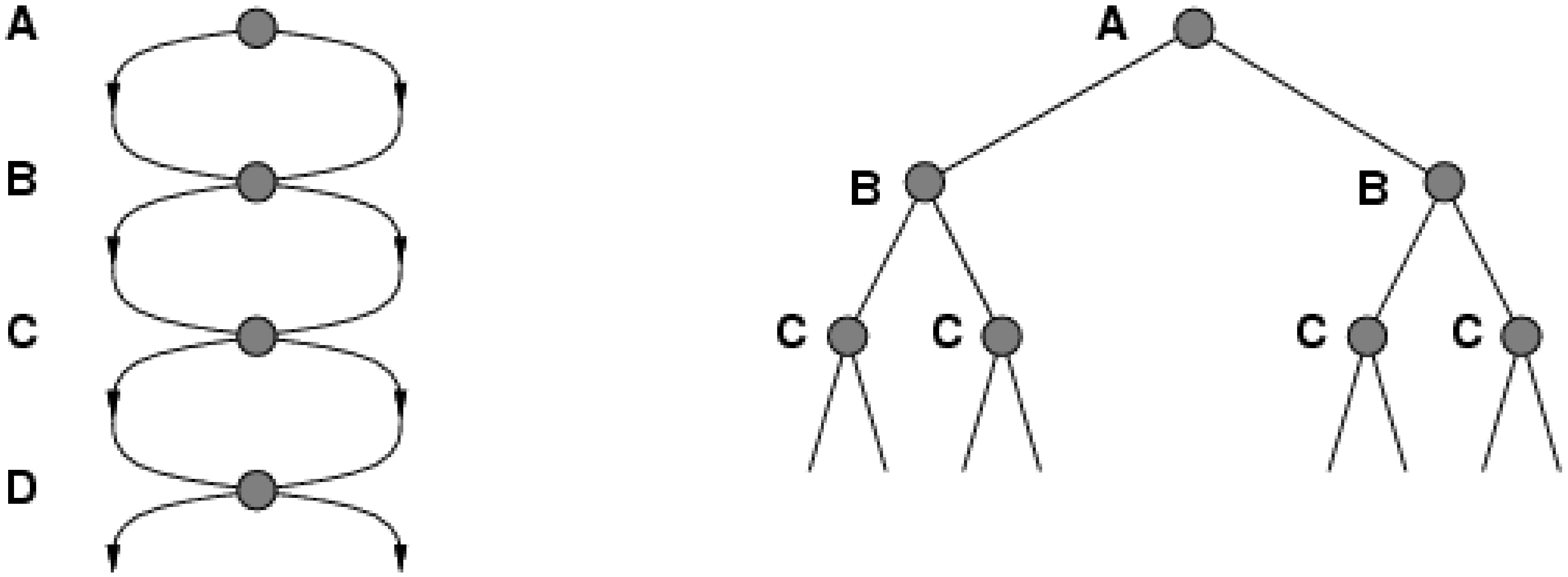


Uninformed search: summary

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!

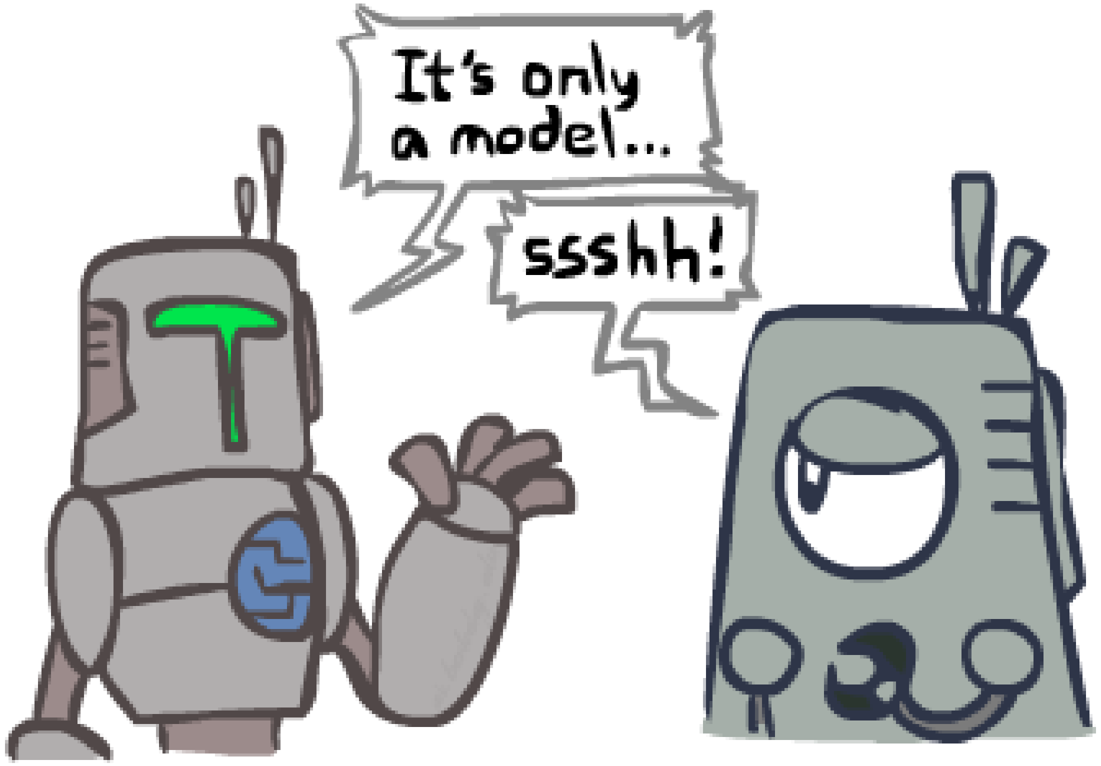


Graph search

```
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
```

Search and Models

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world!
  - Planning is all "in simulation"
  - Your search is only as good as your models…



Kind of problems to practice/solve

- 5 elements required by problem-solving agents
- Specify state space for a problem (identifying relevant and irrelevant environment information) – list complete list of states
- Draw state transitions diagram for a problem
- Differences (advantages/disadvantages) between various uninformed search techniques
  - Know each algorithm and compare performance
- Show a trace of solving a problem using x,y,z approaches
- etc

Search Gone Wrong?