Artificial Intelligence

Search…

Lecture 3

CS 444 – Spring 2019
Dr. Kevin Molloy
Department of Computer Science
James Madison University

JAMES MADISON UNIVERSITY

Outline for Today

- Problem-solving agents

- Problem types

Agents and Environments

```
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 ← Recommendation(seq, state)
    seq ← Remainder(seq, state)
    return action
```

Note: this is offline problem solving, solutions executed “eyes closed”.

Example: A trip in Romania

On holiday in Romania, current in Arad.

Flight leaves tomorrow from Bucharest.

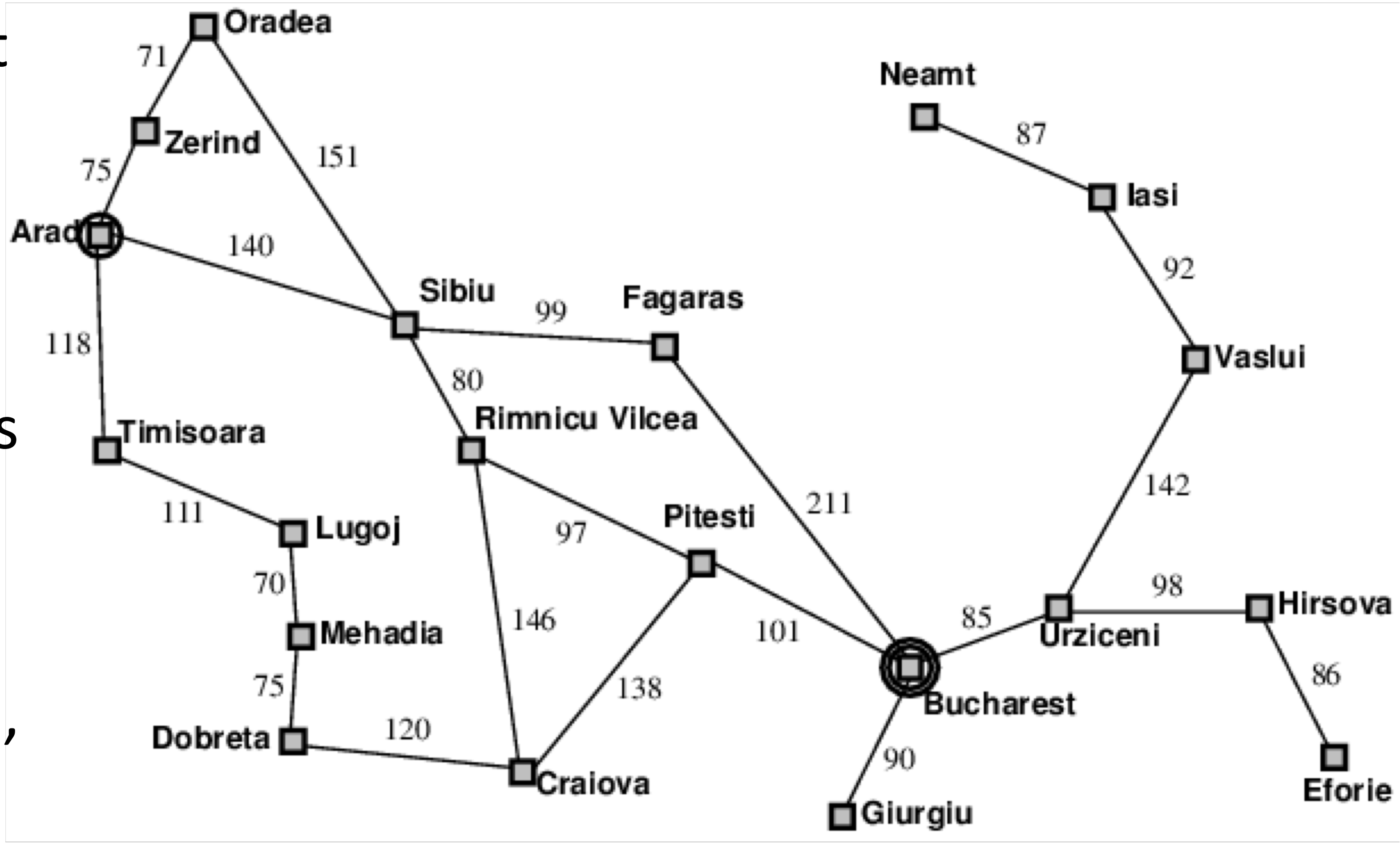
Formulate goal: be in Bucharest

Formulate problem:

- states: various cities
- Actions: drive between cities

Find solution:

- Sequence of cities, e.g. Arad, Sibiu, Fagaras, Bucharest



Problem Types

- Fully-observable, Known, Deterministic → single-state problem

  Agent knows exactly which state it will be in; solution is a sequence of actions that can be executed eyes closed

  open loop: no need to sense environment during execution

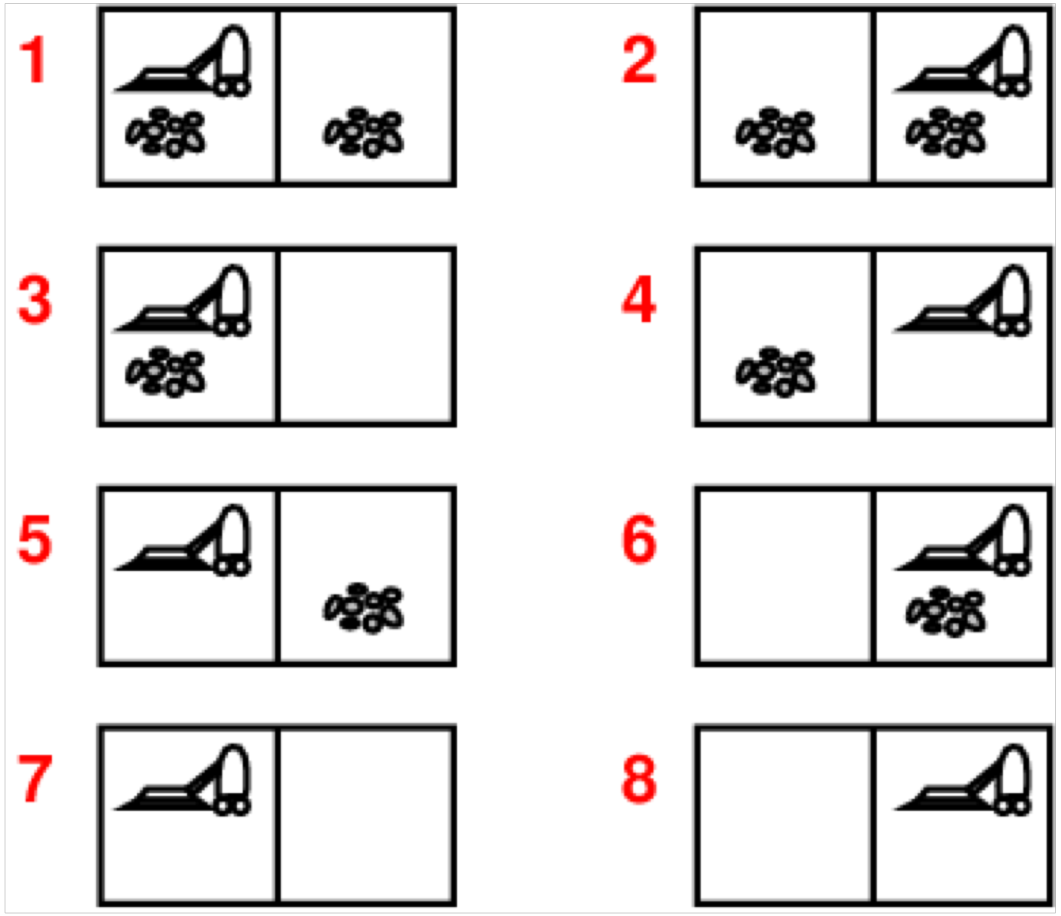
- Non-observable → conformant problem

  Agent may have no idea where it is; solution (if any) is a sequence. Also known as multi-state problem: agent knows which states it might be in

- Nondeterministic and/or Partially Observable → contingency problem

  Percepts provide new information about current state. Solution is a contingent plan or a policy. Often interleave search, execution. Plans contain conditional parts based on sensors

Example: Vacuum World



Single-state, start in #5.

Solution? [Right, Suck]

Conformant, start in {1,2,3,4,5,6,7,8}.

Solution? [Right, Suck, Left, Suck]

Formulation of a Problem

1. Initial state(s): the state(s) the agent starts in

2. Action/operators: given any state *s*, ACTION(s) returns set of actions that can be executed from *s*

3. Transition model: maps state-action pairs to states, given a state *s* and action *a*. RESULT(*s*, *a*) returns the state that resylts from carrying out action *a* on *s*

    1-3 implicitly define state space, which can be encoded as a directed graph (nodes are states and edges are actions). What is a path in this graph?

4. Goal test determines whether a given state is a goal state (defined explicitly or via a property).

5. Path cost: computational cost of the execution of the path/plan

Single-state Problem for Route-Finding

1. Initial state(s): e.g., “In(Arad)”

2. Action/operators: e.g.

   ACTION(Arad) = {Arad → Timisoara, Arad → Sibiu, …, Arad → Zerind}

3. Transition model:

   RESULT(Arad, Arad → Zerind) = Zerind

4. Goal test:

   Explicit e.g. ”In(Bucharest)”

5. Path cost (additive)

   e.g. sum of distances, number of actions executed, etc.

   c(x, a, y) is the step cost (assumed to be non-negative)

Single-state Problem for Route-Finding

1. Initial state(s): e.g., “In(Arad)”

Solution:

A **solution** is a sequence of actions leading from the initial state to a goal state.

The process of looking for a solution is called **search**.

5. Path cost (additive)

    e.g. sum of distances, number of actions executed, etc.

    c(x, a, y) is the step cost (assumed to be non-negative)

Abstraction: Defining a State Space

Real world is absurdly complex

State space must be **abstracted** for problem solving

(Abstract state) = set of real states

(Abstact) action = complex combination of real actions.

e.g. Arad → Zerind represents a complex set of possible routes, detours, rest stops, etc.

(Abstract solution) = set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem.

State Space Graph

State space graph: A mathematical representation of a search problem

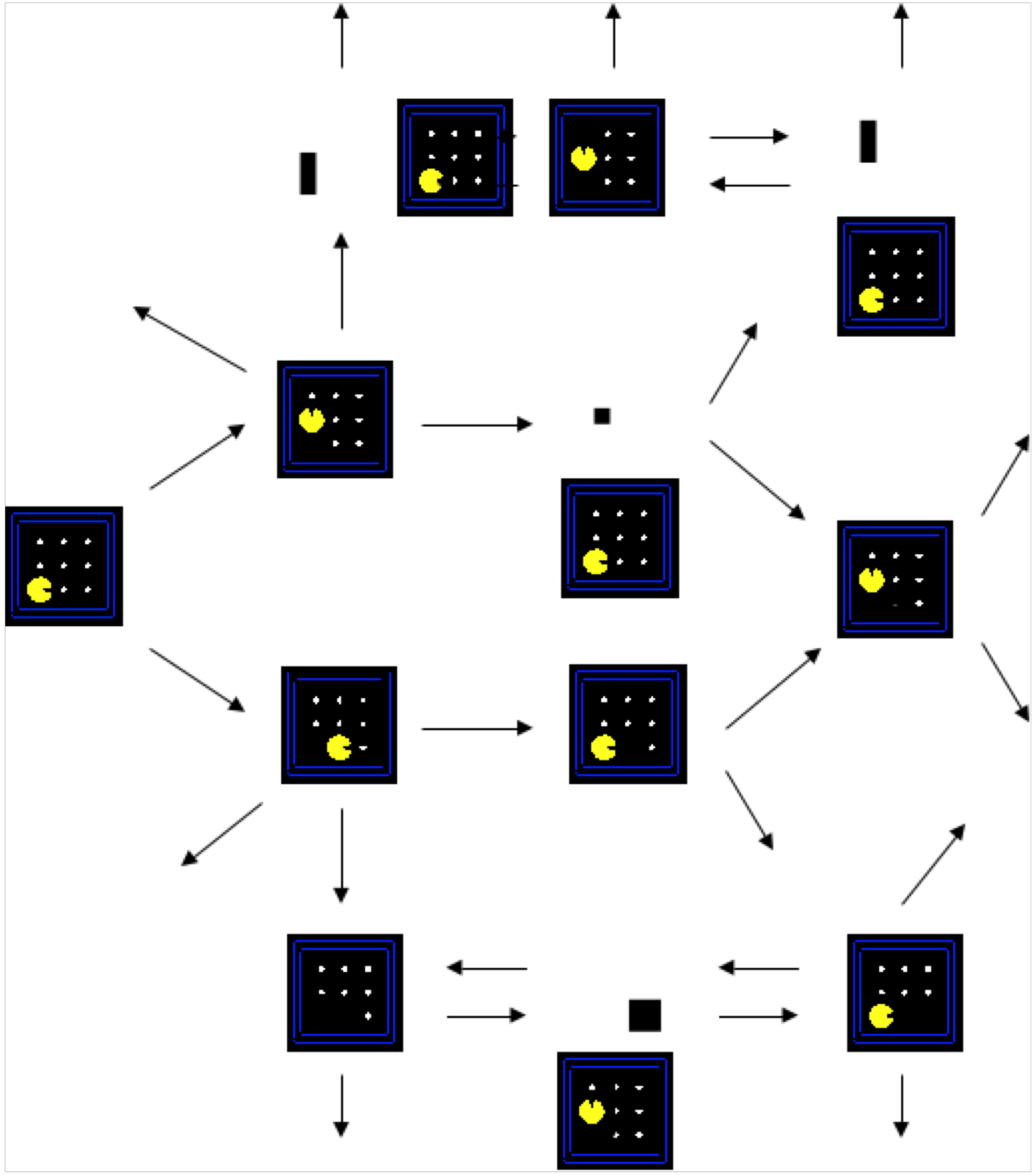
Nodes are (abstracted) world configuration

Arcs/edges represent successors (action results)

Goal test is a set of goal nodes (maybe one)

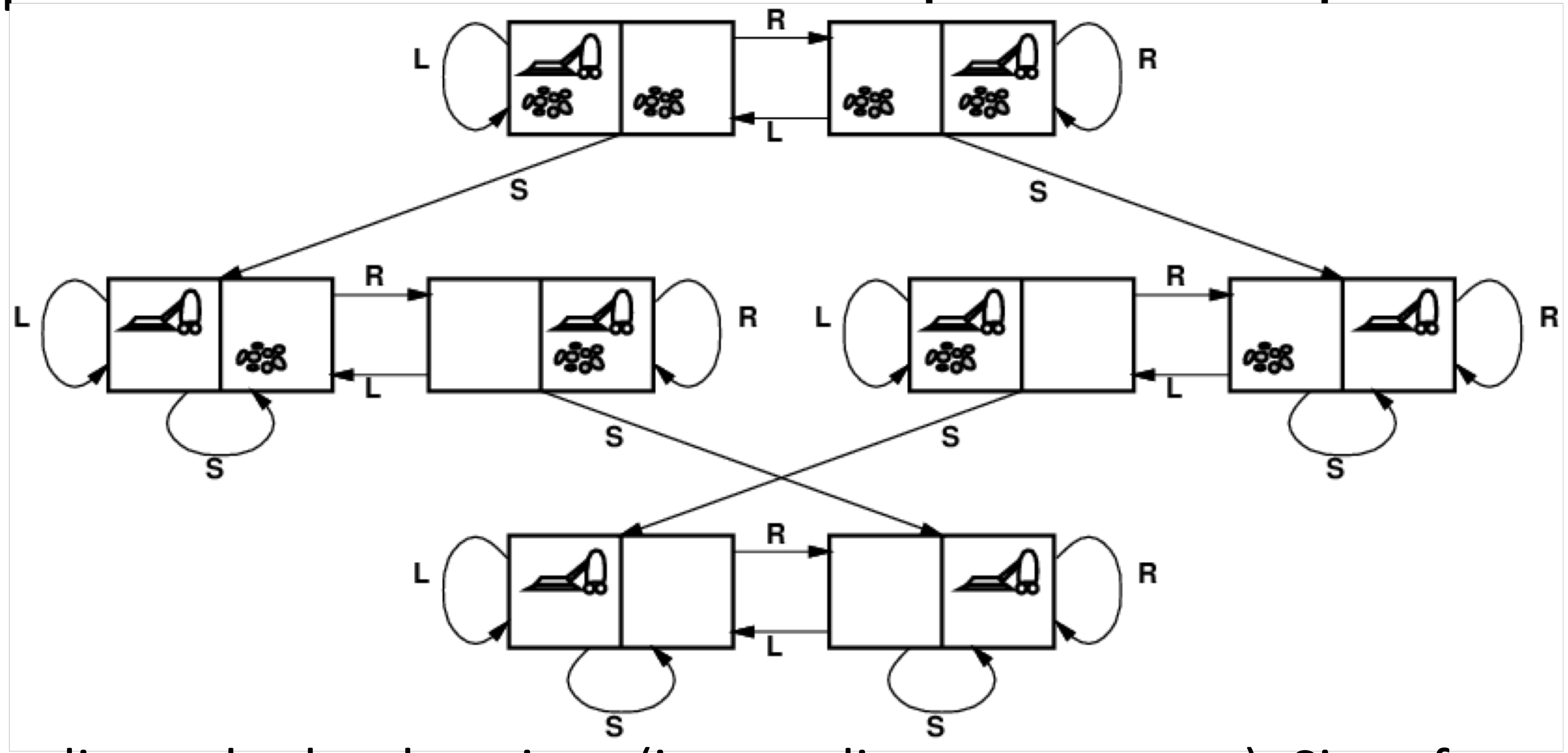
In a state space graph, each state occurs only once.

We can **rarely** build this full graph in memory, but it's a useful idea.



JMU JAMES MADISON UNIVERSITY

Example: Vacuum World Space Graph



States: Integer dirt and robot locations (ignore dirt amounts, etc). Size of state space?

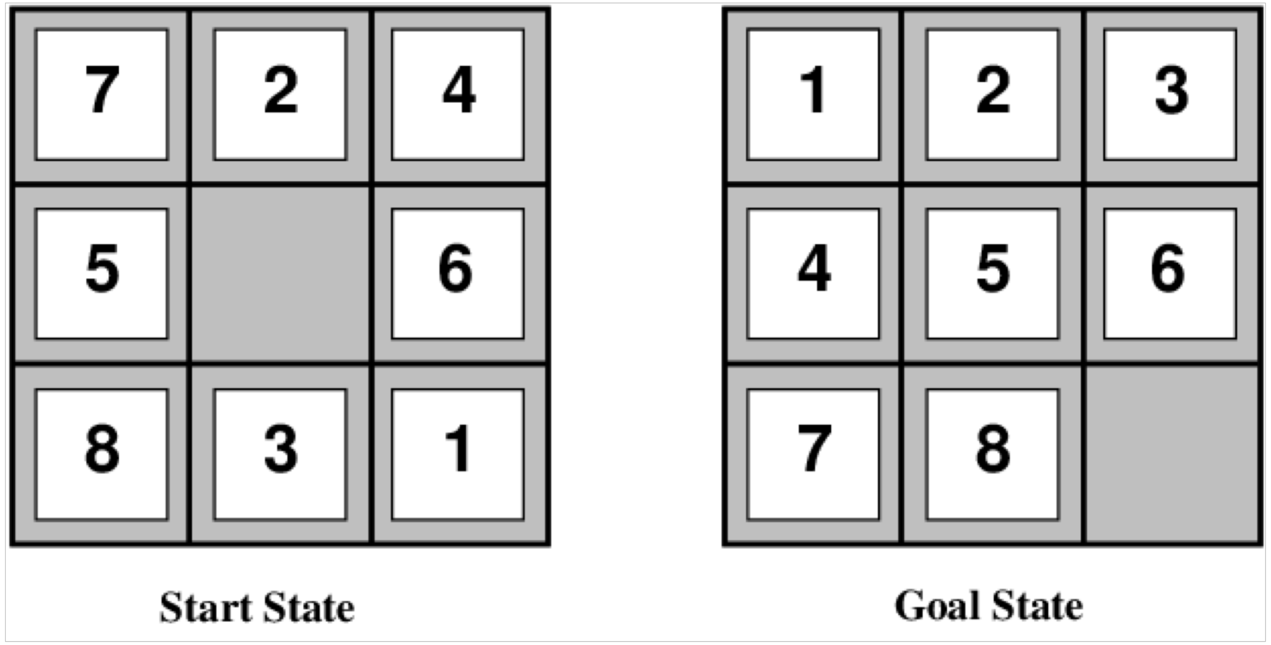
Actions: Left, right, suck, NoOp

Transition model: ([A, dirt], Suck) → [A, clean]

Goal test: No dirt

Path cost: 1 per action (0 for NoOp)

Example: The 8-puzzle



States: Integer location of the tiles. State space size?

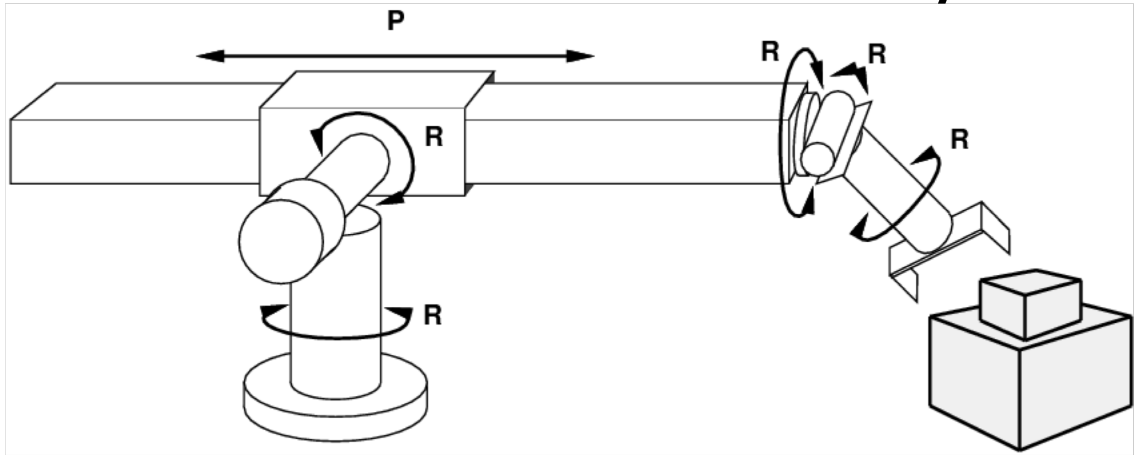
Actions: Blank space “moves” (Left, right, up, Down)

Transition model: Given state and action returns resulting state.

Goal test: Do we match the explicit goal state

Path cost: 1 per move (optimal solution is NP-hard !]

Example: Robotic Assembly



States: Real-valued coordinates of the robot joint angles + parts of the object to be assembled.

Actions: Continuous motions of robot joints

Transition model: state + action yields new state

Goal test: Complete assembly

Path cost: Time to execute

Route-finding and Tour-finding Problems

The vacuum cleaner problem, 8-puzzle (block sliding), 8-queens, and others are examples of toy, route-finding problems.

Real-world route-finding problems can be found in robot navigation, manipulation, assembly, airline travel web-planning, and more.

Tour-finding problems are slightly different: “visit every city at least once, starting and ending in Bucharest.”

Traveling salesperson problem (TSP): find shortest tour that visits each city exactly once, NP-hard.

Other related, complex problems: packing, scheduling, VLSI layout, protein folding, protein design.

Searching for Solutions

Choosing states and actions:

- Abstraction: remove unnecessary information from representation, makes it cheaper to find a solution

Searching for Solutions:

- Operators expand a state: generate new states from present ones
- Fringe or frontier: discovered states to be expanded
- Search strategy: tells which state in fringe set to expand next

Measuring Performance:

- Does it find a solution?
- What is the search cost?
- What is the total cost (path cost + search cost)?

Search trees

A “what if” tree of plans and their outcomes

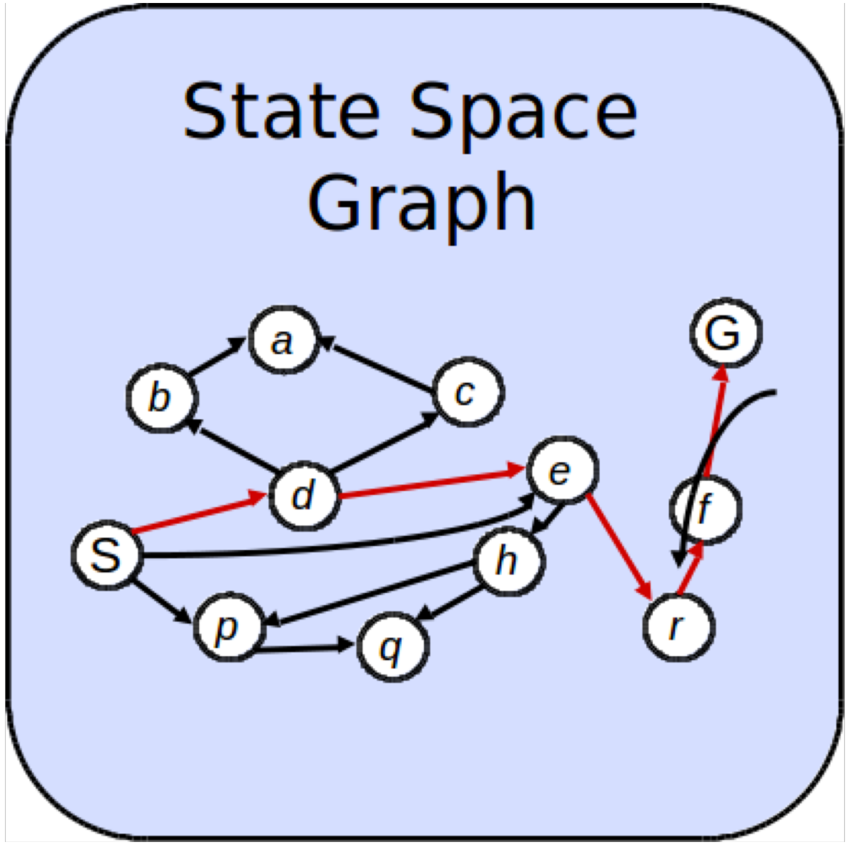
The start state is the root node

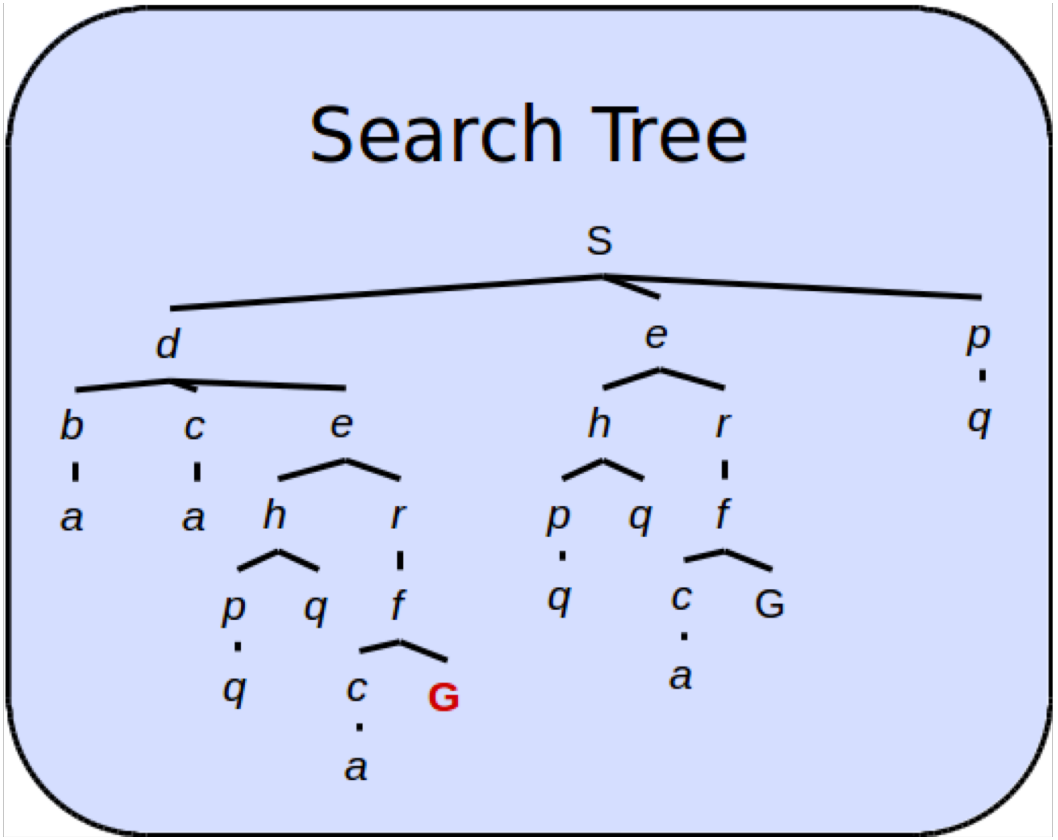
Children correspond to successors

Nodes show states, but correspond to PLANS that achieve those states

For most problems, we can never actually build the whole tree

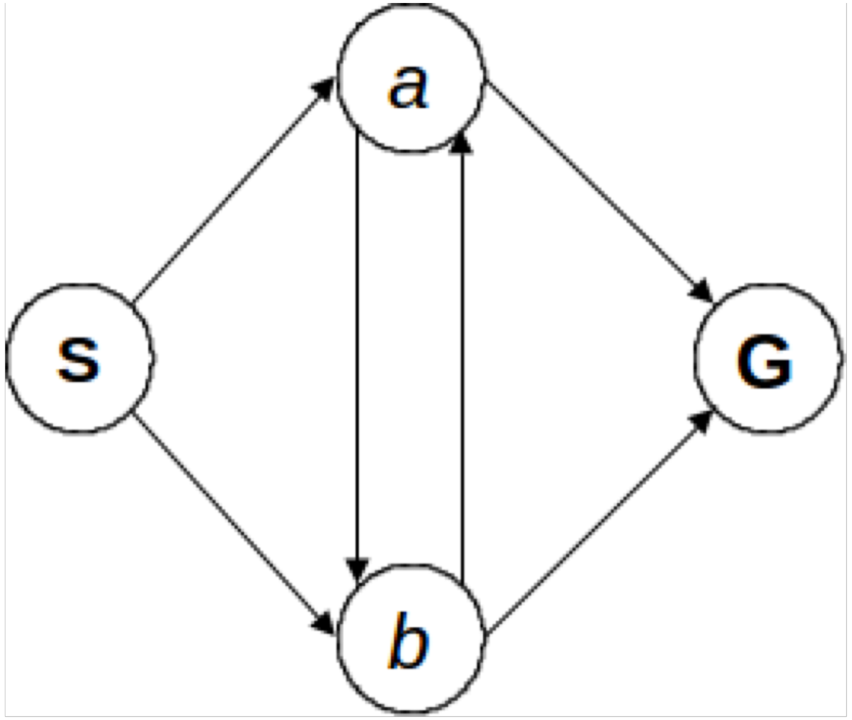
State Space Graphs vs Search Trees





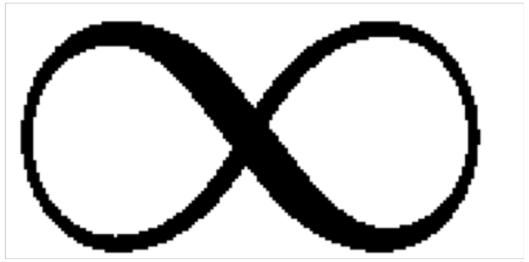
We construct both on demand and we construct as little as possible.

State Space Graphs vs Search Trees

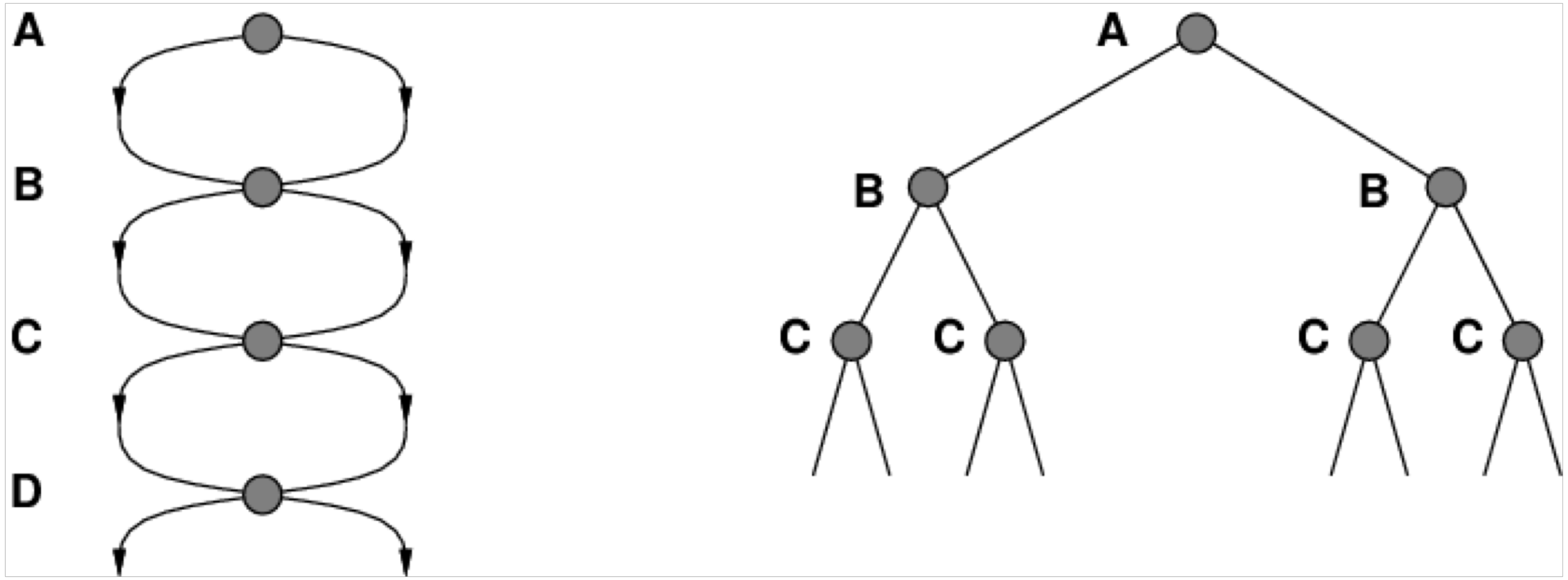


Consider the 4-state space graph on the left. How big is it’s search tree?

Lots of repeated structures !!



Repeated States

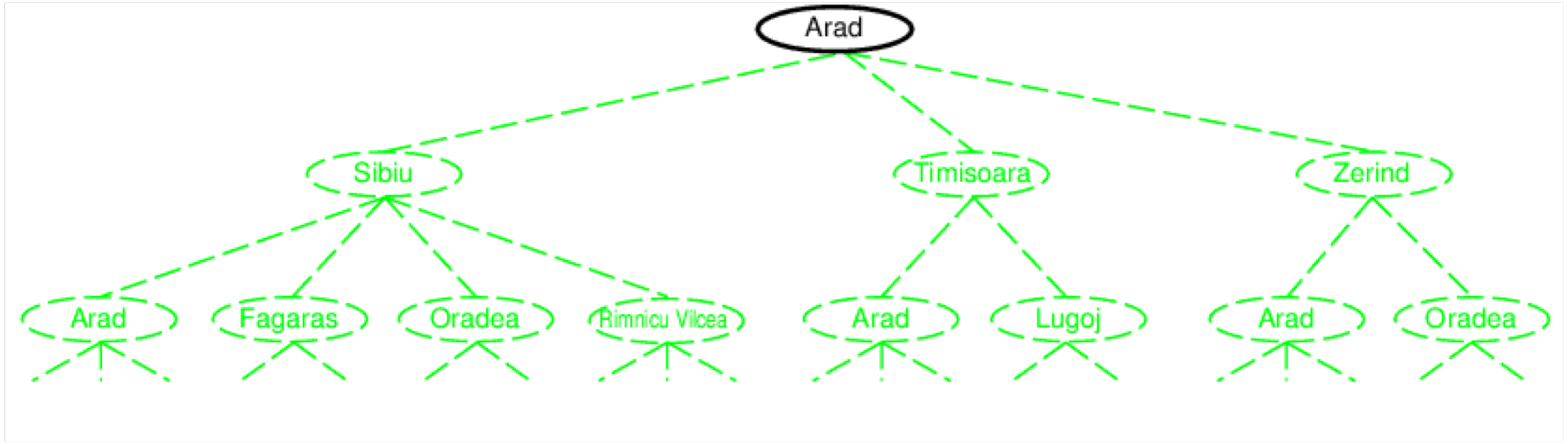


Can repeated structure be easily avoided. If so, how?

Avoiding Repeated States

```
function Graph-Search(problem, fringe) returns a solution or failure
    closed ← an empty set
    fringe ← INSERT(Make-Node(Intitial-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)
```

Searching with a Search Tree



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

(Discrete) Search Algorithms

Basic idea:

- Offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

function Tree-Search(problem, strategy) **returns** a solution or failure
    Initialize the search tree using the initial state of the 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

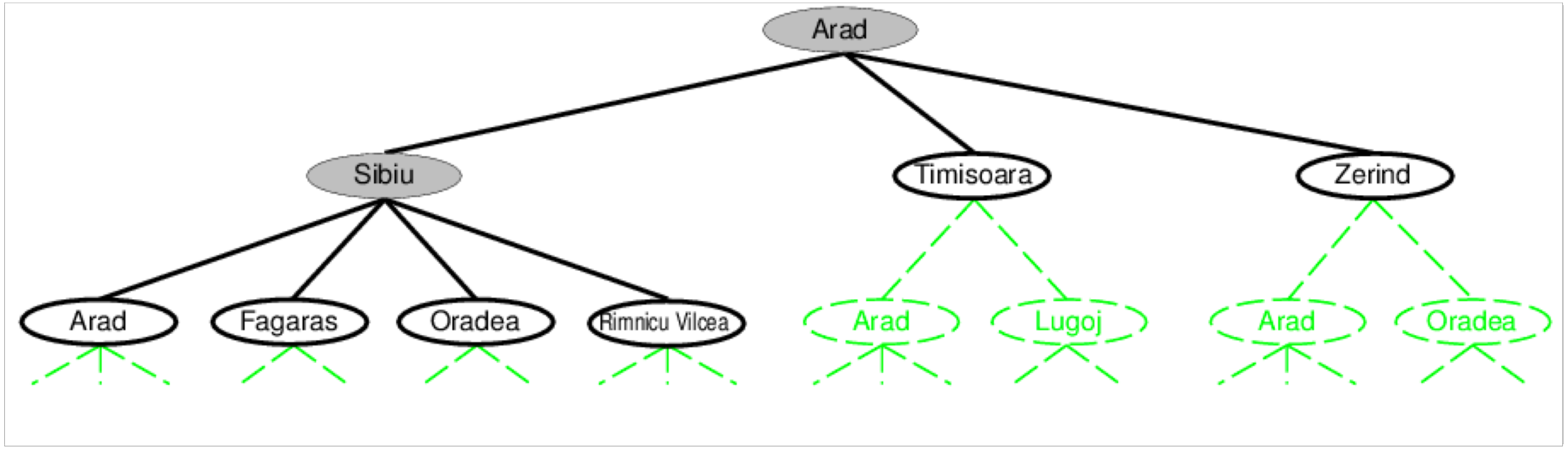
Fundamental Properties of Discrete Search Algorithms

Fundamental to Graph Search/Traversal Algorithms:

- Successor function: generate successors/neighbors and distinguish a goal state from a non-goal state.

Completeness Goal should not be missed if a path exists.

Efficiency No edge should be traversed more than twice.



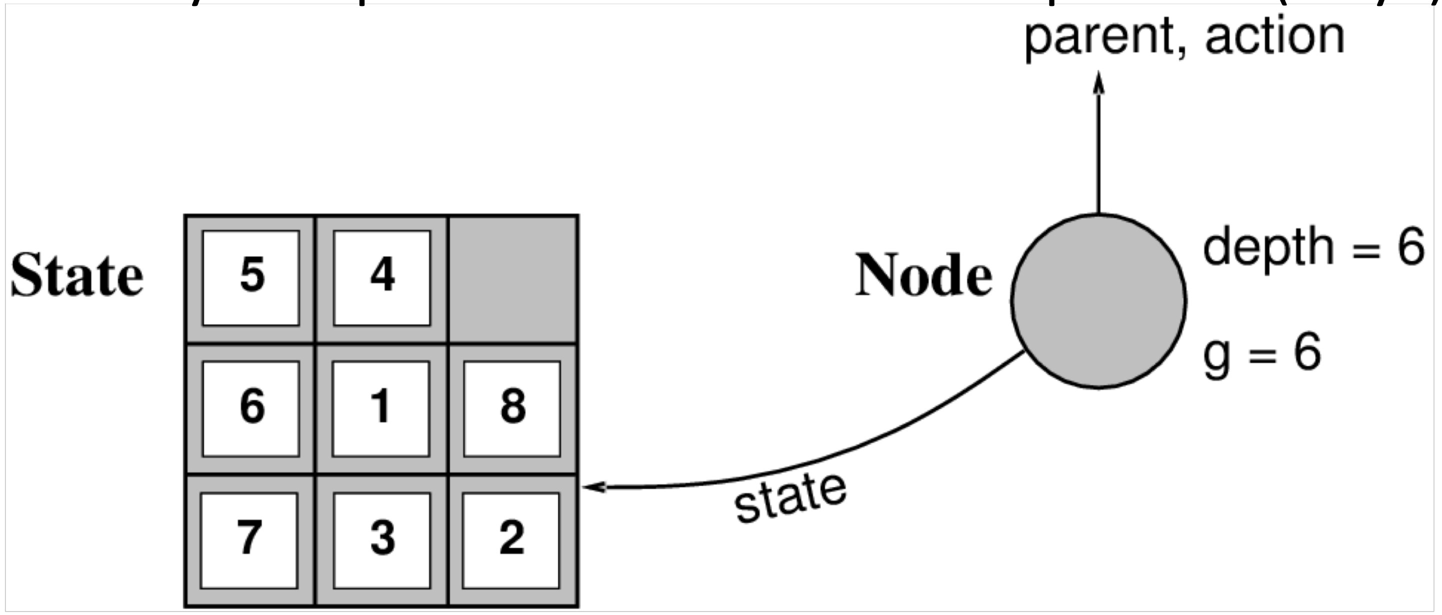
Implementation: States vs Nodes

A state is a (representation of) a physical configuration

A node is a data structure constituting part of a search tree and include parent, children, depth, path cost *g(x)*

States do not have parents, children, depth, or path costs!

- Try to expand as few tree nodes as possible (why?)



The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

General Tree Search

Important insight:

- Any search algorithm constructs a tree, adding to it vertices from state-space graph G in some order
- G = (V, E) – look at it as split in two: set S on one side and V –S on the path
- Search proceeds as vertices are taken from V-S and added to S
- Search ends when V – S is empty or goal found
- First vertex to be taken from V-S and added to S?
- Next vertex (expansion…)
- Where to keep track of these vertices (fringe/frontier)

General Tree Search

Important ideas:

- Fringe (frontier into V– S/border between S and V – S)
- Expansion (neighbor generate, enables adding to fringe)
- Exploration strategy (what order to grow S?)

Main question:

- Which fringe/frontier nodes to explore/expand next?
- Strategy distinguishes search algorithms from one anothe

Search strategies

A 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/expanded
- Space complexity – maximum number of nodes in memory
- Optimality – does it always find a least-cost solution?

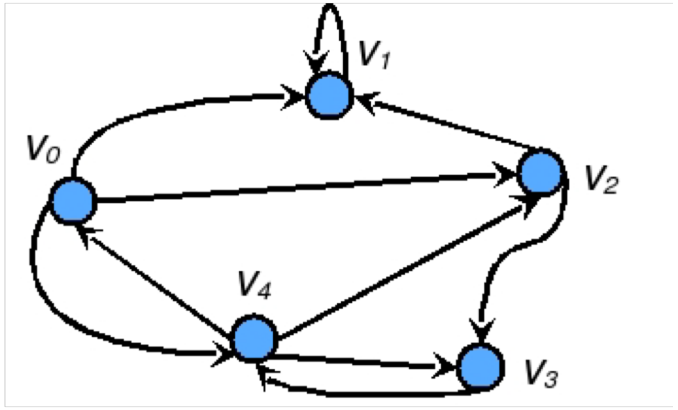
Time and space complexity are measured in terms of:

- b – maximum branching factor of the search tree
- d – depth of the least-cost solution
- m – maximum depth of the state space (may be ∞)

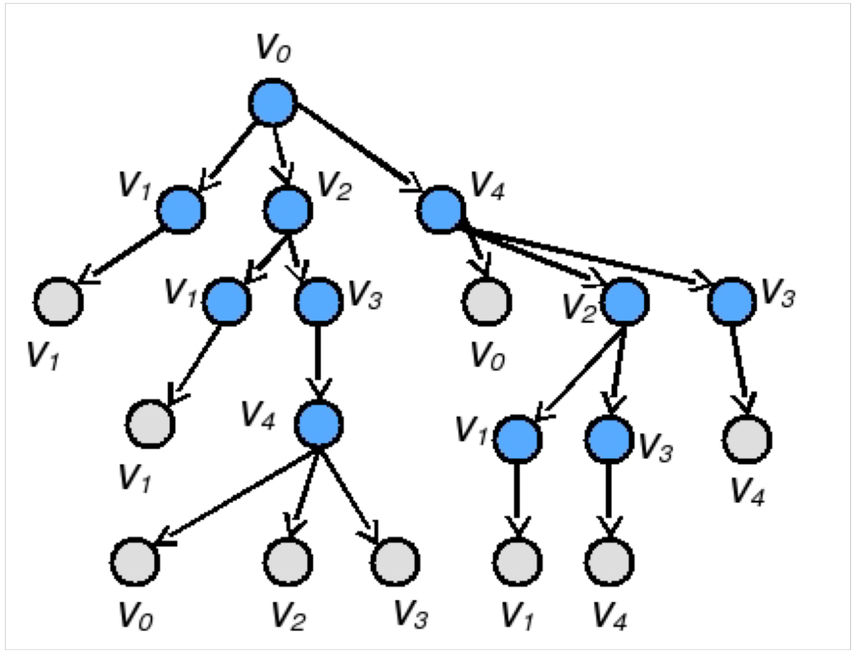
Uninformed Graph Search

Characteristics of Uninformed Graph Search/Traversal:

- There is no additional information about states/vertices beyond what is provided in the problem definition.
- All that the search does is generate successors/neighbors and distinguish a goal state from a non-goal state.



The systematic search "lays out" all paths from the initial vertex, it traverses the search tree of the graph.

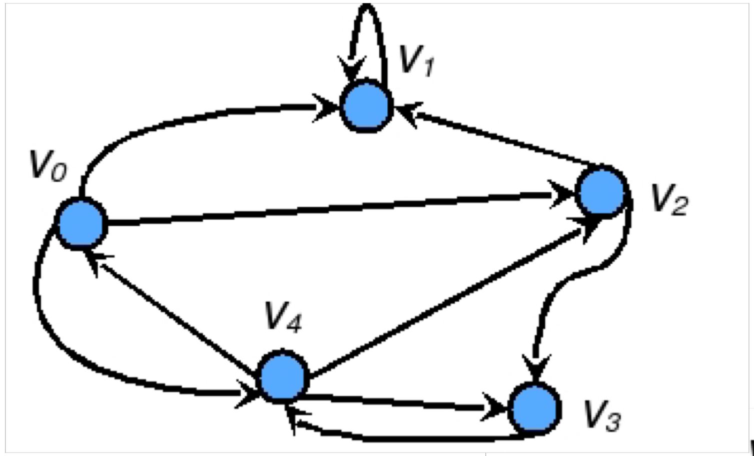


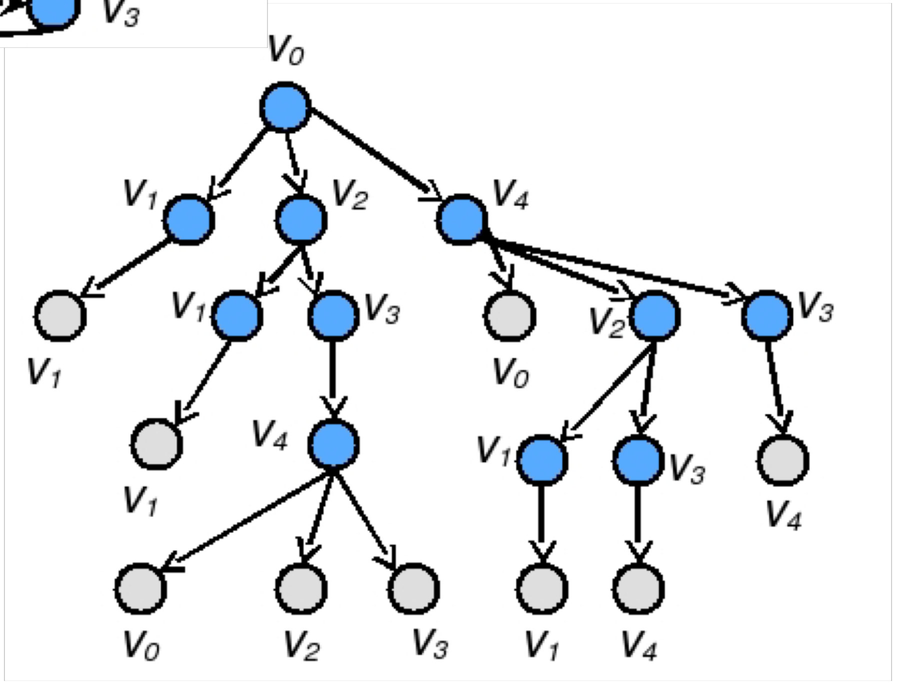
Uninformed Graph Search

F: search data structure (fringe)

Parent array: stores “edge come from” to record visited states

```
F.insert(v)
Parent[v] ← true
While not F.isEmpty() do
  u ← F.extract()
  if isGoal(u) then
     return true
  for each v in outEdges(u) do
    if no parent[v] then
      F.insert(v)
      Parent[v] ← u
```

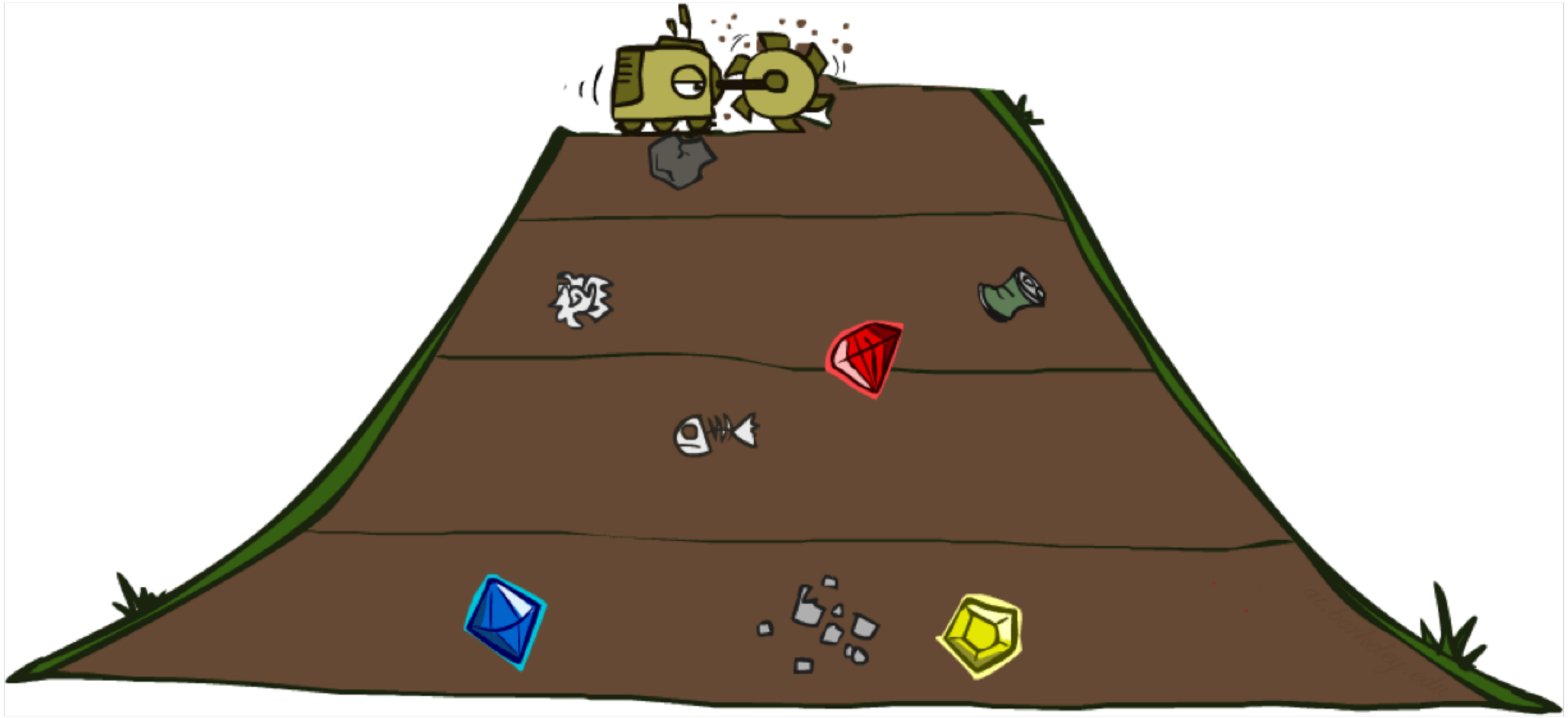




Uninformed Graph Search

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Depth-limited search (DLS)
- Iterative Deepening Search (IDS)

Breadth-first Search (BFS)

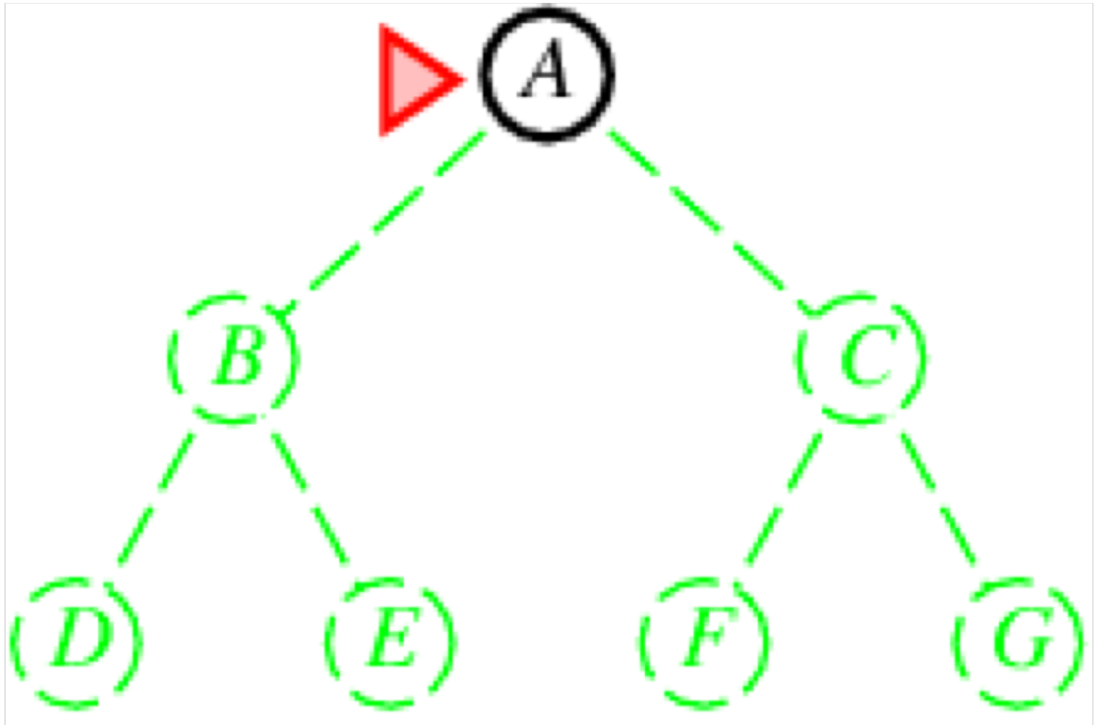


Breadth-first Search (BFS)

Strategy: Expand shallowest unexpanded node

Implementation:

Fringe = first-in first-out (FIFO), i.e., unvisited successors go at end (F is a queue)

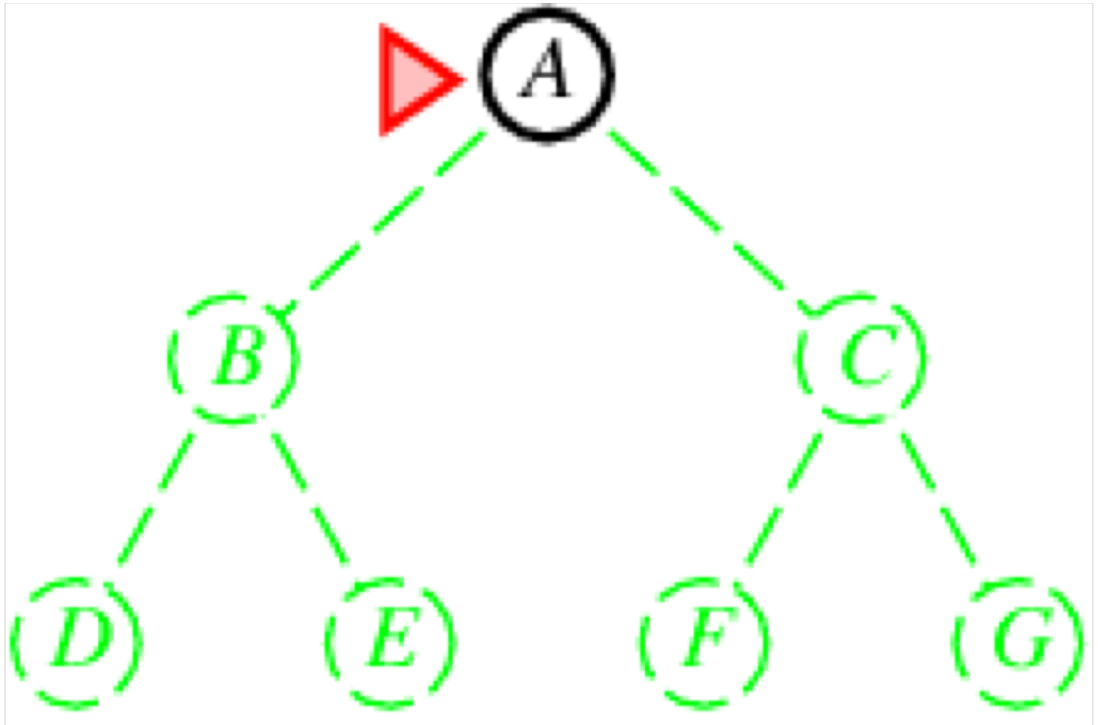


Breadth-first Search (BFS)

Strategy: Expand shallowest unexpanded node

Implementation:

Fringe = first-in first-out (FIFO), i.e., unvisited successors go at end (F is a queue)

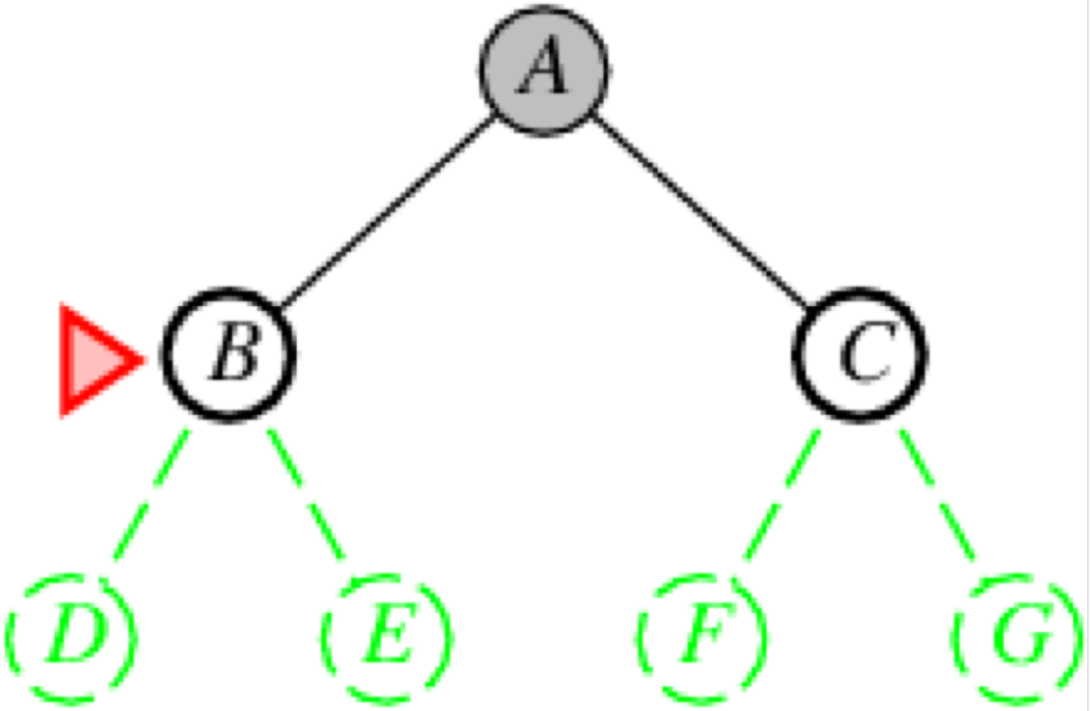


Breadth-first Search (BFS)

Strategy: Expand shallowest unexpanded node

Implementation:

Fringe = first-in first-out (FIFO), i.e., unvisited successors go at end (F is a queue)

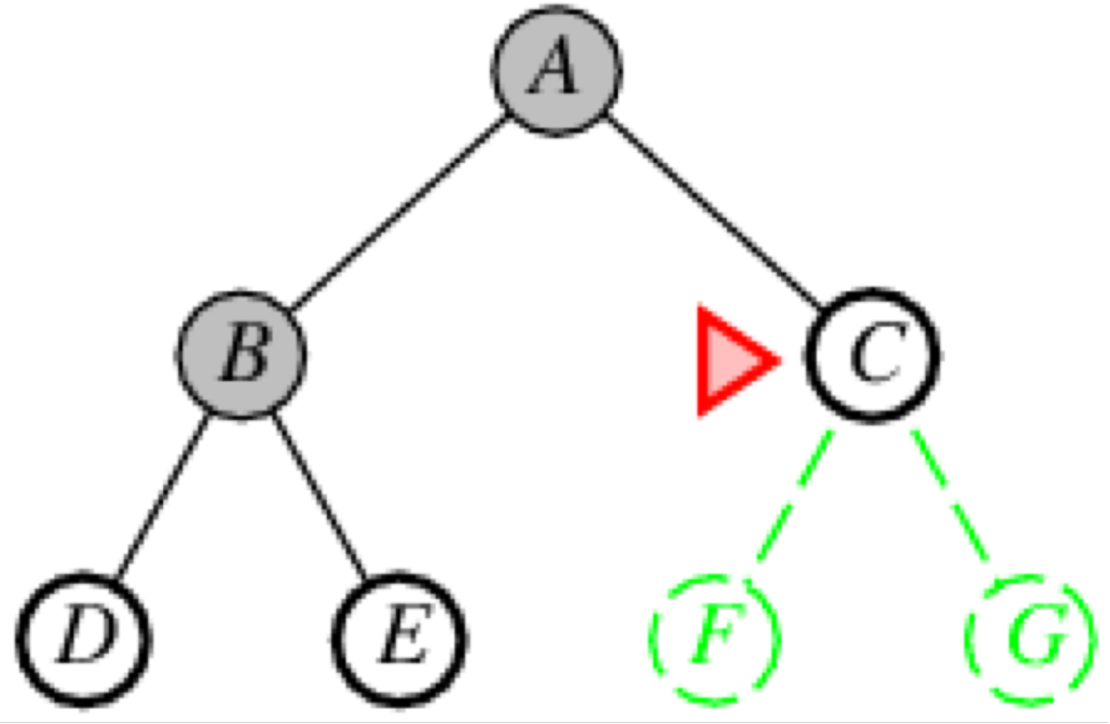


Breadth-first Search (BFS)

Strategy: Expand shallowest unexpanded node

Implementation:

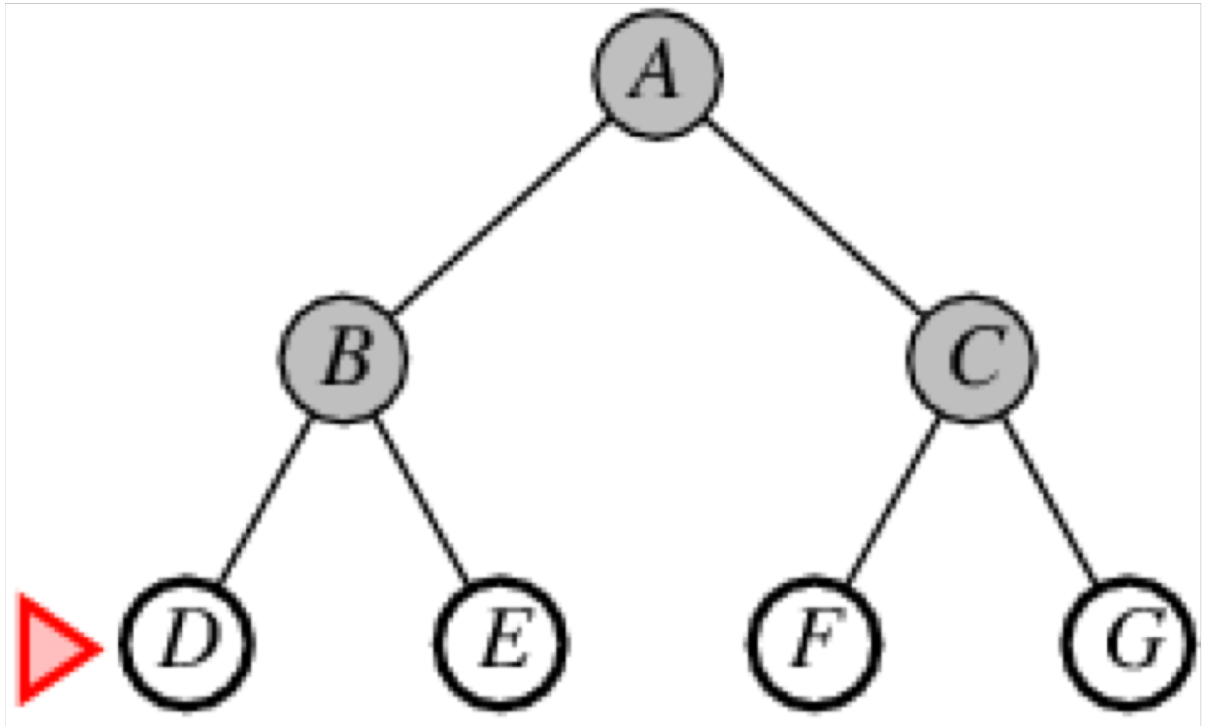
Fringe = first-in first-out (FIFO), i.e., unvisited successors go at end (F is a queue)



Breadth-first Search (BFS)

Strategy: Expand shallowest unexpanded node

Implementation: Fringe = first-in first-out (FIFO) (F is a queue)




```
F.insert(v)
Parent[v] ← true
While not F.isEmpty() do
   u ← F.extract()
   if isGoal(u) then
      return true
   for each v in outEdges(u) do
     if no parent[v] then
        F.insert(v)
        Parent[v] ← u
```

Running time?

Properties of Breadth-first Search (BFS)

Complete? Yes (if *b* is finite)

Time? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^{d-1}) = O(b^{d+1})$, ie, exp. In *b*

Space? $O(b^{d+1})$ (keeps every node in memory)

Optimal? Yes, (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100 MB/sec, so, 24 hrs = 8.6 TB).

Basic Behavior:

- Expands all nodes at depth *d* before those at depth *d+1*.
- The sequence is root, then children, then grandchildren in the search tree.

Properties of Breadth-first Search (BFS)

Problems:

- If the path cost is a non-decreasing function of the depth of the goal node, BFS is optimal (uniform cost search a generalization).
- A graph with no weights can be considered to have edges of weight 1, in this case, BFS is optimal.
- BFS will find the shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.
- BFS is not very popular because time and space complexity are exponential (with respect to d).
- Memory requirements of BFS are a bigger problem.