Introduction to Computational Intelligence

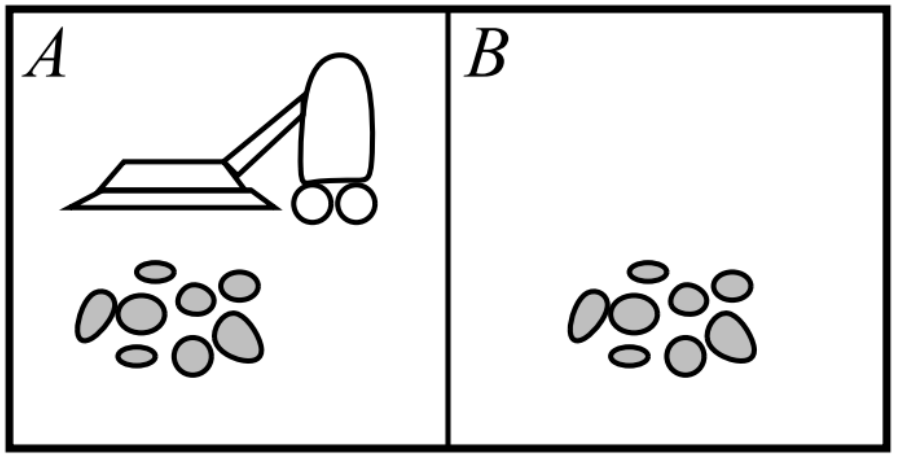
Lecture 3

Outline

- Intelligent Agent
- Formulating a problem and solution
  - ✓ State space
  - ✓ Path
  - ✓ Practical examples
- Pseudocode
- State space vs search tree
- Pop quiz

Formulating Problems and Solutions – Example # 1

- Vacuum world



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

Actions: Left, Right, Suck, NoOp

Goal test: no dirt left in any location (square)

Path cost: each action costs something, e.g., time and energy

➢ Pseudocode

function REFLEX-VACUUM-AGENT([*location*,*status*]) **returns** an action

if *status* = *Dirty* **then return** *Suck*
else if *location* = *A* **then return** *Right*
else if *location* = *B* **then return** *Left*

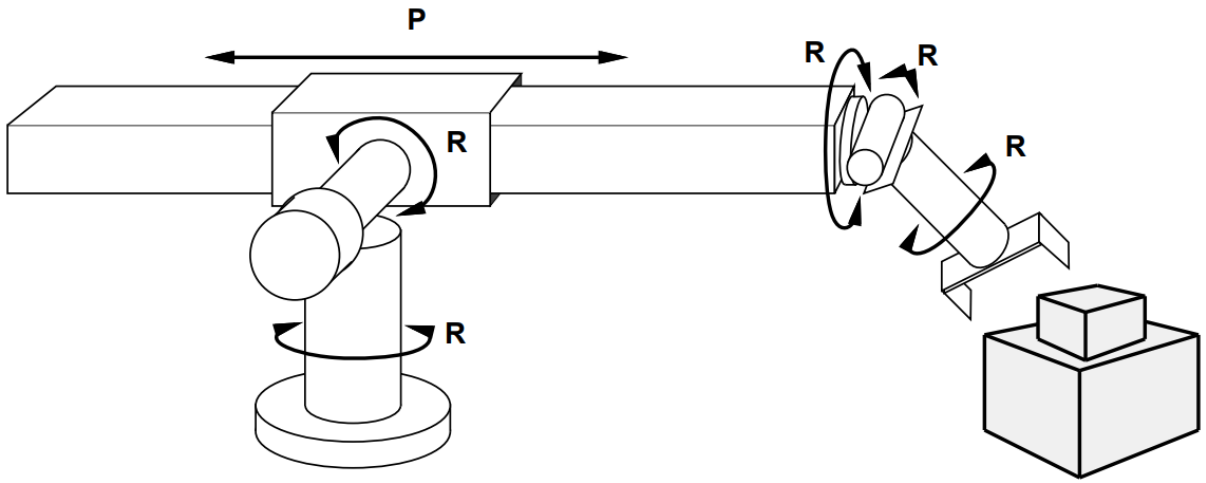
Formulating Problems and Solutions – Example # 2

- Taxi driver agent

Agent Type	Percepts	Actions	Goals	Environment
Taxi driver	Cameras, speedometer, GPS, sonar, microphone	Steer, accelerate, brake, talk to passenger	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers

Formulating Problems and Solutions – Example # 3

- Robotic assembly



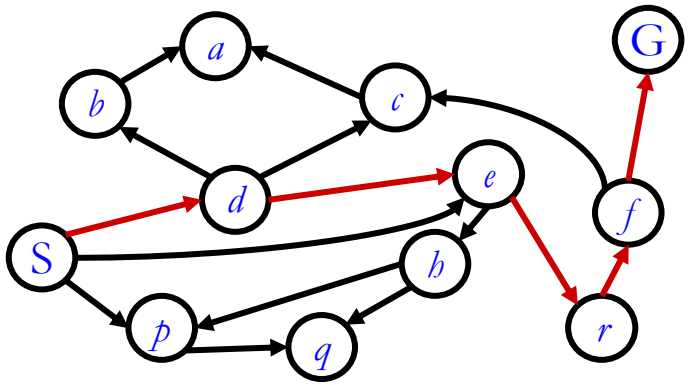
- **States**: Real-valued coordinates of robot joint angles, parts of the object to be assembled.
- **Actions**: Continuous motions of robot joints.
- **Goal test**: Complete assembly.
- **Path cost**: Time to execute.

Formulating Problems and Solutions - A simple problem-solving agent [2]

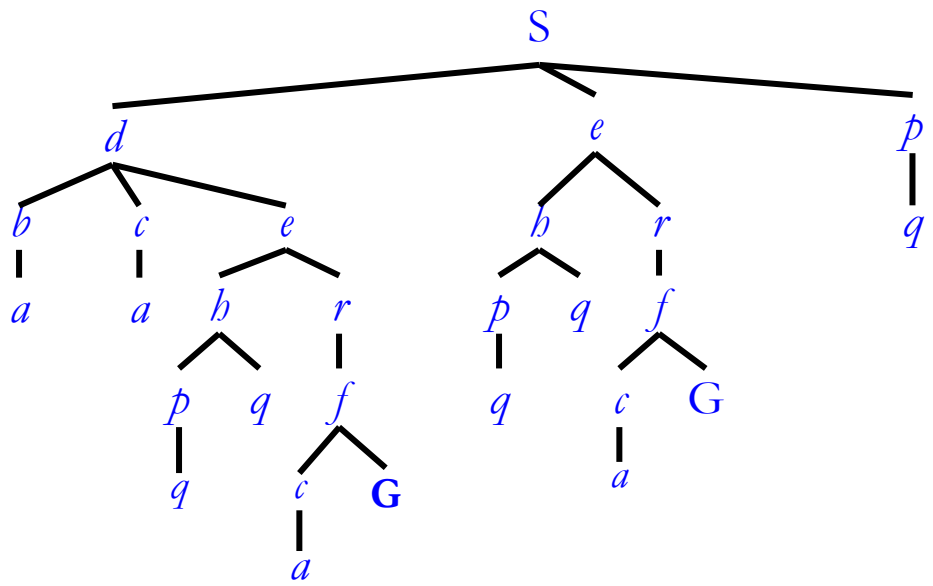
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq: an action sequence, initially empty
              state: some description of the current world state
              goal: a goal, initially null
              problem: a problem formulation
  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a NoOp
  action ← FIRST(seq)
  seq ← REST(seq): removing action step from the sequence
  return action
```

State Space Graphs vs. Search Trees

State Space Graph



Search Tree

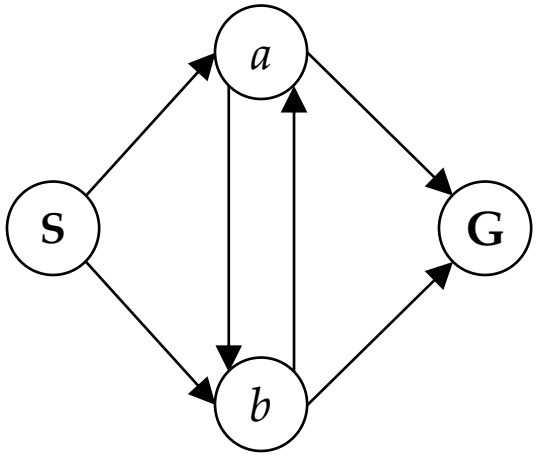


- In state space graph, each state appears only once, but in the search-tree states many reappear in different branches.
- Search tree is mainly built up based on the state space graph when running a search algorithm.
- Each node in the search tree is an **entire path** in the state space graph.
  - E.g., node *p* in the right most branch does not just encode the state *p*, but it encodes the entire path planning from *S*: S→P. Similarly, the node *p* in the middle branch will reflect the path, *S*→*e*→*b*→*p*.
- We construct both on demand – and we construct as little as possible.

Adopted from Anca Dragan, University of California, Berkeley

State Space Graphs vs. Search Trees

Consider this 4-state graph:

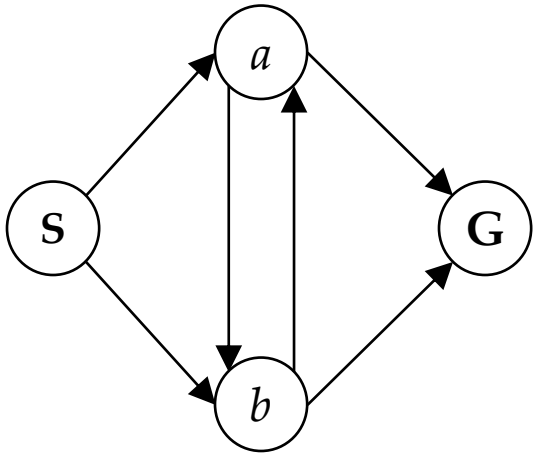


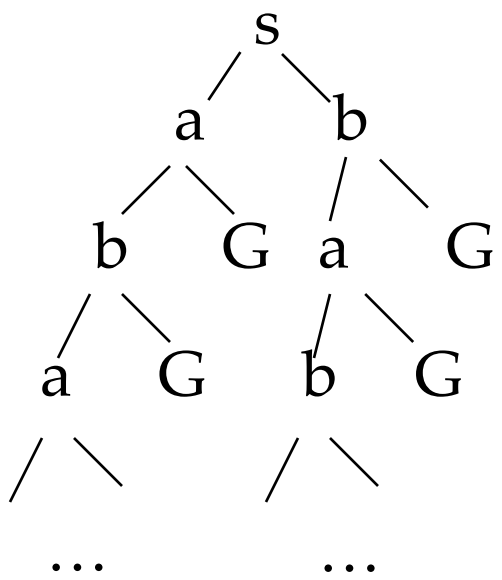
How big is its search tree (from S)?

State Space Graphs vs. Search Trees

Consider this 4-state graph:

How big is its search tree (from S)?

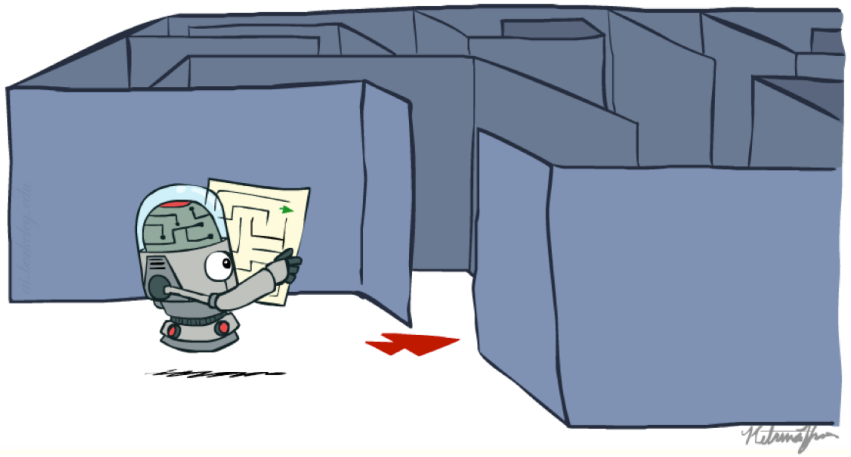




Adopted from Anca Dragan, University of California, Berkeley

Search Strategies in Artificial Intelligence

Lecture 3

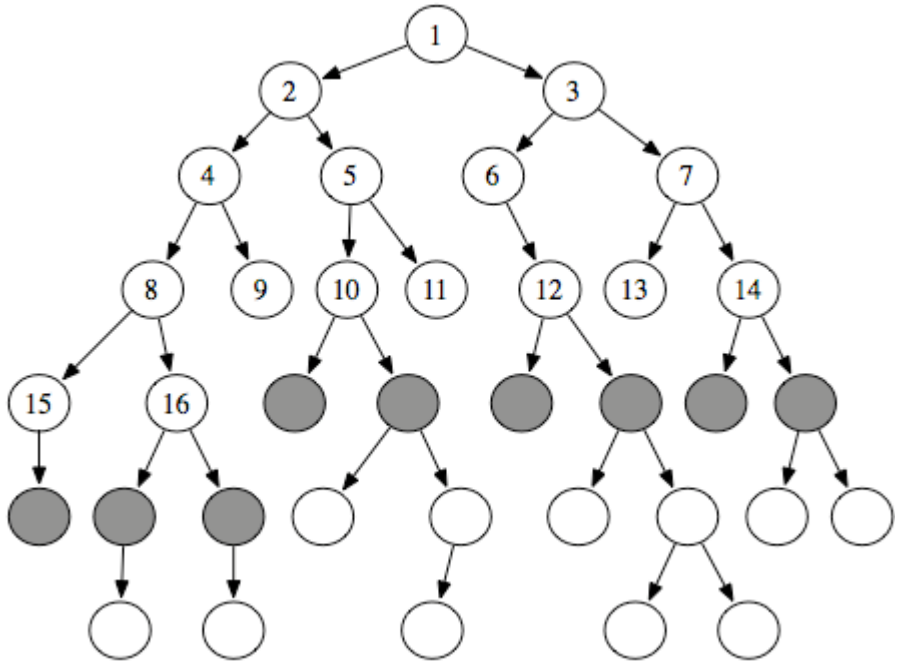


This session

- Searching for solutions
- Properties of search strategies
- Uninformed search strategies
  - BFS
  - Uniform cost search
  - Depth first search
  - Iterative deepening search
- Informed search strategies
  - Greedy best-first search
  - A* search

Searching for Solutions

- Recall that a solution is an action sequence.
- **Search algorithms** work by considering various possible action sequences.
- The possible action sequences starting at the initial state form a search tree with the initial state at the root;
- **Branches** are **actions**, and **nodes** correspond to **state** in the state space of the problem.
- To consider taking various actions, we expand the current state – thereby generating a new set of states.
- In this way, we add branches from the parent node to children nodes.

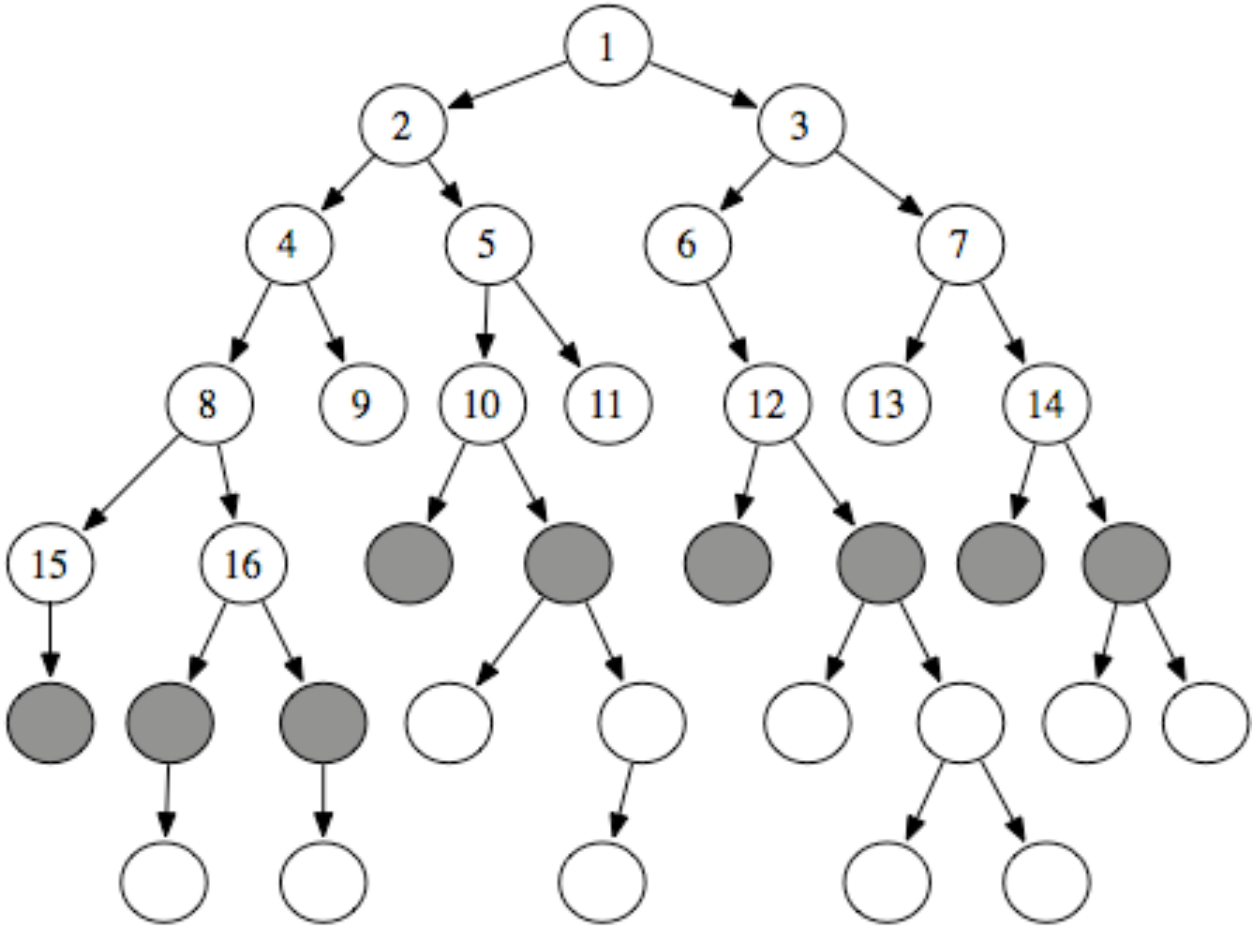


Slide adopted from A.I.: Solving problems by searching by Anthony D. Rhodes

Searching for Solutions Cont.

- A node with no children is called a **leaf**;

- **Frontier or fringe:** The set of all nodes available for expansion at any given point is called the.
  - The frontier (in this case the gray nodes) always separates the explored region of the state space (the numbered nodes) from the unexplored region (white nodes)

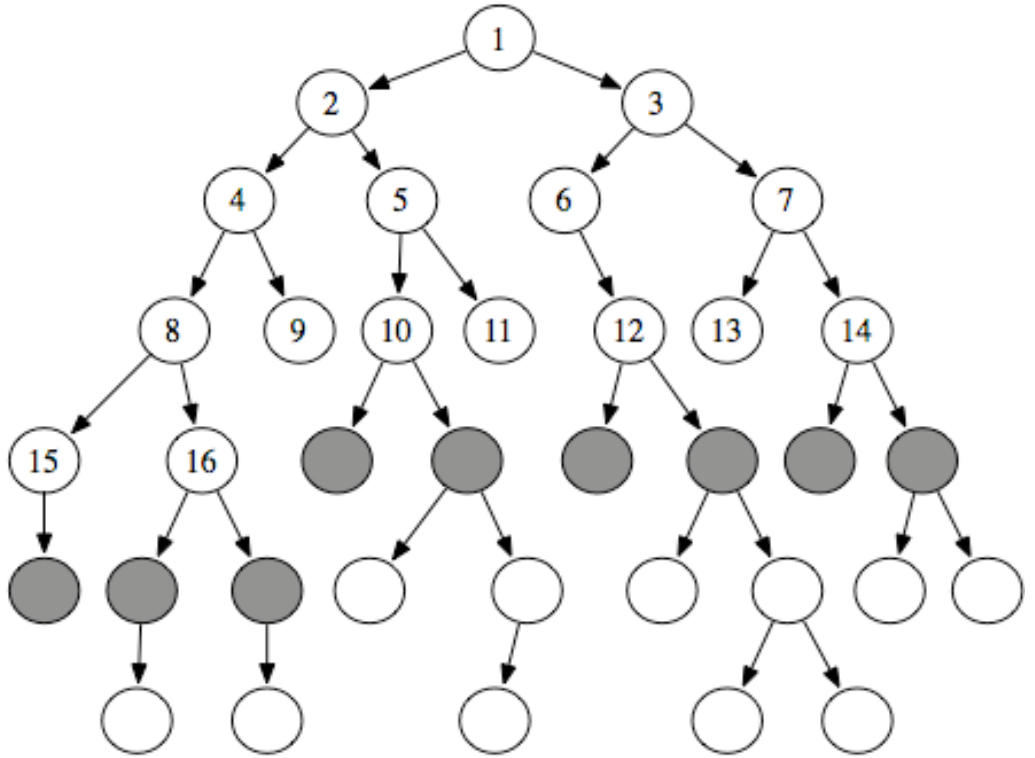


Slide adopted from A.I.: Solving problems by searching by Anthony D. Rhodes

Searching for Solutions Cont.

- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

- All search algorithms share this basic structure; they vary primarily according to how they choose which state to expand next – the so-called **search strategy**.



- In general, a TREE-SEARCH considers all possible paths (including infinite ones), whereas a GRAPH-SEARCH avoids consideration of redundant paths.

Slide adopted from A.I.: Solving problems by searching by Anthony D. Rhodes

General Searching for Solutions

```
function GENERAL-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 – no more nodes available

        choose a node from the available node list for expansion according to strategy

        if the node contains a goal state
            then return the corresponding solution - path from initial state to the node

        else expand the node and add the resulting nodes to the search tree – generate all successors of
                                                                              the node and merge them into fringe

    end
```

The fringe will have many nodes, the strategy will decide which node to be expanded first

Tree Search vs Graph 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 node from the frontier and remove it from the frontier queue

        if the node contains a goal state
            then return the corresponding solution - path from initial state to the node

        expand the chosen node, adding the resulting nodes to the frontier queue
    end
```

Tree Search vs Graph Search Cont.

- **Issue:** algorithms that forget their history **are doomed to repeat it**.
- **Solution:** **to avoid exploring redundant** paths is to **remember where one has been**.
- **Idea:**
  - Augment the TREE-SEARCH algorithm with a data structure called the **explored set** (or the **closed list**), which **remembers every expanded** node.
  - Discard the 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.

Tree Search vs Graph Search

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial stale of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty
            then return failure
        choose a node from the frontier 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
    end
```

Note: Red ink shows the lines that are added to the tree search algorithm.

Search Strategies

- A **search strategy** is defined by picking the order of node expansion
- Evaluation criteria:
  - **Completeness -** does it always find a solution if one exists?
  - **Optimality -** does it always find a least-cost solution?
  - **Time complexity -** number of nodes generated (worse or average case) to find a solution
  - **Space complexity -** maximum number of nodes in memory

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the optimal solution
  - *m*: maximum length of any path in the state space (may be ∞) or
    in other words, the maximum depth of the search tree.

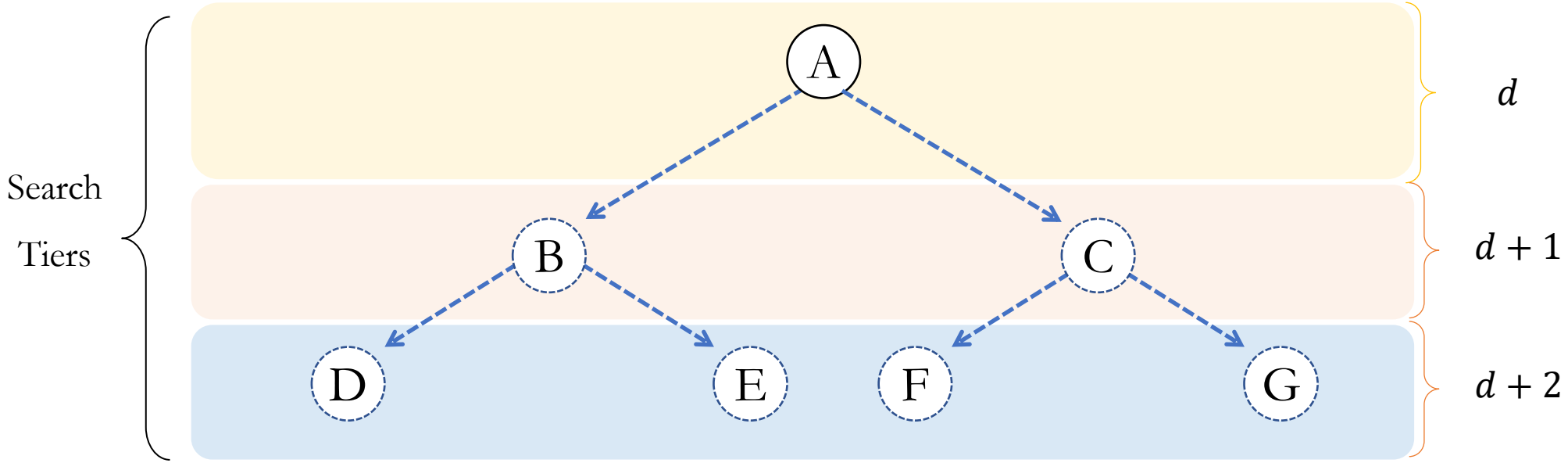
Uninformed Graph Search Strategies

- **Uninformed** search strategies use only the information available in the problem definition

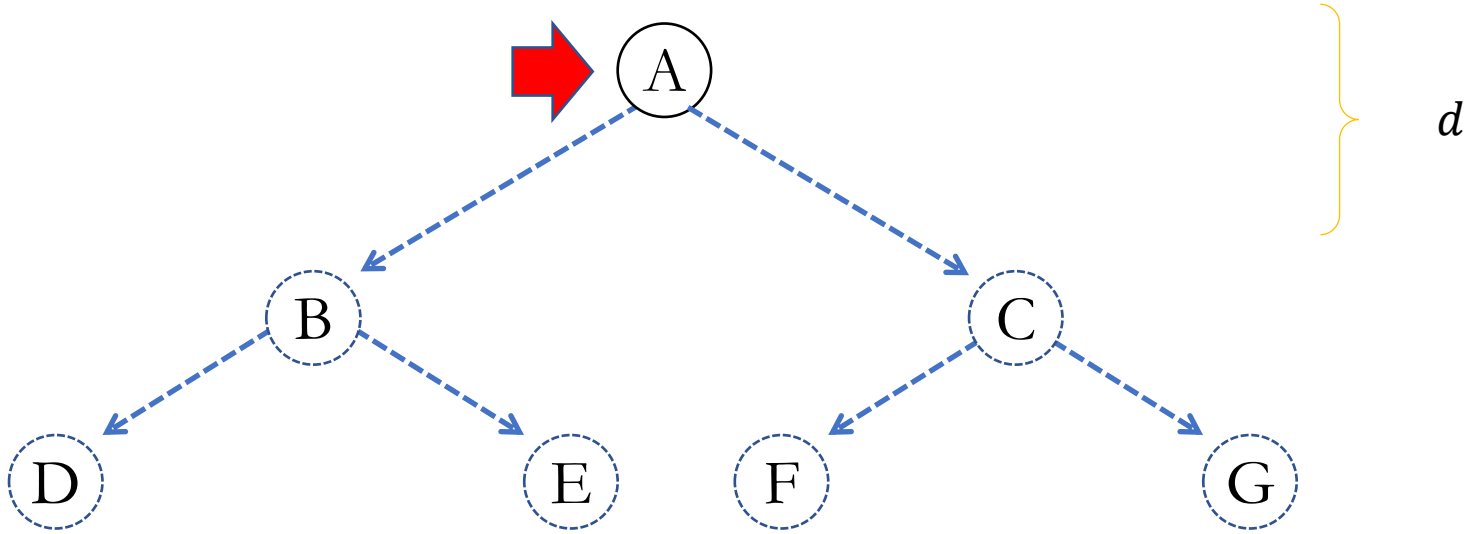
  - Breadth-first search (BFS)
  - Uniform-cost search (UCS)
  - Depth-first search (DFS)
  - Iterative deepening search (IDS)

Breadth-First Search

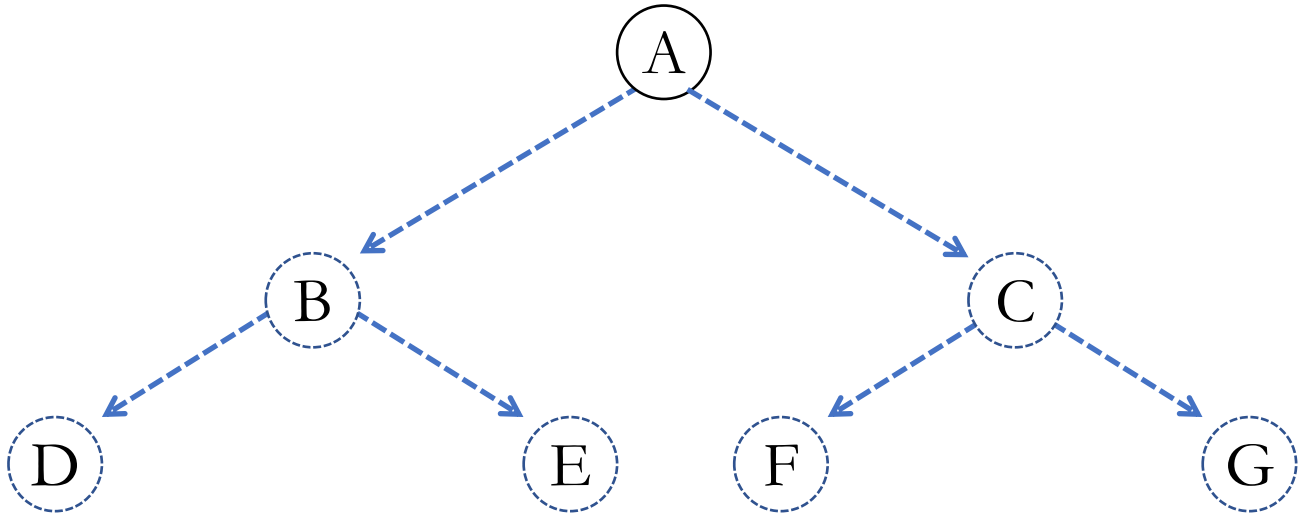
- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on.
- Expand shallowest unexpanded node: all the nodes at depth d in the search tree are expanded before the nodes at depth $d + 1$.



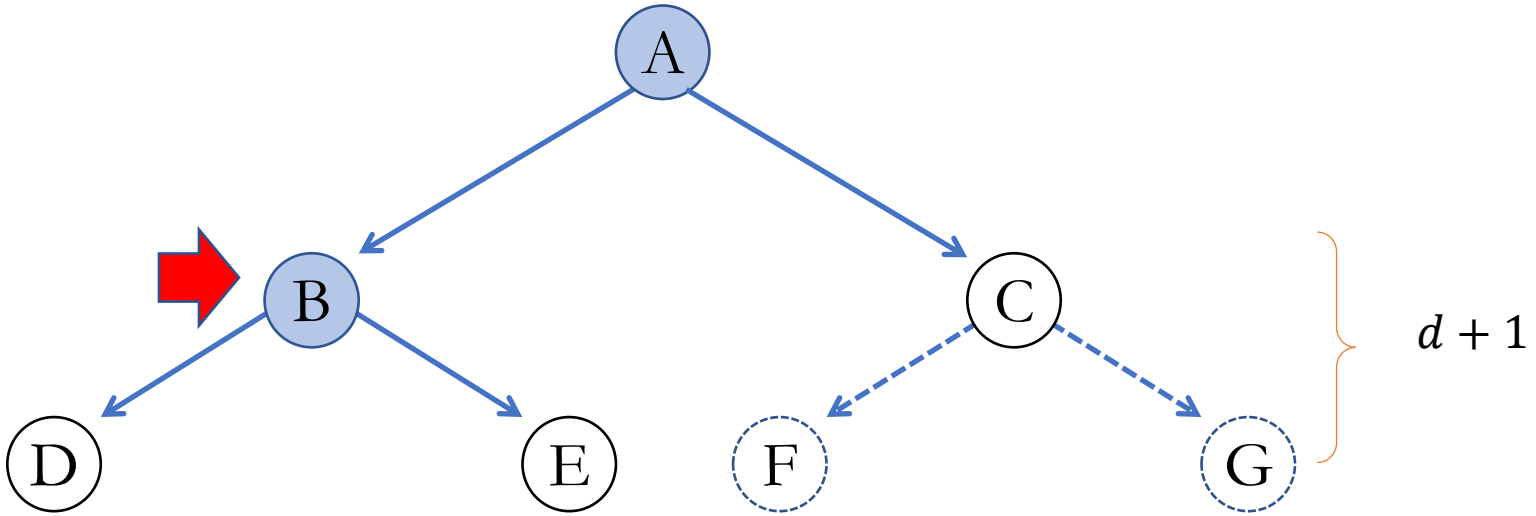
Breadth-first search cont.



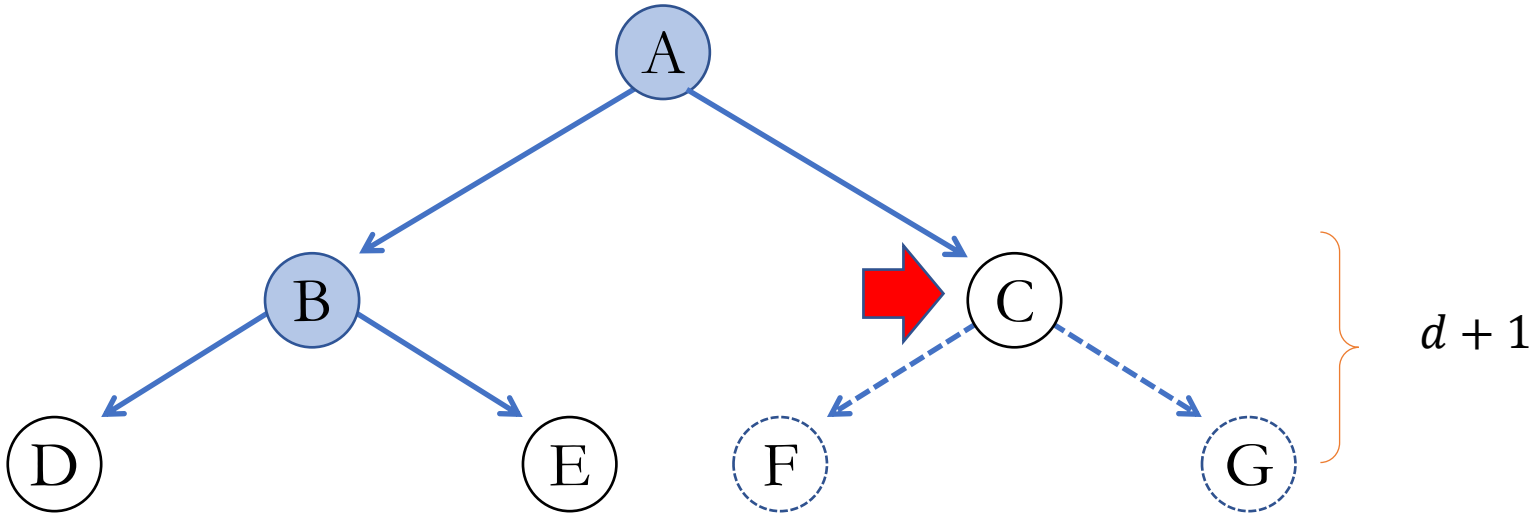
Breadth-first search cont.



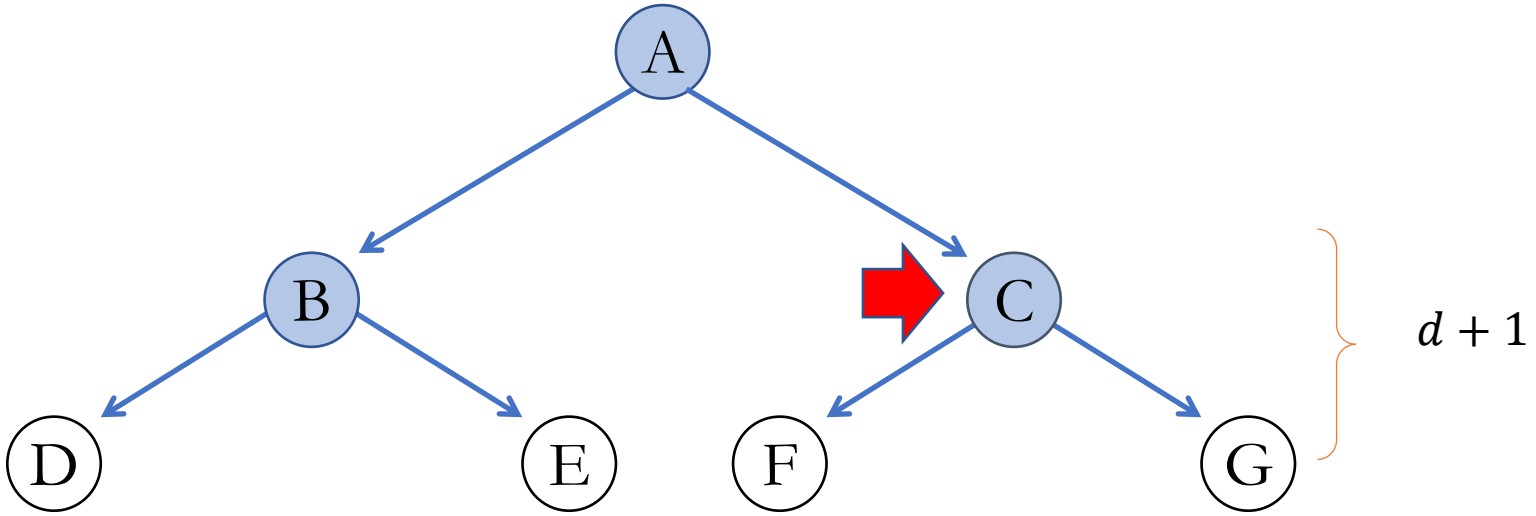
Breadth-first search cont.



Breadth-first search cont.



Breadth-first search cont.



Properties of BFS

- Let's check on the evaluation criteria
  - **Complete?**

    Yes (if branching factor *b* is finite)

  - **Optimal?**

    Yes – if cost = 1 per step (*or each action takes same cost*)

    Yes, it is even optimal if you consider the minimum number actions taken to reach the goal.

- How about space and time?

Properties of BFS - Time and Space

- **Uniform tree case:**
  - **Tree formation** - Root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on.
  - **Worst case solution** - the solution for this problem has a path length of $\boldsymbol{d}$ (it is the last node generated at that level).
    - The total number of nodes in level $d = b^d$
    - max # nodes expanded $= 1 + b + b^2 + b^3 + \bullet\bullet\bullet + b^d$
    - Note - the solution could be found at any point on the $\boldsymbol{d}^{\text{th}}$ level ($d \leq D$)
    - In the best case, therefore, the number would be smaller.

- **Timing complexity: $\boldsymbol{O(b^d)}$**
- **Space: $\boldsymbol{O(b^d)}$** - *fringe will have b^d number of nodes in the memory at any given point*
  - Space is the bigger problem (more than time)

Pop quiz

1. Define in your own words:
   (a) intelligence, (b) artificial intelligence, and (c) agent.

2. If "S" is the root node of the following state graph, write down the BFS traversal.